NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

ISBN 0-315-56395-8

PROPOSAL FOR A TASK-LEVEL PROGRAMMED ASSEMBLY SYSTEM

by

Charles A. Gauthier B.A.Sc.

A thesis submitted to the
School of Graduate Studies
in partial fulfillment of the requirements
for the degree of
Master of Applied Science
in Electrical Engineering

Ottawa-Carleton Institute for Electrical Engineering

Department of Electrical Engineering
Faculty of Engineering
University of Ottawa
January 1989

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

The thesis analyzes the problems of robotic assembly and shows that hybrid force/position control is required to carry out assembly tasks successfully. Since hybrid force/position control strategies are difficult to devise and program, the use of task-level programming is proposed. Task-level programming systems offer a very high-level language to specify assembly tasks and automatically devise force/position control strategies. The thesis defines a task-level language and proposes an architecture to support it. This architecture also supports automatic error recovery which was neglected in all previous proposals. The proposed system provides a test-bed in which new planning algorithms and object representation schemes can be evaluated. In doing this, the thesis also surveys the current state of the art in object representation and planning for assembly, and identifies major areas of further research. To the best knowledge of the author, this thesis also represents the first attempt in recent years to integrate the various aspects of task-level programming for robotic assembly. Finally, the thesis serves as a study of the feasibility of implementing complete task-level programmed assembly systems. The integration of the robotic assembly system with the remainder of the plant is not discussed.

Acknowledgements

I am deeply grateful to my thesis supervisor and friend Dr. Moshe Krieger for encouraging me in this undertaking and for giving me the freedom to work on my own. I must also thank him and the Department of Electrical Engineering of the University of Ottawa for giving me the opportunity to teach; I found it an invaluable experience, even though it delayed completion of the thesis.

Je remercie aussi mes parents pour leur encouragement et leur support moral et financier. Ces dernières années ont probablement été plus difficiles pour eux que pour moi.

I also thank my colleagues, Robert Joannis, Jean-Pierre Lachance, and Vojislav Radonjic for their support and ideas.

Finally, I have to thank the Department of Electrical Engineering of the University of Ottawa, the ARTT project, the Fonds pour la Formation de Chercheurs et l'Aide à la Recherche (FCAR), and the National Science and Engineering Research Council (NSERC) of Canada for their financial support.

Contents

List of Figures

Chapter 1

Introduction

1.1 Task-Level Programming for Assembly

Since George Devol patented his Programmed Article Transfer Device, the world first true robot, in 1961, the world's manufacturing robot population has increased steadily every year. In many cases, the use of robots in manufacturing can reduce costs and increase productivity and product quality.

Early robots mostly tended machine tools, moving parts or effecting tool changes. While still heavily employed for such activities, an increase in positioning accuracy has allowed them to perform a variety of more complex tasks such as welding, painting, drilling, cutting, etc. Such activities are relatively simple to program using contemporary robot programming languages. Unfortunately, as argued in chapter 2, a general robotic assembly requires the use of active compliance which is not easily implemented using current robot control techniques. Furthermore, most robot programming languages do not offer a programming paradigm which properly supports robotic assembly since they are too low-level. So far, when designing a product, it has been necessary to design its assembly process simultaneously. Any change to the product implies potential redesign of the the work cell, the grippers, or the sensors, and implies the reprogramming of the robots, often from square one. This is the equivalent in computing of designing a computer to solve each problem and programming it at the assembly language level. The inherent high cost of such a design process means that the benefits of robotic assembly are only realized in

large scale manufacturing. To bring robotic assembly to medium and small size manufacturing, where the high cost of tailoring a product to its assembly process cannot be tolerated, requires general-purpose work cells and high-level programming languages. Because of the nature of assembly, true cost-effective general-purpose work cells are not realistic, thus increasing the need for even higher-level languages.

The use of task-level languages permits a very high-level description of assembly tasks, isolating programmers from the details of the robot and the work cell. In a task-level programmed assembly system, a planner uses the description of the physical and geometric characteristics of parts and the robot to convert the task-level program into an executable robot program. This approach promises to reduce drastically robot programming time and its associated costs.

Previous attempts at designing such systems failed because of the lack of understanding of the problem and its complexity. However, much research has been done in the last few years on several aspects of the problem. Still, as of the date of writing, no complete task-level system has been demonstrated.

This thesis analyzes the problem of robotic assembly, proposes an architecture for a task-level programmed assembly system, and outlines its implementation. This system will provide a test-bed in which new planning algorithms and object representation schemes can be evaluated. In doing this, the thesis also surveys the current state of the art in object representation and planning for assembly, and identifies areas requiring further research.

The thesis does not discuss the integration of the robotic assembly system with the rest of the manufacturing plant. The integration of the robot into a flexible manufacturing environment cannot be investigated until the problems of robotic assembly have been solved satisfactorily.

1.2 Outline of Thesis

Chapter 2 begins with an overview of modern robotics for the unfamiliar reader. The various robot geometries and power sources are first covered, followed by a cursory look at grippers and sensors. The basic concepts of Flexible Manufacturing are then discussed. The chapter continues with robotic assembly and the problems of using pure position control. It is then argued that hybrid force/position control

is required. Task-level programming is then introduced and shown to be an effective way of programming assembly tasks utilizing hybrid force/position control. Finally, previous work in task-level programming and related fields is outlined.

Chapter 3 proposes an architecture for a task-level programmed assembly system. It begins with a definition of a task-level language. Then the minimum requirements of a task-level programmed system are discussed, and the various assumptions and design decisions made are explained. Following this, the architecture of the system is shown, and its operation outlined. The chapter closes by arguing the use of LISP as an implementation language.

Chapter 4 details the requirements of the World Model. It argues that a hybrid Constructive Solid Geometry/Boundary-representation scheme is the ideal scheme for representing solid objects in the system. The use of Variational Graphs is then proposed to represent tolerance information.

Chapter 5 discusses the implementation of the World Model. A small example is given to prove the feasibility of using a CSG/B-rep hybrid scheme.

Chapter 6 discusses the architecture and operation of the Planner. Existing algorithms for gross motions are reviewed, along with the existing work on fine motion planning and grasp planning. The implementation of the Parser is then outlined, followed with the implementation of the Main Planner. Finally, a typical planning algorithm is discussed in detail.

Chapter 7 summarizes the work done in the thesis, and proposes several avenues of further research.

1.3 Research Contributions

This thesis is, to the best knowledge of the author, the first attempt in recent years to integrate the various aspects of task-level programming and robotic assembly. It also discusses the feasibility of implementing a task-level programmed robotic assembly system given the current state of the field, and proposes a task-level language and an underlying architecture to support it. The thesis also proposes an automatic error recovery mechanism, something ignored in all previous proposals. In doing this, the thesis surveys the current state of task-level programming for robotic assembly, extracting the more relevant research contributions.

What the thesis concludes is that complete task-level programmed assembly systems are not likely to be possible for a number of years yet.

Chapter 2

Manufacturing Technology

This chapter begins with a brief review of modern robotics. This review covers robot design, gripper design, and robot programming techniques. The chapter then looks at the benefits of using robots in manufacturing, and briefly discusses the concepts of Flexible and Computer Integrated Manufacturing. Following this, the problems of using robots in assembly are detailed, and active compliance suggested as a general approach to solve the problem. Task-level programming is then introduced as a very high-level language that eliminates the need for programmers to write low-level compliant programs. Finally, previous work in task-level programming is briefly reviewed. The reader already familiar with robotic technology may skip to section 2.4 on page 23.

2.1 Robots

This section reviews several aspects of robot technology, from the mechanical configurations of robots to existing control and programming techniques. A robot is defined by the Robotics Institute of America (RIA) to be any "reprogrammable, multifunctional manipulator designed to move materials, parts, tools or specialized devices through variable programmed motions for the performance of a variety of tasks" [17]. This definition excludes telemanipulators (remote controlled manipulators), exoskeletons (devices which attach on to a human operator and gives him or her more power) and prostheses or artificial limbs.

Robot systems have three essential components:

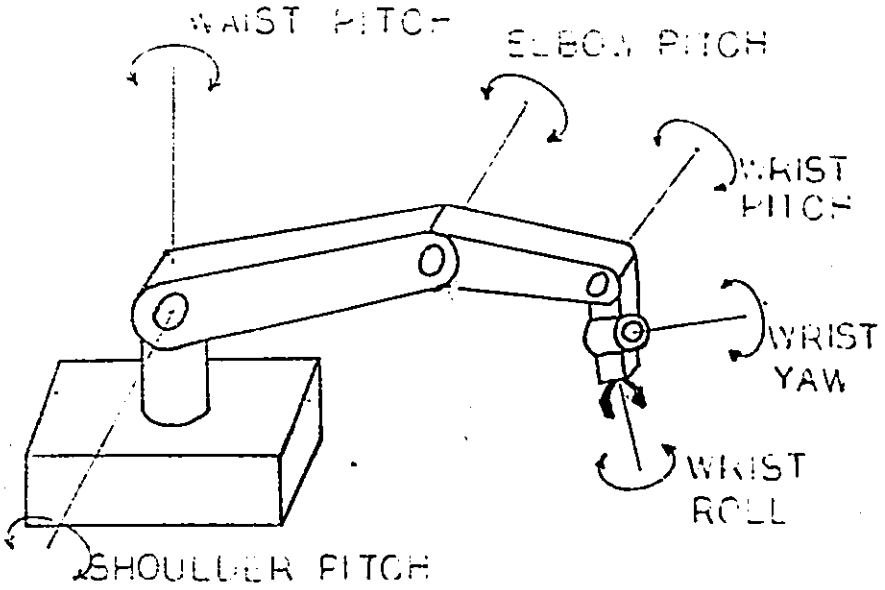

Figure 2.1: Typical robot arms

1. **An arm**—The arm is often made up of two or three rigid links interconnected by three rotating or sliding joints—a waist or base joint, a shoulder joint and an elbow joint—and terminated by a wrist joint. The wrist is a complex joint; it supports the gripper and is capable of motions in three planes. Figure 2.1 illustrates a typical robot arm, the names of the various joints and their possible motions. It is traditional to exclude the motion of the wrist and its gripper from the motion of the arm.

2. **A gripper**—The arm supports the gripper whose function is to hold parts without damaging them. The arm/gripper combination constitutes the *manipulator*.

3. **A controller**—The controller commands and coordinates the motions of the manipulator.

Robot system can also include *sensors*. The following sections look at the details of manipulators, grippers, sensors, and programming methods for robots.

2.1.1 Arms

A robot arm is characterized primarily by its geometry, its most apparent aspect. A robot's geometry determines its configuration as it moves through space and is consequently a crucial element in planning collision free trajectories. Less obvious characteristics of an arm are its power source which determine to a large extent its load carrying capacity, and its accuracy and repeatability.

Geometries

Robots come in various geometries, as illustrated in figure 2.2. The particular geometry of a robot refers to the geometry of the robot's arm only; it does not include the motion of the robot's wrist. The set of space points that can be reached by the robot's gripper mounting constitutes the *work envelope*. Since grippers can be changed and are often designed by the robot user for a specific application rather than purchased from the robot manufacturer, the space a gripper can reach is usually not considered part of the manipulator's work envelope.

- **Polar geometry**—The earliest configuration is the polar geometry. It was used for the world's first industrial robot which was manufactured by Unimation. That corporation used the polar configuration exclusively for years, but now also employs the revolute configuration in their PUMA machines. A polar or spherical configuration robot consists of an arm that moves in and out, rotates in a horizontal plane around the machine's base, and can rotate, to a limited extent, in a vertical plane around a pivot in the machine's base, as illustrated in figure 2.3. One significant drawback of polar robots is that the extension arm cannot be easily sealed from the environment, except where the in and out motion of the arm is restricted. Sealing is accomplished by placing bellows over the extension arm.

- **Cylindrical geometry**—These robots have an extension arm mounted on a vertical axis. The arm therefore sweeps out a volume of space defined as a partial cylinder, as shown in figure 2.4. Cylindrical robots have an additional extension mechanism over the polar robots to seal from the environment.

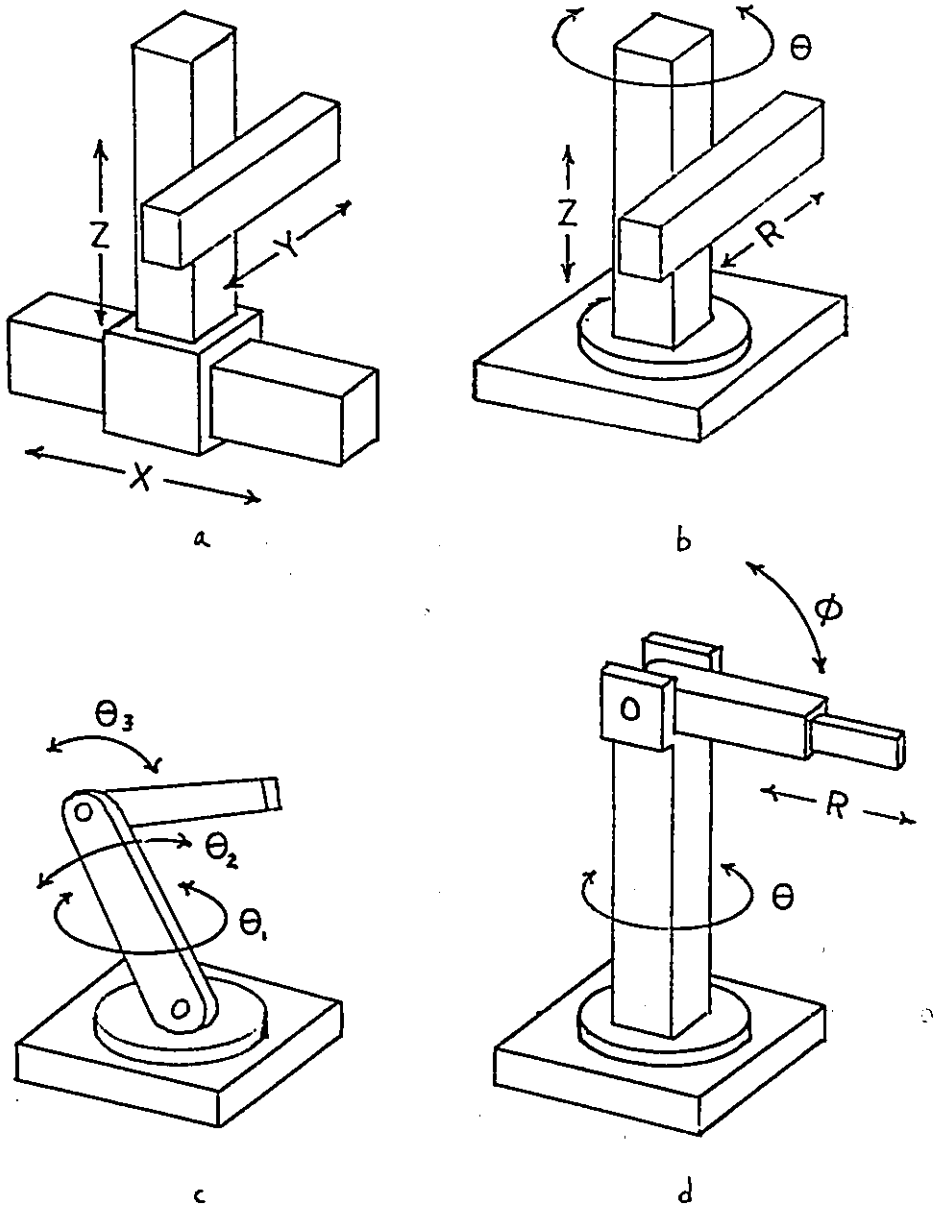

Figure 2.2: Robot arm geometries
(a) Cartesian (b) Cylindrical (c) Revolute (d) Polar

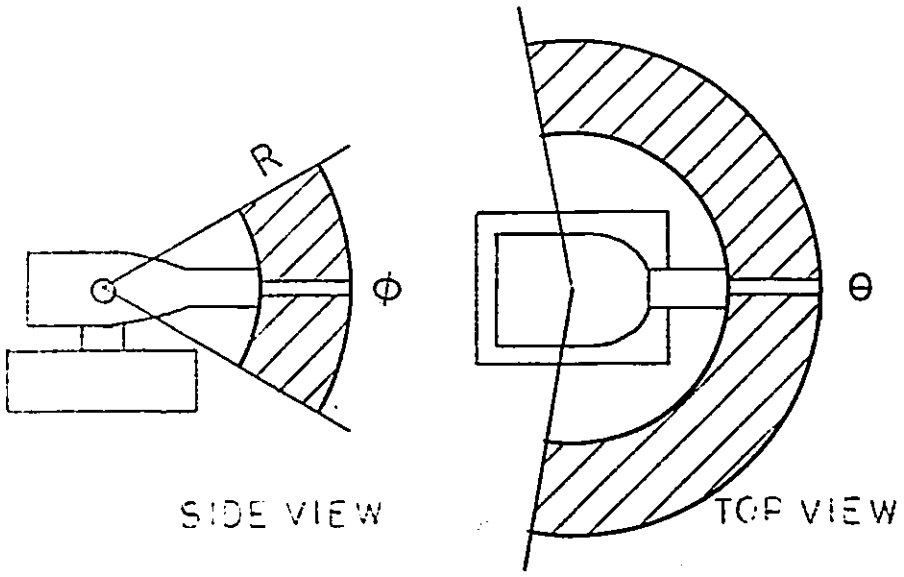

Figure 2.3: Polar robot work envelope

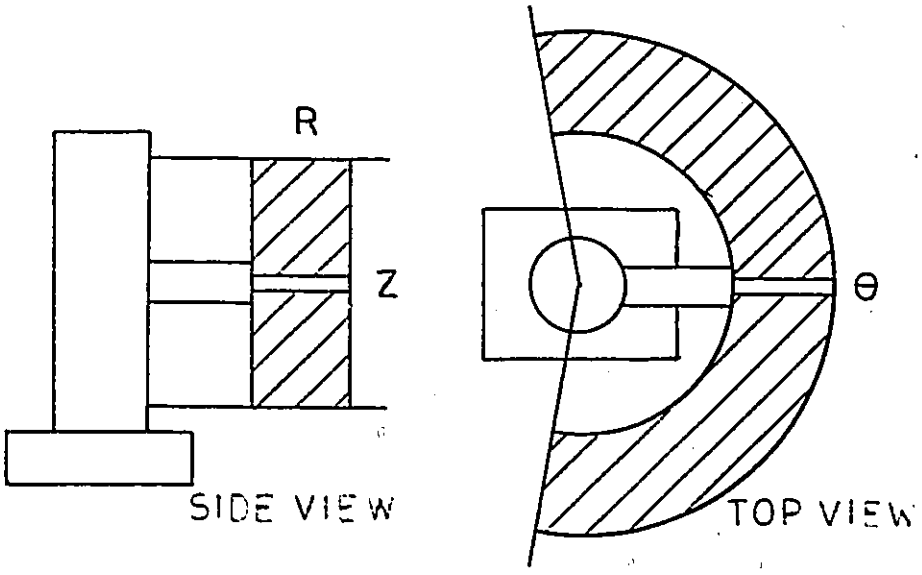

Figure 2.4: Cylindrical robot work envelope

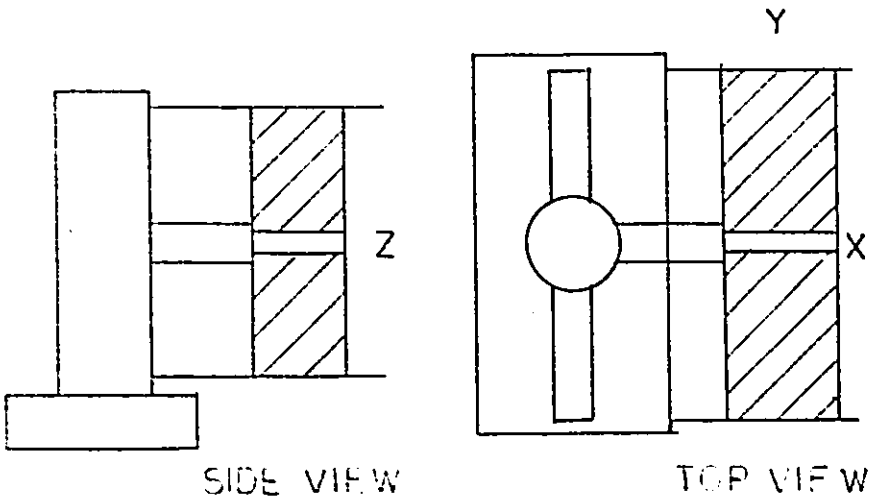

Figure 2.5: Cartesian robot work envelope

- **Cartesian geometry**—The simplest robot configuration is the Cartesian or rectangular one. Here, an extension arm is mounted on a vertical axis which can move along a track perpendicular to the in and out motion of the arm. The robots are extremely easy to program, since most people can easily visualize three-dimensional motions. However, they are more difficult to seal against the environment than the polar or cylindrical robots since three sliding members must be protected instead of one or two. Another advantage of these robots is that they are easily adapted to work over large areas in a production line because of their linear motion.

  Cartesian robots are often used for assembly work. In such cases, they are generally mounted "upside-down" on an overhead gantry above a table. Such a configuration simplifies the problem of avoiding obstacles: the robot merely lifts up and travels over the objects on the table below.

