ARTIFICIAL INTELLIGENCE APPLICATIONS IN TELEOPERATED ROBOTIC ASSEMBLY OF THE EASE SPACE STRUCTURE

by

HERBERT E. M. VIGGH

Bachelor of Science in Aeronautics and Astronautics
Massachusetts Institute of Technology
(1985)

Submitted in Partial Fulfillment of
the Requirements for the Degrees of

MASTER OF SCIENCE IN

AERONAUTICS AND ASTRONAUTICS

and

MASTER OF SCIENCE IN

ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February, 1988

Signature of Author
Department of Aeronautics and Astronautics
December 31, 1987

Certified by ____
Assistant Professor David L. Akin
Thesis Supervisor, Department of Aeronautics and Astronautics

Certified by __
Professor Marvin Minsky
Thesis Reader, ~~Department of Electrical Engineering and~~ Computer Science

Accepted by
Professor ~~Arthur~~ C. Smith
e Students

Accepted by
Professor Harold Y. Wachman
Chairman, Departmental ~~Graduate~~ Committee

ARTIFICIAL INTELLIGENCE APPLICATIONS IN TELEOPERATED ROBOTIC ASSEMBLY OF THE EASE SPACE STRUCTURE

by

Herbert E.M. Viggh

Submitted to the Department of Aeronautics and Astronautics
and to the Department of Electrical Engineering and Computer Science
on January 8, 1988 in partial fulfillment of the requirements for the degrees of
Master of Science in Aeronautics and Astronautics and
Master of Science in Electrical Engineering and Computer Science

Abstract

A teleoperated robot was used to assemble the EASE space structure under neutral buoyancy conditions, simulating a telerobot performing structural assembly in the zero gravity of space. The teleoperator was manually controlled by a human operator at a remote control station. The neutral buoyancy testing of the teleoperator was done at the Neutral Buoyancy Simulator of the NASA Marshall Space Flight Center in Huntsville, Alabama.

Video tape data was collected of the telerobot performing structural assembly. Times for all of the subtasks completed by the teleoperator were taken from the video tapes. These times were used to obtain a breakdown of the total assembly time into different types of tasks. Data was also collected on operator fatigue and performance.

These results were used to propose several possible Artificial Intelligence (AI) applications for improving the teleoperator system. One of these applications, a real time assembly sequence planning program, was selected and developed.

This planning program was written in Prolog and implemented on an IBM AT computer. The program modeled the assembly process and could plan out the assembly of the EASE structure from any partial assembly point. The program generated a graphics display which presented information to the operator. This planning program demonstrated the application of certain artificial intelligence techniques in a teleoperator system.

Acknowledgements

This thesis is dedicated to my two loving parents, Mats and Anita Viggh. They brought me to this country, raised me, and put me through MIT. Without them, none of this would have been possible.

I would first like to thank my advisor Prof. Akin for giving me the type of working environment in which I work best. He entrusted me with the responsibility of the BAT project, gave me the free reign to pursue my own ideas and solutions, and was always there with the right answers and advice when I got stuck. Second, I would like to thank my thesis reader Prof. Minsky for the many interesting discussions and his strong support for the field of space telerobotics. One big thank you goes to all of the members of the SSL with whom I have worked over the past six years, as an undergrad urop'er and as a grad student. More than any other lab, ours is truly a team effort.

To Julia, thanks for all the friendship, caring, understanding, and cookies.

My final acknowledgement goes to the brotherhood of the Iota Mu Chapter of the Fraternity of Phi Gamma Delta. This thesis represents the culmination of over six years at MIT, five of which were spent living at the house. The friendships and support I found there were often the only things which kept me at MIT. Living at FIJI was a great learning experience and the most fun I've ever had. I am convinced that the "people skills" which I learned there will eventually be more important to my success than any technical education which I may have picked up along the way. Damn glad to be a FIJI.

Perge!

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Space Applications for Teleoperated Robots

To date, astronauts have demonstrated their ability to perform many tasks in space. Astronauts have explored the moon, completed emergency repairs on a space station, serviced satellites, and assembled structures on orbit.

The potential exists for using robots in the place of astronauts to complete these same tasks. Several possible benefits could be realized from this. The tasks could be completed without risk to human life and without the need to launch humans and their life support systems to the worksite. Robots could be designed to have greater strength and endurance than astronauts, allowing them to complete tasks which astronauts are incapable of. If designed properly, the use of robots could be more economical than humans due to lower launch costs, greater endurance, and less ground support than is needed for manned operations.

1.2 Spectrum of Control Strategies for Space Teleoperated Robots

The spectrum of possible strategies for controlling such a robot ranges from full manual control of the teleoperated robot's every action, to a robot which is capable of fully autonomous operation. In between these two extremes is a wide range of possible control strategies which involve partial human control of a teleoperator which has some degree of semi-autonomous behavior. This type of mixed control is usually referred to as "supervisory" control since it involves the human operator acting as a supervisor who gives commands to, and monitors the behavior of, a semi-intelligent machine.

Since fully autonomous robot behavior in space operations is still beyond current technology, some form of either manual or supervisory control must be used in the near term. As autonomous capabilities mature, it will still be desirable to use a mix of all three control strategies to optimize system performance, at least until the AI control capabilities become greater than those of a human.

1.3 AI Applications for Space Teleoperated Robots

Robots require intelligent control to be useful. In a completely manually controlled teleoperator, this intelligent control is provided by a human. In a semi-autonomous or completely autonomous system, some or all of the intelligent control must be provided by artificial intelligence.

In a semi-autonomous system, AI can provide control either directly or indirectly. Direct control would be manifested in supervisory control capabilities which directly make decisions and command the hardware. Indirect AI control would involve using AI to help the human operator to better perform manual control functions. This could involve having the AI reason about a problem and provide the operator with information or advice concerning how the task should be completed.

Therefore, there are possible AI applications in the manual, supervisory, and autonomous control strategies. In the case of manual control, AI can be used to help the operator plan things out, generate useful displays, or to warn the operator when he is about to make a mistake. In the case of supervisory control, AI can be used to make the semi-autonomous behavior of the teleoperator more capable, causing less manual control to be needed. In the case of autonomous control, the AI would be providing all of the necessary control.

1.4 A General Space Telerobot System

What type of control strategy is best? The author is of the opinion that a truly effective and flexible teleoperated space robot will employ all three types of control: manual, supervisory, and autonomous. While specific tasks will require a specific control strategy, a system which must perform several different types of tasks will need all three.

Such a robot will be referred to as a general space telerobot. As an example, consider a telerobot which is based at a space station and whose operator is inside the station. The telerobot has three main modes of operation.

The first mode is autonomous control. Once a day during a time when the telerobot isn't scheduled to be used for a while, it will automatically leave its garage and maneuver about the station making a full inspection. If problems are found, the telerobot fixes them on its own or reports it to the crew if they are beyond its autonomous repair capabilities.

The second mode is supervisory control. Assume a problem was discovered during the telerobot's inspection. It turns out that the problem could be fixed by the telerobot, except that it is a very unusual failure which the telerobot is unfamiliar with and therefore cannot repair. The telerobot, therefore, signals the crew of the space station, asking for help. A human operator analyzes the situation, figures out how the telerobot could repair the problem, and then gives the telerobot step by step instructions under supervisory control.

The third mode is manual control. Again, assume that the telerobot found a problem during its inspection. Only this time the problem is a very difficult one. Something has broken in an unanticipated way, and the telerobot was not designed to handle such a problem and cannot complete some or all of the steps of the repair, even under supervisory control. Therefore the human operator has no choice (other than to go out in a space suit and fix it himself) than to take over direct manual control of the telerobot and attempt to make the repair.

The general model of a telerobot is therefore one which employs a combination of all three types of control. If time lags make manual control impossible over all regimes in which the telerobot functions, then only supervisory and automatic control would be used (for a discussion of the problems caused by time delays, please see Reference 1).

Notice how the type of control needed is essentially a factor of what the AI controller of the robot is capable of. The idea is to let the robot do whatever it is capable of until a situation arises which is beyond these capabilities. Then the human operator must help out by providing either supervisory commands or direct manual control inputs. This will therefore be heavily influenced by the current level of AI technology available during the design of the system.

However, once the robot can function autonomously as well as, or better than, a human, supervisory and manual control will no longer be needed (except perhaps an "OFF" switch of some kind). One of the greatest challenges will be to build the robot so that new advances in AI can be implemented into the existing hardware system during this evolution.

1.5 Description of Thesis

In an attempt to answer basic questions about how such a teleoperated space robot should be designed and built, the Lab of Orbital Productivity (LOOP) group of the M.I.T. Space Systems Lab (SSL) built a teleoperated robot system which simulated a space robot by operating underwater under neutral buoyancy conditions. This teleoperated robot was named the Beam Assembly Teleoperator (BAT).

This thesis describes work which involved using BAT to perform structural assembly with a direct manual control strategy. Several possible AI applications to the system were then considered based on the results of the manual control testing. One of these possible AI applications was selected to be implemented. The application chosen was a real time structural assembly sequence planning program; its design and performance are discussed.

Chapter 2

Making Manual Control Operational in the BAT System

2.1 Structural Assembly as a Teleoperator Task

As mentioned in Chapter 1 and discussed in Appendix A, the LOOP group of the SSL set out to build a structural assembly robot which was named the Beam Assembly Teleoperator (BAT). This teleoperator system was seen as a logical extension of of the LOOP group's research involving human assembly of structures in space. It was also desired that BAT should provide general results for applying robotics to space operations. The choice of structural assembly as BAT's main task was chosen to satisfy both of these goals.

Structural assembly testing involves maneuvering, navigating, moving objects around, and completing dextrous manipulations. It can be argued that most other space applications of robotics will involve at least one or more of these tasks. Therefore, structural assembly was a good choice as a task for providing research results applicable to general space teleoperator systems. Selecting one main task also helped drive the design and development of BAT to a definite completion, and was useful for judging the success of the project. Structural assembly was also well suited for these purposes, since it is a well defined task and success can be measured by assembly times.

The EASE structure (See Appendix A for a description of the EASE flight experiment.) was selected as the baseline structure for BAT to assemble. The EASE flight experiment (EASE stands for Experimental Assembly of Structures in Extra-vehicular activity) involved having two astronauts in EVA (Extra-Vehicular Activity) repeatedly assemble and disassemble the EASE structure in the orbiter's payload bay.

There were two reasons for selecting the EASE structure. The first reason was that it is the simplest polyhedral structure (a tetrahedron) and was therefore a good place to start. Second, the EASE structure was also used as the baseline for the LOOP group's research into human structural assembly productivity and its use would therefore facilitate a direct comparison between human and teleoperator performance in the same task.

2.2 The EASE Structure

The EASE structure is shown in Figure 2.1. It consists of six beams and four nodes. The nodes are called joint clusters, and will be referred to simply as clusters. Before assembly, all of the beams and three of the clusters are stored in parts racks which consist a beam rack and a cluster rack. The fourth cluster is hard mounted to a solid surface and is referred to as the base cluster. The remaining three clusters are called top clusters and are all identical. The three beams which are attached to the base cluster are referred to as upright beams, or simply uprights. The remaining three beams are called cross beams. All six beams are identical and are 12 ft. long.

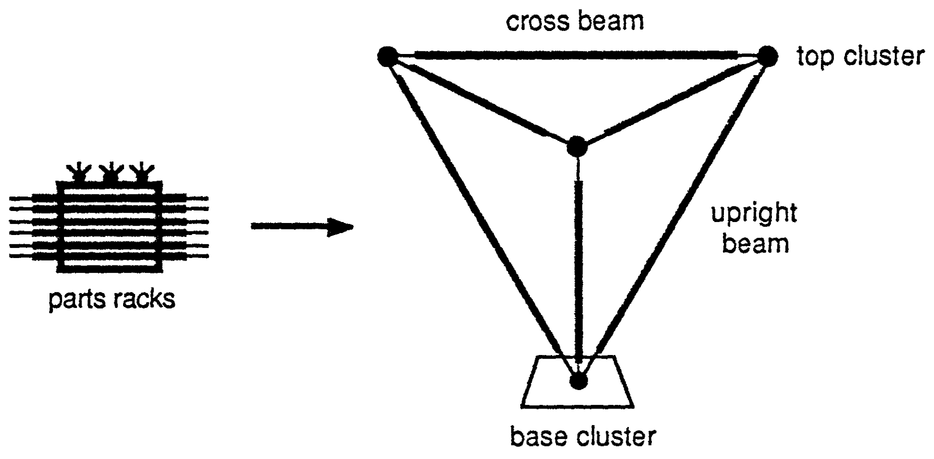


Figure 2.1 The EASE Structure

A beam and cluster are connected together by a joint as shown in Figure 2.2. A joint has two parts, the beam end and the cluster end, which is referred to as a mushroom end due to its shape. Both ends are made from aluminum stock 1-3/16 inches in diameter.

The joint is made by first transversely inserting the mushroom end into the receptacle of the beam end. A cylindrical sleeve on the beam end is then slid over the mushroom end until a spring loaded button pops up and keeps the sleeve from sliding back onto the beam end. A rivet attached to the sleeve and running in a groove in the beam end, keeps the sleeve from sliding too far onto the mushroom end. The button and rivet secure the sleeve in place, which keeps the mushroom end in the beam end receptacle.

This design intentionally requires very precise beam alignment to complete the joint. Due to drag effects, the beams used in neutral buoyancy testing require higher alignment forces than equal length beams used to build structures in space. This design then provides a conservative simulation of what an actual space joint would require for alignment accuracy by forcing the neutral buoyancy beams to have to be aligned very accurately.

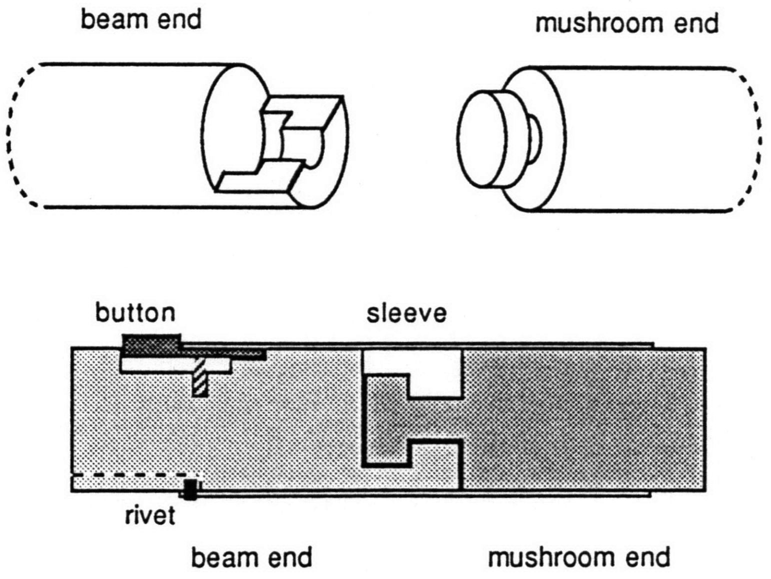


Figure 2.2 The EASE Joint

2.3 Initial Configuration of BAT System

It was intended that BAT be a flexible test bed for many ideas concerning the design and control of space teleoperators, specifically research on the degree of automation needed, with the advantage of testing those ideas out in a complete system with a realistic task. As a starting point, BAT was first to be made operational under direct manual control. This would provide a baseline with which to compare more automated control schemes, and would provide results which could be helpful in the design of higher levels of automation.

For a description of the history and early development of BAT, please see Appendix A. Appendix A describes prior work done in the M.I.T. Space Systems Lab, and is included in this thesis as it is essential for understanding the work described in this chapter.

Figure 2.3 shows the configuration of BAT at the initiation of the work described in this thesis. BAT consisted of a frame with eight propellor thrusters mounted to it for maneuvering. The frame was 2x2.5x3 ft. in dimensions. Mounted to the frame were two arms. The first was a simple grappling arm for holding beams. The second was a dextrous arm for manipulating clusters and making joints. A camera was mounted to a tilt and pan unit for the purpose of providing the remote operator with video feedback from the worksite. A control station for the human operator (Integrated Control Station (ICS)) was also built, and is described in Appendix A.

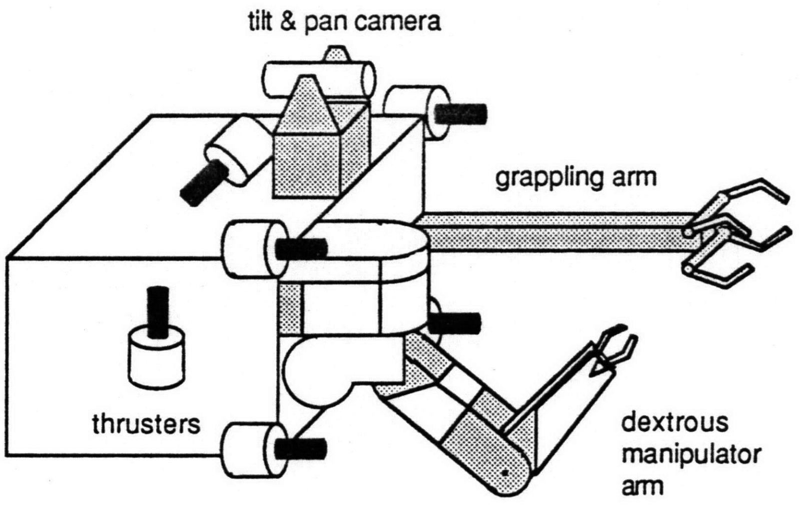


Figure 2.3 BAT's Initial Configuration

2.4 Inadequacies of Initial BAT System

The first step in the work of this thesis was to make BAT operational, so that it could be used to assemble the EASE structure under direct manual control. When the author took over the project, BAT and ICS were built and in the configuration described in Appendix A, but had not yet performed any structural assembly. BAT had performed certain assembly subtasks like flying and grappling beams. However, BAT had yet to successfully attach a beam to a cluster.

Some of the difficulty in getting BAT to assemble the EASE structure lay in mechanical and electrical problems in the system. Several months were spent eliminating these bugs, but once these problems were worked out, it became obvious that there were deeper, more fundamental problems in the system. These more serious problems prevented BAT from being able to assemble the EASE structure, and required changes both in the hardware and in the overall strategy of how BAT was to function and be controlled.

There were many problems, but there were only two basic types. The first type resulted from an underestimate of what specific tasks demanded of the hardware. This resulted in the hardware not being capable of completing that task. The second type of problem involved underestimating what was required of the operator to control the hardware. This resulted in operator overload to the point where the operator could not effectively control BAT to complete the task. For certain tasks, both types of problems were present.

There are also two reasons why the discussion of these problems and their solutions is important to understanding possible applications of AI to a teleoperator system.

The first reason is that it is important to gain a good understanding of the overall system. What does the task of performing structural assembly with robotic hardware involve, regardless of whether human or AI control is used? Paper design studies often miss crucial problems which will not be found until a hardware system is built. Problems for a human operator will in general also be a problem for a computer controller. Since manual control capability will be needed in a general space telerobot system (see Chapter 1), it is important to understand the manual control mode so that it can be effectively combined with the other modes involving AI control.

The second reason for discussing the development of an operational manual control teleoperator system is more subtle than the first: the solutions found to the problems encountered were all manual control solutions. There were other possible solutions which were not pursued. Therefore, it would be a mistake to simply take the manually controlled BAT system, analyze the problems with it, and attempt to solve them through applications of AI technology. One should also consider trying to "re-solve" the problems solved here through manual control techniques by trying to solve them in a different way using AI automation techniques.

This chapter describes the problems encountered with the initial BAT and ICS system, how they were solved, and the important points learned. The problems discussed are divided into three areas: the video system, attaching a free beam to a fixed cluster, and carrying beams.

2.5 The Video System

Closed loop control of a system requires feedback of the proper kind and quality. While humans are endowed with five senses for feedback from the environment, the principal form of feedback from the BAT worksite consists of video views. Without them, the operator cannot run BAT, even if it is functioning. Unfortunately, the initial video system on BAT did not provide the operator with the proper feedback to allow him to successfully control BAT.

The initial configuration of the video system was a single black and white camera mounted on a tilt and pan unit. The tilt and pan unit had two degrees of freedom which allowed the camera to be tilted up and down and panned left and right. The picture from the camera was displayed on a nine-inch monitor in the control panel (See Appendix A for a description of the BAT control station.).

The tilt and pan motions of the camera were controlled by two potentiometers (pots) mounted in the top section of the control panel. The pots were located so that the operator typically used his right hand to adjust them, but either hand could be used. Since it would be difficult to adjust the tilt and pan while busy flying or using the master arm, the camera had a wide angle lens on it to give the operator a wide field of view to minimize the need to repoint the camera.

The strategy for using the video system was as follows. For flying, the operator was to position the camera prior to flying and then not have to reposition it until he had finished flying and had docked again. For making joints with the dextrous arm, the operator was to point the camera down onto the general work area of the manipulator arm, and then adjust the view with the left hand (the operator's right arm would be busy using the master arm) only when the operator needed to specifically look around for something outside of the field of view. There were several problems with this strategy.

2.5.1 Problems with Original Video System Encountered while Flying

Since BAT's flying capabilities matured early, the first problems with the video system were encountered while attempting to fly. Several sets of experiments were done involving the operator flying BAT and attempting to dock to a fixed beam. These experiments are more fully described in Reference 2. The results of these experimental trials are summarized below.

For flying, the camera was pointed straight ahead along the x-axis. Although the camera had a wide angle lens on it, when it was pointed straight ahead the grappling claw was no longer in the field of view. This made it impossible to tell when the grappling claw was in position to grapple the beam which one was trying to dock to. As a matter of fact, no part of BAT was in the field of view, so there was no real visual reference as to whether or not the camera actually was pointed straight ahead, and therefore, what BAT's orientation was with respect to the surroundings.

Another problem lay in the fact that the camera was offset from BAT's axes of roll and pitch motion, which made it extremely difficult to accurately control the roll and pitch of BAT using the feedback from the camera. This, combined with the inability to see the grappling claw or any other part of BAT, made it virtually impossible to dock to anything with the camera pointing straight ahead.

For this reason, the operators tried to fly with the camera pitched down about ten degrees so that the grappling claw was in view. This made it possible to see when one should close the claw, except for a problem with depth perception. However, tilting the camera did nothing to help the problems associated with the offset of the camera from the roll and pitch axes. Unfortunately, this added the problem that the camera now also had an angular offset in pitch from BAT's axes of motion, so that commanding a forward thrust with the hand controller did not correspond to forward motion with respect to the camera's view. This made flying even more difficult.

In an attempt to fix these problems, a second wide angle camera was mounted on the front of BAT between the dextrous arm and the grappling arm. The camera was rigidly mounted, and pointed straight ahead along the x-axis . This solved the problem of the camera view being displaced from the thrust axis and put the grappling arm into the field of view. This made flying much easier and the only remaining problem was that of poor depth perception.

A switch was added in ICS to allow switching between the tilt and pan camera and this second camera. The second camera came to be known as the "belly" camera since it was mounted on the belly of BAT.

2.5.2 Problems with Original Video System Encountered while Manipulating

The next set of problems arose when the video system was used while attempting to make a simplified joint. In these manipulation tests BAT was secured to the bottom of the pool so that no flying was necessary (or possible). The grappling arm held a beam, and the dextrous arm held a mushroom end which had been removed from a cluster and was therefore easier to manipulate. The joint was to be made by inserting the mushroom end into the beam end and then sliding the sleeve. The tilt and pan camera was tilted down to look at the work area of the arm (view centered on the beam end).

The first problem encountered was that the wide angle lens made it extremely difficult to see the small details of the joint's mushroom end and beam end. The poor resolution caused by the wide angle lens simply was not good enough to allow consistent assembly of the joint. This was also true of the belly camera which also had the additional problem of viewing the joint from a poor angle for joint assembly.

The second problem was that the single camera provided no depth perception. It was difficult to tell which structural end was further away from the camera. This lack of depth cues further aggravated the difficulty of making the joint.

The final problem was simply that the pots were a cumbersome and inefficient way of controlling the tilt and pan unit.

In an attempt to fix these problems, a third camera was added. This camera had a narrow field of view and was mounted on the shoulder of the dextrous arm. The camera was mounted in such a way that it was slaved to the arm's shoulder yaw, so to pan it back and forth, the operator needed only to move the arm back and forth at the shoulder. The "shoulder" camera was pointed down at a fixed angle which would put the dextrous arm's claw at the center of the field of view when the arm was in a typical joint making position.

The addition of the shoulder camera solved the low resolution problem by using a narrow field of view camera. This also provided a less cumbersome way of looking around, except that the operator also had to move the arm to do so, and the pitch angle of the camera was fixed. However, the operator now had to choose and switch between three different camera views.

Unfortunately, the addition of the shoulder camera did not solve the lack of depth perception. Once the resolution problem was solved, it became apparent that the lack of depth perception was quite significant and still made joint assemblies difficult.

2.5.3 The New Video System

It was concluded that the main problem with the initial video system was that one video camera was being used for two very different viewing tasks and was not particularly well suited for either. To solve this problem it was decided to divide the video system into two subsystems, one for flying and one for working with the manipulator arm.

The three original cameras (tilt and pan, belly, and shoulder) were all removed from BAT and a new video system was installed as described below.

To work with the manipulator arm required a narrow field view, the ability to change the view quickly and easily, and stereo vision for depth perception. To accomplish this, a pair of narrow field vision cameras were mounted in the tilt and pan unit in place of the single camera. To better control the tilt and pan, a head controller was built and installed in ICS and is shown in Figure 2.4. This head controller consisted of a helmet with a mechanical linkage which measured tilt and pan motions of the operator's head. The control system slaved the tilt and pan unit attitude to the corresponding position of the operator's head. The gimbal arrangement also provided head roll information which was not used. The head controller helmet also served as a mounting point for two small video monitors, which presented the stereo images of the cameras to the operator's eyes.

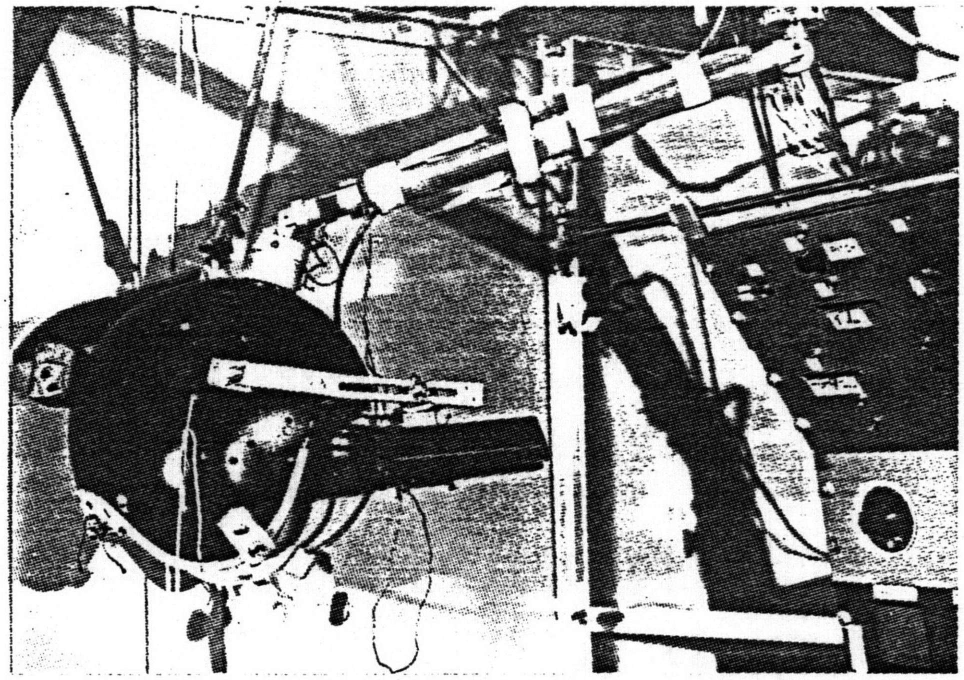


Figure 2.4 ICS Head Controller

For flying, one requires four things: a wide field view for navigation, stereo vision for depth perception, the ability to see the grappling claw, and a fixed position near the center of, and aligned with, the axes of motion. To provide all of this, a fixed pair of wide field stereo cameras was mounted on the front of BAT between the two arms where the old belly camera had been mounted. A switch was installed in ICS to allow the operator to switch between the tilt and pan cameras and the fixed flying cameras ("belly" cameras). The images from the selected camera pair were displayed on the stereo viewers mounted on the head controller helmet. The new video configuration is shown in Figure 2.5. Note that BAT's dextrous arm is not shown, so that the belly cameras are visible from this perspective.

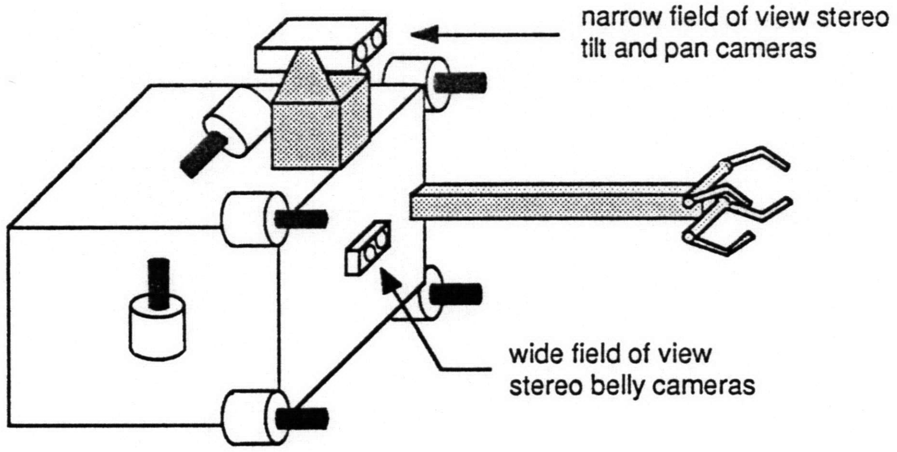


Figure 2.5 New Video System Configuration

2.5.4 Lessons for Manual Control from Video System Problems

The main lesson for designing the video system for a space teleoperator is that it must provide the operator with visual feedback for two (possibly more) separate and different tasks. The tasks of flying and manipulating place very different demands on the video system. This makes it difficult to design a system which uses one camera for everything.

In the case of BAT, all of the tasks can be roughly split into tasks involving flying and tasks involving manipulation. These two task groups each require different fields of view, different camera angles, and different resolutions (flying requires a wide field of view, which means low resolution, whereas manipulation requires high resolution which means a narrow field of view). In such cases it makes sense to use two different sets of cameras, instead of using one set which would be hard pressed to meet the requirements of both types of tasks.

For flying, humans need to have a simple one to one correspondence between the thruster commands which they input and the way the camera view changes. A rotation command should make the camera view rotate about axes which are centered in the field of view. Similarly, translation commands should cause the proper effect to the operator's view as well. In the case of BAT, if the tilt and pan cameras are used for flying and are pointed straight ahead, then a forward translation would make a point in the center of the view move downward instead of staying in the center, due to the fact that the tilt and pan cameras are displaced above the center of motion of BAT. Any angular displacement will make these problems worse.

Manipulations with the dextrous arm are similar enough to things which humans normally do (unlike flying about in space with six DOF) as to make the use of telepresence techniques worthwhile. The use of a head slaved stereo vision system allows the human operator to look around the remote worksite as he would look around a place where he actually was at. This works better than using pots or a joystick to control the tilt and pan of the camera, since the operator uses his own innate reflexes to point the cameras.

Also, the BAT experience supports the findings of other researchers that stereo vision systems provide important depth cues which are critical for good human operator performance.

2.5.5 Possible AI Control Solutions to Video System Problems

Most of the problems with the video system involve the quality of the video feedback to the operator, and therefore do not lend themselves to applications of AI techniques. However, one idea for using machine control to help with the video system is particularly interesting. This possible partial solution to the problem of flying with a camera which is angularly or linearly displaced from the axes of motion is to implement a control system which takes the operator's flying commands, assumes that they are with respect to the view which the tilt and pan camera is currently providing, and transforms the thruster commands to execute those commands. While this would not solve all of the problems discussed, it could be useful in situations where the tilt and pan is being used while manipulating and some flying needs to be done to help with the manipulation task.

2.5.6 Lessons for General Space Telerobots from Video System Problems

The main lesson for trying to automate a teleoperator through AI is that feedback from the world and the task at hand is critical for success. This point cannot be over stressed. The current level of machine vision technology must be strongly pushed in order to meet the requirements of space teleoperation. Other currently available sensing techniques must also be taken advantage of to achieve semi-autonomous or fully autonomous capabilities. For navigation while free flying, techniques involving inertial and satellite positioning systems should be exploited. Strategies for combining data from many different sensor systems also must be developed.

2.6 Assembly of a Free Beam to a Fixed Cluster

Figure 2.6 depicts BAT attaching a free cluster to a fixed beam. The beam is considered "fixed" in that it is already attached to the structure, designated by a cross hatched cutoff. This situation is very similar to the simple joint connection tests described in Section 2.5.2, except that a whole cluster is used rather than just a mushroom end. After solving the problems with the video system, BAT could complete this task.

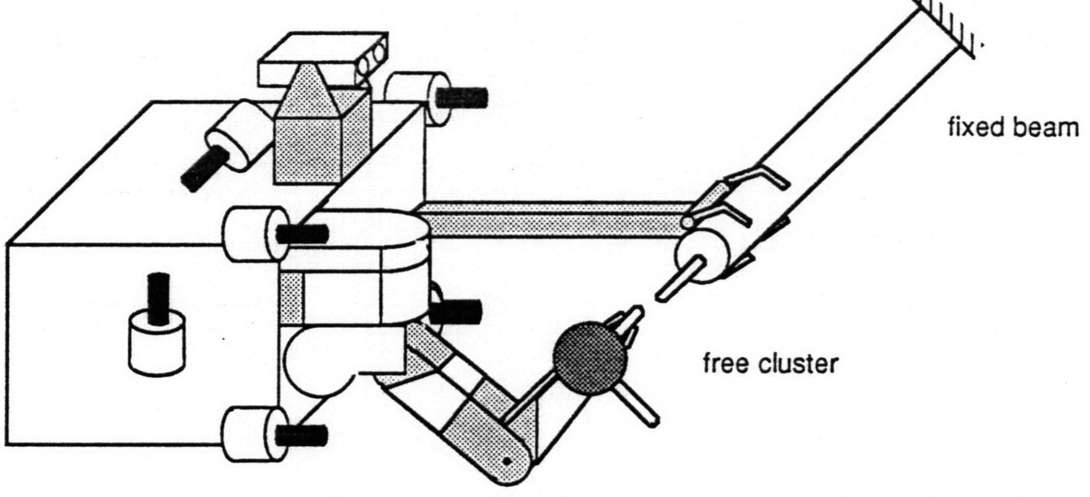


Figure 2.6 Attaching a Free Cluster to a Fixed Beam

Consider the requirements of successfully attaching a free cluster to a beam as shown in Figure 2.6. With a cluster held in the dextrous arm (the arm can be commanded to hold position without the need of constant control with the master arm), the operator first flies up to and docks with the beam, using the two hand controllers to fly and the belly cameras for visual feedback. Next, the operator switches to the tilt and pan cameras, and uses the master arm to control the dextrous arm to move the cluster and make the joint. No problems were encountered when this was attempted and accomplished.

In contrast, Figure 2.7 shows how BAT had been intended to attach a free beam to a fixed cluster. In Figure 2.7, the "fixed" cluster is attached to a beam which is connected to the rest of the structure, and the beam held by BAT is "free" in that it is not attached to the structure. To make the connection, the dextrous arm must be capable of moving the mass of BAT about, as well as that of the free beam held by BAT. Problems where encountered when attempting to complete this task.

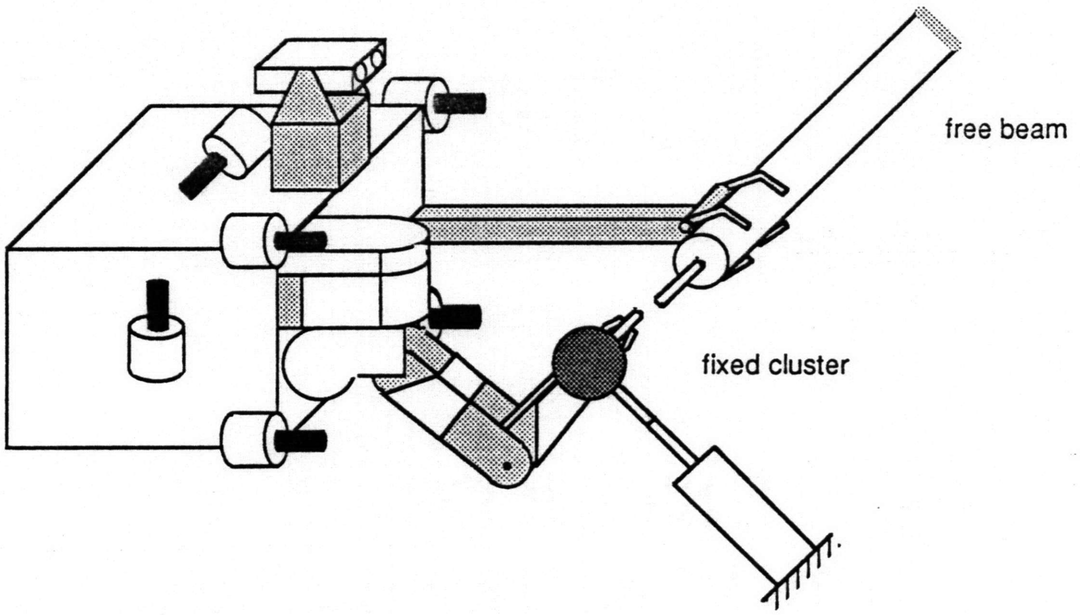


Figure 2.7 Attaching a Free Beam to a Fixed Cluster

2.6.1 Problems Involved with Attaching a Beam to a Cluster

There were two main problems encountered while trying to attach a free beam to a fixed cluster. The first was that the operator was unable to control BAT to grab the mushroom end of the fixed cluster with the dextrous arm; given this, the dextrous arm still lacked the needed strength to make the joint.

Consider what is involved in attaching a free beam to a fixed cluster . Assuming that BAT already has a beam and is holding it in the claw of the grappling arm, it must now fly up to the cluster and and grab one of the mushroom ends with the slave arm. This is where the first problem was encountered.

In order to fly BAT up to the cluster, the operator needs both his hands to use the two hand controllers. Then, upon getting close enough to the cluster to grab one of the mushroom ends, the operator must do the following:

1) Release the right hand controller, locate the master arm, and insert his arm into the master arm.

2) Connect (turn on) the master arm, power up the dextrous slave arm, and unstow the slave arm from the position in which it is kept in while flying.

3) Switch from flying cameras to tilt and pan cameras.

4) Reach out with the slave arm and grab onto the desired mushroom end.

However, by the time step 1) is finished, BAT has drifted so far away from the cluster that the operator needs to fly back to it, which he cannot do unless he undoes step 1) by releasing the master arm and using his right hand to fly again.

The problem was that ICS's control configuration did not allow the operator to fly and control the manipulator arm simultaneously. An alternate solution, having the control system provide station-keeping at the cluster, was impractical due to a lack of reliable position sensors.

In contrast, the case of attaching a free cluster to a fixed beam (Figure 2.6) has a clean division of the flying task and the manipulation task, both of which require the use of the operator's right arm. Due to this division, both tasks can be accomplished since the operator completes one before beginning the other. However, in the case of attaching a free beam to a fixed cluster, this division is not present.

The second problem became apparent after BAT was helped by support divers to grab the mushroom end so that the operator could at least attempt to attach the free beam to the fixed cluster, thus bypassing problem of grabbing the mushroom end.

The design of the dextrous arm was based on a frame size much smaller than the frame which is shown in Figure 2.7 (see Appendix A). The larger frame was required to hold all of the BAT subsystems which were underestimated in the original concept. Unfortunately, this change in frame size occurred after the dextrous arm was completed.

The net result of this design revision was that the dextrous arm now had to move much more mass than it had originally been designed and built for. All attempts to make the joint in this mode failed. The dextrous arm simply was not strong enough to maneuver the combined mass of BAT (with the large frame) and a beam.

2.6.2 The Solution: A New Grappling Arm

The key to the solution of the problems involved with attaching a free beam to a fixed cluster lay in the fact that BAT could easily attach a free cluster to a fixed beam. There are two important ways in which attaching a free cluster differs from attaching a free beam. The first difference is that, in the case of attaching a free cluster, the operator never has to fly and use the manipulator arm at the same time. The second difference is that, when attaching a free cluster, the grappling arm rigidly fixes BAT to the fixed beam, and the manipulator only has to move the mass of the cluster, whereas when attaching a free beam, the manipulator has to move both BAT and the beam, which proved to be impossible. Tests showed that the manipulator arm was strong enough to move a free beam with enough control to make a joint. Therefore, if the grappling arm could be made to grab a fixed cluster, then BAT would be capable of attaching a free beam to a fixed cluster the same way BAT successfully attached a free cluster to a fixed beam.

Therefore, a new grappling arm shown in (Figure 2.8) was built which could grab and dock to either a beam end or a mushroom end. The new grappling arm was designed with a narrow claw which could grab either a mushroom end or a beam end. This did not affect the way a free cluster was attached to a fixed beam, since BAT could still dock to the beam end with the new grappling arm. The difference was that now the operator could also fly and dock to a cluster, and then use the manipulator arm to attach a beam without having to fly and manipulate simultaneously. Figure 2.9 shows how the new grappling arm was used to dock to a cluster.

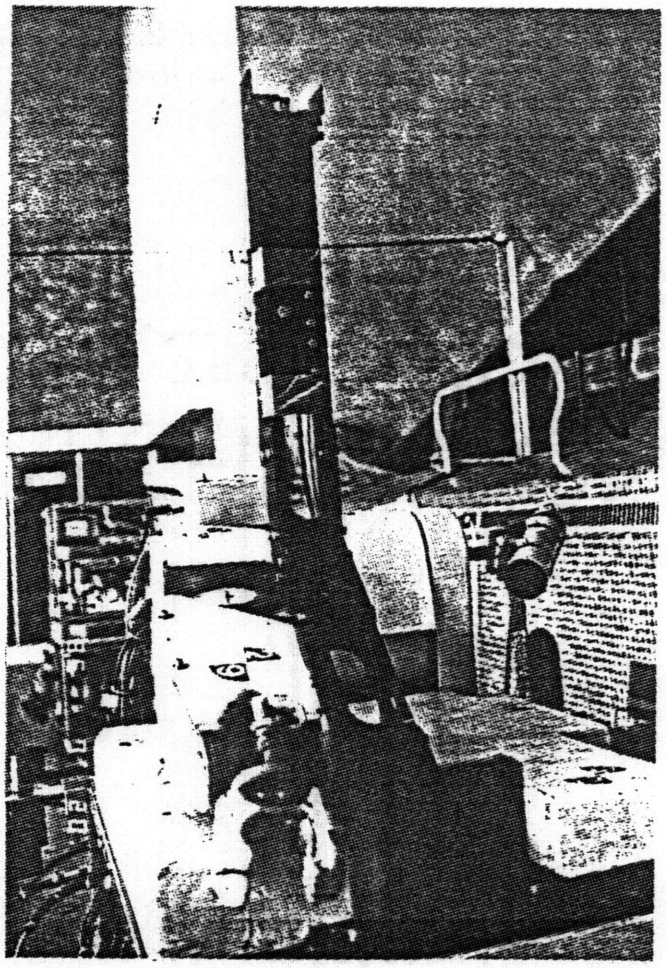


Figure 2.8 BAT's New Grappling Arm

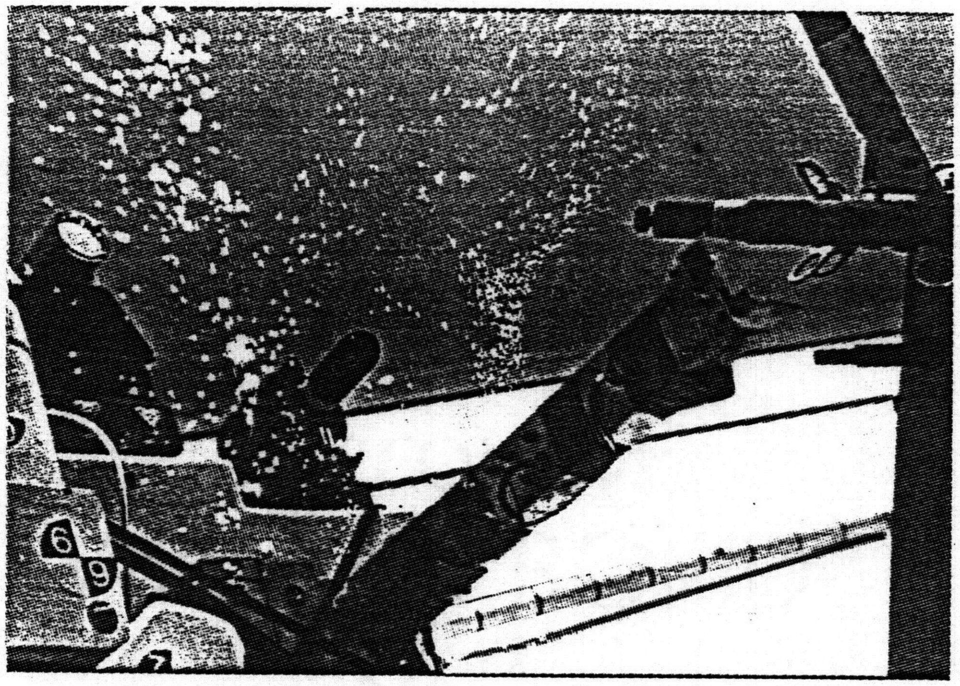


Figure 2.9a BAT Approaching a Cluster

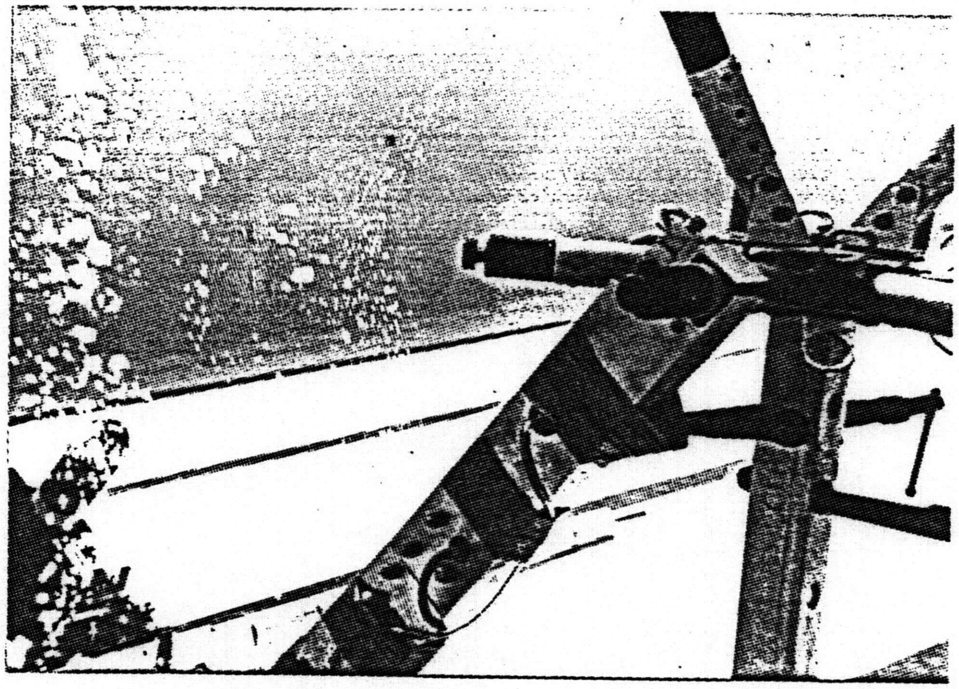


Figure 2.9b BAT Docked to a Cluster

2.6.3 Lessons Learned for Manual Control from Beam Attachment Problems

The operator must never be expected to perform two task simultaneously which require using the same body part to operate two different controls. If possible, the two tasks should be split up so that they are not concurrent.

