1985

Using artifical intelligence to improve the man-machine interface in robotic assembly systems /

Keith James Werkman
Lehigh University

Using Artificial Intelligence To Improve The Man-Machine Interface In Robotic Assembly Systems

by

Keith James Werkman

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

1985

This thesis is accepted and approved in partial fulfillment of the requirements for the Degree of Master of Science.

12/16/85
(date)

Glenn David Blank
Professor in Charge

Donald J. Hillman
CS Division Chairman

Leo H. Thompson
CSEE Department Chairman

Acknowledgments

I would like to take this time to acknowledge all those who have helped to bring about this paper and the various concepts that are expressed.

Initially, I would like to thank my parents for their years of support, guidance, love and understanding which without I could not have produced this paper. I would like to thank Dr. Glenn Blank for being a caring teacher and advisor during the research project. I would also like to thank Dr. Roger Nagel for providing the necessary background, robot laboratory facilities, and the opportunity to conduct research in this field. Special thanks to Jim Rausa for his endless rereadings and critiques of earlier versions of this manuscript and to Stephen Corbesero for his assisting with the Scribe text formatter. A special thanks to Bob Morein of Automata Design Associates for providing me with the latest virtual memory version of VMA Prolog for the IBM PC. The expanded version of the MIC Planner could not have been done without it. Finally, I would like to thank all my friends and fellow graduate students who supported me during the research project.

Trademarks

The following are registered trademarks.

AML, PC-DOS and IBM PC
- International Business Machines Corporation

MS-DOS - Microsoft, Inc.
Turbo Pascal - Borland International
VAL and Puma - Unimation, Inc.
VMA Prolog - Automata Design Associates

Table of Contents

List of Figures

List of Tables

Abstract

As a means of improving man-robot communications, researchers at Lehigh University have been developing a natural language system call Register Vector Grammar (***RVG***). RVG, a relatively compact, efficient, general purpose natural language system written in Turbo Pascal, should prove well suited to real-time, dynamic applications such as robotics. Integrated with the RVG system is a task-level, object-oriented hierarchical planning system called ***MIC Planner***. Prototyped in Prolog for MS-DOS microcomputers, MIC Planner uses a rule-based approach to decompose high-level abstract assembly tasks into lower level robot motion primitives. MIC Planner attempts to achieve a goal state while manipulating objects in a world model that is maintained by the RVG natural language system. As an option, the user can also log MIC robot motion commands generated by the planner to a file. These robot commands can then be compiled by the MIC Compiler System into native operation codes and down loaded to the robot. This overview of the RVG system along with a review of current robot control strategies should give some insight into the requirements that must be met in order to develop intelligent and easy to program robot control systems.

Chapter 1
Introduction

This research is an application of artificial intelligence (AI) techniques that can be applied to enhance the man-machine interface in robotic assembly. More specifically, we focus on two heavily researched subfields of AI, that of natural language processing and planning systems. Artificial intelligence is a well established field in computer science that has recently gained great attention due to the popularity of knowledge based expert systems. Back in the early 1970's, robots and AI applications to robotics were considered the state-of-the-art research topic for AI researchers. This came about after Winograd published his Ph.D. dissertation on understanding natural language in 1971 [26]. Winograd's now classical 'blocks world' system allowed the user to manipulate items in a blocks world domain with a graphical robot called SHRDLU simply by entering natural language English commands. This work generated much excitement among the AI community and raised many hopes for developing robust systems that could understand natural language.

During the middle to late 1970s, many of the expectations of AI researchers were not met. These disappointments led to a decrease in interest in both the areas of natural language and robotics research. Only within the past few years has there been a renewed interest in combining AI techniques with robots in an effort to develop the next generation of autonomous or "smart" robots.

Due to rising labor costs in many industrial manufacturing operations, robotic assembly has now become technologically and economically feasible. Thus, technological improvements at any level to the current state-of-the-art in

robotics assembly is seen as very desirable by industry. Funding by large industrial manufacturers to academic robotic research institutions is on the increase. Not to be left behind, the U.S. government is also funding research in the fields of AI and robotics, partly for military reasons and partly for the development of robotic and automation standards.

This paper is the result of a research project started during the summer of 1985 at Lehigh University in the area of natural language processing. The original goal of the project was to develop a version of the **R**egister **V**ector **G**rammar (RVG) natural language parsing system to run on PC/MS-DOS based microcomputers. As in earlier natural language and planning systems, the robot domain was chosen as a limited applications area for the RVG system to interact with. The domain or the robot's world provided realistic boundaries in terms of what types of sentences the RVG parsing system would initially be required to handle.

In addition to the parsing system, a planning system had to be developed to allow the parsed sentences to actually interact with the problem domain of the robot. The planning system had to be able to efficiently model and manipulate items in the robot's world. Also, the planner needed an integrated interface with the parsing system to allow for the passing of commands and parameters to the planner. In addition to the planning system, a separate robot interface was developed to handle needed calculations for the robot's arm movements and to communicate these movements to the robot.

During the design of the planner, several classical data and goal driven planning systems were reviewed. The result was a prototype planner which incorporated some of the aspects of these earlier systems. The MIC Planner,

named after the MIC robot control language developed by Werkman [24], is a task level object-oriented hierarchical planning system, written in Prolog, targeted to run on PC/MS-DOS microcomputers.

The particular robot used in the project was the TeachMover instructional robot by Microbot Incorporated. The reasons for this robot were severalfold. First, the system developers were familiar with the capabilities of this 5 degree of freedom robotic arm. Second, using an inexpensive instructional robot was safer and more practical as opposed to using an expensive industrial robot. This is especially true when developing a new robot control system where an unexpected action could result in the robot putting it's arm through a wall. Third, the TeachMover is a classical example of a first generation robot with respect to programming and control. Since many robots in use in industry today have similar restricted programming and control systems, any improvement in the interface to the TeachMover robot would also apply to those similar robots in industry.

This paper will provide an overview of the RVG system, focusing on aspects of the system that are related to improving the human-robot interface. Several areas will be reviewed including current robot programming methods and robotic software systems, classical planning strategies and new methods which use world knowledge to guide the planning. The National Bureau of Standards **R**obot **C**ontrol **S**ystem (RCS) will be examined for its use of hierarchical decomposition of robot tasks [1]. The RVG system planner, MIC Planner, will be discussed and compared with the NBS system. Finally, the RVG system will be contrasted with other natural language systems that have been interfaced with robots.

Chapter 2
Current Robotic Control Methods

Many of the robots used in industry today are programmed by one of the following two methods [16]:

1. *Lead Through* - Many points are recorded as the robot's controller continuously samples positional feedback from the robot's actuators as the arm is physically led through all the points in the robot program.

2. *Teach Pendant* - The robot's arm is moved to destination points in the robot's program by flipping toggle switches on the hand held guiding device These fewer destination points are then recorded as part of the robot program.

Both of these methods require that the spatial positioning of the robot's arm be recorded. These spatial arm configurations are know as points in a robot program. The difference in these methods is in the actual number of points recorded. The *lead through* programming method records many points at very small increments and thus tends to be more difficult to edit. In the *teach pendant* method, only the end points of motions are recorded. A teach pendant is a mobile keypad that allows the robot operator to move the robot to desired locations and record points. The order in which the points are recorded reflects the sequence of arm motions that the robot will perform when the robot program is played back.

To aid in the robot program editing process, both methods are usually accompanied by associated vendor-supplied blackbox software. In the case of the lead through method, the robot program is broken into several parts and linked together by the software. If, for example, a move taught to the robot was incorrect, the robot operator must teach that move over again. In the case of

the teach pendant method, only the end points of a series of motions need be retaught. In this method, intermediate points in the arm's path are generated by the robot's controller and can be modified by software.

Similarly, the TeachMover instructional robot system can only be programmed by using a teach pendant. Thus, the same editing problems associated with large industrial robots are also found with the TeachMover. In order to make the TeachMover's programming environment more palatable, Werkman developed a simple robot programming language called MIC which stands for Microbot Instruction Code [24]. The MIC language is an example of a first generation robot language that provides off-line program editing, program transportability, and the ability to saving and restoring programs to and from secondary storage.

2.1 First Generation Robot Languages

The blackbox software systems fall into the category of first generation robot languages. In order for the robot manufacturers to successfully market their robots, they had to provide a means of programming them that was easy and straightforward. Thus, first generation robot languages were written for users who had very little programming experience. Many of these languages resembled the BASIC computer language with extensions to allow for robot motion, simple sensor I/O, and basic operating system support for saving and restoring program files to floppy disk. The languages in widest use today is VAL. VAL was developed by Unimation Inc. for their line of Puma industrial robots [23]. In fact, since VAL is so easy to learn and use, many robot vendors have borrowed its simple syntax and feel for their own robot control languages.

While its true that these first languages are easy to understand and use,

their capabilities are somewhat limited. The computational ability of most first generation robot languages are limited to simple operators like addition and subtraction. These languages are not very extensible, unlike common 4th generation computer languages such as Pascal. Thus, users cannot write user-defined functions in their programs. The language's ability to communicate with external systems is also limited to only the robot's controller. External systems such as vision systems and other robots cannot be easily interfaced with many first generation robot software control systems.

2.2 Second Generation Robot Languages

The answer to the limitations of the first generation languages has been a new generation of robot languages which borrow heavily from modern computer science languages. These second generation languages initially took the form of procedural languages. offering user extensibility and limited scope of variables. Several first generation features have been kept by the new language systems, such as using the teach pendant to move the robot arm and record points. Two of the more popular second generation robot languages include AML by IBM [4] and VAL II by Unimation.

Figure 2-1 gives example program code, in VAL I and AML, for the task of placing a peg in a hole [12]. Some of the execution branching in the example programs is made upon sampling input signals received from pressure sensors located in the robot's gripper. The user will immediately notice that the format and logic of each program differs. The VAL program is very terse and looks much like BASIC with its many GOTO statements. Each robot step follows in sequence from the top of the program to the bottom. The AML program on the other hand consists of a group of user defined procedures much

TABLE 18-1 Examples of VAL and AML Programs for Placing a Peg in a Hole

VAL

```
     SETI     TRIES = 2
     REMARK   If the hand closes to less than 100 mm, go to statement
                labelled 20.
10   GRASP    100, 20
     REMARK   Otherwise continue at statement 30.
     GOTO     30
     REMARK   Open the fingers, displace down along world Z axis and
                try again.
20   OPENI    500
     DRAW     0, 0, -200
     SETI     TRIES = TRIES - 1
     IF       TRIES GE 0 THEN 10
     TYPE     NO PIN
     STOP
     REMARK   Move 300mm above HOLE following a straight line.
30   APPROS   HOLE, 300
     REMARK   Monitor signal line 3 and call procedure ENDIT to STOP
                the program.
     REMARK   if the signal is activated during the next motion.
     REACTI   3, ENDIT
     APPROS   HOLE, 200
     REMARK   Did not feel force, so continue to HOLE.
     MOVES    HOLE
```

AML

```
PICKUP: SUBR (PART_DATA, TRIES);
  MOVE(GRIPPER, DIAMTER(PART_DATA)+0.2);
  MOVE(<1, 2, 3>, XYZ_POSITION(PART_DATA)+<0, 0, 1>);
  TRY_PICKUP(PART_DATA, TRIES);
  END;
TRY_PICKUP: SUBR(PART_DATA, TRIES);
  IF TRIES LT 1 THEN RETURN ('NO PART').
  DMOVE(3,-1.0);
  IF GRASP(DIAMETER(PART_DATA)) = 'NO PART'
    THEN TRY_PICKUP(PART_DATA, TRIES -1);
  END;
GRASP: SUBR(DIAMETER, F);
  FMONS: NEW APPLY ($MONITOR, PINCH_FORCE(F));
  MOVE(GRIPPER, 0, FMONS);
  RETURN (IF QPOSITION (GRIPPER) LE DIAMETER/2
    THEN 'NO PART'
    ELSE 'PART');
  END;
INSERT: SUBR (PART_DATA, HOLE);
  FMONS: NEW APPLY ($MONITOR,
    TIP_FORCE(LANDING-FORCE));
  MOVE(<1, 2, 3>, HOLE+<0, 0, .25>);
  DMOVE(3, -1.0, FMONS);
  IF QMONITOR(FMONS) = 1
    THEN RETURN ('NO HOLE');
  MOVE(3, HOLE(3) + PART_LENGTH(PART_DATA));
  END;
PART_IN_HOLE: SUBR (PART_DATA, HOLE);
  PICKUP (PART_DATA, 2.);
  INSERT (PART_DATA, HOLE);
  END;
```

Figure 2-1: Examples of VAL and AML Programs.

like what a Pascal programmer would write.

The AML program in Figure 2-1 is initiated when the user makes a call to the main program procedure *PART_IN_HOLE*. This procedure then calls two additional procedures, *PICKUP* and *INSERT*. Each of these procedures is then broken down into smaller parts which perform the desired function. This *top down* approach to designing robot programs is very desirable. Such a software design methodology allows the user to create libraries of functions that other users can then reference when developing their robot programs.

Thus, the major advantages of second generation robot languages includes the ability by the user to create and tailor robot software to meet specific robot tasks. Once written, the tailored program can then be used by nonprogrammers who need not be aware of the deeper intricacies of the program. Second generation languages also provide enhanced computational power, mainly through the addition of more complex operators. These languages are also able to communicate better with complex sensors such as vision systems and force sensors. The operating systems that accompanied the new language systems are also improved. Table 2-1 by Gevarter lists other first and second generation robot languages, their inventors, their uses, and their status.

Even though many of the problems of the first generation languages are addressed, the enhancements of the second generation languages still have their shortcomings. Because of the complexity of the new languages nonprogrammers cannot fully utilize all of the capabilities of the new robot control software. Thus, experienced programmers are needed to develop robot applications. Personal experience in instructing students and engineers in using advanced robot languages proves this to be true. Since additional programming support costs

Table 2-1: Current Robot Control Languages.

Robot Language	Organization	Control Mode			Manipulation Type		Status		Capability		Comments
		Position	Force	Vision	Rectilinear	Jointed	Commercial	Research	Assembly	Control of Multiple Arms	
AL	Stanford University	X	X	X		X	X	X	X	X	World model capability
AML	IBM	X	X		X		X		X		Menu capability
HELP	GE	X	X		X		X		X	X	
JARS	JPL	X	X	X		X		X			Designed for visual and force servoing
MCL	McDonnell-Douglas	X				X	X			X	For off-line programming of robots from a CAD data base
RAIL	Automatix	X		X	X	X	X		X		Developed for visual inspection, assembly, and arc welding
RPL	SRI	X		X		X		X			For control of machines comprising a robot work cell
VAL	Unimation	X		X		X	X		X		

Source: Based on Gruver et al. (1983).

money, interest has shifted from just versatility to both versatility and ease of use. It seems that a new form of robot programming is needed. A new state-of-the-art robot control and programming methodology has recently been developed by the National Bureau of Standards in Washington, D.C.

2.3 The NBS Hierarchical Control Strategy

The National Bureau of Standards is one government agency that is attempting to address the need for standardization in the automation industry. One of the issues that NBS is looking into is that of robot integration into the manufacturing process, an issue that has not been adequately addressed by the robot manufacturers. Many current robot systems cannot easily interface with systems from different vendors. Each vendor designs his software system to work only on his particular line of robots. Thus, programs written to run on one vendor's robots usually will not run on other vendor's. Software incompatibility is one of the major problems in robotics today, preventing the advent of fully automated factories.

The Industrial Systems Division of the NBS has created an Automated Manufacturing Research Facility (AMRF) as a testbed for studying robot control strategies and interface standards. In January of 1984, NBS researchers implemented an integrated manufacturing system using several machine tool workstations, two fixed base robots, a mobile cart robot, a gripper control system, a safety system, a database, all connected by a network [14]. Figure 2-2 shows a schematic of the AMRF.

The control strategy used by NBS's Robot Control System (RCS) is that of task decomposition at successive levels in a control hierarchy. Each level in the hierarchy decomposes simpler command strings to the next level in the

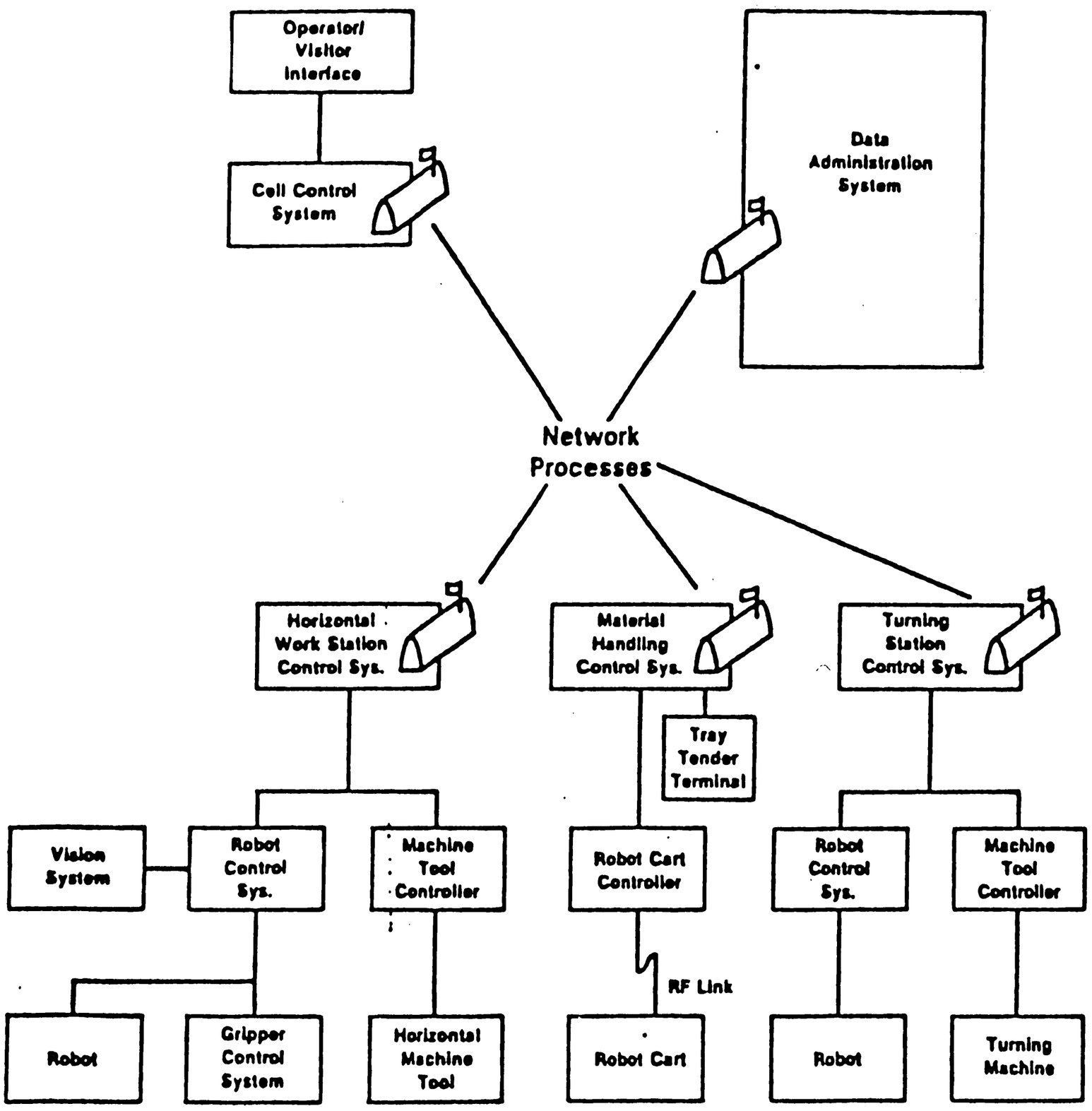


Figure 2-2: Conceptual diagram of the AMRF.

decomposition until the lowest levels generate the actual drive signals for the robot, the gripper, and other actuators in the system [14]. Each level in the hierarchy has a specific responsibility to perform and is independent of other task levels, thus allowing for greater modularity and easier updating of the system structure.