- **Revolute geometry**—The most complex and versatile configuration is the revolute or anthropomorphic configuration. A revolute robot is constituted by two or more members connected together by rotary joints. These robots are easy to seal against the environment, have a large work envelope, and are quite flexible, often able to reach behind a vertical section. However, the

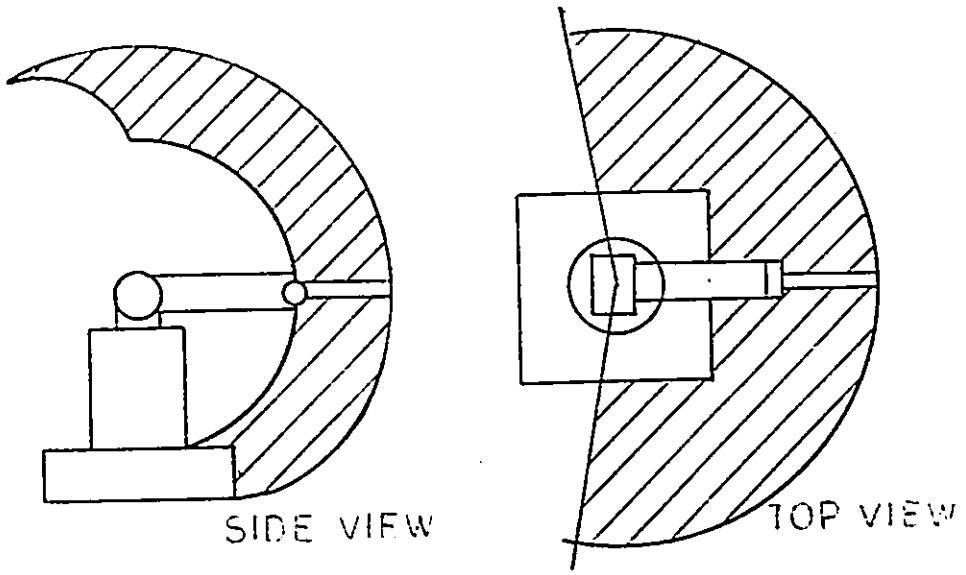

Figure 2.6: Revolute robot work envelope

complexity of controlling such a robot demands the use of computers. Before the advent of microprocessors, computers were so expensive that most robots built were of the simpler polar type. Today, almost all new robots are of the revolute type.

Degrees of Freedom

The number of *Degrees of freedom* is another characteristic of a robot manipulators. A degree of freedom is simply an axis of motion. Typical robots have five or six degrees of freedom. A robot with less than five degrees of freedom is seriously limited in its ability to position a part in space. A robot with six or more degrees of freedom is not as limited in its capability to position itself and objects; but, in many cases, there exist more than one robot configuration that will place a part in a given configuration, thus complicating the planning and programming process. Also, the more degrees of freedom a robot has, the more complex its control unit has to be. For these reasons, simple robots are often given five degrees of freedom, and the more complex robots six degrees of freedom. Because many applications do not require six degrees of freedom, many robot manufacturers offer the sixth degree of freedom as an option.

Power sources

There exist three power sources for robots: hydraulic, pneumatic and electric.

- **Hydraulic robots** are driven by linear hydraulic actuators: cylinders containing pistons pushed by oil or other hydraulic fluids pressurized by external compressors. Hydraulic systems are capable of generating large forces. Because of this, and because the robot arm does not contain the power source itself but only the much lighter actuators and pipes, hydraulic robots are capable of handling heavy loads. The best known hydraulic robots are the polar Unimates. The Unimate 4000 can handle a load of 120 kilograms at a full extension of over 3 meters. These robots are ideal in pick and place applications where heavy loads do not need to be positioned with high precision. Hydraulic robots can be operated in a simple stop and go fashion in which the cylinders are driven from one position to another without any intermediate stops, or in a servoed fashion in which information from position sensors is used by a closed loop controller to position the robot precisely at intermediate positions. Hydraulic rotary motors, similar in operation to electrical motors, are also used to drive robots. An inherent advantage of all hydraulic and pneumatic robots is that they are safe in explosive atmospheres, such as those encountered in painting applications, where any spark generated by an electric motor could cause an accident. However, they have some serious drawbacks. For one, the oil must be kept absolutely clean; any debris in the oil can seriously gouge a piston. Another problem is the high pressure (1500 to 2000 psi) of the oil; any break in a pipe can spread oil over large areas.

- **Pneumatic robots** are identical to hydraulic robots, except that air is substituted for oil. However, since air is compressible, pneumatic robots are not as powerful as their hydraulic counterparts. Conversely, pneumatic robots are generally lighter than hydraulic robots, do not require the same degree of filtering of the driving fluid, and do not wreak havoc when a pressurized air line breaks. The compressibility of air also makes pneumatic robots inherently compliant to external forces. Hydraulic and pneumatic robots are generally controlled by electrically operated solenoid valves, making computer control

possible, although not necessary: plug boards, limit switches with sequencers, and paper tape readers are also used.

- **Electric robots** generally utilize rotary joints rather than linear joints because of the nature of electric motors. However, many Cartesian machines use electric motors coupled to worm gears or rack-and-pinion drive mechanisms to realize linear joints. Electric robots are intermediary between hydraulic and pneumatic machines for power, but are generally more accurate. Electric robots have become the favorites in applications requiring high precision or only modest weight carrying capacity.

Accuracy and repeatability

A robot manipulator is also characterized by its *accuracy* and its *repeatability*. The accuracy of a robot refers to the difference between a a commanded position and the actual position the robot moved to. Accuracy is specified as the radius of a sphere centered on the commanded position in which the actual position of the manipulator's reference point is guaranteed to lie at the end of the motion. For a modern electric revolute robot, that error is often less than .5 millimeters.

Repeatability refers to the positioning error made by a robot manipulator when moving repeatedly to the same commanded position under the same conditions. Repeatability is better than accuracy in all cases. Accuracy is limited not only by the inherent accuracy of the robot's actuators and sensors, but also by the robot's compliance, i.e. its deformation under external loads, including its own weight. However, since a robot should deform identically under the same conditions when commanded to a given position, the actual positions reached should all be enclosed in a sphere whose radius is smaller than that of accuracy.

2.1.2 Grippers

Grippers or *end effectors* are used to hold parts when moving them. Grippers are usually developed in house for specific applications and attached to a purchased manipulator. Consequently, there is little consistency in gripper designs, but all should hold parts firmly without causing damage. A few general gripping techniques

can be identified:

1. Vacuum cups
2. Electromagnets
3. Scoops, ladles or cups
4. Hooks
5. Hands with two or more fingers
6. Adhesive tapes

While tools are not grippers per se, they are often mounted like grippers on the robot's wrist joint. Such an arrangement permits better handling of the tool than holding it in a gripper.

2.1.3 Sensors

To carry out many operations, robots require sensory capabilities. For example, finding objects requires vision or tactile sensing; picking up objects requires touch sensing to insure the object was picked up; avoiding collisions requires proximity or contact sensing, etc. Even moving from one point to another requires sensing to detect when the destination is reached. Sensing capabilities can be divided into several main groups:

1. **Propriosensing**—When moving about, the robot controller must be able to sense the current configuration of the arm to properly position the manipulator. Shaft encoders and rotary potentiometers on rotary joints, and linear variable differential transformers and linear potentiometers on linear joints are typical propriosensors. The limit switches used on simpler robots that turn off power to the actuators may also be considered forms of propriosensors.
2. **Touch sensing**—Contact switches and pressure sensors can be used in touch sensing to detect the presence of an object in the gripper, or to detect contact with a surface.

3. **Tactile sensing**—Artificial skins or other tactile array sensors can be used to determine surface contours and the outlines of parts. Handheld Coordinate Measuring Machines or CMM's can be used for this function but they require the robot to move around and record sets of contact coordinates [51].

4. **Force and torque sensing**—Force and torque sensing has become increasingly important in robotics, and is indispensable in robotic assembly, as discussed in section 2.4. Force/torque sensing can be used to prevent damaging parts or the robot itself from excessive forces, but its main role is to enable the robot to react to misalignment of parts in assembly by measuring the generated forces and torques.

5. **Proximity sensing**—Optical, capacitive, inductive or acoustic proximity sensors can be used to avoid collisions, give an indication of position, or detect components. In many installations, optical proximity sensors are used over contact switches to detect the presence of a component in the robot gripper. After supposedly picking up a part, the robot holds its hand in front of an optical sensor to verify is the part was in fact picked up. Such a method avoids running electrical wiring to the moving gripper.

6. **Vision sensing**—Vision can be used to recognize parts or to measure positions. Parts recognition is useful when components are delivered haphazardly on conveyor belts or slides. Stereo vision can provide extremely accurate positional and dimensional information. However, proper vision sensing is often difficult: attention must be paid to proper lighting and camera positioning.

2.1.4 Programming Techniques

Several techniques were developed to program robots. These techniques can be divided into two broad categories: *on-line* techniques in which the robot and its controller are used to create a new program, and *off-line* techniques in which programs are developed on a completely different system. The various programming techniques are outlined below.

1. **Teach-by-guiding and Teach-by-teach-box**— The simplest way to program a robot is to "teach" it on-line. When the controller is in teach mode,

an operator guides the robot through a series of motions and records its joint coordinates in the controller's memory. The resulting program consists of vectors of joint coordinates, possibly with the velocity of the movement from one position to another, and of activation signals to external devices. In playback mode, the controller commands the robot's joints through the sequence of recorded coordinates with a specified velocity. In teach-by-guiding, the robot is taught a sequence of operations by an operator that either moves the arm itself or a lighter more manageable teach arm. This method is ideally suited to program robots for spray painting in which the machine relentlessly repeats the actions of a master painter. In teach-by-teach-box, a hand-held control box is used to move the robot and record its joint coordinates. The teach-box can also be used to insert more complex instructions in the robot program, such as conditional branches on the status of sensors. Teaching allows quick programming of simple tasks and is the predominant method used in the industry, with upwards of 90% of robots being programmed this way [17].

2. **Joint-level programming**—There exist several levels of off-line programming. The lowest level of is the so called joint-level, where a program consists of sequences of joint coordinates, external device activation instructions, and conditional branches on the states of external sensors. The only advantage of this approach over the teaching approach is that the reprogrammed robot may continue working while the new program is being developed and tested by simulation.

3. **Manipulator-level programming**—In manipulator-level programming, the actions of the robot are specified in terms of the position and orientation of the Tool Center Point or TCP. The TCP is a point that lies somewhere on the roll axis of the robot (see figure 2.1) at a specified distance from the gripper mounting plate. All orientations of the wrist axes take place about this TCP. Mathematical techniques that account for the kinematics and dynamics of the robot arm are used by the controller to convert the manipulator-level program to a set of joint coordinates. Manipulator-level programming languages are by far the most widespread off-line programming languages in use today.

Traditional joint-level and manipulator-level programming languages, usually referred to as *robot-level* languages, have been imperative and often quite similar to BASIC in syntax. In such languages, the programmer must code all motions of the robot, and explicitly write procedures that verify the progress of the program and take corrective actions if necessary. Creating adaptive robotic systems, i.e. systems in which the robot modifies its behavior in response to external sensory information, has proven quite difficult because programmers must anticipate a large number of situations, and because of the difficulty of interpreting complex sensory information. The following approaches attempt to address this problem.

4. **Adaptive systems**—An adaptive system modifies its behavior in response to sensory information from its environment. Typically in robotics, adaptive systems are implemented in order to perform active compliance, described in section 2.5.2. To date, only one system, AL, developed at the Stanford Artificial Intelligence Laboratory, allows specification of adaptive behavior. AL was never implemented as a commercial product.

5. **Task-level programming**—Also called object-level programming, task-level programming allows the programmer to specify *what* operations to perform on objects rather than *how* those operations should be performed. While task-level programs written for simple tasks could conceivably be implemented by executing a procedure for each task-level statement, in more complex applications, a planner must derive the actual sequence of robot actions required to carry out the operations.

6. **Objective-level programming**—The most ambitious type of programming, objective-level programming uses only a description of the part to be manufactured or assembled. A planner derives the sequence of operations required and then converts those operations to robot actions. It is envisaged that the output of a CAD system could be used directly to prepare robot programs. So far, deriving programs from assembly drawings has proven impossible because of the inherent ambiguities contained in such drawings and because the human knowledge used in interpreting assembly drawings has not been captured.

This concludes the brief review of modern robotic technology.

2.2 Incentives For Using Robots In Manufacturing

In 1981, a group of graduate students mainly from Carnegie-Melon University, in cooperation with the Robot Institute of America, conducted a survey of about forty major U.S. corporations that have installed robots in their manufacturing plants. Results of this survey, referred to as the CMU survey appear in Ayres and Miller [1] and are briefly summarized below. This survey identified six major reasons for introducing robots in new or existing plants:

1. **Reduced labor costs**

   Respondents to the survey considered reduced labor costs as the main motivation for installing robots. In support of this, Joseph Engelberger, president and founder of Unimation Inc., states in his book: "The prime issue in justifying a robot is labor displacement. Industrialists are mildly interested in shielding workers from hazardous working conditions, but the key motivation is the saving of labor cost by supplanting a human worker with a robot" [21]. Furthermore, the Chairman of General Motors stated that whenever the wage of workers increases by $1 per hour, 1,000 more robots become economical [1].

2. **Eliminations of tedious and dangerous jobs**

   The elimination of tedious and dangerous jobs was second in importance after labor savings. Dissatisfaction with working conditions is a major cause of falling productivity. Removing people from dangerous or mindless jobs is often directly related to labor cost reductions: it is cheaper, in the long run, to replace humans by robots than chance paying, in the future, damage awards to former employees who have developed job related ailments.

3. **Increased output rate**

   Increase in the amount of goods produced in a given time period is often cited after the introduction of robots in a plant. Such increases are partly attributable to a revision of the manufacturing process itself, which is unavoidable when installing robots in an existing plant. Even so, robots may

increase productivity by working faster or longer than humans, and robots don't strike or take coffee breaks. Furthermore, the boredom of assembly work and the lack of fulfillment felt by the workers have created a severe problem of absenteeism in heavy and other industries costing millions of dollars per year in North America alone. Robots excel at mindless repetitive work: they are continuously on the job, except when they break down.

4. **Increased product quality**

   The survey showed that increased product quality was the second most important incentive for corporations planning to use robots and fourth for those already using robots. Prospective users of robots probably consider that Far East countries, especially Japan, achieve consistent high product quality by using robots extensively. By using robots themselves, they should be able to duplicate the foreign competition's results. In fact, most corporations that retrofitted robots in their plants reported an actual increase in product quality.

5. **Materials savings**

   Materials savings applies mostly to painting applications. In manual spraying, only about 30% of the paint actually gets on the surface to be painted. The rest is lost on the walls and floor, or is sucked up by large exhaust systems that protect workers from the paint fumes. Considerable paint can be saved by having a master painter "teach" the robots to paint. General Motors hoped to increase its spray efficiency by 50% by carefully training its robots.

6. **Increased flexibility**

   Flexibility in allocating a resource is needed in three cases:

   (a) when that resource, such as a machine tool or robot, is used in manufacturing more than one component within a given time period,

   (b) when that resource is used in a single manufacturing task within a period but the duration of that task is variable,

   (c) when a resource breaks down and production must continue with the remaining resources.

Changes in the demand for a product or breakdown of a machine may force a manufacturer to reallocate manufacturing resources. Reallocation is a potential problem for all manufacturers, especially for high-volume producers who have specialized their resources (hard automation machines and specialized labor) to achieve economies of scale. The next section briefly describes Flexible Manufacturing in which robots are a key component.

2.3 Flexible and Computer Integrated Manufacturing

Robots realize their full potential in Flexible Manufacturing Systems or FMS's. Flexible Manufacturing Systems were developed in response to an increasing need to reduce costs in small- and medium-scale batch production where flexibility in production schedules is of primary importance. But more than that

> It is predicted that there will be an increasing demand for small quantities of a large number of different models of items, and that only through the flexibility of the Future Factory concept will it be possible to produce them economically and of high quality. [17]

Obviously, the range of production run sizes for which FMS technology is more cost effective than manual labor or hard automation depends strongly on the product and on the existing costs of the various options. However, it is generally acknowledged that hard automation is more cost effective in large-scale production of a single item because dedicated machines are inherently more productive than general purpose machines. For small production runs, manual labor may be more effective since the setup time and costs are generally lower than for computerized systems.

2.3.1 Elements of Flexible Manufacturing Systems

An FMS is composed of the following elements:

1. **Machining cells** that integrate machine tools and robots. The majority of robots are employed in pick-and-place fashion to tend the machine tools,

changing cutters, drill bits, etc, or loading and unloading parts. However, robots are increasingly being used in machining operations per se, wielding cutting tools, deburring tools, etc, as well as performing other operations such as welding, flame cutting, etc.

2. **Assembly cells** in which robots assemble components into finished goods or larger sub-assemblies.

   To date, FMS's have been employed almost exclusively for machining operations; assembly has not been implemented, except in rare cases. Robotic assembly poses special problems which have not been completely solved yet. These problems are discussed in section 2.4.

3. **Inspection cells** in which automatic quality assurance testing is performed, such as verifying tolerances on components and inspecting goods for defects. Computer Aided Inspection, CAI, and Computer Aided Testing, CAT, is expected to grow in importance in the near future and integrated into the actual manufacturing process [71]. While special Coordinate Measuring Machines or CMM's are available, robots are expected to perform a larger proportion of quality control work, often within machining and assembly cells, because of their lower costs [51].

4. **Automated warehouses** in which robots provide parts to the work cells. The automated warehouse concept extends to the storage and shipment of finished goods: e.g. the automatic loading of trucks.

5. **A transport system** interconnecting the various work cells and warehouses. This transport system consists of Autonomous Guided Vehicles or AGV's and conveyor belts or similar devices. The flexibility of FMS's stems from the ability to reconfigure the transport system logically: i.e. the flow of parts can be modified to match the required processing of each product, or even in response to the traffic in the plant.

Computers are used to control and supervise the operation of FMS's. These computers are typically organized in a hierarchy, the lower level computers being responsible for controlling individual machines while the higher level computers

coordinate the overall operation of the system. Data acquisition systems collect information from the work cells and the transport system. The information collected ranges from time taken to complete a process to the time and number of retrials to complete individual operations. This information can be used to fine-tune the manufacturing processes, possibly in real-time, and gain information on the productivity of the plant.

Computer Integrated Manufacturing, or CIM, was developed in parallel to Flexible Manufacturing. While CIM and FMS are different concepts, the full potential of FMS's can only be achieved in a CIM environment. Computer Integrated Manufacturing designates a set of integrated software tools that assist in four phases of product development and manufacturing:

1. **Planning**—During planning, a product concept is determined.

2. **Design**—During the design phase, parts are conceived using CAD or Computer Aided Design tools, and manufacturing and assembly steps determined and tasks assigned to individual work cells using CAPP or Computer-Aided Process Planning tools.

3. **Implementation**—In the implementation phase, the various part programs that drive numerically controlled machine tools, the assembly programs, and other programs to control the operation of the transport system in an FMS environment are written, compiled, and tested. Most recent CAD systems are able to produce part programs in one of the most popular languages, APT, but no commercial system exists to produce assembly programs.

4. **Support**—The support phase comprises all activities performed after implementation: modification to programs, collection of performance data and streamlining of operations.

So far, CIM has met with some degree of success in many areas, but not in the area of robotic assembly where, to this day, no one has succeeded in creating a system capable of translating a high-level description of an assembly process into a sequence of executable robot commands. The second half of the chapter looks at the issues raised by robotic assembly.

2.4 Robotic Assembly

Robotic assembly, in which robots *assemble* elements into complete finished products or into subcomponents of a finished product, has so far resisted automation. This section looks at robotic assembly and the problems it poses. How assembly robots interact with the remainder of the plant is not considered. It is sufficient to assume that the interaction problem has a solution similar to the one used in Flexible Manufacturing Systems. Consequently, the term *robotic assembly* will be used rather than flexible assembly.

In machining, the machine tools are the primary transformation agents: they alone work the parts into desired shapes. The robots are simply used to tend the machine tools, loading and unloading parts in pick-and-place fashion. The following two reasons explain why present day robots are well suited for machining work:

1. **Contemporary robots have sufficient positioning accuracy** for manufacturing work. When serving a machine tool, a robot typically positions a part in the machine's jaws or chucks. Once in position, the robot generates a signal which can be used to command the closing of the jaws. The jaws actually compensate for any positioning error of the robot. Furthermore, the machine tool is usually able to sense the position of the part and take it into consideration when operating. Similarly, when the robot comes to remove the part from the machine tool, it can open its gripper wide enough to compensate for any positional error.

2. **Robots are easy to program** for pick-and-place operations. The sequence of motions is usually short and simple. Furthermore, because the number of operations a robot must perform is limited, switches can be mounted on the robot itself and on the machine tools to sense the robot's position. Such a technique allows even simpler programming and increases the accuracy of the robot.

The only manufacturing operation which is not so simple is seam welding. In some cases, vision or tactile feedback is necessary to allow the robot to follow a seam as the parts being welded heat up and deform. However, since seam welding

is a single operation, a dedicated robotic system can be designed to tackle the task. In fact, welding systems are among the main users of vision system. The General Electric WeldVision System, the Unimation Univision System, the Adaptive Technologies Adaptavision 3-D system and the Automatix Robovision Welding System are some examples of successful commercial systems employing vision. Another welding system using a laser range finder to track a seam is being developed by M. Rioux at the National Research Council of Canada and was demonstrated to the author.

Unfortunately, contemporary robots are not so easily adapted to robotic assembly because of the inherent difficulties which are discussed in the next section. Consequently, assembly has and continues to elude automation, except in a few rare cases. One such case is the use of so-called Adaptable-Programmable Assembly Systems, or APAS, in which robots and humans work side by side on a common assembly line, the robots performing simple assembly tasks for which they have enough intrinsic accuracy while humans carry out the more complex assembly tasks requiring active compliance. Unimation's PUMA robots were created especially for such an application. PUMA series robots are anthropomorphic, mimicking the shape and size of a human torso with a single arm. They are programmed using VAL-II, Unimation's robot-level programming language [26].

2.5 Problems of robotic assembly

As mentioned above, robots are not as easily integrated in small to medium scale assembly operations as they are in machining for the following reasons:

- robots have insufficient positioning accuracy and
- programming them for assembly is extremely difficult.

These problems are discussed below in detail along with possible solutions.