Another possible solution is to develop control modes where the operator uses different body parts to control the two tasks simultaneously. This was actually tried with BAT to solve the problem of flying and manipulating simultaneously, and is referred to as cross-modal control. Two different modes of control were tried for flying and manipulating at the same time. The first involved using the head controller in place of the right hand controller to control BAT's pitch, roll, and yaw. This would free up the operator's right arm to use the master arm, and allow the operator to simultaneously fly and manipulate. The operator would not ,however, be able to control the tilt and pan at the same time.

A second mode involved using the master arm to control both the dextrous arm and the vehicle rotations. A switch was used to toggle between using the arm as a master arm and as a maneuvering hand controller. As a hand controller, the pitch of the elbow, the roll of the wrist, and the yaw of the wrist were used to command the pitch, roll, and yaw of BAT. While this did not allow simultaneous control of both flying and manipulating, it did make the switching between them quick and easy. This mode also had the advantage of allowing the operator to control the tilt and pan with head position.

While these two control modes showed promise and were useful in certain situations (while docked to the structure), they tended to be unstable in free flight since the current vehicle flight control system uses rate commands and neither the head controller nor the master arm had physical return to center as the hand controllers did. Perhaps with better designed controls such as force reflecting master arms, return to centers could be implemented in software and make these cross-modal control approaches viable.

2.6.4 Possible AI Solutions for Beam Attachment Problems

There is another approach to solving the problems involved with attaching a free beam to a fixed cluster which was not pursued. This other approach involves advanced computer control modes, which would either help the operator to fly BAT while the operator is busy controlling the dextrous arm, or would control the arm while the operator does the flying.

To solve the problem of not being able to perform a free flying grab of the mushroom end with the dextrous arm, one could implement a station-keeping mode. After flying up to the cluster, the operator could command a supervisory routine to hold BAT's attitude and position while the operator finds and uses the master arm to reach out and grab the mushroom end. This would require position and attitude sensors, along with the algorithms needed to control BAT's thrusters to keep BAT from drifting away. Similarly, a supervisory routine could be developed which controls the arm to reach out and grab the mushroom end once the operator flies BAT close enough to it. This would require the ability to select the proper mushroom end and sense its position. Combining these two would allow the whole approach and grapple to be done completely under supervisory control.

To solve the problem of the weakness of BAT's manipulator arm in the initial assembly configuration, another advanced control mode could be developed which utilizes BAT's thrusters to help the operator move BAT and the beam, taking some or all of the load off of the dextrous arm. The computer could use the geometric position of the dextrous arm's joints (encoder positions) for feedback about BAT's position (and therefore beam position) with respect to the mushroom end which the dextrous arm is grabbing.

Similarly, the operator could do the flying to get the beam end close to the mushroom end, and a supervisory routine could then use the arm to help with the terminal guidance of inserting the mushroom end into the beam end. Again, combining the two supervisory routines would allow the joint making to be done completely under supervisory control.

Also note that if the operator is performing one task, and the supervisory control the other, complications can arise. Since the operator is busy with his own task, he may not be able to supervise the AI control at all. This would force the system into a control situation where two autonomous agents are each controlling different things which must work together for success. How the two would communicate, and which has precedence when both attempt to use the same resources, must be worked out.

2.6.5 Lessons for General Space Telerobots from Beam Attachment Problems

In general, space teleoperators may be called upon to complete more than one task at a time. For example, a satellite servicer may need to fly up to and match rotation with a disabled satellite and simultaneously reach out with multiple grappling arms to grab the satellite. A single human operator would be hard pressed to fly the servicer and control the grappling arms at the same time. The ideal situation, given no restrictions on time or resources, would be to automate both tasks.

A second possibility would be to only automate one of the tasks and have the human operator perform the other, as described in section 2.6.5. Perhaps the operator will have many supervisory routines which can be combined in different ways to solve new problems, and still allow the operator to simultaneously use some manual control to perform tasks for which there are no supervisory capabilities.

If neither of the two tasks can be automated, as could happen in many contingency situations, the only remaining resort is to have the human operator try to complete both tasks. Manual control strategies should be available through which the operator can do two things at once, or sequentially with minimal transition time and effort. This means that cross-modal controls schemes should be developed, along with flexible controls to make them work. Hand controllers which can be used both for flying and controlling arms should be investigated.

2.7 Carrying Beams

The use of the new grappling arm did, however, aggravate another problem: there was no really good way to carry beams with BAT.

Cluster are easy to carry with BAT. They are small in size and weight, are low in drag and can be carried with the dextrous arm. Beams, on the other hand, are large, massive, and have a high drag cross section along the transverse axes.

The original grappling arm had a large claw which could grab a beam anywhere along the length of its major diameter. A roller mechanism in the end effector could translate the beam along its longitudinal axis so that the grapple point could be moved to any point on the beam. This feature was used to center the beam on the BAT propulsion unit for symmetrical drag properties during free flight.

2.7.1 Problems Involved with Carrying Beams

When flying BAT while carrying a beam in the original grappling claw, the beam was held at its center to reduce drag induced yaw effects. In this position the beam was lined up in position to be attached to a cluster, but unfortunately this position also maximized beam drag in the most common direction of flight: straight ahead. This, combined with the fact that the beam also blocked the view of the belly cameras, made flying BAT difficult while carrying a beam. To make matters worse, when getting close to the cluster to which the beam was to be attached, the roller in the grappling claw would have to be used to change the grappling point to the end of the beam. Otherwise the right side of the beam would be in the way of the manipulator arm grabbing the mushroom end. This required the final and most critical flying and docking maneuvers to be attempted with almost all of the beam sticking straight out to one side, causing adverse yaw effects.

The new grappling arm complicated matters since it could only grab a beam by one of its beam ends, so that all of the flying would then have to be done as above with the entire beam sticking out to one side. A more substantial problem was that the beam would have to be handed off to the dextrous arm before a grapple could be completed to a cluster. It was clear that the grappling arm would no longer be useful for carrying a beam while translating.

2.7.2 The Beam Carrier

An attempt was made to carry beams with the manipulator arm. To reduce the drag problem, the right arm held the beam over BAT's shoulder, so that the beam pointed straight backwards. This is shown in Figure 2.10 in which BAT is carrying a "mini-beam" which was a shortened version of the EASE beam used in developmental work (in Figure 2.10 the mini-beam also has a cluster attached to one end).

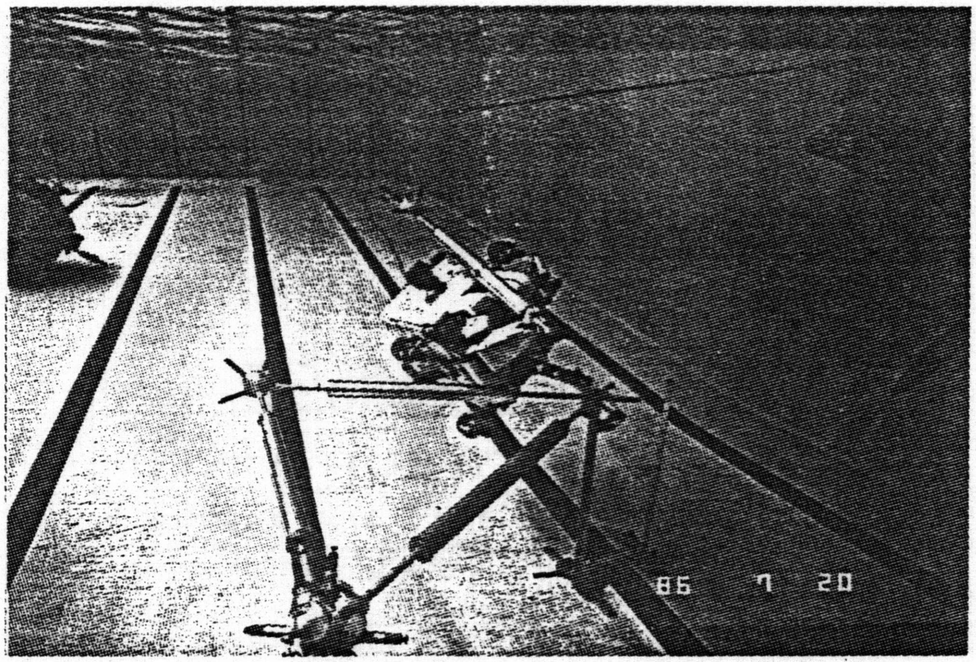


Figure 2.10 Carrying a Beam with the Manipulator Arm

After docking to a cluster with the new grappling arm, the dextrous arm would then be used to rotate the beam around and make the joint. This worked well the first few times it was tried. However, soon there after it became clear that the arm had not been designed for repeated large angle beam rotations: motors burned out and chains broke. Although the arm was capable of providing the sufficient static torque to rotate the beam, neither the motors nor the drive mechanisms were rated for continual application of maximum torque.

Instead of carrying beams with the manipulator arm, it was decided to develop a third arm to carry a beam over BAT's shoulder, which could rotate the beam out into a suitable position for the manipulator arm to grab the beam and make the joint. Figure 2.11 shows the beam carrier in its two positions. Figure 2.12 shows BAT flying with a beam in the beam carrier. By carrying the beam over the shoulder and pointing straight back, the beam's smallest cross section faced the most common direction of flight, reducing drag effects. The beam carrier used pneumatic actuation to produce large torque levels with significantly greater robustness than the electric motors and drive trains of the dextrous arm. The beam carrier thus solved the hardware problems associated with carrying and rotating beams.

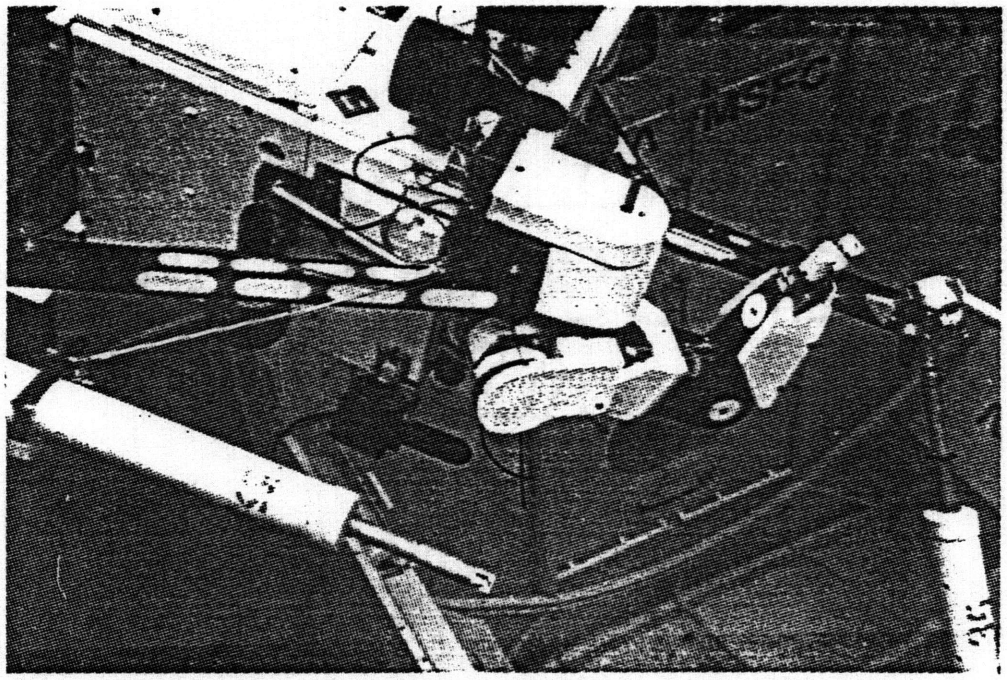


Figure 2.11a Retracted Beam Carrier

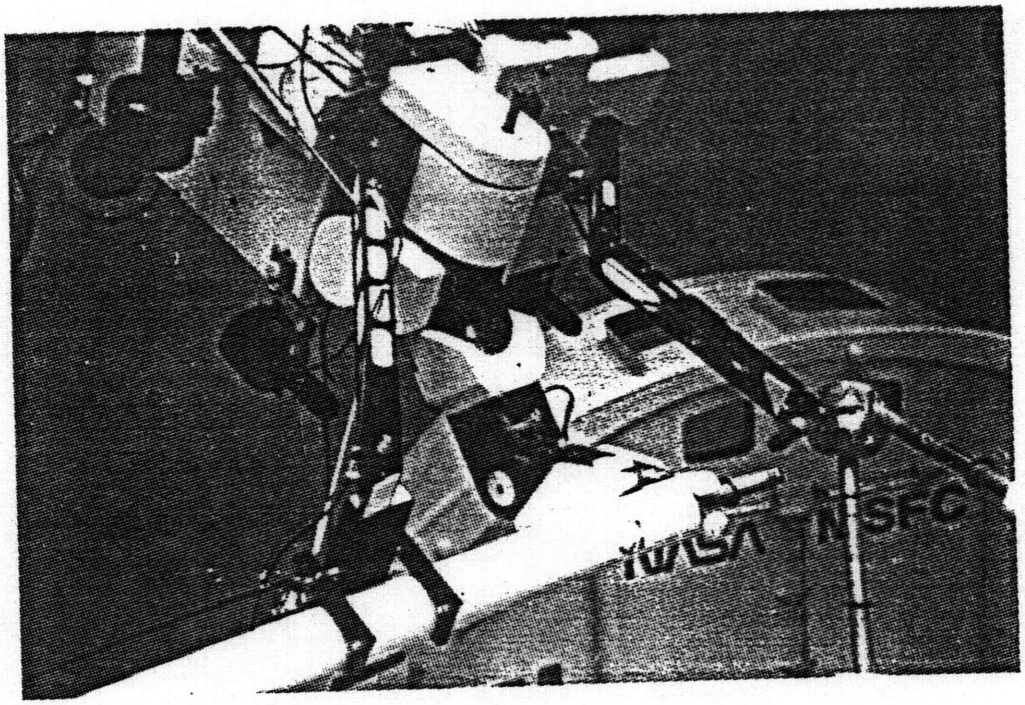


Figure 2.11b Extended Beam Carrier

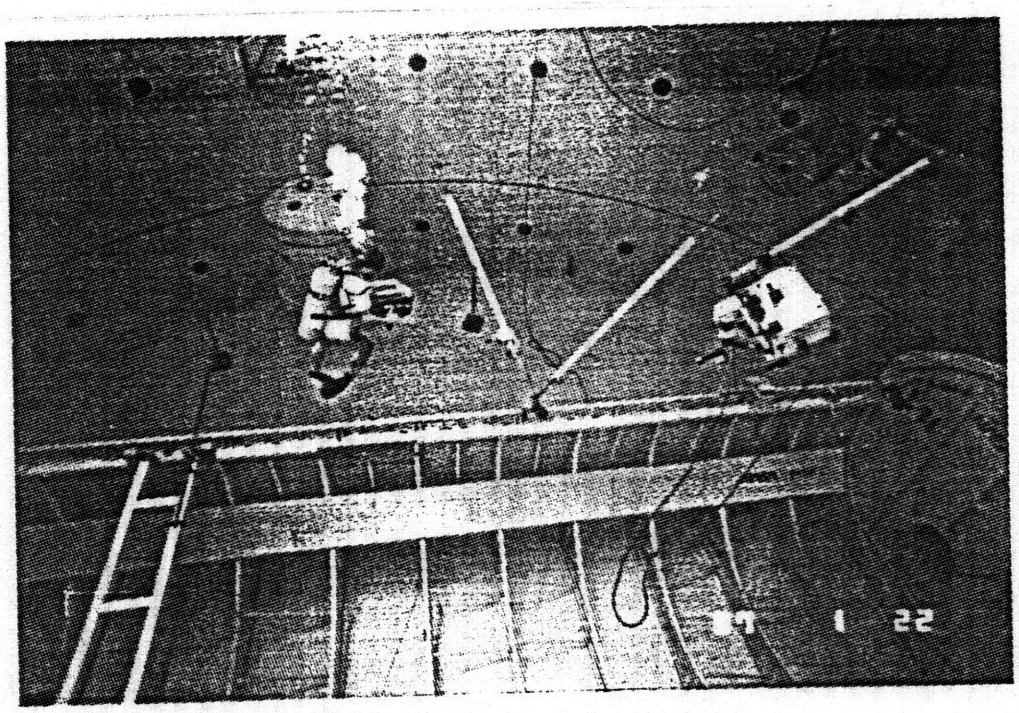


Figure 2.12 BAT Flying with a Beam in the Beam Carrier

2.7.3 Lessons Learned from Problems with Carrying Beams

The one lesson learned from the development of the beam carrier is that it is unrealistic to expect a single dextrous arm to be capable of both fine manipulations needed to assemble joints, and also to be capable of gross manipulation of massive structural pieces. For BAT, it was found preferable to use two separate arms for the two very different tasks.

Using several simpler, specialized arms may in some circumstances be better than using a few complicated dextrous arms. Especially for a task like structural assembly which involves repeated and well defined tasks, one could design a set of specialized arms which are faster and more accurate than more complicated arms. For example, one could imagine building one or two reduced degree of freedom arms for BAT which take the beam from the beam carrier and make the joint. However, having dextrous, non-specialized arms allows the system to handle a wider range of contingency events.

Chapter 3

Teleoperator Assembly of the EASE Structure Under Manual Control

3.1 Neutral Buoyancy Structural Assembly Testing

Once BAT was operational under manual control it was taken to the Neutral Buoyancy Simulator (NBS) of the NASA Marshall Space Flight Center to assemble the EASE structure. Although most of the developmental neutral buoyancy work with BAT had been accomplished at the MIT swimming pool, the NBS was needed to accommodate the large size of the EASE structure.

There were two goals for BAT in this first set of structural assembly tests. The first goal was to demonstrate that a teleoperator could perform structural assembly in a weightless, six degree of freedom environment similar to space. Much had been written and researched about using teleoperators for this, but it had not yet been tried. The second goal was to obtain data on structural assembly productivity of a teleoperator that could be compared to that of space suited humans. This data could be useful for quantifying the trade-offs involved with using teleoperators in place of humans in space.

3.1.1 Neutral Buoyancy Simulator

The Neutral Buoyancy Simulator is a large, cylindrical tank of water, 40 ft. deep and 75 ft. in diameter. Figure 3.1 shows a sketch of the NBS, which is commonly referred to as "the tank". At the bottom of the tank sits a mockup of the Space Shuttle payload bay with the doors open. A neutral buoyancy version of the Shuttle's RMS manipulator arm can also be mounted to the edge of the bay, but was not used in the testing of BAT.

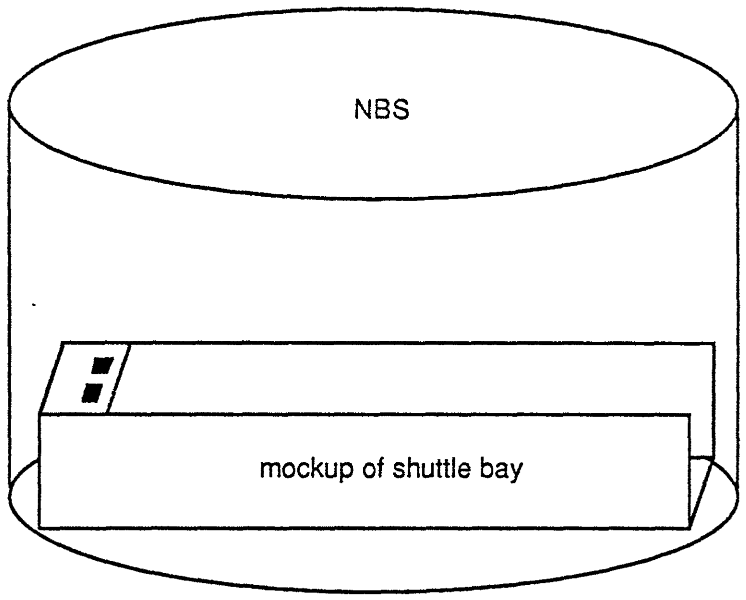


Figure 3.1 Neutral Buoyancy Simulator

3.1.2 Equipment and Test Setup

The testing of BAT required a set of neutrally buoyant beams and joint clusters for building the EASE structure. These were built by the SSL and brought to the NBS.

Figure 3.2 shows the test setup used. The base cluster of the EASE structure was secured to a flat plate normally used for mounting the RMS. Ideally, all of the EASE pieces should be stored in a rack to which BAT would fly and dock in order to obtain beams and clusters. Unfortunately, a rack was not completed in time for the tests. Instead, a rod was mounted to the edge of the bay where the rack would have been. BAT would then dock to the rod, simulating docking to the rack, and a support diver would hold a beam or a cluster in the proper place where the piece would normally have been found in the actual rack.

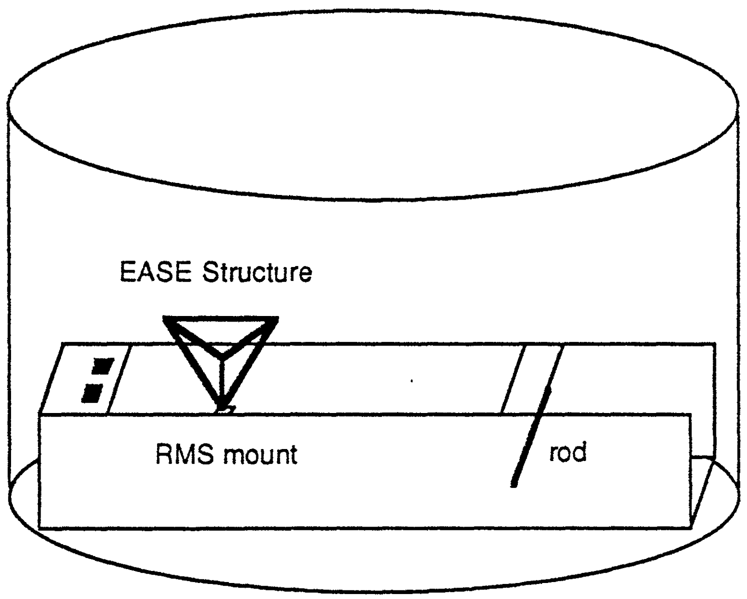


Figure 3.2 Test Setup in NBS

ICS was stationed on a walkway on the outside of, and two-thirds of the way up, the wall of the tank. Although the tank has three levels of portholes circling it, ICS was parked in such a way as to prevent the operator from being able to see into the tank. BAT, when not being used, was serviced on the ground next to the tank. A crane was used to lift BAT in and out of the water. Figure 3.3 shows one side of the tank and the walkway on which ICS was parked. BAT can be seen on its cart on the ground next to the tank, and ICS was parked directly above BAT on the second walkway (third level of portholes).

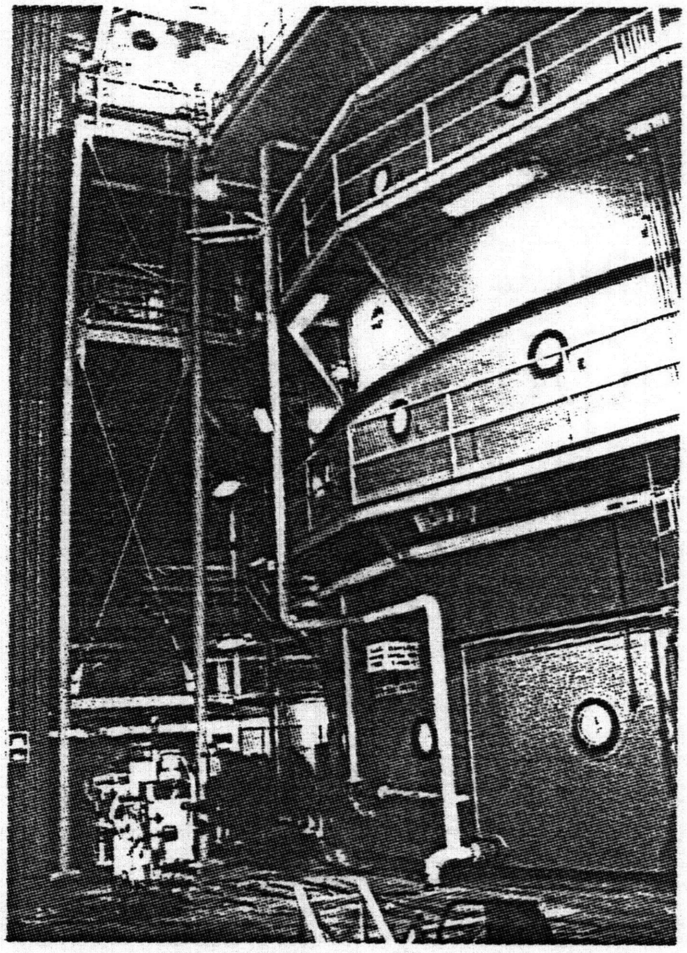


Figure 3.3 Location of ICS During Tests

All data recording was done on video tape. A video cassette recorder on ICS recorded the video image which was seen by the operator through the stereo monitors, and a diver used an underwater video camcorder inside the tank to record BAT's activities. The operator's voice was also recorded on the ICS VCR.

3.1.3 Testing Procedure

Due to limited run time, BAT was not able to assemble the complete EASE structure in one run. A typical run would begin as follows: After all batteries and pressure bottles were charged and installed in BAT, and a successful deck check was completed, BAT would be hoisted up and into the tank as shown in Figure 3.4. Once in the water, two divers would spend anywhere from 15 to 45 minutes balancing BAT to get it neutrally buoyant, in both depth and attitude.

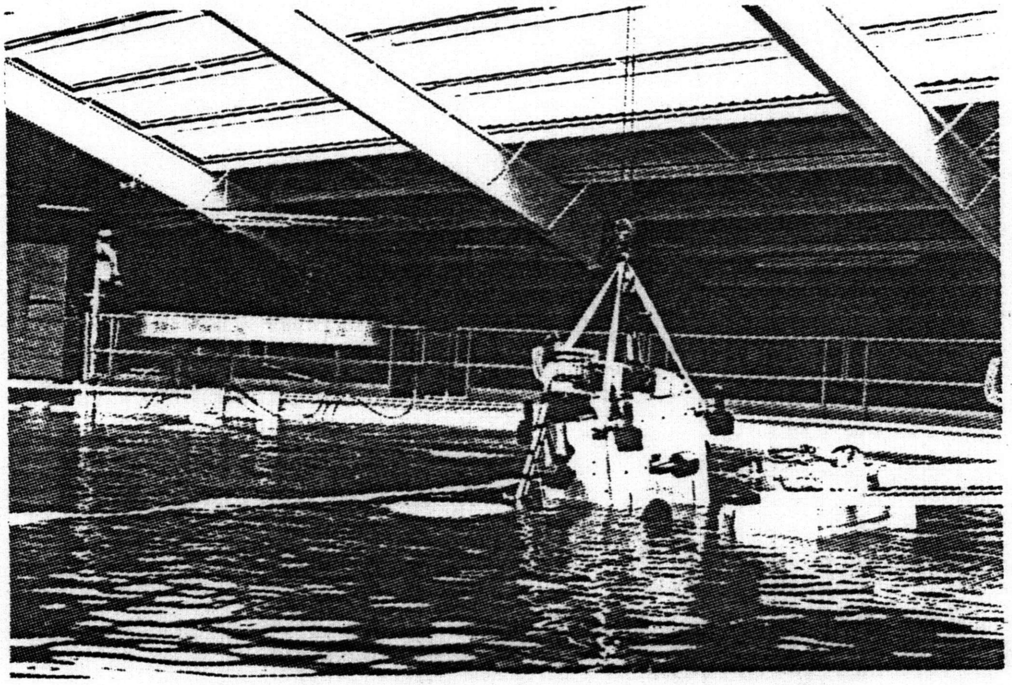


Figure 3.4 Hoisting BAT into the NBS

Once neutrally buoyant, BAT would be powered up, one system at a time. This checkout would typically take 10-30 minutes. Once powered up and running, BAT would be capable of about 45 minutes of active structural assembly. This limit was due to main battery life (used to power the thrusters, dextrous arm, and tilt and pan motors) and assumes a full battery charge, which was not always the case. If there were significant leaks in the low pressure system, then a run could be cut short before all battery power was used up, due to lack of air pressure which was needed to prevent water leaks into the electronics. Also, if the checkout was a long one, control battery power (used to power the on board control electronics) might run out before main battery power would.

During those 45 minutes of of actual run time, things would break, support divers would be busy when needed, and delays involved with taking data and coordinating with other tests going on in the tank simultaneously, would limit the actual run time even further.

The reality of limited run time for BAT, and the uncertainty of BAT's ability to successfully complete the EASE structure, affected how the testing was carried out. Rather than using naive and inexperienced test subjects as operators of BAT (naive and inexperienced test subjects were sometimes used for the space suited human structural assembly testing), it was decided to use an experienced operator in order to achieve at least one successful assembly of the EASE structure in the time we had.

It was also clear that BAT would take several runs to complete a single assembly. Again, in pursuit of the first goal of demonstrating BAT's feasibility, it was decided to first demonstrate each of the steps needed to build the structure. In other words, certain steps in the assembly which were repeated several times (like: fly to the beam rack and get another beam) were only done once to show that they could be done. However, all steps were eventually successfully completed and BAT did eventually perform an end-to-end assembly of the EASE structure over several sessions.

Once it had been established that BAT could assemble the EASE structure, it was decided to have the experienced operator repeat many of the steps several times. The reasons for this were to make sure that none of the first assembly step times were erroneous, and to look for possible signs of learning by the operator, since although the operator was considered experienced, at this point in the tests he had nearly doubled his total operating hours of BAT.

3.2 Data Reduction

As mentioned previously, all of the data was in the form of video tapes of what the operator saw and said (VCR on ICS), and what BAT was doing (swim camera). These video tapes were used to get times for the different tasks in the EASE assembly.

The assembly of the EASE structure was broken down into a combination of four main tasks. They were:

1) Attach a free cluster to a fixed beam
2) Attach a free upright beam to the base cluster
3) Attach a free cross beam to a fixed top cluster
4) Complete a triangle

Figure 3.5 shows examples of each of the four basic subtasks involved in completing the EASE structure.

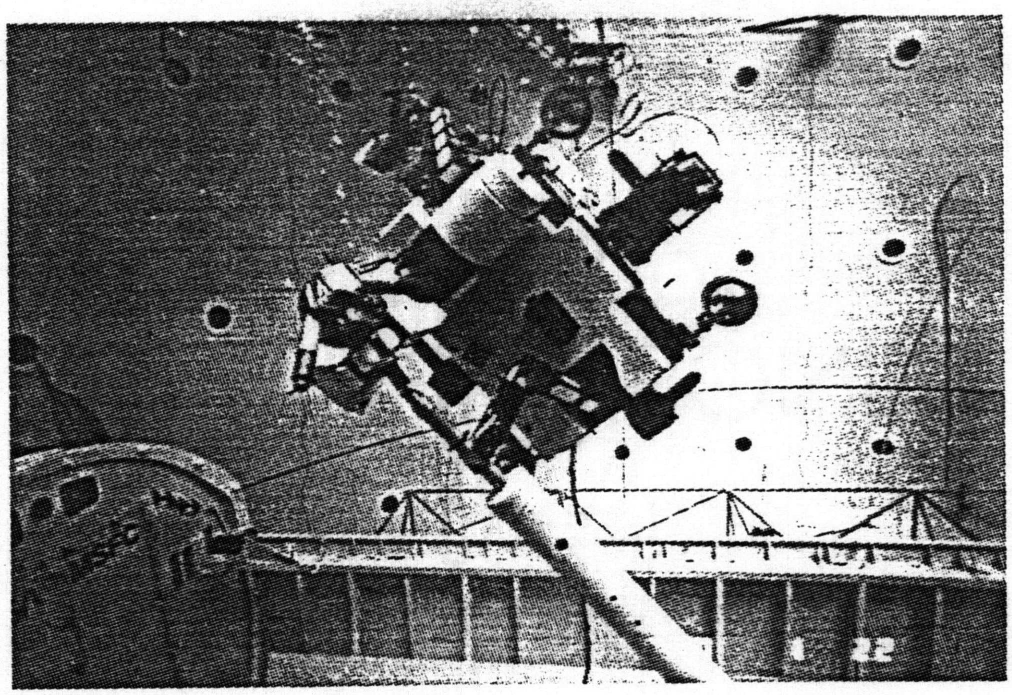


Figure 3.5a Attaching a Free Cluster to a Fixed Beam

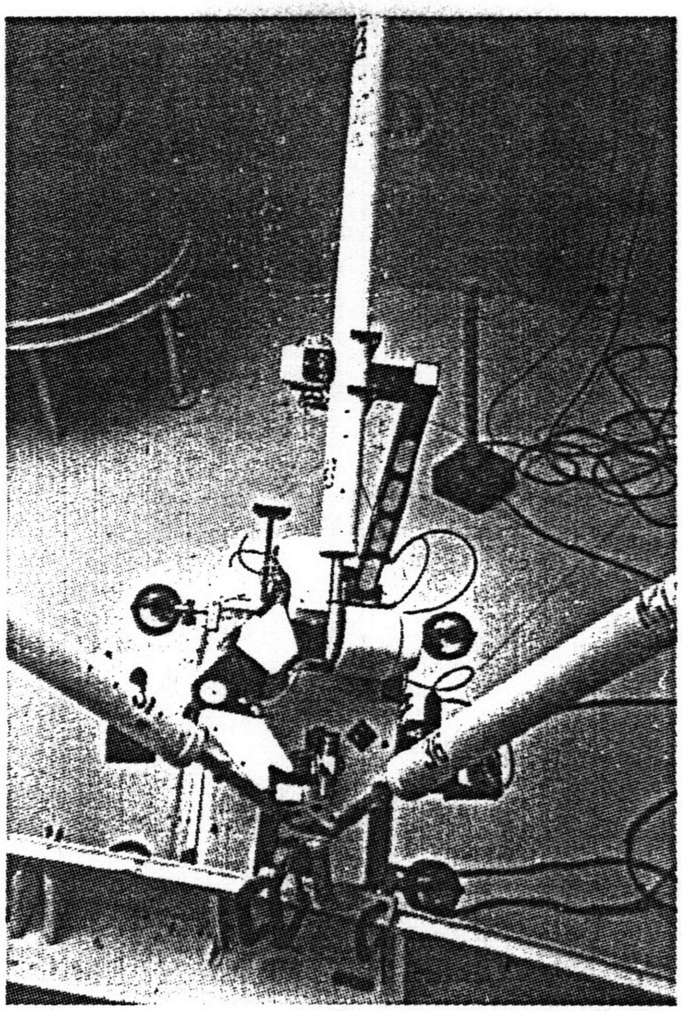


Figure 3.5b Attaching a Free Upright Beam to the Base Cluster

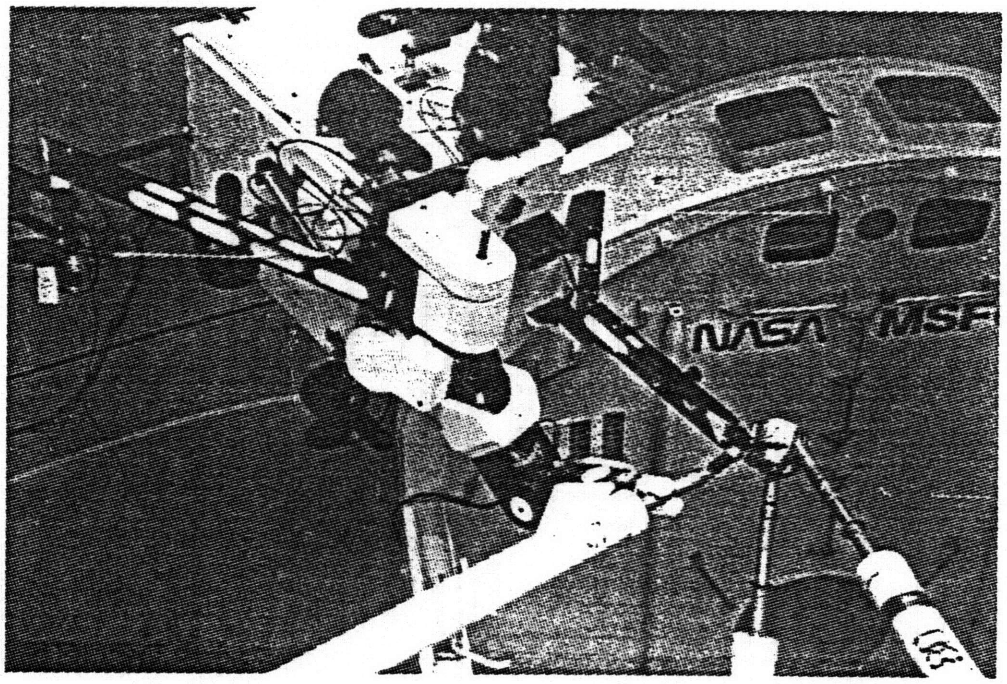


Figure 3.5c Attaching a Free Cross Beam to a Fixed Top Cluster

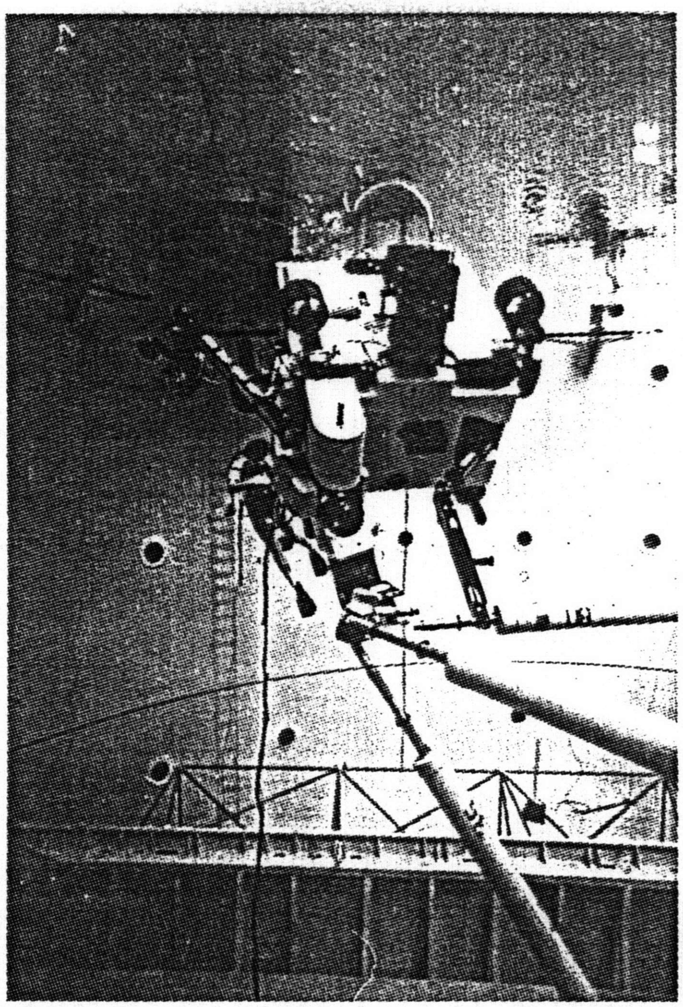


Figure 3.5d Completing a Triangle

In the first main task, a fixed beam can be either a beam which is in the parts rack or a beam which is attached to the structure. While Figure 3.5a shows BAT attaching a top cluster to an attached beam, in a typical assembly, top clusters would first be attached to racked beams, and then the racked beam would be attached to the structure. This saves flying time, since BAT can carry a beam with a cluster over to the structure in one trip.

A complete assembly sequence consisted of the following series of main task completion steps. Note that each step describes which parts were connected and the type of main task (1-4) is indicated at the end of each step.

1) Attach the first top cluster to the first racked upright beam (1).

2) Attach the first upright beam to the base cluster (2).

3) Attach the second top cluster to the second racked upright beam (1).

4) Attach the second upright beam to the base cluster (2).

5) Attach the first cross beam to the first top cluster (3).

6) Complete the triangle involving the first cross beam and the second top cluster (4).

7) Attach the third top cluster to the third racked upright beam (1).

8) Attach the third upright beam to the base cluster (2).

9) Attach the second cross beam to the second top cluster (3).

10) Complete the triangle involving the second cross beam and the third top cluster (4).

11) Attach the third cross beam to the first top cluster (3).

12) Complete the triangle involving the third cross beam and the third top cluster (4).

Note that in steps 3, 4, and 8, the uprights are attached with a cluster attached at one end. While this sequence is not unique, it represents a typical assembly sequence.

By finding a time for each of the four main tasks, the total time of an assembly could be calculated. To find the times for each main task, each main task was broken down into a series of subtasks, which were in turn broken down into a series of task primitives. Times for task primitives were measured from the video tapes and then used to find times for subtasks and the four main tasks.

As an example, one of the common subtasks was "change to belly camera viewing" which involved the operator completing three task primitives:

+ Change to belly camera viewing

  - Center tilt and pan camera view
  - Disconnect head controller
  - Switch to belly cameras

Minus signs (-) denote task primitives, and plus signs (+) denote subtasks. A complete description of the task breakdowns and times measured is presented in Appendix B.

Once times were found for all the tasks, the subtasks were classified into five categories of subtasks:

1) Flying and Docking
2) Beam Manipulation
3) Connections/Clusters
4) Video Switching
5) Manipulator Stowing

Flying and Docking involves all subtasks performed while the grappling claw is not grappling anything (BAT is flying free). Beam manipulation includes subtasks involved with using the beam carrier and the manipulator arm to move and align beams. Connections /Clusters includes all subtasks which involve making joints and handling clusters with the manipulator arm. Video Switching involves switching between the two sets of cameras and using the head controller. Manipulator Stowing includes all subtasks involved with stowing and unstowing the manipulator arm from the position where it is kept while flying.

The times of all of the subtasks included in each category where then totalled to find what percentage of the total assembly time was spent in each category.

3.2.1 Results

BAT was able to successfully assemble the EASE structure.

The time for a complete assembly of the EASE structure by BAT was 89 minutes.

The percentage of total assembly time spent in each category of subtasks were:

1) Flying and Docking	33.9 %
2) Beam Manipulation	18.6 %
3) Connections/Clusters	28.2 %
4) Video Switching	6.5 %
5) Manipulator Stowing	12.8 %

3.2.2 Discussion of Results

The tests demonstrated that a teleoperator could be built to assemble the EASE structure under neutral buoyancy conditions. No problems were found which would indicate that similar systems could not be developed to build structures in space.

A quantitative result was obtained for how long it takes BAT to assemble the EASE structure. How does this time compare to that of humans in space suits? Before answering that question, it should be made clear that this is not a comprehensive attempt to compare the relative merits of using teleoperators or humans to perform structural assembly in space. That is a complicated issue, which involves considering economic and operational factors as well as assembly times.

An experienced space suited human test subject can assemble the EASE structure in 10-15 minutes. This clearly is much better than the performance of BAT under the control of an experienced operator. It is interesting, however, that the initial assembly time for an inexperienced human in a space suit is between 70-80 minutes, which is comparable to that of BAT's first assembly time. It is natural to look at this and say that a teleoperator will be slower than a human in a space suit when performing structural assembly. However, the only accurate conclusion is that BAT, in its current configuration, is significantly slower than a human in a space suit.

The idea behind using naive test subjects in the testing of BAT stems from a desire to quantify a learning rate on a person learning to use BAT to assemble the EASE structure. This could then be compared to the learning curve of how a naive test subject learns to assemble the EASE structure in a space suit, and thereby give a second comparison between teleoperators and EVA. This however would probably be a mistake. The technology involved with designing space suits is quite mature when compared to the technology involved with designing teleoperators. Since improvements in space suit technology have greatly improved astronauts' capabilities in EVA since the beginning of the space program, possible improvements to telerobotic devices hold similar potential for increases in performance.

3.3 Why Improve Control Strategy Instead of Hardware

After having performed the first testing of BAT, it became obvious that there were several ways in which the SSL could change or improve the robotic hardware to make BAT faster at assembling the EASE structure. One example of this would be to modify BAT to carry several beams at once to cut down on the flying time required, since flying and docking accounted for nearly 34% of the total assembly time. Successive improvements such as this to the system may result in better and better assembly times which could be plotted as a "learning" curve of sorts, involving not how a person learns to use a teleoperator, but rather how the system design matures.

Since the current BAT configuration had demonstrated its capacity to assemble the EASE structure, it was more reasonable (in terms of limited test opportunities) to use the existing hardware to investigate advance control system strategies and applications, rather than to spend a great deal of time and effort improving the hardware.

3.3.1 Problems with Manual Control Strategy

The assembly time results presented in section 3.2.1 were not the only results to come out of the EASE assembly testing of BAT. Several important results concerning problems and difficulties with operating BAT under manual control came up which are not reflected in the assembly time data. The following problems involving the operator using manual control were found during the testing:

1) The human operator began to experience significant fatigue if two runs were completed in the same day.
2) The human operator often made mistakes in critical sequences of task primitives.
3) The human operator often became confused as to where to dock to the structure or what to grab with the manipulator arm, and often did not perform sequences of main tasks and subtasks in an optimum manner.

3.3.2 Operator Fatigue

The first problem of operator fatigue implies that there may be severe limitations on how long a single human can operate a space teleoperator system under total manual control. If fatigue sets in after only about an hour of actual run time (A typical run lasted 45 minutes), than that is much worse than the six hours of EVA which an astronaut in a space suit is capable of. This result therefore strongly suggests that a reduction in the operator's workload will be necessary (something other than total manual control at all times) for a teleoperator to be useful. (This assumes that the operator is in space also, and is therefore a limited resource. If the operator is on the ground, then an alternate answer would be to simply have an army of operators at hand.) Problems with confusion may also be caused in part by operator fatigue.