The NBS Robot Control System also attempts to deal with control in a real-time dynamically changing world. As pointed out by NBS researchers, AI planning systems differ from control systems in that most planning systems that have been developed for robot control have never dealt with real world time

constraints. Planning is usually done in static worlds that don't change while backtracking is being performed to find possible solutions in a search space[1]. Control systems on the other hand deal with constant sensor input from the world in the form of force, vision, binary and other sensors [3]. The goal of the NBS system is to use a hierarchical decomposition of abstract task descriptions in combination with real-time control systems utilizing sensor feedback. Given such a system, a goal-directed task can be accomplished in spite of perturbations in the manufacturing environment [5].

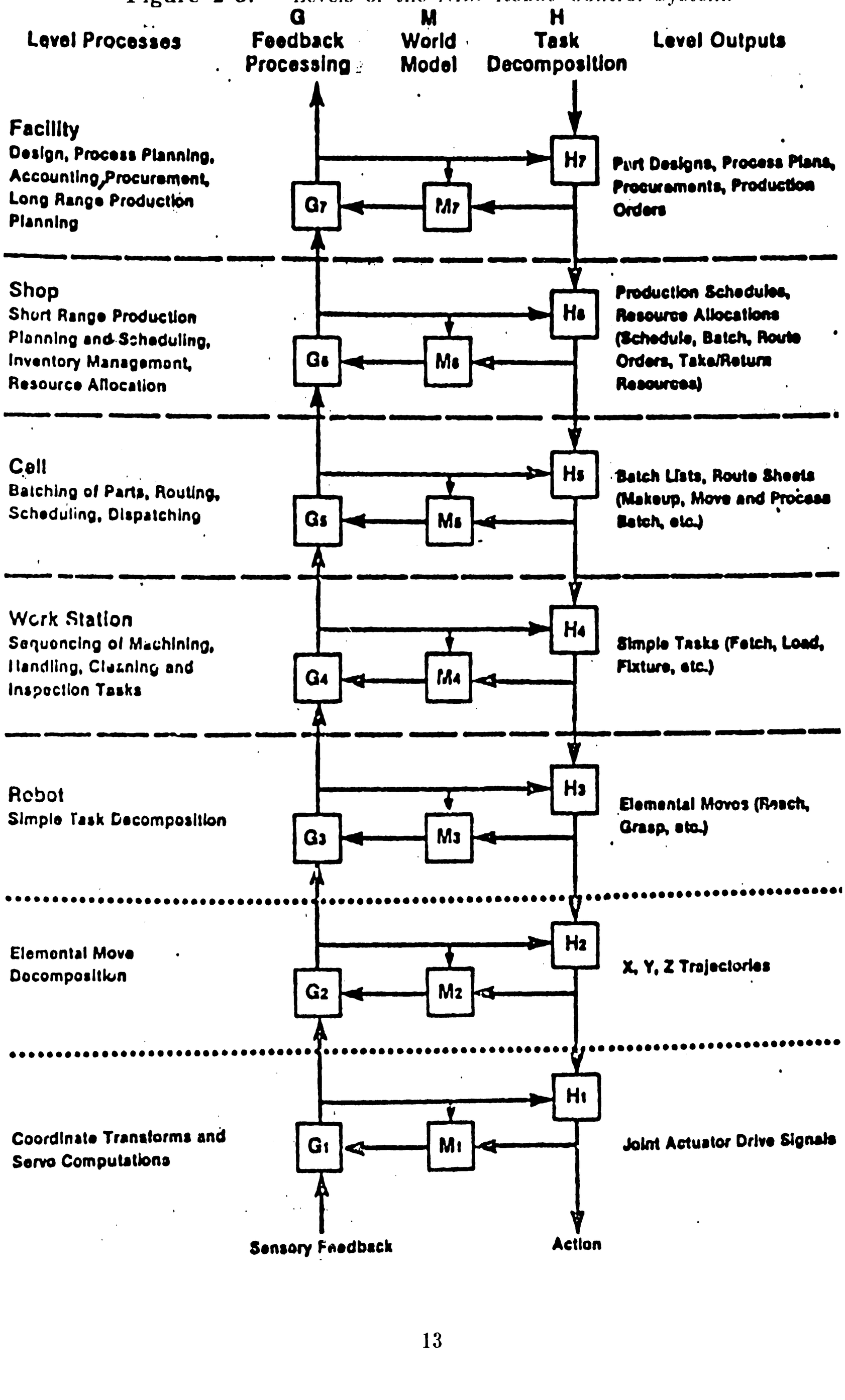
There are seven levels of hierarchical control in the current AMRF project as described by Albus [2] and shown in Figure 2-3. The most abstract level (seventh level) is that of *facility control.* This level includes such things as the product design, process planning, accounting and other long range planning. The next level down the hierarchy is the sixth level, that of *shop control.* At this level, shorter range production planning is done along with scheduling, inventory management and resource allocation. It is at this level that timing of the overall assembly process is taken into account to manage the entire shop floor assembly process.

Time constraints are also important at the fifth level in the control hierarchy which deals with *cell control.* Supplies of various tools for robots and the machine tools along with raw materials are managed to meet the assembly demand. Requests are sent to materials transport workstations to delivered the needed resources. These resources are then managed by the fourth level of hierarchy. This is the *workstation control* level. Here, abstract robot and

---

[1]Planning is discussed in depth in the next chapter

Figure 2-3: Levels of the NBS Robot Control System.



machine tool commands are generated to perform the actual assembly operation. Example commands might include a clamping set up command given to a robot, then a set of commands given to a machine tool to actually machine the part, and then a set of commands to the robot to remove the clamping fixture and set up for the next operation [2].

The uppermost levels of the NBS Robot Control System are designed for total factory automation. The next three levels deal specifically with controlling robots and machine tools. The third level, the *robot control* level, takes commands generated by the workstation control level above and decomposes these commands into robot motion commands that move the arm. Sample commands might include [2]:

```
FETCH A
MATE B TO A
LOAD TOOL C WITH PART D
```

These abstract robot actions are then decomposed into subactions that the second level *environmental control* system deals with. At this level, commands like *fetch* are refined into actions like *Reach for part*, *Grasp*, and *Lift*. Each of these actions is done only after conditions are tested for that would cause such an action to fail. One such case would be that of a part not found. Finally, the lowest level of the control hierarchy is the *coordinate control* system. This system performs the actual coordinate transformations servo computations that are needed to move the arm to a desired location in space.

2.4 Future Trends

As a result of the work done by NBS on complex, real-time task decomposition, the next generation of robot languages being developed will most likely incorporate task or object-level commands. [16]. These task-level languages will allow robot operators to specify assembly actions in terms of a series of high level tasks. Once the assembly action is described and presented to the system, the task-level software will refine the abstract tasks into lower level actions until the lowest level robot movement primitives are generate. This is similar to the actions performed by the hierarchical control levels of the NBS Robot Control System.

Ideally, one would like to program a robot for a task much as one would describe the task to a fellow worker. In the future, robot control system will be able to read natural language input describing a task and then act upon them accordingly. For example [16]:

> **Mate the part with the hole in it with the part with the peg in it, so that the peg and the hole are aligned, and the corners of the surface are aligned.**

Indeed, parsing and interpreting complex sentences as above is a major goal of the RVG natural language system. In the near term though, reading and interpreting simpler instruction found on manufacturing process sheets would be a substantial improvement to robot programming. To provide this degree of high level task-oriented description capability, further work will be needed not only in the area of natural language understanding systems, but also in the area of task-decomposition by planning systems. The goal of the RVG system is not to write VAL I program in English, but to allow one to interact with the robot at a much higher level, leaving the details of physical locations and arm mo-

tions to a planning system.

Chapter 3
Robot Task Planning

The ability to communicate with a robot at an abstract level is major issue currently being examined by many researchers around the world. Many operations must be coordinated in order for an assembly task to proceed smoothly and efficiently in a manufacturing environment. Various different machines will have to communicate with each other in real-time, most likely across a local area network. The entire operation will be controlled by either one large control program or several independent programs executing asynchronously on each machine and communicating with each other. In either case, the task of programming these independent systems would be much easier for the shop floor engineer if he could simply describe the complete assembly task in terms of high level assembly procedures.

The ideal intelligent assembly control system would allow the engineer to describe these assembly task in terms of natural language sentences. The system might provide a library of abstract assembly functions that could be combined to accomplish the complete assembly operation. The actions of these routines might be displayed on a graphics monitor for verification before they are implemented on the shop floor. Such an intelligent robot control system will be composed of various interacting modules (sensor system, vision, natural language input, function library) all of which will be coordinated by a central control module. The control module will most likely be the system planning module with a global database that will act as a world model. This database will be updated by the other system modules, much like the *blackboard* concept used in the Hearsay II speech recognition project. A key component to the suc-

cess of any intelligent robot control system will undoubtedly be the planning module.

3.1 Planning Strategies

Planning and *problem solving* are usually considered as one related field by the AI community. Planning involves the process of combining groups of *subplans* in a specific order to achieve a goal. The goals of subplans can generally be replaced by other more refined subplans that are generally ordered in a hierarchical fashion. Most planning strategies generate a linear or partial ordering of problem solving operators that deal with the problem's search space. In the current AI literature four general planning strategies have been implemented. Each one of these methods will be briefly examined. These approaches include hierarchical planning, nonhierarchical planning, script-based planning, and opportunistic planning [8].

3.1.1 Hierarchical vs Nonhierarchical Planners

Hierarchical and *nonhierarchical* planning methods, as well as other planning methods, generate a hierarchy of subgoals to be achieved. The two methods differ in the way that they represent their plans. Hierarchical planners, like ABSTRIPS and NOAH, depict their plans in hierarchical levels of abstraction, the highest being the most abstract and the lowest being the most refined. This approach allows the planner to deal with very abstract aspects of a plan instead of specific details. Lower level details are postponed until they are needed and thus precious computational effort is not lost if that particular branch of the hierarchy is never chosen. In this sense, hierarchical planners can be said to be very efficient. Nonhierarchical planners, like STRIPS, HACKER,

and INTERPLAN, have only one representation for a plan. In the nonhierarchical approach, even though the subgoals are ordered in a hierarchical fashion, all subplans are examined in one abstraction space at the same level. Thus, the planner may waste time on reviewing one subplan that may not directly be related to the success of the overall goal.

When a failure occurs in a hierarchical planning system, the system usually backtracks to an earlier decision point in the search space to find another possible path. Plans generally fail when their preconditions in the world are not met. Plans can also fail when subgoals interact with each other. Subgoals are a series of conjunctive goals that must be attained by the planning system for the main goal to succeed. The order in which subgoals are applied can cause plans to fail. This is especially true for subgoals that undo the actions of previous subgoals when they change the state of the world.

The HACKER and INTERPLAN nonhierarchical planning systems deal with the problem of subgoal interaction by allowing the system to correct the offending condition as the plan proceeds. This approach is based on the theory of *linear assumption* where subgoals are considered to be independent and thus are achievable by any ordering scheme [11]. When a subgoal interaction is found, both systems attempt to reorder the subgoals, but only at the current level in the subgoal hierarchy. In the INTERPLAN system, the subgoal that failed is moved before other subgoals in the subgoal hierarchy [21]. This reordering between subgoal hierarchical levels is not available in HACKER. As a result, the INTERPLAN system has proven to be more efficient in generating plans that cause protection violations or the undoing of earlier subgoals.

A different approach for handling interacting subgoals is used in the

NOAH hierarchical planning system. NOAH uses two methods. First, the system does not arbitrarily order subgoals until their is some reason to do so. Second, the NOAH system examines each level of the developing plan and corrects subgoal interactions before they arise [17]. Unlike the nonhierarchical planners described above which commit themselves to a particular ordering of subgoals, HOAH adopts the approach of least commitment. NOAH avoids committing itself to a specific planning path until it has examined all possible paths at the current level in the planning hierarchy. This allows the system to constructively correct interacting subgoals without any need to backtrack.

NOAH (Nets of Action Hierarchies) represents plans in terms of procedural nets. Procedural nets represent procedural as well as declarative knowledge about problems. The procedural or domain knowledge includes functions that expanded goals into subgoals. The declarative knowledge contains information about the results of plans once they are executed. Thus, if NOAH puts a block on top of another, the supporting block is noted as not having a clear top. With this declarative world knowledge information, NOAH can reason about actions before it performs them [8]. A set of *critics* are employed by the system to review the declarative knowledge and prevent redundant actions.

When NOAH is given a goal, the system uses the procedural domain specific knowledge to expand the goal into several nodes in the procedural net. Then critics examine the net for any interacting subgoals. If any are found, other domain specific procedures are called to reorder the subgoals at the current abstraction level. The new ordering is tried and if found successful, the redundant subgoals are eliminated. Thus, through domain specific procedural information as well as a current model of the world, a hierarchical planning sys-

tem such as NOAH can avoid much backtracking.

3.1.2 Script-Based Planners

The *script-based* method of planning deals with plan generation in a much different way than do the hierarchical planners. Script-based or skeletal planners attempt to model human planning methods. When humans are given problems to solve, they tend to think of the problem in terms of a related problems that they are familiar with. This similar problem acts as a skeleton or framework which is expanded and updated as new information is discovered. A planning system that employs this method can deal with complex plans efficiently without the need to search through the entire rule-base for specific planning rules. Instead, the system only has to search the rules that reside within the outer general framework of the main goal.

The MOLGEN system uses the script-based approach to aid molecular biologists in laboratory procedures [19]. The system maintains a database of skeletal plans that range from very general to very specific planes. Once a skeletal plan is chosen, the plan refinement process begins. As additional information is learned about the problem area, only those subgoals related to the outer generalized plan skeleton are searched. If the currently selected subgoal is too specific, the system backs up to the general level skeleton plan and selects another subgoal, if any, within the framework of that plan. Thus a planning hierarchy is maintained by the system, the advantage being that frequently used plan skeletons are referenced first, reducing the search space and the search time.

3.1.3 Opportunistic Planners

A fourth methodology of planning is the *opportunistic* approach. This method uses a control strategy which is more flexible than the other methods described. Developed by the Hayes-Roths from their cognitive science research in human planning, the opportunistic method uses a blackboard structure much like that used in the Hearsay II speech recognition system [8]. Here, information relating to the plan being generated is made available to all levels of the planning system. As the planning proceeds, planning *specialists* examine the blackboard in an asynchronous fashion and suggest alternate planning possibilities.

The Hayes-Roths believe that this model is similar to the way humans formulate plans. Human use newly acquired information, much like the specialists in the model, to update their plans. Thus, when an opportunity arises to make a plan more efficient, human generally tend to modify their plans. The Hayes-Roths tested various subjects on errand-related tasks where the subjects were given several tasks to accomplish (places to go, items to pick up). The subjects then spoke aloud while they formulated their plans. The Hayes-Roths noticed that none of the subjects followed the plan that they initially generated. Instead, subjects continuously modified their plans to take advantage of opportunities as they arose (e.g., pick up an item at the store since we happen to be passing by on our way to another destination) [13].

The Hayes-Roths also noticed that subjects did not form plans hierarchically (top-down), but rather in a bottom-up (data driven) fashion. Subjects developed small pieces of their plans (islands subplans) when they though them to be logically feasible and then linked these island subplans together to form

an overall plan. This bottom-up incremental process of plan generation based on opportunities that arise in the subjects world prevents humans from spending large amounts of time replanning when a plan fails. In the opportunistic method, little or no backtracking is performed. Instead, the plan is constructively modified to take into account new opportunities or, in the case of a subgoal failure, the next best possible subgoal to prevent total plan failure. Indeed, there is almost never any complete plan failure because the opportunistic planner always plans to accommodate the changing conditions in his world. The opportunistic planning method may be especially appropriate for real-time dynamic applications, where events are continuously changing and causing the world model to be updated. In fact, the NBS Robot Control System is similar to the opportunistic planning method in that it maintains its world model in the blackboard fashion and updates this model frequently from sensors in the robot's environment [16].

3.2 The MIC Planner System

The MIC Robot Planner System is the applications system of the RVG natural language processing system. The natural language system will read sentences from a keyboard, and understand them, as they refer to a database that models the robot's world. If the sentence is imperative, the natural language system will issue a task, in the form of a Prolog predicate, to the MIC Planner System. A complete description of the RVG system and its component modules is given in the next chapter.

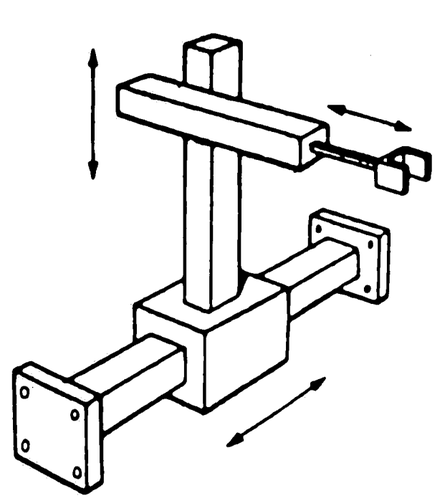
3.2.1 System Overview

MIC Planner is a self contained planning system that maintains all the necessary information needed to move bricks and cubes about in a robot's work cell. Unlike many early "blocks world" planning systems which only generated block movement plans, MIC Planner generates plans and carries them out by instructing a robot to move bricks around in the real world. Authors of several earlier planning systems believed that arm motions were unimportant trivial extensions which could easily be added to their systems. Upon interfacing the planner with an actual robot to perform in a real-world environment, many interesting and unexpected problems emerged.

One specific problem worth mentioning is the physical limitations of the robot arm. In order to develop a complete plan for, say stacking blocks in the world, one must take into account the actual range of motion of the robot manipulator. A plan's solution may be foiled simply because the robot's arm is physical incapable of attaining a specified height. Thus, a plan to stack multiple blocks that exceeds the robot's maximum arm height should also be noted by the planner and cause the stacking task to fail. Additional motion limitations can be cause by the robot's body (upper arm, elbow, forearm) bumping into other objects in the world. This is the heavily studied problem of collision avoidance. At Lehigh University, CAD/CAM researchers have placed a sphere around the gripper of a graphically simulated robot and check for any interference between the sphere and other objects in the robot's work cell [16].

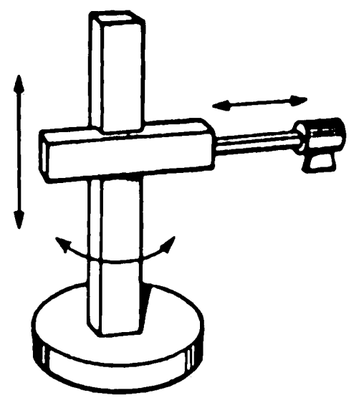
Figure 3-1 displays several common robot manipulator configurations. Given these four basic robot configurations, plans that succeed on spherical robot may fail on a rectangular robot simply because of the robot's physical

reach limitations. The TeachMover instructional robot, which is controlled by the MIC Planner System, is a 5-degree of freedom articulated joint spherical robot. Articulated joint robots closely model human arms and provide a wide range of motions compared to other robot arm configurations.



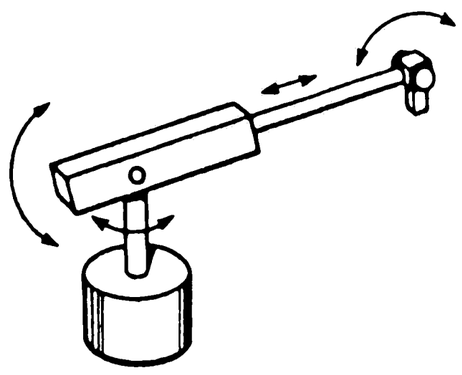
Rectangular
(Cartesian Coordinates)

a.



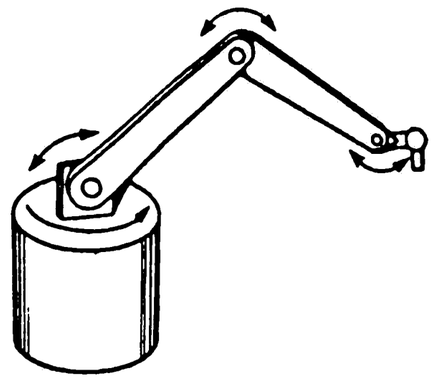
Cylindrical Coordinates

b.



Spherical
(Polar Coordinates)

c.



Articulated or Jointed Spherical
(Revolute Coordinates)

d.

Figure 3-1: Four basic manipulator configurations.