2.5.1 Insufficient positioning accuracy

The greatest problem of robotic assembly is the need for arbitrary precision in positioning the end-effector and its load. In manufacturing tasks where robots

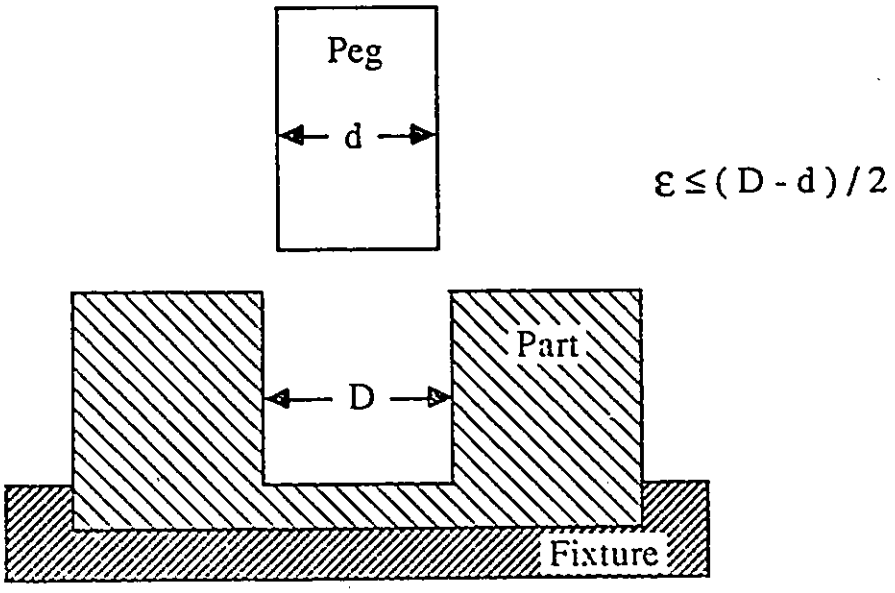

Figure 2.7: Insertion of a peg in a hole

position parts and tools in machine tools, the machine tools can compensate for the robots' low positioning accuracy. In assembly, there is no such possibility: the robot must position objects on or in another static object. The latter object will not move to meet the former, nor will it increase the size of its hole to facilitate an insertion operation.

Consider the "traditional" 2-dimensional peg-in-hole problem, illustrated in figure 2.7. In this example, the robot must insert a peg in a hole. While attempting this, three types of *errors* can occur: the peg may miss the hole partly or entirely, it may jam in the hole or it may wedge itself in the hole.

Looking at figure 2.7 it is easy to see that if $|\epsilon_x| > (D - d)/2$, where ϵ_x is the lateral positioning error of the robot from the commanded position P, the peg will partly miss the hole. If $|\epsilon_x| > D$ then the peg may miss the hole entirely. In fact, it would be an incredible chance that the peg would find its way into the hole.

Wedging or *jamming* of the peg may also occur. Wedging occurs when the peg enters the hole at such an angle that it touches the opposite sides of the hole before it gets very far. In such a case, the peg and hole begin to deform: any attempt at forcing the peg in any further results in more deformation. The only remedy for wedging is to remove the peg and try again. Wedging can occur at any time when the ratio of the depth of insertion l to the width of the hole D is less than the coefficient

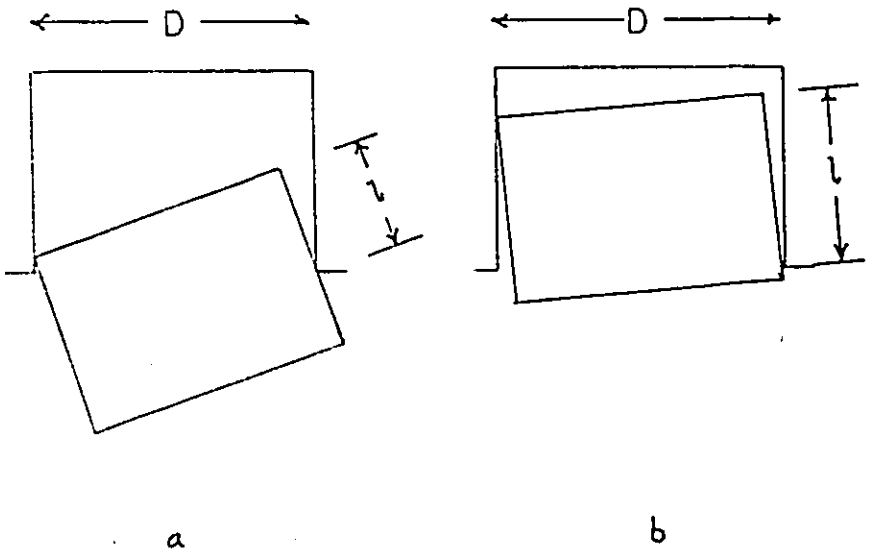

Figure 2.8: Wedging and jamming
(a) Wedging: $l/D < \mu_s$ (b) Jamming: $l/D \geq \mu_s$

of friction between the peg and the sides of the hole μ, i.e. whenever $l/D < \mu$ [48]. On the other hand, if $l/D \geq \mu$, wedging cannot occur, although the peg may jam. Jamming may be cured by changing the direction of the applied insertion force. The difference between wedging and jamming is illustrated in figure 2.8.

Building more accurate robots does not solve the problem of their insufficient positioning accuracy for the following reasons:

- there will always be cases where the size of a hole or component, or the clearance between two components is smaller than the best positioning accuracy available in a robot, and
- there will always be some uncertainty concerning the position and orientation of components. For instance, the size and position of holes drilled in a given part are known only to a limited precision, or *tolerance*. Furthermore, the position of fixtures used to hold parts while assembly is being performed is only known to a given precision. When a part is mounted in a fixture, the tolerances of the various elements compound, making the location of a particular feature even more uncertain.

Consequently, some mechanism other than more accurate robots is needed to successfully perform assembly. The use of compliance can solve these problems.

2.5.2 Compliance

Webster's dictionary defines compliance as the deformation of an object in response to the application of external forces. In robotics, the concept of compliance is better defined as the ability of a manipulator to *react* to external forces. As such, compliance is an inherent property of any physical object. When submitted to external forces, such as gravity or, of interest here, contact forces, a robot's members deform. Such compliance can be used advantageously. Nevins describes an assembly experiment in which a robot with a repeatability of .5 millimeter had to insert a crankshaft in the housing of a small gasoline engine, with a clearance of about .05 millimeter, a tenth of the robot's positioning accuracy [48]. Because of the inherent compliance of the robot manipulator and the presence of chamfers at the lips of holes, assembly was carried out successfully, but not without generating large forces. Such large forces may damage the components being assembled, or the robot itself. Clearly, such forces are unacceptable. What is therefore needed is a method to control the compliance of the manipulator to prevent damage to components. Two methods exist to perform this: passive compliance and active compliance.

Passive Compliance

Passive compliance attempts to build wrists and grippers that will yield to external forces. Nevins describes one such device consisting of two linkages [48]. One linkage, the remote-center compliance linkage, allows the peg to rotate about its tip if it is angularly misaligned with the hole. The second linkage, in line with the first, allows the peg to move from side to side without rotating in order to correct for lateral positioning errors. The remote-center compliance device reduces the chances of jamming the peg in the hole and reduces contact forces during assembly.

When combined with special design features, such as chamfers at the lip of a hole, passive compliance can successfully accomplish difficult assembly tasks in little time. Nevins reports succeeding in inserting a bearing in its housing with a clearance ratio of .0004 in a fifth of a second starting with a lateral error of 1 millimeter and

an angular error of 1.5 degrees [48].

Passive compliance has several drawbacks, the most important being the following ones:

1. Passive compliance does not completely solve the wedging problem, nor does it solve the problem of finding the hole in the first place. It is only of value once insertion has begun within a cone of a certain angle.

2. A particular passive compliance device is only suited for those tasks it was designed for. For instance, the remote-center compliance hand described cannot be used to slide a component on the surface of another component. Hence, for each particular type of compliance required, a specific device must be built.

Active Compliance

Active compliance is the use of force and torque sensing to modify immediately the behavior of the robot manipulator. It has the advantage over passive compliance of being programmable: the sensory information provided by force/torque sensors can be incorporated in decision statements within the controlling robot program to modify the actions of the robot appropriately.

By introducing *force/torque feedback* in an assembly system, all classes of compliant motions can be realized by the appropriate programs; there is no need to change grippers, except where the characteristics of the object to be held dictate it.

It is necessary to introduce the ideas of pure position control and pure force control to formalize the concept of active compliance. Mason [46] defines pure position control and pure force control in these terms:

> A manipulator with pure position control is a transducer whose input is in symbolic form and whose output is encoded as the position and orientation of the end effector in some predetermined coordinate system. Input and output are just a different encoding of the same six-dimensional vector function of time $\mathbf{p}(t)$. In pure position control the user is allowed to specify the effector position trajectory completely. Pure force control can be defined in an analogous way—the user provides a vector function $\mathbf{f}(t)$ which specifies the forces to be exerted by

the end effector.

The vectors alluded to by Mason are simply the end effector position vector and end effector force/torque vector:

$$\mathbf{p}(t) = \begin{bmatrix} x \\ y \\ z \\ \theta_x \\ \theta_y \\ \theta_z \end{bmatrix} \quad \text{and} \quad \mathbf{f}(t) = \begin{bmatrix} f_x \\ f_y \\ f_z \\ T_x \\ T_y \\ T_z \end{bmatrix}$$

where x, y and z indicate the position of the end effector reference point relative to some Cartesian coordinate system; θ_x, θ_y and θ_z rotations about the x, y and z axes; f_x, f_y and f_z forces along the axes and T_x, T_y and T_z torques about the axes.

To further clarify the distinction between the two types of control, consider the following cases:

- If the manipulator tries to move a fixed object, position control makes no sense. However the manipulator has total force control since any force or torque it applies to the object will be accepted since that object has no positional freedom.

- If the manipulator is in free space, then force control makes no sense since there is nothing to exert a force on. However, the manipulator has total positional freedom. In this case, position control is appropriate.

In assembly, the general case is to have situations in which the manipulator is free to move only in certain directions, other directions being blocked. To carry out assembly tasks successfully, a control technique is needed that combines both types of control, i.e. it is necessary to control position along certain axes while controlling forces along the remaining axes. In fact, force and position control can be combined into two types of motions: *compliant motions* and *guarded motions*.

Compliant motions occur when the manipulator follows the surface of an object. This is implemented by moving the manipulator while keeping a constant contact force. Guarded motions are used to approach and touch a surface without generating

excessive forces. This is implemented by slowly moving towards the surface to touch while closely monitoring force/torque sensors to detect the moment of contact.

By using pure position control in moving from one region of space to another, and by using compliant and guarded motions when assembling components, any assembly task should be possible. It even becomes possible to do a guarded motion to a surface and search for a particular detectable feature such as a hole by moving around and monitoring force and position sensors to detect the feature. Also, wedging and jamming can be avoided by designing appropriate sequences of compliant and guarded motions.

If compliant and guarded motions are so wonderful, why aren't assembly systems in widespread use today? This question brings us to the problem of writing assembly programs using active compliance.

2.5.3 Programming for assembly

Programming is usually simplified by decomposing the program hierarchically, with code at each level written using primitives provided by lower levels. In a robotic system, the lowest level provides direct control of a robot's actuators and sensors. In most applications, this is too low a level. For this reason, manufacturers usually provide higher level primitives that allow the programmer to command only the end position and motion velocity of the end-effector. In such systems, a computer in the robot controller converts these primitives into actual joint coordinates and individual link speeds.

These higher-level primitives constitute *control strategies*. A control strategy is simply a set of conventions "for combining input from higher level code and from sensors to produce signals for the actuators" [46]. In a "traditional" robot system, a pure position control strategy is used. This emphasis on position control is attributable to the simplicity of current machining tasks.

However, robotic assembly requires the use of more complex control strategies. When writing a program to perform assembly, the following steps must be taken:

1. The programmer must select the appropriate control strategy for a given task and geometry. The choice of such a strategy depends not only on the geometry of the target goal, but also on the starting point of the task. For example, the

strategy needed to successfully insert a peg in a hole depends on the geometry of the peg and the hole, on the materials of the peg and the hole which determine the coefficient of friction between them, as well as the initial position of the peg relative to the hole when the strategy is initiated. Generally, it is not possible to pick a single successful strategy unless the starting position can be guaranteed to lie within a region dependent on the selected strategy. Determining such a region is a very arduous task.

2. Since each control strategy is appropriate only for a single manipulator function and a restricted class of task geometries, a sequence of strategies must be employed to complete a single assembly operation. Determining a consistent sequence is also a long and difficult task, especially if some optimization, such as of execution time, is required.

So far, it was assumed that the control strategies picked by the programmer could be carried out by the robot. With today's robots, this assumption does not hold since commercial robot controllers implement position control only. However, sensors can be attached to most controllers, and their values read under program control. These values can be used to modify the control flow of the execution program. The programmer can therefore write the required control primitives for a particular robot and set of sensors. Fortunately, this need be done only once.

Given the difficulty in generating a program for assembly, it is generally acknowledged that task-level programming or automatic programming is needed.

2.6 Task-Level Programming

Task-level programming simplifies the programming of complex assembly tasks by letting the programmer specify only the desired geometric relationships among the objects being assembled rather than the sequence of robot dependent motions required to achieve these relationships. Thus, task-level programs are entirely robot independent, no paths or robot manipulator positions are specified by the user. A manipulator-level program with sensory input checking procedures and error recovery is derived automatically by a translator program or planner using the geometric, kinematic and dynamic description of all parts and of the robot manipulators.

2.6.1 Task-level instructions

A task-level program consists of a sequence of instructions specifying desired real-world states or goal states. The goal states may be expressed in numerical terms as in most robot programming languages, or in symbolic terms. Numerical goal specification is an extremely tedious task; the programmer must evaluate the coordinates and forces used in a program as well as the uncertainties in positions and forces. Furthermore, most languages constrain the goal state to a single point in the state space, so that positions and forces are limited to single values rather than ranges. This can lead to goal overspecification and in turn to failure.

A preferable approach is to define goal states symbolically. In such a description, the desired final positions of objects are specified symbolically, along with any spatial relationships or constraints that must be met, such as two faces being in contact, or two edges being aligned. Symbolic specification of goal states also allows for the expression of sets of goals satisfying the final constraints. Merely specifying that the pin should be inserted in the hole allows the pin to be inserted at any angle about its longitudinal axis of symmetry.

The mere specification of the geometry of the goal configuration is often insufficient. For instance, when specifying that two parts are to be screwed together, no mention of the torque to be placed on the screw is made. Furthermore, most people think in terms of actions to be carried out to achieve the desired goals rather than in terms of those final goals themselves. For that reason, the task-level language proposed in this thesis is action directed: task-level statements specify what must be done rather that the result of those actions. A task-level program is therefore a sequence of actions to achieve desired goal states, each *Task-Level Instruction* or *TLI* specifying a single goal state. Each TLI must be augmented with the specification of the goal constraints that must be satisfied. These constraints must be limited to the classes of constraints that can be sensed for. For instance, it is very difficult to check if two surfaces are coplanar using vision or touch. A simpler approach is to check that the coplanar surfaces are in contact with a third guide surface. By using compliant and guarded motions, it is possible to align parts with one another by utilizing guide surfaces and fixtures, rather than by relying on vision.

2.6.2 Minimum Requirements of Task-Level Programming

A task-level program must be translated to a robot-level program if it is to be executed. The problem of generating a sequence of actions to meet a goal specified in a task-level program is called *planning*. Automating planning on a computer requires a representation of the robot's world along with a representation of the robot actions and their effect on that world. It also involves controlling the search for a sequence of actions to fulfill the specified goals. Thus, task-level programming requires, at the very least, a planner coupled to a world model. The description of such a system is the subject of this thesis.

2.6.3 Previous work in this field

Automatic programming for robotic assembly has its roots in Carl Hewitt's Planner language, a pioneering work on planning performed in the late sixties [19]. While Planner was never implemented, it provided the initial framework for planning in robotics. Later, an interpreter for a subset of Planner, Micro-Planner, was implemented in LISP by Gerry Sussman, Terry Winograd and Gene Charniak at MIT [19]. Micro-Planner was first applied to Terry Winograd's famous block world program SHRDLU, but was quickly picked up by various researchers in artificial intelligence. The language possessed many features of PROLOG. However, Micro-Planner had several drawbacks which caused it to be abandoned in the early seventies.

AUTOPASS, developed at IBM by Lieberman and Wesley, was one of the first serious attempts at automatic planning for robotics [37]. AUTOPASS combined a planner to a world model which contained all the pertinent information concerning the layout of an assembly work cell, the geometry and dynamics of the robot manipulator, as well as the geometry and relevant physical properties of the objects the robot would have to manipulate through the cell. The core assumption in AUTOPASS was that the problem domain was sufficiently constrained that each assembly statement could invoke a prestored template that described a sequence of robot-level procedures to carry out the assembly statement. The calling of some of these procedures was dependent on the state of external sensors. Each template had all the motions, sensory tests and error correction procedures needed to carry out a specific statement, with only some parameters left to be specified at compile

time. The compiler would analyze a situation, select the appropriate template and compute the values of the template's parameters.

At about the same time, Russel Taylor was developing a similar system at the Stanford Artificial Intelligence Laboratory. He concentrated on computing the values of the template parameters based on estimates of the uncertainties in the environment.

Also at this time, the LAMA system was being developed at MIT by Tomás Lozano-Pérez [44]. LAMA was similar to AUTOPASS, except that it could modify a template by introducing sensing operations to reduce uncertainties in the environment.

All these systems assumed that the problem domain was sufficiently constrained that the use of templates was possible. Experience acquired with such systems show that, in general, this assumption does not hold: two slightly different geometries may require quite different templates, making the number of required templates unmanageable.

A somewhat different approach was taken in RAPT [53], a language designed to communicate a formal model of assembly tasks to robots. RAPT was not an automatic programming system; it concentrated only on specifying the assembly tasks and ignored issues such as collision avoidance or grasping. The characteristics of objects and the spatial relationships that must hold between some of their features at the end of an assembly operation were specified using a set of mathematical expressions.

All these attempts at task-level programming failed because they oversimplified the problem and because appropriate techniques to find collision free paths and synthesize fine motions were lacking. Consequently, researchers turned their attention to specific areas of planning, and especially to collision detection and the generation of collision free paths [6,11,22,33,60,12,18], and to the automatic generation of fine motions [7,23,43], to cite only a few of the most recent or significant publications.

After much work in path planning and some work in fine motion synthesis, Lozano-Pérez proposed a new task-level programming system called ATLAS [41]. In this system a fine motion synthesizer generates a sequence of fine motions whenever the planner cannot use predefined templates. ATLAS also attempts to automatically:

1. establish the physical layout of the assembly cell, in order to minimize the uncertainties in the work space and optimize some characteristic, such as assembly speed, hardware costs or operating costs;

2. determine the optimum method of introducing parts in the work cell. This implies that the system selects parts feeders in order to maximize performance and minimize uncertainties in the parts' configurations;

3. determine the type and placement of fixtures to be used in assembly according to some criterion;

4. generate sequences of fine motions (compliant and guarded motions) to carry out an assembly task with maximum likelihood of success;

5. Select grippers and grip configuration for objects.

6. Compute collision free trajectories when moving through the work space.

As of the date of writing, several components of ATLAS are known to have been developed, but the overall system is not complete.

Less ambitious attempts at designing compliant systems were also undertaken in the mid seventies. One such system, AL, was developed at the Stanford Artificial Intelligence Laboratory. AL is a structured language resembling Pascal but with more appropriate data types [17]:

- `SCALAR`s which are real values,

- `VECTOR`s which consist of three real numbers,

- `ROT`s, `FRAME`s and `TRANS`es which are homogeneous coordinate transformation matrices. `ROT`s consist of an axis of rotation and an angle. `FRAME`s contain both a position vector and a full rotation matrix to locate an object. `TRANS`es code the relative position of one coordinate system relative to another. They also consist of one position vector and a full rotation matrix. Homogeneous coordinate transform matrices are discussed in chapter 5.

AL can show the spatial relationship between objects by using the AFFIX primitive. When the FRAME representing the configuration of an object is "affixed" to the FRAME of another object, AL maintains the spatial relationship of both objects until an UNFIX statement is encountered. A summary of the AL commands appears on page 308 of Critchlow [17].

While AL does provide for force control strategies, it does not help the programmer determine which types of motions are required in a given geometry, or how to guarantee a given geometry for a given selected strategy.

Finally, one of the most recent programming systems for robot's, IBM's AML system, must be discussed, if only because it is produced by IBM [67,17]. AML is more than a language, it is a complete programming environment at the core of IBM's RS 1 Manufacturing System. Not only does AML provide a general programming language capable of controlling robots, it also provides means to design control interfaces to these robots. However, AML does not directly support compliance, although procedures can be coded for this purpose and used as primitives in an AML program.

2.7 Conclusion

This chapter examined current robot technology, from the configuration of robot arms to the problems of robotic assembly and the need for active compliance. It was also shown that the complexity of programming compliance requires the use of automated programming, preferably at the task or object level. Finally, previous work in this and related fields was reviewed very briefly.

Chapter 3

Proposed System for Robotic Assembly

3.1 Introduction

It was stated in section 2.5.3 that automatic or computer assisted programming was highly desirable in robotic assembly. The meaning of automatic programming has changed over the years. In the late fifties, it meant using high-level language compilers; today, it designates systems in which the user simply specifies what a software program is to do and *Computer Aided Software Engineering* tools, or CASE tools, figure out how to solve the problem and then write the code [72,58].

Automatic programming for assembly seeks to reduce the time, cost, and required programming skill for developing assembly programs by providing a higher-level robot programming language than those available today, and by integrating this programming environment with CAD databases and planning software. Automatic programming also seeks to expand the applicability of robotic assembly, especially towards smaller production run sizes.

This chapter proposes a task-level language to communicate assembly tasks to a robotic assembly cell in a Flexible Manufacturing environment, and describes the architecture and operation of a system capable of translating a task-level program into an executable program. A mechanism is also proposed to enable the system to recover from unexpected situations during execution. The language does not

provide for the specification of the overall control of the manufacturing process, such as the number of components to assemble or the routing of those components through the plant. Control can be specified using the Process Activity Language PAL [35] or its predecessor, the graphical language called Process Activity Diagrams or PAD's [52]. In the context of PAL or PAD's, a task-level program specifies a *single activity* that is activated by the control program whenever all preconditions are satisfied and executes *once* to its end. The control program may activate the assembly program several times to make multiple instances of an assembly.