The first problem involving fatigue is indicative of the fact that operating BAT is an extremely taxing process for the human operator and successful automation of any of the subtasks involved should help. The human operator is often saturated with things to do, and the ability to hand off any of his tasks to a computer would at least reduce operator workload, and thereby fatigue, and thereby extend the length of time that the operator can run BAT. This handing off could involve either low level tasks involving the details of accomplishing specific subtasks, middle level tasks like keeping track of the location and orientation of BAT and structural elements, or high level tasks such as planning and decision making about both the assembly process and the performance of the BAT system.

3.3.3 Operator Mistakes

The second problem involving operator mistakes is potentially the most dangerous one for the system. There are several times in the operation of BAT in which if a mistake in completing a task primitive, creates the possibility of failure of the subtask, or even physical damage to the robot.

These types of mistakes were often made when the operator was confronted with a long series of task primitives, like all of the details involved with attaching a beam to a cluster. The mistakes were either completing the task primitives in the wrong order, substituting the correct completion of a task primitive with the completion of a different and incorrect task primitive, or simply leaving out completely the completion of one of the task primitives. Depending on the situation, the result of any of these mistakes ranged from the insignificant, to the aborting of the current subtask, to damage to the hardware.

Computers tend to be quite good at keeping track of details such as the smaller tasks required to complete a task. Knowledge about the specific order of the tasks, along with if and when a task can be left out, misplaced in the sequence, or swapped for another equally effective task, can effectively be included in a computer program. This could either be used as part of a supervising program (in place of the human as supervisor) which decides when to use certain supervisory routines, or as part of a monitoring system which would prevent the human operator from making mistakes.

3.3.4 Operator Confusion

The problem of operator confusion is caused by inherent limitations in a person's ability to not get confused in a complicated situation, by a high operator workload associated with manual control of a complicated machine like BAT, and by problems with the video system which made it difficult for the operator to continuously look about and orientate himself.

The operator tended to get confused as to where to dock to the structure while attempting to attach a piece. This resulted in the operator actually having to ask for help from the support divers or someone on deck. Once confused, the one way for the operator to try to figure out where to dock would be to look around at the structure. However, this would often necessitate backing up and either panning the belly cameras around with the thrusters, or switching to the tilt and pan, which both took time and often only confused the operator further, as flying with the tilt and pan camera is quite confusing.

The operator did not always accomplish series of subtasks or main tasks in the same sequence, often resulting in a sequence which was not the best. This indicates that the human operator had difficulty effectively planning out the assembly sequence and the substeps involved, due to having many other details to worry about.

The above results suggest that AI techniques could be used to help the operator to plan out the assembly and then provide information as to where to dock and what component to attach next. The computer would keep track of high level goals, while the operator would take care of the small details. In a sense, taking care of the details is more difficult than doing the high level planning (due to the sensing, recognition, and control problems associated with the details) and therefore may be the job which should be given to the human operator. This would be particularly important for a structure more complicated than the EASE structure, since its greater complexity would aggravate the problems of confusion and difficulty in planning while busy with details.

Chapter 4

Design Goals and Specifications for ROBIN

4.1 Chapter Organization

After having investigated BAT's performance under a manual control strategy, it was decided to pursue research into AI applications for the system. This chapter outlines several possible AI applications which were considered, and how one of them was selected for implementation.

Four different forces shaped the process of developing ideas for possible AI applications and selecting the most feasible one. Two of these forces drove the generation of ideas, while the other two controlled the selection of which one to implement.

4.2 The Two Idea Generating Forces

The two forces which guided the generation of ideas were:

1) The practical results from the development of BAT and neutral buoyancy structural assembly tests, which indicated that some kind of automation was needed to solve the problems of manual control and to improve BAT's performance.

2) Near and long term research goals which involved investigating the use of AI in controlling a teleoperator.

4.2.1 Neutral Buoyancy Test Results

From the structural assembly testing of BAT, it became apparent that the human operator of BAT was overloaded. This resulted in operator fatigue which limited the operational run time of BAT. Since the operator was responsible for all control decisions and commands, mistakes were often made when a task required a precisely ordered execution of many small steps. Also, the many details of operating BAT and performing structural assembly often confused the operator, resulting in reduced performance.

Several other results came from the developmental work on making BAT operational under direct manual control. The most important was that situations arose which required the completion of two tasks simultaneously. If the two tasks both required the operator to use the same body part to control each task, then the operator could not complete the compound task. With BAT, such situations were handled by splitting up the tasks so that the operator only needed to control one task at a time. However, there still arose times when the operator had to fly and manipulate at the same time. This occurred during the completion of triangles, but happened while docked to an attached beam and only involved small motions. This situation was handled by using the left hand to operate both hand controllers, while the right hand controlled the master arm. Such situations would be more difficult to handle in a teleoperator system with two or more dextrous arms.

Another general result is exemplified by the fact that over 12% of the operator's time was spent stowing and unstowing the manipulator arm from the position it is kept in while flying. If the arm was commanded to hold position while flying, power would be consumed by its motors (to overcome drag forces), and the motors would also experience excessive wear, reducing their lifetime. Therefore, the arm was positioned to grab a handle on the grappling arm and then powered down before flying was begun. This meant that before flying the operator would have to stow the arm, and then unstow it again after flying. This forced the operator to spend large amounts of time taking care of small details associated more with the specific design of the hardware than the general nature of the task.

4.2.2 Near and Long Term Research Goals

BAT was first made operational using a manual control strategy. However, it had always been the LOOP group's intention to implement more advanced control strategies which involve machine intelligence. As will be described below, BAT unfortunately was built in such a way that it would be extremely difficult to implement many of the possible ideas which were proposed. This motivated the initiation of a new teleoperated robot project called Apparatus for Space TeleRobotic Operation (ASTRO). ASTRO was to build upon what was learned from BAT's development and was to be designed so that it would be much easier to implement advanced control strategies involving AI.

However, since ASTRO would not be ready for some time, it was decided to implement whatever was possible on BAT in the way of AI. This would allow the demonstration of certain limited capabilities which would provide near term research results, and would also help to lay the ground work for similar research efforts with ASTRO.

4.3 Possible AI Applications for BAT

There were four main ideas for how AI could be used to improve the control strategy of BAT.

4.3.1 Supervisory Control Routines

To help lessen operator overloading and fatigue, and to eliminate the need for the operator to take care of hardware specific details (like stowing the arm) it was proposed that much of what the operator does should be downloaded into supervisory control routines. Typically, the tasks which are the best candidates for being done with a supervisory routine are those which are well specified and repetitive.

The first area where such routines could be useful would be for manipulation. Supervisory functions which could control the arm would reduce or eliminate the need for the operator to use the master arm. Since the manipulation tasks required by structural assembly are very regular and are repeated over and over again, they seem to be ideal candidates for supervisory control.

The second type of tasks which are suitable candidates for supervisory control are flying tasks. Developing supervisory flying would be useful for repetitive and uniform tasks such as flying to and docking with the parts racks. This would also be especially useful in situations in which the operator needs to both fly and manipulate at the same time. By handing off the flying task to a supervisory routine (possibly a position and attitude hold function), the operator can then concentrate on the control of the manipulator arm. That the reverse could also be done, by having the operator fly and a supervisory routine control the arm.

4.3.2 Operator Monitor

To help the operator avoid making mistakes while in manual control, a program could be designed which monitors the operator's commands to BAT and then warns of, or actually steps in to prevent, incorrect command sequences.

The complexity of such an operator monitor could vary considerably. The simplest system would be one which would only monitor simple commands and recognize series of commands which could lead to a bad situation. The most complicated system would be one which models the whole assembly process, senses what is happening at the worksite, monitors the operator's commands, and attempts to prevent mistakes. Such a system could even monitor the operator's physical state to warn of onsetting fatigue.

As such a system becomes better at understanding the task of operating a teleoperator, it could become more capable of taking over the job itself, approaching autonomous control. Also, such a monitoring system could possibly be made to learn from the operator by observing the commands the operator generates, and what the result are at the worksite.

4.3.3 Assembly Sequence Planner

To help the operator resolve confusion about where to dock to the structure, or what assembly step should be done next, some kind of planning program could be developed which could be queried by the operator for information. Such a program could also contain the details of each step in the assembly process. Such a system would then be an expert system which would provide the operator with information about all aspects of the assembly process, and what should be done at any given point in the assembly. Such a system could have a varying degree of feedback from the worksite and the actual state of the assembly. This type of system would also demonstrate many of the high level intelligence capabilities which would be needed for autonomous operation.

4.3.4 Autonomous Control

It would be desirable if a teleoperator could evolve into a completely autonomous robot by incorporating new AI capabilities as they are developed. One idea for this would be to keep developing more and better supervisory routines which the human operator uses as tools to get the job done. Once this set of tools is complete, then all one needs to develop is a program which decides how to use those tools and then the robot becomes autonomous.

This, however, may not work. Supervisory control routines are developed in part as a response to a human's weaknesses. Their autonomous capabilities are also limited to a specific task. Therefore, to control the complete set used by the human operator, one would need to write a program which basically is an artificial human.

Instead, autonomous capabilities should be added which are always functioning, instead of being turned on and off by calls to a supervisory routine. The human operator would then learn to use the teleoperator system with these autonomous capabilities operating in the background. While supervisory control could still be used, autonomous capabilities should probably be incorporated in such a way that they are truly autonomous, in that the human does not control them, only live with them as part of the system. This approach would be more likely to facilitate the evolution of the teleoperator into an autonomous robot.

Therefore, the demonstration of AI technologies which would be useful for autonomous control should involve ones which are continually operating independently of the human operator, except for when the two need to communicate.

4.4 The Two Constraining Forces

There were two forces which controlled the selection of one of the above ideas to be implemented:

1) Inherent limitations in the BAT system involving the feedback available from the worksite, and limitations in the ability to interface any new software or sensor systems into the existing control system of both BAT and ICS.

2) The limited availability of certain resources, specifically computer resources with which to implement different AI systems, and down time on BAT for major changes needed for integration of sensor or AI systems into the BAT system.

4.4.1 Inherent Limitations in the BAT System

There were two basic problem areas with the BAT system which make it difficult to incorporate any high level machine intelligence control.

The first problem was that of providing the necessary feedback to the computer which is running the AI software. First of all, BAT itself had virtually no feedback data available which a computer could use. Only joint positions and video was fed back up through the commlink. This would not be enough feedback unless a machine vision system could be developed to extract information from the video images.

The second problem involves the computing and interface capability of the IBM PC used in ICS. The custom operating system used to run ICS is written in C, which is not particularly well suited for high level AI programming. The memory limitations of the PC also would restrict what could be added by way of AI software. These two facts meant that any AI software system should probably run on a separate computer.

Unfortunately, the interface capability of the PC was essentially used up in interfacing with all of the controls in ICS. This meant that it would be difficult to interface another computer with the ICS PC with out major downtime for the redesign and rebuilding of the working computer control system.

4.4.2 Available Resources

There were two restrictions on the resources available for implementing new systems on BAT.

First, alterations to the BAT system which involve major downtime were unacceptable due to the importance of the other research projects involving robotic manipulator control algorithms and further testing aimed at generating a larger database on manually controlled teleoperator assembly of space structures.

Second, although it was best to use a second computer, the choice of available machines was limited. Due to funding constraints, the only computer available was an IBM AT, which is not as capable as other machines designed specifically for AI research.

4.5 Selection of One Application

The first possible AI application considered was supervisory control. It was felt that the limited computing capabilities of the ICS PC were still adequate for implementing some simple manipulation routines. Unfortunately, the lack of any feedback other than joint positions made this quite difficult. Real time vision systems and other active sensing techniques for use under water were investigated without much success.

However, an attempt was made by another graduate student to develop a set of open loop control routines which relied on moving the arm through preplanned trajectories which assumed specific states of the world (see Reference 2). The most promising application for such open loop routines was the stowing and unstowing of the arm, since the geometry of the handle and arm was always the same, unlike the structural elements which are being assembled. While such routines would be useful in studying how an operator would use supervisory routines, they would not demonstrate real supervisory control, since no closed loop control would be involved.

Supervisory flying routines were also a possibility. An acoustic positioning system was under development but not yet operational. Another neutral buoyancy vehicle (MPOD, see Reference 3) had been chosen as the proving ground for supervisory flying, and had been equipped with attitude and rate sensors. Unfortunately, neither the positioning system nor the needed control algorithms were to the point where they could be transferred to BAT.

The second possible application was the operator monitor. The main barrier to implementing this was the lack of interface capability with the ICS PC. Without access to the commands being sent to BAT, such a system could not be implemented. Even if the interfacing was possible, the lack of sensor information from the worksite made its potential effectiveness questionable, since the system would have to evaluate the operator's commands against an assumed state of the world. While it would be possible to have the operator update the system's model of the world state, this would not be possible during the times when the operator is busy making commands and the monitoring system would be most useful.

This left the assembly sequence planner as the most promising candidate for implementation. Even without any feed back from the worksite, such a system could still be useful in that it could plan out the assembly and answer question about how things should be done. Developing an assembly sequence planner would also demonstrate part of what would be needed for autonomous control.

Therefore, it was decided to develop an assembly sequence planning program. The planner was to help solve two of the problems encountered in the neutral buoyancy testing of BAT. The problem of the operator not being able to consistently perform sequences of main tasks in the proper order was to be solved by providing the operator with a step by step plan of how to perform the assembly. Operator confusion was to be eased by providing the operator with information on what pieces were involved with each step, and how and where they were to be attached, so that the operator would know where to dock to the structure.

4.6 ROBIN

This assembly planning program was named ROBIN as an acronym for ROBot INtelligence. The ROBIN project had two main goals.

The first goal was to develop an AI system which could do the high level problem solving involved with performing structural assembly. This would be useful as a real time planner for assisting a teleorobot operator, and would also demonstrate intelligence capabilities which would be needed by an autonomous robot performing structural assembly.

The second goal was to develop an AI system which would be useful in the BAT system during actual structural assembly. ROBIN was to help solve the problems of operator planning difficulty and confusion.

The EASE structure was selected as the structure for which to develop ROBIN. It can be argued that the EASE structure is too simple to represent a realistic space structure problem. The author, however, argues that large structures will be built one cell at a time, and a planning program for them could consist of one module which plans out one cell, and another module which plans out how the cells are put together. Therefore, a large part of the problem involves planning the assembly of a single cell structure like EASE.

4.7 Design Specifications for ROBIN

ROBIN was to be a structural assembly planning program which would be useful to the operator of BAT while assembling the EASE structure.

ROBIN was to model the EASE structure and the assembly process in a way which reflects the true physical nature of both the hardware and how it is put together by BAT. It was felt that it was important to have ROBIN model the real world and its physical constraints, using them to plan out assembly steps in real time. Otherwise, a canned program which simply presented possible preplanned assembly sequences depending on initial conditions could serve the same function as ROBIN, without demonstrating the problem solving capabilities needed for autonomous control. Also, three dimensional modeling of the structural pieces and the work environment should be done to facilitate integration of feedback data from future sensors such as the acoustic positioning system under development.

Since the most efficient way for ROBIN to communicate to the human operator is through visual information, ROBIN was designed to drive a graphics display to which the operator could refer whenever he needed information. ROBIN's display was to graphically show the state of the structure built so far and the state of the parts racks. The display should also show what the next step suggested by ROBIN was, along with the pieces involved, where they were in the parts racks, and how and where on the structure they were to be attached.

Since ROBIN would not have access to direct information concerning the real world and the state of the structure, it would have to depend on the human operator to make sure that ROBIN's model of the state of the world was accurate. This, however could be done during times when the operator is not overloaded, such as after having completed a step. As long as the operator was performing the assembly as suggested by ROBIN, the operator would only need to tell ROBIN that each step was successfully completed, which would require only one bit of information per step.

However, when the operator did deviate from ROBIN's suggested plan, the operator would have to inform ROBIN of what was done instead, since ROBIN could not sense what the operator did. ROBIN would therefore need the capability of being able to replan the rest of the assembly, from whatever state the operator had put the partially assembled structure into with his alternate step. If ROBIN eventually was able to independently sense the state of the structure, there would be no need for the operator to communicate information to ROBIN at all, since ROBIN could then update its internal model without the operator's help.

The operator should have a quick and efficient way of communicating what was done as an alternate step. A multi-leveled menu system was decided upon, which required the use of only four keys to communicate any possible alternate step. The communication was simplified by having ROBIN figure out what possible steps in any situation were possible, and then presenting only those to the operator to select from. The menu system also was designed to screen out attempts by the operator to specify illegal alternate steps.

Chapter 5

Knowledge Representation in ROBIN

5.1 Knowledge Representation with Prolog

ROBIN was written in the computer language Prolog (specifically Turbo-Prolog © Borland Int.) and runs on an IBM Personal Computer AT with 640k RAM.

Prolog is a resolution based theorem proving language which uses an automatic backtracking strategy to search for possible solutions to a specified goal. References 4 and 5 present a complete description of the language and how it functions.

In prolog, facts are in the form of "predicate(object1,object2,...)" and their existence indicates that the predicate is true for that object. Facts can be stated explicitly in the clause statements of the program, in the active database, or in the execution environment. In the last two instances, the facts are generated by the rules of the program. Objects are entities which do not have a truth value, and objects can have the form object(object1,object2,...).

5.2 Modelling of Structural Elements

The EASE structure is assembled out of beams and joint clusters. It was necessary to represent these structural pieces in as simple a form as possible, but still retain the important geometric qualities of each.

5.2.1 Point Representation of Beams and Clusters

Figure 5.1 shows conceptually how a beam is represented in ROBIN. An actual beam is made up of a thick beam body with two thinner beam ends. The beam is simplified to consist of four points. The body of the beam spans the two interior points, and each beam end spans from one of the interior points out to the nearest outer point.

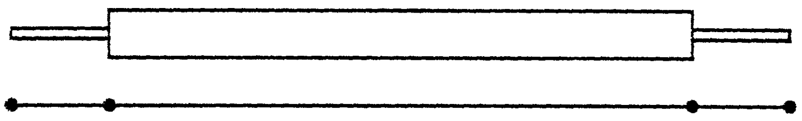


Figure 5.1 Beam Representation

Figure 5.2 shows conceptually how a cluster is represented in ROBIN. An actual cluster is made up of three mushroom ends connected together at a vertex. The cluster is also simplified to consist of four points, which form a tetrahedron. Each of the three mushroom ends span from the point at the vertex out to one of the three mushroom end points.

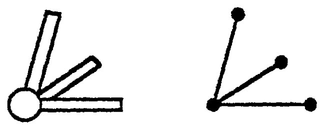


Figure 5.2 Cluster Representation

5.2.2 Prolog Domain Declarations of Beams and Clusters

Beams are objects of the form beam(beam_num) where beam_num belongs to the symbol object domain. There are six beams in the EASE structure and the objects associated with them are:

```
beam(beam1)
beam(beam2)
beam(beam3)
beam(beam4)
beam(beam5)
beam(beam6)
```

Similarly, joint clusters are objects of the form cluster(cluster_num) where cluster_num belongs to the symbol object domain. There are four clusters in the EASE structure: one base_cluster which is always fixed in place, and three top clusters:

cluster(base_cluster)

cluster(top_cluster1)

cluster(top_cluster2)

cluster(top_cluster3)

Each beam has two beam ends associated with it. Beam ends are objects of the form beam_end(beam_num,beam_end_num) where beam_num describe which beam is involved and beam_end_num belongs to the symbol object domain and is either be1 or be2. Figure 5.3 shows a beam and the three objects associated with it so far.

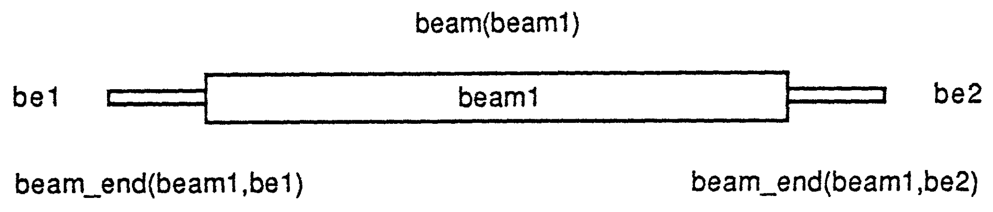


Figure 5.3 Beam and Associated Objects

Each cluster has three mushroom ends associated with it. Mushroom ends are objects of the form mush_end(cluster_num,mush_end_num) where cluster_num describes which cluster is involved and beam_end_num belongs to the symbol object domain and is either me1, me2, or me3. Figure 5.4 shows a cluster and the four objects associated with it so far.

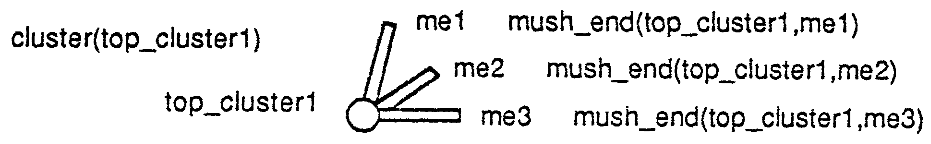

cluster(top_cluster1)
top_cluster1
me1
mush_end(top_cluster1,me1)
me2
mush_end(top_cluster1,me2)
me3
mush_end(top_cluster1,me3)

Figure 5.4 Cluster and Associated Objects

To simplify certain logical constructs it was decided to use the general object piece to refer to either a beam(_) or a cluster(_) (note that in Turbo Prolog the underscore character "_" is the argument wildcard). Similarly, the general object end refers to either a beam_end(_,_) or a mush_end(_,_).

5.2.3 Representation of Geometric Positions

Each piece must have a geometric position to be used for world modelling and graphics generation. The three dimensional coordinates of a point are represented by the object coords which is a list whose elements belong to the object domain of real numbers. While the length of the list is not specified, only the first three elements are used. Similarly, a position is a list of four coords which correspond to the four points of a piece as described in Section 5.2.1. A list in Turbo-Prolog is represented as elements separated by commas enclosed by square brackets: [e1,e2,...,en].

Therefore, a coords has the form: [real,real,real] where real designates the real number object domain, and position has the form: [coords,coords,coords,coords].

The location of a piece in space is given by the object location(piece,position), in which piece is either a beam(_) or a cluster(_), and position is a list of four coords as described in the preceding paragraph. The object location(_,_) is actually a database predicate but is described here for continuity (Section 5.3 describes the other database predicates.). Two typical examples of this are:

```
location(beam(beam1),[[X1,Y1,Z1],[X2,Y2,Z2],[X3,Y3,Z3],[X4,Y4,Z4]])
location(cluster(top_cluster1)),[[X1,Y1,Z1],[X2,Y2,Z2],[X3,Y3,Z3],
        [X4,Y4,Z4]])
```

Note that in Turbo-Prolog variables begin with a capital letter. In the example above, the actual numbers of the coordinates are represented by variables.

In the location of a beam, the four coords of that beam's position correspond to the four points of the beam. The convention for point and coords correspondence is shown in Figure 5.5.

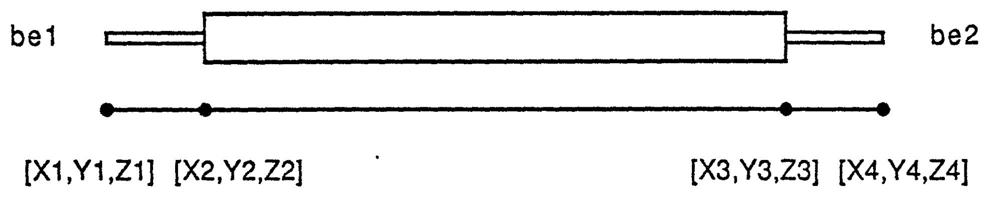


Figure 5.5 Beam Point to coords Correspondence

In the location of a cluster, the four coords of that cluster's position correspond to the four points of the cluster. The convention for point and coords correspondence is shown in Figure 5.6.

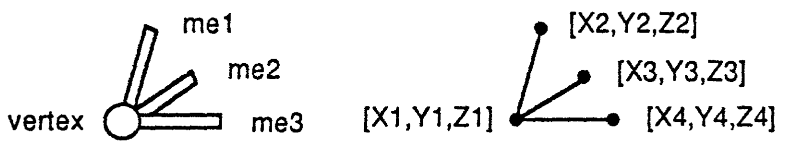


Figure 5.6 Cluster Point to coords Correspondence

5.2.4 Storage of Pieces in Parts Racks

Both the beams and clusters are assumed to be stored in parts racks if they have not yet been attached to the structure. Conceptually the part racks are simply lists of pieces. The beams are stored as a list of pieces called beams and the clusters are stored in a list of pieces called clusters.

5.3 Database Management

Turbo Prolog has a dynamic database to which one can add facts (assert) and delete facts (retract). Here the terms fact and clause will be used interchangeably. The important general predicate clauses which are declared as database clauses are listed below:

```
beam_rack(beams)
cluster_rack(clusters)
attached(piece)
racked(piece)
free_mush_end(end)
free_beam_end(end)
free_racked_beam_end(end)
connected(end,end)
location(piece,position)
```

The two parts racks are stored in the database under the predicates beam_rack(beams) and cluster_rack(clusters).

The predicates attached(piece) and racked(piece) keep track of whether a piece is attached to the structure or is still in one of the parts racks.

The predicates free_mush_end(end) and free_beam_end(end) denote which ends of pieces attached to the structure are available to have another piece attached to them. The predicate free_racked_beam_end(end) is needed because one can either attach a cluster to a racked beam or a beam already attached to the structure.

The predicate connected(end,end) is used to keep track of which beam ends are connected to which mushroom ends. The convention for the end order is always connected(Beam_end,Mush_end).

The predicate location(piece,position) is described in Section 5.2.3 and is the fact which is used for graphics generation.

Note that there are several other database predicates not mentioned here, mainly for taking care of lower level details. Appendix C contains a complete listing of the ROBIN code.

5.3.1 Turbo Prolog Database Commands

Turbo Prolog searches through the database from the top down, and provides three predicates for accessing the database.

The first two predicates are asserta(FACT) and assertz(FACT). Both predicates take the clause FACT and add it to the database. The difference between the two predicates is that asserta adds FACT to the top of the database before all other clauses, while assertz adds FACT to the end of the database after all other clauses. Except for a few special instances, ROBIN primarily used assertz so that the earliest assertions are the first found in the search of the database.

The third predicate is retract(FACT) which searches through the database for the first occurrence of a match for FACT. If found, the match is deleted from the database and the retract(FACT) predicate returns true. If no match for FACT is found, it returns false.

5.3.2 ROBIN's Permanent and Temporary Database Needs

It was decided to use the database to store both the current state of the world and possible future states which are generated during planning, as well as suggested steps from both ROBIN and the operator. Therefore it was necessary to establish a convention of differentiating permanent facts from temporary ones. *Permanent facts* are considered to be any fact which is associated with the partial structure which has already been built. There are two types of *temporary facts*. The first are those which the planning algorithm generates and which will definitely not be needed once the plan is finished and should be purged from the database. The second type of temporary facts are those which are generated when the next possible step in the assembly process is either suggested by ROBIN or the operator. In either case, all facts involved with actually executing that next step are generated. If both ROBIN and the operator agree on the next step, then those temporary facts are made permanent and that step is assumed to be part of actual assembly sequence so far. If the suggest step is not agreed upon, then the temporary facts are purged so that a different step may be suggested.

Most often the planning algorithm will be asked to plan out the assembly of the remaining structure (temporary facts) after part of the structure is actually built (permanent facts). The planning algorithm therefore must be able to handle permanent and temporary facts in the same way, since a temporary fact and a permanent fact both have equal impact on the planning of the rest of the assembly.

5.3.3 Implementing Both Permanent and Temporary Database Storage

Two database predicates were defined to handle the permanent vs temporary fact problem. They are temp_assert(dbasedom) and temp_retract(dbasedom). The object dbasedom is a standard domain which covers all database predicates like those described in Section 5.2.4.

If a fact is a permanent fact, then in order to add it to the database one would use the standard assertz(Permanent). To add a temporary fact one would use two statements:

```
assertz(Temporary)
assertz(temp_assert(Temporary))
```

Note that the addition of a temporary fact usually goes hand in hand with the retraction of some other fact which could be either a permanent or a temporary one. For example, when adding the fact attached(beam1), one must also remove racked(beam1) to indicate the removal of the beam from the rack during its attachment to the structure.

This means that while in a process like planning or suggesting, in which one is asserting temporary facts, any facts retracted must be temporary retractions, regardless of whether or not the retracted fact was a temporary or permanent assertion:

```
retract(Temp_or_perm)
assertz(temp_retract(Temp_or_perm)
```

In the case of planning, once a new plan has been found, the program should save the plan and then restore the database to its original state. To do this the rule restore_old_database is used. This rule first restores all facts which were temporarily retracted. This is done by searching through the database and finding all temp_retract(X) statements, and executing retract(temp_assert(X)) and assertz(X). Again, note that this is done for all temporarily retracted facts, whether or not they are permanent or temporary.

Next, restore_old_database erases all temp asserts by searching through the database and finding all temp_assert(X) statements and executing retract(X).

In the case of suggesting a step which may or may not be included in the actual assembly sequence, two options are possible. The first possibility is that the suggested step is rejected, in which case restore_old_database is executed to undo the suggested step.

The second possibility is that the step is accepted and the rule keep_current_database is executed instead.

The rule keep_current_database simply unmarks each temp_retract(X) with retract(temp_retract(X)), and unmarks each temp_assert(Y) with retract(temp_assert(Y)). This simply removes all temp_ statements and leaves the statements themselves, so that any temporary facts become permanent, and any temporarily retracted facts stay retracted.

Chapter 6

ROBIN's Planning Algorithm

6.1 Representation of Plans

It was desirable to have a concise way of representing plans which was as simple as possible, and yet would completely specify a unique assembly sequence.

6.1.1 Five Assembly Steps

Plans are represented as a sequence of steps in which each step specifies the connection of a beam end to a mushroom end. Since the geometric location of the base cluster is always known, the geometric positions of the pieces attached through the execution of a plan are found through the propagation of geometric constraints. In the domains declaration, a step is defined as one of five possible objects, and a sequence is defined as a list of steps (In Turbo-Prolog, name* denotes a list.):

```
step =  beam_to_attached_cluster(end,end);
        cluster_to_attached_beam(end,end);
        cluster_to_racked_beam(end,end);
        beam_with_cluster_to_attached_cluster(end,end);
        triangle(end,end)

sequence = steps*
```

Each step involves the making of a joint between a beam end and a mushroom end. The five steps represent the five basic things which can be done during the course of an assembly. Each end contains both the number of the piece it belongs to and the end involved. For example, beam_end(beam3,be2) represents be2 of beam3.

Note that in Chapter 3 there were only four basic steps identified. Here, the two steps cluster_to_attached_beam(end,end) and cluster_to_racked_beam(end,end) are two variations of the Chapter 3 step of attaching a free cluster to a fixed beam. In the coding of ROBIN, it was found to be useful to make this distinction due to the different ways graphics generation is done for the structure and the parts racks.

The first step is beam_to_attached_cluster(_,_) and involves taking a beam from the rack and connecting its specified beam end to the specified mushroom end of a cluster which is already attached to the structure. Similarly, cluster_to_attached_beam(_,_) involves removing a cluster from the rack and attaching it to a beam which is already part of the structure.

The step cluster_to_racked_beam_end(_,_) takes a cluster from the cluster rack and then attaches it to a beam which is still in the beam rack. The step beam_with_cluster_to_attached_cluster(_,_) removes a beam from the rack which already has a cluster attached to it, and then attaches the beam to the mushroom end of another cluster which is attached to the structure.

The last step triangle(_,_) does not involve attaching any pieces to the structure, rather it involves completing the last connection in a triangle of beams and clusters.

6.1.2 Database Storage of Plans

In the database, two predicates are declared for storing sequences of assembly steps. They are:

```
completed(sequence)
proposed(sequence)
```

The predicate completed(sequence) stores the sequence of steps which has been completed so far. This predicate is mainly used by the graphics refresh routines for refreshing the graphics display when the database is restored.

The predicate proposed(sequence) stores a sequence of steps which, if executed, will complete the structure from the point that it is currently at. The proposed sequence is generated by the planning algorithm and is then used to suggest steps to the operator one at a time.

6.2 Heuristics for Optimum Assembly

There are two main heuristics which when used, limit the search space of possible assembly sequences so that an acceptable assembly sequence is ensured.

The first heuristic is that a triangle involving the base cluster should be completed as soon as possible for the purpose of rigidizing the partially assembled structure. This is due to the fact that the joint between a beam end and a mushroom end is free to rotate about the length wise axis of the beam unless the cluster is part of a triangle. Until a triangle is completed, whatever structure is attached to a top cluster will be free to rotate around and cause difficulties. Figure 6.1 demonstrates this.

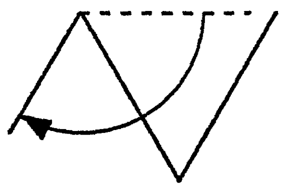


Figure 6.1 Structural Instability

The second heuristic is that whenever you attach a beam to the structure, you should attach a cluster to the beam first, as long as a cluster is available, and that this does not cause two clusters to be at the same node at once. This avoids needlessly carrying only clusters from the racks over to the structure. This assumes that the beam and cluster racks are located near each other as compared to their respective distances to the structure.

6.3 Main Loop of Planning Algorithm

When the operator selects plan rest is from the menu system, or after an alternate step is completed, the rule plan is called.

```
plan :-
        restore_old_database,
        proposed(X),
        retract(proposed(X)),
        assertz(proposed([])),
        plan_rest_of_assembly,
        restore_old_data_base.
```

The rule plan calls restore_old_database to restore the old database in order to delete the assembly step which is currently being suggested. The rule plan also deletes the current proposed plan by retracting proposed(X) and asserting proposed([]).

The rule plan then calls the rule plan_rest_of_assembly which produces a new proposed(sequence) in the database. Note that all database assertions and retractions done by plan_rest_of_assembly are temporary ones. Once plan_rest_of_assembly is done, the restore_old_database is once again called to delete all of the facts generated in planning out the rest of the assembly. This then returns the database back to the state in which only what has been completed of the structure is present, but with a new proposed plan present.

6.3.1 Planning the Rest of the Assembly

The rule plan_rest_of_assembly embodies the main loop of the planning algorithm. Due to details of Turbo Prolog's automatic backtracking, plan_rest_of_assembly had to be broken down into the two rules shown below:

```
plan_rest_of_assembly :-
        true_default_rules,
        get_next_piece(X),!,
        attach(X),!,
        plan_rest_of_assembly,!;
        true.
```

```
true_default_rules :-
        first_triangle,
        complete_any_triangles,
        beam_with_cluster,
        complete_any_triangles.
```

The rule plan_rest_of_assembly is a recursive rule which repeatedly gets the next piece and attaches it to the structure, until it runs out of pieces, fails, and then returns true. The rule true_default_rules always returns true and takes care of completing any triangles which are made possible by the attachment of a piece, and also embodies the two heuristics of completing a triangle as soon as possible and attaching beams with clusters already attached whenever possible.

Plan_rest_of_assembly can still successfully plan out a non-optimal sequence if true_default_rules was replaced by complete_any_triangles. This version of the rule will be called plan_rest_of_assembly_old.

```
plan_rest_of_assembly_old :-
        complete_any_triangles,
        get_next_piece(X),!,
        attach(X),!,
        plan_rest_of_assembly_old,!;
        true.
```

This form is instructive for understanding the algorithm, and was the original form of the rule used before the two heuristics were incorporated. Its operation will be explained first.

Again, in each call, complete_any_triangles checks to see whether or not the geometry is correct for completing a triangle. If a triangle is one step away from completion, then complete_any_triangles does so and adds triangle(_,_) to the proposed sequence, and the rule returns true. If no triangle can be completed, the rule still returns true. Section 6.4.2 describes how the geometry is checked to see whether or not it is possible to complete a triangle in the structure.

Next, get_next_piece(X) gets a cluster or a beam from the parts racks. The rule will get a beam as long as there are free mushroom ends on the structure to attach them to. If only free beam ends are available, then the rule will remove a cluster from the parts racks instead of a beam.

Finally, attach(X) takes whatever piece was obtained by get_next_piece(X) and attaches it to the structure. If the piece is a beam then the rule connects be1 of the beam to the first free mushroom end found in the database. Similarly, if the piece is a cluster, the rule connects me1 of the cluster to the first free beam end in the database.

Notice that this algorithm makes it impossible to try to put two beam or two clusters in the same place. This potential is present if a triangle is almost complete as in Figure 6.4. There is the possibility of trying to attach a fourth cluster to the free beam end, or a fourth beam to the free mushroom end. However, the use of complete_any_triangles in each recursive call eliminates both possibilities.

6.3.2 Adding Heuristics to Planning Algorithm

Once again, consider the rule plan_rest_of_assembly which uses true_default_rules instead of simply complete_any_triangles.

```
plan_rest_of_assembly :-
        true_default_rules,
        get_next_piece(X),!,
        attach(X),!,
        plan_rest_of_assembly,!;
        true.

true_default_rules :-
        first_triangle,
        complete_any_triangles,
        beam_with_cluster,
        complete_any_triangles.
```

Since true_default_rules still incorporates complete_any_triangles, the underlying function of plan_rest_of_assembly_old is preserved. However, two new rules are used.

The rule first_triangle checks whether or not a triangle involving the base cluster has been completed yet. If not, first_triangle calls the rule complete_first_triangle, which plans out the completion of the first triangle involving the base cluster. Once the first triangle is completed, complete_any_triangles is called in case the completion of the first triangle makes it possible to complete another triangle without attaching any other piece.

```
first_triangle :-
        check_for_triangle;
        complete_first_triangle.
```

Next, the rule beam_with_cluster is called. This rule checks and sees if it is possible to attach a beam with cluster to the structure before going ahead and attaching just a beam.

As will be described below, the rule complete_first_triangle also uses the heuristic of attaching a beam with a cluster if it is possible to do so.

6.3.3 Completing the First Triangle

As described above, the first triangle is completed by calling the rule complete_first_triangle:

```
complete_first_triangle :-
        check_for_triangle,!;
        step_towards_first_triangle,!,
        complete_first_triangle.
```

This rule recursively calls itself. On each call it first checks whether a triangle has been completed. If not, it calls the rule step_towards_first_triangle which adds one step to the sequence,bringing the first triangle one step closer to completion.

The rule step_towards_first_triangle employs a backwards chaining strategy which, when given any partial first triangle, will select the next step in the sequence of completing that first triangle:

```
step_towards_first_triangle :-
        make_last_connection;
        attach_third_beam;
        attach_third_cluster;
        attach_third_beam_with_cluster;
        attach_second_cluster;
        attach_second_beam_with_cluster;
        attach_first_beam_with_cluster.
```

Note that step_towards_first_triangle is composed of a series of rules which are OR'ed together (in Turbo Prolog, semicolons are OR's and commas are AND's). Prolog will start with the first rule and call each one until one of them turns out to be true. Each one of the functions checks whether or not the structure fits a certain geometry; if it does, the next step is done by that rule.

For example, make_last_connection checks to see whether or not a triangle is almost complete (requiring only the last connection to be made). If this is the case, then the connection is made and the step triangle(_,_) is added to the proposed sequence. If this is not the case, make_last_connection fails and the next rule is tried.

This next rule is attach_third_beam and works just like make_last_connection. The structure is checked for a geometry which involves two upright beams attached to the base cluster and a cluster at the top of each beam. If this is found, a third beam is attached and true is returned. On the next call of step_toward_first_triangle, make_last_connection will succeed and the triangle will be completed. If attach_third_beam fails, then the next rule is tried. In this manner, progressively simpler partially completed triangles are looked for until one is found.

Prolog therefore works down through the rules until some partial triangle geometry is found, even if it happens to be only the empty base cluster. Once a partial triangle geometry is found at some depth in the rules, then the next required step is done, and on successive calls, the rules succeed one step higher each call, doing the steps needed to complete the first triangle. Figure 6.2 shows all of the possible partial triangle geometries and how the rules propagate upward once one is found.

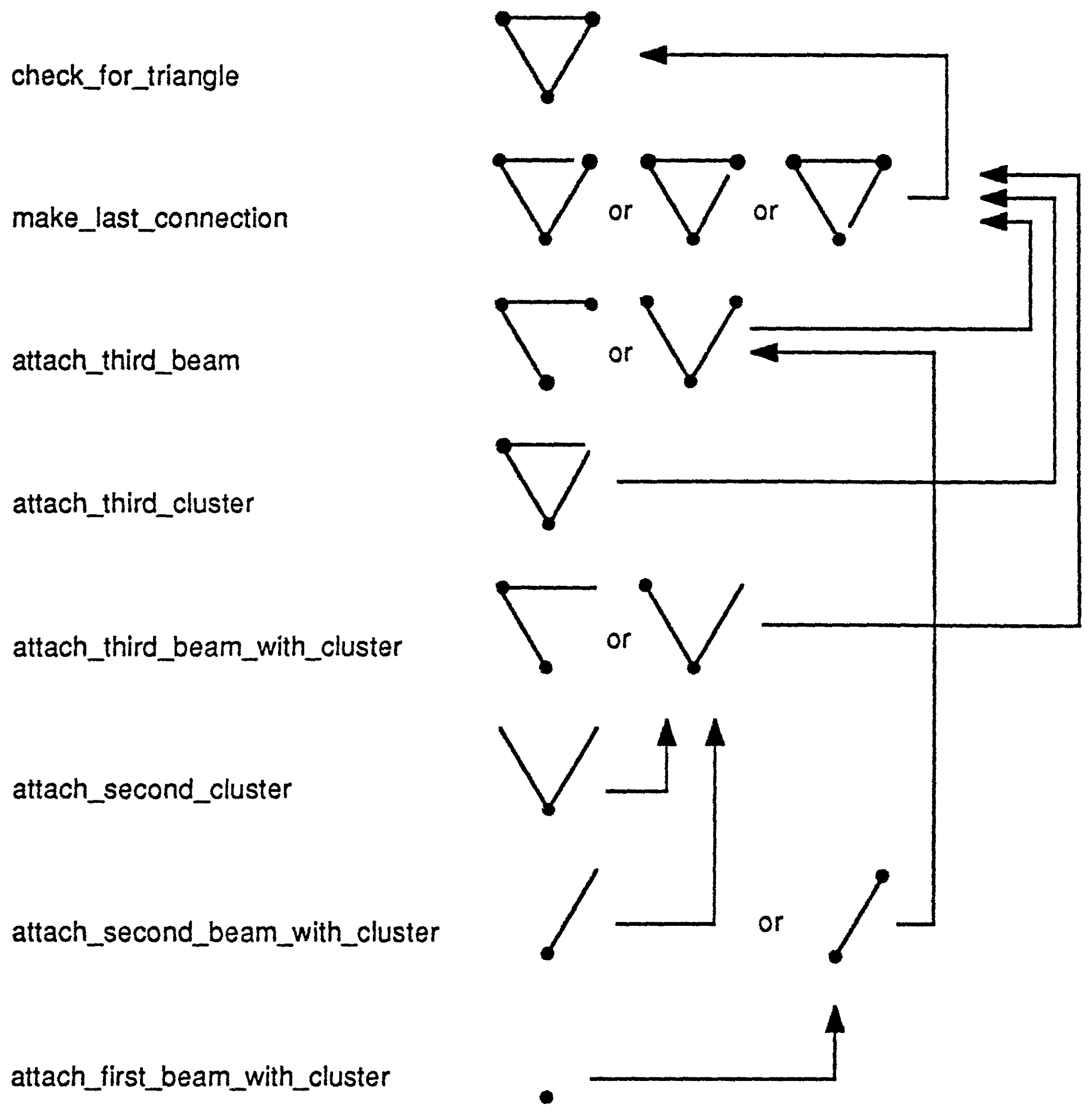


Figure 6.2 Propagation of First Triangle Rules

6.4 Geometrical Information Used by Planning Algorithm

To plan an assembly of the EASE structure ROBIN utilizes four areas of geometrical knowledge about the EASE structure and the pieces out of which it is built. They are:

1) Each beam has two ends. One end of a beam is beam end be1, the other end is be2.

2) Each cluster has three mushroom ends, and looking into the cluster opposite from the vertex, me2 is clockwise from me1, me3 is clockwise from me2, and me1 is clockwise from me3.

3) Three beams connected together with three clusters will form a triangle as long as the the proper mushroom ends are used on those clusters.

4) In the EASE structure, attaching six beams and four clusters together so that no ends are free results in a tetrahedron.

The above four facts are all the basic concepts needed to plan out the assembly of the EASE structure. This is inherent in the fact that the EASE structure is a tetrahedron, which is the simplest three dimensional structural element. No three dimensional coordinate information is needed.

6.4.1 End Geometries

The knowledge of 1) and 2) above was incorporated in a set of simple clauses describing the geometric relationships of ends on pieces. These are shown in Figure 6.3.

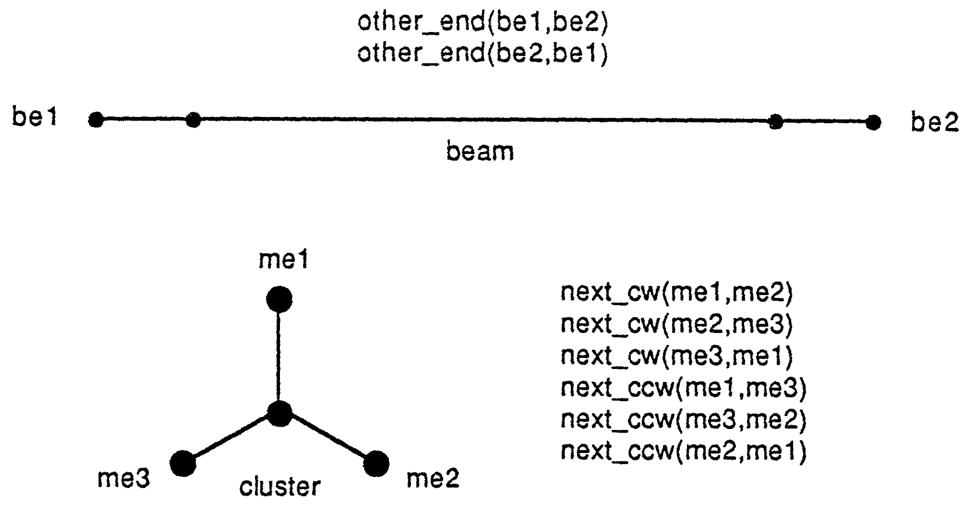


Figure 6.3 Geometric Relationships of Ends

These clauses are then used in such a way so that the knowledge in 3) can be used. These relationships are used in rules which trace through the structure looking for triangles, and parts of triangles as described in Section 6.3.3. This is best explained by an example.

6.4.2 Checking Geometries

All of the rules in step_toward_first_triangle and complete_any_triangles use geometry checking rules which attempt to trace through the structure looking for certain geometries.

Assume that the planning algorithm is checking to see if there are any triangles which require only the connection of the last joint as in Figure 6.4. The planner would then use a rule which, when given a free beam end and a free mushroom end, will check if the geometry is okay for the two to be connected together in a triangle.