In addition to the problems generated by differences in arm configurations, one must take into account the freedom of movement of the end effector (hand and wrist configuration) of the robot. Figure 3-2 depicts the three basic wrist motions which are found in some robotic arms. The TeachMover robot only provides for *pitch* and *roll* motions. The side to side *yaw* motion can be simulated by rotating the gripper 90 degrees and then pitching the hand to either side.

In the MIC Planner System, all high level planning related to the move-

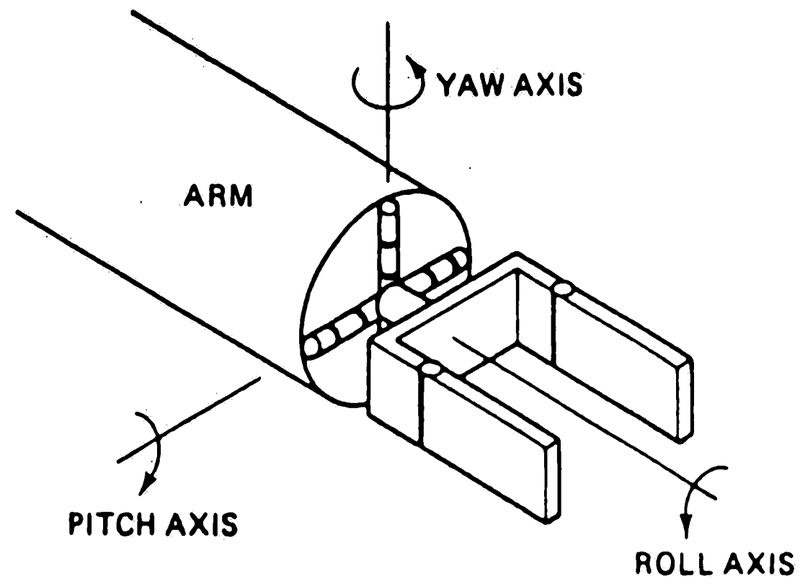


Figure 3-2: Three common end effector movements.

ment of objects by the robot's arm is done by the planner. The only thing that is not done by the planner is the actual conversion of XYZ Cartesian coordinate points into specific commands to move the attached robot. Instead, the planner generates two points (the current arm location and the new arm location that is to be attained) and writes them to an output file. The output file is then read by a secondary program, CALC, which performs the necessary coordinate to joint calculations and then sends the join movement information to the robot. While the arm is moving to its new destination, the planning system is halted. When the arm reaches its new destination, a handshake signal is sent back to the CALC interface program. If the handshake signal shows success, then the CALC program exits and the planner continues merrily along its way. If the handshake signal shows failure (in the case of an emergency stop), then the CALC program passes this info back to the planner and the planner acts upon it accordingly. In the case of a user generated emergency stop, CALC queries the TeachMover's controller for the current location of the arm and returns it to the planner. This insures that the planner knows where the

arm is at all times[2]

The CALC program, compiled in Turbo Pascal, calculates the coordinate to joint transformations needed to move the arm. The XYZ Cartesian coordinate points are first converted into intermediate joint angles and then into actual motor step counts that each joint's stepping motor must achieve in order to position the arm in the desired location. The motor step counts are then sent to the TeachMover's onboard microprocessor by a communications link. Upon receiving the stepping instruction, the robot's controller moves the arm to the destination point in a smooth but nonlinear fashion. During this process, CALC performs a more complete level of robot motion range checking as compared to the simple range checking done by planner. If the destination point specified by the planner is out of range, CALC returns this info to the planner and the arm movement fails. The entire MIC Planner System and external CALC interface program runs on an IBM PC or compatible MS-DOS microcomputer with at least 512K of memory and communicates to the robot via a single RS-232C serial link.

The points generated by the MIC Planner System actually consists of six values. The first thee values are the absolute X, Y and Z Cartesian coordinates (in inches) of the point in the world. Three additional coordinate features are provided. These are the *pitch*, *roll*, and *grip width* values of the TeachMover's gripper. Given this six-valued data structure for points, the planner can represent various object and their orientations within the robot's work

---

[2]Note, since the TeachMover does not provide absolute position resolvers on its joints, it is quite possible that the arm can go out of calibration. Thus the current arm location returned to the planner by CALC may not be the actual location.

cell. Each XYZ coordinates relates to the *center point* of the *bottom surface* each object in the world model. The *Pitch* and *roll* values allow the system to model the current orientation of each object. The *gripper width* field represents the width that the TeachMover's gripper must attain in order to grasp and hold a specific object. The following Prolog fact shows how the object's locations are represents in the MIC Planner System.

```
OBJECT    LOCATION(INCHES)   OBJ. ORIENTATION (DEGREES)
 NAME      X     Y     Z      PITCH  ROLL GRIP (INCHES)

LOC( PART1, [5.0, -5.0, 0.0], [-90.0, 0.0, 1.5] )
```

In addition to the orientation and location of objects, the planner also knows some simple features about each object. The current features known by the planner only include the dimensions of the bricks and cubes in the world model. This is shown in the Prolog fact below:

```
OBJECT    DIMENSIONS (INCHES)
 NAME      X      Y      Z
          LEN  WIDTH  HEIGHT

FEAT( PART1, [1.0, 1.5, 2.0] )
```

Thus, given the two Prolog facts above, the planner knows that *part1* is an object located at [5, -5, 0] with an normal gripper approach orientation having *pitch* of -90 degrees (gripper approaches this item from the positive Z axis), *roll* of 0 degrees (the gripper's roll is parallel to the Y axis), and a *grip* width of 1.5 inches (this is the same as the Y dimension). The planner knows that *part1* is 2 inches tall. Thus, any object placed on top of *part1* will have its Z axis coordinate component set to 2 inches.

3.2.2 Prolog as a Prototyping Language

The MIC Planning System is written in VMA Prolog on an IBM PC microcomputer. The Prolog programming language provides a good development environment for tasks which are easily described in terms of rules or procedures. Usually such a system requires a database facility for the many facts that the rules manipulate. Prolog provides a built in *inference engine* mechanism that can exhaustively search its database for all related facts. Thus, Prolog readily supports the construction of *expert systems.*

Because Prolog provides these features, the language leans itself to quick prototyping and testing of experimental software systems. This allows system developers to get a small but complete version new systems up and running usually in a matter of nours.

Firstly and most importantly, Prolog provides a built in database and inferencing engine facility. For this planning system project, the Prolog database proved to be quite sufficient in modeling the robot's world. In addition to the flexible format database, Prolog also provides built in predicates which allow the programmer to easily manipulate items in the database. These predicates allow the programmer to add and delete both *facts* and *rules.* Facts are declarative pieces of data that usually reflect the current state of the system. Rules generally contain procedural information that make up the Prolog program which manipulates declarative data items in the database. Many of the features provided for free by Prolog had to be developed separately in earlier planning systems such as BUILD [10] and NOAH. In the case of the BUILD system, Fahlman spent several months developing the necessary database structure and manipulation functions. To provide a similar Prolog database support facility in

another language, such as Pascal, considerable time and effort and would have to be given in coding the database inferencing functions.

Secondly, Prolog programs support a rule-based approach. Prolog rules control the inferencing mechanism by manipulating data elements in the database. These rules are generally made up of other user defined predicates, which may in turn be rules. This allows programs to be written in a top-down fashion where abstract rules are defined in terms of lower level primitive rules. A rule-based approach is a natural way to describe abstract assembly tasks that are the fundamental to developing a robot planner. High-level assembly tasks can be described as a series of conjoined lower-level subtasks or subgoals. These subgoals are refined in terms of lower predicates until a set of simple primitives are called upon to update data elements in the database (e.g., a block's location is modified when it is moved). Thus, by combining other predefined predicates, programming a hierarchy of decomposable abstract assembly tasks is very straightforward.

Thirdly, since Prolog is an interpreted language, it allows the programmer to quickly test newly defined predicates without the delay usually caused by compilation and linking. The PC based VMA Prolog used in this project compiled all Prolog predicates into a semantic network before execution[3] . This increases the execution speed of the planner up to acceptable limits. Even though the program is compiled, the programming environment is still interpreted. Thus, the programmer (and eventually the user) can easily interact with any part of the system, simply by typing any predicate name be it user defined or

---

[3]Virtual Memory Prolog, written by Robert Morein of Automata Design Associates.

built in. This allows the system developer to easily trace and debug new predicates that have been added to the planning system in an interactive fashion. The programmer can also assert new facts into the database as well as whole new rules. Thus, the programmer can link several planning predicates together to see their combined effect, and, if desirable, he can then save their combined effect as a new assembly predicate. This predicate can then be used later as a subtask of an even higher user defined predicate. This provides for a flexible and extensible programming environment that is not available in most 4th generation computer languages. Other AI languages such as LISP do provide for language extensibility and have interactive programming environments. But these other systems lack the built-in database and inferencing mechanism found in Prolog.

3.2.3 The TeachMover Robot

The MIC Planner System instructs the TeachMover robot to moves bricks about within its environment. Before further discussion on how the planner actual generates robot arm motion commands, it would be beneficial to look at the TeachMover's configuration, limitations, and features. This will provide a better understanding of the steps that have been taken to improved the programming interface to the TeachMover.

The TeachMover robot is a self contained robot manipulator and control system that is powered by an 8-bit 6502A microprocessor with 2K bytes of read-write random access memory. The 2K bytes of RAM can store programs created through the use of the hand held teach pendant or programs which have been generated off-line and down loaded to the robot from a host computer system. Figure 3-3 shows the various parts of the TeachMover robot and its coor-

dinate axis system [22].

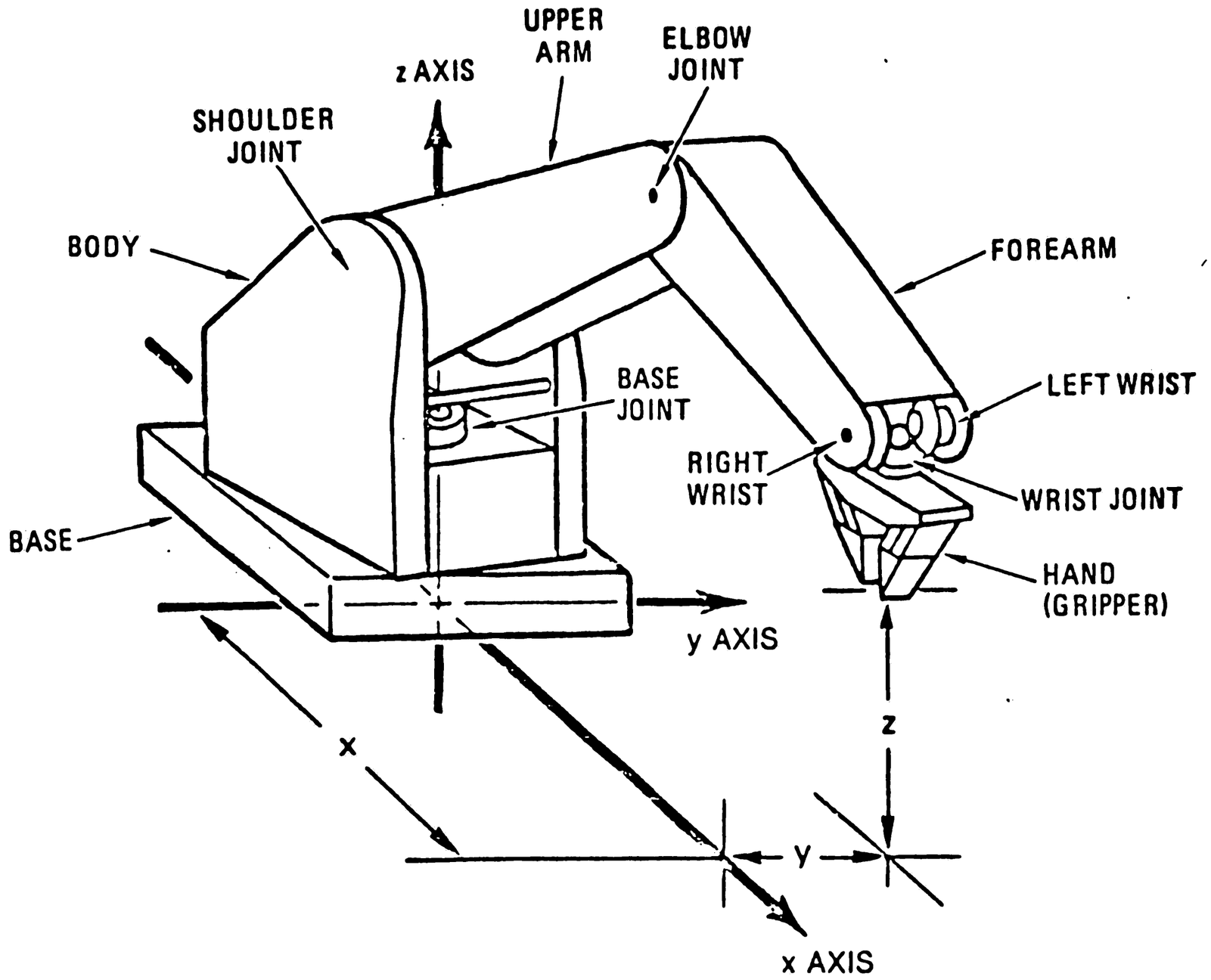


Figure 3-3: The TeachMover robot.

Simple point to point motions can be quickly and easily programmed via the hand held teach pendant. If the user records a point incorrectly, he can use the teach pendant to move the arm to the correct location and record the correct point. Then, once all the points have been recorded, the user presses the green start key on the teach pendant and the TeachMover moves through the previously taught series of points. The user can also perform some minor program editing of points with the teach pendant. Old locations can be reprogrammed by simply moving to a new location and pressing *record* on the teach pendant. The difficulty in editing increases as more robot instructions as

added, such as *GOTOs* and *JUMPs*. Here, the user must mentally record the program step number and refer back to it when he wants to change the flow of control in his program. This becomes quite difficult if there are many branching statements because there is no program listing to refer to. Also, new instructions can not be inserted into the program. The user can only overwrite existing TeachMover program statements. This unforgiving editing scheme was one of the major motivation for the development of the off-line robot control language, MIC.

With an off-line language, editing of control logic of the robot's program is much easier. With the MIC Compiler System, the user can edit his MIC program with an editor of his choice and then compile and down load his program to the TeachMover robot [24]. The MIC Compiler System checks for range errors which can occur when the arm is instructed to move outside the robot's reach. These error are logged to a listing file with the user's program and can be printed out to allow the user to review his logic or redefine his points to be within the robot's reach. For those interested in programming the TeachMover by using the teach pendant should refer to the *TeachMover User Reference Manual* by Microbot Incorporated [22] or to *Introduction To The Microbot TeachMover Robot* by Werkman [25]. Those interested in programming the TeachMover via the MIC robot control language should refer to *The Microbot Instruction Code Compiler* manual [24].

The TeachMover robot can be controlled from an external host computer by two methods. The robot can receive programs from a host computer and store them in memory (this is what the MIC Compiler System does) or the TeachMover can be controlled by a series of interactive commands. The second

method of control is incorporated into the MIC Planner System. The interactive commands range from simple point to point *STEP* commands to more complex commands such as *GRIP*, which close the robot's gripper until it squeezes an object, and *READ* which returns the robot's current arm location to the host computer. The MIC Planner System currently only issues *STEP* commands for every arm movement that is generated.

3.2.4 MIC Planning Task Examples

Since the MIC Planner System uses the Prolog interpreter itself as its user interface, the user has access all the features of Prolog when developing abstract assembly tasks. Thus, the user can execute predefined assembly predicates that exist in the planning system, or he can link several of them together to form new high level assembly predicates. The user can also look directly at the Prolog database to see what items are available for manipulation.

When the system is initially consulted, it initializes itself and sets up a sample world of three parts named *part1*, *part2*, and *part3*. These brick-like objects can be move around in the world by user initiated motion predicates. The user can *fetch* parts, *place* them *at* locations, *grip* parts, or even *insert* or *delete* parts from the world. If the user forgets what motion predicates are available for use, he can type *help.* at the Prolog system prompt and a screen of help will be displayed. Table 3-1 gives a listing of possible high and low level motion and assembly predicates that the user can issue.

In order to demonstrate an abstract assembly task, the planner has been taught a general procedure for assembling two parts at a given assembly location. The predicate *assemble* takes three parameters; the first part's name, the second part's name and a location at which the assembly is to take place. All

```
*= Robot Planning Commands from Highest To Lowest Abstraction Level =*
TASK   : stack(A, B, C, [X,Y,Z])  - Stack A on B on C at loc [X,Y,Z].
         assemble(A, B, [X,Y,Z])  - Assemble A and B at loc [X,Y,Z].
OBJECT: putOn( A, B )             - Puts A on top of B.
         fetchFromTo( A, [X,Y,Z]) - Fetch A and place at [X,Y,Z].
         mate( A, B )             - Fetch A and mate to B.
         fasten( A, B )           - Glue A to B (A becomes part of B)
         fetch( Object )          - Move arm to Object and grip.
         placeAt( [X,Y,Z] )       - Move arm/grip contents to [X,Y,Z].
ARM    : approach( [X,Y,Z] )      - Safely move arm to [X,Y,Z].
MOTION   depart( DistReturned )   - Raise arm at least 5 in. in Z axis.
CMDS     moveArm( [X,Y,Z] )       - Move the arm directly to [X,Y,Z].
         rotateGripper(Degrees )  - Rotate grip (& object) + or - Deg.
         grip                     - Closes grip on object at grip loc.
         unGrip                   - Opens grip & release grip contents.
OTHER:   world                    - Show the current world items.
USER     remove( Object )         - Removes an Object from the world.
CMDS     insert( Object )         - Inserts an Object into the world.
         speed( 0..15 )           - Sets arm speed for next move.
         connected( yes/no )      - Connect & Move/disconnect robot.
         saveMoves( yes/no )      - Saves MIC moves to file MOVES.MIC.
         traceMoves( yes/no )     - Toggle ON/OFF trace of arm motions.
```

Table 3-1: MIC Planner System Predicates.

locations are given in terms of real numbers and always includes X, Y and Z Cartesian coordinates in units of inches. The user initiates the assembly predicate as follows:

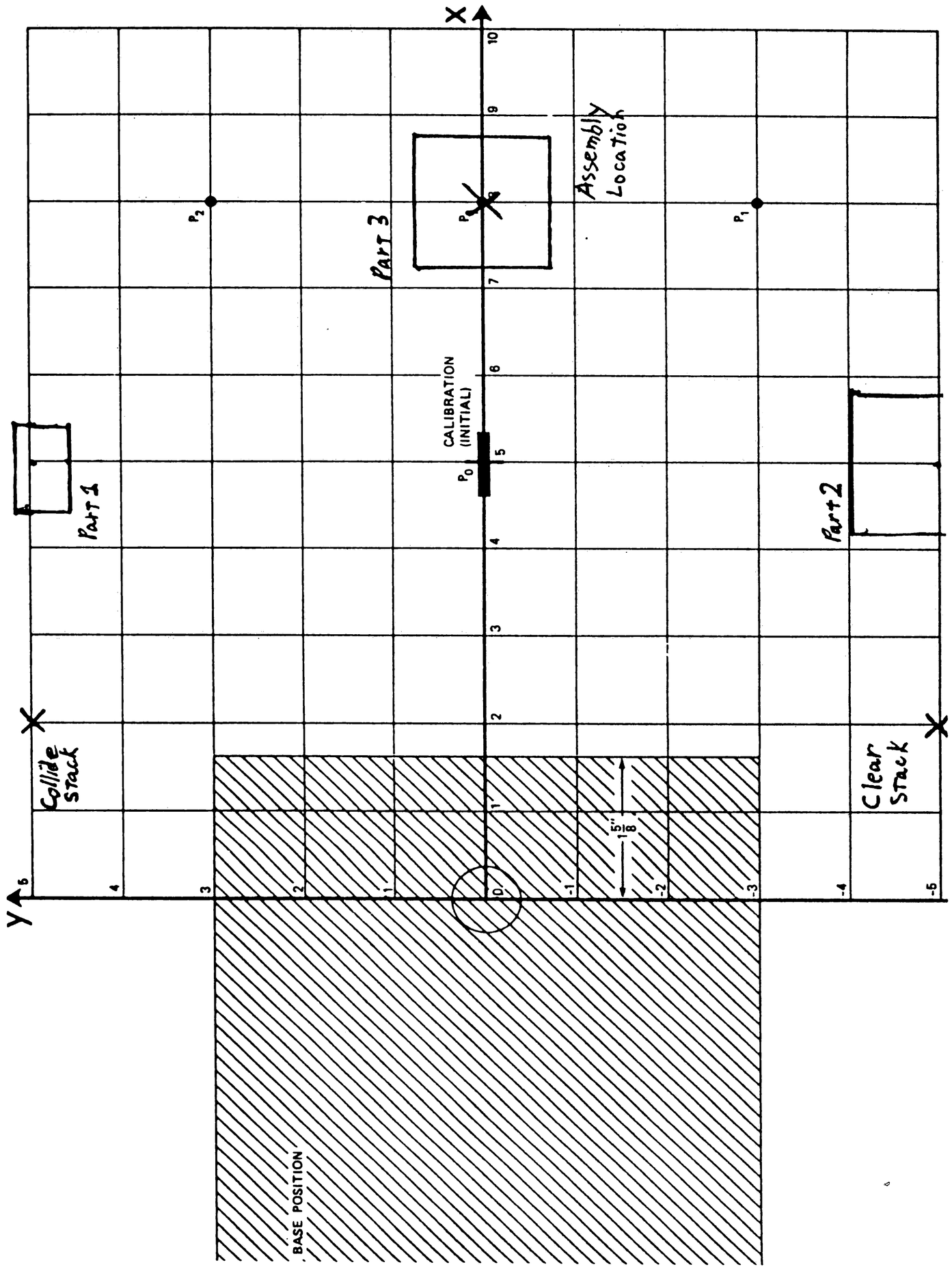
```
ASSEMBLE( PART1, PART2, [ 8.0, 0.0, 0.0 ] ).
```

The above predicate will attempt to assemble two objects, *part1* and *part2* at the predetermined location [8,0,0]. Figure 3-4 shows the layout of the world at the start of the assembly operation. The *assemble* main task predicate is divided up into four general actions.