The chapter begins with a description of the proposed task-level language and follows with a discussion of the architecture of the system and the function and characteristics of each module of the system.

3.2 Proposed task-level language

In standard practice, assembly operations are described using assembly drawings or *Gozinto charts* (see Chase [15] for several examples). Assembly drawings are inherently ambiguous and extracting assembly information from them requires experience and common sense. Gozinto charts are prepared from assembly drawings and clearly define how parts are to be mated and in what order, and often the overall material flow, i.e. at which work stations particular assembly operations are carried out. Gozinto charts are used mainly as a planning tool. *Process work sheets* are used to explain assembly work to workers. Also called *operation and route sheets*, process work sheets are used to describe, in detail, machining and assembly operations to be carried out to make a part. For each operation, the work cell or department where the operation is to be carried out is indicated, along with the specific machine, tools, and fixtures that are to be used. Setup times for each operation and the number of parts processed per hour are also indicated. Figure 3.1 shows an assembly sheet meant for a human assembler to build a toy wagon.

The illustrated assembly sheet shows that assembly descriptions are task oriented, describing in one sentence an entire sequence of motions to achieve a goal. Furthermore, these tasks are specified in an active voice describing the actions to be carried out rather than the goals to be met. Furthermore, it can be seen that

OPER. NO	OPERATION DESCRIPTION
1	Position wagon on fixture
2	Position rear axle support on wagon and hand fasten with 4 .25-20 UNC-2A X .5 screws to nuts
3	Insert rear axle
4	Tighten rear axle screws to nuts
5	Position front axle assembly and hand fasten with with 4 .25-20 UNC-2A X .5 screws to nuts
6	Tighten front axle assembly screws
7	Position rear wheel #1 and fasten hub cap
8	Position rear wheel #2 and fasten hub cap
9	Position front wheel #1 and fasten hub cap
10	Position front wheel #2 and fasten hub cap
11	Position wagon handle shaft on front axle assembly and hand fasten bolt and nut
12	Tighten bolt and nut

Figure 3.1: Assembly sheet for a Toy Wagon
Adapted from [15, page 229]

all statements begin with a verb, followed by the objects to be acted upon, and the relationships, whether stated or implied, that should hold between these objects at the end of the operation. The proposed language has an identical syntax: each statement begins with an action verb followed by the objects to act upon and the required relationships between them at the end of the operation.

However, the assembly sheet illustrated assumes that the human assembler has sufficient knowledge to supplement the missing information, such as the exact positioning of the axles on the wagon, the location of the screws, the tools used to tighten the screws and nuts, and the torque to be applied when fastening them. Such information must be supplied explicitly in the task-level program or the World Model. Consequently, the language provides constructs to describe the exact geometric constraints or relationships between objects in symbolic terms, to specify the use of any tools, and to specify minimum and maximum torques and forces exerted on objects. The World Model provides information about the expected location and orientation of objects, and estimates on the associated uncertainty.

This syntax allows any assembly task to be specified in a manner familiar to people in industry. The list of primitives required to specify all possible assembly tasks in a given system depends on the specific assembly operations in that system. Some operations will be common to all assemblies, but others are dependent on the fastening techniques in use and assembly tasks to be performed. New primitives must be created to meet specific requirements.

Unlike most programming languages, the proposed language does not support variables or assignment, nor does it provide conditional branching or looping statements. Such statements are *never* used in assembly sheets and are therefore not required here. The simplicity of the language makes parsing it trivial, as will be shown in chapter 6. Looping and branching occurs at the control level as mentioned in section 3.1, and at the underlying execution level that implements compliance using sensory feedback loops.

Since the system is to be implemented in LISP, the syntax of the language was selected to be easily manipulated by a LISP program. The reasons for selecting LISP are outlined in section 3.4. The syntax of the language is described fully in figure 3.2.

In the following description of the language, BNF (Backus-Naur Form) is used

with the addition of {...} to designate an optional item. Names of task-level primitives are shown in upper-case TYPEWRITER font in the body of the text.

3.2.1 CREATE primitive

The main task-level program consists of a CREATE statement which will create a *named* node in the World Model with the given object name that represents the main part to be assembled. In the body of the CREATE statement will appear object manipulation statements, as well as other CREATE statements which cause the creation of sub-assemblies. Execution of the body CREATE will cause the new named node to be connected to already existing objects.

The CREATE statement also divides a task-level program into a number of logical modules. A CREATE block is not a procedure: it has no parameters and must appear in the chronological sequence where it is used.

Since the system is to be implemented in LISP, and because of some restrictions imposed by the language, each object in the World Model must have a unique name.

The CREATE statement has the following structure:

```
(create <object-name> by
  (<object manipulation statement 1>
   :
   <object manipulation statement n>))
 end <object-name>)
```

3.2.2 Object manipulation primitives

These primitives are for the control of the manipulation of objects directly implicated in the assembly of a final part or sub-part present in the assembly cell. They do not control the fastening of objects, or the control of tools.

MATE-TO primitive

```
(mate-to <object-1> <object-2> such-that
  (<constraint 1> ... <constraint n>))
```

```
<module> ::=  ( create <object-name> by ( <list of statements> )
              end <object-name> )

<list of statements> ::=  ( <statement> )
                        | ( <statement> ) <list of statements>

<statement> ::=  <manipulation statement>
               | <fastening statement>

<manipulation statement> ::=  <module>
                            | <object-verb> <operand-1> {<operand-2>}
                              such-that ( <list of constraints> )

<fastening statement> ::=  <fastening-verb> <operand-1-class> <operand-2>
                           using <tool-name>
                           such-that ( <list of constraints> )

<object-verb> ::=  mate-to | fit-in | fit-over | extract-from | drop-on

<fastening-verb> ::=  glue-on | screw-in | rivet-in

<list of constraints> ::=  ( <constraint> )
                         | ( <constraint> ) < list of constraints >

<constraint> ::=  <position constraint>
                | < force constraint>

<position constraint> ::=  against <face-1> <face-2>
                         | inserted-in <object-1> <object-2>
                         | height <constraint range>

<force constraint> ::=  force <constraint range>
                      | torque <constraint range>

<constraint range> ::=  ( <minimum value> <maximum value> )
```

Figure 3.2: Syntax of the proposed language

This primitive places object-1 against object-2 with the list of constraints specifying which faces are to be in contact. This primitive is to be used when the geometric constraints are relaxed, i.e. there is no danger of jamming or wedging.

FIT-IN primitive

(fit-in <object-1> <object-2> such-that
(<constraint 1> ... <constraint n>))

This primitive mates two objects, where object-2 presents a geometry that may cause problems, such as a hole. Here, the geometric constraints are quite tight and care must be exercised in moving one object in the other's space to avoid jamming or wedging. The constraints are again a set of surfaces that should be in contact. Insertion to a given depth relative to a position, such as one surface of object-2, is not allowed because of the impossibility of guaranteeing that depth in general.

FIT-ON primitive

(fit-on <object-1> <object-2> such-that
(<constraint 1> ... <constraint n>))

This primitive mates two objects, where object-1 presents a difficult geometry, such as a hole. This is similar to the FIT-IN primitive, but now object-1 is fitted over object-2. A typical application of this is the placement of object-1 on a fixture where holes in object-1 are inserted through pins in the fixture to form a stacked product, a quite common operation in assembly.

EXTRACT-FROM primitive

(extract-from <object-1> <object-2> such-that
(<constraint 1> ... <constraint n>))

This primitive is the opposite of the FIT-IN, FIT-ON and MATE-TO primitives. It removes an object from an open or enclosed position. The removed object is held in the gripper and moved to the nearest free space.

DROP-ON primitive

(drop-on <object-1> <object-2> such-that
(<constraint 1> ... <constraint n>))

This primitive will position object-1 in free space above object-2 and open the grippers of the robot end effector until object-1 has been detected to have fallen, or until the grippers are opened wide enough to guarantee object-1 has dropped. This primitive is used in conjunction with the extract primitive; if an object is extracted and discarded, it is a waste of time to MATE it to a waste basket or something of the sort. This primitive will simply get rid of it over a target area defined by object-2. The list of constraint can specify a maximum and minimum height from which object-1 should be let go relative to a designated surface or point of object-2.

3.2.3 Fastener manipulation primitives

These statements cause fetching of the given fastener, and may cause a tool change at the beginning of operation. Fasteners are *generic* objects: it is not convenient to treat each fastener of a given class as a unique object. Rather, all fasteners of a given class are considered "identical" and each class is represented by a single object in the world model. Each fastening statement assumes that a new instance of a fastener of a given class is required. Individual fasteners can only be distinguished by creating separate classes for each instance. Thus, if in a given assembly program it is necessary to refer to two or more instances of a bolt of a given class, separate classes would have to be created for each instance.

SCREW-IN primitive

(screw-in <screw-type> <hole-name> using <tool-name>
from <screw-feeder> such-that (<constraint 1> ... <constraint n>))

RIVET-IN primitive

(rivet-in <rivet-type> <hole-name> using <tool-name>
from <rivet-feeder> such-that (<constraint 1> ... <constraint n>))

GLUE-ON primitive

(glue-on <glue-type> <surface-name> using <glue-dispenser>
such-that (<constraint 1> ...<constraint n>))

3.2.4 Constraints

Constraints can be specified in all statements. They may also appear in the World Model, explicitly or implicitly as default values. Specifying constraints in the statements that also appear in the World Model permits the detection of logical errors.

AGAINST constraint

(against face-1 face-2)

The AGAINST constraint generates a strategy which uses force feedback to detect when a surface has been contacted.

INSERTED-IN constraint

(inserted-in object-1 object-2)

The INSERTED-IN constraint specifies that a sub-component of a part is to be inserted in a feature of another part. It may be that satisfying this subgoal automatically satisfies the goal in the task-level statement. The satisfaction of this constraint then becomes the main goal.

HEIGHT constraint

(height (minimum-value maximum-value))

The HEIGHT constraint is only used in the DROP statement to specify the maximum and minimum heights at which an object can be released from the target surface.

```
(create motor by
  ((fit-on motor-casing fixture)
   (fit-in rotor-assembly motor-casing
    such-that ((inserted-in rotor-assembly.shaft motor-casing.hole)
               (against rotor-assembly.shaft.bottom motor-casing.hole.bottom)))
   (fit-on cover fixture
    such-that ((against cover.bottom motor-casing.top)))
   (screw screw-type-1 cover.hole1
    such-that ((torque (10 12))))
   (screw screw-type-1 cover.hole2
    such-that ((torque (10 12))))
   (extract-from motor fixture)
   (drop-on motor out-conveyor
    such-that ((height (.05 .5))))
   end motor)
```

Figure 3.3: Example electric motor assembly program

FORCE and TORQUE constraints

```
(force (minimum-value maximum-value))
(torque (minimum-value maximum-value))
```

The FORCE and TORQUE constraints specify the minimum and maximum forces and torque that can be used when pressing down on or turning an object.

3.2.5 Example assembly program

Figure 3.3 shows an electric motor assembly program to illustrates the use of the language. It is assumed that the motor casing is complete and is provided with guide holes which will fit over guide pins in the fixture. The motor's cover also fits over the same pins. The whole assembly is then screwed together, and dumped on a conveyor belt.

In the example, *motor-casing.hole.bottom* is a typical name. It refers to the *bottom* of the *hole* in *motor-casing*, the motor's casing.

3.2.6 Extension of the language

The proposed language is not complete. For example, a manufacturer may employ a certain type of fastener that requires a special insertion tool and certain sequences of motions. In order to maintain a task-level description of the assembly process, an instruction must be created for this type of fastener. Fortunately, new task-level instructions that reflect particular assembly processes can be easily created in this system. New instructions need not have a unique verb name but may be distinguished from other instructions by the type of their parameters.

Verification statements are a type of instruction a user may want to provide. It may be necessary for the robot to verify that an assembly is within specification by making measurements with a hand-held coordinate measurement tool or by tactile feedback, or simply to verify results using vision. Since the Planner must and has control over the cell's sensors, such statements can be created.

The language should also provide for a declaration section in which the components to be used or created are described. Such declarations would allow the system to retrieve the description of existing objects from a CAD database and construct the appropriate World Model. It would also provide a-priori information about objects to be created, such as their functional and geometric classes, as discussed in section 5.3.1.

3.3 Overview of robotic assembly system architecture

This section outlines the architecture of the software system that converts the task-level program into an executable program and that monitors its execution. The function of each module is described in detail, as well as the overall operation of the system.

Figure 3.4 illustrates the software architecture of the proposed system. The function and operation of each module is outline below.

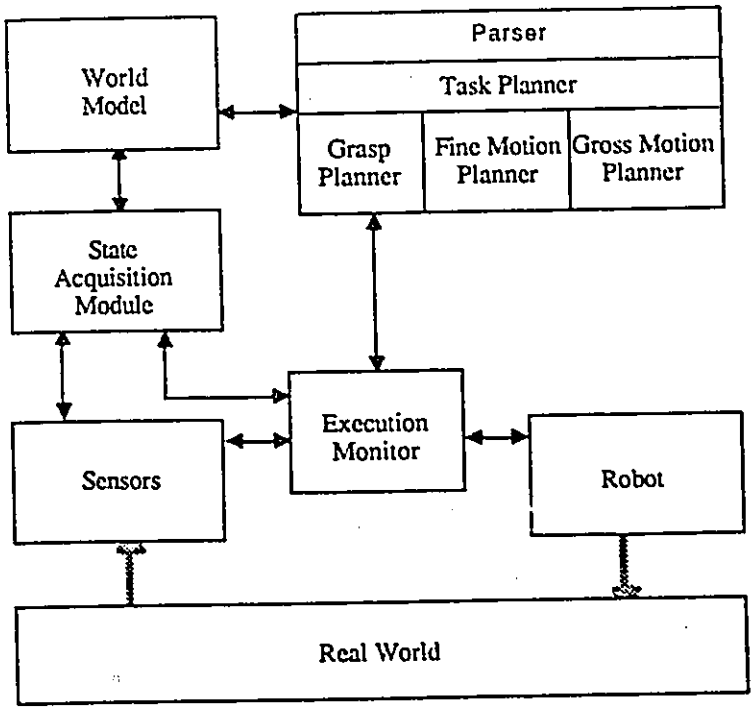

Figure 3.4: Robotic Assembly System Architecture

3.3.1 World Model

The motions of the robot are constrained by the shape, position, and properties of objects, and by the capabilities of the robot itself. This information is stored in a database called a World Model. The World Model differs from CAD type databases in that

1. it models the full three-dimensional nature of parts and their physical properties rather than simply their appearance when projected in two-dimensions as many CAD systems do, and

2. it models the possible variations in shape and size of objects of a given class. The amount of possible variations are called *tolerances*.

Objects are represented internally as nodes in a binary tree as illustrated in figure 3.5. Leaf nodes represent primitive or *predefined* solids. Primitive objects are defined using a Boundary Representation scheme (see section 4.2.3 item 5). Primitives are volumetric entities constructed from the union of surfaces stored internally with their bounding edges. Intermediate and root nodes represent either

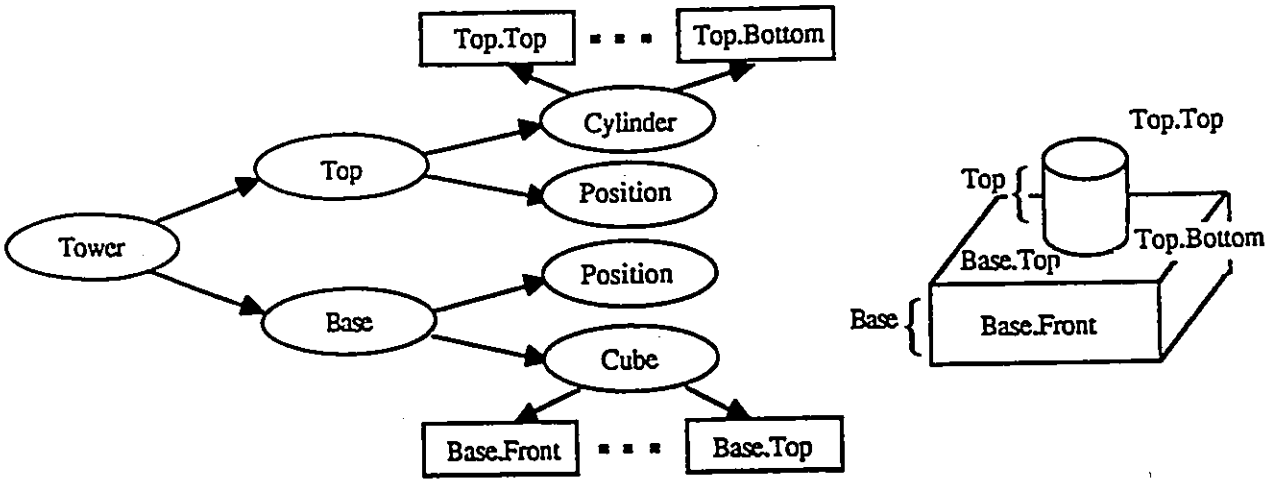

Figure 3.5: Example objects and corresponding names

fixed or variable geometry compound objects. Fixed geometry objects are represented using Constructive Solid Geometry binary trees (see section 4.2.3 item 3) and the "gluing" operation (see section 4.2.4) rather than the more general union and difference operations. The representation of variable geometry objects has not been fully addressed in the literature at this time, and will be ignored here. With this scheme, a new part is created from its constituent sub-parts by attaching the sub-trees representing the sub-parts to the node representing the main part. This scheme also allows the computation of the properties of a compound object from the properties of its constituent objects.

Symbolic names can be attached to any sub-component at any level, and to specific surfaces of a primitive object. Since object descriptions are stored in LISP global variables and retrieved by name rather than reference, object names, which correspond to the variable name, must be *unique*.

As planning progresses the World Model information is updated to reflect the state of the real world. Note that this entails more than just updating the information on the position and orientation of objects: as the assembly progresses and parts are successfully mated, new objects with new physical properties are created. The database must also contain information on the dynamics and sensing abilities of the robot to be able to generate sensor-based motion strategies. The World Model

provides structures to hold explicit information about objects, and provides functions to derive other properties not stated explicitly, such as the center of gravity of an object based upon the center of gravity and relative position of its constituent elements.

Ideally, the World Model would be constructed automatically from the CAD database. However, at this time there is no standard representation scheme for 3-dimensional objects and their dynamic behavior in CAD systems. Therefore there seems little point in investigating the extraction of World Model information from a CAD database, since such an extraction mechanism would be directly tied to a particular CAD system. Rather, effort was put on investigating and finding a simple internal representation scheme for objects in order to create a usable World Model and allow latter experimentation both with various representation schemes and various planning algorithms. At this time, the World Model is created by executing a LISP program that creates objects and specifies their relationships and properties. There is no mechanism in place to view the created objects graphically or to check their validity at this time.

3.3.2 The Planner

In can be seen from figure 3.4 that the Planner is composed of five modules: a Parser, a Main Planner, a Gross Motion Planner, a Fine Motion Planner, and a Grasp Planner.

The overall Planner converts the task-level program into a robot program that has a very high probability of succeeding without errors, such as dropping a part or jamming component in a hole, even in the presence of uncertainties in the real world. This is done by generating for each task-level instruction a ***partial plan*** that will take the system from an initial state, the result of the previously executed Task-Level Instruction or *TLI*, to the final goal state specified by the current TLI. Each partial plan consists of several definite robot actions and sensor operations.

Design decisions that simplify planning

The difficulty in planning is the non local nature of decisions taken in each step of a partial plan. For instance, the choice of a particular grasp configuration determines which motions are required to position the held part at its destination. In turn, the choice of a grasp configuration is influenced by the subsequent steps. For example, the gripper cannot grasp a part in such a way that one of its fingers is in contact with a surface to be mated to another surface. Because of these dependencies, planning initially appears as a monolithic operation; everything must be decided simultaneously. In fact, the planning process itself can be broken down and simplified by making the following design decisions:

1. **The Planner performs only grasp, gross and fine motion planning—** Lozano-Pérez's proposed in his ATLAS system to plan the optimum layout of the cell and to determine how parts should be introduced in the work environment, as discussed in section 2.6.3. In a FMS or batch production environment, changing the layout of work cells too often is not cost effective. Consequently, a cell layout, including feeders and fixtures, is likely to be fixed and predetermined, making layout planning unnecessary so that only grasp, gross and fine motion planning need be performed.

   The function of the Gross Motion Planner is to compute collision free paths for the manipulator from one region of space to another using only *pure position control.* Optimization could be performed on the paths to increase productivity, i.e. select the fastest possible routes. The Fine Motion Planner's function is to select or synthesize sequences of guarded and compliant motions and sensing operations, and to determine the region of space from which the generated sequence guarantees that an assembly operation will succeed, even in the presence of uncertainties in the environment. Finally, the Grasp Planner selects grippers and stable grasp configurations that do not interfere with gross and fine motions when picking up and moving objects but does not generate any motions.

2. **The Planner treats each task-level instruction independently**—Since the planner only does grasp and motion planning, each task-level instruction can be treated independently from all other instructions, unlike Lozano-Pérez's ATLAS system. In this system, effects of previous instructions are taken into account when the World Model is updated to reflect the new cell state.

   The assumption of independent treatment of each task-level instruction is possible because

   (a) of the restriction of planning to gross motions, fine motions, and grasping, and because

   (b) the problem domain has been sufficiently constrained by the selection of an appropriate description level, as in LAMA or AUTOPASS, discussed in section 2.6.3.

3. **Each independent partial plan consists of a set of *predefined* planning steps**—Where LAMA and AUTOPASS invoked prestored templates of robot-level code, this system invokes *prestored templates of subgoals*, giving it more power to synthesize sequences of robot motions, while avoiding the inherent complexity of ATLAS.

   While the subgoals for each TLI are predefined in a *subgoal template*, the exact manipulator commands and their parameters are determined by the Planner based on (a) the initial state of the system, (b) the current TLI and (c) its list of constraints. A typical set of subgoals for a generic TLI might be the following:

   - generate a sequence of fine motions away from the current object (presumably the object ungrasped in the previous TLI);
   - generate a sequence of gross motions from the current position to the vicinity of the object to be moved in the current TLI;
   - generate a sequence of fine motions to grasp the object;
   - generate a sequence of fine motions away from the initial position of the held object;

- generate a sequence of gross motions to the vicinity of the destination position of the held object;
- generate a sequence of fine motions to position the held object at its destination or to perform some operation with a tool;
- generate a sequence of fine or gross motions to ungrasp the object or the tool.