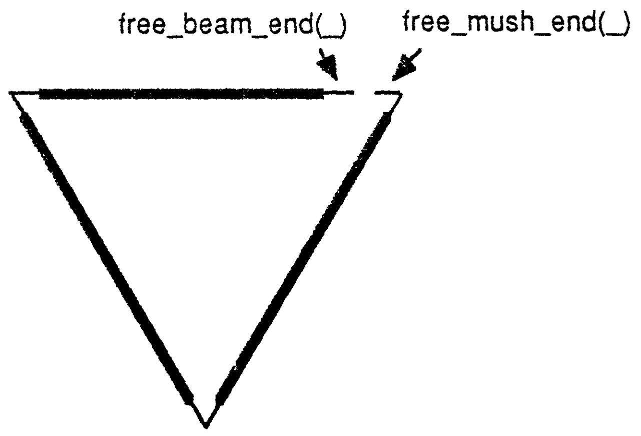


Figure 6.4 Geometry Needed for Completing a Triangle

The rule complete_all_triangles uses the rule geometry_ok(X,Y) to check for such a triangle:

```
geometry_ok(X,Y) :-
        equal(X,beam_end(B1,E1B1)),
        equal(Y,mush_end(C1,M1C1)),
        other_end(E1B1.E2B2),
        connected(beam_end(B1,E2B1),mush_end(C2,M1C2)),
        next_ccw(M1C2,M2C2),
        connected(beam_end(B2,E1B2),mush_end(C2,M2C2)),
        other_end(E1B2,E2B2)
        connected(beam_end(B2,E2B2),mush_end(C3,M1C3)),
        next_ccw(M1C3,M2C3),
        connected(beam_end(B3,E1B3),mush_end(C1,M2C1)),
        next_ccw(M2C1,M1C1);

        (same thing but with next_cw instead next_ccw).
```

The geometry checking rule will then attempt to trace through all possible paths which begin at the free beam end; it will signal success if a path is found which ends with the given free mushroom end, and which went through the proper geometric path for a triangle.

The rule begins by first checking the other end of the beam to which the free beam end belongs. This is done by first looking at the free beam end, since it contains both the beam and end in its object: beam_end(beam3,be2) for example. Then, using the relation other_end(be2,be1) it knows that the other beam end is beam_end(beam3,be1). Next, the rule checks if that beam end is connected to a mushroom end. If it is, then the rule proceeds to check if any beam end is connected to the next counter-clockwise (next_ccw(_,_)) mushroom end of that first cluster. This tracing proceeds until either a solution is found or all possible paths fail.

Chapter 7

ROBIN's User Interface

7.1 Elements of Interface

ROBIN was run on an IBM Personal Computer AT. The human operator communicated to ROBIN through four keys on the computer keyboard, and ROBIN communicated to the human operator through the computer's graphics display.

The keyboard used was a standard IBM Personal Computer AT keyboard. The four keys used were completely software definable, but function keys F1-F4 were used for simplicity and ease of recognition.

The graphics display used was a color IBM Professional Graphics Display, operating in Color Graphics Adapter Mode emulation mode. The display driver software was capable of medium resolution of 320 columns and 200 rows, in four colors. The four colors selected were black for the background and white, blue (cyan), and red (magenta) for the foreground colors.

Turbo Prolog facilitates the division of the display into separate windows. Figure 7.1 shows how the screen is divided into the four windows used by ROBIN.

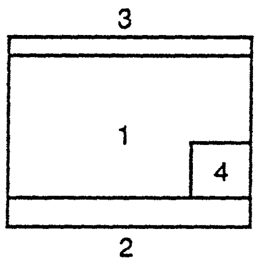


Figure 7.1 Four Windows Used in ROBIN

Windows 1 and 4 are used for graphics. Window 1 is used to draw the structure as it is built, and window 4 is used to draw the parts racks. Windows 2 and 3 are used for text. Window 2 displays the menu options associated with the four keys F1-F4. Window 3 is the message window, and describes what step is being suggested and by whom, along with other messages concerning what ROBIN is doing, such as planning or initializing.

7.2 General Strategy of Operation of User Interface

This section describes the strategy of operation of the human computer interface between ROBIN and the human operator.

7.2.1 Information Exchange

ROBIN provides the following information to the human operator:

1) A graphic display of the structure built so far to help the operator when he cannot view the actual structure.
2) The next step to be done in the assembly, including which piece or pieces are to be attached, and where they are to be attached to the structure .
3) Other legal options of what the operator can do as an alternate step to the one being suggested by ROBIN.

The human operator provides the following information to ROBIN:

1) Whether or not the proposed step was completed.
2) If not, what was done instead.

7.2.2 ROBIN's Main Loop

The main loop of ROBIN is outlined in Figure 7.2. In step 1), the structure built so far is drawn, along with the next suggested step. In step 2) the operator either accepts the suggested step because it was completed, or he completed an alternate step and needs to describe it to ROBIN. In step 3 the operator inputs the alternate step completed, after which ROBIN adds the step to its internal model of the structure and replans the rest of the assembly from that point on. The menu system is used by the operator to communicate information to ROBIN in steps 2) and 3).

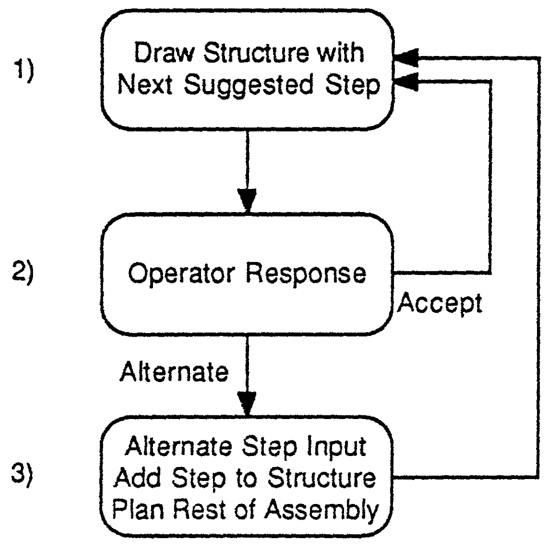


Figure 7.2 ROBIN's Main Loop

7.3 Menu System

Figure 7.3 shows a flow chart of the menu system which allows the operator to communicate to ROBIN.

The program is started with the Run command. This puts the program in the Top Menu, used for initializing ROBIN. The operator first hits F1 to select Initialize . This adds the proper facts to the database concerning the location of the base cluster and the state and location of the parts racks. Hitting F4 exits ROBIN while F3 has no effect. Hitting F2 in the Top Menu drops the operator down into the Main Menu.

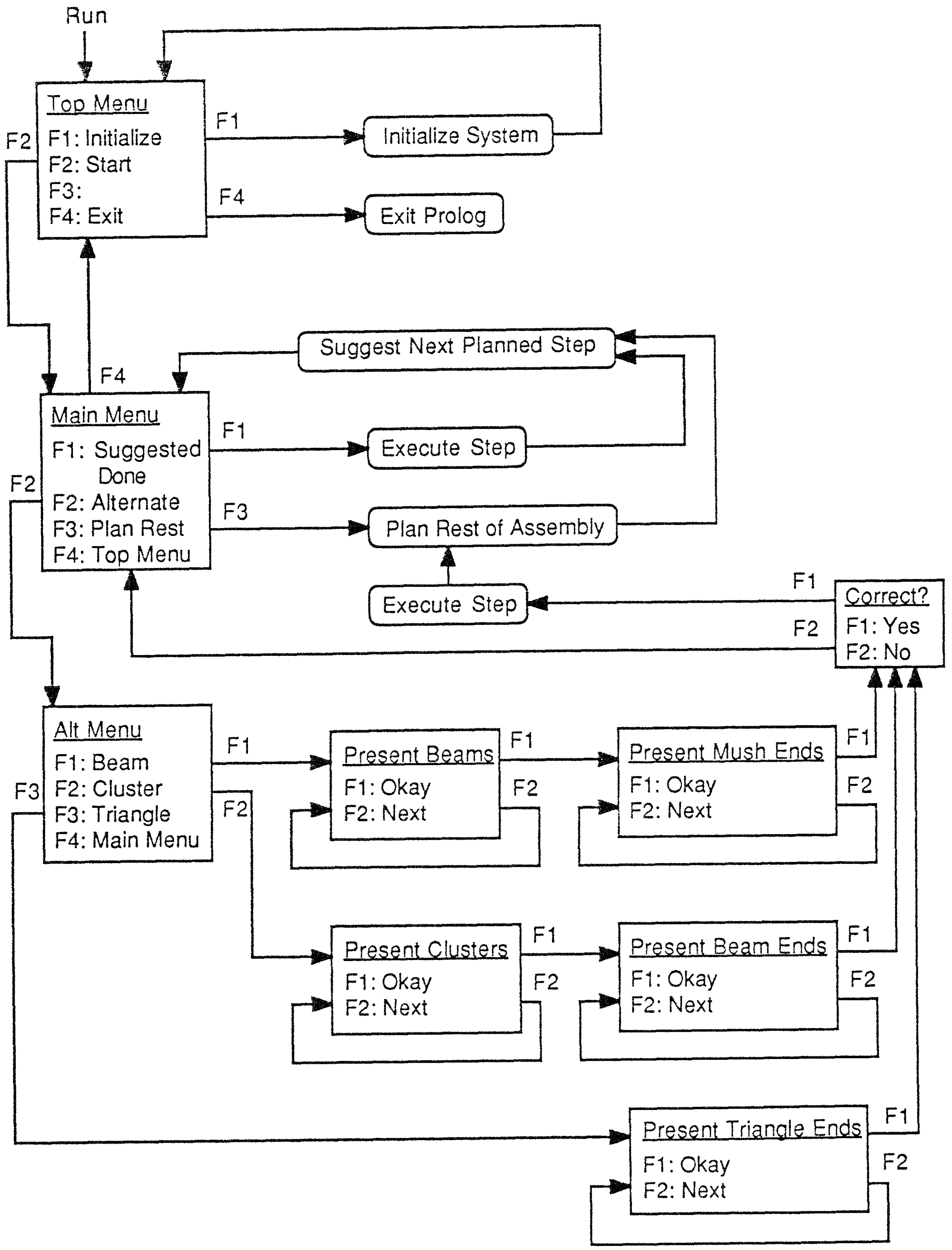


Figure 7.3 The Menu System

In the Main Menu, with the structure partially assembled, ROBIN has a plan for completing the structure from the current state, stored as a sequence of assembly steps. ROBIN displays the structure built so far, plus the next step to be completed in its plan. The already completed portion of the structure will be drawn in red (clusters) and white (beams). The next step suggested by ROBIN will be presented by highlighting in blue the piece(s) in the parts rack which are involved, and drawing in blue how the step will change the current structure. A description of the suggested next step will be written in window 3.

This tells the operator what the next step is, what piece(s) is involved, and how it should be attached to the structure. This also tells the operator where the step should take place on the structure, and therefore where BAT should dock to the structure.

In the Main Menu, the operator's first option is to hit F1 to choose Suggested Done, signalling that the suggested step has been completed. If this is done, ROBIN redraws the structure in red and white with the suggested step now included in the structure, plus the next suggested step from the plan drawn in blue.

The operator's second option in the main menu is to choose Alternate by hitting F2. This puts the operator into the alternate step menu. Here the operator selects whether he wants to attach a beam, attach a cluster, or complete a triangle.

As an example, assume that the operator wants to attach a beam. ROBIN will redraw in blue the first available beam in the rack. If no beams are available for attaching, ROBIN will so inform the operator. If the beam drawn in blue is suitable, then the operator can hit F1 to select that beam (Okay). Otherwise, hitting F2 will cause the next available beam to be highlighted in blue instead (Next). The operator can keep on running through the possible beams (with automatic wrap around) to select from until the one he wants is highlighted and he hits F1 to select it.

Once the desired beam is selected, the operator can in the same way run through the possible mushroom ends on the structure to which the beam can be attached. Once both the beam and the mushroom end attachment point are selected, ROBIN draws the piece attached to the structure highlighted in blue, writes the operator suggested step at the top of the screen, and queries the operator if this is correct.

If the operator responds Yes (F1) then the step is added to the assembled structure, the rest of the assembly is planned out, and the structure so far and the next suggested step in that new plan are displayed. Otherwise, hitting F2 (No) returns to the Main Menu.

The process is similar for attaching a cluster as an alternate step. For completing a triangle, only the possible pairs of ends are run through.

Chapter 8

ROBIN's Performance

8.1 Testing of ROBIN

Although ROBIN was completed and ready to be integrated into ICS (by simply connecting video to one of ICS large monitors and positioning the keyboard within the operator's reach) the author completed his graduate studies before another set of test at the NASA Marshall NBS were possible. Therefore, the author was unable to test ROBIN under actual structural assembly conditions.

What is presented in this chapter are actual screen copies from the display which demonstrate that ROBIN met all of its design criteria. This also should help the reader to better understand the operator interface.

8.2 Successful Assembly Planning by ROBIN

Figure 8.1 shows the sequence of display screens which is generated if the operator lets ROBIN plan out the entire assembly and then performs all of the steps suggested by ROBIN. Figure 8.1 is composed of 17 separate screen displays which are numbered as Sections 8.1.1 through 8.1.17. The screen dumps are distorted slightly in the vertical direction, causing the EASE structure to appear wider than it is with respect to its height.

In these screen displays, three colors (white, red, and blue) are shown as three different types of line shadings. White lines appear as solid black lines or lines made of large dots. Red lines appear as lines made up of thin vertical line segments. Blue lines appear as lines made up of small black dots. Lines often will be described as being drawn in a particular color, which will actually refer to the type of black line that they are drawn with in the screen dumps. Unfortunately, poor reproductions of this thesis may not show the needed detail to discern the different line types.

The following sections describe in detail the individual sections of Figure 8.1.

8.1.1: The top menu has just been entered by running the program. All of the text is in white. The operator hits the F1 key to initialize the system.

8.1.2: The system is initialized with the parts rack full of parts and the base cluster in place. The beams are colored white and clusters are colored red. The operator hits F2 to start the main menu.

8.1.3: The main menu has just been entered and will not be left during the rest of this example. The operator hits F3 to have ROBIN plan the rest of the assembly.

8.1.4: This screen is displayed for the 2 seconds it takes ROBIN to plan the assembly.

8.1.5: ROBIN here suggests that the next step to be completed is attaching a top cluster to a racked beam. The left most cluster in the rack is colored blue, indicating that it is the cluster selected to be attached. The same cluster is redrawn in blue attached to the selected racked beam. Thus the selected step is drawn in blue. The operator hits F1 to signal that the suggested step has been carried out. For this assembly, the operator will hit F1 at each step, so it will not be mentioned again.

8.1.6: ROBIN suggests attaching the racked beam with cluster attached (both drawn in blue in the rack) to the base cluster. For the rest of this example, "ROBIN suggests attaching" will simply be written as "Attach".

8.1.7: Attach the second top cluster to a racked beam. Notice that ROBIN is using the heuristic of attaching a cluster to a beam before attaching the beam to the cluster.

8.1.8: Attach the racked beam with cluster to the base cluster.

8.1.9: Attach a beam (without a cluster this time) to one of the top clusters. Notice that ROBIN has stepped efficiently towards completing a triangle involving the base cluster as soon as possible. Also notice the space left at the unconnected end of the beam. This helps the operator to better see which cluster the beam is attached to.

8.1.10: Complete the first triangle by making the triangle connection. Both the beam end and mushroom end (upper right vertex) involved are drawn in blue.

8.1.11: Attach a top cluster to a racked beam (again, using the second heuristic).

8.1.12: Attach the beam with cluster to the base cluster.

8.1.13: Attach a beam to one of the top clusters.

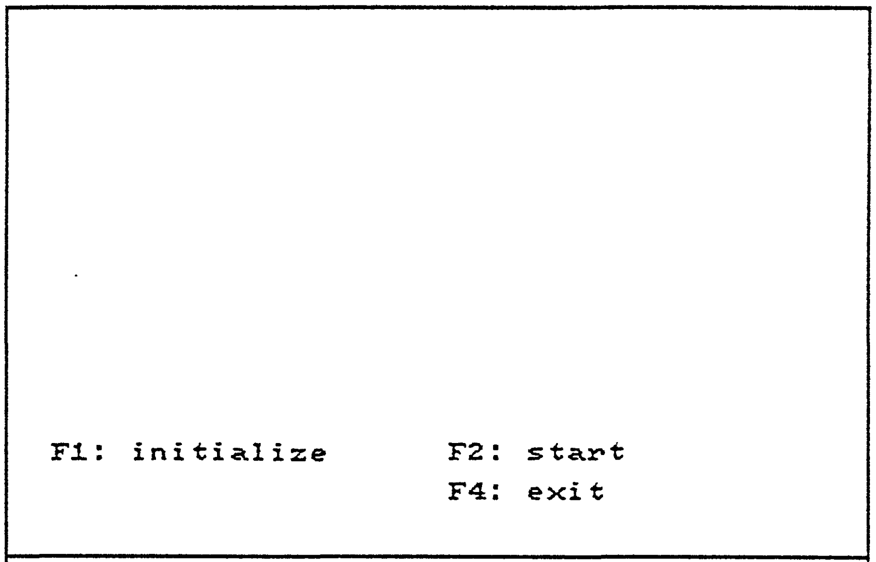
8.1.14: Complete the triangle.

8.1.15: Attach the last beam to one of the top clusters.

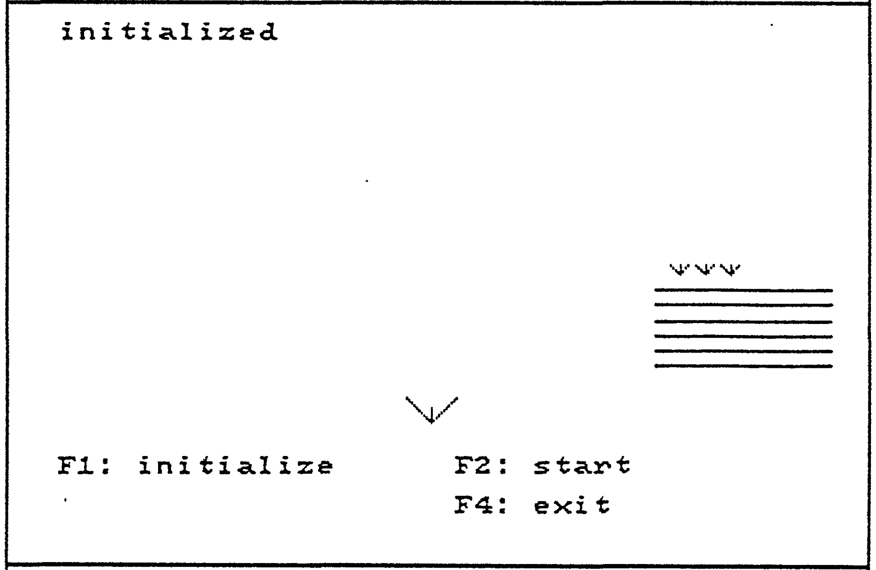
8.1.16: Complete the last triangle.

8.1.17: Screen showing the completed structure.

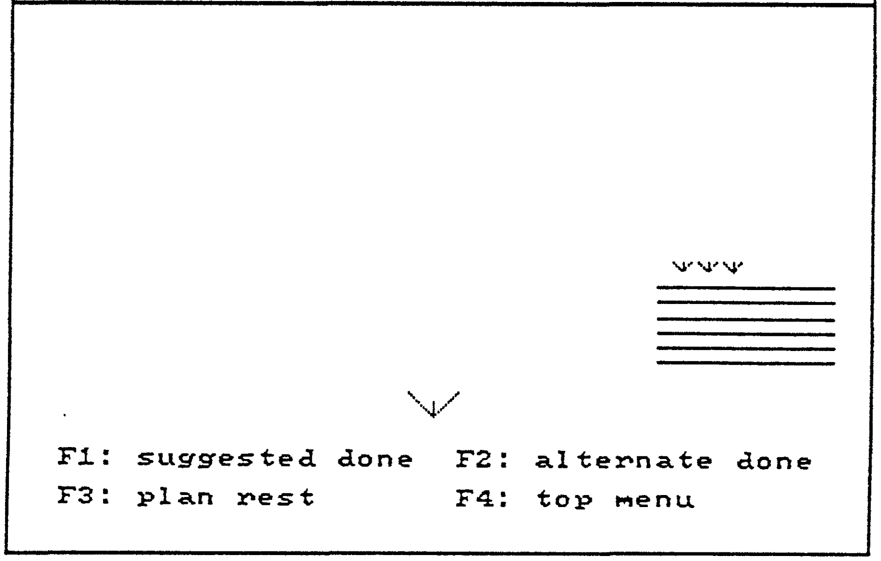
This demonstrates that ROBIN is able to plan out the assembly of the EASE structure, using both heuristics of first completing a triangle involving the base cluster, and when possible, attaching clusters to racked beams before attaching the beams to the structure.



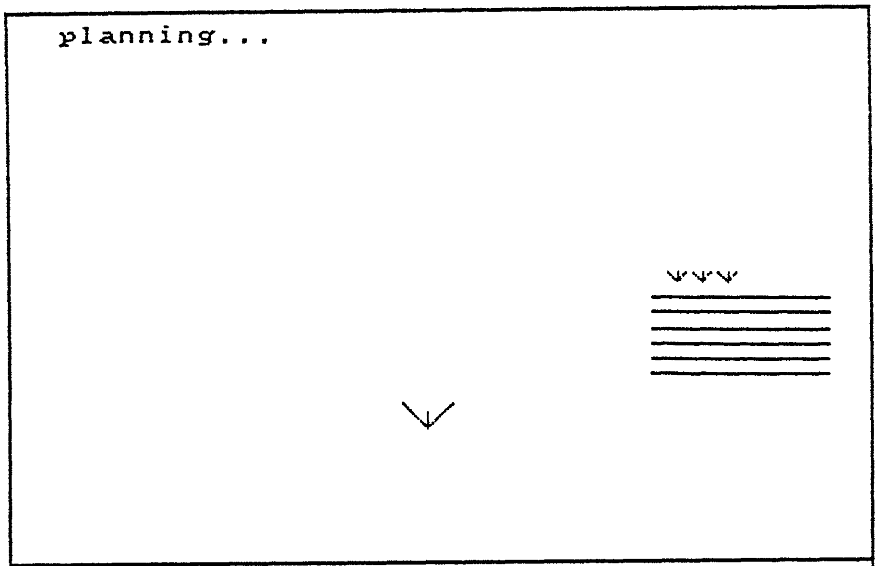
8.1.1


initialized

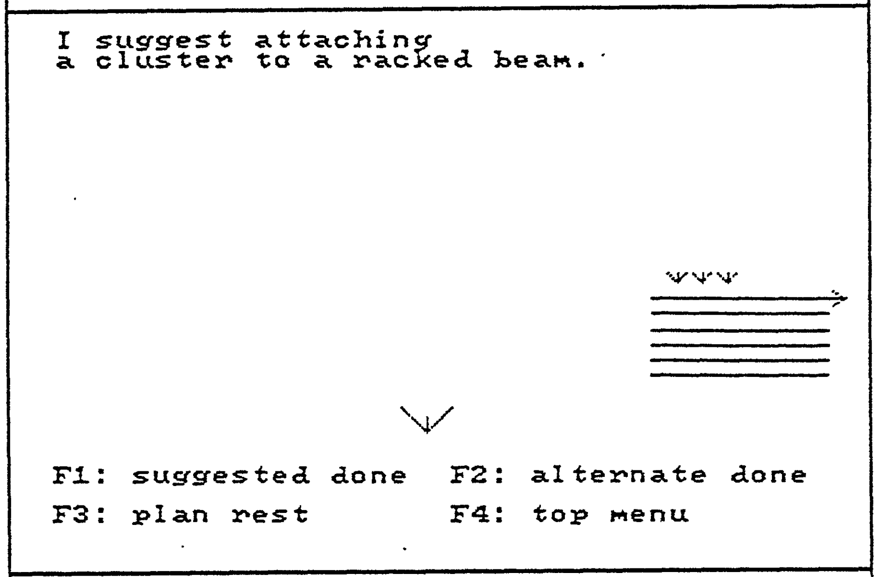
8.1.2



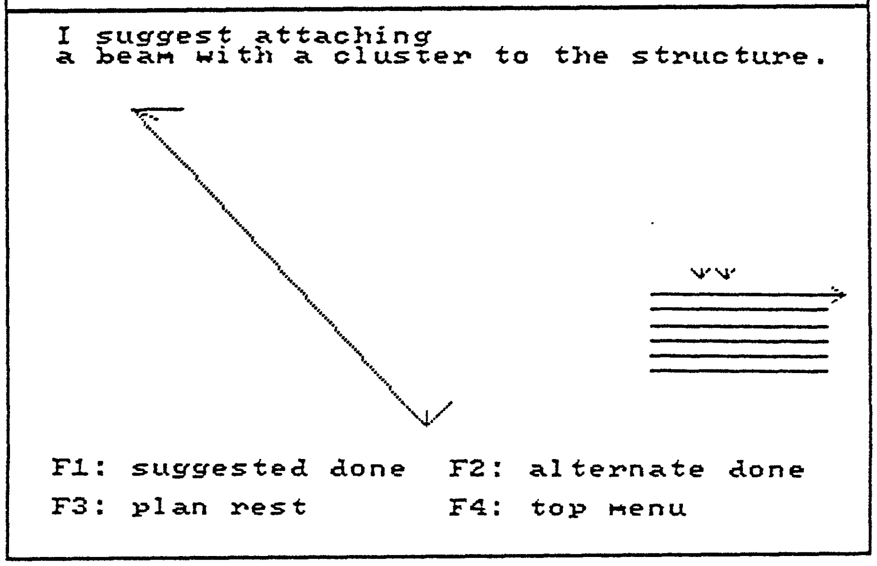
8.1.3


planning...

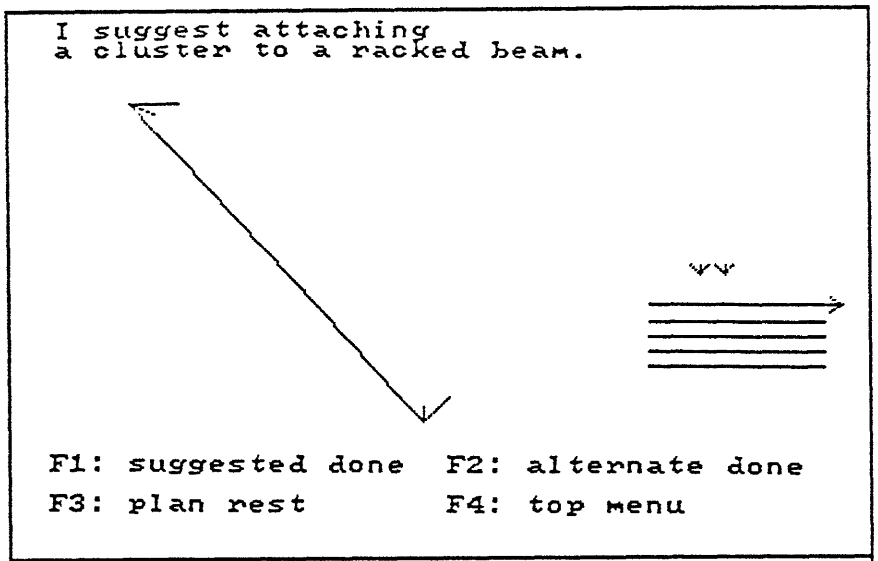
8.1.4


I suggest attaching
a cluster to a racked beam.

8.1.5


I suggest attaching
a beam with a cluster to the structure.

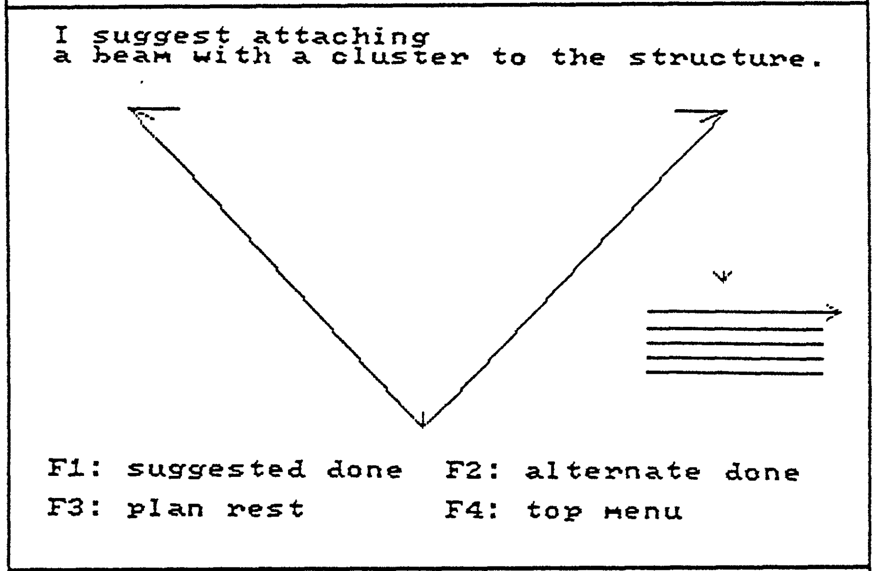
8.1.6



I suggest attaching
a cluster to a racked beam.

F1: suggested done F2: alternate done
F3: plan rest F4: top menu

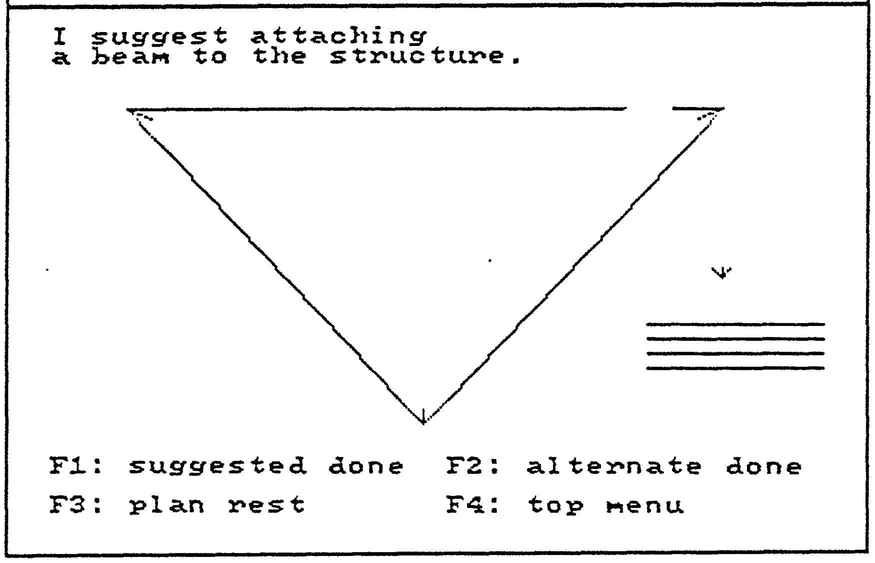
8.1.7



I suggest attaching
a beam with a cluster to the structure.

F1: suggested done F2: alternate done
F3: plan rest F4: top menu

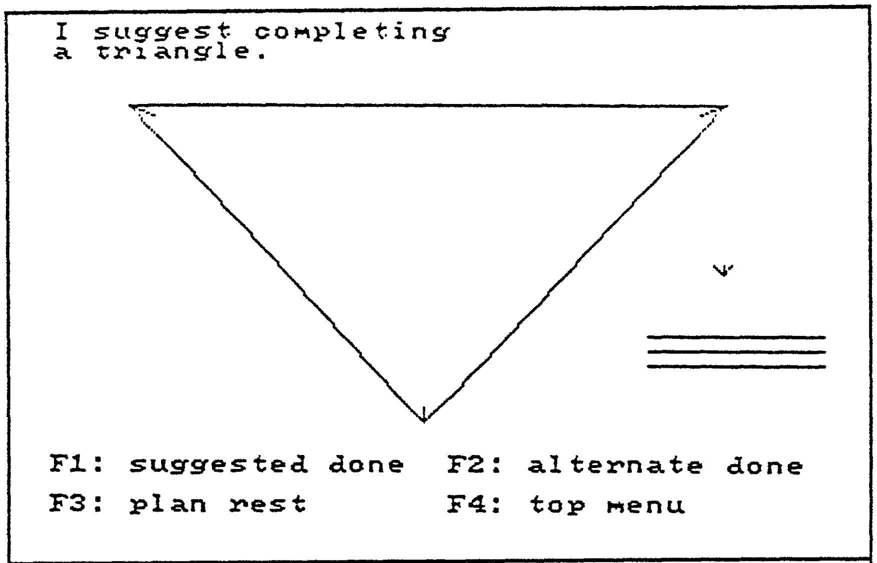
8.1.8



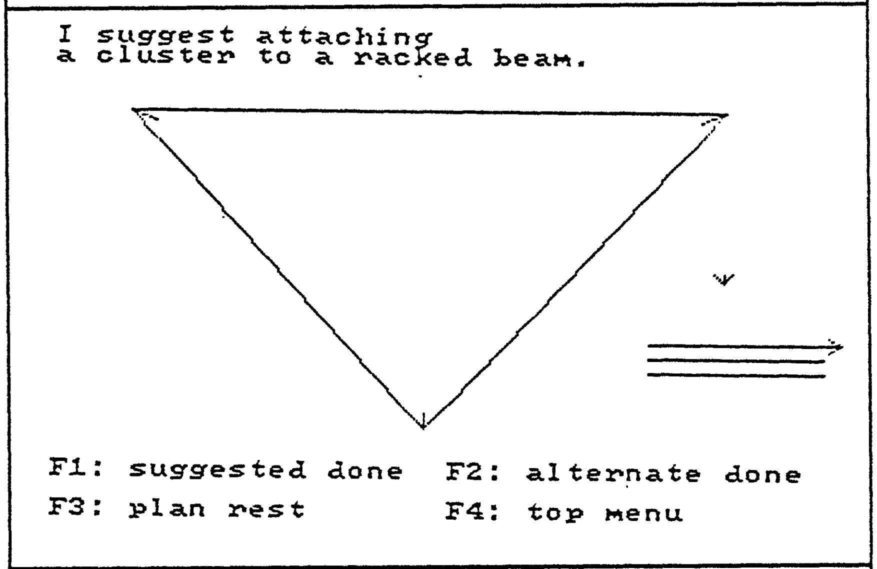
I suggest attaching
a beam to the structure.

F1: suggested done F2: alternate done
F3: plan rest F4: top menu

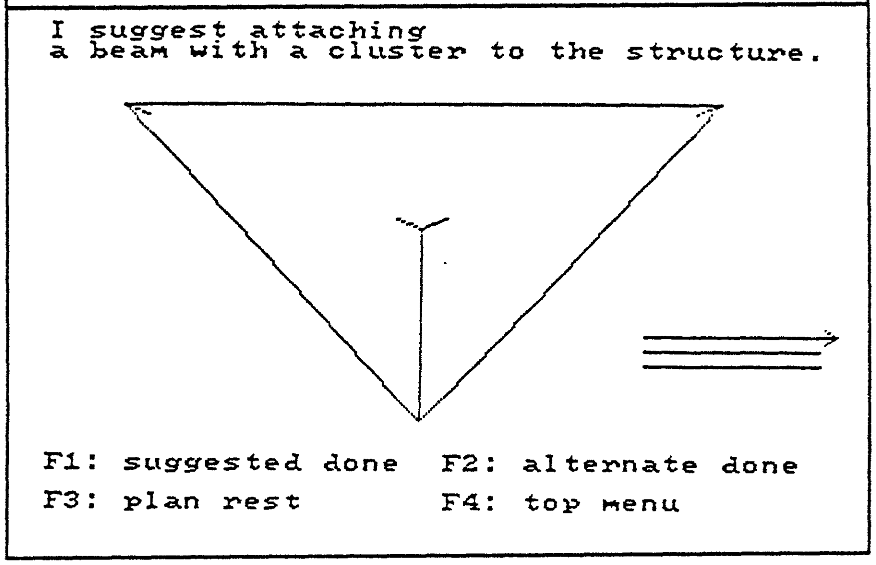
8.1.9


I suggest completing
a triangle.

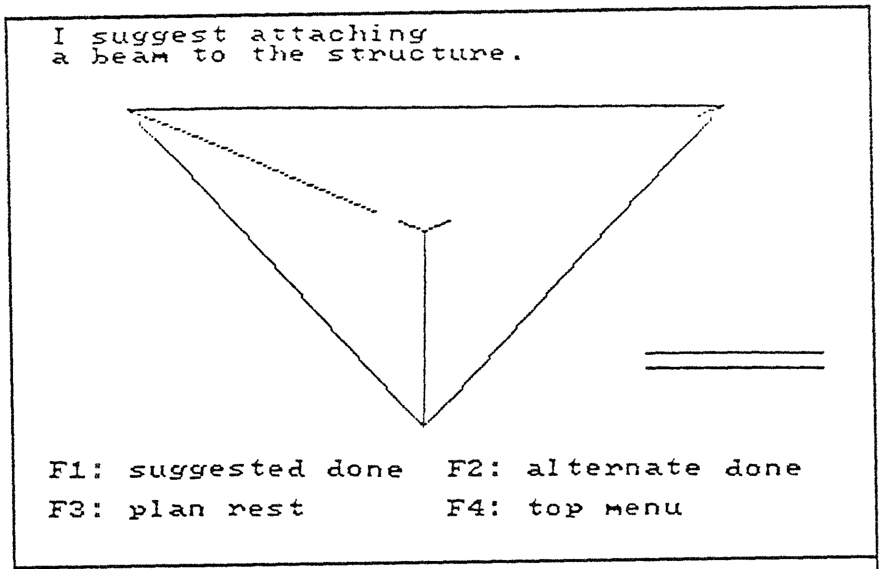
8.1.10


I suggest attaching
a cluster to a racked beam.

8.1.11

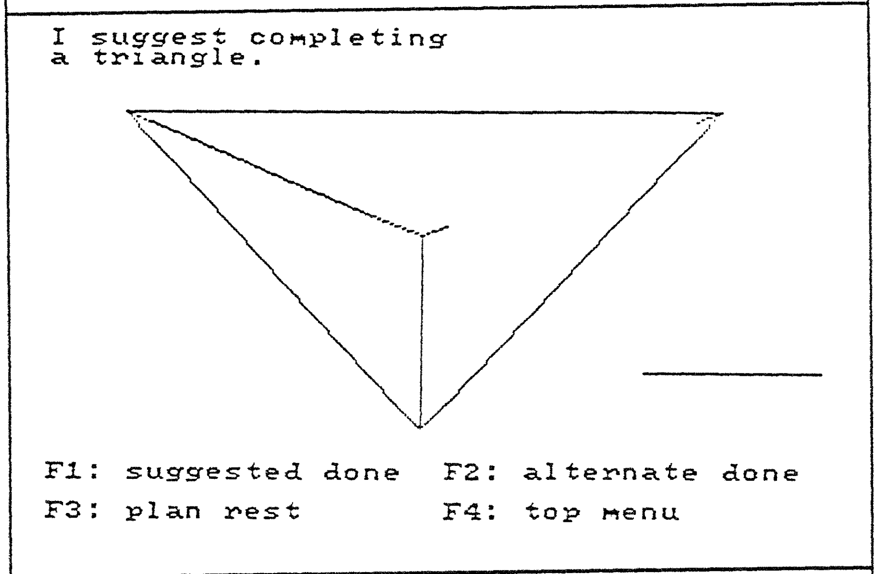

I suggest attaching
a beam with a cluster to the structure.

8.1.12


I suggest attaching
a beam to the structure.

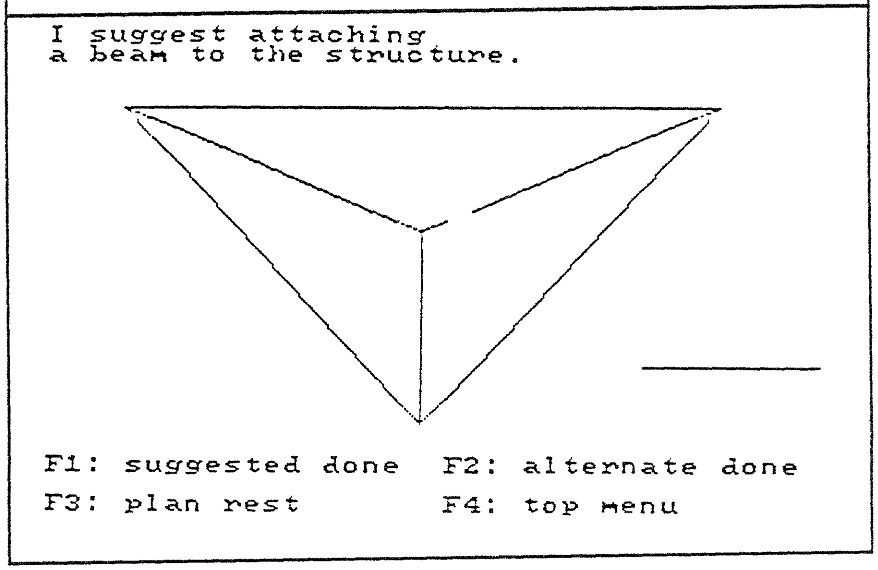
F1: suggested done F2: alternate done
F3: plan rest F4: top menu

8.1.13


I suggest completing
a triangle.

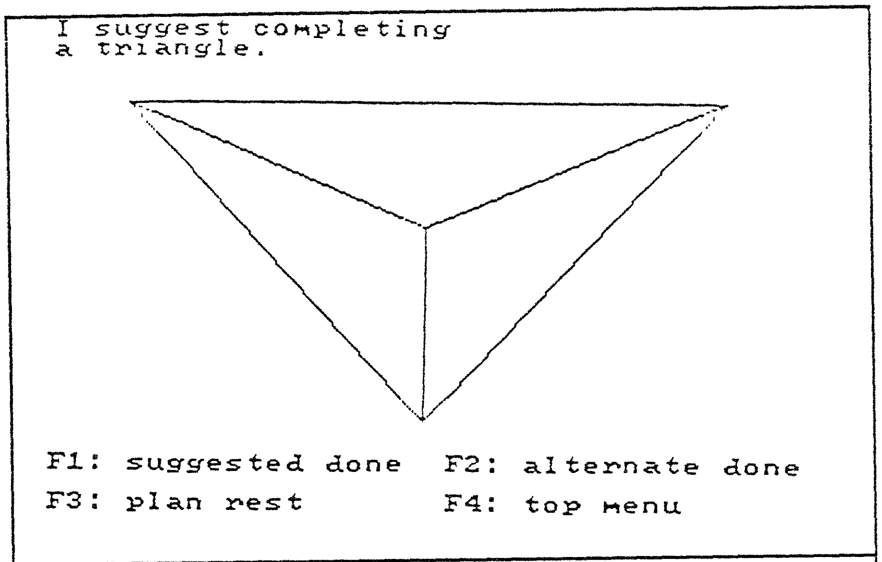
F1: suggested done F2: alternate done
F3: plan rest F4: top menu

8.1.14


I suggest attaching
a beam to the structure.

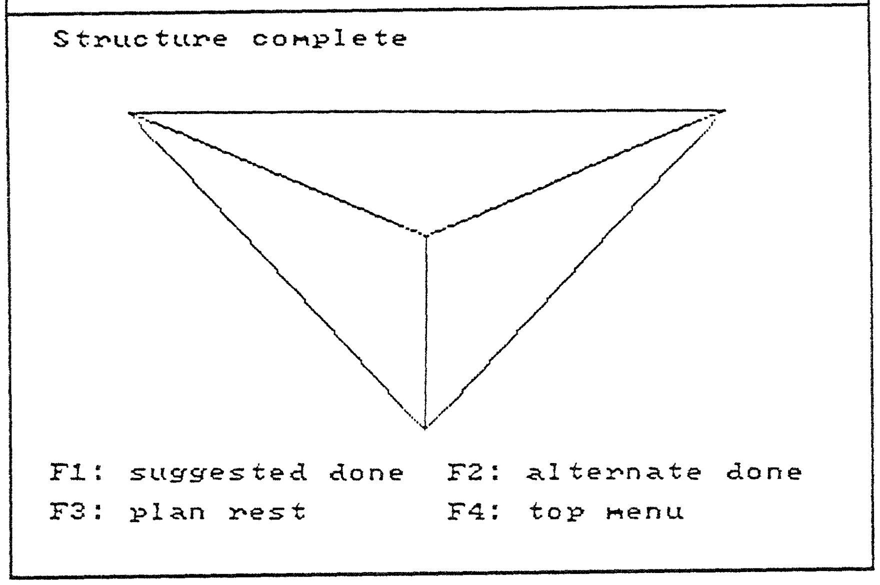
F1: suggested done F2: alternate done
F3: plan rest F4: top menu

8.1.15



I suggest completing
a triangle.

8.1.16



Structure complete

8.1.17

8.3 Completing the First Triangle

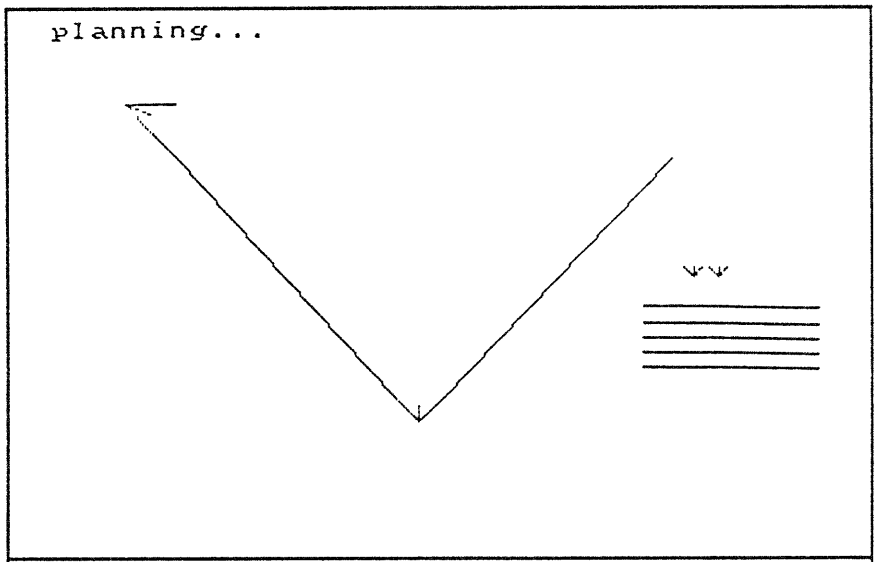
Figure 8.1 showed one example of how ROBIN completed a triangle involving the base cluster as soon as possible. This was done by attaching two upright beams with clusters to the base cluster, and then attaching a cross beam between them. To demonstrate that ROBIN can handle other partial structure situations, and still complete a triangle as soon as possible, several examples are presented in this section. In all of these examples, the operator has built a partial structure and asked ROBIN to plan the rest of the assembly.