1. Fetch *part1* and Place At [8,0,0].
2. Fetch *part2* and Place At [8,0,0].
3. Mate *part1* to *part2* (make a nonpermanent bond).
4. Fasten *part2* to *part1* (make one object).

In the MIC Planner System, the *assemble* predicate is programmed in Prolog as follows:

Figure 3-4: Initial state of the world before *assemble*.



```
assemble(A,B,AtLoc):-
   print('\nAssembling',A,' and ',B,' at ',AtLoc,'.\n'),
   asserta(motionCommand(assemble,A,B)) /* Set Assemble Flag   */
   fetchFromTo(A,AtLoc),                /* Fetch A to AtLoc    */
   fetchFromTo(B,AtLoc),                /* Fetch B to AtLoc    */
   mate(A,B),                           /* Mate (snap) A to B  */
   fasten(B,A),                         /* Fasten (glue) A to B*/
   retract(motionCommand(assemble,A,B)) /* Reset Assemble Flag.*/
   print('Assembly Task: Completed.\n'),/* Print out completion*/
   ! .
```

The first two part fetches are done by the Prolog predicate *fetchFromTo(Object_To_Fetch, Location_to_place_at)*. This predicate fetches an object from it's current location and places it at the specified location. This entire abstract task must be done in a fairly safe manner in order to avoid collisions between the arm and objects in the robot's world. After the two parts are brought together at the assembly point, the planner calls *mate(Object1, Object2)*. This instructs the robot to nonpermanently connect the two objects. The final subtask of the *assemble* main task is the fasten operation. This occurs when the *fasten(Object2, Object1)* predicate is called. This causes *Object2* to become permanently fastened to *Object1*. The additional clauses listed above simple print out informative messages and set and remove flags which are needed for lower level tasks during the assemble main task.

The first subtask of the assemble command requires the planner to tell the robot to ***fetch*** the first part, *part1* and place it at the specified assembly location, [8,0,0]. But, another object (*part3*) is already at this location. Therefore a planning conflict occurs and the planner must react in some fashion to prevent the current Prolog predicate from failing, and consequently causing the main *assemble* predicate to fail. The planner knows that the assembly of *part1* and *part2* must take place at location [8,0,0], possibly because there is a special mounting fixture at that location which is needed for the assembly. The plan-

ner corrects the conflict by clearing the assembly location of *part3*.

But before location [8,0,0] can be cleared of *part3*, the planner must rid the robot's gripper of the current object (*part1*) so that it can grab and remove the offending object. So the planner clears the gripper of the current object by placing it (*part1*) at a predetermined location known as the *clear stack*. The robot moves the arm to the clear stack location and places *part1* there. The result of this operation is shown in Figure 3-5. After the gripper is cleared, the planner tells the robot to go back to [8,0,0] and fetch the offending part (*part3*). This part is then placed at a secondary location known as the *collide stack*. The collide stack is where objects are placed when they are found occupying a location that they should not be in. This stack grows as the number of incorrectly placed objects are found to be in the wrong locations. The result of clearing the offending object (*part3*) to the collide stack is shown in Figure 3-6.

After *part3* has been placed at the collide stack location, the planner knows that it must restore the object that was in the gripper before the collision detection occurred. So the arm returns to the clear stack location and fetches *part1*. The planner finally instructs the robot to place *part1* at the newly cleared location [8,0,0]. At this point, the first *fetchFromTo* subtask has been successfully completed. The results of the first *fetchFromTo* subtask can be seen in Figure 3-7. The second *fetchFromTo* subtask fetches *part2* and places it on top of *part1* without any problems.

The reader might wonder why the planner does not also clear the assembly location ([8,0,0]) of *part1* like it did when it found *part3* there. However, a special "assembly condition", asserted into the Prolog database when

Figure 3-5: World after collision detected and grip cleared.

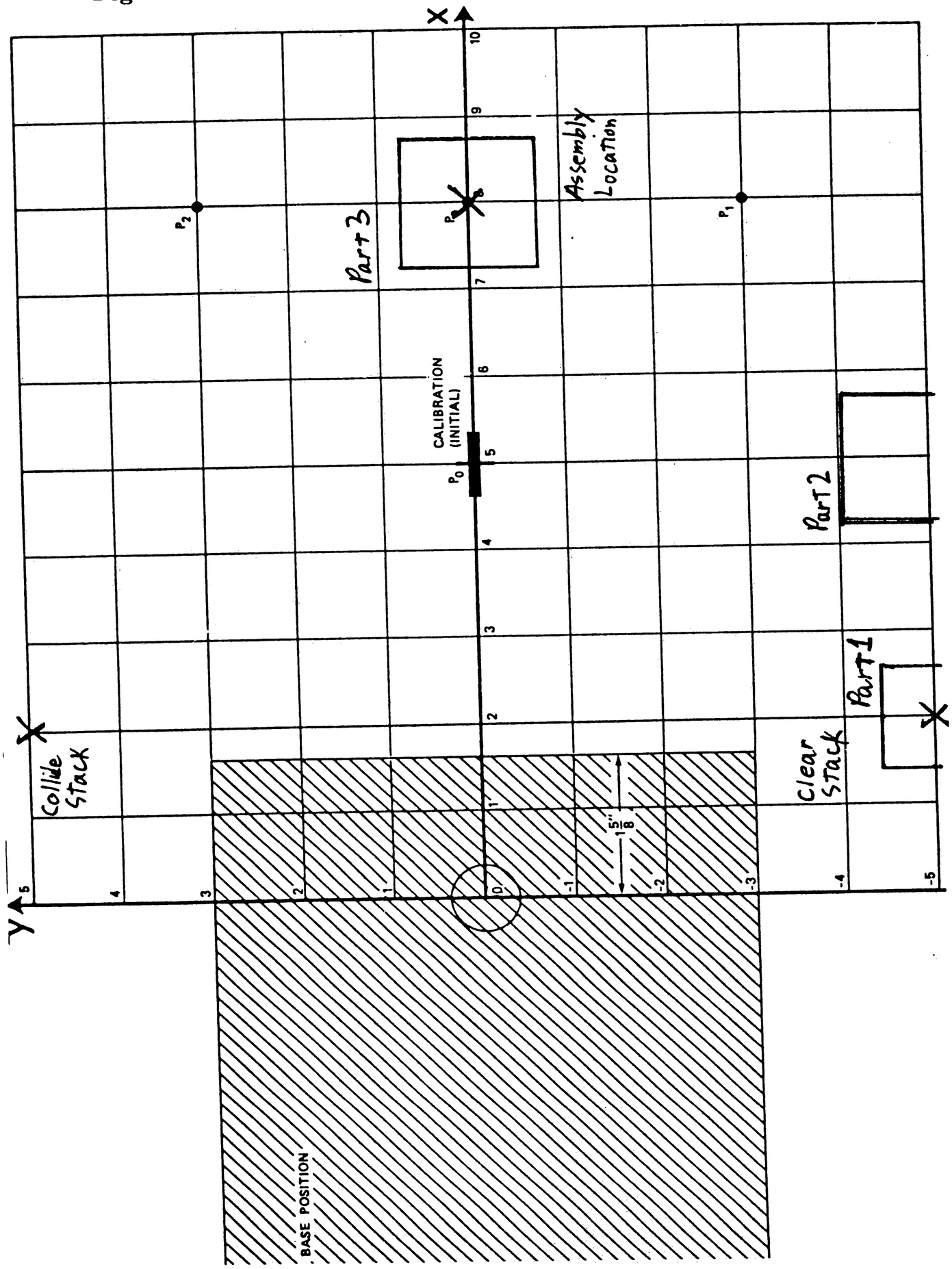


Figure 3-6: World after *part3* cleared from assembly location.

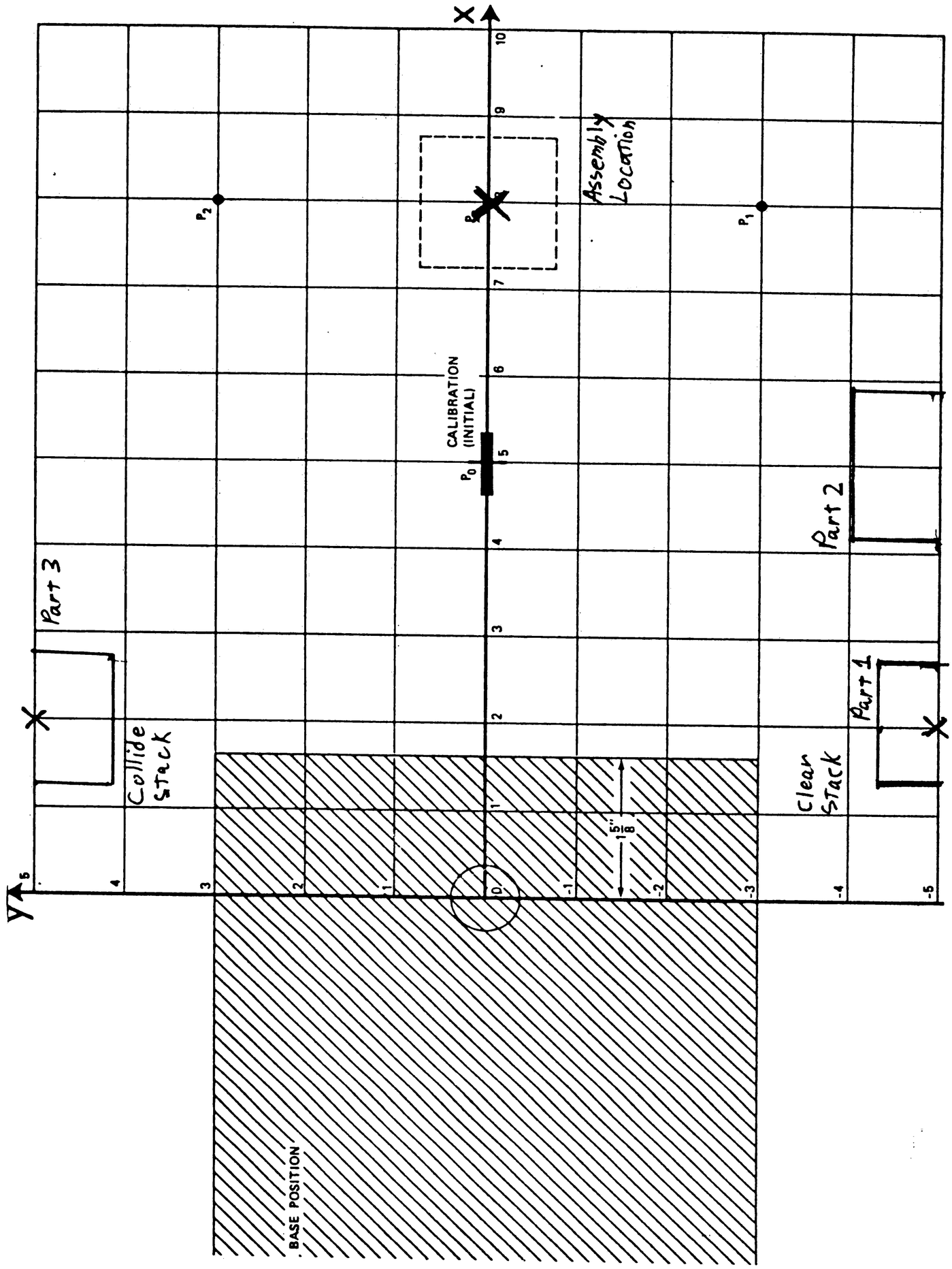
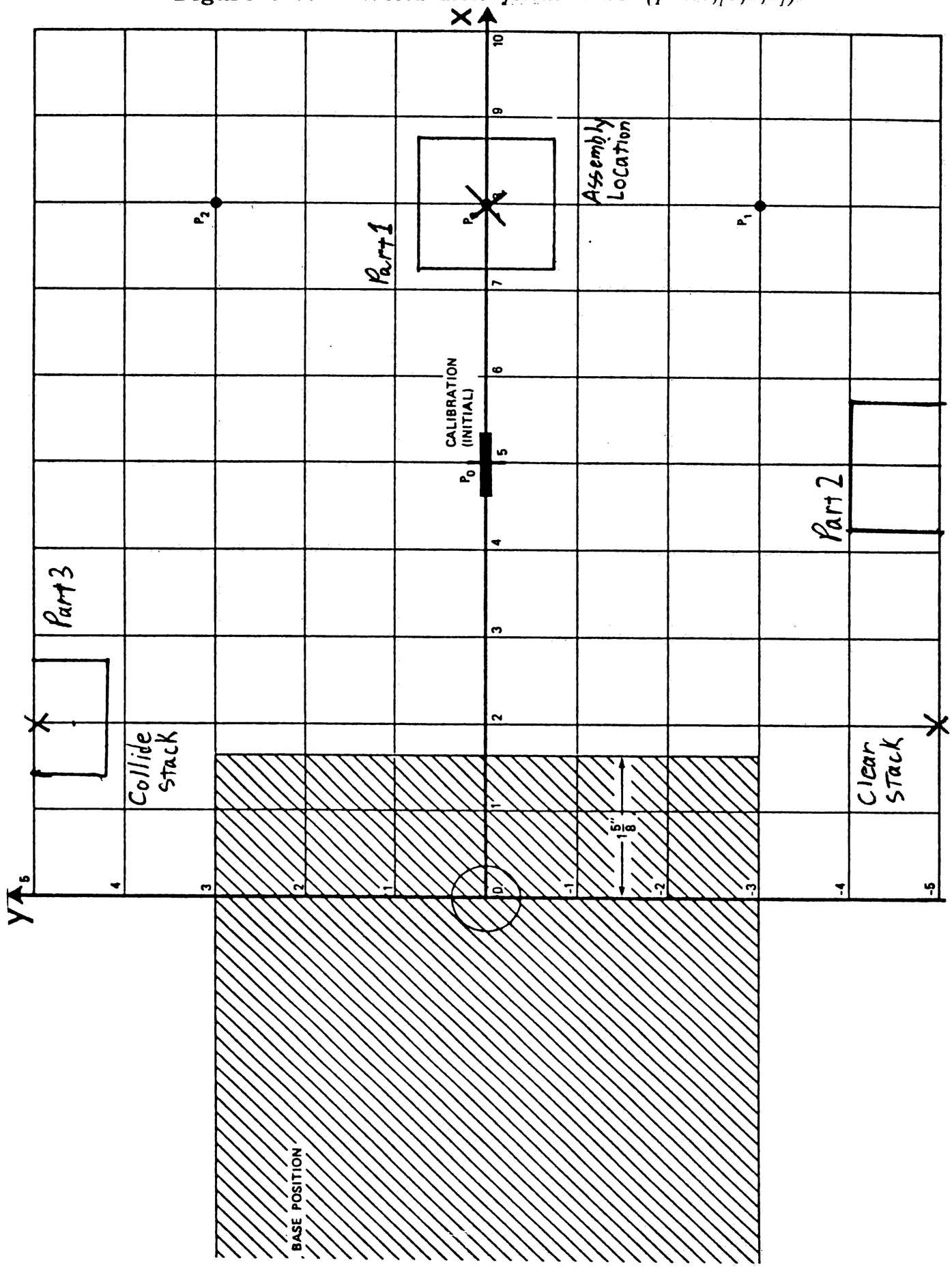


Figure 3-7: World after *fetchFromTo(part1,[8,0,0])*.

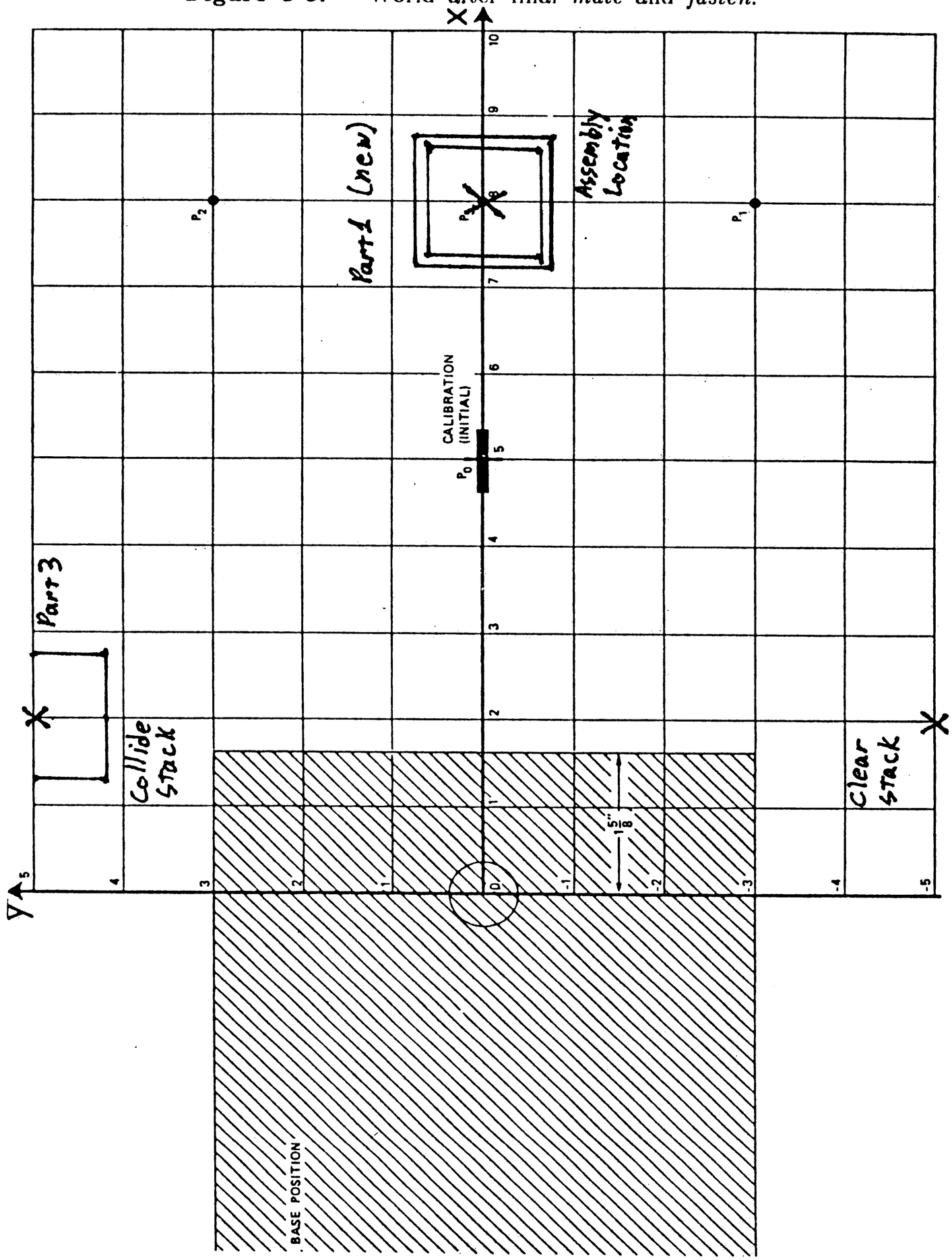


the *assemble* predicate was called, does not exist anymore. This specific "assembly condition" is built in and checked for by lower level arm movement predicates. If a specific collision condition occurs, then the arm movement predicates resolve it by removing the offending object. If the specific "assembly" condition is not in effect, then the arm movement predicates can resolve the collision problem by simply placing the currently held object on top of the object that happens to be at the destination location. Passively placing an object at a specified location, the default condition, allows the robot to safely complete its task with a minimum of damage to its environment, and a minimum number of arm movements.