It may also be necessary to effect a tool or gripper change, in which case, appropriate actions must be taken.

Note that the Planner makes no attempt to satisfy the goals sequentially. Quite the contrary, it attempts to solve all goals simultaneously. Since this is impossible, it will find solutions to each goal until all goals are solved in a consistent manner, or until it has expended all possibilities.

The use of predefined subgoals means that the Planner does not create subgoals *dynamically* when translating a task-level program into a robot-level program. When an object is to be placed on top of another object whose top is cluttered, some planning systems would introduce a new subgoal: the removal of the obstructions. Introduction of subgoals "on the fly" is called dynamic goal creation.

Dynamic goal creation can lead to several problems. One problem is that automatically generating new subgoals leads to unexpected robot behavior. Furthermore, there is always a danger that satisfying the new subgoal will void a solution to a previous subgoal, or make the original goal unsatisfiable. This problem is commonly called *subgoal annihilation*. Dynamic subgoal creation can also lead to some form of *thrashing*, in which the system spends most of its time moving the obstructing object from one place to another because it keeps getting in the way, or it may solve one problem only to create more.

An impossible operation in this system is considered a clear indication of a logical error in the program. Consequently, whenever a goal cannot be satisfied, the Planner flags the impossible operation and the reasons why it cannot be performed before stopping.

While the Planner uses predefined goals, each planning module is free to introduce as many *local* subgoals as it requires to solve its specific problem. For instance, the Gross Motion Planner may plan a path from A to B by first planning a path from A to C, and then planning a path from C to B, and so on recursively. Intermediate points constitute subgoals, but are local to the Gross Motion Planner.

4. **Use of constraint propagation**—The possible values of the parameters in a step are constrained by the physical state or configuration of the system, the capabilities of the robot, the goal state to be achieved and the previous and subsequent steps. These constraints provide a clean and efficient way to represent and handle the dependencies between the various steps of a partial plan. The use of constraints is said to achieve "global consistency through local computation" [76]. This means that a consistent plan can be prepared by planning each step independently from the others and by propagating symbolic and numeric constraints back and forth through the various steps. Chapter 6 briefly discusses the theory of using constraints in planning.

5. **The cell uses a single manipulator**—Some researchers have proposed to use two or more robots working cooperatively to accomplish a single task, much as a human uses both hands cooperatively. A typical suggestion is to hold a part in one manipulator while holding another part in another manipulator, and commanding both manipulators through sequences of motions to mate the parts. The use of multiple manipulators in cooperative operation raises two key issues:

   (a) **determination of cooperative strategies**—A sequence of motions has to be produced for each robot manipulator that guarantees success of mating operations.

   (b) **synchronization**—The manipulators have to move in lock-step in order to mate parts in the proper positions and orientations, and have to avoid colliding the parts together and avoid colliding between themselves.

   The synchronization issue has received some attention and can be resolved [45]. However, no method exists to assign tasks to each hand automatically

in general cases. In fact, observing people reveals that, when faced with a new "situation", some experimentation is performed until an actual sequence of actions is found. Dufay and Latombe [20] have proposed an approach to automatic robot programming for assembly based on such experimentations. By monitoring sensors during an *on-line* training phase, their system decides which motions are to be executed. The system generates multiple traces of execution by performing a given task several times. Following this, during an off-line "induction" phase, the system applies transformation rules to the traces generated during the training phase and builds a manipulator-level program including symbolic variables, conditional statements and loops in the LM manipulator-level language. Dufay and Latombe thus avoid the use of a World Model containing uncertainty information; this information is contained implicitly in the execution traces generated during the training phase and shapes the resultant robot program during the induction phase. Unfortunately, this approach

(a) diminishes productivity since, while the robot is training, it does not perform any useful task;

(b) and prohibits the preparation of programs ahead of execution time. Indeed, since the uncertainties are inferred from the robot's environment during the training phase, the resultant robot program is only valid for a given configuration of the work cell. When a new product is to be assembled, the work cells need to be configured before the robots are trained and work can begin.

(c) Furthermore, there is no possibility of handling situations that did not occur in the trial runs, such as dropping a part.

Operation of the system

Conversion of task-level programs into partial plans is accomplished off-line, which offers several advantages over on-line techniques:

1. It allows the robot to continue working while its new program is being prepared, thus increasing its productivity.

2. Unlike some on-line programming techniques, off-line programming allows programs to be written well ahead of time of the production start date of a new component. This also means that fewer "programmers" are required than with some on-line programming techniques since they can work over longer periods of time.

3. Programming can be done on work-stations or on a mainframe computer that integrates powerful software tools with a CAD/CAM system from which the description of parts can be extracted. Furthermore, translation need not proceed in real-time, as it would were translation done on-line, thus allowing the use of fewer computers, or slower less expensive ones.

4. The execution of the resultant program is simulated during the off-line translation process, thus avoiding potentially dangerous situations to the programmer and equipment in case of gross programming mistakes. Furthermore, that simulation may provide estimated execution times for given assembly tasks which can be incorporated in the manufacturer's overall planning and scheduling. Obviously, the validity of any simulation is dependent on the validity of the description of the real world supplied to it.

When converting a task-level program off-line, the *Parser* reads each TLI, verifies its syntax and, if correct, expands it into a predefined sequence of planning subgoals. The initial constraints on the values of the *planning variables*, the variables appearing in a partial plan, are provided by (a) the current state of the world contained in the World Model, (b) the final goal state expressed in the current TLI, and (c) the explicit constraints specified in the TLI. The Parser presents these initial constraints and the final goal in a uniform manner to the Main Planner.

Extensibility of the language is provided by having a parser for each task-level instruction verb, something easily done using LISP. Each parser "knows" about the various types of its parameters and can thus select the appropriate subgoal template and present the required set of constraints and the desired goal state.

The Main Planner itself is then invoked. It will call upon the Grasp, Fine Motion and Gross Motion planners according to the predefined subgoal template associated with each task-level instruction. Each planning module will either

- produce for each invocation a new sequence of robot actions to satisfy the goal and a set of constraints that describe the set of preconditions for which the sequence of actions is valid, or
- fail with or without a list of constraints that describe situations in which it may be successful.

It is possible that decisions taken in one planner prevent another planner from finding a solution to its goal. Consequently, the Main Planner incorporates a *backtracking* mechanism by which incorrect decisions can be undone. By re-invoking a planner with a new set of constraints, a new solution can be attempted. By propagating constraints back and forth between planners, possibilities can be explored until a consistent plan is found or all possibilities are exhausted, in which case the Main Planner signals failure and stops. The complete planning algorithm is outlined in chapter 6.

The output of the planner is a sequence of partial plans forming a complete plan or assembly program. A plan consists of a sequence of *plan primitives* which are compliant and non-compliant motion commands, as well as sensor commands, with fully instantiated parameters. It is assumed for simplicity in this thesis that the underlying executor supports an AL-type language, described in section 2.6.3. However, if such an executor is not available, compliance could still be achieved by having the Planner expand compliant motions into procedures of non-compliant motions using sensory feedback. Such procedures would be written before hand and stored in a library.

In order to carry out error recovery, the planner must also output the original task-level instructions and a sequence of instructions to update the World Model. This output is then fed to the Execution Monitor for on-line execution. Figure 3.6 illustrates the output of the planner.

3.3.3 Execution Monitor, State Acquisition Module, and Evaluator

All previous task-level programmed assembly systems produced executable code meant to execute without supervision. Some assumed that a correct plan would

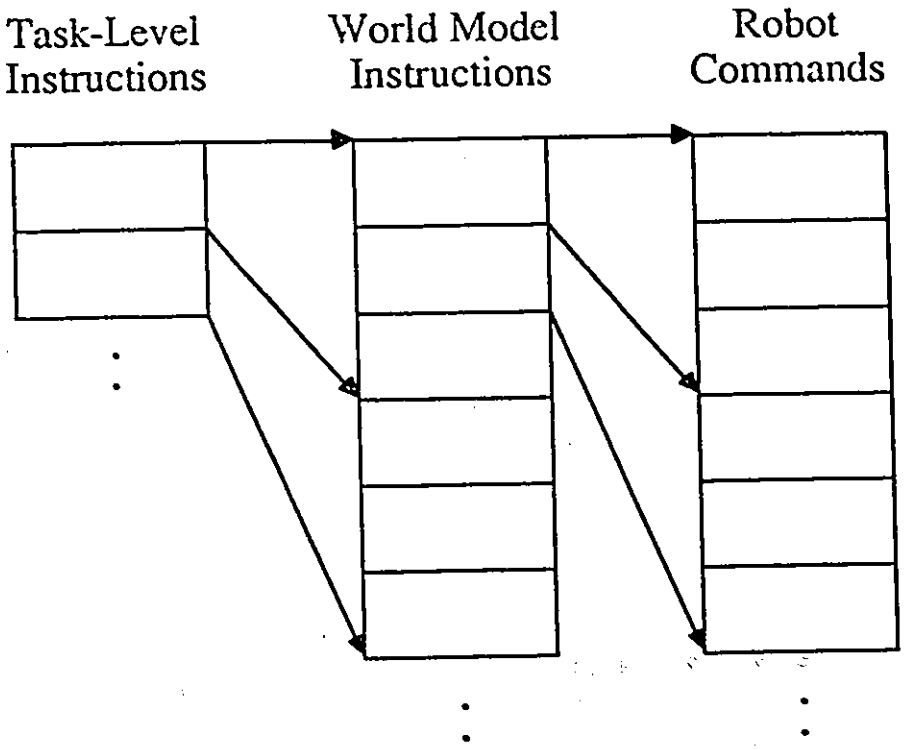

Figure 3.6: Typical Planner Output

execute to completion without any problems [37,44], while others recognized that unforseen situations could arise but did not propose any mechanisms to deal with them [42]. In all cases where errors were considered, it was assumed that the error correction procedures were embedded within the executable robot program, such as in [52]. This would allow the execution of the resultant robot program on the target robot system without any form of supervision. However, this assumes that errors can be predicted before hand and code generated to correct them. Clearly, this is not possible. Therefore, some form of on-line error recovery is necessary.

The term *error* is used here to designate a different real world state than the one expected. It must not be confused with situations that can be predicted such as partly or totally missing a hole, wedging, or jamming when inserting a peg in a hole. Since the magnitude of the uncertainties in the environment and the accuracy of the robot are known, the Planner can anticipate some problems and generate appropriate plans to avoid or recover from them. Such cases do not constitute "errors". However, not finding the hole in the region of space where it should have been constitutes an error. Such divergences between the real world and the World Model cannot be predicted. They occur whenever there is a mechanical failure of the robot, transport system or part feeders, or when a component was misplaced, is out of tolerance, or broken.

On-line error recovery implies three steps: (1) detection of the error, (2) acquisition of the error state, and (3) replanning to correct the error.

1. **Error detection**—One way to detect an error is to have the Planner generate test procedures to verify that intermediate subgoals, and the final target goal, are reached during execution. More specifically, the Planner embeds within the assembly program test procedures that verify the state of selected sensors and compare their readings to expected values. Sensing the environment obviously has some cost attached to it. For instance, using vision to verify the location of a component can be time consuming. The planner should therefore attempt to keep the tests simple. One way to achieve this is to integrate testing in the planning activity by treating tests as a constraint, i.e. a given subgoal should be testable, and the *cost* of that test minimized.

   Another way is to have a mechanism to deal with situations that require

immediate attention, such as collisions or the presence of a person in the work envelope of the robot. Whenever such a situation occurs, work in the cell is stopped. This is best accomplished by using an interrupt mechanism. Which sensors are active at a given time is determined by statements in the planner generated robot program.

Certain types of errors may be detected with either method, while other types are best detected using one or the other. For example, the loss of a component held in the gripper may be detected immediately by an interrupt type mechanism or later when inspecting the result of the assembly operation, while the breakdown of the robot can only be signaled by an interrupt mechanism.

2. **State Acquisition**—Whenever an error is detected, it is certain that the World Model no longer corresponds to the state of the real world. Consequently, it is necessary to build a representation of the real world in the World Model. This is the function of the *State Acquisition Module* or *SAM*. The SAM uses the cell's sensors and the description of *existing* parts provided by the World Model kept and updated by the Execution Monitor to provide input to image-recognition and scene-understanding procedures. These procedures attempt to identify objects and report their locations. The SAM can then update the World Model to reflect the state of the real world.

3. **Replanning**—If the SAM succeeds in building a representation of the error state, the Planner is invoked on-line to generate a new partial plan to replace the one that failed. This new partial plan must take the system from the current error state to the current desired state reflected in the current TLI. Since the TLI's are passed to the Execution Monitor with the assembly plan, there is no problem recuperating the desired state corresponding to the partial plan that failed. Planning is a time consuming task, so the amount of replanning should be minimized by using as much of the old plan as possible. The Planner attempts to generate a plan from the current error state to the closest established subgoal in the previously generated partial plan. If a plan can be generated successfully to this subgoal, program execution can resume from this point. If not, then planning should be attempted to the subsequent

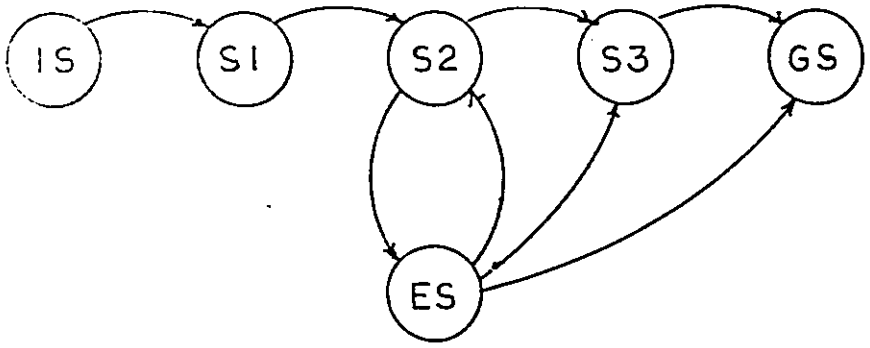

Figure 3.7: Replanning uses as much of existing partial plan as possible

subgoal, and so on until the closest subgoal is the target goal. This is illustrated in figure 3.7 in which the system goes from subgoal S_2 into the error state ES. The system attempts to replan back to S_2. If this fails, replanning will be attempted to S_3, and finally to the goal state GS.

Simply calling an operator to remove the cause of the error so that program execution can resume is not practical. Not only is time wasted waiting for the operator, but he or she must identify the error from messages issued by the system and reconstruct *exactly* the state the system should have been in. After all, the Planner took care in determining a grasp configuration consistent with the remainder of the plan. In some cases, it may not be possible to reconstruct the expected state, and in most cases, it would be difficult. For this reason, replanning either from a controlled state created by an operator or from the current state acquired by the SAM is the only solution.

Error recovery implies a mechanism to evaluate the chance of success of replanning. Obviously, if the error state comes from a broken down robot, no amount of replanning can cure the situation. In the case of a dropped bolt which cannot be found, the system must decide whether to proceed with a new bolt, or call for an operator. The lost bolt may have fallen somewhere where it will cause further problems. It is not obvious how an *Evaluator* should be written. Clearly, it would need knowledge about geometry and physics, as well as rules to evaluate the danger or chances of success in proceeding with a new plan.

During normal operations, the Execution Monitor feeds the plan primitives to

the cell controller which in turn interprets them in the robot's native language. It also monitors the execution of these primitives. Whenever a command terminates successfully, the Execution Monitor updates its internal copy of the World Model according to the instructions given by the Planner. However, if an error is detected, the Execution Monitor halts the execution of the program and calls upon the State Acquisition Module or *SAM* to build a representation of the work cell state in the World Model. If the SAM succeeds in acquiring enough information and the Evaluator decides replanning may succeed, the Execution Monitor calls upon the Planner to generate a new partial plan to replace the one that failed. Once the error is corrected, control is returned to the first partial plan primitive corresponding to the *next* TLI. On the other hand, if the Task Planner is incapable of generating a plan around the error, or if the State Acquisition module cannot acquire the current state, or the acquired state cannot be distinguished from the desired state, then a human operator is called. The operator can change the current state so that replanning may be retried, hopefully with success.

Even in cases where the SAM may not be able to determine completely the current error state, the Evaluator may decide that replanning might be successful. Say that a nut has slipped from the robot's gripper and that it has rolled away out of the field of vision of a camera. Even if it is not known where the nut went, the current position of the robot and the fact that the gripper is empty is known; a partial plan could therefore be generated that would fetch a new nut a reattempt the intended operation.

3.4 Implementation of the system

LISP is proposed as the programming language for the entire system for the following reasons:

1. **Need for symbolic processing**—Planning requires extensive manipulations of symbolic information, such as geometric constraints, preconditions for actions, etc. LISP, PROLOG and SNOBOL are currently the only widespread languages that support symbolic processing, with SNOBOL being oriented towards string processing, making it very difficult to use for the stated purpose.

2. **Need for numeric processing**—The individual planning module must perform extensive numeric processing. This rules out the use of PROLOG which is not well suited for extensive numeric computation.

3. **Need for complex user-defined data-structures**—LISP allows the programmer to create and manipulate complex data-structures with little effort, and does this better than PROLOG.

4. **Need for modularity**—Breaking a complex problem down into smaller problems is a well established programming methodology. This methodology can be best applied using LISP rather than PROLOG. In fact, the latest version of LISP, COMMON LISP, provides the programmer with Ada-like packages. These packages encapsulate functions and their associated data, isolating internal details from the programmer. Furthermore, the latest COMMON LISP standard supports object-oriented programming, providing a high degree of data-abstraction.

In fact, of all the programming languages around today, LISP is the one that provides the greatest flexibility: one simply *creates the required language* in LISP to solve a particular problem by defining the appropriate data structures and their associated operation, as well as the appropriate control structures.

3.5 Conclusion

This chapter discussed the implementation of a task-level programmed robotic assembly system. While much literature has been written over the last 15 years on developing planners and planning algorithms, no one has described a complete system in detail.

The chapter defined an extensible task-level language for assembly that is in line with current industrial practice. A system made up of a World Model (discussed in detail in chapters 4 and 5), and a Planner (outlined briefly in chapter 6), was described to convert a task-level program into an executable robot program. It was also argued that on-line replanning is both necessary and unavoidable. Such an on-line error recovery mechanism, ignored in all previous proposals, was proposed.

It was also shown that the planning problem is simplified by the use of specific task-level statements and by performing only gross motion, fine motion and grasp planning, thus constraining the problem domain more than ATLAS. It was also argued that the use of predefined goal templates over predefined code templates increases the power of the Planner over that of LAMA and AUTOPASS.

The only element of the system not discussed in the thesis is the State Acquisition Module. While much research has been done in the area of image processing and object recognition, there exist too little material and knowledge about scene understanding to pretend that a workable system could be devised. Furthermore, there is a need for further research in the development of an evaluator of the chances of success of replanning. However, by restricting error recovery to relatively simple cases, a workable State Acquisition Module could be designed.

Chapter 4

World Model

4.1 Introduction

This chapter and chapter 5 propose a World Model intended to be used in a task-level programmed robotic assembly system test-bed. It must therefore provide enough information that any planning algorithm may be implemented in the Planner.

The World Model holds information about the geometry of parts, their physical properties, and their interrelationships. Since real objects are not perfect but differ slightly in shape and size from the ideal case, the World Model must also contain information about the possible deviations of an object from its ideal or *nominal* representation. The magnitude of these deviations constitutes *variational data*, more commonly known as *tolerances*. Tolerance information plays a fundamental role in fine motion planning but can often be ignored in grasp planning and gross motion planning. In order to provide the individual planners with only the data they require, the World Model maintains the separation of nominal and variational information.

The author believes that this is the first attempt at selecting a representation scheme for objects based on the specific requirements of automatic programming for robotic assembly. While much research has been done in many aspects of robotics in the last 10 to 15 years, the problem of representing objects for robotic assembly

has been mostly ignored. Consequently, this chapter goes into much detail, beginning with some generalities, definitions, and fundamental theory pertaining to the representation of solids. Following this, various existing representation schemes are outlined along with their strength and weaknesses. Then the use of Constructive Solid Geometry/Boundary-Representation—CSG/B-rep—for the representation of nominal objects is justified based on the specific requirements of robotic assembly and automatic planning. After this, Variational Graphs are introduced as a means of representing tolerancing information. The implementation of the World Model is discussed in chapter 5.

The chapter ignores the problem of providing a user interface to enter and display World Model information, except where it influences the selection of a representation scheme. While such facilities are necessary in a complete system for robotic assembly, they add nothing to the present study nor to the demonstration of the feasibility of the proposed approach.

4.2 The Representation of Nominal Objects

4.2.1 Some General Definitions

Mathematical models of real solids, sometimes called abstract solids, are really only approximations of reality. Computer models of real objects are further approximations of the mathematical models. With this in mind, a representation scheme can be defined formally as a relation

$$s : M \mapsto R$$

where M is a *mathematical modeling space* whose elements are abstract solids which are mapped into the *representation space* R of the scheme. This is illustrated in figure 4.1. The *domain* D of s is the set of elements of M that can be represented by the scheme s. Similarly, the *range* V is the set of images of elements of D. The elements of V constitute *valid* representations, i.e representations that are syntactically and semantically correct.