In Figure 8.2 ROBIN is faced with a situation in which two beams are attached to the base cluster, but only one of them has a top cluster attached. This is shown in 8.2.1 in which ROBIN is planning the rest of the assembly. In 8.2.2, ROBIN has finished planning, and suggests that the next move should be to attach a cluster to a racked beam. The operator then hits F1 to agree, and in 8.2.3 ROBIN suggests that the next move should be to attach the beam with a cluster to the already attached top cluster. This leaves only one last triangle connection which is suggested in 8.2.4. Notice that ROBIN also used the second heuristic that a cluster should be attached to a racked beam, rather than to transport the cluster alone over to the structure for attachment.

No example so far has shown ROBIN attach a cluster to a beam which is already part of the structure. This occurs in the situation shown in Figure 8.3. In 8.3.1 ROBIN is planning the completion of the structure after being presented with a triangle which is missing a cluster. Here, it would be a mistake to attach a cluster to a racked beam, or to do anything other than to attach a cluster to one of the two free beam ends on the structure. In 8.3.2 ROBIN has made the correct choice, and suggests that the operator attach a cluster to the structure. In 8.3.3 ROBIN suggests completing the first triangle.

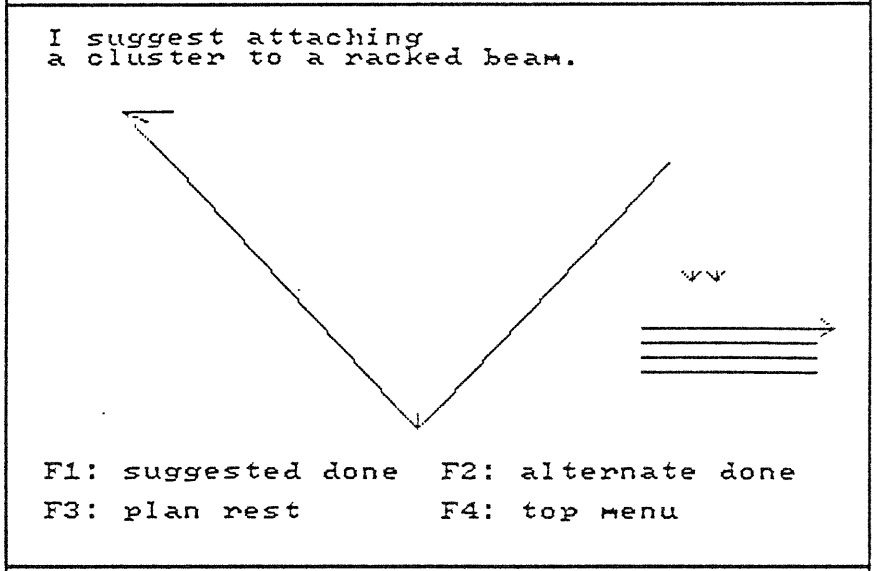
The partial structure presented to ROBIN in Figure 8.4 demonstrates that ROBIN does not always complete the back triangle first as has been seen so far, and that ROBIN can distinguish between the top triangle of EASE which does not involve the base cluster, and the three side triangles which do.

In 8.4.1 ROBIN is presented with a partial structure in which the top triangle is only one step away from being completed and no side triangles are complete. The attempt is to bait ROBIN into completing the top triangle instead of pursuing the completion of one of the side triangles involving the base cluster. In 8.4.2, ROBIN correctly analyzed the structure and suggests that the operator attach a beam to the base cluster, which brings a side triangle one step away from completion. In 8.4.3 ROBIN completes the left side triangle (thereby satisfying the first heuristic), and then finally finishes the top triangle in 8.4.4.



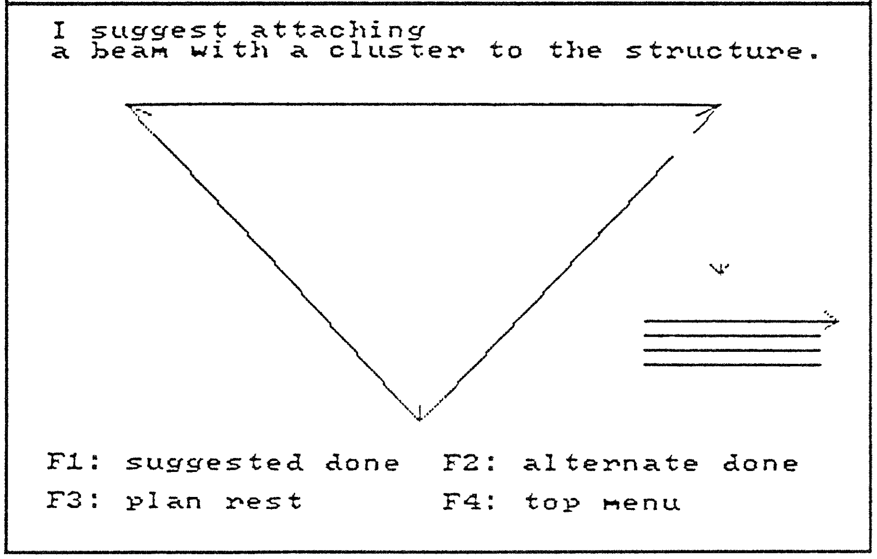
```
planning...
```

8.2.1



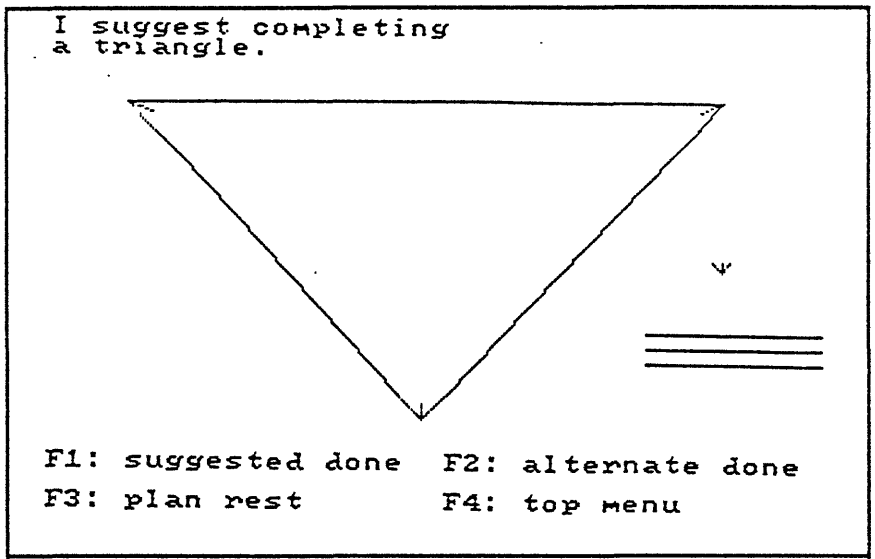
```
I suggest attaching
a cluster to a racked beam.

F1: suggested done   F2: alternate done
F3: plan rest        F4: top menu
```

8.2.2



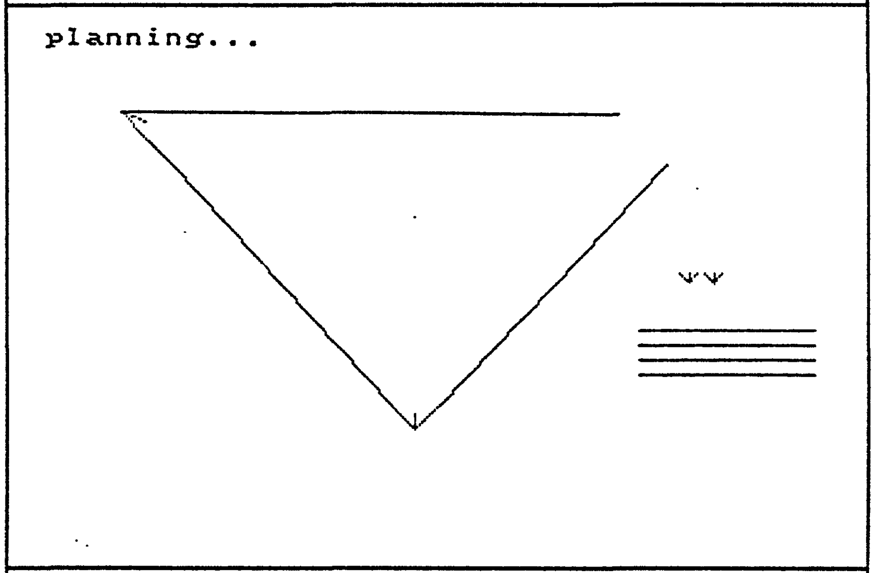
```
I suggest attaching
a beam with a cluster to the structure.

F1: suggested done   F2: alternate done
F3: plan rest        F4: top menu
```

8.2.3

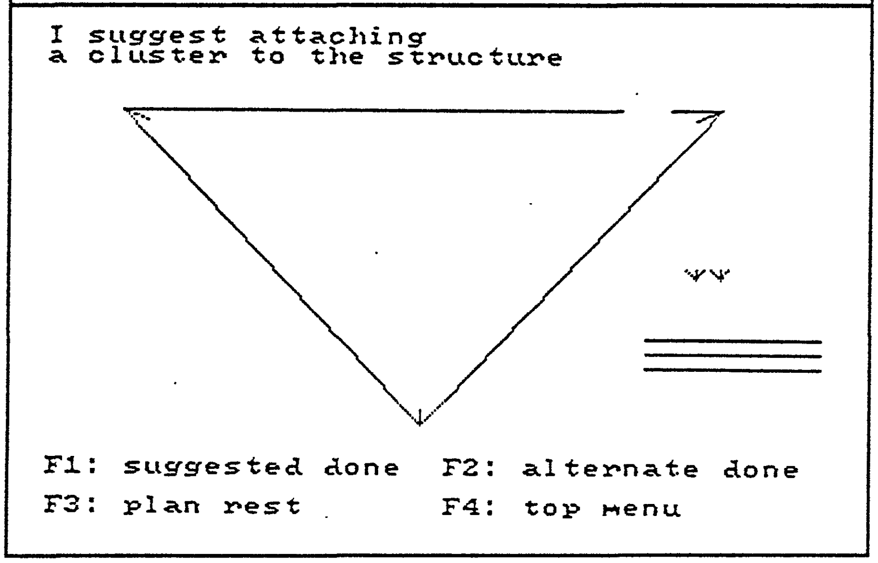

I suggest completing
a triangle.

F1: suggested done   F2: alternate done
F3: plan rest   F4: top menu

8.2.4

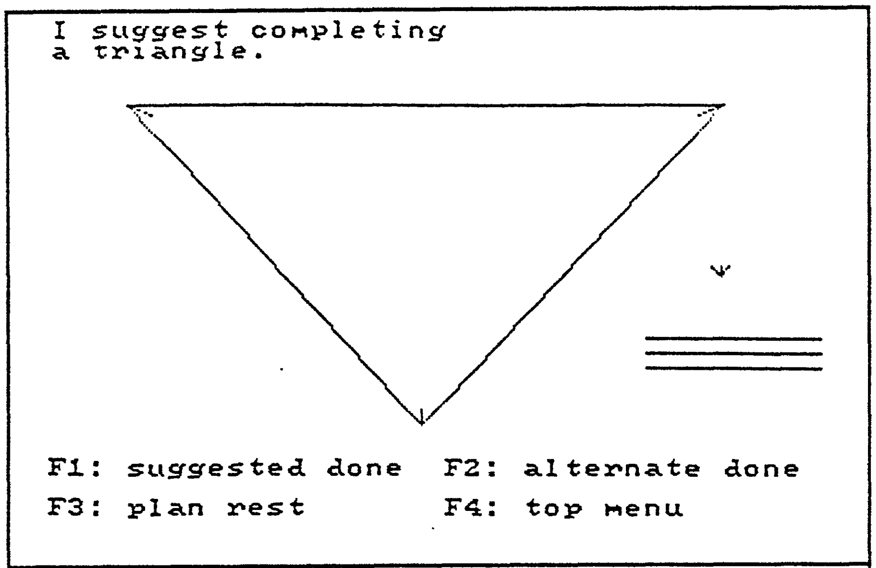

planning...

8.3.1


I suggest attaching
a cluster to the structure

F1: suggested done   F2: alternate done
F3: plan rest   F4: top menu

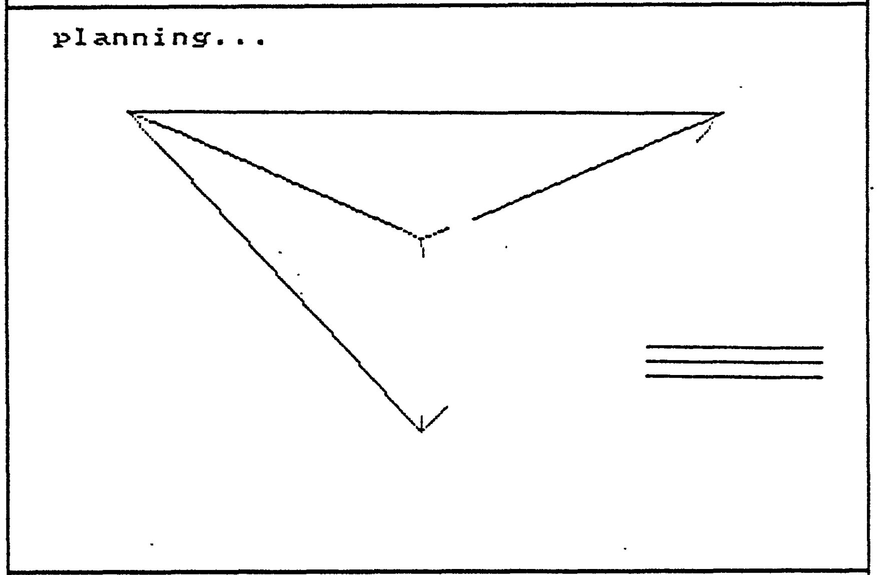
8.3.2



I suggest completing
a triangle.

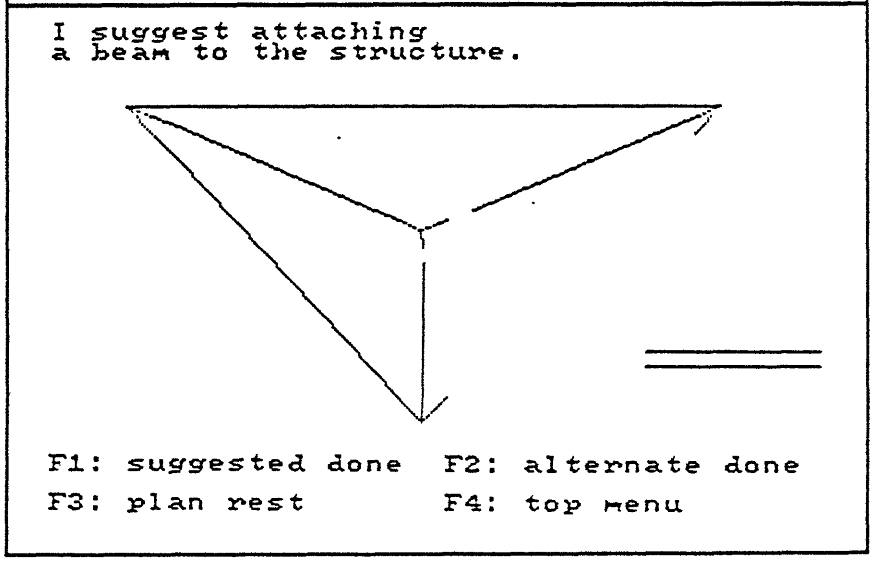
F1: suggested done   F2: alternate done
F3: plan rest   F4: top menu

8.3.3



planning...

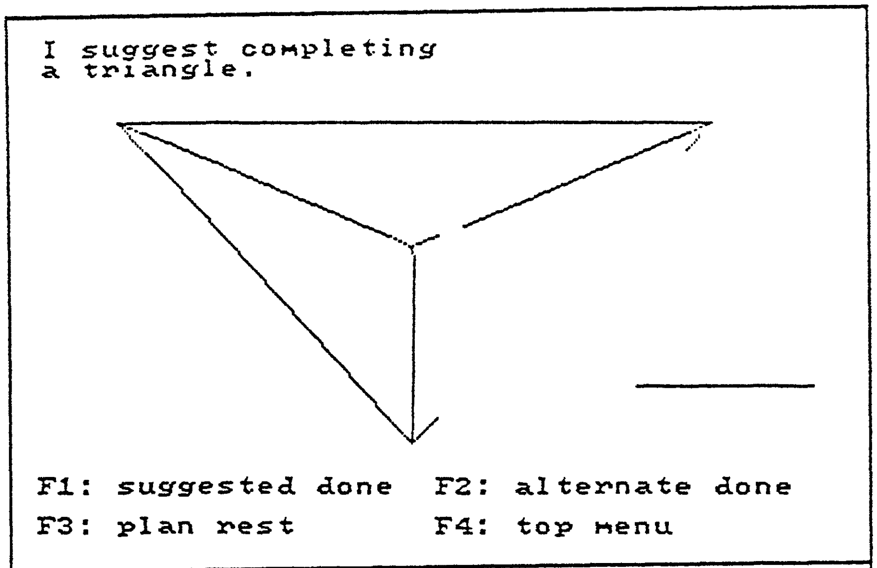
8.4.1



I suggest attaching
a beam to the structure.

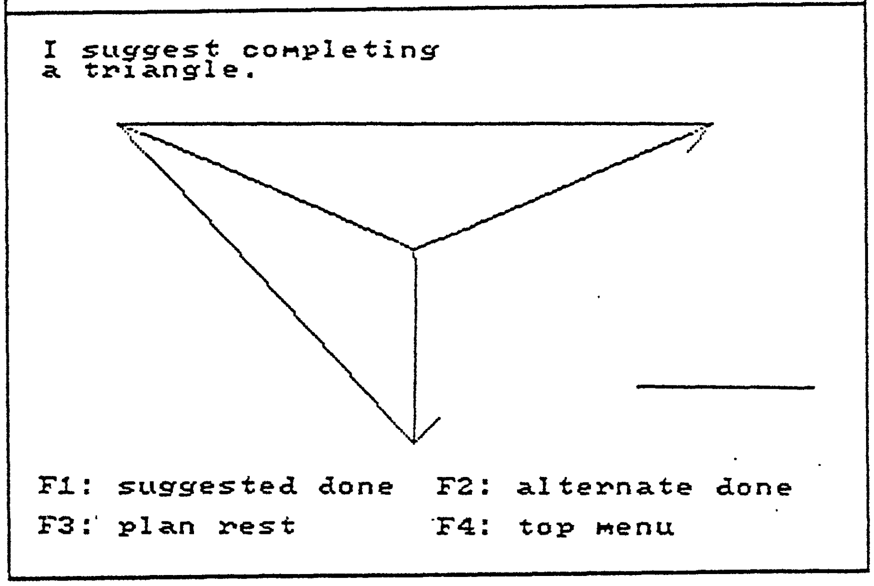
F1: suggested done   F2: alternate done
F3: plan rest   F4: top menu

8.4.2



```
I suggest completing
a triangle.

F1: suggested done   F2: alternate done
F3: plan rest        F4: top menu
```

8.4.3



```
I suggest completing
a triangle.

F1: suggested done   F2: alternate done
F3: plan rest        F4: top menu
```

8.4.4

8.4 Alternate Step Procedure

The operator uses only four keys to communicate an alternate step to ROBIN. This section demonstrates the actual sequence of key strokes, and how they allow the operator to tell ROBIN exactly what was done.

8.4.1 Attaching a Beam

Figures 8.5.1 through 8.5.8 show how the operator communicates an alternate step which involves attaching a beam to the structure. In Figure 8.5.1, two beams are already attached to the base cluster and one top cluster is also attached. ROBIN is suggesting that a second top cluster should be attached to a racked beam.

In the situation depicted by Figure 8.5.1, the operator did not attach a cluster to a racked beam. Instead, the operator decided to attach a beam to the attached top cluster. The beam selected was the beam which is currently the second from the top in the beam rack. The mushroom end to which the beam was attached is the one which brings the structure closer to a triangle.

8.5.1: To begin the alternate step procedure, the operator hits F2 to enter the alternate step menu, which is shown in Figure 8.5.2.

8.5.2 Here there are four choices: F1 if a beam was attached, F2 if a cluster was attached, F3 if a triangle was completed, and F4 to return to the main menu if the operator was mistaken in that he wanted to input an alternate step. Since the operator used BAT to attach a beam, he hits F1.

8.5.3: ROBIN is now in the "present beams" mode in which the operator is presented with the choices of beams which are available for attachment. The first available beam (the top one) in the rack is highlighted in blue. The operator can hit F1 to select that beam or hit F2 to be presented with the next choice. Since the operator attached the second beam in the rack and not the first (which is currently highlighted), the operator hits F2.

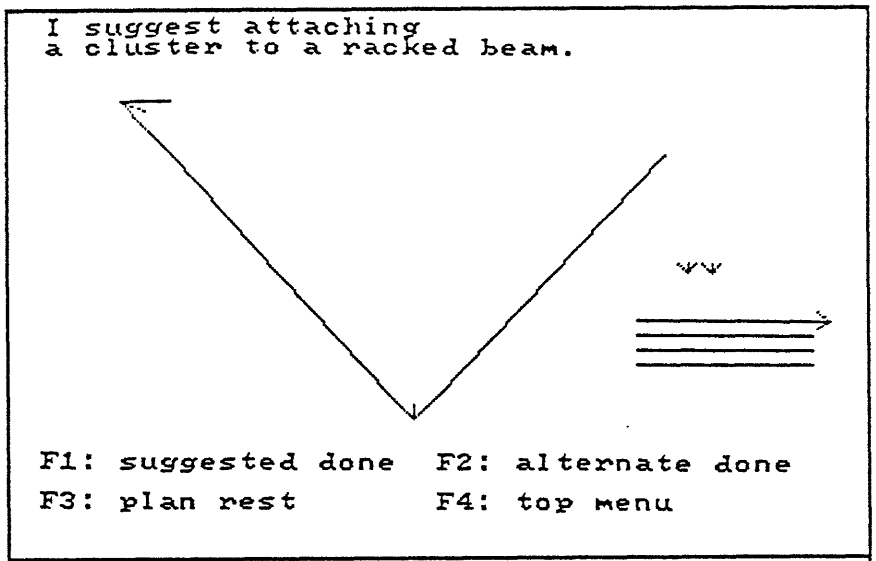
8.5.4: ROBIN now presents the next available beam by highlighting the second racked beam from the top. The previously presented beam is no longer highlighted in blue, but is now drawn in white. Note that the operator could keep striking F2, and ROBIN would eventually cycle through and present the top racked beam again. This allows the operator to recover if he passed by the proper beam. However, since the currently presented beam is the desired one, the operator selects it by hitting F1.

8.5.5: Now that the beam which was attached is known by ROBIN, the operator must convey which mushroom end the beam was attached to. In this figure ROBIN is in the "present mushroom ends" mode, which works exactly the same as the present beams mode except that ROBIN cycles through presenting the available mushroom ends to which a beam could be attached. The first one presented is the attached top cluster's horizontal mushroom end (as it appears in the perspective of the figure) which is the correct one. The operator therefore hits F1 to select it.

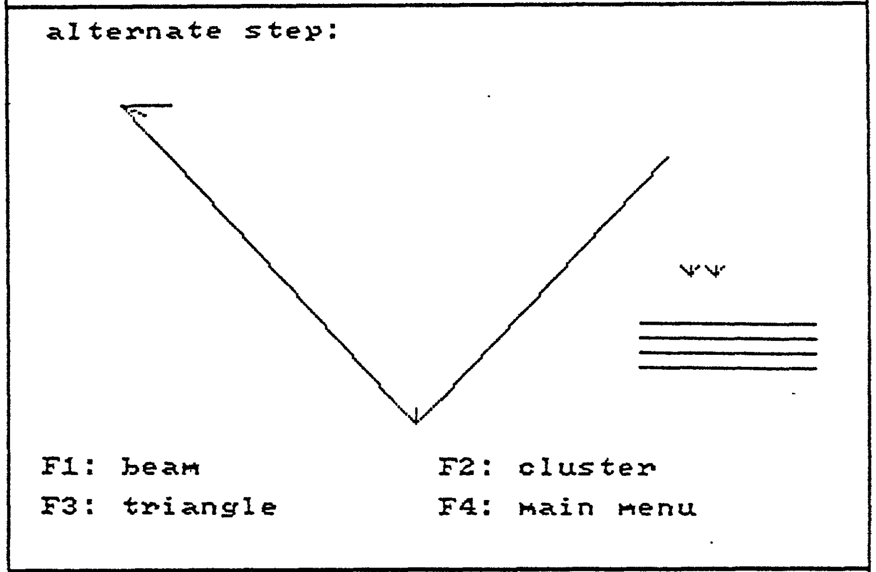
8.5.6: Now that ROBIN knows the beam and where it was attached, it displays the assembly step in blue and asks if this is correct. Since it is, the operator strikes F1 to make this step part of the structure completed so far.

8.5.7: ROBIN then automatically plans the rest of the assembly from the new partial structure.

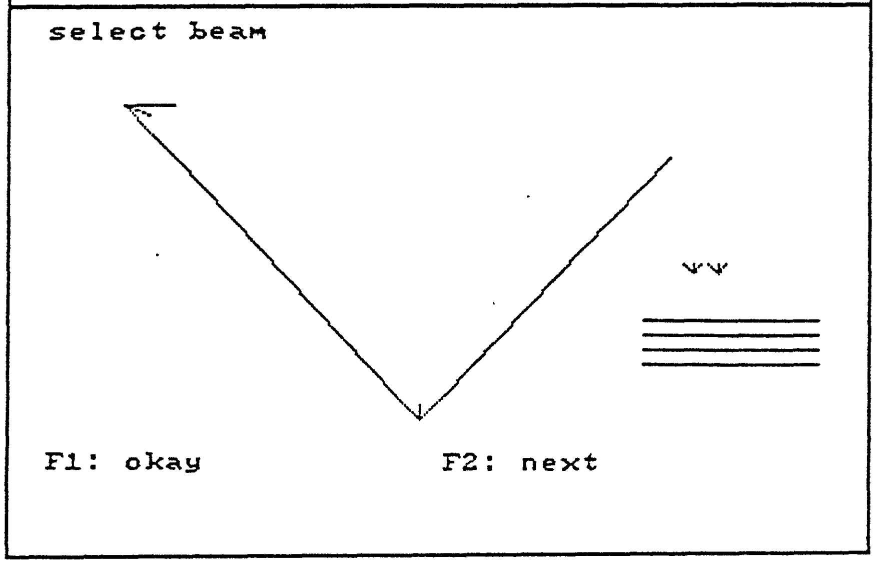
8.5.8: The first step in the new plan involves attaching the left racked top cluster to the right upright, working towards completing the first triangle.


I suggest attaching
a cluster to a racked beam.

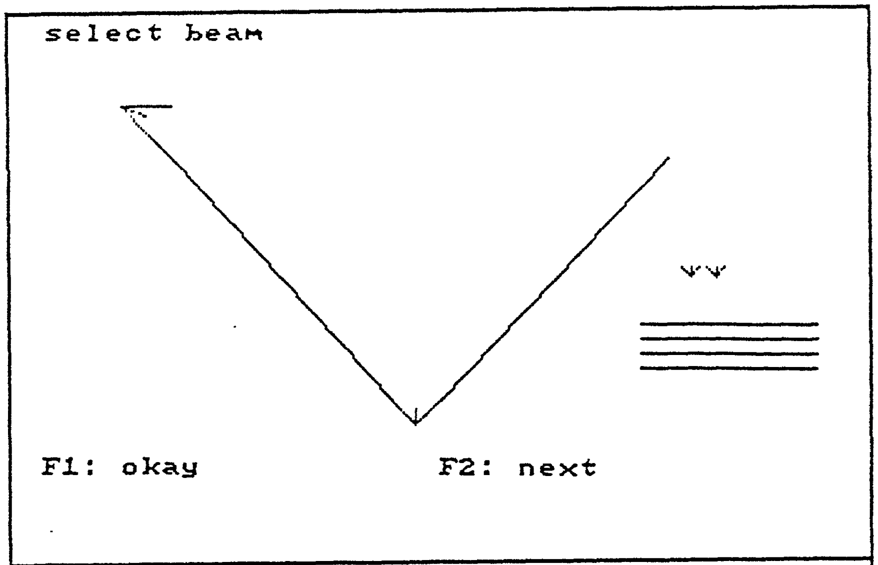
8.5.1


alternate step:

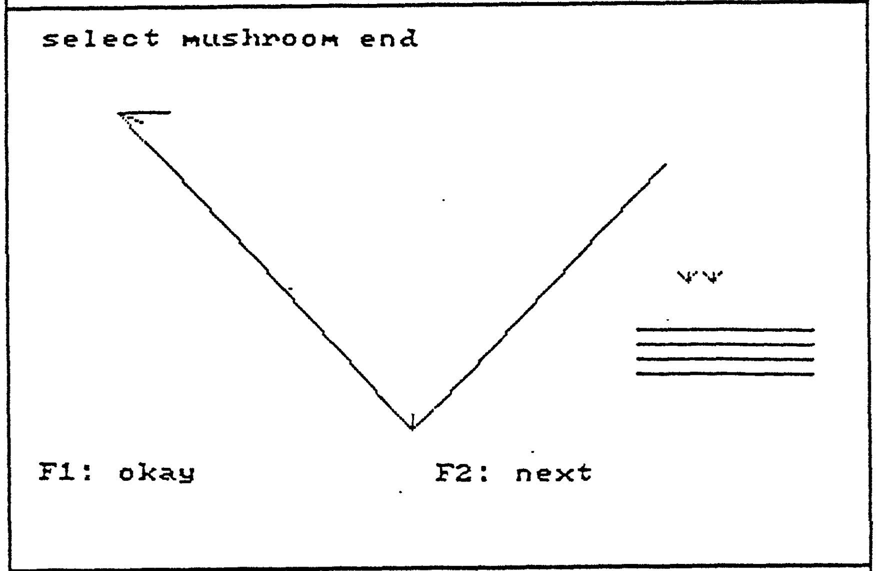
8.5.2


select beam

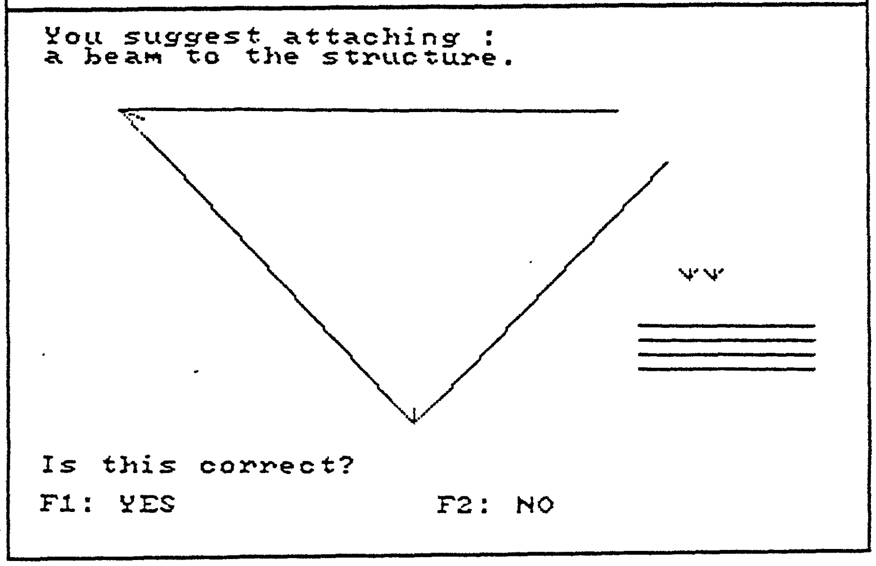
8.5.3


select beam

8.5.4

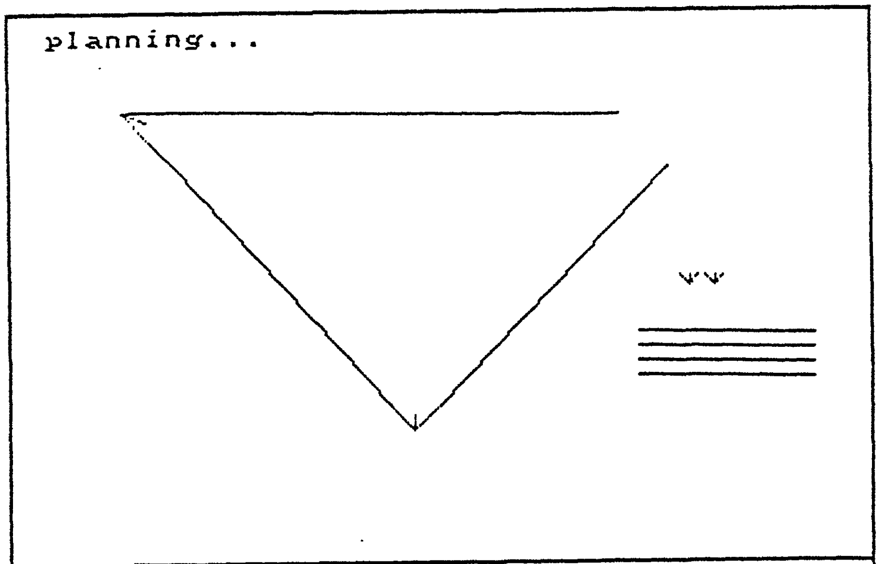

select mushroom end

8.5.5

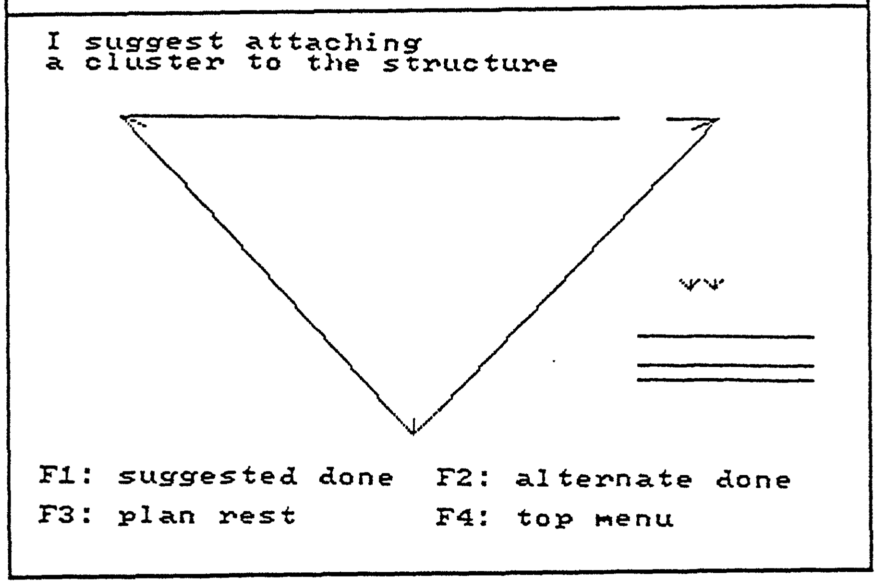

You suggest attaching :
a beam to the structure.

Is this correct?

8.5.6


planning...

8.5.7


I suggest attaching
a cluster to the structure

F1: suggested done F2: alternate done
F3: plan rest F4: top menu

8.5.8

8.4.2 Attaching a Cluster

This example continues with the same assembly as in Figure 8.5. To demonstrate the alternate attachment of a cluster, let us pretend that the operator attached the right racked top cluster instead of the left as suggested in 8.5.8. This is shown in Figures 8.6. Therefore, in 8.5.8, the operator hit F2 to get the alternate step menu as shown in 8.6.1.

8.6.1: The operator hits F2 to attach cluster.

8.6.2: ROBIN presents the left racked cluster. The operator wants the right one, so he hits F2.

8.6.3: ROBIN presents the right racked cluster. The operator hits F1 to okay it and to put ROBIN into presenting beam ends mode.

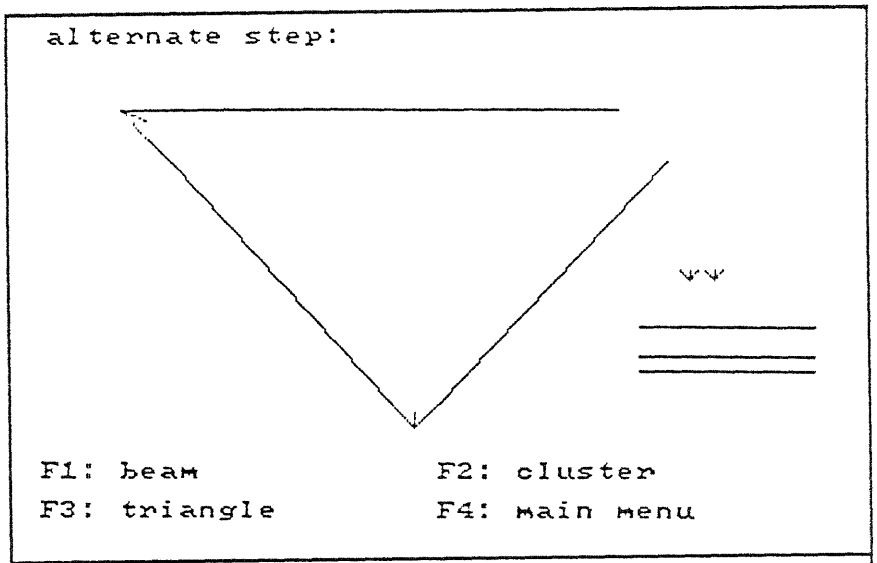
8.6.4: ROBIN begins by presenting a racked beam as the beam to attach the cluster to. Since the beam is drawn small, and since the cluster can only be attached to one of its ends, the whole beam is highlighted in blue. Since the operator attached the cluster to the right upright of the structure, he hits F2 three times to run through the beams in the rack.

8.6.5: ROBIN next presents the first beam end available on the structure which is the one on the right upright. Since this is the correct beam end the operator hits F1.

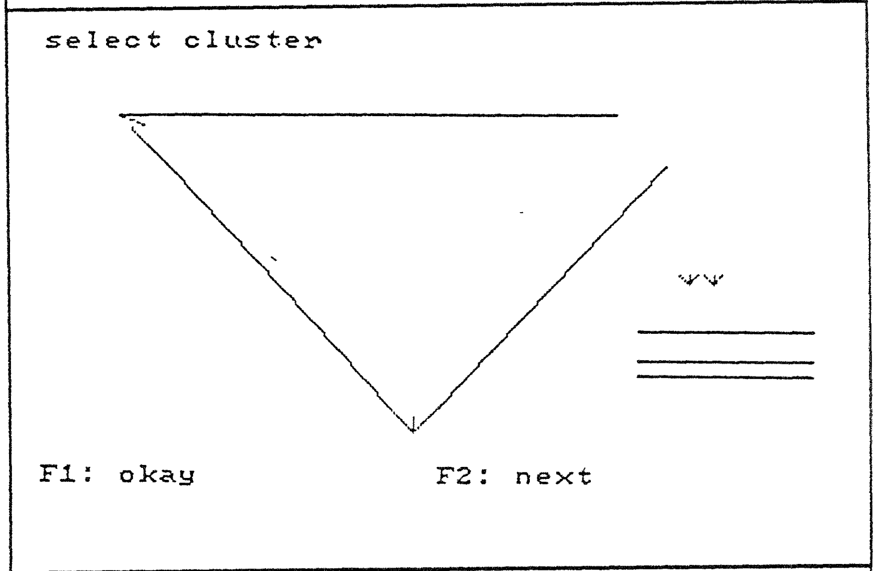
8.6.6: ROBIN then shows the assembly step and asks the operator if it is correct. The operator hits F1.

8.6.7: ROBIN plans the rest of the assembly.

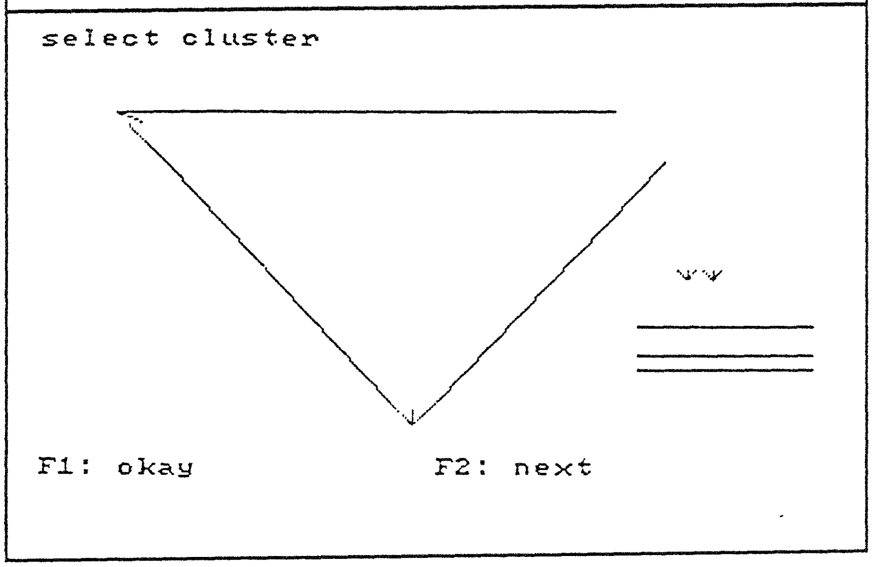
8.6.8: ROBIN suggests completing the triangle.


alternate step:
F1: beam
F2: cluster
F3: triangle
F4: main menu

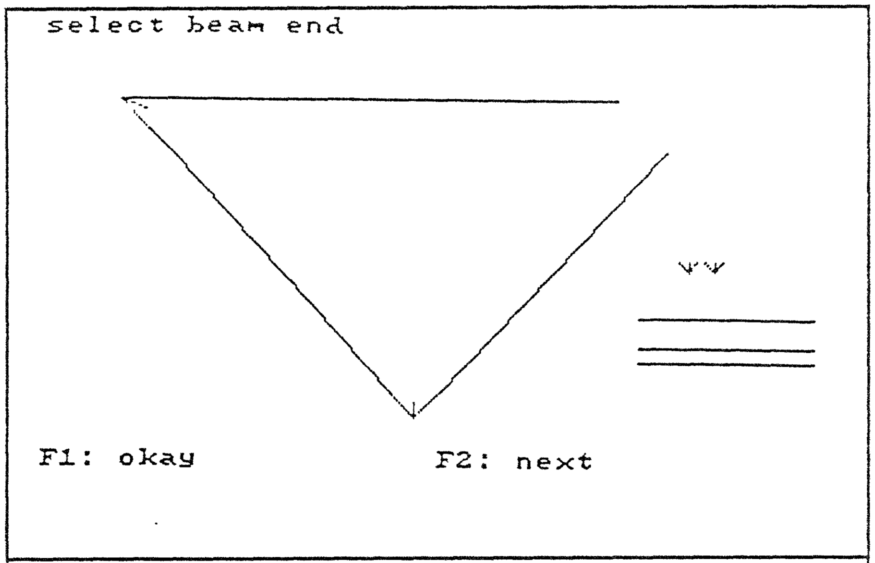
8.6.1


select cluster
F1: okay
F2: next

8.6.2

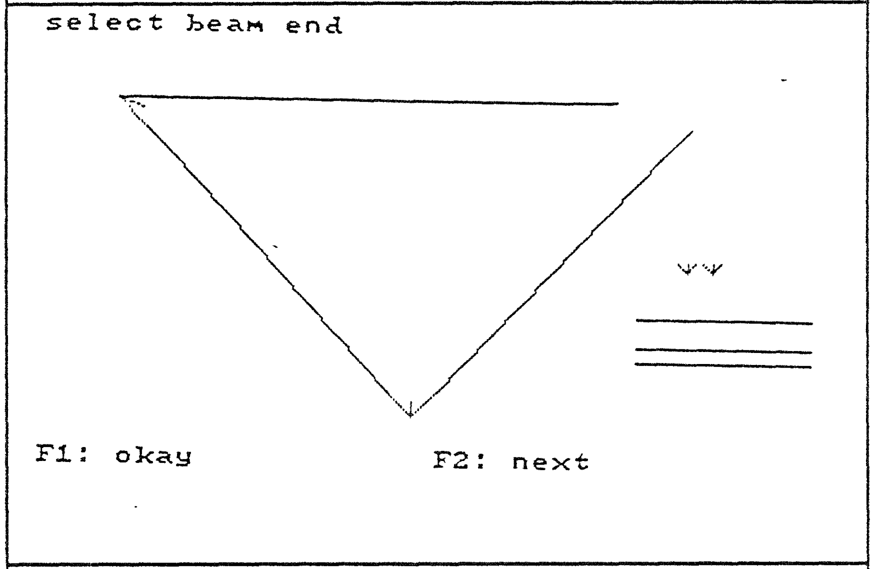

select cluster
F1: okay
F2: next

8.6.3


select beam end

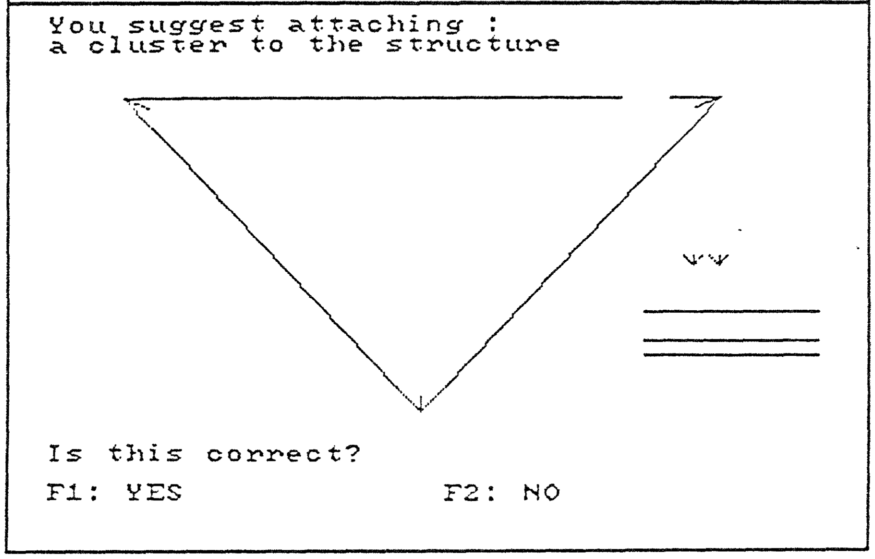
F1: okay F2: next

8.6.4


select beam end

F1: okay F2: next

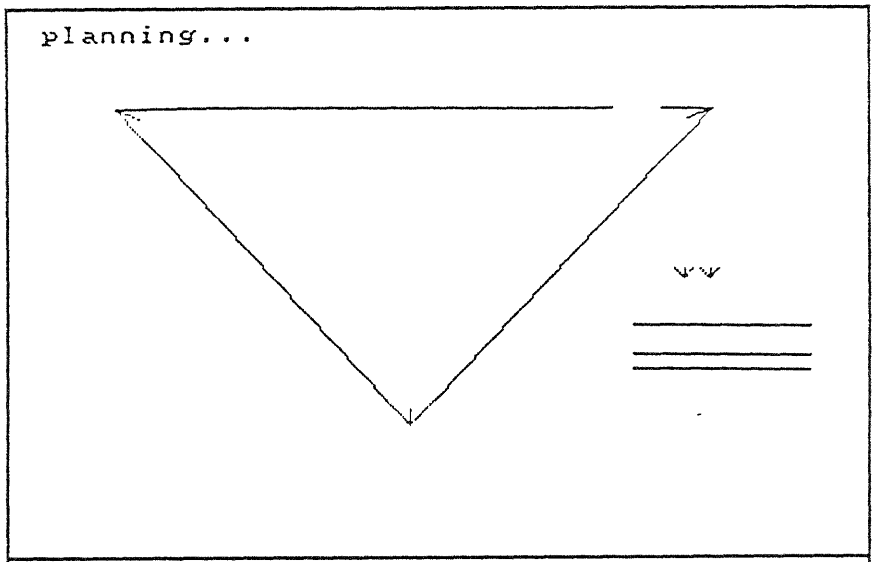
8.6.5


You suggest attaching :
a cluster to the structure

Is this correct?