The third subtask of the assembly task requires that the planner instruct the robot to perform a ***mate*** operation. In this particular assembly operation a mate subtask is the joining of two objects in a nonpermanent fashion. The *mate* predicate causes the robot to fetch *part2* and twist it 90 degrees until its simulated screw locking mechanism clicks. Now the parts are mated. They can be *unMated* by rotating the gripper in the opposite direction.

The final subtask of the main assembly task is the ***fasten*** operation. In this task, *part2* (which is on top of and mated to *part1*) is melted together with *part1* to form a single object, *part1*. The final state of the world is depicted in Figure 3-8. To maintain a consistent world model, the planner removes *part2* from the Prolog database since it now no longer exists (its now part of *part1*). The planner also must realize that *part1* has "grown" in its Z axis dimension to now include it's previous height plus the height of *part2*. The planner also updates the X and Y axis dimensions of *part1* to reflect the addition of *part2*.

Figure 3-8: World after final *mate* and *fasten*.



A complete trace of the above assembly task can be found in Appendix A through E. Appendix A lists the normal user readable output generated during a planning operation of the assembly task. Appendix B provides the reader with a verbose trace of the same assembly operation showing the exact points that the robot's arm moved through while assembling *part1* and *part2*. Appendix C lists the *MOVES.MIC* output log file and is generated by the planner when the *saveMoves(yes)*. predicate is invoked. This file contains valid MIC robot programming commands that can be compiled by the MIC Compiler System written by Werkman [24]. These commands can then down loaded to the TeachMover robot to provide native code for real time execution of the assembly task. Appendix D shows the listing output generated by the MIC Compiler System after compiling the program in Appendix C. Appendix E shows the actual TeachMover opcodes that were generated by the MIC Compiler System.

Appendix F provides a trace generated by the planner for a stacking operation in which three parts of a stack are rearranged. This stacking operation involves a worst possible case condition that is handled efficiently by the planner. The initial state of the stack is shown on the left side in the example below. The final state is shown on the right side.

```
START          FINISH
ORDER          ORDER
------         ------
PART 2         PART 1
PART 3  ===>   PART 2
PART 1         PART 3
```

The user initiates the stacking operation by issuing the following Prolog predicate:

```
LOC( PART1, STACKLOCATION, _ ),
STACK( PART1, PART2, PART3, STACKLOCATION).
```

The *loc* predicate causes the Prolog database to be queried for the XYZ

location of *part1*. The location is returned in the variable *StackLocation*. This variable is then used as a parameter in the *stack* predicate. The *stack* predicate in the above example states that *part1* should be placed on top of *part2* which should in turn be placed on top of *part3*. The location of the stack should be at *StackLocation*.

To perform this task, the *stack* predicate is defined in Prolog as follows:

```
stack(A,B,C, AtLoc) :-
  ( ( (A == B; B == C; C == A),  /* If 3 items NOT UNIQUE,     */
       print(A,' and ',B,' and ',C,' are not unique! Aborting.\n'),
     !, fail                     /*  Then QUIT!                */
     )
    ;
     (asserta(motionCommand(assemble,C,B)),/Set Assemble Flag */
      print('STACKING:',A,' ON ',B,' ON ',C,' at ',AtLoc,'.\n'),
      fetchFromTo( C, AtLoc ),    /* Put C at location AtLoc.   */
      putOn( B, C ),              /* Then put B on top of C,    */
      putOn( A, B ),              /* And put A on top of B.     */
      retract(motionCommand(assemble,_,_)),/*Reset Assem Flag.*/
      print('Stack Operation: Completed.\n')/*Print completion.*/
      )
  ), !.
```

The first action that is performed once this predicate is called is a check to make sure that the three objects being stacked are all unique objects. If they are not, a warning message is printed and the *stack* predicate fails. If *stack* had been called by another predicate, then that predicate would also fail. The failure of the stack predicate in this case is justified because this predicate, *stack*, requires three objects to manipulate. Chances are that any predicate that calls stack also expects to find three unique objects in the world and would also fail.

This condition of high-level predicates failing more quickly than low-level predicates is a general rule among the hierarchy of predicates in the MIC Planner System. The low-level arm motion predicates tend to deal with abnormalities of greater subtlety than do the higher level more abstract action predi-

cates. This occurs because many gross errors are initially screened out at the higher levels of abstraction where they are most obvious. The low-level motion predicates are coded to deal with more subtle problems such as in the case where objects are passively stacked on top of other objects when their destination location is occupied.

Given the case that all three object parameters of the *stack* predicate are unique, the next action is to assert the "assemble flag" which enforces the condition that the first object to be placed is indeed placed at the location specified, *StackLocation*. This same condition was also used in the assembly task example discussed above. After this condition is inserted in the the database, an informational message is displayed.

The stacking task is divided into three main subtasks:

1. Fetch *part3* and Place At *StackLocation*. (Causes any objects on top of *part3* to be removed.)
2. Put *part2* on top of *part3*.
3. Put *part1* on top of *part2*.

In an attempt by the planner to place *part3* at *StackLocation*, it must first resolve the conflict that *part2* is on top of *part3*. Given this condition, the robot can not physically grasp *part3*. Thus, *part2* is cleared to the *Collide Stack* location as seen below.

```
MAIN                COLLIDE
STACK                STACK
------              -------
------              -------
PART 2 ===>
PART 3
PART 1              PART 2
```

Now, *part3* is accessible by the robot and can be placed at the destination location. But, it just happens that the destination location is where *part1* cur-

rently resides. Thus, since we have enforced the "assembly flag" condition which states that the object to be placed (*part3*) must occupy the specified location (*part1's* residence). The conflict is resolved by the planner telling the robot to clear the gripper of *part3* and then place *part1* on the *Collide Stack*. This is seen in the following two examples.

```
CLEAR          MAIN          COLLIDE
STACK          STACK          STACK
------         ------        -------

          <===  PART 3
PART 3         PART 1        PART 2

            * THEN *

CLEAR          MAIN          COLLIDE
STACK          STACK          STACK
------         ------        -------

                             PART 1
PART 3         PART 1  ===>  PART 2
```

After the stacking location has been cleared, the planner restores the object that was previously in the gripper (*part3*) and then completes the original *placeAt* subtask and places *part3* at *StackLocation* as seen below.

```
CLEAR          MAIN          COLLIDE
STACK          STACK          STACK
------         ------        -------

                             PART 1
PART 3  ===>   PART 3         PART 2
```

The next subtask of the *stack* main task predicate is ***putOn(part2, part3)***. But the execution of this task is hampered by the fact that *part1* is on top of *part2*. Thus, the robot can not get at *part2*. As done earlier, the planner clears the top of the current object being fetched. This time, the standard *Collide Stack* location can not be used because this would not help resolve the current conflict. Instead, this would just put *part1* back on top of *part2*. At this point, the planner realizes that it must find another "clear spot" to place

the offending object. It decides to place the object at the location, *NearBy*. This action is depicted below:

```
NEARBY          MAIN            COLLIDE
STACK           STACK            STACK
------          ------          -------

                <==================  PART 1
PART 1          PART 3          PART 2
```

This location is generated by taking into consideration simple volumes of bricks that exist in the world. The planner knows the XYZ dimensions of *part1* and *part2*. It decides to place *part1* just outside of the collision range of *part2*. In this case, the planner always tries to place the offending *Collide Stack* object near the X axis which is more toward the middle of the robot's work cell and should thus be within the range of the robot.

Now that all objects on top of *part2* have been cleared, the planner instructs the robot to resume the *putOn* subtask and place *part2* on top of *part3*.

```
NEARBY          MAIN            COLLIDE
STACK           STACK            STACK
------          ------          -------

                PART 2
PART 1          PART 3   <===   PART 2
```

The final subtask of the *stack* predicate, *putOn(part1, part2)* is then executed and succeeds. The final state of the world is show below.

```
NEARBY          MAIN            COLLIDE
STACK           STACK            STACK
------          ------          -------

                PART 1
                PART 2
PART 1   ===>   PART 3
```

THE GOAL STATE CONFIGURATION

As can be seen in the examples above, the assembly and stacking tasks are treated by the MIC Planner System as a conjoined series of ordered subtasks that must be achieved in the given sequence in order to satisfy the main

action predicates, *assemble* and *stack*. In the specific case of the assembly task, the given order of the subtasks ***cannot*** be changed in the event that a subtask fails. Each predicate in the MIC Planner System is designed to succeed given minor perturbations in the robot's world. In several cases, the planner will ask the user if he wants to perform a specific operation if it is deemed inappropriate by the planner. If the user agrees to a inappropriate operation, the planner will try its best complete its primary task by dealing with the resulting world model. Only when an impossible situation exists will the planner give up (e.g. when the user tries to fetch an object that is not in the world).

The MIC Planner System is capable of successfully handling perturbations in it's environment for two reasons. First, its assembly tasks are well understood and take into account all possible task exceptions. Second, each assembly task is represented in a planning hierarchy of subgoals where each subgoal relies on the successful completion of lower level subgoals. Such a task description hierarchy allows the human task planner to clearly describe tasks in terms of easy to understand primitives.

3.2.5 Comparison with the NBS Robot Control System

The NBS Robot Control System, as described above, uses a hierarchy of task decompositions to successfully control robots and other machine tools through their manufacturing operations. The discussion here centers on the lower four levels of the NBS RCS hierarchy where parallels in task decomposition can be found between both the NBS and MIC Planner Systems. The lower levels of the RCS include the *Work Station* level, the *Task* level, the *E-Move* level, and the *Primitive* level.

Figure 3-9 shows the lower four levels of the NBS RCS that were used in

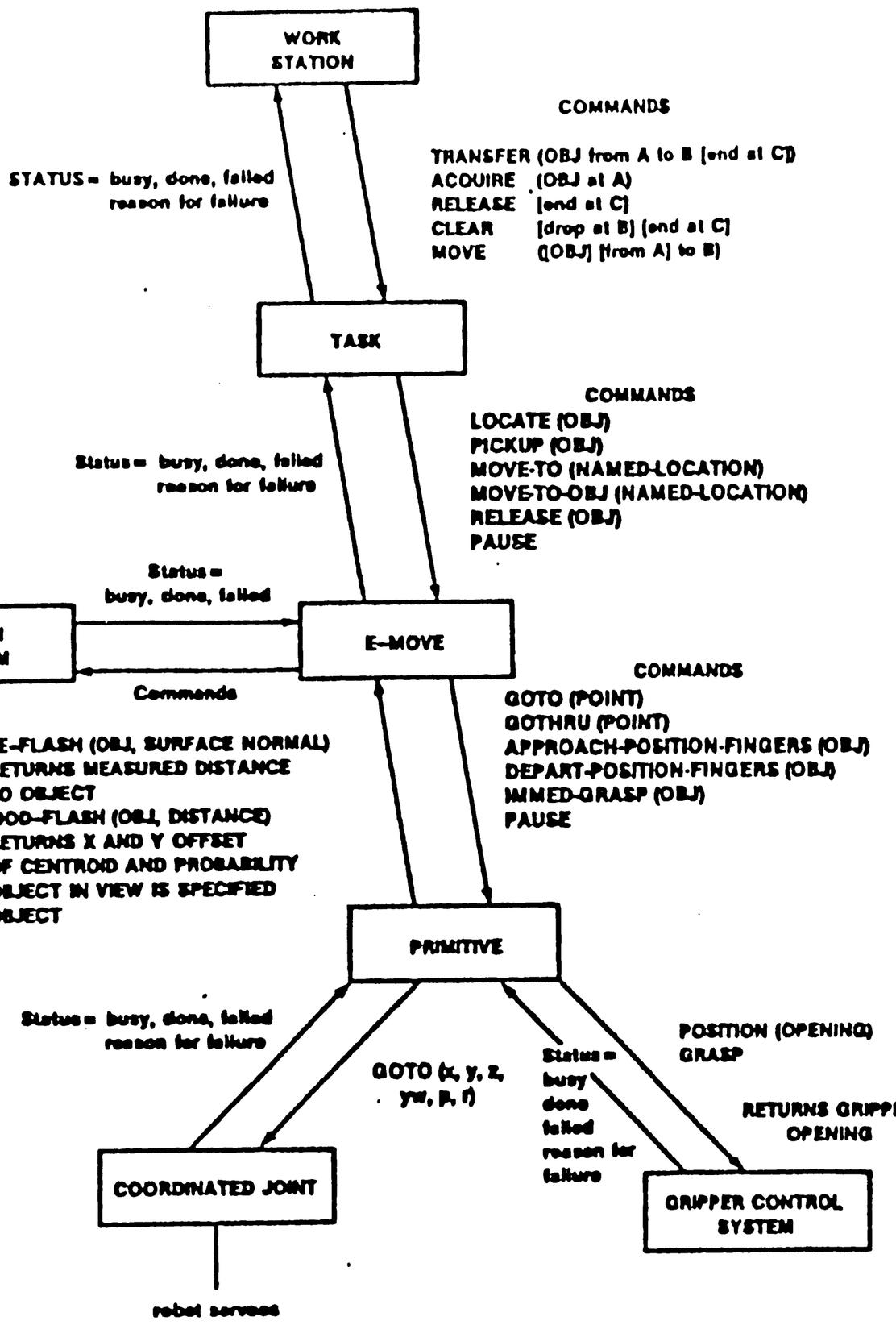


Figure 3-9: RCS commands used at each task level.

the Automated Manufacturing Research Facility (AMRF) project. The *Work Station* level allows the operator to specify fairly abstract motion actions such as [14]:

- *TRANSFER([Obj from A] to B [end at C])*, robot acquires *Obj* from A, moves to location B, releases the *Obj*, and then moves to location C.

- *ACQUIRE (Obj at A)*, robot moves the arm from its current position to the *Obj's* location at A and grasps the *Obj*.

- *RELEASE / end at A /*, the robot releases the object it is currently holding and moves to location A.

- *CLEAR / drop at B / / end at C /*, the robot releases the currently held object at location B and then proceeds to location C.

- *MOVE (/ Obj / / from A / to B)*, causes the robot to move from location A with or without *Obj* to location B. The *Obj* is not released.

These actions are in turn defined in terms of lower level actions that the *Task* level sends to the *E-Move* level.

To see just exactly how a *Work Station* command is actually performed, we will examine the decomposition of the *TRANSFER* task defined at the *Task* level in the control hierarchy. The *TRANSFER* task requires that 1) the robot locate an object at location A, 2) move the object to location B, and 3) then optionally end at location C or a safe point above location B[4] . To locate an object in the work cell, the system has to interact with a vision system. This is done through the *LOCATE(Obj)* subtask which is defined at the *E-Move* level in the control hierarchy. Once the object is found, the system has to move the robot's arm to the object and grasp it. This is done by first calling the *E-Move* subtask *MOVE-TO-OBJ(Location)*. Now that the arm is at the object's location, the object must be grasped. *PICKUP(Obj)* is called to perform this subtask.

The system now decomposes the second part of the *TRANSFER* task. This requires the robot to move to location B while holding the object. So, the *E-Move* level subtask *MOVE-TO(Location)* is called. Once at location B, the object must be released before the arm can proceed to location C. The *E-Move*

---

[4]A *safe point* is a term used to describe a location usually vertically above a previous point. This point is considered "safe" in that it is out of the way of other objects in the robot's work cell.

subtask *RELEASE(Obj)* does this. The last part of the *TRANSFER* task moves the robot's arm to location C (or a safe location if C is not specified), so again the *E-Move* subtask *MOVE-TO(Location)* is called. Finally the *E-Move* level's *PAUSE* subtask signals the completion of the *TRANSFER* task. Each of the arm motion subtasks defined at the *E-Move* level's is in turn decomposed into lower level arm movement subtasks defined at the *PRIMITIVE* task level (level 1 in the hierarchy). Tasks defined at this level include more refined motion operations such as *GOTO(Point)*, *GOTHRU(Point)*, *APPROACH-POSITION-FINGERS(Obj)*, *DEPART-POSITION-FINGERS(Obj)*, *IMMED-GRASP(Obj)*, and *PAUSE*.

There are similar task operations and arm movement primitives in MIC Planner System. Analogous to the NBS *TRANSFER* task is the MIC Planner's *fetchFromTo* predicate.

FETCHFROMTO(OBJECT, LOCATION).

Here, the user specifies an *Object* to fetch and then a *Location* at which to place the object. The *fetchFromTo* predicate is decomposed into two subtasks, *fetch(Object)* and *placeAt(Location)*. The *fetch(Object)* predicate initially locates the object in the world model and then causes the robot to move to the object and grip it. To do this, *fetch* is further decomposed: *exists(Object,Space)*, *approach(Space)*, and *grip*. The *exists* predicate checks to see if the object exists in the database. If it does, then the location *Space* is returned. This information is then passed to the *approach* predicate which moves the arm to the specified location safely. Once there, the *grip* predicate instructs the robot to grasp the object at the current location.

The second part of the *fetchFromTo* main task is the *placeAt* predicate.

PlaceAt contains three major subtasks: *approach(Location)*, *unGrip* and *depart*. After the object is placed at the specified location, it is released and the arm is instructed by the planner to back away to a safe point location. *fetchFromTo*, thus performs motion actions much like the NBS system's *TRANSFER* task. Both include task decomposition hierarchy and both use task level feedback.

Each one of these task levels in the NBS Robot Control System hierarchy receives feedback from its surrounding control levels at clocked time intervals. The feedback comes from three sources: commands from higher level control modules, status results of commands executed by lower level modules, and sensory feedback from external sensors and vision systems as shown in Figure 3-10. An example would be the following. The *Task* level receives abstract action commands from the *Work Station* level above. The *Task* level processes these command by decomposing them into lower level task commands and sends these task commands to the *E-Move* level below. The *E-Move* level further decomposes the task commands into elemental robot movement commands such as *GOTO(Point)*. If the robot can successfully attain this point, a signal stating this result is then passed back to the *Task* level.

Each control level is made up of related task processes. The process is the basic element of the NBS system. A process is divided into three parts; an input, the process, and an output [14]. The input part of a process formats the incoming command into the desired format needed by that process. The output part converts the data into a format usable by other processes in the system. The actual process itself is a state table that is divided into input states with associated output functions. This state table structure depicted in Figure 3-11 is the same for each task level in the NBS RCS system.