A *syntactically correct* representation in this context is a finite combination of symbol structures—representing either primitive or compound surfaces or solids and

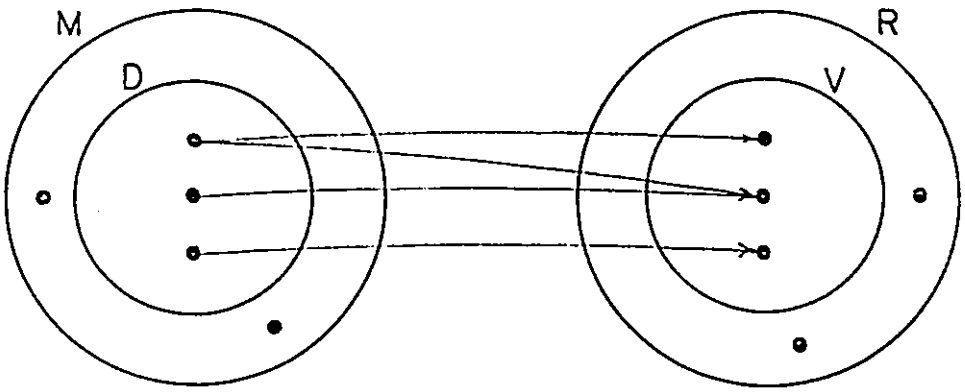

Figure 4.1: Representation scheme

operations—constructed according to some syntactic rules. The set of all syntactically correct representations constitutes the representation space R of a scheme.

A *semantically correct* representation is one that represents a meaningful geometric entity. Thus the semantics of a representation scheme are defined by associating geometric entities with syntactically correct representations.

Figure 4.1 shows that in general a representation scheme s does not allow the representation of all possible real-world objects, and that it allows the representation of some impossible or semantically incorrect ones. For instance, in some schemes, it is possible to "represent" disconnected solids—two or more non-touching solids in a fixed spatial arrangement—which have no physical interpretation. In general, a representation scheme is not a function: an element of D may have several images, and an element of V may be the image of more than one element of D.

The domain of a representation scheme determines the *descriptive power* of that scheme, that is, the set of objects representable in that scheme. Generally, the larger the domain, the better the scheme. Similarly, a representation scheme's range V should ideally correspond to its representation space R. In such a case, it is impossible to represent objects that have no possible physical interpretation. Unfortunately, this is rarely the case. Traditionally, insuring the validity of representations has been the responsibility of the users. Clearly, in an assembly environment where the planner creates objects, validity must be insured automatically, at least for those objects created by the system; the validity of initial objects can still be

the responsibility of those who define them.

A representation scheme is *complete* or *unambiguous* if every element of the range V is the image of a *single element* of the domain D, i.e. the inverse relation s^{-1} is a function.

A representation scheme is *unique* if to every element of the domain D corresponds a single element of the range V, i.e the relation s is a function. While uniqueness is not necessary in a representation scheme, it is often necessary to have procedures that can recognize when two different representations stand for the same object.

Schemes that are both complete and unique are bijective or one-to-one mappings. In such schemes, to each object corresponds a single representation, and vice-versa. Recognizing that two representations stand for the same object merely entails checking the syntax of the representative structures. Geometric representation schemes are rarely unique because substructures in a given object may often be permuted or enumerated in differing order.

Several desirable characteristics of representation scheme can be identified. One, *conciseness*, refers to the "size" of a representation. Size may be measured objectively as the amount of memory a given scheme requires to represent classes of objects, or subjectively as the amount of information that must be supplied to define an object of a particular class, defining a certain "verbosity".

Another is the *ease of creation* of valid representations. In general, concise representations are easier to create than verbose ones.

One more is *efficiency*, which refers to the computational load of the various operations carried out on a representation. Efficiency is extremely difficult to quantify as it depends on many factors such as the operation being carried out, the implementation language of the system, and the computer being used.

4.2.2 Modeling Solids

All representation schemes model real world solids by using abstract solid entities which are subsets of three-dimensional Euclidean space R^3. However, few models of R^3 are adequate to represent physical solids. To be useful, a model should capture the following properties of solids:

1. **Rigidity**—A solid's shape is independent of its position and orientation.

2. **Homogeneous three dimensionality**—A solid cannot have 2- or 1-dimensional components such as isolated or dangling faces or edges, as illustrated in figure 4.2(a).

3. **Finiteness**—Every solid occupies a finite portion of space.

4. **Closure under motion and certain boolean operations**—Moving a solid around (translation and rotation) or operations to add or remove material (boolean union/difference operation) must invariably produce other solids. The regular set intersection of two solids may not produce a solid.

Furthermore, the model must possess the following characteristics:

5. **Finite describability**—The representation scheme must be able to represent an object with a finite number of features, e.g. finite number of faces or edges, to be usable in a computer.

6. **Boundary determinism**—The representation scheme must provide solid boundaries that unambiguously determine the space enclosed by the solid, and therefore, what comprises the solid.

Requicha has shown that suitable models for solids are *r-sets* or *regular sets*, subsets of R^3 that are bounded, closed, regular and semianalytic [55]. Exact definitions of these terms appear in Appendix A. Informally, a bounded set representing a solid is one that occupies a finite portion of space. A closed regular set contains its boundary but does not contain dangling edges or surfaces, contrary to figure 4.2(a). A semianalytic set is well behaved, unlike the one shown in figure 4.2(b) where the top surface oscillates infinitely fast as it approaches the left face.

The problem with regular sets is that they are not closed under the conventional set operations, such as intersection, union and difference. This means that the intersection of two regular sets may not be a regular set. For instance, if two sets representing blocks are placed such that the surface of one is in contact with the

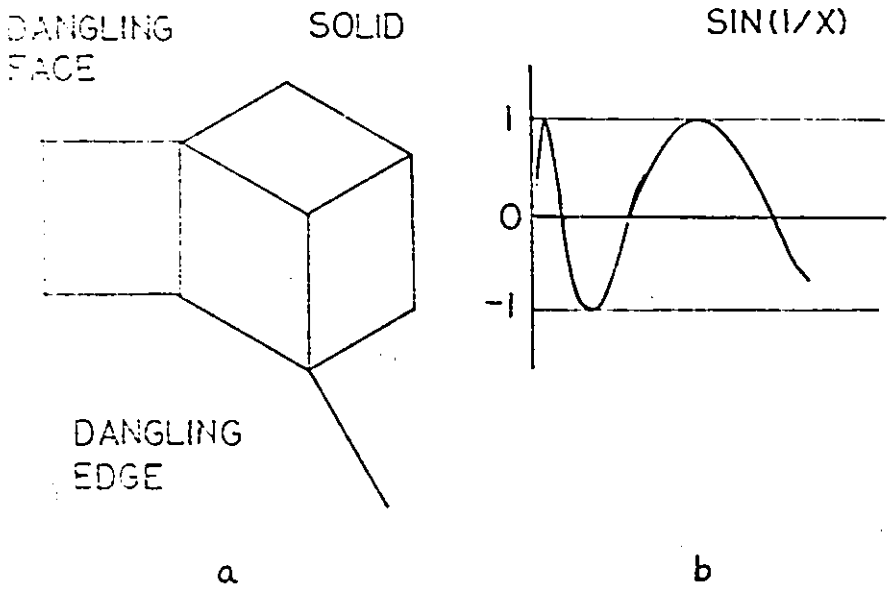

Figure 4.2: Subsets of R^3 that are not r-sets
(a) Nonregular set. Object has two- and one-dimensional components that do not enclose the solid's interior. (b) Nonsemianalytic set. Profile of a surface that oscillates infinitely fast as it approaches the origin.

surface of the other, then their intersection would be a surface, not a volume. To preserve regularity, the *regularized set operators* $\cap^*, \cup^*, -^*$, and c^* are defined as

$$X \cap^* Y = r(X \cap Y)$$

$$X \cup^* Y = r(X \cup Y)$$

$$X -^* Y = r(X - Y)$$

$$c^* X = r(cX)$$

where r denotes the regularization operation defined formally in definition 8 of Appendix A and c denotes the usual set complement. Informally, the regularization of a set is obtained by first taking the interior of a set and putting a boundary on it. This eliminates any dangling faces, edges or vertices created by the boolean operation.

Regular sets need not be connected. It is possible to model a single object which is in fact composed of multiple disjoint objects with fixed spatial relationships. It will thus be necessary to verify connectivity when verifying the validity of a representation.

4.2.3 Existing Representation Schemes

This section looks at the various existing representation schemes for solids. Schemes for modeling solids can be divided into three general classes [3]:

1. volume, which can be further subdivided into
   (a) pure primitive instancing,
   (b) spatial occupancy enumeration,
   (c) cell decomposition, and
   (d) constructive solid geometry (CSG).
2. sweep, and
3. boundary.

This section briefly looks at each of these schemes.

1. **Pure Primitive Instancing Scheme**

   In this scheme, objects are instances of a *generic primitive solid.* For instance, a particular cylinder is defined as a tuple of the form (CYLINDER, H, R), where CYLINDER indicates the *family* of the object, in this case a cylinder, and H and R are the height and radius of the cylinder, thus defining a particular instance of a cylinder.

   Pure instancing is distinct from other primitive instancing schemes such as Constructive Solid Geometry in that it does not allow the various instantiated primitives to be combined to make more complex solids.

   This scheme is unambiguous, unique, concise, easy to validate and easy to use. However, its extremely small domain makes it of very limited use.

2. **Spatial Occupancy Enumeration**

   Spatial occupancy enumeration schemes can be divided into four main categories:

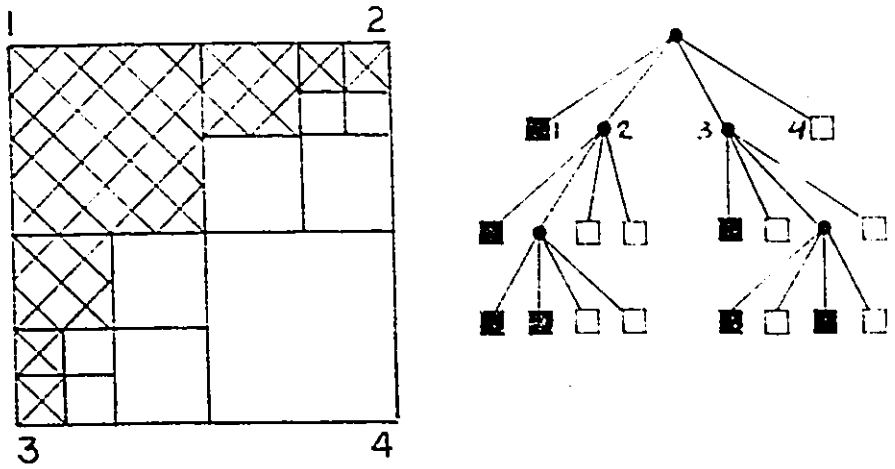

Figure 4.3: Quadtree representation scheme
Two-dimensional object and its corresponding quadtree

(a) Voxel representation scheme

This scheme is akin to the representation of pictures using pixels. Here, solids are modeled by sets of *voxels* or *volume elements*, small cubes of fixed size lying in a fixed spatial grid.

Voxel representation is unambiguous, unique and easy to validate, but quite verbose, unless only coarse approximations of solids are desired, in which case the voxels can be larger, thus reducing the number required to define a volume.

(b) Octree representation schemes

In this scheme, volumes are decomposed into cubes of different sizes, and these cubes organized into a tree structure with the largest cube as the root node and the smallest one as the leaf nodes. Each node of the tree represents a cube that points to eight other nodes describing the volume occupancy of the corresponding octant subcubes of the branching node cube. The two-dimensional equivalent, quadtrees, is shown in figure 4.3.

This representation is more compact than the voxel representation scheme and provides successive levels of precision of volume occupancy.

(c) **Cell decomposition**

In tetrahedral cell decomposition, volumes are decomposed into disjoint tetrahedral elements of varying sizes that meet exactly at surfaces, edges or vertices, leaving no "empty" spaces.

General cell decomposition is a generalization of tetrahedral cell decomposition. To reduce the number of cells required to describe an object, cells may vary in size and shape, having any number of sides.

These schemes are unambiguous but nonunique. Furthermore, checking for validity is computationally intensive, and cell decomposition of curved solids difficult to create.

(d) **Hyperpatch representation**

Here, volumes are decomposed into hyperpatches or tricubic solids defined by the equation [14]:

$$\mathbf{Z}(r,s,t) = \sum_{i=1}^{4}\sum_{j=1}^{4}\sum_{k=1}^{4} \mathbf{A}_{ijk} r^{4-i} s^{4-j} t^{4-k}$$

which maps solids from a parametric space defined by the parameters r, s, t into the range—the representation space—defined by the normal x, y, z axes. The degree of the mapping is a cubic in each variable, thus yielding an equation with 64 A_{ijk} vector coefficients, or 192 scalars (the A_{ijk} coefficients are vectors of order 3). The 6 surfaces of the represented solid are described by the set of bicubic surface patches obtained by setting each of the parameters to 0 or 1 while the remaining other two parameters are allowed to vary from 0 to 1. Similarly, the 12 edges of the solid are described by cubic curves obtained by setting two parameters to 0 or 1 while the remaining third parameter varies from 0 to 1, and the 8 vertices of the solid are obtained by setting each parameter to 0 or 1. Thus a hyperpatch equation maps the unit cube from the parametric space into a distorted one in the representation space.

Hyperpatches are currently the most powerful tool for the specification of sculptured solids, being able to associate internal density variations to a solid where all other schemes assume uniform interiors. Also, the

specification of a hyperpatch is both unambiguous and unique. However, to represent even simple shapes, several hyperpatches must be "glued" together, where gluing represents the union of mutually disjoint hyperpatches. Since each hyperpatch requires 192 scalars, the computational load increases quite dramatically with object complexity. Since most objects commonly used in assembly are made up of components of uniform density, the complexity of the hyperpatch representation often outweighs its advantages.

One area of success of hyperpatches has been in the design of CAD systems that can automatically perform finite element analysis. By fixing the value of two parameters, the finite elements can be quickly delineated by a mesh of curves.

3. Constructive Solid Geometry

Constructive Solid Geometry schemes are a family of schemes that represent solids as Boolean combinations of primitive solid components using the regularized set operators $\cap^*$, $\cup^*$ and $-^*$ defined before.

CSG representations are ordered binary trees in which nonterminal nodes represent translations, rotations or generalized set operators, and terminal nodes represent either primitive instances of solids or rotation and translation arguments. This can be summarized using BNF notation as:

<CSG tree> ::= <primitive leaf>
| <CSG tree> <regularized set operator> <CSG tree>
| <CSG tree> <motion operator> <motion arguments>

<regularized set operator> ::= $\cap^*$
| $\cup^*$
| $-^*$

<motion operator> ::= translation
| rotation

<primitive leaf> ::= (<name>, <object class>, <object parameters>)

There exist two main CSG schemes: the first uses primitives that are bounded and is called "CSG based on bounded primitives" or simply "CSG" when there

is no danger of confusion; the second scheme uses unbounded primitives and is called "CSG based on general half-spaces".

(a) **CSG based on bounded primitives**

Here, the primitive solids are r-sets. As long as the primitive solids are valid, the properties of r-sets guarantee that a syntactically correct CSG tree represents a valid r-set, assuming of course that the regularized set operators are used. Remember also that r-sets allow disconnected objects, that is, an object whose components are in a fixed spatial relationship but need not be in contact. Thus, checking the validity of a representation merely entails checking its syntax and verifying the position of the various components.

Furthermore, ***generic objects*** can be represented by CSG trees with ***uninstantiated*** primitives. These objects are also called procedure-, macro-, or parametric-objects, or object schemata. Their validity as r-sets can easily be determined.

(b) **CSG based on general half-spaces**

In this scheme, the primitives are unbounded half-spaces, i.e. regions of space lying on one side of a plane or curved surface. CSG trees can therefore represent invalid r-sets, and invalid objects, if the various surfaces do not completely enclose a space and define its boundary. Testing for validity is computationally intensive.

This scheme resembles the hybrid CSG/B-rep scheme, except that the hybrid scheme uses ***bounded*** surfaces rather than unbounded ones. Hybrid schemes are discussed further on.

Constructive Solid Geometry schemes are non-unique but unambiguous. They are quite compact and allow conceptually easy computations of ***integral properties*** such as mass, center of gravity, and rotational inertia. Integral properties derive their name from their defining function

$$I = \int_S f(p)\, dV$$

where I is the integral property in question, S the solid of interest, $f(p)$ a real valued function over the points $p = (x, y, z)$ comprising the solid S, and dV the volume differential. According to the function $f(p)$, the various properties computed are:

$f(p) = \rho(x, y, z)$	I: mass M
$f(p) = x\rho(x, y, z)/M$	I: x-coordinate of the center of mass x_G
$f(p) = (x^2 + y^2)\rho(x, y, z)$	I: moment of inertia about the z axis

where $\rho(x, y, z)$ is the material density at (x, y, z). The integral properties of a compound object are obtained by applying the following formulas recursively

$$\int_{A \cup^* B} f\,dV = \int_A f\,dV + \int_B f\,dV - \int_{A \cap^* B} f\,dV$$

$$\int_{A -^* B} f\,dV = \int_A f\,dV - \int_{A \cap^* B} f\,dV$$

The evaluation of the "primitive" integrals can be done directly or using approximation techniques.

CSG also permit users to easily define *unsculptured* objects, i.e. objects with simple surfaces. However, they are extremely cumbersome to use for *sculptured objects*, i.e. those with complex curves such as a human face.

4. Sweep Representations

The basic idea in sweep representations is that a set moving through space sweeps out a volume that may be represented by the moving set and its trajectory. Sweep representations come in three broad categories.

(a) Translational and rotational sweeps

In *translational sweeping*, a *fixed* two-dimensional set is swept along some trajectory to define a volume. Representing that volume merely entails representing the 2-D set along with the associated trajectory.

In *rotational sweeping*, a *fixed* two-dimensional set is *rotated* about some axis to sweep out a volume. Translational and rotational sweeping are illustrated in figure 4.4.

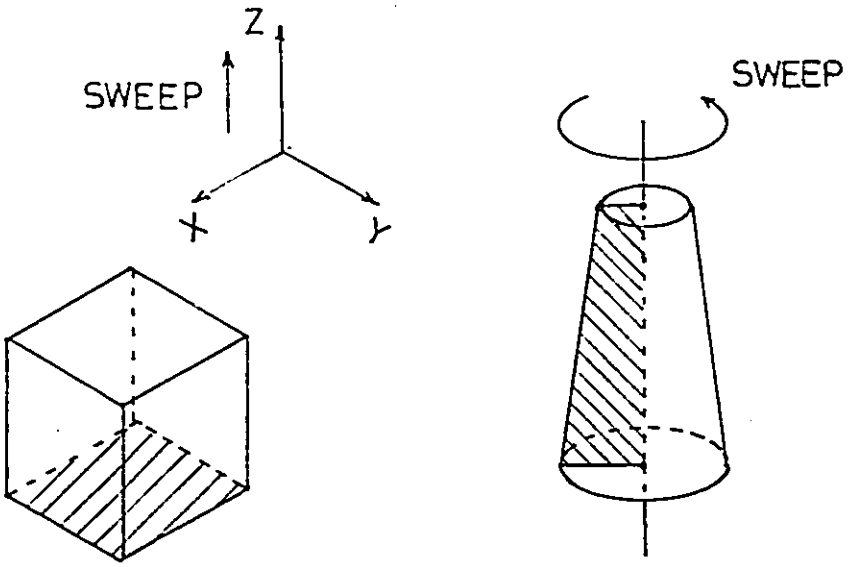

Figure 4.4: Translational and rotational sweeping

Translational and rotational sweeps are unambiguous but not unique. While also quite compact, their domain is limited to those solids having translational or rotational symmetry.

(b) **Solid sweeps**

In this scheme, solids are modeled by moving another solid through space. Such schemes are used in machining, especially in material removal applications, such as milling. However, algorithms to compute properties of the represented solids are lacking.

(c) **Generalized sweeps or generalized cylinders**

The most powerful and least understood sweep representation scheme, the method of generalized cylinders was introduced by T.O. Binford in the early seventies for image recognition. Unlike the translational sweep scheme, the swept two-dimensional set is allowed to change *continuously* as it moves along the trajectory, as illustrated in figure 4.5.

This scheme is unambiguous but not unique. It is well adapted to model objects which are composed of one or more cylinder-like components, such as aircrafts or animals, with the addition of union and difference operators. Such a scheme, generally considered to be a hybrid CSG/generalized sweep, is being used by R.A. Brooks in ACRONYM, an image recognition system [5].

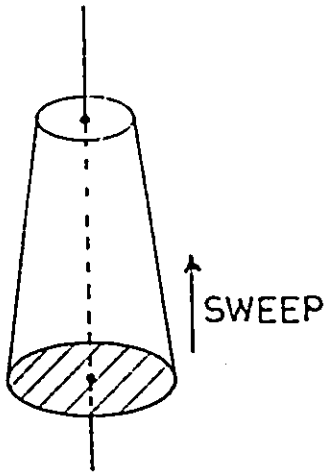

Figure 4.5: General sweeping

5. **Boundary Representation**

In B-rep, as it is often called, a solid is modeled by segmenting its boundary or surface into a finite number of *bounded* 2-D subsets called *faces*. In turn, each face can be represented by its bounding edges and vertices. The resultant symbol structure is a graph depicting the relationship between the object, its faces, its edges and its vertices, as illustrated in figure 4.6.

While not unique, B-rep schemes are unambiguous provided that surfaces are modeled unambiguously. However, when curved surfaces must be modeled, edges are no longer sufficient to represent them unambiguously: surfaces must be modeled explicitly. Provided such a representation is available, B-rep schemes are amongst the most powerful along with Constructive Solid Geometry, but a lot more verbose.

Several approaches exist for describing surfaces in algebraic terms.