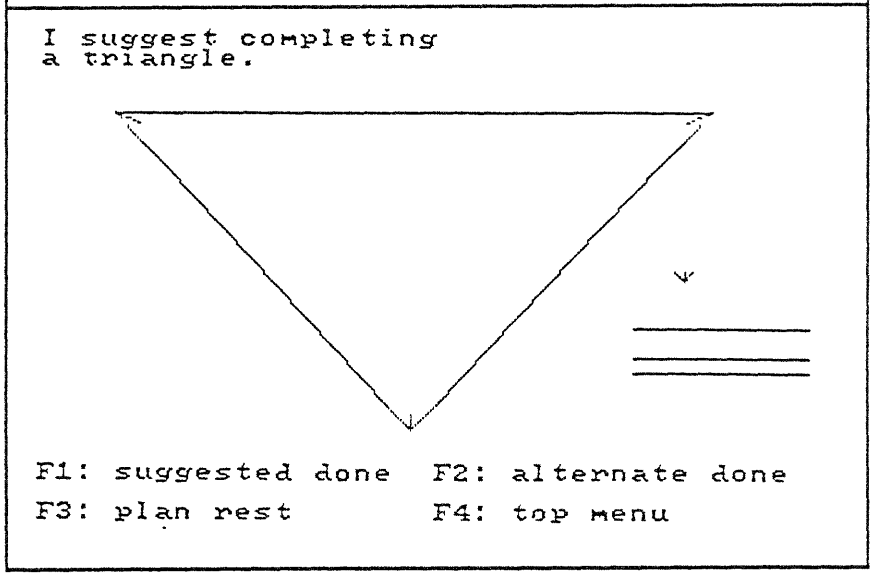
F1: YES F2: NO

8.6.6



planning...

8.6.7



I suggest completing
a triangle.

F1: suggested done F2: alternate done
F3: plan rest F4: top menu

8.6.8

8.4.3 Completing a Triangle

As a quick example of specifying an alternate step which involves completing a triangle, assume that in 8.6.8 the operator hits F2 to enter the alternate step menu as shown in Figure 8.7.1.

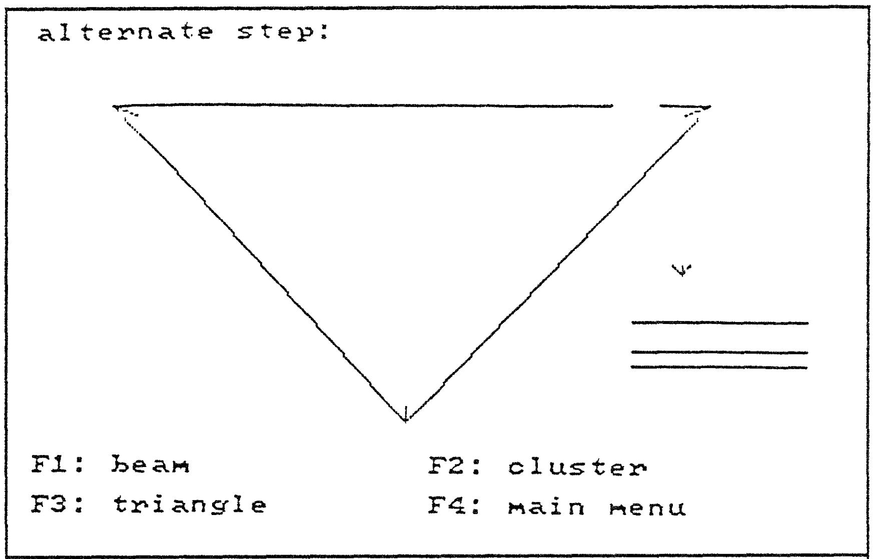
8.7.1: The operator hits F3 to indicate that a triangle completion was done.

8.7.2: Here ROBIN is in the present possible triangle ends mode. The operator can cycle through all of the possible pairs of beam and mushroom ends which could be connected to complete a triangle. Since there is only one possible in this case, ROBIN presents the one possible pair and the operator hits F1 to agree.

8.7.3: ROBIN then asks if this is correct. The operator hits F1.

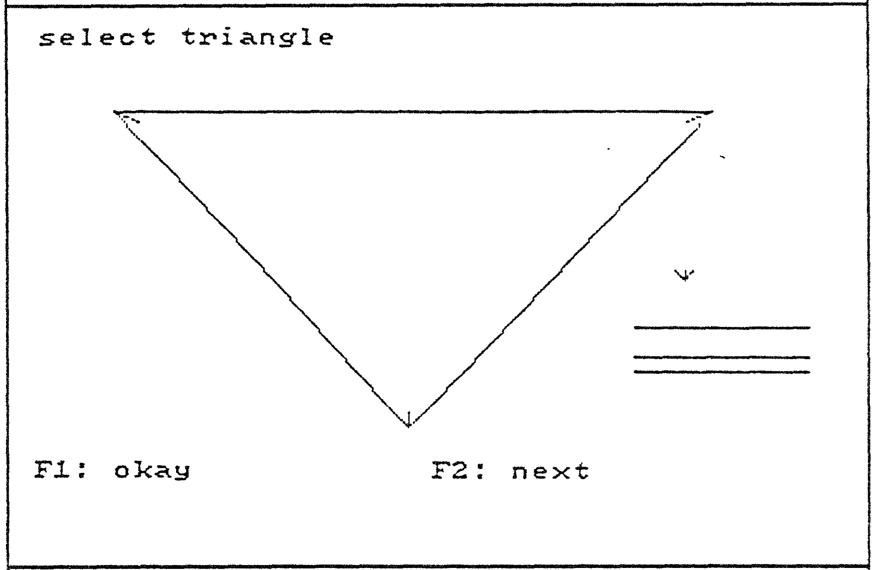
8.7.4: ROBIN plans rest of assembly.

8.7.5: ROBIN suggests attaching a cluster to a racked beam.



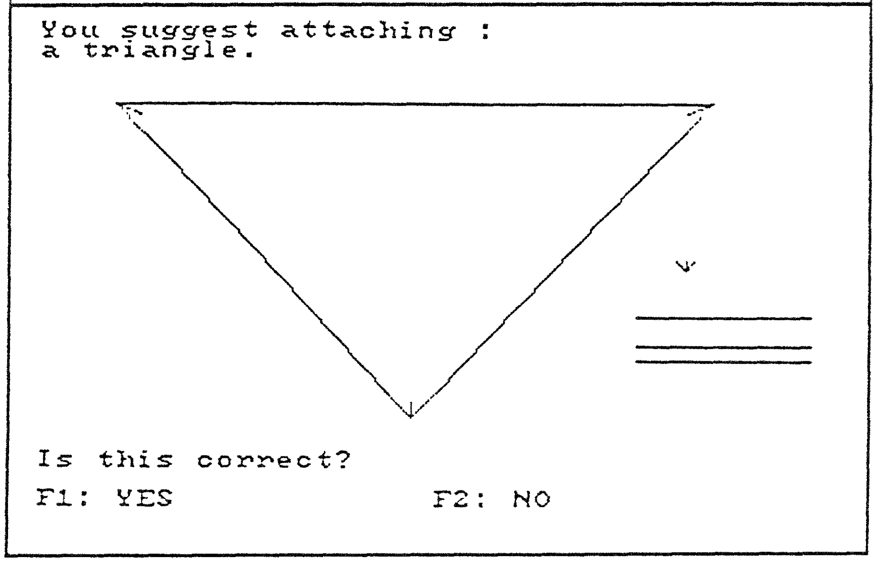
alternate step:

8.7.1



select triangle

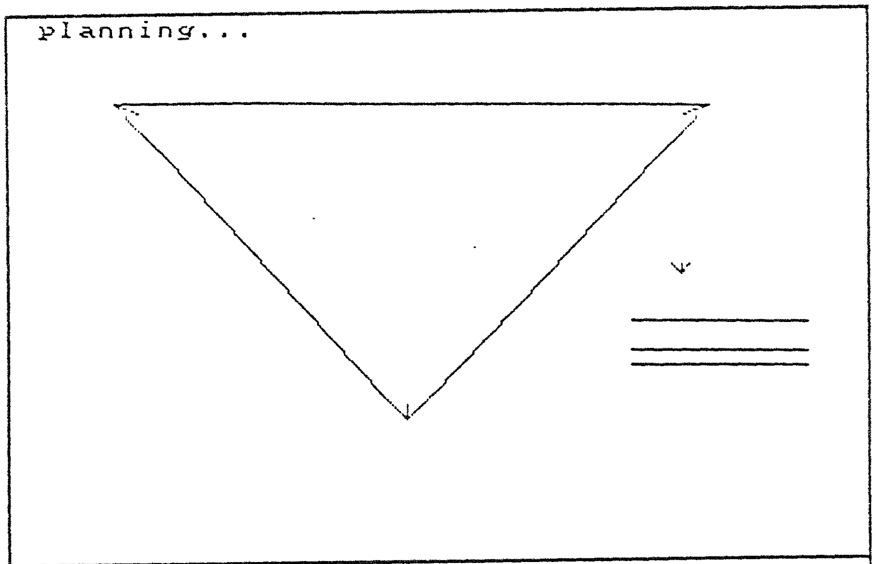
8.7.2



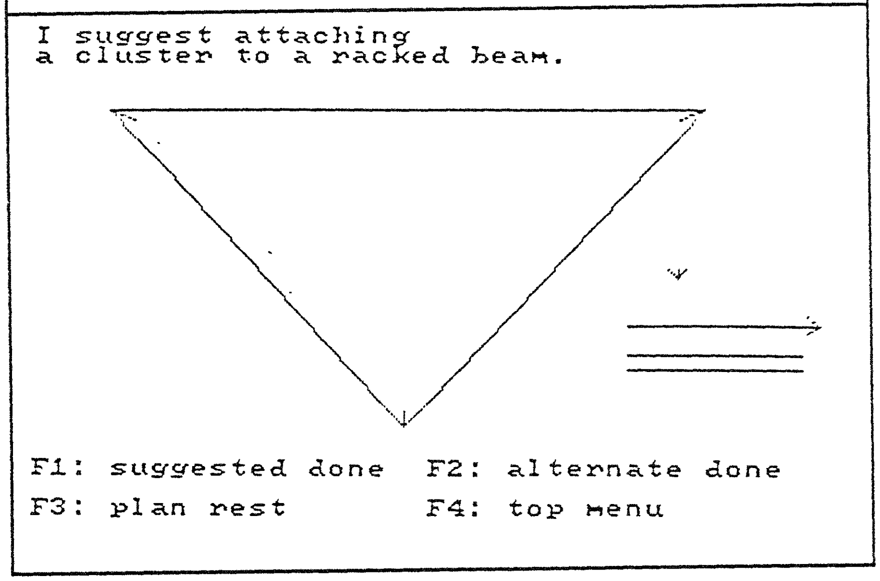
You suggest attaching :
a triangle.

Is this correct?

8.7.3


planning...

8.7.4


I suggest attaching
a cluster to a racked beam.

8.7.5

8.4.4 Alternate Step Mistake Prevention

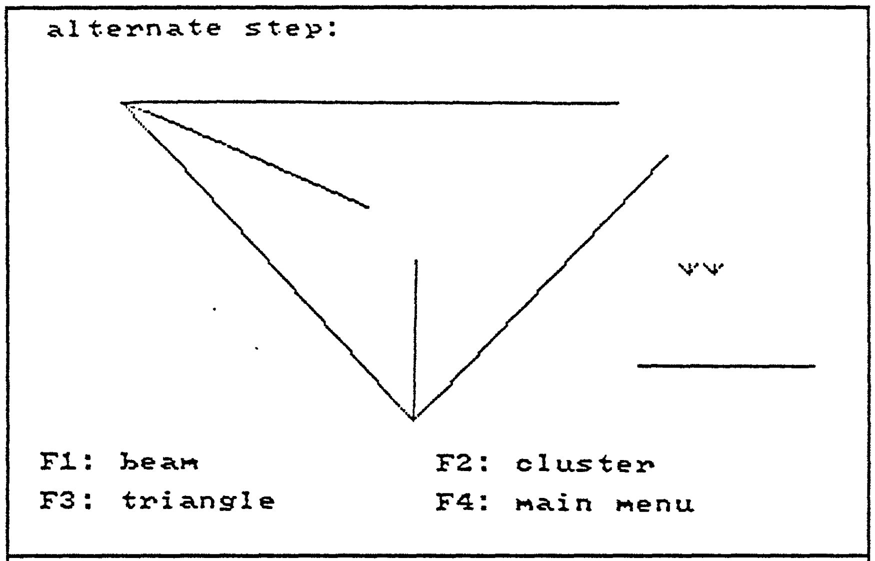
While the logical structure of the alternate menu system prevents the operator from attempting to attach pieces where they can't go, ROBIN also checks for such things as trying to attach a cluster when there are none left in the rack to attach, or trying to attach a beam when there are no free mushroom ends on the structure to attach it to. While these examples are presented as if the operator is trying to do something incorrect, this merely means that the operator is inputting incorrect information to ROBIN. These mistake prevention features use information about the physical nature of the assembly process to identify operator input mistakes.

Figures 8.8.1 and 8.8.2 show what happens if the operator attempts to attach a beam when there are no free mushroom ends. Figure 8.8.1 shows a partial structure in which there are no free mushroom ends, and we are in the alternate step menu. The operator hits F1 to try to attach a beam. In Figure 8.8.2 ROBIN informs the operator that there is no place to attach a beam. After hitting any key, ROBIN returns to the main menu and presents the next suggested step.

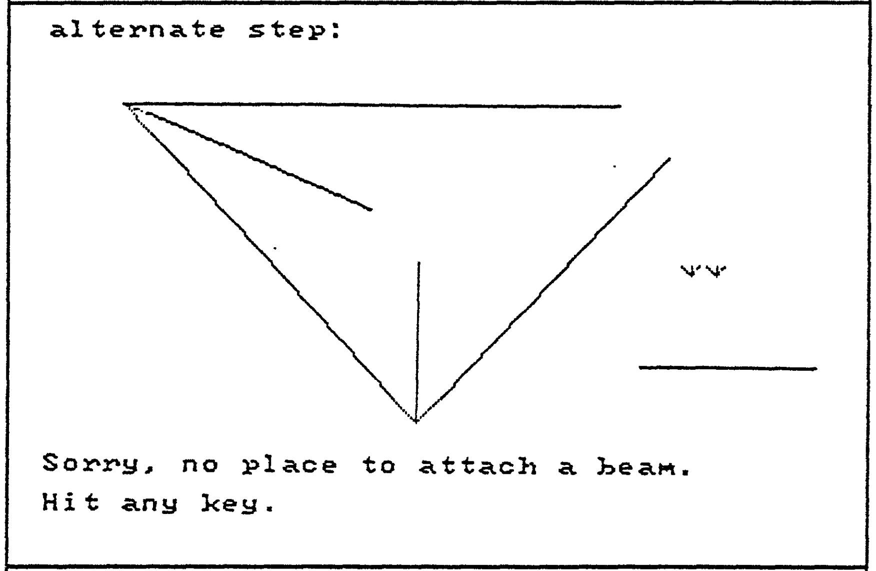
Figures 8.9.1 and 8.9.2 demonstrate the second problem which can arise in trying to attach a beam. The partial structure in Figure 8.9.1 has used up all six beams. When the operator hits F1 to try to attach a beam, ROBIN responds in Figure 8.9.2 with the fact that there are no more beams available.

Since clusters can be attached to racked beams, there never arises the problem of not having any free beam ends to attach a cluster to. However, Figures 8.10.1 and 8.10.2 show what happens when all of the clusters are used up but the operator attempts to attach one any way. In Figure 8.10.1 all of the clusters are attached, and if the operator hits F2 in an attempt to attach a beam, ROBIN responds with the fact that no more clusters are available.

Figures 8.11.1 and 8.11.2 show what happens when no triangles are available, but the operator tries to complete one. In Figure 8.11.1 there is no way to complete a triangle. When the operator hits F3, ROBIN responds in Figure 8.11.2 with the fact that no triangles are possible.

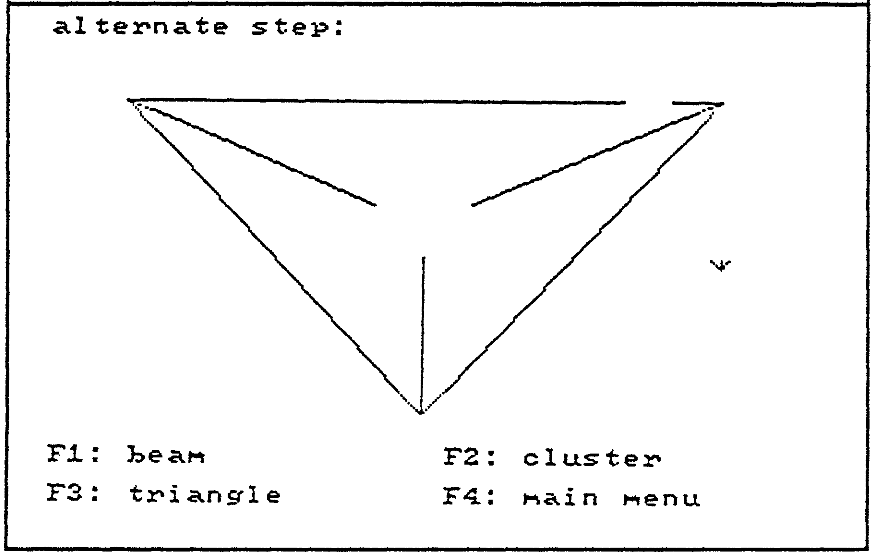

alternate step:

8.8.1

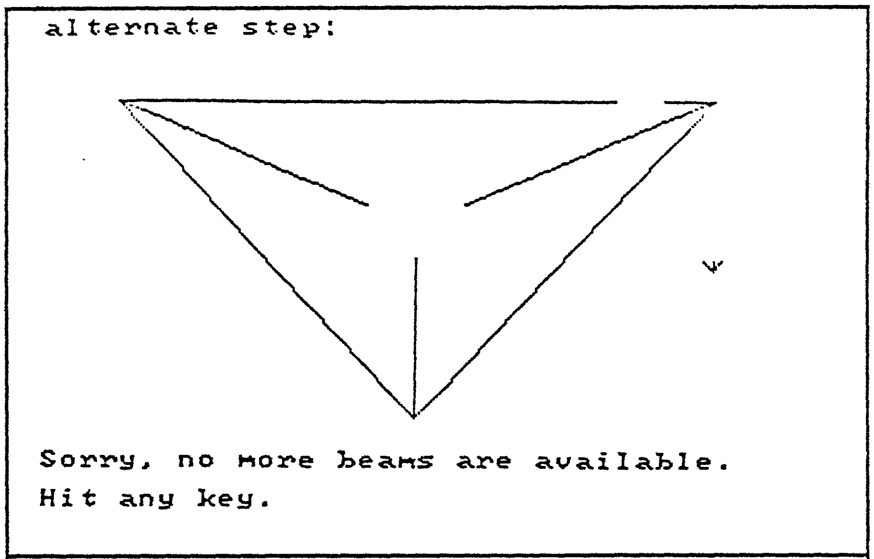

alternate step:

Sorry, no place to attach a beam.
Hit any key.

8.8.2

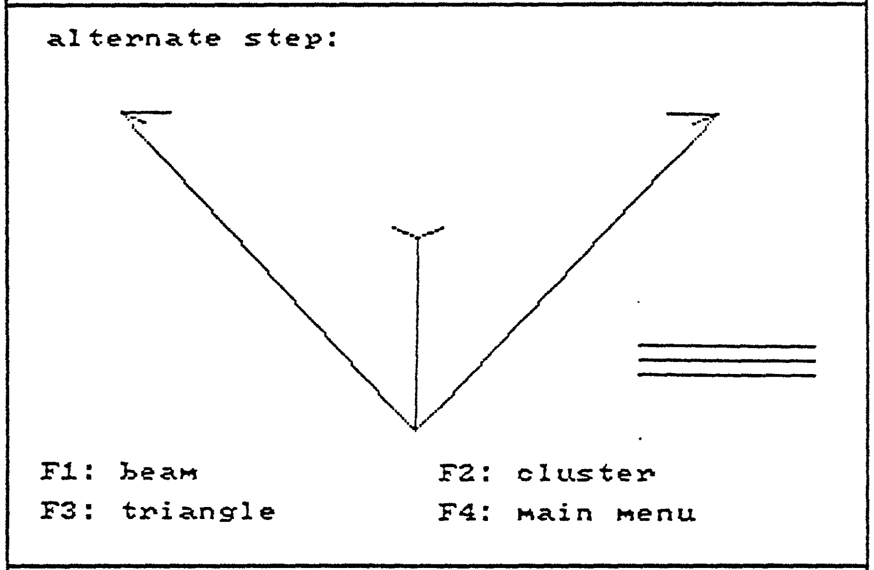

alternate step:

8.9.1


alternate step:

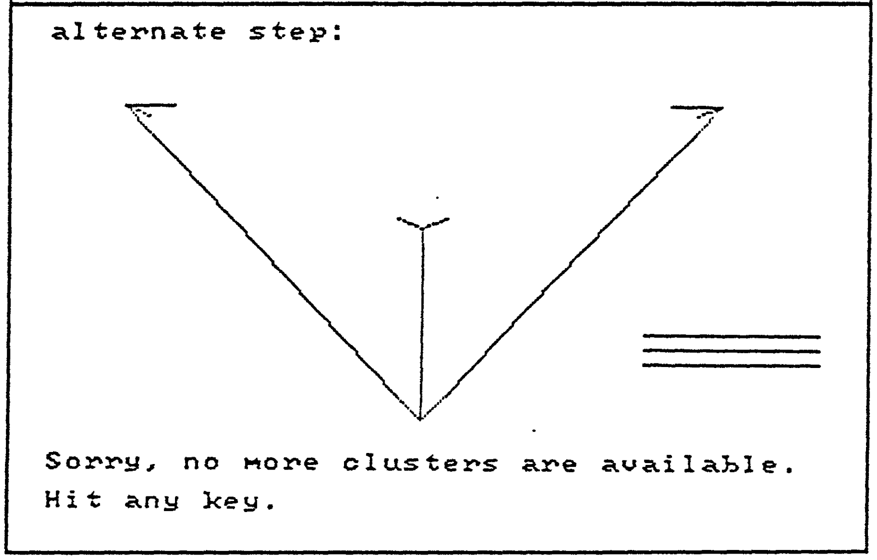
Sorry, no more beams are available.
Hit any key.

8.9.2


alternate step:

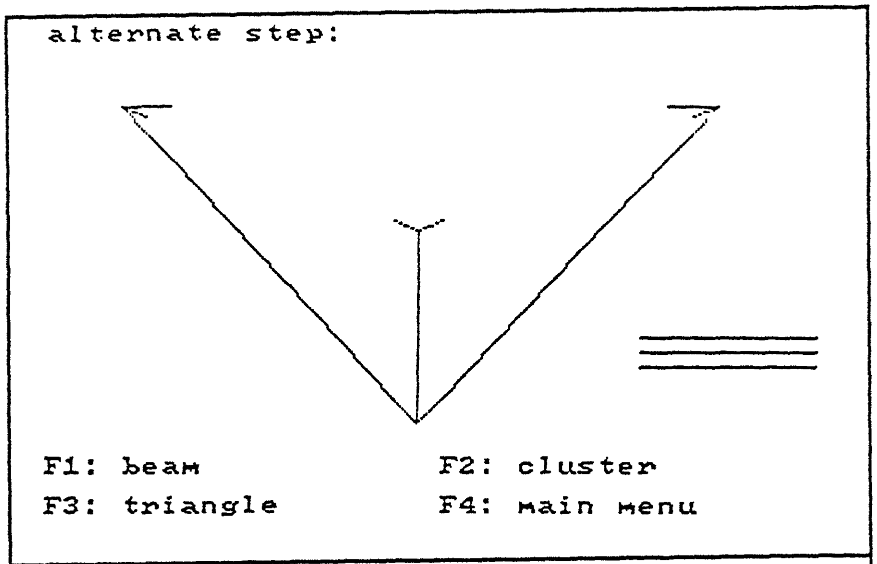
F1: beam F2: cluster
F3: triangle F4: main menu

8.10.1

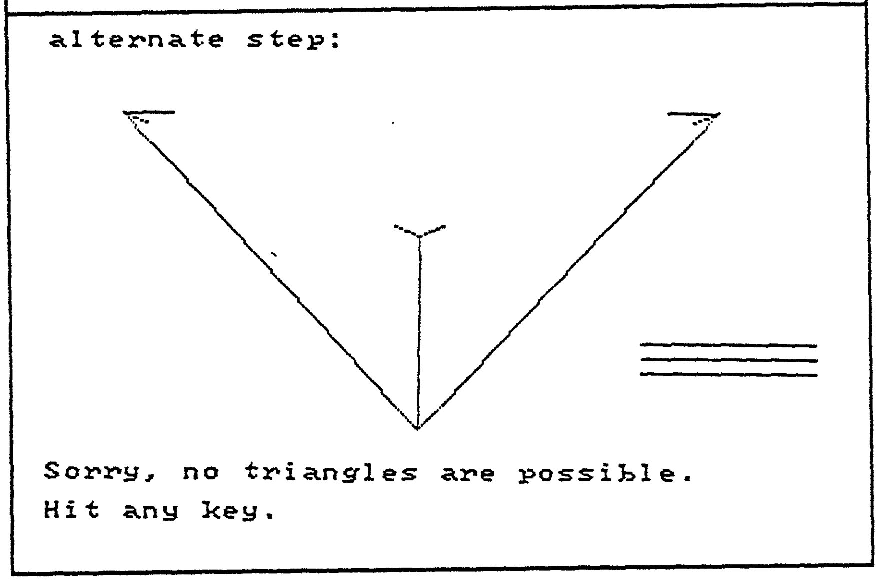

alternate step:

Sorry, no more clusters are available.
Hit any key.

8.10.2



8.11.1



Sorry, no triangles are possible.
Hit any key.

8.11.2

Chapter 9

Conclusions and Recommendations

9.1 Conclusions

The results of the neutral buoyancy testing of BAT performing structural assembly under manual control are presented and discussed in Chapter 5. After needed modifications, BAT was able to assemble the EASE structure. This demonstrates that a teleoperated robot can be built to perform a complex task which was originally designed to be done by a space suited human.

It was found that direct manual control of a robot designed for space structure assembly tends to overload a single human operator, due to many control tasks needing to be performed at once. The greatest such problem involved the need to maneuver and manipulate simultaneously, and was solved by designing the robot so that such situations would not occur during normal operation. Also, the operator experienced fatigue and confusion which could severely limit the amount of time that one person can spend controlling the robot.

To ease the problems of operator confusion and assembly sequence planning difficulty, a real time planner program was developed. This program served the purpose of improving the manual control strategy through an application of AI, and also demonstrated the ability of a computer to model the structural assembly task, as would be needed by an autonomous control system.

9.2 Recommendations

Much of this thesis has involved discussing possible ways of improving teleoperator systems like BAT. Therefore, recommendations for future research are spread throughout this thesis, specifically in Chapter 4.

Several areas of research should be pursued in order to continue the development of useful teleoperated robots. These areas all fall into three major categories: manual control strategies, autonomous capabilities, and tying them together.

Experience with the BAT system shows that it is very taxing for a single human operator to control a space teleoperator. Actual space systems will probably be more complicated than BAT, aggravating this problem. Strategies for overcoming operator overload need to be investigated.

One of the sources of operator fatigue was the difficulty of using the controls in ICS. Work on controls which are easier to use, combined with telepresence concepts, could be very worthwhile.

Progress is needed in many AI technologies before semi or fully autonomous robots will be possible. The biggest challenge exists in the area of sensing what is in the world and building a good representation of it for a computer to reason about. In the case of ROBIN, this function had to be performed by the human.

However, the greatest challenge of all lies in how one combines manual and autonomous control capabilities into a single system. How to hand control back and forth between the two, or how to have the human operator controlling one task while the computer completes another, both offer challenging possibilities for further research.

Appendix A

The Beam Assembly Teleoperator System

A.1 Lab of Orbital Productivity

In the Space Systems Laboratory of M.I.T., the Lab of Orbital Productivity (LOOP group) has performed research into the productivity of man and machines working in space. The motivation for this research was the limited number of manned space shuttle missions available and the limited Extra-Vehicular Activity (EVA) time available per mission. These limitations would severely affect the time available for using astronauts in EVA to construct large space structures such as a space station. Therefore, a research program was begun to study how well astronauts could work in space while performing structural assembly, with the eventual goal of improving their performance in such tasks.

A.1.1 Neutral Buoyancy Testing

The LOOP group's main method of simulating the orbital space environment was through neutral buoyancy. Neutral buoyancy refers to the condition which exists when an object's average density is the same as that of water, so that when underwater the object neither sinks nor floats. This condition simulates the zero gravity condition of the orbital environment. The major difference lies in the drag effects of the water, which are not present in space.

The LOOP group performed its early research by putting human subjects in space suits and having them repeatedly assemble and disassemble simulated space structures underwater. The structural pieces and the space suited test subjects were all made neutrally buoyant for the tests. Learning and productivity of the subject was measured and studied, and new assembly strategies and assembly aids were developed and tested.

The testing was done at the Neutral Buoyancy Simulator (NBS) at the NASA Marshall Space Flight Center in Huntsville, AL. The NBS is basically a large cylindrical tank of water, 40 feet deep and 75 feet in diameter, with test support facilities.

As part of this research, the LOOP group also built and flew a flight experiment on the Space Shuttle. The experiment was called Experimental Assembly of Structures in EVA (EASE). The EASE experiment involved having two astronauts repeatedly assemble and disassemble a single tetrahedral truss in the payload bay of the shuttle. The results of this experimented provided real data on human productivity in space, and permitted a correlation between neutral buoyancy simulation and actual space activities.

A.2 Beam Assembly Teleoperator

Running parallel with the LOOP group's research into human productivity in space was another research program aimed at developing a teleoperator which could perform the same type of structural assembly. It was not considered feasible to build an autonomous robot, but it was intended that this teleoperator, once successfully working under direct human control, could be upgraded towards autonomous operation. Such a teleoperator had the potential of easing or even eliminating the problem of limited astronaut EVA time for tasks like structural assembly by assisting or replacing the astronauts.

This teleoperator was named the Beam Assembly Teleoperator (BAT), since its main task was to assemble beams into a space structure. BAT was designed to work underwater under neutral buoyancy conditions to simulate zero gravity. It was intended that a direct comparison could be made between its performance and that of space suited humans in EVA.

Just as astronauts in space suits are capable of both structural assembly and other tasks like satellite repair and moving payloads about, BAT was also capable of completing tasks other than structural assembly. However, BAT's prime goal was to build the EASE tetrahedral structure, and only work involving BAT's performance in this task is discussed in this thesis.

A.3 Initial Design and Configuration of BAT

It was intended for BAT to be a flexible test bed for many ideas concerning the design and control of space teleoperators, specifically ideas concerning the degree of automation needed, with the advantage of testing those ideas out in a complete system with a realistic task. As a starting point, BAT was first to be made operational under direct manual control. This would provide a baseline with which to compare more automated control schemes , and would provide results which could be helpful when designing these advanced control systems.

Figure A.1 shows the initial concept of what BAT was to look like. BAT was designed to be able to complete all of the subtasks needed to complete the assembly of the EASE structure, and to do so in such a way as to simulate the operation of a similar teleoperator in space. The best way to simulate a complete space teleoperator system was to build one which worked underwater under neutral buoyancy conditions. This meant that all of BAT's systems had to be submersible and neutrally buoyant. All electrical systems had to be potted or kept in pressurized boxes, and buoyant foam had to be added to make negative hardware neutral.

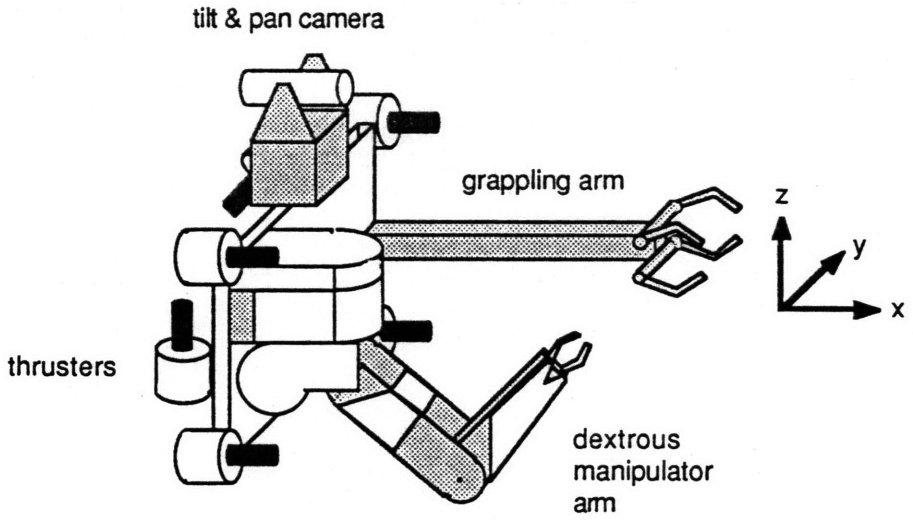


Figure A.1 BAT's Initial Design Configuration

BAT needed to be able to maneuver about underwater, so it was designed to use electrically driven ducted propeller thrusters. These thrusters were powerful enough to overcome drag effects of the water, even when flying while holding a beam or a cluster. Eight such thrusters were used, four facing forward and back (along the x-axis), two facing side to side (along the y-axis), and two facing up and down (z-axis). All thrusters were bidirectional. The four x-thrusters provided translation in the x-direction, as well as pitch (about y) and yaw (about z). The y and z-thrusters were used for translation in their respective directions and were used together to provide roll (about x).

BAT needed to be able to grab and position beams. A rigid grappling arm was designed for this purpose, so that while holding a beam BAT's thrusters could be used to maneuver the beam about. This arm was designed with a large claw for grappling beams, and had a electrically driven roller mechanism for translating the beam along its longitudinal axis (BAT's y-axis) as shown in Figure A.2.

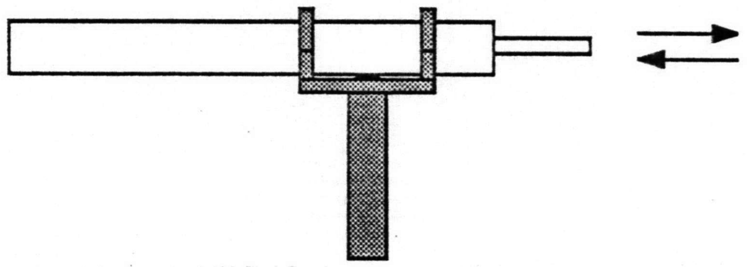


Figure A.2 Translation of Beam with Grappling Arm

BAT needed to be able to both grab and manipulate clusters, in order to make a joint connection. A dextrous manipulator arm was designed for this task. The strength and dexterity of the arm were designed to handle the worst case arm loading depicted in Figure A.3. Here BAT is grabbing a free beam (not yet attached to the structure) and the dextrous arm has grabbed a mushroom end of a fixed cluster; the arm must now move both BAT and the beam to make the joint connection. The arm needed to be powerful enough to move the combined mass of the beam and BAT, with enough accuracy to make the joint (See Reference 6 for details on the manipulator arm design).

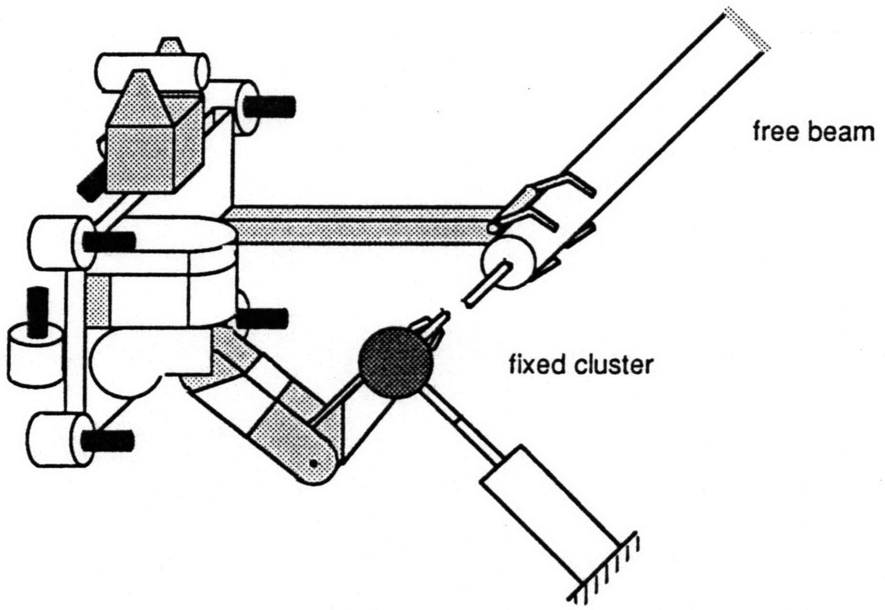


Figure A.3 Worst Case Arm Loading

The operator needed some type of feedback from the worksite to successfully control BAT. Therefore, a camera system was designed which could look around and view the work area of the manipulator arm, and the general vicinity. This system consisted of a black and white video camera with a wide angle lens mounted in a tilt and pan unit (see Figure A.1).

Both the manipulator arm and the tilt and pan unit were driven by electric DC motors and used optical encoders for determining joint positions. Both the grappling claw and the dextrous arm's claw were driven by pneumatic cylinders.

The initial robotic hardware of BAT therefore included a six degree of freedom (DOF) motion platform with eight thrusters, a grappling arm for grabbing and holding beams, a five DOF manipulator arm for grasping mushroom ends and assembling joints, and a two DOF tilt and pan unit for pointing a video camera.

At the beginning of this thesis work, BAT was built and in the configuration shown in Figure A.1, except for the fact that the motion frame was much larger and boxlike, as shown in Figure A.4. This increase in the size of the frame of the motion platform was due to initial underestimates of the room needed to fit all of the electronics, batteries, and pressure bottles which were required. As was described in Chapter 4, this caused problems since the mass which the dextrous arm would need to be able to move in the worst case situation (Figure A.3) was now greater than the arm had originally been designed for.

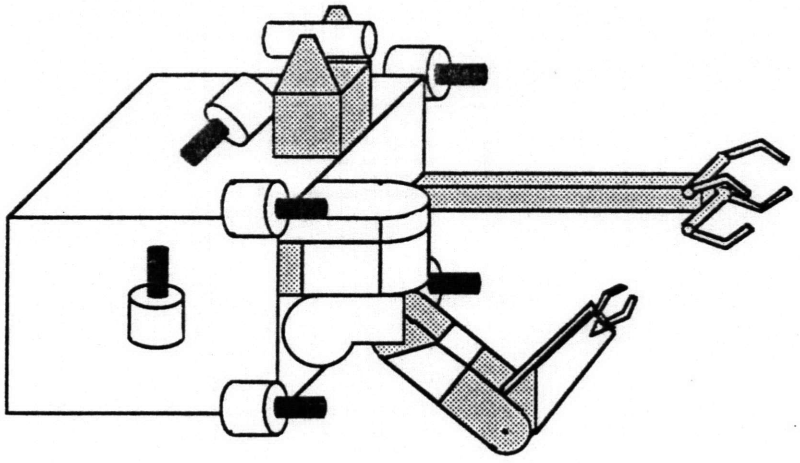


Figure A.4 BAT's Actual Initial Configuration

A.4 Initial Design and Configuration of ICS

BAT was designed and built to be a teleoperator, so by definition a control station for the remote human operator needed to be built. Figure A.5 sketches the Integrated Control Station (ICS) which housed the controls and displays by which the human operator was to control BAT, and the computers and control electronics needed to interpret the operator's commands and communicate with BAT.

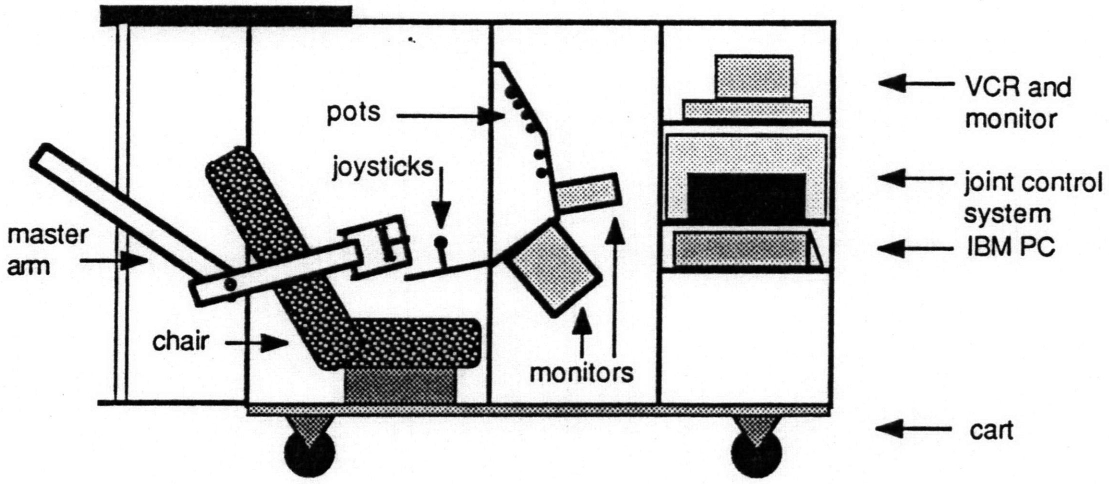


Figure A.5 ICS's Initial Configuration

Since ICS needed to be moved between the LOOP group's lab and the M.I.T. swimming pool for developmental testing, ICS was built upon a wheeled cart. Onto this cart was bolted a rectangular aluminum frame. A chair for the operator was mounted at one end of the station. At the opposite end several shelves were installed to hold the electronics and computers. For the purpose of mounting controls and video monitor displays, four rack panels were mounted in the middle of the station within reach of the operator. These four rack panels are referred to as the control panels.

On the top shelf was mounted a VCR and a video monitor. The VCR was used to record the video output of BAT's tilt and pan camera, but could also be used to record video from external underwater cameras trained on BAT. The monitor displayed whatever was being recorded on the VCR.

On the middle shelf was mounted the joint control system (JCS). The JCS was a custom built computer system which controlled the positions of the joints of the dextrous arm and the tilt and pan. The JCS took joint positions as inputs and performed the closed-loop control required to move the joints into that position. The JCS also contained the electronics for driving the serial communications link (commlink).

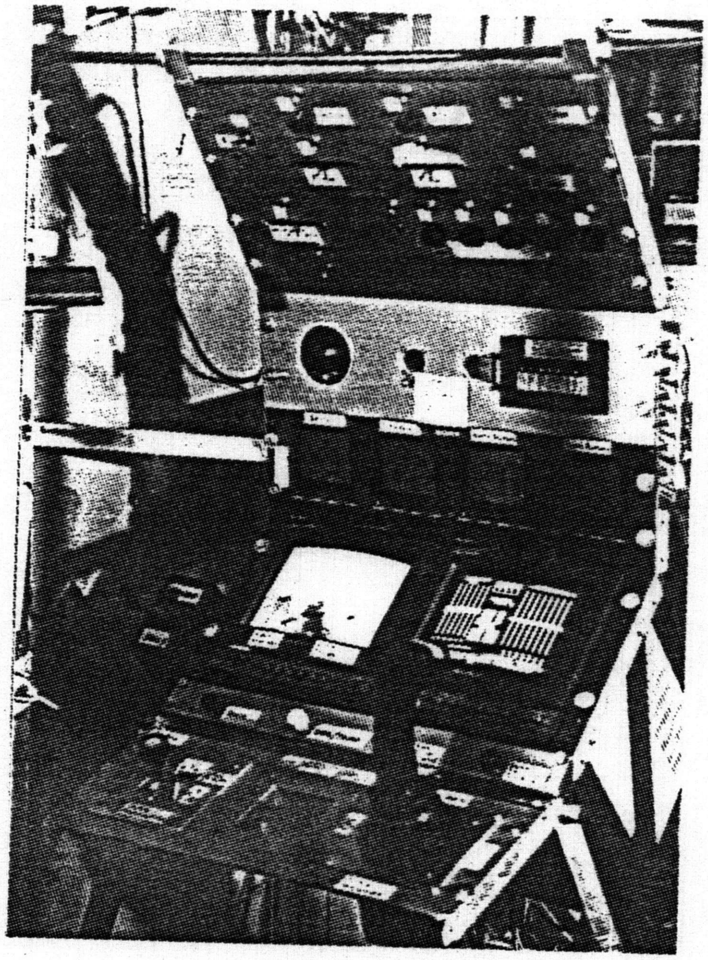


Figure A.6 ICS's Control Panels

On the bottom shelf was mounted an IBM PC. The PC was used to read input commands from the controls used by the operator and translate them into joint commands for the JCS. The PC was also used to drive a graphics display of what commands were being input. This display was shown on the right large monitor in the second control panel. This can be seen in Figure A.6, which shows a better view of the control panels than is provided in Figure A.5.

The operator needed a way to control BAT's thrusters so that BAT could be flown (technically though, BAT does not fly -- it swims). Therefore two three DOF hand-controllers were mounted in the lower control panel. The right hand-controller controlled

BAT's pitch, roll, and yaw, while left controlled translations forward and back, side to side, and up and down (plus and minus x, y, and z directions respectively). As can be seen in Figure A.6, the left hand-controller is actually a joystick, while the right hand-controller is a standard spacecraft rotational hand-controller.

To control the roller of the beam grappling arm, and to control the tilt and pan unit, several pots (potentiometers) where mounted in the top two control panels. Extra pots in the panels were used during development before other controls were installed, and served as backup controls.