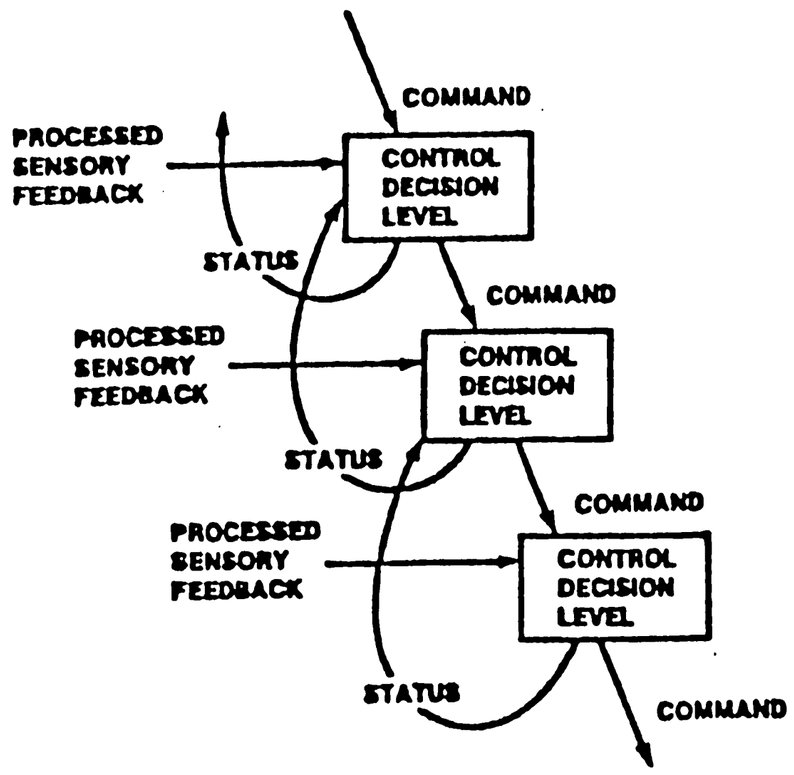


Figure 3-10: Input/output to/from each hierarchical level.

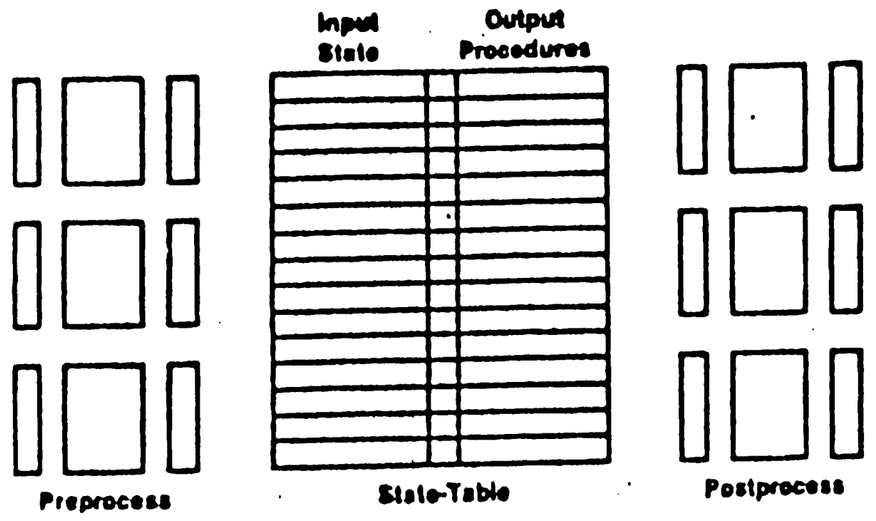


Figure 3-11: A control level in the NBS System.

The NBS researchers chose the state table representation for each task level in the NBS control hierarchy for several reasons. First, they believe that any procedurally written robot program can be rewritten in terms of a finite-state automaton (FSA) [1]. This has several benefits. First, it makes for an explicit means for expressing simultaneous processes which are occurring at each level of the control hierarchy. Secondly, a FSA facilitates explicit error handling conditions. Thirdly, is allows for the addition of new sensor input devices by simply adding new lines into the state table. A fourth advantage to the FSA approach is that is provides a formalized an structured approach to

describing each task level. Thus, as in the case of adding new sensor input or new error condition checking, the programmer simply adds new lines (rules) to the state table much like he would add rules to an expert system. A final advantage to using FSAs and state tables over procedural robot programming languages is that it is easier to debug.

The MIC Planner System demonstrates that a rule-based approach can provide some of the benefits of the state table approach used by NBS in their RCS. Though there is explicit state table, descriptions of how task are decomposed maybe more straight forward in terms of Prolog rules than in terms of state tables. The ability to increase the planner's knowledge with new rules is also easily done in Prolog. Given Prolog's automatic backtracking control scheme, once a predicate fails, the MIC Planner System would attempt to use other versions of the failed predicate that it had in its database in order to find a solution. Also, Prolog provides an interactive environment with a built-in trace facility. This allows the programmer to easy debug his task descriptions.

3.2.6 Possible Future Enhancements

The MIC Planner System was designed to operate under the "STRIPS assumption" in that the world is static and only action predicates change items in the world. A worthwhile enhancement to the system would be to add real-world feedback which would allow it to react to changing world conditions. Thus, given continuous visual input from a simple overhead mounted vision system, the planner could keep track of objects at least in X-Y space on the table top. If a second side-mounted vision system were added, the planner could also use this input to decide exactly where objects were in 3-space. Possibly the vision system could report the specific dimensions of the objects in both the X-Y

plane and Z axis. Once a ***fasten*** operation is performed, the planner would have to inform the vision system monitoring the Z axis that the two objects are now to be considered as one. Thus the Z axis dimension of height returned to the planner would have to reflect both the top and bottom parts' heights.

The MIC Planner System could easily be modified to use various sensor inputs given the current hierarchical structure of the programmed tasks. Ultimately, the lowest level arm motion tasks would be responsible for checking the availability of a destination location before an object is placed. For proper use of this new world knowledge, a outer level monitor program would have to be written which would continuously poll for new feedback. Possibly the input sensors would generate specific "world update" predicates that would be placed in a file and read by the Prolog monitor system[5]. The planner's "blackboard" would contain specific slots (predicates) that could be updated directly by the external sensor system.

The MIC Planner, like the NBS system, can behave in an *opportunistic* fashion. In the field of real-time robotics, complete plan generation before the system makes a move may not be the best route to go. Instead, a piecemeal generation of the plan, as in the opportunistic approach may be more flexible and efficient. Indeed, many current experimental planning systems are starting to take into account real-world constraints such as time dependent travel and deadlines. If a planning system is ever to be used for a real-world application such as robotic control, it will have to deal with the many uncertainties that arise in a changing world.

---

[5]This case of communicating via files is specific to the current implementation of Prolog for the PC used in this project. This may change as new versions of Prolog become available

The use of production rules is seen an an important step in the development of advanced planning systems. A real-time rule-based planning system called the Flexible Planning System (FPS) has been developed, implemented and tested in a real-world environment on the HILARE robot [18]. This rule-based system allows for goal-directed as well as data-directed processing to occur. The FPS system uses the STRIPS assumption in that all updates to the world model are done exclusively through production rules. The FPS system allows for parallel execution of these rules in real-time. The planning method used is similar to the NOAH system in that rules build a procedural net of possible choices. A set of critics then examine and expand promising nodes in the network. These critics select goal expansion operators dynamically utilizing preconditions (environmental context), current conditions, post-conditions, and constraints [18]. The system is made "smarter" by simply adding new rules to its rule base. Because the system planning knowledge is represented in the form of rules, heuristics may be added at any level of the planning hierarchy without much trouble.

The FORBIN planner is a planning system that use spatial and temporal reasoning to generate plans to deal with deadlines and travel time of a mobile robot [15]. Here, a mobile robot is used to move about the factory floor supplying various machines with supplies and machined parts. The FORBIN system uses a hierarchical structure much like NOAH's but modifies its plans based on time. Two new modules are incorporated into the system to do this; the *Time Map Manager* TMM and the *Time Optimizing Scheduler* TOS. As deadlines pass, the system reorders its subgoals by looking up temporal information in the TMM module and then explores the possible repercussions of the subgoal

reordering via the TOS module. A feasible extension to an opportunistic version of the MIC Planner System would be to provide a means for the system to deal with real-time constraints for given assembly operations.

Chapter 4
Using Natural Language To Control A Robot

The next level of research in developing intelligent robotic systems will be the enhancement of user interface. An enhanced user interface might include a speech recognition system that translates words and phrases spoken by humans into ASCII text in computer systems. Indeed, there are several system now available on microcomputers that recognize the voice input of individuals. Many of the currently available systems record a series of words spoken by a single user and then let the user assign a secondary microcomputer command to the spoken phrase. Once spoken, the speech recognition system generates microcomputer commands that can be used by applications programs of the operating system. Such a system will allow easier access to complex applications programs to both normal and handicapped users alike.

In advanced speech recognition systems, each word will be detected and converted to ASCII text. These sentence or phrase of words will then be handed over to a natural language processing system which will parse the sentences and determine a semantic meaning for each sentence parsed. The semantic meanings will then be translated into commands in an artificial language which is understood by the planning and control systems. Through the decomposition of these artificial language commands, possibly into other lower level commands that are only understood by the robot's controller, the planning systems will guide the attached machinery through what will be considered intelligent actions. It is also likely that low-level modules will return their command languages up to the high levels which in turn will translate the artificial

languages back into natural language responses. These responses might be displayed on a CRT or possibly translated back into speech signals that will be broadcast over a loudspeaker. Hence, natural language processing systems will play an important part in the development of the next level of intelligent robotic systems.

Using natural language to describe tasks has several advantages [9]. First, and probably most important, it provides for an interesting interaction between the human and the machine. Programming a task for a robotic system will not be as boring or as difficult to accomplish as it is currently is with first and second generation robot programming languages. Natural language also tolerates imprecision better than does the strict syntax of a robot programming language. Also, and amount of information about a task that is to be conveyed can often be condensed by using natural language. It is also feasible to claim that describing a task in natural language will be faster than writing the task in a robot control program.

4.1 The RVG System

The Register Vector Grammar natural language processing system is an efficient, compact, finite-state parsing system that is well-suited for real-time situations.

4.1.1 Overview of Register Vector Grammar NL Processing

The key feature that makes the RVG natural language processing system a good choice for real-time applications is that it its syntactical as well as lexical knowledge are represented in a compact data structure, called ternary feature vectors. Ternary feature vectors are fixed length ordered vectors which allow for three values for each feature. The values range from 0 to 2 where 0 means the feature is off or doesn't exist, 1 means the feature is on or does exists, and 2 mean that the system doesn't care what the value of the feature is (a mask). The ordering of features in a ternary vector is significant only in that the processing system knows which features it is dealing with. The actual order of features defined in a ternary vector by the RVG programmer is insignificant to the processing performance of the system. The ternary feature vectors are implemented as a pair of bitvectors (Pascal sets) and are compared by ternary operators in a very fast bit-wise fashion. Thus, whole vectors are processed in parallel on current machine architectures. It is theoretically feasible to build an RVG machine which exploits ternary operations on these feature vectors.

A register vector grammar consists of a table of syntactic production rules which contain two vectors, a condition vector (the vector that must be matched in order to use that rule) and a result vector (the action to be taken once that rule is fired). A RVG lexicon consists of fixed-size lexical entries, which, in addition to information pertaining to morphology and syntactic categories, is also made up of ternary feature vectors for representing semantic constraints. The RVG parser uses the grammar and lexicon to parse input sentences and generate semantic descriptions. It is important to note that semantic forms are built and checked in step with the syntactic parse of the sentence. Thus,

semantics can help constrain on syntactic parsing.

4.1.2 Overview of RVG System Components

Figure 4-1 shows an overview of the RVG natural language processing system, including its various components, one of which is the MIC Planner System. The components of the system include:

- ***The Editor Subsystem*** - Allows the RVG programmer to modify *syntactic* productions and *lexical* entries.

- ***The Parser Subsystem*** - Parses sentences and reports its results in the form of two registers, the Current Syntactic State Register, or CSSR and the Current Predicative State Register, or CPSR.

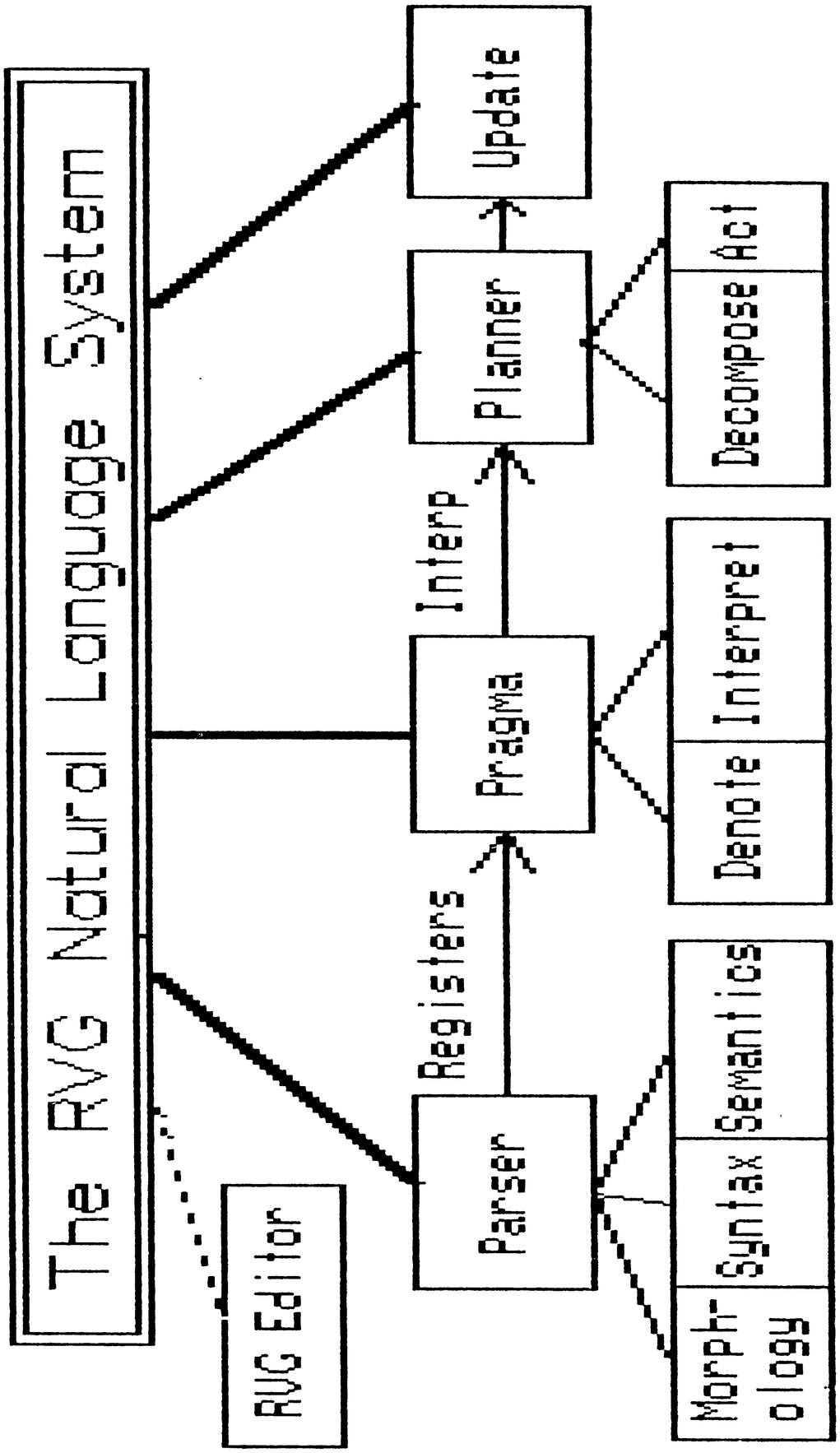
- ***The Pragma Subsystem*** - This module receives the registers from the parser, interprets them, and generates *task* commands for the planner.

- ***The MIC Planner Subsystem*** - This is the planner. It receives Prolog predicates from the Pragma subsystem and generates robot moves. After moves, *Update* is called.

- ***The Update Subsystem*** - This subsystem performs all the necessary world model updates.

The RVG language programmer can initially defines his syntax and lexical dictionary by using the *RVG Editor*. The *Editor* subsystem is called from the main RVG program and acts as a separate module independent from the other RVG subsystems. The *Editor* provides the RVG language programmer the means to add and delete syntax productions and lexical entries. With the *Editor*, the programmer can also add ~~or~~ delete entire features (a column in the ternary vector) globally throughout all vectors in both the syntax and lexicon. Mostly, the *Editor* is used by the RVG grammar designer to modify existing ternary features in the grammar and lexicon. Modifying grammar features will determine the number of productions that are available at a given point in the

Figure 4-1: Major components of the RVG System.



syntactic parse. Modifying semantic features in the lexicon will determine which words are semantically valid at any point in the parse.

Once the changes have been made, the *Editor* prompts the user if he wants to save the changes. The user is not forced to save any changes. Instead, he may choose to exit the *Editor*, return to the main RVG options menu, and choose the *Parser* subsystem to try out the changes that he has made. If the changes are not agreeable, then the user can reselect the *Editor* subsystem and make corrections.

The RVG *Parser* consists of three interacting modules:

1. A lexical component (Morphology and Lookup),
2. A syntactic component (Syntax),
3. And a semantic component (Semantics).

The *Parser* contains two special registers that it updates as it parses an input sentence. These are the Current Syntactic State Register (CSSR) and the Current Predicative State Register (CPSR). The CSSR, a ternary feature vector expressing syntactic state, is generated and updated as the parse proceeds. The CPSR is a set of pointers to semantic entities that are generated from lexical materials, and retried as new information is made available.

The parser starts with the CSSR register initialized to a current starting state. As words are read in, the morphology and lookup modules are called upon to recognize each word. These routines return a category (noun, transitive verb, etc) as well as a ternary vector containing semantic information about the word. The RVG parser then searches for productions that match the category of the word. This is the *match* operation and is demonstrated below:

TERNARY MATCH OPERATION

CSSR = 1222122011
CONDITION VECTOR = 1122122022

Here, a production's *condition* vector matches the CSSR by matching each feature either exactly (by 0 or 1) or by using the mask (the "don't care" condition of a 2) condition. Since this production matches, it is a possible parse state for the word in the sentence. A copy of this CSSR is then made and kept for the next word that is parsed. Before storing, this copy of the CSSR is changed by the matching production's *result* vector as is demonstrated below:

TERNARY CHANGE OPERATION

CSSR COPY = 1222122011
RESULT VECTOR = 0011022022
NEW CSSR COPY = 0011022021

The changed CSSR is now placed in a queue along with other possible syntactic parse path CSSRs. Each one is then selected from the queue and tested via the *match* operation once the next word in the input stream is parsed. Because of this straight forward matching and updating of a vector of features at one time, the entire RVG natural language processing system is fast. Moreover, RVG avoids recursion, as found in common ATN parsers, and hence lots of memory overhead is not required. Embedding is instead modeled by a short array simulating human short-term memory. The system is basically a fast, forward-chaining production system.

As the parse progresses, semantic information is also being built up and stored in the CPSR. Here, the information, all available within lexical entries, is returned by the *Lookup* module is used by predicative semantics to build semantic forms in the CPSR. Since semantic information is available during the parse, it can be called upon to help further constrain processing by eliminating

bogus parses of the sentence. Ternary vectors allow constraints to be propagated down the parse chain without any extra overhead. Since the constraints are initially there, they only need be checked for during the *match* ternary operation. New constraints are added as the CPSR is refined. For a further explanation of RVG syntax and semantic parsing, refer to [6] and [7].

The next major subsystem of the RVG natural language system is the *Pragma* module. This subsystem receives both the CSSR and CPSR from the *Parser* upon the completion of a successful parse of the input sentence. The function of this module is to take the semantic information built up during the parse in the CPSR, made of abstract lexical material, and see if it *refers* to particular objects in the robot's world. If so, and if the sentence is imperative, *Pragma* will generate Prolog task predicates that are passed to the planner for actual execution.

The *Pragma* subsystem is divided into two modules, *Denote* and *Interpret*. The *Denote* module takes CPSR and tries to establish reference with respect to a *database* (the robot's world model). If a particular object cannot be referenced in the database, the module asks the user if he wants to instantiate a new object with the given name obtained during the parse of the original input sentence. Thus, the *denotational semantics*, takes abstract meaning stored in the CPSR and allows it to reference objects in the world.

Once *Denote* has established reference, the *Interpret* module is ready to assembly appropriate task predicates in the form of Prolog predicates. These Prolog task predicates are then passed to the planner subsystem where physical actions actually are planned and carried out. For more information on the *Denote* module and *denotational semantics*, refer to [20].

The *Planner* subsystem, know as the MIC Planner, then reads in the Prolog task predicates and acts on them accordingly. MIC Planner is divided into two modules, *Decompose* and *Act*. The *Decompose* module allows for complete task decomposition from abstract meaning into lower level robot arm motion primitives. As pieces of the task are decomposed, the *Act* module is called. Act is a predicate that generates the robot controller commands for the TeachMover robot. These commands are sent to the CALC program which actually calculates the arm trajectory and interfaces with the robot's controller. After an object is moved in the world, the *Planner* calls the *Update* subsystem.