(a) **Polygonal Approximation**—Perhaps the most widespread method of modeling is the approximation of a solid by a polyhedron, a multifaceted solid whose every face is a planar polygon. All polygonal approximation schemes model the polyhedron as a set of polygons which are themselves modeled as sets of straight edges modeled in turn simply by their end

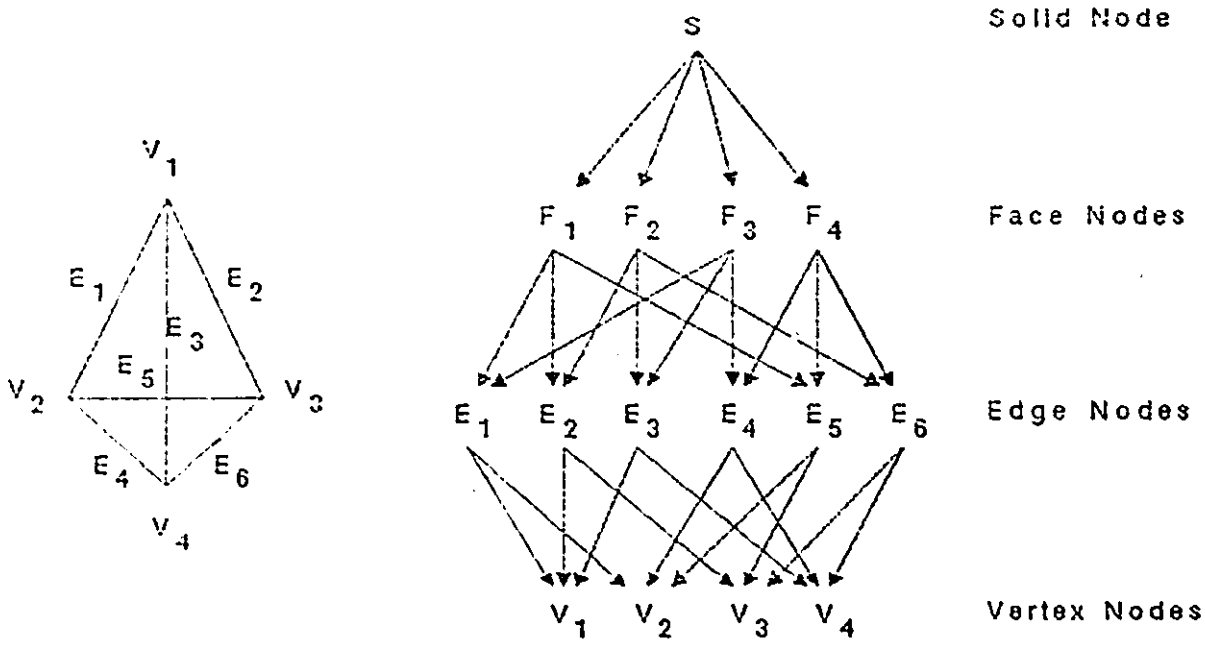

Figure 4.6: Boundary representation of a tetrahedron

points. Thus to describe a polyhedron, one only needs to store the vertices of the solid in such a way that edge and face descriptions may be retrieved. There exist several variations on polyhedral approximation differing only in the way that vertices are stored so that edge and face information may be retrieved. Polygonal approximation is discussed in many introductory texts on computer graphics, and in [49] and [24] in particular.

Polygonal approximation methods have the advantage of utilizing planes to represent surfaces and straight lines to represent edges, both of which are described by linear equations. Their extensive use in computer graphics was motivated by the fact that the problem of removing hidden lines is transformed into the resolution of a set of linear equations. The "roughness" of the polygonal approximation can often be tolerated in graphic applications, and if not, smoothing algorithms can be applied to obtain apparently continuous surfaces. However, modeling for CAD often

requires closer approximations than for image rendering. Polygonal approximation has the disadvantage that the amount of information required to model curved objects grows very rapidly as the approximation is refined by using successively smaller polygons. Furthermore, it does not preserve the significant topological features of an object, making its use in planning difficult. While Turner [69] has proposed a method for calculating the exact representation of the boundary of an object from in polyhedral approximation, and many continue utilizing polygonal approximations [62] in solid modeling, alternate boundary representation schemes have been devised.

(b) **Use of Quadric Surfaces**—A quadric surface is a surface that is represented in Cartesian coordinates by the following general equation:

$$Ax^2 + By^2 + Cz^2 + 2Fxy + 2Gyz + 2Hxz + 2Px + 2Qy + 2Rz + D = 0$$

Special cases of the general quadric equation are the sphere, spheroid, ellipsoid, hyperboloid, paraboloid, cone, cylinder, and plane. A great advantage of quadrics is that they can be represented in matrix form by the equation

$$P^T Q P = 0$$

where

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \text{ and } Q = \begin{bmatrix} A & F & H & P \\ F & B & G & Q \\ H & G & C & R \\ P & Q & R & D \end{bmatrix}$$

By multiplying the matrix that represents the quadric in its master coordinate system with a homogeneous coordinate transform matrix H, it is possible to obtain a description of the quadric in any configuration, i.e.

$$Q' = (H^{-1})^T Q H^{-1}$$

The intersection of two quadrics defines a planar curve, such as a circle, ellipse, or straight line, or a non-planar curve described by a polynomial of degree 4. These curves are easily expressed as functions of one parameter.

While quadrics offer much more expressive power than polygons, they by no means offer a complete set. One very common shape that cannot be modeled exactly by the quadric equation is the torus. However, the simplicity and elegance of quadrics makes them ideal in cases where they offer sufficient expressive power and unlike polygons, they preserve the essential topology of a surface. Quadrics are used in GMSolid, a solid modeler designed by General Motors that maintains a dual representation of each object: a CSG one and a B-rep one [4].

(c) **Use of Bicubic Surface Patches**—A bicubic surface patch describes a surface with the use of three equations, one for each of the coordinates x, y, and z, in terms of two parameters, s and t. Each equation is a cubic in both s and t, hence the term bicubic. The general form of a bicubic is

$$\begin{aligned} x(s,t) = {} & a_{11}s^3t^3 + a_{12}s^3t^2 + a_{13}s^3t + a_{14}s^3 + \\ & a_{21}s^2t^3 + a_{22}s^2t^2 + a_{23}s^2t + a_{24}s^2 + \\ & a_{31}st^3 + a_{32}st^2 + a_{33}st + a_{34}s + \\ & a_{41}t^3 + a_{42}t^2 + a_{43}t + a_{44} \end{aligned} \tag{4.1}$$

and similarly for $y(s,t)$ and $z(s,t)$. The equation can be rewritten in matrix form as

$$\mathbf{Z}(s,t) = \sum_{i=1}^{4}\sum_{j=1}^{4} \mathbf{A}_{ij}s^{4-i}t^{4-j}$$

from which it can be seen that bicubics are a restriction of hyperpatches. As both parameters are allowed to vary from 0 to 1, all points on the surface are generated.

The four "edges" of the patch are obtained by fixing each parameter to 0 or 1 while varying the other from 0 to 1. The resultant curves are cubics of one parameter.

Bicubic surface patches are used in the automotive and aeronautic industries to model free form shapes such as wings and car bodies. The parameters of the bicubic equations are derived by fitting curves through existing points using Hermite, B-splines or Bezier techniques. The resultant bicubics are then called the Hermite form, B-spline form, or Bezier

form of the bicubic [24,14,13]. Surface patches are used in PATRAN [13] and Geomap-III [34].

Hybrid schemes combine one or more representation schemes in an attempt to obtain the advantages of each individual scheme without their disadvantages. Typically, these schemes combine the simplicity and compactness of CSG for the overall description of an object with a different scheme to represent the primitive solids, usually generalized sweeps or boundary representation. Boundary representation allows easy description of an object's surface, while the spines of a generalized sweep offer some information for object recognition.

Wire-frames are another type of representation scheme that represents solids by their edges. They were used extensively in early computerized drafting tools and are still in widespread use today.

However, wire-frame are ambiguous, and are therefore unsuitable for automated processing. In fact, they are being replaced by other modeling techniques that employ surfaces or volumes as primitives rather than edges in all new CAD systems. Ambiguities in wire-frame representations can be removed by associating edges to faces, but this results in a de facto B-rep scheme in which surfaces are defined by their bounding edges.

4.2.4 Selection of a Representation Scheme

This section justifies the selection of a CSG/B-rep scheme for representing solids in the World Model based on the requirements imposed by robotic assembly.

Assembly is the Union of Parts

During assembly, new parts are created by mating or *gluing* solids together, where gluing is the union of mutually disjoint sets [28]. In assembly, parts can never occupy the same space, so mating parts really is a gluing operation. Gluing simplifies the computation of the integral properties of the newly created object since the intersection of the various solid components need not be evaluated.

Some form of Constructive Solid Geometry is strongly suggested as the most

appropriate scheme to use since it captures the essential nature of assembly. Furthermore, the computation of the integral properties of assembled objects is trivial since gluing is used.

The Need for Surface Information

Information about the shape, position and orientation of surfaces, as well as information about a surface's finish—to evaluate friction—is required if the Planner is to generate compliant motions and the State Acquisition Module to recognize objects.

Object recognition is essentially the matching of sensed data to some model of what should have been sensed. Sensory input is usually provided by a vision system, a range sensing system, or a tactile sensing system. Most of the research performed over the last 20 years in object recognition has centered on the use of digitized gray-scale intensity images as sensory data. Such images are easily obtained using conventional black and white video cameras and consist essentially of arrays of numbers encoding the brightness of points on a regularly spaced grid. Such images convey very little direct information about the full 3-dimensional shape of objects. Various techniques for extracting such information from intensity images are mentioned in [2] such as shape from binocular stereo [25], shape from motion [30,70], shape from shading [29], shape from photometric stereo [16,78], shape from texture [77], and shape from contour [32].

More recently, with the development of better ranging sensors, the use of range data has increased in popularity. The extraction of shape information from range data is conceptually simpler than from intensity data since the "pixels" of a range image encode actual depth information. The acquired depth information depends only on the positions and configurations of the various objects in a scene relative to the range sensor, and on the particular characteristics of the sensor.

Tactile sensing has also been used [17,43]. Some research groups have used tactile arrays that provide encoded surface contour information, somewhat like range image depth maps. These arrays can locate surface features such as holes, bumps, edges, and vertices. Other groups have used the angles at which a gripper's fingers come to rest on an object to provide shape information.

Regardless of their particularities, all techniques sample surfaces. Since the

World Model must provide a model of what should have been sensed, it must describe surfaces. In addition, the modeling scheme used must provide sufficient information to permit the display of parts for human verification and for visual simulation of assembly operations. What are displayed are surfaces, edges, and vertices rather than full volumes.

Surface information can be obtained from many representation schemes, such as CSG based on bounded primitives, hyperpatches or sweep representations, but the algorithms are not always trivial and often quite computationally intensive. To avoid unnecessary computations, either a B-rep or CSG scheme based on half-spaces is the natural choice [2]. B-rep has the advantage over CSG based on half-spaces of using bounded surfaces, making the evaluation of those boundaries—surface intersections—unnecessary and thus reducing the computational load even more.

Selected Scheme

The above discussion strongly points to a CSG/B-rep hybrid scheme as the optimum scheme for representing objects in the World Model. In order to provide the widest possible domain, B-rep is used to allow the representation of sculptured and unsculptured objects, and to provide surface information about all objects existing before assembly begins. Since assembly is essentially a gluing operation, CSG is selected for its simplicity and compactness to represent objects created during assembly operations.

It is assumed that the World Model is created either from a dialogue with a part designer through an intelligent user interface, or automatically from some CAD database. This implies that the selected representation scheme is not imposed on the part designer. Rather, conversion from whatever scheme parts were designed in into CSG/B-rep is accomplished. This also implies that the World Model does not have to compute the integral properties of objects present before planning begins, nor does it have to insure the validity of those initial objects. It only needs to compute the integral properties and check the validity of objects created during planning, something trivial when the gluing operation is used.

The particularities of modeling the robot are discussed further on.

4.3 Variational Data

The previous section dealt with the representation of nominal objects. This section proposes the use of *Variational Graphs* or *VGraphs* to represent tolerancing information. Tolerances on objects in the World Model play a crucial role in determining the amount of uncertainty in the position, orientation, and size of components and in selecting a sequence of fine motions that assures the success of mating operations.

4.3.1 Geometric Tolerancing

A tolerance specification defines a region of space called a *tolerance zone* within which an object's features are constrained to lie. Traditionally, tolerances were indicated as $\pm$ deviations from nominal dimensions or as the lower and upper limit on dimensions. This notation is still in use but can be ambiguous and incomplete. Figure 4.7(a) illustrates a simple rectangular plate with a circular hole cut in it. The conventional $\pm$ tolerance on the width of the plate may be interpreted as: (b) constraining the position of the edge of the plate which is parallel to edge B; (c) as expressing the required amount of parallelism of the edge relative to edge B; (d) or as expressing a tolerance on the "flatness" of the sides.

To avoid such ambiguities, geometric tolerancing has been developed. Standardized in ANSI Y14.5, geometric tolerancing is a rich notation that provides unambiguous [36] interpretations of position and shape tolerance information. ANSI Y14.5, outlined in Appendix B, defines numerous types of geometric tolerances which are divided into three main categories: position, form, and runout. With the addition of size tolerances, ANSI Y14.5 provides for the complete description of a part's form and size.

Currently, very few solid modelers provide facilities for representing and manipulating tolerance information. Requicha reports the existence of only three projects attempting to integrate tolerancing information into CAD and CAM: one sponsored by CAM-I, Computer-Aided Manufacturing International, a non-profit industrial consortium headquartered in Arlington, Texas; another sponsored by the U.S. Air Force under the I-CAM project, and his own efforts at the University of Rochester under the Production Automation Project [57]. The CAM-I project, DMIS, the

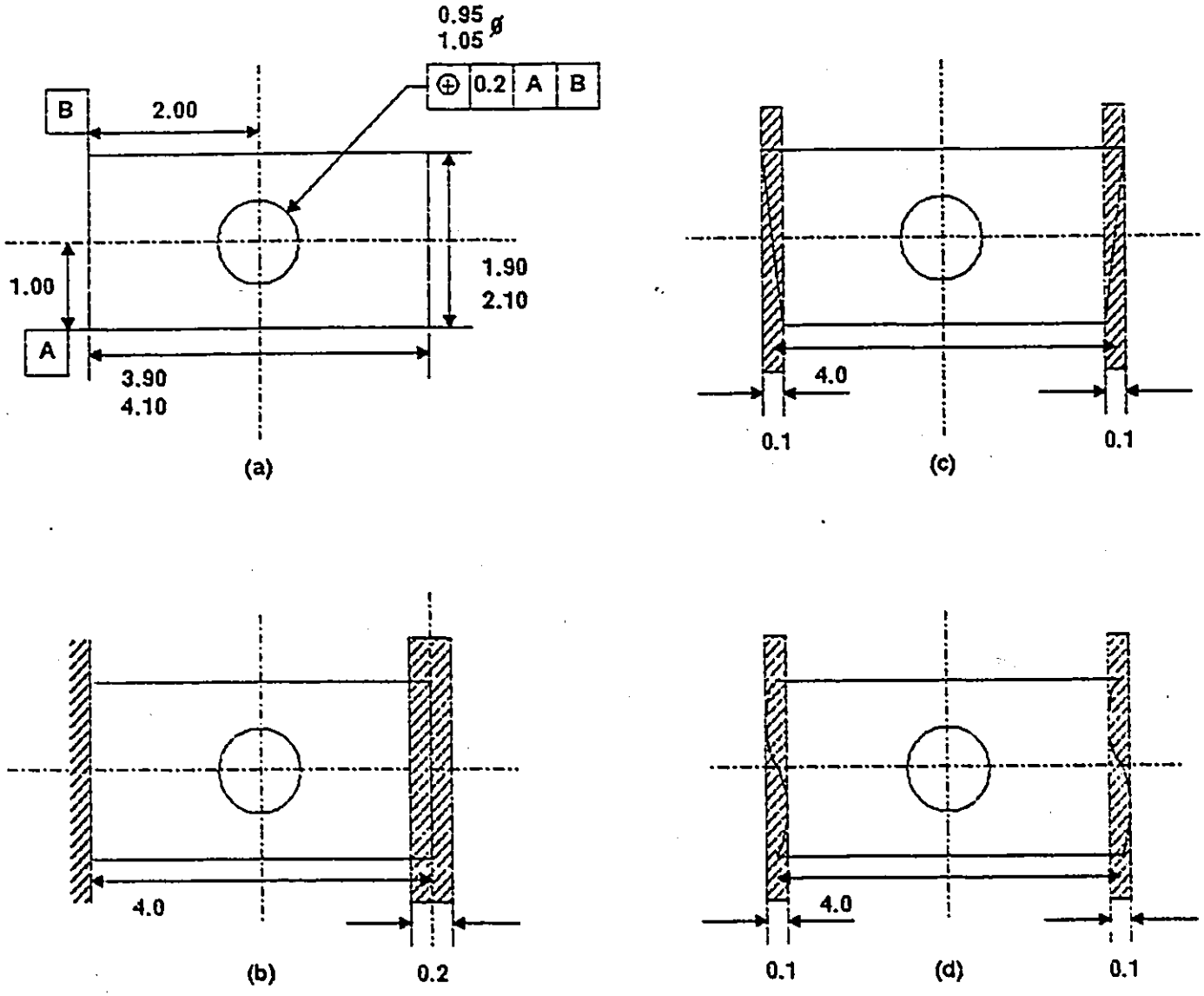

Figure 4.7: Conventional tolerancing practice
(a) plate showing dimensions and tolerances
(b) tolerance on the width of the plate
(c) tolerance on the form of the plate
(d) tolerance on the form of the edges

Dimensional Measuring Interface Specification, is concerned with the specification rather than the representation of tolerance information. DMIS is a tentative standard for a "high-level language interface between quality assurance equipment and CAD systems" that permits "bi-directional data transfer between CAD systems and any computerized dimensional inspection equipment or data analysis system" [9]. Requicha's project concentrates on using tolerance information in automatic production planning and in automatic code generation for machine tools.

The Planner for robotic assembly does not require the full set of ANSI Y14.5 geometric tolerances. These detailed tolerances do not describe different tolerance zones. Flatness, cylindricity and profile of a surface all describe the same thing, the profile of a surface, but add information about the intended shape of that surface. Geometric tolerances also describe which quality control process should be used to verify that parts are within specified tolerances. The assembly Planner only requires tolerancing information about the size, position and orientation of parts and features. Consequently, it uses generalizations of the ANSI Y14.5 geometric tolerances, as explained in the next section.

4.3.2 Interpretation of Tolerances

Tolerance information is associated with *nominal surface features* or NSF's. Features, for short, are two-dimensional subsets F_i of a solid's S boundary, *bd(S)*, that satisfy the following equation:

$$\cup F_i = \mathrm{bd}(S)$$

A *simple* nominal feature is the lowest level entity and lies within a single primitive surface. A *composite* feature is the union of simple features. A simple nominal feature corresponds to an individual surface in the B-rep representation that models an *actual* or real feature G, while composite features correspond to unions of such surfaces. A *tolerance specification*, a collection of geometric assertions A_{ij}, can be associated to each feature F_i.

Tolerance zones are constructed using *offsetting operations* [59,56]. Offsetting is based on the notions of maximum material condition (MMC) and least material condition (LMC) used in conventional tolerancing practice [36]. The MMC corresponds to the use of the maximum amount of material for a part while the LMC

corresponds to the use of the least amount. To define these concepts formally one needs to define the notion of distance of a point p from a solid S which is given by

$$d(p, S) = \min\{d(p, q), q \in S\}$$

where $d(p, q)$ is the ordinary three-dimensional Euclidean space distance. When S is represented as a r-set, q obviously lies on the boundary of the solid. Now let D_p be a *positive* real number and define the ***positive single offset solid*** $O(D_p; S)$ as

$$O(D_p; S) = \{p : d(p, S) \leq D_p\}$$

Similarly, let D_n be a *positive* real number and define the ***negative single offset solid*** $O(-D_n; S)$ as

$$O(-D_n; S) = S -^* O(D_n; c^* S)$$

where $-^*$ and c^* are the regularized difference and complement operators. The positive offset solid corresponds to the MMC case while the negative offset solid corresponds to the LMC case:

$$\text{MMC}(D_p; S) = O(D_p; S)$$

$$\text{LMC}(D_n; S) = O(-D_n; S)$$

Offset solids are built from other solids while tolerance information applies to features which are surfaces, and in some cases to lines and curves. In order to build offset solids from a feature F, it is necessary to introduce the notion of the *extended feature* H of a feature F. An extended feature H associated to a nominal surface feature F is a possibly unbounded half-space defined as the Boolean composition (union and intersection) of half-spaces H_i and satisfying the following conditions:

1. $F \subset \text{bd}(H)$,
2. H does not contain in its definition halfspaces H_i that contribute two-dimensional subsets to $\text{bd}(H)$ but not to F, and
3. the "material" sides of H correspond to the material sides of the solid S bounded by the features F_i,

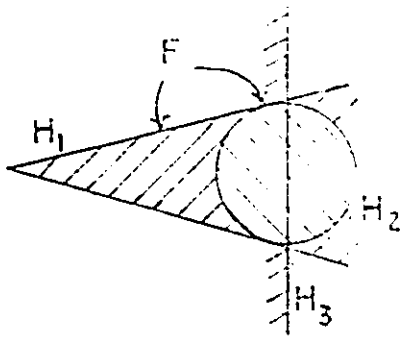

Figure 4.8: Extended features of an object
H_3 does not contribute to F nor to bd(H) where $H = (H_1 \cap H_3) \cup H_2$), but it cannot be omitted from the expression of H.

An extended feature may contain half-spaces that contribute nothing to bd(H) nor to F, as shown in figure 4.8 which shows a slice through a 3-D cone and a 3-D sphere.

With this in mind, a tolerance zone $T(D_p, -D_n; F)$ on a two-dimensional feature F can be constructed as

$$O(D_p; H) -^* O(-D_n; H)$$

and interpreted as generating a solid shell extending into the "material" from the nominal surface by a thickness D_n and out of the material by a value D_p. Note that such a definition is consistent with the definitions of MMC and LMC conditions: in the case of a solid cylinder, D_p describes the difference between the nominal and *maximum* radii of the cylinder, while in the case of a cylindrical hole, D_p describes the difference between the nominal and *minimum* radii of the hole.

Variational Graphs

The function of the Variational Graph, a bi-directional graph, is to represent intrinsic and extrinsic tolerance information about faces and edges and to reference the extrinsic tolerances to the appropriate datums. The logical structure of a VGraph is explained below and illustrated in figure 4.9, using the nomenclature introduced by Requicha in [57] as much as possible.

The basic entities in a VGraph are Nominal Face nodes or NFaces for short.