To control the dextrous manipulator arm, a "master" arm was built and mounted behind and to the right of the chair. The master arm is a geometric equivalent of the "slave" dextrous arm with potentiometers mounted at each joint. When the operator moves the end effector of the master arm about, the computer reads the position of the pots and commands the slave arm to move in the same way.

To control the claw of the rigid grappling arm, and any other open/close or on/off functions of BAT (dextrous arm claw, main power relay, and software mode options), a system of software definable switches was installed near the hand-controllers and on the grip of the master arm.

A computer keyboard was also within reach of the operator but was intended mainly for initializing the computers and not as a control to be used while performing structural assembly. It is not shown in Figures A.5 or A.6.

Six video monitors, two large and four small, were also installed in the control panels. The two large monitors where mounted in the second control panel, while the four smaller monitors where mounted in the lower half of the third control panel. One of the large screens was for the picture from the video camera on BAT, while the other was used for the computer generated display of control commands (See Figure A.6.). The four small screens were used for outside cameras located on pool walls or carried by a support diver.

Therefore the initial control configuration of ICS was as follows: Two three DOF hand-controller for flying BAT, switches for controlling claws and software modes, pots for controlling the beam roller and tilt and pan unit, a five DOF master arm for controlling the dextrous slave arm, and six video screens for visual feedback to the operator.

A.5 Initial Control Strategy for Controlling BAT with ICS to Build EASE

There were three main subtasks identified to be involved with completing the EASE structure. They were:

1) Attaching a free cluster to a beam.
2) Attaching a free beam to a cluster.
3) Completing a triangle.

Below is described how the operator was to use the controls and displays in ICS to get BAT to complete each of the three subtasks.

To attach a free cluster to a beam the operator was to do the following:

1) Using the appropriate pots, adjust the tilt and pan so that the camera was oriented for flying.
2) Using the joysticks, fly BAT over to the cluster rack, and dock to the rack by grappling the appropriate fitting with the grappling arm.
3) Use pots to adjust the tilt and pan to look down onto the work area.
4) Use the master arm to control the slave arm to reach out, grab a cluster, and remove it from the rack.
5) Use pots to adjust the tilt and pan for flying.
6) Use the joysticks to fly to and grapple the beam to which the cluster is

to be attached.

7) Use pots to adjust the tilt and pan to look down onto the work area.

8) If necessary, use a pot to control the grappling arm's roller to adjust BAT's position at the end of the beam.

9) Use the master arm to control the slave arm to move the cluster to the beam and make the joint.

To attach a free beam to a cluster the operator was to do the following:

1) Using the appropriate pots, adjust the tilt and pan so that the camera was oriented for flying.

2) Using the joysticks, fly BAT over to the beam rack, and dock to the rack by grappling the appropriate fitting with the rigid grappling arm.

3) Use the joysticks to fly BAT so as to remove the beam from the rack and then fly over to the cluster to which the beam is to be attached.

4) Use pots to adjust the tilt and pan to look down onto the work area.

5) If necessary, use a pot to control the grappling arm's roller to adjust the beam's position in the grappling arm's claw.

6) Use the master arm to reach out with the slave arm and grab the mushroom end to which the beam will be attached.

7) Use the master arm to control the slave arm to move BAT, and thereby the rigidly grappled beam, to make the joint.

To complete a triangle the operator was to do the following:

1) Using the appropriate pots, adjust the tilt and pan so that the camera was oriented straight ahead for flying.

2) Using the joysticks, fly BAT over to the triangle joint to be completed, and dock to the beam involved by grappling it with the rigid arm.

3) Use pots to adjust the tilt and pan to look down onto the work area.

4) Use the master arm to reach out with the slave arm and grab the mushroom end involved.

5) Use the master arm to control the slave arm to move BAT and the rigidly grappled beam to make the joint.

Appendix B

EASE Assembly Time Data

Included in this appendix are outlines of the four basic tasks involved with assembling the EASE structure. Each step outline is composed of subtasks (+) and the task primitives (-) which make them up. A task primitive is defined as the smallest task for which a time could be found from the video tape data. A subtask may also be composed of other subtasks.

In the outlines, a time in seconds is written after every subtask or primitive task for which a time was found. If the task primitives making up a subtask have times, then the time for the subtask is the sum of these. Otherwise subtasks simply have a time without any times for their task primitives, implying that a time was found for the subtask, but not for all of its task primitives

Also, an outline showing how these four steps are combined for a full assembly is presented.

- + ASSEMBLY OF EASE STRUCTURE - 5,352
  - + Attach first top cluster to first upright beam - 393
  - + Attach first upright beam to base cluster - 522
  - + Attach second top cluster to second upright beam - 393
  - + Attach second upright beam to base cluster - 522
  - + Attach third top cluster to third upright beam - 393
  - + Attach third upright beam to base cluster - 522
  - + Attach first cross beam to first top cluster - 442
  - + Complete first triangle - 427
  - + Attach second cross beam to second top cluster - 442
  - + Complete second triangle - 427
  - + Attach thrid cross beam to third top cluster - 442
  - + Complete third triangle - 427

+ Attach first top cluster to first upright beam - 393
    + Fly to cluster rack - 97
        - Command left claw to open -
        - Maneuver with PUMA to cluster rack -
        + Dock to proper handle on cluster rack -
            - Terminate on proper handle -
            - Command left claw to close on handle -
    + Attach cluster to beam - 296
        + Get cluster from rack - 62
            + Change to T&P viewing - 10
                - Switch to T&P cameras -
                - Connect head controller -
            + UNSTOW right arm - 21
                - Pan to STOW handle -
                - Power up slave arm -
                - Command right claw to open -
                - Move slave arm into general work area -
            + Grab cluster - 1
                - Move right claw onto proper mushroom end -
                - Command right claw to close -
            + Secure cluster for flying - 27
                - Remove cluster from rack -
                - Position arm and cluster for flying -
                - Disconnect master arm -
            + Change to belly viewing - 3
                - Center T&P cameras -
                - Disconnect head controller -
                - Switch to belly cameras -
        + Fly to end of beam - 90
            - Command left claw to open -
            - Maneuver with PUMA to right end of proper beam in beam rack -
            + Dock to beam end of proper beam -
                - Terminate on proper beam end -
                - Command left claw to close -
        + Connect mushroom end to beam end - 144
            + Change to T&P viewing - 10
                - Switch to T&P cameras -

  - Reconnect head controller -
+ Attach cluster to beam - 46
  - Pan to look at cluster -
  - Reconnect master arm -
  - Insert mushroom end into recepticle to set rocker -
+ Slide sleeve - 45
  + Cycle master arm -
    - Disconnect master arm -
    - Reconnect master arm -
  - Command right claw to open -
  - Move right claw onto sleeve -
  - Command right claw to close -
  - Slide sleeve onto mushroom end -
  + Cycle master arm -
    - Disconnect master arm -
    - Reconnect master arm -
  - Command right claw to open -
  - Move right arm into general work area -
+ STOW the right arm - 39
  - Move right claw onto the STOW handle -
  - Command right claw to close -
  - Power down right arm -
+ Change to belly viewing - 4
  - Center T&P cameras -
  - Disconnect head controller -
  - Switch to belly cameras -

- + Attach first upright beam to base cluster - 522
  - + Fly to beam rack - 56
    - - Command left claw to open -
    - - Maneuver with PUMA to beam rack -
    - + Dock to proper handle on beam rack -
      - - Terminate on proper handle -
      - - Command left claw to close on handle -
  - + Load beam in beam carrier - 41
    - + Change to T&P viewing -
      - - Switch to T&P cameras -
      - - Connect head controller -
    - + Grab beam and swing back with beam carrier -
      - - Pan to look over shoulder -
      - - Command beam carrier to swing out -
      - - Visually check beam in claw -
      - - Command beam carrier claw to close -
      - - Command beam carrier to swing back -
    - + Change to belly viewing -
      - - Center T&P -
      - - Disconnect head controller -
      - - Switch to belly cameras -
  - + Fly to base cluster - 109
    - - Command left claw to open -
    - - Maneuver with PUMA to base cluster -
    - + Dock to proper mushroom end on base cluster -
      - - Terminate on proper mushroom end -
      - - Command left claw to close -
  - + Connect beam end to mushroom end - 316
    - + Change to T&P viewing - 5
      - - Switch to T&P cameras -
      - - Connect head controller -
    - + Swing out beam - 18
      - - Pan to look over shoulder -
      - - Command beam carrier to swing -
      - - Visually check beam in proper position -
    - + UNSTOW right arm - 68
      - - Pan to STOW handle -
      - - Power up slave arm -

- Command right claw to open -
    - Move slave arm into general work area -
+ Transfer beam from carrier to right arm - 32
    - Move right claw onto beam end -
    - Command right claw to close -
    - Pan to look over shoulder -
    - Open beam carrier claw -
    - Command beam carrier to swing -
    - Pan to look at beam end -
- Move beam end recepticle over mushroom end to set rocker - 112
+ Slide sleeve - 52
    + Cycle master arm -
        - Disconnect master arm -
        - Reconnect master arm -
    - Command right claw to open -
    - Move right claw onto sleeve -
    - Command right claw to close -
    - Slide sleeve onto mushroom end -
    + Cycle master arm -
        - Disconnect master arm -
        - Reconnect master arm -
    - Command right claw to open -
    - Move right arm into general work area -
+ STOW the right arm - 22
    - Move right claw onto the STOW handle -
    - Command right claw to close -
    - Power down right arm -
+ Change to belly viewing - 7
    - Center T&P cameras -
    - Disconnect head controller -
    - Switch to belly cameras -

- + Attach first cross beam to first top cluster - 442
  - + Fly to beam rack - 46
    - - Command left claw to open -
    - - Maneuver with PUMA to beam rack -
    - + Dock to proper handle on beam rack -
      - - Terminate on proper handle -
      - - Command left claw to close on handle -
  - + Load beam in beam carrier - 34
    - + Change to T&P viewing -
      - - Switch to T&P cameras -
      - - Connect head controller -
    - + Grab beam and swing back with beam carrier -
      - - Pan to look over shoulder -
      - - Command beam carrier to swing out -
      - - Visually check beam in claw -
      - - Command beam carrier claw to close -
      - - Command beam carrier to swing back -
    - + Change to belly viewing -
      - - Center T&P -
      - - Disconnect head controller -
      - - Switch to belly cameras -
  - + Fly to proper top cluster - 116
    - - Command left claw to open -
    - - Maneuver with PUMA to proper top cluster -
    - + Dock to proper mushroom end on cluster -
      - - Terminate on proper mushroom end -
      - - Command left claw to close -
  - + Connect beam end to mushroom end - 246
    - + Change to T&P viewing - 10
      - - Switch to T&P cameras -
      - - Connect head controller -
    - + Swing out beam - 13
      - - Pan to look over shoulder -
      - - Command beam carrier to swing -
      - - Visually check beam in proper position -
    - + UNSTOW right arm - 26
      - - Pan to STOW handle -
      - - Power up slave arm -

  - Command right claw to open -
  - Move slave arm into general work area -
+ Transfer beam from carrier to right arm - 32
  - Move right claw onto beam end -
  - Command right claw to close -
  - Pan to look over shoulder -
  - Open beam carrier claw -
  - Command beam carrier to swing -
  - Pan to look at beam end -
- Move beam end recepticle over mushroom end to set rocker - 97
+ Slide sleeve - 49
  + Cycle master arm -
    - Disconnect master arm -
    - Reconnect master arm -
  - Command right claw to open -
  - Move right claw onto sleeve -
  - Command right claw to close -
  - Slide sleeve onto mushroom end -
  + Cycle master arm -
    - Disconnect master arm -
    - Reconnect master arm -
  - Command right claw to open -
  - Move right arm into general work area -
+ STOW the right arm - 15
  - Move right claw onto the STOW handle -
  - Command right claw to close -
  - Power down right arm -
+ Change to belly viewing - 4
  - Center T&P cameras -
  - Disconnect head controller -
  - Switch to belly cameras -

+ Complete first triangle - 427
  + Fly to free end of beam - 91
    - Command left claw to open -
    - Maneuver with PUMA to free end of beam -
    + Dock to free end of beam -
      - Terminate on beam end -
      - Command left claw to close -
  + Make free flying cluster grab - 214
    + Change to T&P viewing - 10
      - Switch to T&P cameras -
      - Connect head controller -
    + UNSTOW right arm - 22
      - Pan to STOW handle -
      - Power up slave arm -
      - Command right claw to open -
      - Move slave arm into general work area -
    + Position BAT for free flying cluster grab - 32
      - Select and switch to proper flying mode -
      - Maneuver with PUMA to proper position -
    + Grab cluster - 150
      - Move right claw onto proper mushroom end -
      - Command right claw to close -
  + Assemble joint - 122
    + Make free flying connectiom of beam end to mushroom end - 73
      - Select and switch to proper flying mode -
      - Maneuver with PUMA to line up joint -
      - Insert mushroom end into beam end to set rocker -
    + Slide sleeve - 28
      + Cycle master arm -
        - Disconnect master arm -
        - Reconnect master arm -
      - Command right claw to open -
      - Move right claw onto sleeve -
      - Command right claw to close -
      - Slide sleeve onto mushroom end -
      + Cycle master arm -
        - Disconnect master arm -
        - Reconnect master arm -

- Command right claw to open -
- Move right arm into general work area -

+ STOW the right arm - 15
  - Move right claw onto the STOW handle -
  - Command right claw to close -
  - Power down right arm -
+ Change to belly viewing - 6
  - Center T&P cameras -
  - Disconnect head controller -
  - Switch to belly cameras -