The *Update* subsystem must update the world model database for the RVG natural language system when the planner sends it a parameter list of object features to be updated.

4.1.3 Future System Goals

As of the writing of this paper, several major subsystems of the RVG natural language system have been completed. Some additional integration of submodules remains to be done. One specific integration problem related to the planner is that of maintaining the world model. Since the *Planner* subsystem has been written in Prolog, the *Pragma* subsystem, in order to pass information to the *Planner*, must write task predicates as Prolog predicates to an intermediate file which is then read by the *Planner*. Thus, two world models must be maintained by the RVG System, one for the *Pragma* subsystem to reference with its *Denote* module, and one for the *Planner* to reference upon decomposing motion tasks. The *Update* subsystem, receiving another file from the *Planner*, will change the RVG database.

Thus we note that the two world models need not contain the same infor-

mation. The RVG database used by *Pragma* contains various features about objects in the world such as object color, size, etc., that can be referenced by the user through natural language queries. The *Planner* world model does not have to contain this information. Instead, the *Planner*'s world model only has to maintain simple labels for each object as well as XYZ positional information. Thus, "Moving the red block to the top of the green block" may require the denotational system to search the RVG database for a specific instance of a block that is red (block1) and another instance for a block that is green (block2). It would then have to return labels used by the planning system's world model and give them to *Interpret*. *Interpret* would then generate the *Planner* predicate *putOn* and combine the object parameters to finally produce the following MIC Planner input:

```
PUTON( BLOCK1,
BLOCK2 ).
```

Chapter 5
Summary

Many of the topics discussed in this paper are currently under investigation by researcher around the world. Proposals have been made for the development of a universal task-oriented robot programming languages that will be transportable from one robot to another. We have seen that Prolog planning systems allows for functional extensibility through the linking together of lower level predicates in a hierarchical fashion. Thus, a programmer can easily describe abstract assembly tasks to a Prolog planning system in a relatively short time. We have also seen that a planning system that incorporates a hierarchy of task predicates is in itself a step toward the next generation of robot manipulator languages. Robot operators in the near future might use such object-oriented task languages to write complete assembly procedures by simply piecing together a few Prolog predicates from a larger library of possible lower level primitive assembly task predicates.

One major problem that must be examined before any generalize task-oriented language can be developed is the definition of robot independence. Because there exists many different robot geometries and configurations, it is difficult for many researchers to agree on what set of robot motion primitives are universally acceptable for use by all robotic systems [16]. A rotation primitive of a robot's base will work fine for cylindrical and spherical coordinate robots, but how will this primitive action be performed by XYZ Cartesian coordinate robots or mobile robots that have no base? Indeed, robot independence and the definition of specific motion primitives is another research area in which the National Bureau of Standard's skills and expertise will be needed.

It is conceivable that in the future, robot operators will communicate verbally with robots and instruct them to perform a variety of tasks. The user interface will combine the technologies of a signal processing speech recognition system and natural language parsing system. The underlying meaning of the spoken sentence will most likely be converted into some form of intermediate high-level object-oriented task language that the robot understands. The high-level task will then be broken down into basic sets of primitives that control specific robot motions and actions. Various complex input sensors will provide the needed feedback to allow the robot to react autonomously in a complex and dynamically changing environment.

This thesis has discussed one approach in developing an easy to use interface between man and robot. It is apparent that in order to make intelligent machines, many aspects of artificial intelligence will have to be combined. This will not only include the areas of planning and natural language processing, but also other AI related subfields such as vision systems, object recognition, speech understanding and speech synthesis. Contributions from other disciplines will also be needed such as enhanced grippers and arm configurations from Mechanical Engineering, better planning methods and the integration of Cad/Cam from Industrial Engineering, new microprocessors such as an RVG machine from Electrical Engineering, and integration of shop floor and office networks (MAP and TOP network protocols) from Computer Science. With contributions from each engineering discipline, the advent of easy to use intelligent machine may become a reality.

The advent of autonomous robots is not science fiction. Today, in many academic and industrial artificial intelligence and robotics laboratories around

the world, researchers are working hard at advancing the state-of-the-art in robot path planning systems that handle conflicts, improved vision systems for scene analysis, complex robot sensors including both tactile and force sensing, and improved man-machine interfaces including graphics and natural language input and natural language generation. Though most of this technology is still in the experimental stages, both industry and the government are providing funding for much of the basic research. The United States government's Defense Advanced Research Projects Agency (DARPA) is very interested in such research projects. DARPA has already proposed four advanced AI/robotic research projects, one of which is the development of an autonomous land vehicle. It is estimated that the overall U.S. government research funding for robotics alone in FY 1982 and 1983 was approximately $20 million per year [12].

The aim of this thesis was to provide an overview and a demonstration of how AI may help robot manufactures develop intelligent robots as well as what might be expected by robot operators in dealing with industrial robots. Through the use of various new computer technologies, robot operators in the future will be able to use natural language, graphics, teach pendants, and high-level task languages to quickly, easily, and safely train their industrial robots to perform a variety of tasks.

References

1. Albus, J.S., A.J. Barbera, and M.L. Fitzgerald. Programming A Hierarchical Robot Control System. 12th International Symposium on Industrial Robots, National Bureau of Standards, Washington, D.C., 20234, June 9-11th, 1982, pp. 505-517. From the 6th International Conference on Industrial Robot Technology, Paris, France.

2. Albus, J.S., C.R. McLean, A.J. Barbera, M.L. Fitzgerald. Hierarchical Control For Robots In An Automated Factory. 13th ISIR/Robots 7 Symposium, National Bureau of Standards, Washington, D.C. 20234, April, 1983, pp. 1-14.

3. Albus, J.S., A.J. Barbera and R.N. Nagel. Theory And Practice Of Hierarchical Control. Twenty Third IEEE Computer Society International Conference Proceedings, National Bureau of Standards, Washington, D.C. 20234, September, 1981, pp. 18-35.

4. *AML Reference Manual.* IBM Corporation, Boca Raton, Florida 33432, 1981. Second Edition.

5. Barbera, A.J., J.S. Albus, and M.L. Fitzgerald. Hierarchical Control Of Robots Using Microcomputers.

6. Blank, Glenn D. A New Kind Of Finite-State Automaton:Register Vector Grammar. IJCAI-85, Dept. of CSEE, Lehigh University, Bethlehem PA 18015, 1985, pp. 749-755. Proceedings of the Ninth International Joint Conference On Artificial Intelligence.

7. Blank, Glenn D. *Lexicalized Metaphores: A Cognitive Model in the Framework of Register Vecotr Semantics.* Ph.D. Th., University of Wisconsin-Madison, 1984.

8. Cohen, P.R., and E.A. Feigenbaum. Planning and Problem Solving. In *The Handbook Of Artificial Intelligence*, William Kaufmann, Inc., Stanford, California, 1982, Chap. 15, pp. 515-562.

9. Evrard, F., H. Farreny, H. Prade. A Pragmatic Interpreter Of A Task-Oriented Subset Of Natural Language For Robotic Purposes. 12th International Symposium on Industrial Robots, Loboratorie Languages Et Systemes Informatiques, Universite' Paul Sabatier, 118, Route de Narbonne - 31062 Toulouse Cedex, France, June 9-11th, 1982, pp. 531-538. From the 6th International Conference on Industrial Robot Technology, Paris, France.

10. Fahlman, Scott, E. "A Planning System for Robot Construction Tasks". *Artificial Intelligence 5* (1974), 1-49. Description of the BUILD planning system.

11. Fikes, R.E. and N.J. Nilsson. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving". *IJCAI-71 1* (1971), 198-208.

12. Gevarter, William B.. *Intelligent Machines: An Introductory Perspective of Artificial Intelligence and Robotics.* Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1985.

13. Hayes-Roth, B. and F. Hayes-Roth. "A Cognitive Model of Planning". *Cognitive Science 3* (1979), 275-310.

14. Haynes, L.S., A.J. Barbera, J.S. Albus, M.L. Fitzgerald, and H.G. McCain. "An Application Example Of The NBS Robot Control System". *Robotics & Computer-Integrated Manufacturing 1*, 1 (1984), 81-95. Industrial Systems Division, National Bureau of Standards.

15. Miller, David, R. James Firby, and Thomas Dean. Deadlines, Travel Time, and Robot Problem Solving. IJCAI-85, Yale University, Dept. of Computer Science, New Haven, Connecticut 06520, 1985, pp. 1052-1054. Proceedings of the Ninth International Joint Conference On Artificial Intelligence.

16. Nagel, R.N. and S.R. Garrigan. An Analysis Of Robot Software And Plans For Its Enhancement. 85-001, Lehigh University, Bethlehem, PA, June, 1985. Prepared for the NATO Advisory Group for Aerospace Research & Development, Lecture Series No. 142, *Artificial Intelligence and Robotics*, September, 1985.

17. Sacerdoti, Earl D. "The Nonlinear Nature of Plans". *IJCAI-75 1* (1975), 206-214.

18. Sobek, Ralph P. A Robot Planning Structure Using Production Rules. IJCAI-85, Laboratoire d'Automatique et d'Analyse des Systems du C.N.R.S., 7, avenue du Colonel-Roche, F-31077 Toulouse Cedex, France, 1985, pp. 1103-1105. Proceedings of the Ninth International Joint Conference On Artificial Intelligence.

19. Stefik, Mark J. *Planning with Constraints.* Ph.D. Th., Stanford University, 1980.

20. Stevens, John C. Reference and Quantification in a Register Vector Grammar Natural Language Processor. Master Th., Lehigh University,1985.

21. Tate, Austin. "Interacting Goals and Their Use". *IJCAI-75 1* (1975), 215-218.

22. *TeachMover User Reference Manual.* Microbot, Inc., Mountain View, California, 1982. Edition 2.

23. *VAL Programming Guide.* Unimation Inc., Danbury, Connecticut, 1980.

24. Werkman, Keith J. *Microbot Instruction Code Compiler: The MIC Compiler System.* Institute For Robotics, Lehigh University, Bethlehem PA 18015, 1985. User's Guide and Programming Manual, Version 2.1.

25. Werkman, Keith J. *Introduction To The Microbot TeachMover Robot.* Institute For Robotics, Lehigh University, Bethlehem PA 18015, 1984. Excerpts Form The TeachMover User Reference Manual, Version 1.1.

26. Winograd, Terry. "Understanding Natural Language". *Cognitive Psychology 3* (1972), 1-191.

Appendix A
Example Run Of Assembly Task

```
[C:\prolog] prolog

                        A.D.A. PROLOG
                type VMA (LARGE MODEL - VIRTUAL MEMORY)
                        Top of memory < 527990
                Workspace Available: 256 Kbytes
                        Version 1.80 - 12/02/85
                        Copy for Keith Werkman
                        Single CPU License
        Copyright Robert Morein and Automata Design Associates 1985
                        Dresher, Pa. (215) - 646-4894

/*****************************************************************
  Consult the prolog based robot planning program.
*****************************************************************/

root/user/?- consult(robot).

Compiling robot.MIC

/*****************************************************************
  Start of main program.
*****************************************************************/

 Planner by Keith Werkman.  Version 1.2, 12/11/85


part1 is located at:[5,5,0], oriented:[-90,0,1],
         features:[1,1,1] .
part2 is located at:[5,-5,0], oriented:[-90,0,1.5],
         features:[1.5,1.5,1.5].
part3 is located at:[8,0,0], oriented:[-90,0,2],
         features:[4,2,1] .
The arm is located at: [5,0,0],  Pitch=-90, Roll=0, Grip=0 .

  The gripper is holding: nothing .


Type  help.  for help.

Yes.
```

```
/*****************************************************************

   THE FOLLOWING IS A LIMITED TRACE OF THE ASSEMBLE TASK
  where two part2 are fastened together into one. In this
  case, there is a collision detected at the assembly point
  and the offending part, PART3, is removed to a collision
  stack. After the collision condition is resolved, the
  assembly task proceeds until completion.

*****************************************************************/

root/user/?- assemble(part1, part2, [8.0,0.0,0.0] ).

Assembling part1 and part2 at [8,0,0].
SUB TASK: FETCH FROM TO.  Fetching part1 and placing at [8,0,0].
==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Released Object: nothing

  Gripping Object: part1

==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==


**** Collision Warning: part3 found at [8,0,0] ****
         Clearing [8,0,0].

---->Storing object in gripper at [2,-5,0]<----

==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Released Object: part1

==Departing arm==
==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Gripping Object: part3

==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Released Object: part3

==Departing arm==
```

```
----->Restoring previous gripper object<----

==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Gripping Object: part1

==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Released Object: part1

==Departing arm==

SUB TASK: FETCH FROM TO.  Fetching part2 and placing at [8,0,0].
==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Gripping Object: part2

==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Released Object: part2

==Departing arm==

SUB TASK: MATE.   Mating part1 to part2 .
SUB TASK: TWIST.   Twisting part2.

  Gripping Object: part2


SUB TASK: FASTEN.  Fastening part2 to part1 .
SUB TASK: MELT.  Melting part2 to part1.

part2 has been removed from the world.
Now part2 is part of part1.

  Released Object: part2

==Departing arm==

Assembly Task: Completed.

Yes.
```

```
/*****************************************************************
  Display the world contents after the assemble operation has
  been completed.
*****************************************************************/

root/user/?- world.
part1 is located at:[8,0,0], oriented:[-90,0,1],
         features:[1.5,1.5,2.5].
part3 is located at:[2,5,0], oriented:[-90,0,2],
         features:[4,2,1].
The arm is located at: [8 , 0 , 5],  Pitch=-90, Roll=90, Grip=3.

  The gripper is holding: nothing .


Yes.

/*****************************************************************

   NOTICE: PART2 is does not exist in the world.  It has been
 fastened to PART1.  Note the new height dimension of  PART1.
 It used  to be  1 inch  high.  Now  it is  2.5 inches  high.
 PART2's height was 1.5 inches.  Thus 1 + 1.5 = 2.5 inches.

*****************************************************************/

root/user/?- exitsys.
Exiting to the operating system.

[C:\prolog]
```

Appendix B
Example Run Of Assembly Task With Full Trace

```
/**************************************************************
  The following is an example run of the ASSEMBLY task
 predicate with the SAVE-MOVES option enabled. Hence, all
 robot arm moves are written out to a file called MOVES.MIC.
 These moves are encoded in commands in the MIC language and
 can be compiled into native code by the MIC Compiler System.

  RESET the system with three parts.
**************************************************************/

root/user/?- initialize.

 Planner by Keith Werkman.  Version 1.2, 12/11/85


part1 is located at:[5,5,Ø], oriented:[-9Ø,Ø,1],
         features:[1,1,1].
part2 is located at:[5,-5,Ø], oriented:[-9Ø,Ø,1.5],
         features:[1.5,1.5,1.5].
part3 is located at:[8,Ø,Ø], oriented:[-9Ø,Ø,2],
         features:[4,2,1].
The arm is located at: [5 , Ø , Ø],  Pitch=-9Ø, Roll=Ø, Grip=Ø .

  The gripper is holding: nothing .


Type  help.  for help.

Yes.

/**************************************************************
  This time, run program with the Save Moves option enabled.
**************************************************************/

root/user/?- saveMoves(yes).
Command output *ENABLED*.
Logging move to MOVES.MIC


Yes.
```

```
/*****************************************************************
  Call the ASSEMBLY predicate again to generate log.
*****************************************************************/

root/user/?- assemble(part1, part2,[8.0,0.0,0.0] ).

Assembling part1 and part2 at [8,0,0].
SUB TASK: FETCH FROM TO.  Fetching part1 and placing at [8,0,0].

==Departing arm==
Moving FROM: [5,0,0]
         TO: [5,0,5].  Gripper holding nothing .
Logging move to MOVES.MIC

==Moving Above Drop Point==
Moving FROM: [5,0,5]
         TO: [5,5,5].  Gripper holding nothing .
Logging move to MOVES.MIC

==Moving arm TO Drop Point==

  Released Object: nothing

Moving FROM: [5,5,5]
         TO: [5,5,0].  Gripper holding nothing .
Logging move to MOVES.MIC

  Gripping Object: part1

==Departing arm==
Moving FROM: [5,5,0]
         TO: [5,5,5].  Gripper holding part1 .
Logging move to MOVES.MIC

==Moving Above Drop Point==
Moving FROM: [5,5,5]
         TO: [8,0,5].  Gripper holding part1 .
Logging move to MOVES.MIC
```

```
==Moving arm TO Drop Point==

**** Collision Warning: part3 found at [8,0,0] ****
          Clearing [8,0,0].

---->Storing object in gripper at [2,-5,0]<----

==Departing arm==
==Moving Above Drop Point==
Moving FROM: [8,0,5]
         TO: [2,-5,5].  Gripper holding part1 .
Logging move to MOVES.MIC

==Moving arm TO Drop Point==
Moving FROM: [2,-5,5]
         TO: [2,-5,0].  Gripper holding part1 .
Logging move to MOVES.MIC

  Released Object: part1

==Departing arm==
Moving FROM: [2,-5,0]
         TO: [2,-5,5].  Gripper holding nothing .
Logging move to MOVES.MIC

==Departing arm==
==Moving Above Drop Point==
Moving FROM: [2,-5,5]
         TO: [8,0,5].  Gripper holding nothing .
Logging move to MOVES.MIC

==Moving arm TO Drop Point==
Moving FROM: [8,0,5]
         TO: [8,0,0].  Gripper holding nothing .
Logging move to MOVES.MIC

  Gripping Object: part3
```

```
==Departing arm==
Moving FROM: [8,0,0]
         TO: [8,0,5].  Gripper holding part3 .
Logging move to MOVES.MIC

==Moving Above Drop Point==
Moving FROM: [8,0,5]
         TO: [2,5,5].  Gripper holding part3 .
Logging move to MOVES.MIC

==Moving arm TO Drop Point==
Moving FROM: [2,5,5]
         TO: [2,5,0].  Gripper holding part3 .
Logging move to MOVES.MIC

  Released Object: part3

==Departing arm==
Moving FROM: [2,5,0]
         TO: [2,5,5].  Gripper holding nothing .
Logging move to MOVES.MIC

---->Restoring previous gripper object<----

==Departing arm==
==Moving Above Drop Point==
Moving FROM: [2,5,5]
         TO: [2,-5,5].  Gripper holding nothing .
Logging move to MOVES.MIC

==Moving arm TO Drop Point==
Moving FROM: [2,-5,5]
         TO: [2,-5,0].  Gripper holding nothing .
Logging move to MOVES.MIC

  Gripping Object: part1

==Departing arm==
Moving FROM: [2,-5,0]
         TO: [2,-5,5].  Gripper holding part1 .
Logging move to MOVES.MIC

==Moving Above Drop Point==
Moving FROM: [2,-5,5]
         TO: [8,0,5].  Gripper holding part1 .
Logging move to MOVES.MIC

==Moving arm TO Drop Point==
Moving FROM: [8,0,5]
         TO: [8,0,0].  Gripper holding part1 .
Logging move to MOVES.MIC
```

```
Moving FROM: [8,0,0]
         TO: [8,0,0].  Gripper holding part1 .
Logging move to MOVES.MIC

  Released Object: part1

==Departing arm==
Moving FROM: [8,0,0]
         TO: [8,0,5].  Gripper holding nothing .
Logging move to MOVES.MIC


SUB TASK: FETCH FROM TO.  Fetching part2 and placing at [8,0,0].
==Departing arm==
==Moving Above Drop Point==
Moving FROM: [8,0,5]
         TO: [5,-5,5].  Gripper holding nothing .
Logging move to MOVES.MIC

==Moving arm TO Drop Point==
Moving FROM: [5,-5,5]
         TO: [5,-5,0].  Gripper holding nothing .
Logging move to MOVES.MIC

  Gripping Object: part2

==Departing arm==
Moving FROM: [5,-5,0]
         TO: [5,-5,5].  Gripper holding part2 .
Logging move to MOVES.MIC

==Moving Above Drop Point==
Moving FROM: [5,-5,5]
         TO: [8,0,5].  Gripper holding part2 .
Logging move to MOVES.MIC

==Moving arm TO Drop Point==
Moving FROM: [8,0,5]
         TO: [8,0,1].  Gripper holding part2 .
Logging move to MOVES.MIC

  Released Object: part2

==Departing arm==
Moving FROM: [8,0,1]
         TO: [8,0,5].  Gripper holding nothing .
Logging move to MOVES.MIC
```

```
SUB TASK: MATE.   Mating part1 to part2 .
SUB TASK: TWIST.  Twisting part2.

Moving FROM: [8,0,5]
         TO: [8,0,1].  Gripper holding nothing .
Logging move to MOVES.MIC

  Gripping Object: part2

Gripper rotated: 90 degrees.
   Object in gripper (part2), also rotated.
Logging move to MOVES.MIC

SUB TASK: FASTEN.  Fastening part2 to part1 .
SUB TASK: MELT.    Melting part2 to part1.

part2 has been removed from the world.
Now part2 is part of part1.

  Released Object: part2

==Departing arm==
Moving FROM: [8,0,1]
         TO: [8,0,5].  Gripper holding nothing .
Logging move to MOVES.MIC

Yes.

/*************************************************************
  Display the world contents after the assemble operation has
  been completed.
*************************************************************/

root/user/?- world.

part1 is located at:[8,0,0], oriented:[-90,0,1],
         features:[1.5,1.5,2.5].
part3 is located at:[2,5,0], oriented:[-90,0,2],
         features:[4,2,1].
The arm is located at: [8 , 0 , 5],  Pitch=-90, Roll=90, Grip=3 .

  The gripper is holding: nothing .

Yes.
```