Appendix C

Listing of ROBIN Code

```
/* ROBIN_5.PRO */

code = 7000
domains
        piece = beam(beam_num) ; cluster(cluster_num)
        end = beam_end(beam_num,beam_end_num) ; mush_end(cluster_num,m
        beam_num = symbol
        cluster_num = symbol
        beam_end_num = symbol
        mush_end_num = symbol
        beams = piece*
        clusters = piece*
        coords = real*
        position = coords*
        key = cr;esc;break;tab;btab;del;bdel;ins;endk;home;
                fkey(integer);up_arrow;down_arrow;left_arrow;
                right_arrow;char(CHAR);other
        step =  beam_to_attached_cluster(end,end);
                cluster_to_attached_beam(end,end);
                cluster_to_racked_beam(end,end);
                beam_with_cluster_to_attached_cluster(end,end);
                triangle(end,end)
        sequence = step*

database
        beam_rack(beams)
        cluster_rack(clusters)
        attached(piece)
        racked(piece)
        free_mush_end(end)
        free_beam_end(end)
        free_racked_beam_end(end)
        connected(end,end)
        location(piece,position)
        first_beam_ratio(real)
        second_beam_ratio(real)
        third_beam_ratio(real)
        center(coords)
        completed(sequence)
        proposed(sequence)
        dummy_beam_one(beams)
        dummy_beam_two(beams)
        dummy_cluster_one(clusters)
        dummy_cluster_two(clusters)
        perm_assert(dbasedom)
        perm_retract(dbasedom)
        temp_assert(dbasedom)
        temp_retract(dbasedom)

predicates

/* Input and Menu system */

        run
        top_menu
        top_refresh
        top_dispatch(key)
        main_menu
        main_menu_refresh
        main_text_refresh
        main_dispatch(key)
        test_if_complete
        alt_do_or_dont(symbol)
        alt_menu(symbol)
        alt_text_refresh
        alt_dispatch(key,end,end)

        beam_menu(end,end)
        cluster_menu(end,end)
        triangle_menu(end,end)

        cycle_text_refresh

        select_beam(piece)
        beam_select_check(key,piece)
        select_alt_mush_end(end,piece)
        mush_end_select_check(key,end)
        refresh_all_graphics
        geometry_not_ok_mush_end(end)
        geometry_not_ok_beam_with_cluster(end,piece)

        select_cluster(piece)
        cluster_select_check(key,piece)
        select_alt_beam_end(end,piece)
        beam_end_select_check(key,end)
        get_alt_beam_end(end)
        geometry_not_ok_beam_end(end)
        racked_cluster_check
        draw_alt_beam_end(end,integer)

        triangle_possible
        select_triangle(end,end)
        triangle_select_check(key,end,end)

        execute_alternate(end,end,symbol)
        check_alt(symbol)
        alt_check(key,symbol)
        clear_screen
        plan

/* Planning rules */

        plan_rest_of_assembly
        true_default_rules
        beam_with_cluster
        first_triangle
        check_for_triangle
        complete_first_triangle
        step_toward_first_triangle

        make_last_connection
        attach_third_beam
        attach_third_cluster
        attach_third_beam_with_cluster
        attach_second_cluster
        attach_second_beam_with_cluster
        attach_first_beam_with_cluster
        get_beam_with_cluster(piece)
        last_connection_geometry_ok(end,end)
        third_beam_geometry_ok(end,end)
        third_cluster_geometry_ok(end,end)
        third_beam_with_cluster_geometry_ok(end,end)
        second_cluster_geometry_ok(end,end)
        second_beam_with_cluster_geometry_ok(end)
        first_beam_with_cluster_geometry_ok(end)

        triple_cluster_check(cluster_num,cluster_num,cluster_num)
        double_cluster_check(cluster_num,cluster_num)
        single_cluster_check(cluster_num)

        get_next_piece(piece)
        get_next_beam(piece)
        get_next_cluster(piece)
        free_beam_end_test(end)
        attach(piece)
        attach_beam(piece)
        attach_cluster(piece)
        connect_beam_to_attached_mush_end(piece,end)
        connect_cluster_to_attached_beam_end(piece,end)
        connect_cluster_to_racked_beam_end(piece,end)
        connect_beam_with_cluster_to_attached_mush_end(piece,end)
        complete_triangle(end,end)
        append(sequence,sequence,sequence)
        append(coords,coords,coords)
        append(beams,beams,beams)
        append(clusters,clusters,clusters)
        add_to_prop_seq(step)
        add_to_comp_seq(step)
        restore_old_database
        restore_temp_retracts
        erase_temp_asserts
        complete_any_triangles
        make_all_possible_connections(end,end)
        true
        geometry_ok(end,end)                    /*  (beam_end,much_enc
        next_cw(mush_end_num,mush_end_num)
        next_ccw(mush_end_num,mush_end_num)
        other_end(beam_end_num,beam_end_num)

/*  Details of attaching things together */

        init
        zap_database
        keep_current_database
        unmark_temp_asserts
        unmark_temp_retracts
        present_next_suggested
        execute_next_suggested
        execute_and_mark(step)
        do_beam_to_attached_cluster(end,end)
        do_cluster_to_attached_beam(end,end)
        do_cluster_to_racked_beam(end,end)
        do_beam_with_cluster_to_attached_cluster(end,end)
        do_triangle(end,end)
        remove_from_beam_rack(piece)
        search_beam(piece)
        check_beam(piece,piece)
        remove_from_cluster_rack(piece)
```

```
search_cluster(piece)
check_cluster(piece,piece)
equal(position,position)
equal(beams,beams)
equal(clusters,clusters)
equal(piece,piece)
equal(end,end)
equal(beam_num,beam_num)
equal(beam_end_num,beam_end_num)
equal(cluster_num,cluster_num)
equal(mush_end_num,mush_end_num)
equal(coords,coords)
equal(symbol,symbol)
equal(key,key)
equal(step,step)
equal(sequence,sequence)
equal(integer,integer)
free_(end)
mush_end_length(real)
beam_end_length(real)
beam_length(real)

/* Graphics for parts racks */

refresh_rack_graphics
draw_attached_clusters
draw_parts_rack(beams,clusters)
draw_beam_rack(beams)
draw_cluster_rack(clusters)
draw_in_rack(piece)
draw_beam_in_rack(piece,integer)
draw_cluster_in_rack(piece,integer)
draw_cluster_attached_to_beam_in_rack(piece,integer)
displace_position(position,coords,position)
shrink_piece(position,position)
shrink_cluster_on_beam(position,position,position)
draw_line_in_rack(coords,coords,integer)

/* Graphics for structure */

refresh_structure_graphics
draw_completed_sequence(sequence)
draw_completed_step(step)
draw_beam_with_free_end(piece,integer)
draw(piece)
draw_beam(piece,integer)
draw_cluster(piece,integer)
draw_line(coords,coords,integer)
draw_beam_end(end,integer)
draw_mush_end(end,integer)

theta(real)
offsets(real,real)
shrink_factor(real)
locate_fixed_cluster(cluster_num,coords)
locate_fixed_beam(beam_num,coords)
new_beam_location(end,end)
assert_beam_location(beam_num,beam_end_num,coords,coords,
        coords,coords)
mush_end_vector(cluster_num,mush_end_num,coords,coords)
```

```
        new_cluster_location(end,end)
        assert_cluster_location(cluster_num,mush_end_num,coords,
                coords,coords,coords)
        beam_end_vector(beam_num,beam_end_num,coords,coords)
        new_cluster_location_in_rack(end,end)
        repeat

        readkey(key)
        key_code(key,char,integer)
        key_code2(key,integer)

goal
        run.

clauses

        true.

        equal(X,X).

        run :-
                graphics(1,1,0),
                trace(off),
                makewindow(1,7,0,"",2,0,20,40),
                makewindow(2,7,0,"",22,0,3,40),
                makewindow(3,7,0,"",0,0,2,40),
                makewindow(4,7,0,"",12,30,11,10),
                top_menu.

        top_menu :-
                shiftwindow(2),!,
                top_refresh,!,
                readkey(KEY),!,
                clearwindow,!,
                shiftwindow(1),!,
                top_dispatch(KEY),!,
                top_menu,!.

        top_refresh :-
                clearwindow,
                cursor(0,0),write("F1: initialize"),
                cursor(0,20),write("F2: start"),
                cursor(2,20),write("F4: exit").

        top_dispatch(KEY) :-
                equal(KEY,fkey(1)),init;
                equal(KEY,fkey(2)),shiftwindow(3),clearwindow,
                main_menu_refresh,
                main_menu;
                equal(KEY,fkey(4)),exit;
                true.

        main_menu :-
                repeat,
                readkey(KEY),
                main_dispatch(KEY),
                fail.

        main_menu_refresh :-
                refresh_structure_graphics,
        refresh_rack_graphics,
        test_if_complete,
        present_next_suggested,
        main_text_refresh.

main_text_refresh :-
        shiftwindow(2),
        clearwindow,
        cursor(0,0),write("F1: suggested done"),
        cursor(0,20),write("F2: alternate done"),
        cursor(2,0),write("F3: plan rest"),
        cursor(2,20),write("F4: previous menu").

main_dispatch(KEY) :-
        equal(KEY,fkey(1)),
        execute_next_suggested,
        main_menu_refresh,!;
        equal(KEY,fkey(2)),
        alt_menu(FLAG),!,
        alt_do_or_dont(FLAG),!,
        main_menu_refresh,!;
        equal(KEY,fkey(3)),
        plan,
        main_menu_refresh,!;
        equal(KEY,fkey(4)),
        restore_old_database,!,
        top_menu,!;
        true.

test_if_complete :-
        beam_rack([]),
        cluster_rack([]),
        not(free_(_)),
        shiftwindow(3),
        clearwindow,
        write("Structure complete");
        true.

alt_do_or_dont(FLAG) :-
        equal(FLAG,"yes"),
        plan;
        equal(FLAG,"no").

alt_menu(FLAG) :-
        restore_old_database,
        refresh_structure_graphics,
        refresh_rack_graphics,
        alt_text_refresh,
        readkey(KEY),
        alt_dispatch(KEY,X,Y),
        execute_alternate(X,Y,FLAG).

alt_text_refresh :-
        shiftwindow(3),
        clearwindow,
        write("alternate step:"),
        shiftwindow(2),
        clearwindow,
        cursor(0,0),write("F1: beam"),
        cursor(0,20),write("F2: cluster"),
        cursor(2,0),write("F3: triangle"),
        cursor(2,20),write("F4: main menu").

alt_dispatch(KEY,X,Y) :-
        equal(KEY,fkey(1)),
        beam_menu(X,Y);
        equal(KEY,fkey(2)),
        cluster_menu(X,Y);
        equal(KEY,fkey(3)),
        triangle_menu(X,Y);
        equal(KEY,fkey(4)),
        equal(X,beam_end("dummy","end")),
        equal(Y,mush_end("dummy","end")).


cycle_text_refresh :-
        shiftwindow(2),
        clearwindow,
        cursor(0,0),write("F1: okay"),
        cursor(0,20),write("F2: next").

beam_menu(X,Y) :-
        beam_rack([]),
        shiftwindow(2),
        clearwindow,
        cursor(0,0),write("Sorry, no more beams are available.
        cursor(2,0),write("Hit any key."),
        readkey(_),
        equal(X,beam_end("dummy","end")),
        equal(Y,mush_end("dummy","end"));

        not(free_mush_end(_)),
        shiftwindow(2),
        clearwindow,
        cursor(0,0),write("Sorry, no place to attach a beam.")
        cursor(2,0),write("Hit any key."),
        readkey(_),
        equal(X,beam_end("dummy","end")),
        equal(Y,mush_end("dummy","end"));

        shiftwindow(3),
        clearwindow,
        write("select beam"),
        cycle_text_refresh,
        select_beam(W),
        equal(W,beam(BEAM)),
        equal(X,beam_end(BEAM,"bel")),
        shiftwindow(3),
        clearwindow,
        write("select mushroom end"),
        cycle_text_refresh,
        select_alt_mush_end(Y,W).

select_beam(X) :-
        refresh_all_graphics,
        racked(beam(BEAM)),
        equal(X,beam(BEAM)),
        draw_beam_in_rack(X,1),
        readkey(KEY),
        beam_select_check(KEY,X);
        select_beam(Y),
        equal(X,Y).

beam_select_check(KEY,X) :-
        equal(KEY,fkey(1)),!;
        equal(KEY,fkey(2)),!,
        draw_beam_in_rack(X,3),!,
        fail;
        readkey(KEY2),
        beam_select_check(KEY2,X),
        equal(KEY,KEY2).

select_alt_mush_end(X,Z) :-
        refresh_all_graphics,
        draw_beam_in_rack(Z,1),
        free_mush_end(X),
        not(geometry_not_ok_mush_end(X)),
        not(geometry_not_ok_beam_with_cluster(X,Z)),
        shiftwindow(1),
        draw_mush_end(X,1),
        readkey(KEY),
        mush_end_select_check(KEY,X);
        select_alt_mush_end(Y,Z),
        equal(X,Y).

mush_end_select_check(KEY,X) :-
        equal(KEY,fkey(1)),!;
        equal(KEY,fkey(2)),!,
        draw_mush_end(X,2),!,
        fail;
        readkey(KEY2),
        mush_end_select_check(KEY2,X),
        equal(KEY,KEY2).


refresh_all_graphics :-
        refresh_structure_graphics,
        refresh_rack_graphics.

geometry_not_ok_mush_end(X) :-
        free_beam_end(Y),
        geometry_ok(Y,X).

geometry_not_ok_beam_with_cluster(Y,Z) :-
        equal(Z,beam(BEAM)),
        connected(beam_end(BEAM,"be2"),mush_end(_,_)),
        free_mush_end(X),

*  right_hand_third_beam(X,Y) :- no cluster check  */
        equal(X,mush_end(C1,M1C1)),
        equal(Y,mush_end(C3,M2C3)),
        next_ccw(M1C1,M2C1),
        connected(beam_end(B1,E1B1),mush_end(C1,M2C1)),
        other_end(E1B1,E2B1),
        connected(beam_end(B1,E2B1),mush_end(C2,M1C2)),
        next_ccw(M1C2,M2C2),
        connected(beam_end(B2,E1B2),mush_end(C2,M2C2)),
        other_end(E1B2,E2B2),
        connected(beam_end(B2,E2B2),mush_end(C3,M1C3)),
        next_ccw(M1C3,M2C3);
```

```
          equal(Z,beam(BEAM)),
          connected(beam_end(BEAM,"be2"),mush_end(_,_)),
          free_mush_end(X),

*    left_hand_third_beam(X,Y) :- no cluster check   */
          equal(X,mush_end(C1,M1C1)),
          equal(Y,mush_end(C3,M2C3)),
          next_cw(M1C1,M2C1),
          connected(beam_end(B1,E1B1),mush_end(C1,M2C1)),
          other_end(E1B1,E2B1),
          connected(beam_end(B1,E2B1),mush_end(C2,M1C2)),
          next_cw(M1C2,M2C2),
          connected(beam_end(B2,E1B2),mush_end(C2,M2C2)),
          other_end(E1B2,E2B2),
          connected(beam_end(B2,E2B2),mush_end(C3,M1C3)),
          next_cw(M1C3,M2C3).



     cluster_menu(X,Y) :-
          cluster_rack([]),
          shiftwindow(2),
          clearwindow,
          cursor(0,0),write("Sorry, no more clusters are availab
          cursor(2,0),write("Hit any key."),
          readkey(_),
          equal(X,beam_end("dummy","end")),
          equal(Y,mush_end("dummy","end"));

          shiftwindow(3),
          clearwindow,
          write("select cluster"),
          cycle_text_refresh,
          select_cluster(W),
          equal(W,cluster(CLUSTER)),
          equal(Y,mush_end(CLUSTER,"me1")),
          shiftwindow(3),
          clearwindow,
          write("select beam end"),
          cycle_text_refresh,
          select_alt_beam_end(X,W).

     select_cluster(X) :-
          refresh_all_graphics,
          racked(cluster(CLUSTER)),
          equal(X,cluster(CLUSTER)),
          draw_cluster_in_rack(X,1),
          readkey(KEY),
          cluster_select_check(KEY,X);
          select_cluster(Y),
          equal(X,Y).

     cluster_select_check(KEY,X) :-
          equal(KEY,fkey(1)),!;
          equal(KEY,fkey(2)),!,
          draw_cluster_in_rack(X,2),!,
          fail;
          readkey(KEY2),
          cluster_select_check(KEY2,X),
        equal(KEY,KEY2).

select_alt_beam_end(X,Z) :-
        refresh_all_graphics,
        draw_cluster_in_rack(Z,1),
        get_alt_beam_end(X),
        draw_alt_beam_end(X,1),
        readkey(KEY),
        beam_end_select_check(KEY,X);
        select_alt_beam_end(Y,Z),
        equal(X,Y).

get_alt_beam_end(X) :-
        free_racked_beam_end(X),
        not(racked_cluster_check);
        free_beam_end(X),
        not(geometry_not_ok_beam_end(X)).

geometry_not_ok_beam_end(X) :-
        free_mush_end(Y),
        geometry_ok(X,Y).

racked_cluster_check :-
        not(check_for_triangle),
        cluster_rack([_]),
        free_beam_end(X),
        free_beam_end(Y),
        not(equal(X,Y)),
        second_cluster_geometry_ok(X,Y).

draw_alt_beam_end(X,CO) :-
        free_racked_beam_end(X),
        equal(X,beam_end(BEAM,_)),
        shiftwindow(4),
        draw_beam_in_rack(beam(BEAM),CO);
        free_beam_end(X),
        shiftwindow(1),
        draw_beam_end(X,CO).

beam_end_select_check(KEY,X) :-
        equal(KEY,fkey(1)),!;
        equal(KEY,fkey(2)),!,
        draw_alt_beam_end(X,3),!,
        fail;
        readkey(KEY2),
        beam_end_select_check(KEY2,X),
        equal(KEY,KEY2).


triangle_menu(X,Y) :-
        shiftwindow(2),
        clearwindow,
        not(triangle_possible),
        cursor(0,0),write("Sorry, no triangles are possible.")
        cursor(2,0),write("Hit any key."),
        readkey(_),
        equal(X,beam_end("dummy","end")),
        equal(Y,mush_end("dummy","end"));

        shiftwindow(3),
                clearwindow,
                write("select triangle"),
                cycle_text_refresh,
                select_triangle(X,Y).


        triangle_possible :-
                free_beam_end(X),
                free_mush_end(Y),
                geometry_ok(X,Y).

        select_triangle(X,Y) :-
                refresh_all_graphics,
                free_beam_end(X),
                free_mush_end(Y),
                geometry_ok(X,Y),
                shiftwindow(1),
                draw_beam_end(X,1),
                draw_mush_end(Y,1),
                readkey(KEY),
                triangle_select_check(KEY,X,Y);
                select_triangle(W,V),
                equal(X,W),
                equal(Y,V).

        triangle_select_check(KEY,X,Y) :-
                equal(KEY,fkey(1)),!;
                equal(KEY,fkey(2)),!,
                refresh_all_graphics,!,
'*              draw_mush_end(Y,2),!,*/
                fail;
                readkey(KEY2),
                triangle_select_check(KEY2,X,Y),
                equal(KEY,KEY2).

        execute_alternate(X,Y,FLAG) :-

                equal(X,beam_end("dummy","end")),
                equal(Y,mush_end("dummy","end")),
                equal(FLAG,"no");

*       alt_beam_to_attached_cluster(X,Y,FLAG) :-        */
                equal(X,beam_end(BEAM,BE)),
                equal(Y,mush_end(CLUSTER,_)),
                not(attached(beam(BEAM))),
                other_end(BE,EB),
                equal(V,beam_end(BEAM,EB)),
                not(connected(V,_)),
                not(connected(X,_)),
                attached(cluster(CLUSTER)),
                free_mush_end(Y),
                do_beam_to_attached_cluster(X,Y),
                completed(Z),
                append(Z,[beam_to_attached_cluster(X,Y)],W),
                retract(completed(Z)),
                asserta(temp_retract(completed(Z))),
                assertz(completed(W)),
                asserta(temp_assert(completed(W))),
                check_alt(FLAG);
```

```
/*              alt_cluster_to_attached_beam(X,Y,FLAG) :-          */
                        equal(X,beam_end(BEAM,_)),
                        equal(Y,mush_end(CLUSTER,_)),
                        attached(beam(BEAM)),
                        not(attached(cluster(CLUSTER))),
                        free_beam_end(X),
                        do_cluster_to_attached_beam(X,Y),
                        completed(Z),
                        append(Z,[cluster_to_attached_beam(X,Y)],W),
                        retract(completed(Z)),
                        asserta(temp_retract(completed(Z))),
                        assertz(completed(W)),
                        asserta(temp_assert(completed(W))),
                        check_alt(FLAG);

/*              alt_cluster_to_racked_beam(X,Y,FLAG) :-          */
                        equal(X,beam_end(BEAM,_)),
                        equal(Y,mush_end(CLUSTER,_)),
                        free_racked_beam_end(X),
                        not(attached(beam(BEAM))),
                        not(attached(cluster(CLUSTER))),
                        do_cluster_to_racked_beam(X,Y),
                        completed(Z),
                        append(Z,[cluster_to_racked_beam(X,Y)],W),
                        retract(completed(Z)),
                        asserta(temp_retract(completed(Z))),
                        assertz(completed(W)),
                        asserta(temp_assert(completed(W))),
                        check_alt(FLAG);

/*              alt_beam_with_cluster_to_attached_cluster(X,Y,FLAG) :-*/
                        equal(X,beam_end(BEAM,BE)),
                        not(attached(beam(BEAM))),
                        equal(BE,"bel"),
                        other_end(BE,EB),
                        equal(U,beam_end(BEAM,EB)),
                        connected(U,V),
                        equal(V,mush_end(CLUSTER,_)),
                        attached(cluster(CLUSTER)),
                        do_beam_with_cluster_to_attached_cluster(X,Y),
                        completed(Z),
                        append(Z,[beam_with_cluster_to_attached_cluster(X,Y)]
                        retract(completed(Z)),
                        asserta(temp_retract(completed(Z))),
                        assertz(completed(W)),
                        asserta(temp_assert(completed(W))),
                        check_alt(FLAG);

/*              alt_triangle(X,Y,FLAG) :-                        */
                        equal(X,beam_end(BEAM,_)),
                        equal(Y,mush_end(CLUSTER,_)),
                        attached(beam(BEAM)),attached(cluster(CLUSTER)),
                        free_beam_end(X),
                        free_mush_end(Y),
                        geometry_ok(X,Y),
                        do_triangle(X,Y),
                        completed(Z),
                        append(Z,[triangle(X,Y)],W),
                        retract(completed(Z)),
                        asserta(temp_retract(completed(Z))),
                assertz(completed(W)),
                asserta(temp_assert(completed(W))),
                check_alt(FLAG);

*          alt_default(FLAG) :-                          */
                equal(FLAG,"no"),
                shiftwindow(2),
                clearwindow,
                write("What you suggest is"),nl,
                write("physically impossible."),nl,
                write("Please hit ENTER..."),readln(_).

           check_alt(FLAG) :-
                shiftwindow(3),
                cursor(0,0),
                write("You suggest attaching :"),
                shiftwindow(2),
                clearwindow,
                cursor(0,0),
                write("Is this correct?"),
                cursor(2,0),write("F1: YES"),
                cursor(2,20),write("F2: NO"),
                readkey(KEY),
                alt_check(KEY,FLAG).

           alt_check(KEY,FLAG) :-
                equal(KEY,fkey(1)),
                equal(FLAG,"yes"),
                execute_next_suggested;
                equal(KEY,fkey(2)),
                restore_old_database,
                equal(FLAG,"no");
                equal(KEY,_),
                readkey(KEY2),
                alt_check(KEY2,FLAG2),
                equal(FLAG,FLAG2).

           clear_screen :-
                shiftwindow(1),clearwindow,
                shiftwindow(2),clearwindow,
                shiftwindow(3),clearwindow.

           plan :-
                shiftwindow(2),
                clearwindow,
                shiftwindow(3),
                clearwindow,
                write("planning..."),
                restore_old_database,
                proposed(X),
                retract(proposed(X)),assertz(proposed([])),
                plan_rest_of_assembly,
                restore_old_database.

* Here are the rules which plan assembly sequences */

           plan_rest_of_assembly :-
                true_default_rules,
                get_next_piece(X),!,
                attach(X),!,
        plan_rest_of_assembly,!;
        true.

true_default_rules :-
        first_triangle,
        complete_any_triangles,
        beam_with_cluster,
        complete_any_triangles.

beam_with_cluster :-
        get_beam_with_cluster(W),
        free_mush_end(Z),
        connect_beam_with_cluster_to_attached_mush_end(W,Z);
        true.


first_triangle :-
        check_for_triangle;
        complete_first_triangle.

check_for_triangle :-
        connected(X,Y),
        last_connection_geometry_ok(X,Y).

complete_first_triangle :-
        check_for_triangle,!;
        step_toward_first_triangle,!,
        complete_first_triangle.

step_toward_first_triangle :-
        make_last_connection;
        attach_third_beam;
        attach_third_cluster;
        attach_third_beam_with_cluster;
        attach_second_cluster;
        attach_second_beam_with_cluster;
        attach_first_beam_with_cluster.

make_last_connection :-
        free_beam_end(X),
        free_mush_end(Y),
        last_connection_geometry_ok(X,Y),
        complete_triangle(X,Y).

attach_third_beam :-
        free_mush_end(X),
        free_mush_end(Y),
        not(equal(X,Y)),
        third_beam_geometry_ok(X,Y),
        get_next_beam(Z),
        connect_beam_to_attached_mush_end(Z,X).

attach_third_cluster :-
        free_beam_end(X),
        free_beam_end(Y),
        not(equal(X,Y)),
        third_cluster_geometry_ok(X,Y),
        get_next_cluster(Z),
        connect_cluster_to_attached_beam_end(Z,X).
```

```
           attach_third_beam_with_cluster :-
                   free_beam_end(X),
                   free_mush_end(Y),
                   third_beam_with_cluster_geometry_ok(X,Y),
                   get_beam_with_cluster(W),
                   connect_beam_with_cluster_to_attached_mush_end(W,Y).

           attach_second_cluster :-
                   free_beam_end(X),
                   free_beam_end(Y),
                   not(equal(X,Y)),
                   second_cluster_geometry_ok(X,Y),
                   get_next_cluster(Z),
                   connect_cluster_to_attached_beam_end(Z,X).

           attach_second_beam_with_cluster :-
                   free_mush_end(X),
                   second_beam_with_cluster_geometry_ok(X),
                   get_beam_with_cluster(W),
                   connect_beam_with_cluster_to_attached_mush_end(W,X).

           attach_first_beam_with_cluster :-
                   free_mush_end(X),
                   first_beam_with_cluster_geometry_ok(X),
                   get_beam_with_cluster(W),
                   connect_beam_with_cluster_to_attached_mush_end(W,X).


           get_beam_with_cluster(W) :-
                   attached(cluster(CLUSTER)),
                   connected(beam_end(BEAM,BE),mush_end(CLUSTER,ME)),
                   not(attached(beam(BEAM))),
                   equal(W,beam(BEAM)),
                   remove_from_beam_rack(W);
                   get_next_cluster(Y),
                   free_racked_beam_end(Z),
                   connect_cluster_to_racked_beam_end(Y,Z),
                   get_next_beam(W).

           last_connection_geometry_ok(X,Y) :-
*                  right_hand_last_connection(X,Y);
                   left_hand_last_connection(X,Y).

           right_hand_last_connection(X,Y) :-      */
                   equal(X,beam_end(B1,E1B1)),
                   equal(Y,mush_end(C1,M1C1)),
                   other_end(E1B1,E2B1),
                   connected(beam_end(B1,E2B1),mush_end(C2,M1C2)),
                   next_ccw(M1C2,M2C2),
                   connected(beam_end(B2,E1B2),mush_end(C2,M2C2)),
                   other_end(E1B2,E2B2),
                   connected(beam_end(B2,E2B2),mush_end(C3,M1C3)),
                   next_ccw(M1C3,M2C3),
                   connected(beam_end(B3,E1B3),mush_end(C3,M2C3)),
                   other_end(E1B3,E2B3),
                   connected(beam_end(B3,E2B3),mush_end(C1,M2C1)),
                   next_ccw(M2C1,M1C1),
                   triple_cluster_check(C1,C2,C3);

*          left_hand_last_connection(X,Y) :-       */
```

```
                    equal(X,beam_end(B1,E1B1)),
                    equal(Y,mush_end(C1,M1C1)),
                    other_end(E1B1,E2B1),
                    connected(beam_end(B1,E2B1),mush_end(C2,M1C2)),
                    next_cw(M1C2,M2C2),
                    connected(beam_end(B2,E1B2),mush_end(C2,M2C2)),
                    other_end(E1B2,E2B2),
                    connected(beam_end(B2,E2B2),mush_end(C3,M1C3)),
                    next_cw(M1C3,M2C3),
                    connected(beam_end(B3,E1B3),mush_end(C3,M2C3)),
                    other_end(E1B3,E2B3),
                    connected(beam_end(B3,E2B3),mush_end(C1,M2C1)),
                    next_cw(M2C1,M1C1),
                    triple_cluster_check(C1,C2,C3).

            third_beam_geometry_ok(X,Y) :-
'*                  right_hand_third_beam(X,Y);
                    left_hand_third_beam(X,Y).

            right_hand_third_beam(X,Y) :-   */
                    equal(X,mush_end(C1,M1C1)),
                    equal(Y,mush_end(C3,M2C3)),
                    next_ccw(M1C1,M2C1),
                    connected(beam_end(B1,E1B1),mush_end(C1,M2C1)),
                    other_end(E1B1,E2B1),
                    connected(beam_end(B1,E2B1),mush_end(C2,M1C2)),
                    next_ccw(M1C2,M2C2),
                    connected(beam_end(B2,E1B2),mush_end(C2,M2C2)),
                    other_end(E1B2,E2B2),
                    connected(beam_end(B2,E2B2),mush_end(C3,M1C3)),
                    next_ccw(M1C3,M2C3),
                    triple_cluster_check(C1,C2,C3);

*           left_hand_third_beam(X,Y) :-    */
                    equal(X,mush_end(C1,M1C1)),
                    equal(Y,mush_end(C3,M2C3)),
                    next_cw(M1C1,M2C1),
                    connected(beam_end(B1,E1B1),mush_end(C1,M2C1)),
                    other_end(E1B1,E2B1),
                    connected(beam_end(B1,E2B1),mush_end(C2,M1C2)),
                    next_cw(M1C2,M2C2),
                    connected(beam_end(B2,E1B2),mush_end(C2,M2C2)),
                    other_end(E1B2,E2B2),
                    connected(beam_end(B2,E2B2),mush_end(C3,M1C3)),
                    next_cw(M1C3,M2C3),
                    triple_cluster_check(C1,C2,C3).

            third_cluster_geometry_ok(X,Y) :-
*                   right_hand_third_cluster(X,Y);
                    left_hand_third_cluster(X,Y).

            right_hand_third_cluster(X,Y) :-        */
                    equal(X,beam_end(B1,E1B1)),
                    equal(Y,beam_end(B3,E2B3)),
                    other_end(E1B1,E2B1),
                    connected(beam_end(B1,E2B1),mush_end(C1,M1C1)),
                    next_ccw(M1C1,M2C1),
                    connected(beam_end(B2,E1B2),mush_end(C1,M2C1)),
                    other_end(E1B2,E2B2),
                    connected(beam_end(B2,E2B2),mush_end(C2,M1C2)),
                    next_ccw(M1C2,M2C2),
                    connected(beam_end(B3,E1B3),mush_end(C2,M2C2)),
                    other_end(E1B3,E2B3),
                    double_cluster_check(C1,C2);

*           left_hand_third_cluster(X,Y) :- */
                    equal(X,beam_end(B1,E1B1)),
                    equal(Y,beam_end(B3,E2B3)),
                    other_end(E1B1,E2B1),
                    connected(beam_end(B1,E2B1),mush_end(C1,M1C1)),
                    next_cw(M1C1,M2C1),
                    connected(beam_end(B2,E1B2),mush_end(C1,M2C1)),
                    other_end(E1B2,E2B2),
                    connected(beam_end(B2,E2B2),mush_end(C2,M1C2)),
                    next_cw(M1C2,M2C2),
                    connected(beam_end(B3,E1B3),mush_end(C2,M2C2)),
                    other_end(E1B3,E2B3),
                    double_cluster_check(C1,C2).

            third_beam_with_cluster_geometry_ok(X,Y) :-
*                   right_hand_third_beam_with_cluster(X,Y);
                    left_hand_third_beam_with_cluster(X,Y).

            right_hand_third_beam_with_cluster(X,Y) :-      */
                    equal(X,beam_end(B1,E1B1)),
                    equal(Y,mush_end(C2,M2C2)),
                    other_end(E1B1,E2B1),
                    connected(beam_end(B1,E2B1),mush_end(C1,M1C1)),
                    next_ccw(M1C1,M2C1),
                    connected(beam_end(B2,E1B2),mush_end(C1,M2C1)),
                    other_end(E1B2,E2B2),
                    connected(beam_end(B2,E2B2),mush_end(C2,M1C2)),
                    next_ccw(M1C2,M2C2),
                    double_cluster_check(C1,C2);

*           left_hand_third_beam_with_cluster(X,Y) :-       */
                    equal(X,beam_end(B1,E1B1)),
                    equal(Y,mush_end(C2,M2C2)),
                    other_end(E1B1,E2B1),
                    connected(beam_end(B1,E2B1),mush_end(C1,M1C1)),
                    next_cw(M1C1,M2C1),
                    connected(beam_end(B2,E1B2),mush_end(C1,M2C1)),
                    other_end(E1B2,E2B2),
                    connected(beam_end(B2,E2B2),mush_end(C2,M1C2)),
                    next_cw(M1C2,M2C2),
                    double_cluster_check(C1,C2).

            second_cluster_geometry_ok(X,Y) :-
*                   right_hand_second_cluster(X,Y);
                    left_hand_second_cluster(X,Y).

            right_hand_second_cluster(X,Y) :-       */
                    equal(X,beam_end(B1,E1B1)),
                    equal(Y,beam_end(B2,E2B2)),
                    other_end(E1B1,E2B1),
                    connected(beam_end(B1,E2B1),mush_end(C1,M1C1)),
                    next_ccw(M1C1,M2C1),
                    connected(beam_end(B2,E1B2),mush_end(C1,M2C1)),
                    other_end(E1B2,E2B2),
                    single_cluster_check(C1);
```

```
*               left_hand_second_cluster(X,Y) :-          */
                        equal(X,beam_end(B1,E1B1)),
                        equal(Y,beam_end(B2,E2B2)),
                        other_end(E1B1,E2B1),
                        connected(beam_end(B1,E2B1),mush_end(C1,M1C1)),
                        next_cw(M1C1,M2C1),
                        connected(beam_end(B2,E1B2),mush_end(C1,M2C1)),
                        other_end(E1B2,E2B2),
                        single_cluster_check(C1).

                second_beam_with_cluster_geometry_ok(X) :-
                        equal(X,mush_end(C1,M1C1)),
                        single_cluster_check(C1),
                        next_cw(M1C1,M2C1),
                        next_cw(M2C1,M3C1),
                        free_mush_end(mush_end(C1,M2C1)),
                        connected(_,mush_end(C1,M3C1)).

                first_beam_with_cluster_geometry_ok(X) :-
                        equal(X,mush_end(C1,M1C1)),
                        single_cluster_check(C1),
                        next_cw(M1C1,M2C1),
                        next_cw(M2C1,M3C1),
                        free_mush_end(mush_end(C1,M1C1)),
                        free_mush_end(mush_end(C1,M2C1)),
                        free_mush_end(mush_end(C1,M3C1)).

                triple_cluster_check(X,Y,Z) :-
                        equal(X,"base_cluster");
                        equal(Y,"base_cluster");
                        equal(Z,"base_cluster").

                double_cluster_check(X,Y) :-
                        equal(X,"base_cluster");
                        equal(Y,"base_cluster").

                single_cluster_check(X) :-
                        equal(X,"base_cluster").


                repeat.
                repeat :- repeat.

                get_next_piece(X) :-
                        get_next_cluster(X);
                        get_next_beam(X).

                get_next_beam(X) :-
                        not(beam_rack([])),
                        free_mush_end(_),
                        beam_rack([X|Y]),
                        retract(beam_rack([X|Y])),
                        asserta(temp_retract(beam_rack([X|Y]))),
                        assertz(beam_rack(Y)),
                        asserta(temp_assert(beam_rack(Y))).

                get_next_cluster(X) :-
                        not(cluster_rack([])),
                        free_beam_end_test(_),
          cluster_rack([X|Y]),
          retract(cluster_rack([X|Y])),
          asserta(temp_retract(cluster_rack([X|Y]))),
          assertz(cluster_rack(Y)),
          asserta(temp_assert(cluster_rack(Y))).

  free_beam_end_test(X) :-
          free_beam_end(X);
          free_racked_beam_end(X).

  attach(X) :-
          attach_beam(X);
          attach_cluster(X).

  attach_beam(X) :-
          equal(X,beam(_)),
          free_mush_end(Y),
          connect_beam_to_attached_mush_end(X,Y).

  attach_cluster(X) :-
          equal(X,cluster(_)),
          free_beam_end(Y),
          connect_cluster_to_attached_beam_end(X,Y).

  connect_beam_to_attached_mush_end(X,Y) :-
          equal(X,beam(Z)),
          retract(free_racked_beam_end(beam_end(Z,"be2"))),
          asserta(temp_retract(free_racked_beam_end
                  (beam_end(Z,"be2")))),
          retract(free_mush_end(Y)),
          asserta(temp_retract(free_mush_end(Y))),
          assertz(connected(beam_end(Z,"be1"),Y)),
          asserta(temp_assert(connected(beam_end(Z,"be1"),Y))),
          assertz(free_beam_end(beam_end(Z,"be2"))),
          asserta(temp_assert(free_beam_end(beam_end(Z,"be2"))))
          assertz(attached(X)),
          asserta(temp_assert(attached(X))),
          retract(racked(X)),
          asserta(temp_retract(racked(X))),
          add_to_prop_seq(beam_to_attached_cluster(beam_end(Z,
                 "be1"),Y)).

  connect_cluster_to_attached_beam_end(X,Y) :-
          equal(X,cluster(Z)),
          retract(free_beam_end(Y)),
          asserta(temp_retract(free_beam_end(Y))),
          assertz(connected(Y,mush_end(Z,"me1"))),
          asserta(temp_assert(connected(Y,mush_end(Z,"me1")))),
          assertz(free_mush_end(mush_end(Z,"me2"))),
          asserta(temp_assert(free_mush_end(mush_end(Z,"me2"))))
          assertz(free_mush_end(mush_end(Z,"me3"))),
          asserta(temp_assert(free_mush_end(mush_end(Z,"me3"))))
          assertz(attached(X)),
          asserta(temp_assert(attached(X))),
          retract(racked(X)),
          asserta(temp_retract(racked(X))),
          add_to_prop_seq(cluster_to_attached_beam(Y,
                 mush_end(Z,"me1"))).

  connect_cluster_to_racked_beam_end(X,Y) :-
```

```
          equal(X,cluster(Z)),
          retract(free_racked_beam_end(Y)),
          asserta(temp_retract(free_racked_beam_end(Y))),
          assertz(connected(Y,mush_end(Z,"mel"))),
          asserta(temp_assert(connected(Y,mush_end(Z,"mel")))),
          assertz(attached(X)),
          asserta(temp_assert(attached(X))),
          retract(racked(X)),
          asserta(temp_retract(racked(X))),
          add_to_prop_seq(cluster_to_racked_beam(Y,
                  mush_end(Z,"mel"))).

connect_beam_with_cluster_to_attached_mush_end(X,Y) :-
          equal(X,beam(Z)),
          retract(free_mush_end(Y)),
          asserta(temp_retract(free_mush_end(Y))),
          assertz(connected(beam_end(Z,"bel"),Y)),
          asserta(temp_assert(connected(beam_end(Z,"bel"),Y))),
          assertz(attached(X)),
          asserta(temp_assert(attached(X))),
          retract(racked(X)),
          asserta(temp_retract(racked(X))),
          equal(W,beam_end(Z,"be2")),
          connected(W,V),
          equal(V,mush_end(U,ME1)),
          next_cw(ME1,ME2),
          next_cw(ME2,ME3),
          assertz(free_mush_end(mush_end(U,ME2))),
          asserta(temp_assert(free_mush_end(mush_end(U,ME2)))),
          assertz(free_mush_end(mush_end(U,ME3))),
          asserta(temp_assert(free_mush_end(mush_end(U,ME3)))),
          add_to_prop_seq(beam_with_cluster_to_attached_cluster
                  (beam_end(Z,"bel"),Y)).

complete_triangle(X,Y) :-
          retract(free_beam_end(X)),
          asserta(temp_retract(free_beam_end(X))),
          retract(free_mush_end(Y)),
          asserta(temp_retract(free_mush_end(Y))),
          assertz(connected(X,Y)),
          asserta(temp_assert(connected(X,Y))),
          add_to_prop_seq(triangle(X,Y)).

append([],List,List).

append([X|L1],List2,[X|L3]) :-
          append(L1,List2,L3).

add_to_prop_seq(X) :-
          proposed(Y),
          append(Y,[X],Z),
          retract(proposed(Y)),
          assertz(proposed(Z)).

add_to_comp_seq(X) :-
          completed(Y),
          append(Y,[X],Z),
          retract(completed(Y)),
          assertz(completed(Z)).
          erase_temp_asserts :-
                  retract(temp_assert(X)),
                  retract(X),
                  erase_temp_asserts,!;
                  true.

          restore_old_database :-
                  restore_temp_retracts,
                  erase_temp_asserts.

          restore_temp_retracts :-
                  retract(temp_retract(Y)),
                  asserta(Y),
                  restore_temp_retracts,!;
                  true.

          complete_any_triangles :-
                  make_all_possible_connections(X,Y);
                  true.

          make_all_possible_connections(X,Y) :-
                  free_beam_end(X),
                  free_mush_end(Y),
                  geometry_ok(X,Y),
                  complete_triangle(X,Y).

          geometry_ok(X,Y) :-            /* X=beam_end, Y=mush_end */
*                 right_hand_rule(X,Y);
                  left_hand_rule(X,Y).

          right_hand_rule(X,Y) :- */
                  equal(X,beam_end(B1,E1B1)),
                  equal(Y,mush_end(C1,M1C1)),
                  other_end(E1B1,E2B1),
                  connected(beam_end(B1,E2B1),mush_end(C2,M1C2)),
                  next_ccw(M1C2,M2C2),
                  connected(beam_end(B2,E1B2),mush_end(C2,M2C2)),
                  other_end(E1B2,E2B2),
                  connected(beam_end(B2,E2B2),mush_end(C3,M1C3)),
                  next_ccw(M1C3,M2C3),
                  connected(beam_end(B3,E1B3),mush_end(C3,M2C3)),
                  other_end(E1B3,E2B3),
                  connected(beam_end(B3,E2B3),mush_end(C1,M2C1)),
                  next_ccw(M2C1,M1C1);

*         left_hand_rule(X,Y) :-  */
                  equal(X,beam_end(B1,E1B1)),
                  equal(Y,mush_end(C1,M1C1)),
                  other_end(E1B1,E2B1),
                  connected(beam_end(B1,E2B1),mush_end(C2,M1C2)),
                  next_cw(M1C2,M2C2),
                  connected(beam_end(B2,E1B2),mush_end(C2,M2C2)),
                  other_end(E1B2,E2B2),
                  connected(beam_end(B2,E2B2),mush_end(C3,M1C3)),
                  next_cw(M1C3,M2C3),
                  connected(beam_end(B3,E1B3),mush_end(C3,M2C3)),
                  other_end(E1B3,E2B3),
                  connected(beam_end(B3,E2B3),mush_end(C1,M2C1)),
                  next_cw(M2C1,M1C1).
```

```
        next_cw("me1","me2").
        next_cw("me2","me3").
        next_cw("me3","me1").
        next_ccw("me1","me3").
        next_ccw("me3","me2").
        next_ccw("me2","me1").
        other_end("be1","be2").
        other_end("be2","be1").

* Attaching things together */

        present_next_suggested :-
                proposed([]),true;
                proposed([X|Y]),
                execute_and_mark(X),
                retract(proposed([X|Y])),
                asserta(temp_retract(proposed([X|Y]))),
                assertz(proposed(Y)),
                asserta(temp_assert(proposed(Y))),

                completed(Z),
                append(Z,[X],W),
                retract(completed(Z)),
                asserta(temp_retract(completed(Z))),
                assertz(completed(W)),
                asserta(temp_assert(completed(W))).

        execute_next_suggested :-
                shiftwindow(2),
                clearwindow,
                shiftwindow(3),
                clearwindow,
                keep_current_database.

        keep_current_database :-
                unmark_temp_retracts,
                unmark_temp_asserts.

        unmark_temp_retracts :-
                retract(temp_retract(_)),
                unmark_temp_retracts;
                true.

        unmark_temp_asserts :-
                retract(temp_assert(_)),
                unmark_temp_asserts;
                true.

        execute_and_mark(X) :-
                equal(X,beam_to_attached_cluster(Y,Z)),
                        do_beam_to_attached_cluster(Y,Z);
                equal(X,cluster_to_attached_beam(Y,Z)),
                        do_cluster_to_attached_beam(Y,Z);
                equal(X,cluster_to_racked_beam(Y,Z)),
                        do_cluster_to_racked_beam(Y,Z);
                equal(X,beam_with_cluster_to_attached_cluster(Y,Z)),
                        do_beam_with_cluster_to_attached_cluster(Y,Z);
                equal(X,triangle(Y,Z)),
                        do_triangle(Y,Z).
```

```
do_beam_to_attached_cluster(X,Y) :-
        equal(X,beam_end(BEAM,BE)),
        equal(Y,mush_end(_,_)),
        retract(free_mush_end(Y)),
        asserta(temp_retract(free_mush_end(Y))),
        assertz(connected(X,Y)),
        asserta(temp_assert(connected(X,Y))),
        other_end(BE,EB),
        assertz(free_beam_end(beam_end(BEAM,EB))),
        asserta(temp_assert(free_beam_end(beam_end(BEAM,EB))))
        assertz(attached(beam(BEAM))),
        asserta(temp_assert(attached(beam(BEAM)))),
        retract(racked(beam(BEAM))),
        asserta(temp_retract(racked(beam(BEAM)))),
        shiftwindow(3),
        clearwindow,
        write("I suggest attaching"),nl,
        write("a beam to the structure."),
        shiftwindow(4),
        draw_beam_in_rack(beam(BEAM),1),
        new_beam_location(X,Y),
        shiftwindow(1),
        draw_beam_with_free_end(beam(BEAM),1),
        retract(free_racked_beam_end(beam_end(BEAM,"be2"))),
        asserta(temp_retract(free_racked_beam_end
                (beam_end(BEAM,"be2")))),
        remove_from_beam_rack(beam(BEAM)).

do_cluster_to_attached_beam(X,Y) :-
        equal(X,beam_end(_,_)),
        equal(Y,mush_end(CLUSTER,ME)),
        retract(free_beam_end(X)),
        asserta(temp_retract(free_beam_end(X))),
        assertz(connected(X,Y)),
        asserta(temp_assert(connected(X,Y))),
        next_cw(ME,ME2),next_cw(ME2,ME3),
        assertz(free_mush_end(mush_end(CLUSTER,ME2))),
        asserta(temp_assert(free_mush_end(mush_end(CLUSTER,ME2)
        assertz(free_mush_end(mush_end(CLUSTER,ME3))),
        asserta(temp_assert(free_mush_end(mush_end(CLUSTER,ME3)
        assertz(attached(cluster(CLUSTER))),
        asserta(temp_assert(attached(cluster(CLUSTER)))),
        retract(racked(cluster(CLUSTER))),
        asserta(temp_retract(racked(cluster(CLUSTER)))),
        shiftwindow(3),
        clearwindow,
        write("I suggest attaching"),nl,
        write("a cluster to the structure"),
        shiftwindow(4),
        draw_cluster_in_rack(cluster(CLUSTER),1),
        new_cluster_location(X,Y),
        shiftwindow(1),
        draw_beam_end(X,1),
        draw_cluster(cluster(CLUSTER),1),
        remove_from_cluster_rack(cluster(CLUSTER)).

do_cluster_to_racked_beam(X,Y) :-
        equal(X,beam_end(_,_/*BEAM,BE*/)),
        equal(Y,mush_end(CLUSTER,_/*ME*/)),
        retract(free_racked_beam_end(X)),
        asserta(temp_retract(free_racked_beam_end(X))),
        assertz(connected(X,Y)),
        asserta(temp_assert(connected(X,Y))),
        assertz(attached(cluster(CLUSTER))),
        asserta(temp_assert(attached(cluster(CLUSTER)))),
        retract(racked(cluster(CLUSTER))),
        asserta(temp_retract(racked(cluster(CLUSTER)))),
        shiftwindow(3),
        clearwindow,
        write("I suggest attaching"),nl,
        write("a cluster to a racked beam."),
        shiftwindow(4),
        draw_cluster_in_rack(cluster(CLUSTER),1),
        new_cluster_location_in_rack(X,Y),
        draw_cluster_attached_to_beam_in_rack
                (cluster(CLUSTER),1),
        remove_from_cluster_rack(cluster(CLUSTER)).

do_beam_with_cluster_to_attached_cluster(X,Y) :-
        equal(X,beam_end(BEAM,BE)),
        equal(Y,mush_end(_,_)),
        retract(free_mush_end(Y)),
        asserta(temp_retract(free_mush_end(Y))),
        assertz(connected(X,Y)),
        asserta(temp_assert(connected(X,Y))),
        assertz(attached(beam(BEAM))),
        asserta(temp_assert(attached(beam(BEAM)))),
        retract(racked(beam(BEAM))),
        asserta(temp_retract(racked(beam(BEAM)))),
        other_end(BE,EB),
        equal(W,beam_end(BEAM,EB)),
        connected(W,Z),
        equal(Z,mush_end(CLUSTER2,C2M1)),
        next_cw(C2M1,C2M2),
        next_cw(C2M2,C2M3),
        assertz(free_mush_end(mush_end(CLUSTER2,C2M2))),
        asserta(temp_assert(free_mush_end(mush_end
                (CLUSTER2,C2M2)))),
        assertz(free_mush_end(mush_end(CLUSTER2,C2M3))),
        asserta(temp_assert(free_mush_end(mush_end
                (CLUSTER2,C2M3)))),
        shiftwindow(3),
        clearwindow,
        write("I suggest attaching "),nl,
        write("a beam with a cluster to the structure."),
        shiftwindow(4),
        draw_beam_in_rack(beam(BEAM),1),
        draw_cluster_attached_to_beam_in_rack
                (cluster(CLUSTER2),1),
        new_beam_location(X,Y),
        new_cluster_location(W,Z),
        shiftwindow(1),
        draw_beam(beam(BEAM),1),
        draw_cluster(cluster(CLUSTER2),1),
        remove_from_beam_rack(beam(BEAM)).


do_triangle(X,Y) :-
        equal(X,beam_end(_,_)),
        equal(Y,mush_end(_,_)),
        retract(free_beam_end(X)),
        asserta(temp_retract(free_beam_end(X))),
        retract(free_mush_end(Y)),
        asserta(temp_retract(free_mush_end(Y))),
        assertz(connected(X,Y)),
        asserta(temp_assert(connected(X,Y))),
        shiftwindow(3),
        clearwindow,
        write("I suggest completing"),nl,
        write("the triangle."),
        shiftwindow(1),
        draw_beam_end(X,1),
        draw_mush_end(Y,1).

remove_from_beam_rack(X) :-
        not(beam_rack([])),
        beam_rack(Y),
        retract(dummy_beam_one(_)),
        assertz(dummy_beam_one(Y)),
        search_beam(X),
        dummy_beam_two(W),
        retract(beam_rack(Y)),
        asserta(temp_retract(beam_rack(Y))),
        assertz(beam_rack(W)),
        asserta(temp_assert(beam_rack(W))),
        retract(dummy_beam_one(_)),
        assertz(dummy_beam_one([])),
        retract(dummy_beam_two(W)),
        assertz(dummy_beam_two([])).

search_beam(X) :-
        dummy_beam_one([Y|Z]),
        retract(dummy_beam_one(_)),
        assertz(dummy_beam_one(Z)),
        check_beam(Y,X),
        search_beam(X);
        true.

check_beam(X,Y) :-
        equal(X,Y),true;
        dummy_beam_two(W),
        append(W,[X],Z),
        retract(dummy_beam_two(W)),
        assertz(dummy_beam_two(Z)).

remove_from_cluster_rack(X) :-
        not(cluster_rack([])),
        cluster_rack(Y),
        retract(dummy_cluster_one(_)),
        assertz(dummy_cluster_one(Y)),
        search_cluster(X),
        dummy_cluster_two(W),
        retract(cluster_rack(Y)),
        asserta(temp_retract(cluster_rack(Y))),
        assertz(cluster_rack(W)),
        asserta(temp_assert(cluster_rack(W))),
        retract(dummy_cluster_one(_)),
        assertz(dummy_cluster_one([])),
        retract(dummy_cluster_two(W)),
        assertz(dummy_cluster_two([])).

search_cluster(X) :-
        dummy_cluster_one([Y|Z]),
        retract(dummy_cluster_one(_)),
        assertz(dummy_cluster_one(Z)),
        check_cluster(Y,X),
        search_cluster(X);
        true.

check_cluster(X,Y) :-
        equal(X,Y),true;
        dummy_cluster_two(W),
        append(W,[X],Z),
        retract(dummy_cluster_two(W)),
        assertz(dummy_cluster_two(Z)).

locate_fixed_cluster(CLUSTER,VCOORD) :-
        mush_end_length(MEL),
        equal([XV,YV,ZV],VCOORD),
        X1=XV-MEL*cos(0.6154797),
        X2=XV-MEL*cos(0.6154797),
        X3=XV-MEL*cos(0.6154797),
        Y1=YV,
        Y2=YV-(MEL/2),
        Y3=YV+(MEL/2),
        Z1=ZV+MEL*sin(0.6154797),
        Z2=ZV-(MEL/2)*0.577350269,
        Z3=ZV-(MEL/2)*0.577350269,
        equal(E1COORD,[X1,Y1,Z1]),
        equal(E2COORD,[X2,Y2,Z2]),
        equal(E3COORD,[X3,Y3,Z3]),
        assertz(location(cluster(CLUSTER),[VCOORD,
                E2COORD,E3COORD,E1COORD])),
        assertz(temp_assert(location(cluster(CLUSTER),[VCOORD,
                E2COORD,E3COORD,E1COORD]))).

locate_fixed_beam(BEAM,ECOORD) :-
        equal([XE,YE,ZE],ECOORD),
        beam_end_length(BEL),
        beam_length(BL),
        YF=YE+BEL,
        YS=YF+BL,
        YT=YS+BEL,
        equal(FCOORD,[XE,YF,ZE]),
        equal(SCOORD,[XE,YS,ZE]),
        equal(TCOORD,[XE,YT,ZE]),
        assertz(location(beam(BEAM),[ECOORD,FCOORD,
                SCOORD,TCOORD])),
        asserta(temp_assert(location(beam(BEAM),[ECOORD,
                FCOORD,SCOORD,TCOORD]))).


new_beam_location(X,Y) :-
        equal(X,beam_end(BEAM,BE)),
        equal(Y,mush_end(CLUSTER,ME)),
        mush_end_vector(CLUSTER,ME,VCOORD,ECOORD),
        equal(VCOORD,[XV,YV,ZV]),
        equal(ECOORD,[XE,YE,ZE]),
        XO=XE-XV,YO=YE-YV,ZO=ZE-ZV,
                first_beam_ratio(FBR),
                second_beam_ratio(SBR),
                third_beam_ratio(TBR),
                XF=FBR*XO+XV,YF=FBR*YO+YV,ZF=FBR*ZO+ZV,
                XS=SBR*XO+XV,YS=SBR*YO+YV,ZS=SBR*ZO+ZV,
                XT=TBR*XO+XV,YT=TBR*YO+YV,ZT=TBR*ZO+ZV,
                FCOORD=[XF,YF,ZF],SCOORD=[XS,YS,ZS],
                TCOORD=[XT,YT,ZT],
                retract(location(beam(BEAM),POS)),
                asserta(temp_retract(location(beam(BEAM),POS))),
                assert_beam_location(BEAM,BE,ECOORD,FCOORD,
                        SCOORD,TCOORD).

        assert_beam_location(BEAM,BE,A,B,C,D) :-
                equal(BE,"be1"),
                assertz(location(beam(BEAM),[A,B,C,D])),
                asserta(temp_assert(location(beam(BEAM),[A,B,C,D])));
                equal(BE,"be2"),
                assertz(location(beam(BEAM),[D,C,B,A])),
                asserta(temp_assert(location(beam(BEAM),[D,C,B,A]))).

        mush_end_vector(CLUSTER,ME,VCOORD,ECOORD) :-
                equal(ME,"me1"),
                location(cluster(CLUSTER),[VCOORD,ECOORD,_,_]);
                equal(ME,"me2"),
                location(cluster(CLUSTER),[VCOORD,_,ECOORD,_]);
                equal(ME,"me3"),
                location(cluster(CLUSTER),[VCOORD,_,_,ECOORD]).

        new_cluster_location(X,Y) :-
*               from_upright(X,Y);
                from_right_cross(X,Y);
                from_left_cross(X,Y).

        from_upright(X,Y) :-    */
                equal(X,beam_end(BEAM,BE)),
                equal(Y,mush_end(CLUSTER,ME)),
                beam_end_vector(BEAM,BE,BCOORD,ECOORD),
                equal(BCOORD,[XB,YB,ZB]),
                equal(ECOORD,[XE,YE,ZE]),
                beam_end_length(BEL),
                mush_end_length(MEL),
                center([/*XC*/_,YC,ZC]),
                XO=XE-XB,YO=YE-YB,ZO=ZE-ZB,
                C1 = (MEL+BEL)/BEL,
                XV=C1*XO+XB, YV=C1*YO+YB, ZV=C1*ZO+ZB,
                /*XCP=XC-XV,*/YCP=YC-YV,ZCP=ZC-ZV,
                /*XEP=XE-XV,*/YEP=YE-YV,ZEP=ZE-ZV,
                XK=YCP*ZEP-ZCP*YEP,
                XK>-0.00001,XK<0.00001,
                MAG=sqrt(YCP*YCP+ZCP*ZCP),
                C2=MEL/MAG,
                YT=C2*YCP,ZT=C2*ZCP,
                XE2P=0, XE3P=0,
                YE2P=YT*0.8660+ZT*(-0.5),
                ZE2P=YT*0.5+ZT*0.8660,
                YE3P=YT*0.8660+ZT*0.5,
                ZE3P=-YT*0.5+ZT*0.8660,
                ZE2 = ZE2P+ZV, ZE3 = ZE3P+ZV, YE2 = YE2P+YV,
                YE3 = YE3P+YV, XE2 = XE2P+XV, XE3 = XE3P+XV,
                VCOORD=[XV,YV,ZV],E2COORD=[XE2,YE2,ZE2],
                E3COORD=[XE3,YE3,ZE3],
                retract(location(cluster(CLUSTER),POS)),
                asserta(temp_retract(location(cluster(CLUSTER),POS))),
                assert_cluster_location(CLUSTER,ME,VCOORD,ECOORD,
                        E2COORD,E3COORD);


*           from_right_cross(X,Y) :-         */
                equal(X,beam_end(BEAM,BE)),
                equal(Y,mush_end(CLUSTER,ME)),
                beam_end_vector(BEAM,BE,BCOORD,ECOORD),
                equal(BCOORD,[XB,YB,ZB]),
                equal(ECOORD,[XE,YE,ZE]),
                beam_end_length(BEL),
                mush_end_length(MEL),
                center([/*XC*/_,YC,ZC]),
                XO=XE-XB,YO=YE-YB,ZO=ZE-ZB,
                C1 = (MEL+BEL)/BEL,
                XV=C1*XO+XB, YV=C1*YO+YB, ZV=C1*ZO+ZB,
                /*XCP=XC-XV,*/YCP=YC-YV,ZCP=ZC-ZV,
                XEP=XE-XV,YEP=YE-YV,ZEP=ZE-ZV,
                XK=YCP*ZEP-ZCP*YEP,
                XK<0, XEP=0,
                MAG=sqrt(YCP*YCP+ZCP*ZCP),
                C2=MEL/MAG,
                YT=C2*YCP,ZT=C2*ZCP,
                ALPHA=0.955316618,
                MAG2=sqrt(YT*YT+ZT*ZT),
                C3=MEL*cos(ALPHA)/MAG2,
                XE2P=MEL*sin(ALPHA),
                YE2P=C3*YT,
                ZE2P=C3*ZT,
                XE3P=0,
                YE3P=YT*0.8660-ZT*0.5,
                ZE3P=YT*0.5+ZT*0.8660,
                ZE2 = ZE2P+ZV, ZE3 = ZE3P+ZV, YE2 = YE2P+YV,
                YE3 = YE3P+YV, XE2 = XE2P+XV, XE3 = XE3P+XV,
                VCOORD=[XV,YV,ZV],E2COORD=[XE2,YE2,ZE2],
                E3COORD=[XE3,YE3,ZE3],
                retract(location(cluster(CLUSTER),POS)),
                asserta(temp_retract(location(cluster(CLUSTER),POS))),
                assert_cluster_location(CLUSTER,ME,VCOORD,ECOORD,
                        E2COORD,E3COORD);

*           from_left_cross(X,Y) :- */
                equal(X,beam_end(BEAM,BE)),
                equal(Y,mush_end(CLUSTER,ME)),
                beam_end_vector(BEAM,BE,BCOORD,ECOORD),
                equal(BCOORD,[XB,YB,ZB]),
                equal(ECOORD,[XE,YE,ZE]),
                beam_end_length(BEL),
                mush_end_length(MEL),
                center([/*XC*/_,YC,ZC]),
                XO=XE-XB,YO=YE-YB,ZO=ZE-ZB,
                C1 = (MEL+BEL)/BEL,
                XV=C1*XO+XB, YV=C1*YO+YB, ZV=C1*ZO+ZB,
                /*XCP=XC-XV,*/YCP=YC-YV,ZCP=ZC-ZV,
                XEP=XE-XV,YEP=YE-YV,ZEP=ZE-ZV,
                XK=YCP*ZEP-ZCP*YEP,
          XK>0, XEP=0,
          MAG=sqrt(YCP*YCP+ZCP*ZCP),
          C2=MEL/MAG,
          YT=C2*YCP,ZT=C2*ZCP,
          ALPHA=0.955316618,
          MAG2=sqrt(YT*YT+ZT*ZT),
          C3=MEL*cos(ALPHA)/MAG2,
          XE3P=MEL*sin(ALPHA),
          YE3P=C3*YT,
          ZE3P=C3*ZT,
          XE2P=0,
          YE2P=YT*0.8660+ZT*0.5,
          ZE2P=-YT*0.5+ZT*0.8660,
          ZE2 = ZE2P+ZV, ZE3 = ZE3P+ZV, YE2 = YE2P+YV,
          YE3 = YE3P+YV, XE2 = XE2P+XV, XE3 = XE3P+XV,
          VCOORD=[XV,YV,ZV],E2COORD=[XE2,YE2,ZE2],
          E3COORD=[XE3,YE3,ZE3],
          retract(location(cluster(CLUSTER),POS)),
          asserta(temp_retract(location(cluster(CLUSTER),POS))),
          assert_cluster_location(CLUSTER,ME,VCOORD,ECOORD,
                    E2COORD,E3COORD).

assert_cluster_location(CLUSTER,ME,A,B,C,D) :-
          equal(ME,"me1"),
          assertz(location(cluster(CLUSTER),[A,B,C,D])),
          asserta(temp_assert(location(cluster(CLUSTER),
          [A,B,C,D])));
          equal(ME,"me2"),
          assertz(location(cluster(CLUSTER),[A,D,B,C])),
          asserta(temp_assert(location(cluster(CLUSTER),
          [A,D,B,C])));
          equal(ME,"me3"),
          assertz(location(cluster(CLUSTER),[A,C,D,B])),
          asserta(temp_assert(location(cluster(CLUSTER),
          [A,C,D,B]))).


beam_end_vector(BEAM,BE,BCOORD,ECOORD) :-
          equal(BE,"be1"),
          location(beam(BEAM),[ECOORD,BCOORD,_,_]);
          equal(BE,"be2"),
          location(beam(BEAM),[_,_,BCOORD,ECOORD]).

new_cluster_location_in_rack(X,Y) :-
          equal(X,beam_end(BEAM,BE)),
          equal(Y,mush_end(CLUSTER,ME)),
          beam_end_vector(BEAM,BE,BCOORD,ECOORD),
          equal(BCOORD,[XB,YB,ZB]),
          equal(ECOORD,[XE,YE,ZE]),
          beam_end_length(BEL),
          mush_end_length(MEL),
          XO=XE-XB,YO=YE-YB,ZO=ZE-ZB,
          C1 = (MEL+BEL)/BEL,
          XV=C1*XO+XB, YV=C1*YO+YB, ZV=C1*ZO+ZB,
          C2=MEL*0.7071,
          XE2=XV-C2,
          YE2=YV-C2,
          ZE2=ZV+C2,
          XE3=XV+C2,
          YE3=YV-C2,
                ZE3=ZV+C2,
                VCOORD=[XV,YV,ZV],
                E2COORD=[XE2,YE2,ZE2],
                E3COORD=[XE3,YE3,ZE3],
                retract(location(cluster(CLUSTER),POS)),
                asserta(temp_retract(location(cluster(CLUSTER),POS))),
                assert_cluster_location(CLUSTER,ME,VCOORD,ECOORD,
                        E2COORD,E3COORD).



        free_(X) :-
                free_mush_end(X);
                free_beam_end(X).

: Graphics */

: Graphics for parts rack */

        refresh_rack_graphics :-
                shiftwindow(4),
                clearwindow,
                beam_rack(X),
                cluster_rack(Y),
                draw_parts_rack(X,Y),
                draw_attached_clusters.

        draw_attached_clusters :-
                connected(X,Y),
                equal(X,beam_end(BEAM,_)),
                equal(Y,mush_end(CLUSTER,_)),
                not(attached(beam(BEAM))),
                attached(cluster(CLUSTER)),
                draw_cluster_attached_to_beam_in_rack
                        (cluster(CLUSTER),2),
                fail;
                true.

        draw_parts_rack(X,Y) :-
                draw_beam_rack(X),
                draw_cluster_rack(Y).

        draw_beam_rack(X) :-
                equal(X,[Y|Z]),!,
                draw_in_rack(Y),!,
                draw_beam_rack(Z),!;
                true.

        draw_cluster_rack(X) :-
                equal(X,[Y|Z]),!,
                draw_in_rack(Y),!,
                draw_cluster_rack(Z),!;
                true.

        draw_in_rack(X) :-
                draw_beam_in_rack(X,3);
                draw_cluster_in_rack(X,2).

        draw_beam_in_rack(X,CO) :-
                equal(X,beam(_)),
        location(X,POS),
        shrink_piece(POS,[BE1,BB1,BB2,BE2]),
        draw_line_in_rack(BE1,BB1,CO),
        draw_line_in_rack(BB1,BB2,CO),
        draw_line_in_rack(BB2,BE2,CO).

draw_cluster_in_rack(X,CO) :-
        equal(X,cluster(_)),
        location(X,POS),
        shrink_piece(POS,[BASE,ME1,ME2,ME3]),
        draw_line_in_rack(BASE,ME1,CO),
        draw_line_in_rack(BASE,ME2,CO),
        draw_line_in_rack(BASE,ME3,CO).

draw_cluster_attached_to_beam_in_rack(X,CO) :-
        equal(X,cluster(CLUSTER)),
        connected(Y,mush_end(CLUSTER,"me1")),
        equal(Y,beam_end(BEAM,"be2")),
        location(beam(BEAM),BPOS),
        location(cluster(CLUSTER),CPOS),
        shrink_cluster_on_beam(BPOS,CPOS,[BASE,ME1,ME2,ME3]),
        draw_line_in_rack(BASE,ME1,CO),
        draw_line_in_rack(BASE,ME2,CO),
        draw_line_in_rack(BASE,ME3,CO).

shrink_cluster_on_beam(X,Y,Z) :-
        equal(X,[E1COORD,_,_,E2COORD]),
        equal(Y,[_,ME1,_,_]),
        equal(E2COORD,ME1),
        equal(E1COORD,[XBE,YBE,_]),
        equal(Y,[[XB,YB,ZB],[XE1,YE1,ZE1],[XE2,YE2,ZE2],
                [XE3,YE3,ZE3]]),
        shrink_factor(SF),
        XBP=(XB-XBE)*SF+XBE,
        YBP=(YB-YBE)*SF+YBE,
        XE1P=(XE1-XBE)*SF+XBE,
        YE1P=(YE1-YBE)*SF+YBE,
        XE2P=(XE2-XBE)*SF+XBE,
        YE2P=(YE2-YBE)*SF+YBE,
        XE3P=(XE3-XBE)*SF+XBE,
        YE3P=(YE3-YBE)*SF+YBE,
        equal(Z,[[XBP,YBP,ZB],[XE1P,YE1P,ZE1],[XE2P,YE2P,ZE2],
                [XE3P,YE3P,ZE3]]).


shrink_piece(X,Y) :-
        equal(X,[[XE1,YE1,ZE1],[XB1,YB1,ZB1],[XB2,YB2,ZB2],
                [XE2,YE2,ZE2]]),
        shrink_factor(SF),
        XB1P=(XB1-XE1)*SF+XE1,
        YB1P=(YB1-YE1)*SF+YE1,
        XB2P=(XB2-XE1)*SF+XE1,
        YB2P=(YB2-YE1)*SF+YE1,
        XE2P=(XE2-XE1)*SF+XE1,
        YE2P=(YE2-YE1)*SF+YE1,
        equal(Y,[[XE1,YE1,ZE1],[XB1P,YB1P,ZB1],
                [XB2P,YB2P,ZB2],[XE2P,YE2P,ZE2]]).

displace_position(X,Y,Z) :-
```

```
                equal(X,[[X1,Y1,Z1],[X2,Y2,Z2],[X3,Y3,Z3],[X4,Y4,Z4]])
                equal(Y,[XD,YD,ZD]),
                X1D=XD,
                Y1D=YD,
                Z1D=ZD,
                XDP=X1-XD,
                YDP=Y1-YD,
                ZDP=Z1-ZD,
                X2D=X2-XDP,
                Y2D=Y2-YDP,
                Z2D=Z2-ZDP,
                X3D=X3-XDP,
                Y3D=Y3-YDP,
                Z3D=Z3-ZDP,
                X4D=X4-XDP,
                Y4D=Y4-YDP,
                Z4D=Z4-ZDP,
                equal(Z,[[X1D,Y1D,Z1D],[X2D,Y2D,Z2D],[X3D,Y3D,Z3D],
                        [X4D,Y4D,Z4D]]).

        draw_line_in_rack(A,B,CO) :-
                equal(A,[XA,YA,_]),
                equal(B,[XB,YB,_]),
                line(XA,YA,XB,YB,CO).
*               equal(A,A),equal(B,B),equal(CO,CO).*/


* Graphics for structure */

        refresh_structure_graphics :-
                shiftwindow(1),
                clearwindow,
                draw(cluster("base_cluster")),
                completed(X),
                draw_completed_sequence(X).

        draw_completed_sequence(SEQ) :-
                equal(SEQ,[STEP|TSEQ]),!,
                draw_completed_step(STEP),!,
                draw_completed_sequence(TSEQ),!;
                true.

        draw_completed_step(STEP) :-
                equal(STEP,beam_to_attached_cluster
                        (beam_end(BEAM,BE),_)),
                other_end(BE,EB),
                not(connected(beam_end(BEAM,EB),_)),
                draw_beam_with_free_end(beam(BEAM),3);
                equal(STEP,beam_to_attached_cluster
                        (beam_end(BEAM,_),_)),
                draw(beam(BEAM));
                equal(STEP,cluster_to_attached_beam(_,mush_end(CLUSTER
                draw(cluster(CLUSTER));
                equal(STEP,cluster_to_racked_beam(_,_)),true;
                equal(STEP,beam_with_cluster_to_attached_cluster
                        (beam_end(BEAM,BE),mush_end(CLUSTER1,_))),
                other_end(BE,EB),
                connected(beam_end(BEAM,EB),mush_end(CLUSTER2,_)),
                draw(beam(BEAM)),
                draw(cluster(CLUSTER2));
        equal(STEP,triangle(X,Y)),true.

draw_beam_with_free_end(PIECE,CO) :-
        equal(PIECE,beam(BEAM)),
        connected(beam_end(BEAM,"be1"),_),
        location(PIECE,[BE1,BB1,BB2,BE2]),
        draw_line(BE1,BB1,CO),
        draw_line(BB1,BB2,CO);
        equal(PIECE,beam(BEAM)),
        connected(beam_end(BEAM,"be2"),_),
        location(PIECE,[BE1,BB1,BB2,BE2]),
        draw_line(BB1,BB2,CO),
        draw_line(BB2,BE2,CO).


draw(PIECE) :-
        draw_beam(PIECE,3);
        draw_cluster(PIECE,2).

draw_beam(PIECE,CO) :-
        equal(PIECE,beam(_)),
        location(PIECE,[BE1,BB1,BB2,BE2]),
        draw_line(BE1,BB1,CO),
        draw_line(BB1,BB2,CO),
        draw_line(BB2,BE2,CO).

draw_cluster(PIECE,CO) :-
        equal(PIECE,cluster(_)),
        location(PIECE,[BASE,ME1,ME2,ME3]),
        draw_line(BASE,ME1,CO),
        draw_line(BASE,ME2,CO),
        draw_line(BASE,ME3,CO).

draw_beam_end(X,CO) :-
        equal(X,beam_end(BEAM,BE)),
        beam_end_vector(BEAM,BE,BCOORD,ECOORD),
        draw_line(BCOORD,ECOORD,CO).

draw_mush_end(X,CO) :-
        equal(X,mush_end(CLUSTER,ME)),
        mush_end_vector(CLUSTER,ME,VCOORD,ECOORD),
        draw_line(VCOORD,ECOORD,CO).

draw_line(A,B,COLOR) :-
/*      equal(A,A),equal(B,B),equal(COLOR,COLOR).*/


        equal(A,[XA,YA,ZA]),
        equal(B,[XB,YB,ZB]),
        theta(THETA),
        offsets(XOFF,YOFF),
        XAR = XA*cos(THETA)-ZA*sin(THETA)+XOFF,
        YAR = YA+YOFF,
        XBR = XB*cos(THETA)-ZB*sin(THETA)+XOFF,
        YBR = YB+YOFF,
        line(XAR,YAR,XBR,YBR,COLOR).


 Keyboard input routines */
```

```
               readkey(KEY) :-
                       readchar(T),char_int(T,VAL),key_code(KEY,T,VAL).

               key_code(KEY,_,0) :-
                       readchar(T),char_int(T,VAL),key_code2(KEY,VAL),!.
               key_code(break,_,3):-!.         key_code(bdel,_,8):-!.
               key_code(tab,_,10):-!.          key_code(cr,_,13):-!.
               key_code(esc,_,27):-!.          key_code(char(T),T,_):-!.

               key_code2(btab,15):-!.          key_code2(home,71):-!.
               key_code2(up_arrow,72):-!.      key_code2(left_arrow,75):-!.
               key_code2(right_arrow,77):-!.   key_code2(endk,79):-!.
               key_code2(down_arrow,80):-!.    key_code2(ins,82):-!.
               key_code2(del,83):-!.
               key_code2(fkey(N),V):- V>58, V<70, N=V-58,!.
               key_code2(other,_):-!.


 Initialization */

               beam_end_length(2000).
               mush_end_length(2000).
               beam_length(16000).
               theta(-0.4).
               offsets(17000,15000).
               shrink_factor(0.35).

               init :-
                       zap_database,
                       assertz(beam_rack([beam(beam1),beam(beam2),
                       beam(beam3),beam(beam4),beam(beam5),beam(beam6)])),
                       assertz(cluster_rack([cluster(top_cluster1),
                       cluster(top_cluster2),cluster(top_cluster3)])),

                       locate_fixed_beam(beam1,[2000,0,0]),
                       locate_fixed_beam(beam2,[3000,0,0]),
                       locate_fixed_beam(beam3,[4000,0,0]),
                       locate_fixed_beam(beam4,[5000,0,0]),
                       locate_fixed_beam(beam5,[6000,0,0]),
                       locate_fixed_beam(beam6,[7000,0,0]),
                       locate_fixed_cluster(top_cluster1,[1000,1000,0]),
                       locate_fixed_cluster(top_cluster2,[1000,2000,0]),
                       locate_fixed_cluster(top_cluster3,[1000,3000,0]),

                       assertz(dummy_beam_one([])),
                       assertz(dummy_beam_two([])),
                       assertz(dummy_cluster_one([])),
                       assertz(dummy_cluster_two([])),

                       assertz(free_racked_beam_end(beam_end("beam1","be2"))),
                       assertz(free_racked_beam_end(beam_end("beam2","be2"))),
                       assertz(free_racked_beam_end(beam_end("beam3","be2"))),
                       assertz(free_racked_beam_end(beam_end("beam4","be2"))),
                       assertz(free_racked_beam_end(beam_end("beam5","be2"))),
                       assertz(free_racked_beam_end(beam_end("beam6","be2"))),

                       assertz(racked(beam("beam1"))),
                       assertz(racked(beam("beam2"))),
                       assertz(racked(beam("beam3"))),
        assertz(racked(beam("beam4"))),
        assertz(racked(beam("beam5"))),
        assertz(racked(beam("beam6"))),

        assertz(free_mush_end(mush_end("base_cluster","me1"))),
        assertz(free_mush_end(mush_end("base_cluster","me2"))),
        assertz(free_mush_end(mush_end("base_cluster","me3"))),
        assertz(attached(cluster("base_cluster"))),
        assertz(racked(cluster("top_cluster1"))),
        assertz(racked(cluster("top_cluster2"))),
        assertz(racked(cluster("top_cluster3"))),
        assertz(proposed([])),
        assertz(completed([])),
        mush_end_length(MEL),
        beam_end_length(BEL),
        beam_length(BL),
        FBR = (BEL+MEL)/MEL, SBR = (BEL+BL+MEL)/MEL,
        TBR = (BEL+BL+BEL+MEL)/MEL,
        assertz(first_beam_ratio(FBR)),
        assertz(second_beam_ratio(SBR)),
        assertz(third_beam_ratio(TBR)),
        SL=BL+2*BEL, MAGC=(SL*0.7071)/2,
        XC=0, YC=0, ZC=0,
        assertz(center([XC,YC,ZC])),
        X=MAGC, Y=YC, Z=ZC, BCV=[X,Y,Z],
        locate_fixed_cluster("base_cluster",BCV),
        keep_current_database,
        refresh_structure_graphics,
        refresh_rack_graphics,
        shiftwindow(3),
        clearwindow,
        write("initialized").


zap_database :-
        retract(_), zap_database;
        true.
```

References

1) Akin, D.L., Minsky, M.L., Thiel, E.D., Kurtzman, C.R., *Space Applications of Automation, Robotics and Machine Intelligence (ARAMIS) - Phase II*, NASA Contractor Report 3734, 1983

2) Manganiello, A.J., *Supervisory Control of the Right Arm of the Beam Assembly Teleoperator*, M.S. Thesis, MIT Department of Mechanical Engineering, May 1985

3) Akin, D.L., Spofford, J.R., Viggh, H.E.M., *Integrated Systems Tests of Space Teleoperator Concepts in the Neutral Buoyancy Environment*, Technical Paper, MIT Space Systems Lab, Department of Aeronautics and Astronautics, 1986

4) Robinson, P.R., *Using Turbo Prolog*, McGraw Hill, Berkely, CA, 1987

5) Clocksin, W.F., Mellish, C.F., *Programming in Prolog*, 2d ed., Springer-Verlag, New York, 1984

6) Shain, E.B., *Design and Control of a Beam Assembly Teleoperator*, M.S. Thesis, MIT Department of Aeronautics and Astronautics, May 1983