Appendix C
MIC Code Generated By MIC Planner

```
(*************************************************************
   The following file was generated by the MIC Planner System
   for the task:
         ASSEMBLE( Part1, Part2, [8.0, 0.0, 0.0] )

   This code can be compiled into native commands for
   the TeachMover instructional robot by Microbot, Inc.
   using the MIC Compiler System.
*************************************************************)

(* =START= File generated by MIC Planner:
   File opened on: Wed Dec 11 15:39:11 1985

*)
(* File Re-Opened For Output *)

move (    5.00,    0.00,    0.00, -90.00,    0.00,    0.00 )
(* MAIN TASK: Assemble part1 TO part2 *)
(* SUB TASK: Fetch part1 TO Location *)
(*    PREDICATE: Fetch part1 *)
(*    PREDICATE: Approach Location *)
(*    PREDICATE: Depart *)
move (    5.00,    0.00,    5.00, -90.00,    0.00,    0.00 )
move (    5.00,    5.00,    5.00, -90.00,    0.00,    0.00 )
open
move (    5.00,    5.00,    0.00, -90.00,    0.00,    3.00 )
grip (    1.00 )
(*    PREDICATE: Place At Location *)
(*    PREDICATE: Approach Location *)
(*    PREDICATE: Depart *)
move (    5.00,    5.00,    5.00, -90.00,    0.00,    1.00 )
move (    8.00,    0.00,    5.00, -90.00,    0.00,    1.00 )
```

```
(** CORRECTION PREDICATE: Clear Space **)
(*    PREDICATE: Fetch part3 *)
(** CORRECTION PREDICATE: Clear Gripper Of Object part3 **)
(*    PREDICATE: Approach Location *)
(*    PREDICATE: Depart *)
move (    2.00,   -5.00,    5.00, -90.00,    0.00,    1.00 )
move (    2.00,   -5.00,    0.00, -90.00,    0.00,    1.00 )
open
(*    PREDICATE: Depart *)
move (    2.00,   -5.00,    5.00, -90.00,    0.00,    3.00 )
(*    PREDICATE: Approach Location *)
(*    PREDICATE: Depart *)
move (    8.00,    0.00,    5.00, -90.00,    0.00,    3.00 )
move (    8.00,    0.00,    0.00, -90.00,    0.00,    3.00 )
grip (    2.00 )
(*    PREDICATE: Place At Location *)
(*    PREDICATE: Approach Location *)
(*    PREDICATE: Depart *)
move (    8.00,    0.00,    5.00, -90.00,    0.00,    2.00 )
move (    2.00,    5.00,    5.00, -90.00,    0.00,    2.00 )
move (    2.00,    5.00,    0.00, -90.00,    0.00,    2.00 )
open

(*    PREDICATE: Depart *)
move (    2.00,    5.00,    5.00, -90.00,    0.00,    3.00 )
(** CORRECTION PREDICATE: Restore Gripper Object **)
(*    PREDICATE: Approach Location *)
(*    PREDICATE: Depart *)
move (    2.00,   -5.00,    5.00, -90.00,    0.00,    3.00 )
move (    2.00,   -5.00,    0.00, -90.00,    0.00,    3.00 )
grip (    1.00 )
(*    PREDICATE: Approach Location *)
(*    PREDICATE: Depart *)
move (    2.00,   -5.00,    5.00, -90.00,    0.00,    1.00 )
move (    8.00,    0.00,    5.00, -90.00,    0.00,    1.00 )
move (    8.00,    0.00,    0.00, -90.00,    0.00,    1.00 )
move (    8.00,    0.00,    0.00, -90.00,    0.00,    1.00 )
open
(*    PREDICATE: Depart *)
move (    8.00,    0.00,    5.00, -90.00,    0.00,    3.00 )

(* SUB TASK: Fetch part2 TO Location *)
(*    PREDICATE: Fetch part2 *)
(*    PREDICATE: Approach Location *)
(*    PREDICATE: Depart *)
move (    5.00,   -5.00,    5.00, -90.00,    0.00,    3.00 )
move (    5.00,   -5.00,    0.00, -90.00,    0.00,    3.00 )
grip (    1.50 )
(*    PREDICATE: Place At Location *)
(*    PREDICATE: Approach Location *)
(*    PREDICATE: Depart *)
move (    5.00,   -5.00,    5.00, -90.00,    0.00,    1.50 )
move (    8.00,    0.00,    5.00, -90.00,    0.00,    1.50 )
move (    8.00,    0.00,    1.00, -90.00,    0.00,    1.50 )
open
(*    PREDICATE: Depart *)
move (    8.00,    0.00,    5.00, -90.00,    0.00,    3.00 )
```

```
(* SUB TASK: Mate part1 TO part2 *)
(* SUB TASK: Twist part2 *)
move (    8.00,    0.00,    1.00,  -90.00,    0.00,    3.00 )
grip (    1.50 )
(*    PREDICATE: Rotate Gripper  *)
move (    8.00,    0.00,    1.00,  -90.00,   90.00,    1.50 )

(* SUB TASK: Fasten part2 TO part1 *)
(* SUB TASK: Melt part2 TO part1 (make one) *)
open
(*    PREDICATE: Depart *)
move (    8.00,    0.00,    5.00,  -90.00,   90.00,    3.00 )
```

Appendix D
Listing File Generated By
The MIC Compiler System

```
Line Addr Source
   1    0  (*************************************************************************
   2    0     THE FOLLOWING FILE WAS GENERATED BY THE MIC PLANNER SYSTEM
   3    0     FOR THE TASK:
   4    0           ASSEMBLE( PART1, PART2, [8.0, 0.0, 0.0] )
   5    0
   6    0     THIS CODE CAN BE COMPILED INTO NATIVE COMMANDS FOR
   7    0     THE TEACHMOVER INSTRUCTIONAL ROBOT BY MICROBOT, INC.
   8    0     USING THE MIC COMPILER SYSTEM.
   9    0  *************************************************************************)
  10    0
  11    0  (* =START= FILE GENERATED BY MIC PLANNER:
  12    0     FILE OPENED ON: WED DEC 11 15:39:11 1985
  13    0
  14    0  *)
  15    0  (* FILE RE-OPENED FOR OUTPUT *)
  16    0
  17    0  MOVE (    5.00,    0.00,    0.00,  -90.00,    0.00,    0.00 )
  18    1  (* MAIN TASK: ASSEMBLE PART1 TO PART2 *)
  19    1  (* SUB TASK: FETCH PART1 TO LOCATION *)
  20    1  (*     PREDICATE: FETCH PART1 *)
  21    1  (*     PREDICATE: APPROACH LOCATION *)
  22    1  (*     PREDICATE: DEPART *)
  23    1  MOVE (    5.00,    0.00,    5.00,  -90.00,    0.00,    0.00 )
  24    2  MOVE (    5.00,    5.00,    5.00,  -90.00,    0.00,    0.00 )
  25    3  OPEN
  26    4  MOVE (    5.00,    5.00,    0.00,  -90.00,    0.00,    3.00 )
  27    5  GRIP (    1.00 )
  28    6  (*     PREDICATE: PLACE AT LOCATION *)
  29    6  (*     PREDICATE: APPROACH LOCATION *)
  30    6  (*     PREDICATE: DEPART *)
  31    6  MOVE (    5.00,    5.00,    5.00,  -90.00,    0.00,    1.00 )
  32    7  MOVE (    8.00,    0.00,    5.00,  -90.00,    0.00,    1.00 )
  33    8  (** CORRECTION PREDICATE: CLEAR SPACE **)
  34    8  (*     PREDICATE: FETCH PART3 *)
  35    8  (** CORRECTION PREDICATE: CLEAR GRIPPER OF OBJECT PART3 **)
  36    8  (*     PREDICATE: APPROACH LOCATION *)
  37    8  (*     PREDICATE: DEPART *)
  38    8  MOVE (    2.00,   -5.00,    5.00,  -90.00,    0.00,    1.00 )
  39    9  MOVE (    2.00,   -5.00,    0.00,  -90.00,    0.00,    1.00 )
  40   10  OPEN
  41   11  (*     PREDICATE: DEPART *)
  42   11  MOVE (    2.00,   -5.00,    5.00,  -90.00,    0.00,    3.00 )
  43   12  (*     PREDICATE: APPROACH LOCATION *)
  44   12  (*     PREDICATE: DEPART *)
  45   12  MOVE (    8.00,    0.00,    5.00,  -90.00,    0.00,    3.00 )
  46   13  MOVE (    8.00,    0.00,    0.00,  -90.00,    0.00,    3.00 )
  47   14  GRIP (    2.00 )
```

```
48   15  (*    PREDICATE: PLACE AT LOCATION *)
49   15  (*    PREDICATE: APPROACH LOCATION *)
50   15  (*    PREDICATE: DEPART *)
51   15  MOVE (    8.00,    0.00,    5.00,  -90.00,    0.00,    2.00 )
52   16  MOVE (    2.00,    5.00,    5.00,  -90.00,    0.00,    2.00 )
53   17  MOVE (    2.00,    5.00,    0.00,  -90.00,    0.00,    2.00 )
54   18  OPEN
55   19  (*    PREDICATE: DEPART *)
56   19  MOVE (    2.00,    5.00,    5.00,  -90.00,    0.00,    3.00 )
57   20  (** CORRECTION PREDICATE: RESTORE GRIPPER OBJECT **)
58   20  (*    PREDICATE: APPROACH LOCATION *)
59   20  (*    PREDICATE: DEPART *)
60   20  MOVE (    2.00,   -5.00,    5.00,  -90.00,    0.00,    3.00 )
61   21  MOVE (    2.00,   -5.00,    0.00,  -90.00,    0.00,    3.00 )
62   22  GRIP (    1.00 )
63   23  (*    PREDICATE: APPROACH LOCATION *)
64   23  (*    PREDICATE: DEPART *)
65   23  MOVE (    2.00,   -5.00,    5.00,  -90.00,    0.00,    1.00 )
66   24  MOVE (    8.00,    0.00,    5.00,  -90.00,    0.00,    1.00 )
67   25  MOVE (    8.00,    0.00,    0.00,  -90.00,    0.00,    1.00 )
68   26  MOVE (    8.00,    0.00,    0.00,  -90.00,    0.00,    1.00 )
69   27  OPEN
70   28  (*    PREDICATE: DEPART *)
71   28  MOVE (    8.00,    0.00,    5.00,  -90.00,    0.00,    3.00 )
72   29  (* SUB TASK: FETCH PART2 TO LOCATION *)
73   29  (*    PREDICATE: FETCH PART2 *)
74   29  (*    PREDICATE: APPROACH LOCATION *)
75   29  (*    PREDICATE: DEPART *)
76   29  MOVE (    5.00,   -5.00,    5.00,  -90.00,    0.00,    3.00 )
77   30  MOVE (    5.00,   -5.00,    0.00,  -90.00,    0.00,    3.00 )
78   31  GRIP (    1.50 )
79   32  (*    PREDICATE: PLACE AT LOCATION *)
80   32  (*    PREDICATE: APPROACH LOCATION *)
81   32  (*    PREDICATE: DEPART *)
82   32  MOVE (    5.00,   -5.00,    5.00,  -90.00,    0.00,    1.50 )
83   33  MOVE (    8.00,    0.00,    5.00,  -90.00,    0.00,    1.50 )
84   34  MOVE (    8.00,    0.00,    1.00,  -90.00,    0.00,    1.50 )
85   35  OPEN
86   36  (*    PREDICATE: DEPART *)
87   36  MOVE (    8.00,    0.00,    5.00,  -90.00,    0.00,    3.00 )
88   37  (* SUB TASK: MATE PART1 TO PART2 *)
89   37  (* SUB TASK: TWIST PART2 *)
90   37  MOVE (    8.00,    0.00,    1.00,  -90.00,    0.00,    3.00 )
91   38  GRIP (    1.50 )
92   39  (*    PREDICATE: ROTATE GRIPPER  *)
93   39  MOVE (    8.00,    0.00,    1.00,  -90.00,   90.00,    1.50 )
94   40  (* SUB TASK: FASTEN PART2 TO PART1 *)
95   40  (* SUB TASK: MELT PART2 TO PART1 (MAKE ONE) *)
96   40  OPEN
97   41  (*    PREDICATE: DEPART *)
98   41  MOVE (    8.00,    0.00,    5.00,  -90.00,   90.00,    3.00 )
```

Appendix E
TeachMover Opcodes Generated By MIC Compiler System

```
0,8705,256,0,0,0,0,0
1,8705,-18176,244,-3072,-1280,253,-768
2,8705,-18060,-16204,-19392,-1021,253,-513
3,8705,-18060,-16204,3392,-1021,253,767
4,8705,-2956,-16328,-28352,-253,255,1023
5,8705,-2956,-16328,-21696,-253,255,255
6,8705,-18060,-16204,10048,-1021,253,-1
7,8705,9728,141,0,-768,253,-256
8,8705,-5178,-8725,24099,-1030,-259,-255
9,8705,-2618,-8744,19235,-6,-257,257
10,8705,-2618,-8744,12579,-6,-257,1025
11,8705,-5178,-8725,17443,-1030,-259,513
12,8705,9728,141,-6656,-768,253,256
13,8705,3840,230,16128,0,254,768
14,8705,3840,230,-13312,0,254,256
15,8705,9728,141,29440,-768,253,0
16,8705,-5317,9195,-11811,-1275,509,254
17,8705,-2757,9176,-16675,-251,511,766
18,8705,-2757,9176,12765,-251,511,1278
19,8705,-5317,9195,17629,-1275,509,766
20,8705,-5178,-8725,17443,-1030,-259,513
21,8705,-2618,-8744,12579,-6,-257,1025
22,8705,-2618,-8744,19235,-6,-257,257
23,8705,-5178,-8725,24099,-1030,-259,-255
24,8705,9728,141,0,-768,253,-256
25,8705,3840,230,22784,0,254,0
26,8705,3840,230,22784,0,254,0
27,8705,3840,230,16128,0,254,768
28,8705,9728,141,-6656,-768,253,256
29,8705,-18035,16564,3520,-772,-3,512
30,8705,-2931,16440,-28224,-4,-1,768
31,8705,-2931,16440,25792,-4,-1,256
32,8705,-18035,16564,-8000,-772,-3,-256
33,8705,9728,141,-18176,-768,253,-256
34,8705,28672,183,-7424,-256,254,0
35,8705,28672,183,4096,-256,254,768
36,8705,9728,141,-6656,-768,253,256
37,8705,28672,183,4096,-256,254,768
38,8705,28672,183,-7424,-256,254,0
39,8705,28672,-32585,-7296,-256,510,254
40,8705,28672,-32585,4224,-256,510,1022
41,8705,9728,-32627,-6528,-768,509,510
42,0,0,0,0,0,0,0
43,0,0,0,0,0,0,0
    .   .   .
124,0,0,0,0,0,0,0
125,0,0,0,0,0,0,0
```

Appendix F
Example Stacking Operation

```
/*****************************************************************
  START with the current world so that the three parts are
 stacked one on top of another as in the LEFT stack and issue
 a stack command to get the parts to be ordered as in the
 RIGHT stack.

                    START          FINISH
                    ORDER          ORDER
                    ------         ------
                    Part 2         Part 1
                    Part 3  ===>   Part 2
                    Part 1         Part 3
*****************************************************************/

root/user/?- world.
part1 is located at:[3,3,0], oriented:[-90,0,1],
         features:[1,1,1] .
part2 is located at:[3,3,2], oriented:[-90,0,1.5],
         features:[1.5,1.5,1.5].
part3 is located at:[3,3,1], oriented:[-90,0,2],
         features:[4,2,1] .
The arm is located at: [3 , 3 , 5],  Pitch=-90, Roll=0, Grip=3 .

  The gripper is holding: nothing .

Yes.

/*****************************************************************
  Execute the STACK operation predicate.
*****************************************************************/

root/user/?- stack( part1, part2, part3, [3.0,3.0,0.0] ).

STACKING: part1 ON part2 ON part3 at [3,3,0].
SUB TASK: FETCH FROM TO.  Fetching part3 and placing at [3,3,0].
    part3 has part2 on top of it.
    Want to clear the top of part3 (y or n)? y
```

```
SUB TASK: FETCH FROM TO.  Fetching part2 and placing at [2,5,Ø].
==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Gripping Object: part2

==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Released Object: part2

==Departing arm==
==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Gripping Object: part3

==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==


**** Collision Warning: part1 found at [3,3,Ø] ****
          Clearing [3,3,Ø].

---->Storing object in gripper at [2,-5,Ø]<----

==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Released Object: part3

==Departing arm==
==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Gripping Object: part1

==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Released Object: part1
==Departing arm==
```

```
----->Restoring previous gripper object<----

==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Gripping Object: part3

==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Released Object: part3

==Departing arm==


SUB TASK: PUT ON.  Put part2 on part3.
SUB TASK: FETCH FROM TO.  Fetching part2 and placing at [3,3,0].
    part2 has part1 on top of it.
    Want to clear the top of part2 (y or n)? y

SUB TASK: FETCH FROM TO.  Fetching part1 and placing at [3.75,4.25,0].
==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Gripping Object: part1

==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Released Object: part1

==Departing arm==
==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Gripping Object: part2

==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Released Object: part2

==Departing arm==
```

```
SUB TASK: PUT ON.  Put part1 on part2.
SUB TASK: FETCH FROM TO.  Fetching part1 and placing at [3,3,1].
==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Gripping Object: part1

==Departing arm==
==Moving Above Drop Point==
==Moving arm TO Drop Point==

  Released Object: part1

==Departing arm==

Stack Operation: Completed.


Yes.

/*****************************************************************
  Display the world contents after the stacking operation has
  been completed.
*****************************************************************/

root/user/?- world.
part1 is located at:[3,3,2.5], oriented:[-90,0,1],
         features:[1,1,1] .
part2 is located at:[3,3,1], oriented:[-90,0,1.5],
         features:[1.5,1.5,1.5].
part3 is located at:[3,3,0], oriented:[-90,0,2],
         features:[4,2,1] .
The arm is located at: [3 , 3 , 5],  Pitch=-90, Roll=0, Grip=3 .

  The gripper is holding: nothing .


Yes.
root/user/?- exitsys.
Exiting to the operating system.

[C:\prolog]
```

VITA

Keith James Werkman, the son of Frank James and Margaret Werkman was born on September 13, 1961 in Bethlehem, Pennsylvania. He attended Lehigh University and received his Bachelor of Science Degree in Chemistry with a minor in Computer Science in June of 1983. While pursuing his Master's Degree in Computer Science, he has served as a teaching assistant and laboratory manager for the Institute For Robotics at Lehigh University and has instructed students and engineers from various industries on robotics. His professional experience includes employment with the several small computer companies. He is currently a researcher for the Institute For Robotics in the areas of artificial intelligence and robot languages.