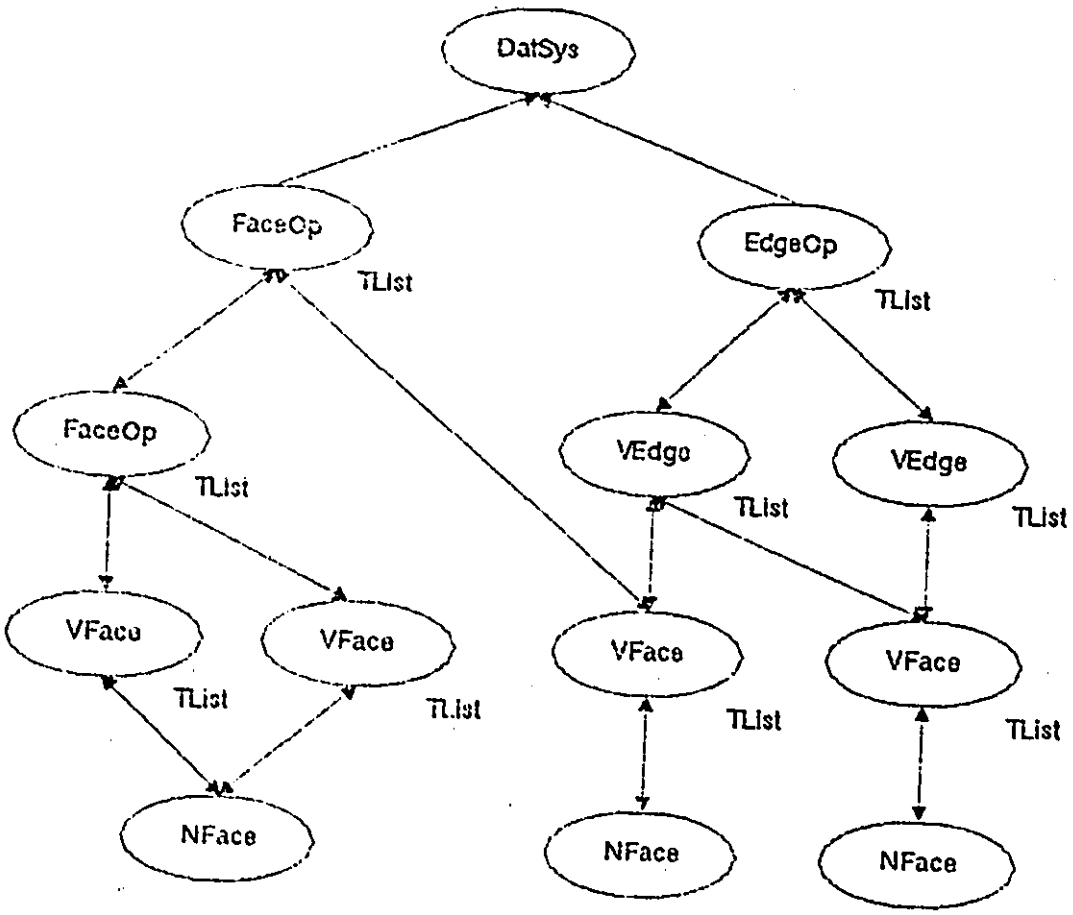

Figure 4.9: Logical structure of a VGraph

NFaces correspond to the nominal faces of an object, i.e. the surfaces represented in the B-rep scheme. NFaces are the only "interface" between the nominal model graph and the VGraph.

Each NFace is associated to one or more Virtual Faces or VFaces to which surface tolerance information is attached. In some rare cases, tolerances must be attached to a feature smaller than a nominal feature. For example, a surface of an object may be loosely toleranced, but have a tightly toleranced area where a part must be mated. Such cases are handled by splitting a single NFace node into several VFaces and associating the tolerance information to the individual VFaces while preserving, in the nominal object model, the fact that there exists only one face in reality. However, in most cases, a single VFace would be associated to each NFace.

Several VFaces can be combined into Face Operation nodes or FaceOps. A FaceOp represents the union of VFaces or other FaceOp nodes "lower" in the graph. VFaces and FaceOps constitute Surface Feature nodes or SFeats. The Surface Feature graph defines a hierarchy of surface features, such as hole patterns, or even patterns of hole patterns, toleranced collectively.

Each VFace is also related to Virtual Edges or VEdges to which curve tolerance information is associated. The VEdges describe the edges of the VFaces. Whenever a VFace corresponds directly to an NFace, its VEdges correspond directly to the edges of the NFace in the nominal B-rep representation.

Several VEdges can be combined into Edge Operation nodes or EdgeOps. VEdges and EdgeOps constitute Curve Features or CFeats.

The topmost nodes in the VGraph are the Datum System nodes or DatSyses. These nodes represent ideal planes, lines or points of nominal objects used as datums. They form a hierarchy pointing back to the root node which is the universal reference datum.

SFeats—individual VFaces and their unions—and CFeats—individual VEdges and their unions—are associated to a Tolerance List or TList. The TList contains a number of tolerances and their values and describes the tolerances on the feature. The types and structure of tolerances are illustrated in figure 4.10 and explained below:

Tolerance type	Structure
Size	$(D_{p1}D_{n1}D_{p2}D_{n2}\ldots)$
Intrinsic Surface Form	T_f
Extrinsic Surface Form	T_f, ↑Reference Datum
Intrinsic Curve Form	T_f, ↑Guide Plane
Extrinsic Curve Form	T_f, ↑Guide Plane, ↑Reference Datum
Position	T_p, ↑Reference Datum

Figure 4.10: Structure of the Tolerance List

- **Size**

  Size tolerances are not geometric tolerances as defined in ANSI Y14.5. According to Requicha, an actual feature G with corresponding nominal feature F and extended feature H satisfies a size tolerance $T_s(D_p, D_n; F)$ if and only if there is a congruent instance $H' = R(H)$, where R is a rigid motion, such that

  $$G \subset \mathrm{MMC}(D_p; H') -^* \mathrm{LMC}(D_n; H')$$

  This means that the feature G must lie within a tolerance zone of width $D_p + D_n$. The position and orientation of this zone are unspecified and can be selected so that G fits in the zone whenever G's dimensions are within tolerance.

  Whenever F is a composite feature, a pair D_p, D_n can be associated to each F_i. The resultant tolerance zone is then of variable thickness.

  This definition is quite general and can be applied to any nominal feature, simple or composite. However, it departs from current practice, except when describing the size tolerances on the radii of spheres, cylinders, and other such objects.

  For instance, when applied to composite features, the resultant tolerance zones constrain the position of individual features. To specify size tolerances on a planar feature G, say a face of a parallelepiped, the tolerance on the distance between two opposite sides of the solid adjacent to the feature being toleranced is specified. This is possible since F, a composite feature, can be taken as

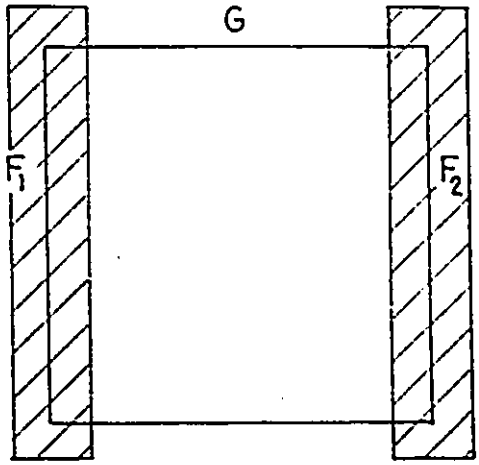

Figure 4.11: Size tolerancing
Orthographic view of parallelepiped

the union of two opposite faces adjacent to G, F_1 and F_2, and offset solids constructed for $H_1 \cup H_2$, the associated extended feature (see figure 4.11). The tolerance zone thus obtained contains F_1 and F_2 whenever the feature G adjacent to F_1 and F_2 satisfies its size tolerance. This in fact converts the size tolerance on G into position tolerances on all its adjacent faces.

When applied directly to a planar and certain non-planar nominal surface features, the tolerance zone really defines a form tolerance.

- **Surface Form**

  A surface form tolerance T_f is a positive real number that defines a tolerance zone

  $$O(T_f/2; F) -^* O(-T_f/2; F)$$

  within which the surface must lie.

  An intrinsic form tolerance is generally a tighter tolerance than the size tolerance on the "finish" of a surface, constraining the size of bumps and pits. It encompasses flatness and cylindricity.

An extrinsic form tolerance constrains the surface to lie in a given relationship to a datum feature or measured entity. It encompasses perpendicularity, angularity, parallelism, profile of a surface, and total runout.

- **Curve Form**

  A intrinsic curve form tolerance consists of a positive real number T_f that defines a tolerance zone

  $$O(T_f/2; F) -^* O(-T_f/2; F)$$

  within which a curve, defined by the intersection of a guide plane with the surface being toleranced, must lie. It encompasses straightness and cylindricity.

  An extrinsic curve form attribute is similar, except that the guide plane is perpendicular to the reference datum. It encompasses circular runout.

- **Position and Orientation**

  Based on previous definitions, one is tempted to define a position tolerance T_p as

  $$O(T_p/2; F) -^* O(-T_p/2; F)$$

  where F is the nominal feature correctly positioned and oriented relative to the reference datum system. However, this definition ignores the size variations of the feature which obviously affects where a feature lies.

  Two cases have to be isolated to derive proper definitions of position tolerances: MMC position tolerances and RFS position tolerances. (See Appendix B for a definition of MMC and RFS position tolerances).

  In the MMC case, the value of T_p is the minimum error in position, which occurs at MMC. An MMC position tolerance then defines a tolerance zone of the form

  $$O(T_p/2 + D_p; F) -^* O(-T_p/2 - 2D_n; F)$$

  where $O(T_p/2+D_p; F)$ is the shell created "outward" at MMC and $O(-T_p/2+2D_n)$ is the shell created "inward" at LMC where the position uncertainty has increased by a value D_n.

In the case of an RFS position tolerance, the position tolerance is independent of any feature size, so that the correct definition is

$$O(T_p/2 + D_p; F) -^{*} O(-Tp/2 - D_n; F)$$

All tolerance zones are built around ideal features positioned correctly relative to the reference datums.

4.4 Representation of Friction

Friction plays a crucial role in robotic assembly as previously discussed in chapter 2. Consequently, a few words are in order concerning the determination of friction forces between two surfaces.

Friction forces are broadly divided into two categories: *static* friction which acts on two surfaces at rest with respect to each other; and *kinetic* friction which acts on two surfaces in relative motion. Friction follows two basic laws:

1. the friction force between two given objects is, as a first approximation, independent of the contact area, and
2. it is proportional to the normal force.

These laws are expressed in the well known equations

$$f_s \leq \mu_s N$$

and

$$f_k = \mu_k N$$

where f_s is the static friction force and f_k is the dynamic friction force. Both forces act tangent to the contact surfaces. The static friction force exists only in response to an external force acting tangent to the contact surfaces: whenever that force exceeds $\mu_s N$, the object begins to move and the kinetic friction appears. μ_s and μ_k are the coefficients of friction, scalar "constants" that satisfy in most cases the relationship $\mu_k < \mu_s < 1$.

Friction is a function of the material of both surfaces, their surface finish, temperature, and surface films or contamination. For these reasons, there is no exact theory of dry friction, only empirical laws. Consequently, friction cannot be handled in an analytic way: the only solution is to store maximum friction coefficients in a matrix indexed by material and surface finish of *both* surfaces.

This also implies that surface material and finish must be stored in the World Model. Using B-rep allows the storage of these attributes on a per surface basis.

4.5 Conclusion

The problem of modeling real world objects for planning has been mostly ignored in the literature, researchers in robotics preferring to concentrate their efforts on more immediate problems such as path planning, robotic vision, object recognition, and robot manipulator design and control. Since a World Model is a crucial component of any system for automatic programming for robotic assembly, the chapter has explored the topic in depth.

It was stated that separation of the nominal and variational data was desirable to preserve information about the essential topology of an object, and to avoid bogging down planning modules with variational data they do not need.

It was argued that a hybrid B-rep/CSG was preferable over the use of pure boundary representations, pure Constructive Solid Geometry, hyperpatches, or sweep representations. Such a hybrid scheme offers the compactness and simplicity of CSG and avoids the computation of boundaries when creating new parts, while maintaining explicit surface information which is necessary for planning compliant motions and for object recognition. It may be that future developments in hardware and computing techniques will make another scheme more attractive, such as pure CSG or hyperpatches.

The use of Variational Graph to model tolerance information was also described, along with the relationship between the nominal model and the variational model. People have just begun to look at formal models of tolerancing. Traditionally, the interpretation of tolerance information has required good judgement and a lot of experience. Even so, inherent ambiguities in tolerance information has led to the development of the ANSI Y14.5 geometric tolerancing standard. However, ANSI

Y14.5 is still meant to be interpreted by humans and contains much redundant information. Requicha's work represents a first attempt at formalizing tolerancing theory for use in computer models. Much more work is needed in the area of automated reasoning about tolerance information.

The modeling of friction has also been discussed briefly. Since there exists no exact theory of friction, friction coefficients must be stored explicitly for each 4-tuple consisting of surface material and finish.

The modeling of free space has not been discussed in this chapter. This topic is discussed in chapter 6, as the particular model used for free space is directly tied to the algorithms used to plan paths through it.

Chapter 5

Implementation of the World Model

5.1 Introduction

This chapter describes the structure and implementation of the World Model, and demonstrates the feasibility of using a CSG/B-rep scheme for representing objects. The demonstration, in section 5.4, consists of the creation of the tower object of figure 3.5, represented using CSG , from its two constituent objects, the base and top, represented using B-rep. The definition of the top cylinder of the tower is used in the text to illustrate the data structures that represent solids, faces, and edges. The complete definition of the two objects appears in Appendix C, along with the LISP code to create and manipulate these objects.

5.2 Overview

The World Model is essentially a collection of binary rooted tree structures, each representing some independent (un-connected) object in the assembly cell. For instance, before assembly begins, the World Model might have a binary rooted tree representing the robot, another tree representing the fixture, as well as other graphs representing objects that have no physical connection to the robot or the fixtures (with the exception of the floor, which may not need to be represented...)

Each node in the binary trees represents some *type* of solid—either a *primitive object*, a *compound object*, a *mechanism*, or an *aggregate*—and each arc in the tree is labelled with the type of mechanical link between the two "lower" nodes connected together- –*rigid attachment*, *non-rigid attachment*, or *conditional attachment*. The relationship between the type of link and the type of nodes is explained below:

1. **Rigid attachment**—As its name suggests, the rigid attachment relationship designates two or more objects joined together in a fixed spatial arrangement. Such a relationship exists whenever two objects are mated together in such a way that moving one component object causes the other rigidly attached object to move, under all normal conditions. This relationship is represented by the gluing operation. Objects held together in a rigid attachment relationship form a *compound object*. All sub-components of a compound object must be either compound or primitive objects. A primitive object is defined using a B-rep scheme.

2. **Non-rigid attachment**—A non-rigid attachment relationship designates two objects joined together by a sliding or rotating joint. Objects held together in a non-rigid relationship form a *mechanism*. Components of a mechanism can be other mechanisms, compound objects and simple objects. The shape of a mechanism is dependent on the position and orientation of its components in space, on the types of link joining the components together, and on the overall motion of the mechanism. For instance, a mechanism consisting of two rods connected by a pin will change its shape under the influence of gravity as the robot moves it around. Because of the inherent complexity of dealing with mechanisms, they are not discussed any further. The robot manipulator constitutes a mechanism that differs from other mechanism in that its motion is computer controlled and the influences of gravity on its positioning ability is, if not negligible, minimal. Consequently, it is possible to deal with this type of mechanism without too much of a problem. However, because it is under program control, the robot is best modeled by a separate module, either in the Planner or in the World Model.

3. **Conditional attachment**—A conditional attachment exists whenever two objects are mated in such a way that moving one component object may or may not cause the other objects to move, depending on their spatial relationship. A typical illustration of conditional attachment is the stacking of two components. Moving the top component will not move the bottom component, while moving the bottom component will move the top component. Furthermore the top component remains on the bottom component only for a given angle of tilt of the latter. Beyond that angle, determined by the coefficient of friction between the surfaces in contact, the top object slips off. Objects held together in a conditional relationship form an *aggregate*. Components of an aggregate can be aggregates, mechanisms, compound objects and simple objects.

Binary trees offer the advantages of being uniform data structures and of maintaining information about the order of assembly operations. But more importantly, they reflect the binary nature of the mechanical relationship between objects and simplifies the overall World Model data structure. Without such a binary structure, arcs running among nodes at the same level in the tree would have to be introduced . However, some operations, such as object recognition, may require that a tree of depth 1 be constructed to provide input information to a pattern matcher. Such a tree has the object of interest as a root node at level 0 and the faces or edges comprising the object as leaves at level 1, and can be easily constructed from the binary tree by simply collecting all the leaf nodes.

5.3 Implementation of World Model

This section describes the various LISP data structures used to represent solids, faces, edges, and tolerance information. For each data structure representing a nominal entity, the corresponding definition of the tower's top component, a cylinder, is reproduced from Appendix C to clarify the discussion.

5.3.1 LISP Data Structure for Nominal Solids

Each node in the World Model consists of a named data object, a LISP symbol, whose value is a LISP structure with multiple fields. The name is either supplied by the user, in the initial World Model or in CREATE statements, or generated internally by the Planner whenever it automatically creates some sub-assembly. Whenever an object is created that has the same user given name as an existing object, both objects have their name converted by the Planner into a path name, according to the rules outlined in section 3.3.1. It is assumed in this thesis that the World Model was created beforehand and that there do not exist two objects with the same name which, at any rate, is impossible in LISP. The characteristics of an object are retrieved from the structure by appropriate procedures.

The fields of the structure representing an object are:

1. **Name**—The name of the object is repeated in this field. The Planner operates on structures stored in the symbol value slot and not on the symbol itself. Consequently, it has no way of knowing the name of objects unless this field is included. The name is either the one provided by the user, a name generated internally, or the path name created by the system.

2. **Type**—This field contains one of the following symbols defined before: primitive, compound, mechanism, or aggregate. The type is selected based on the type of link between the two objects being mated, and on their type field. For instance, rigidly attaching an object to a mechanism must still yield a mechanism.

3. **Geometric Class**—This field contains a list describing the geometric class of the object, such as cube, cylinder, pyramid, etc. This information, along with the parameters describing the object, can be used by the Planner to perform quick checks on the feasibility of an operation. For example, when inserting a cylindrical peg in a cylindrical hole, the Planner can quickly verify that the peg's diameter is less that the diameter of the hole. In a complete system, the geometric class of new objects would be declared before hand in a declaration section of the task-level program as discussed in section 3.2.6,

so that the field may be filled with meaningful information. At this time, the field of new objects is filled with the symbol *general.*

4. **Functional Class**—This field describes the intended purpose, if any, of the object. For instance, screws can be designated as being of the screw functional class. This allows the Planner to flag a logic error in the program quickly, such as attempting to rivet a screw. In a complete system, the functional class of new objects would also be derived from the declaration section of the program, as for the geometric class. Currently, new objects are of functional class "unknown".

5. **Configuration**—This field describes the position and orientation of an object in space. It is a 4×4 homogeneous coordinate transformation matrix. When moving an object about, the new position of any point of an object can be derived simply by multiplying the homogeneous coordinate transformation matrix describing the new configuration with the vector describing the point's previous position suitably expressed in Cartesian coordinates. It is thus not necessary every time an object is moved to recompute the coordinates of each of its features. Similarly, the analytic description of a quadric face can be be derived for any configuration, as explained on page 80.

   The configuration field of an object reflects the position of that object relative to the *world* coordinate system, the system of coordinates used to locate every object in the cell. While the world coordinate system can be made to correspond to that of the robot, usually some point in the robot's base, it may be preferable to set it somewhere else. A proper choice would be some location close to the fixtures. By placing and properly aligning a pin or other suitable object at the origin of the world coordinate system, the robot is able to measure the position and orientation of the world coordinate system relative to its own coordinate system using tactile feedback. In many cases, this may lead to greater accuracy of the robot since it will partially compensate for the deformations due to gravity in that region of space and for that particular robot configuration.

Whenever a new compound object is created, its configuration field is initialized to the identity configuration, i.e. no rotation and no translation. The position of that object relative to the world coordinates is determined by the configuration fields of its two constituent objects. Whenever the new solid is displaced, its configuration field is updated to reflect its new position while the configuration fields of the constituent objects are not modified. This avoids recomputing the position of every constituent element of an object while the Planner "moves" it "forward" while generating new steps, or "backward" while backtracking.

Homogeneous coordinate matrices were developed by J. Denavit and R.S. Hartenberg in 1955. Their use in 3-D modeling and robotics are described in [49] and [17] respectively, and in many other references.

6. **Mass, Center of Mass, and Rotational Inertia**—These three fields hold the mass of the object, a scalar, its center of mass, a 4×1 matrix indicating its Cartesian coordinate, and its rotational inertia, another 4×1 matrix indicating the moment of inertia about the x, y, and z axes.

   Whenever a new rigid or conditionally attached object is created, its integral properties must be computed. This is done in a straightforward manner by computing the integral properties of nominal objects only and ignoring tolerances. Offhand, it can be assumed that the variations in mass, center of mass, and rotational inertia among objects of a given type due to their variation in size and shape would be small enough to be negligible. This assumption can be defended on the following two bases:

   (a) the robot should not pick up objects that weigh more than its maximum load carrying capacity. The increase in mass of an object at MMC relative to its nominal value is assumed to be less than the margin of safety of the maximum weight carrying capacity of the robot. Consequently, tolerancing information can be neglected.

   (b) Generated compliant motions should not depend on the exact value of the integral properties of an object, but should accommodate some variations in the integral properties of the load. Calculating the range of values of

integral properties would be a waste of time and would further complicate the Planner's code.

The mass m_r of a new object is simply the sum of the mass of its constituent objects. Since new objects are formed from the combination of two existing objects, the equation is simply

$$m_r = \sum_{i=1}^{2} m_i = m_1 + m_2$$

Similarly, the center of mass $\mathbf{r}_G$, a vector of order 3, is given by

$$\mathbf{r}_G = \sum_{i=1}^{2} \mathbf{r}_{Gi} m_i / m_r = \frac{\mathbf{r}_{G1} m_1 + \mathbf{r}_{G2} m_2}{m_r}$$

where each $\mathbf{r}_{Gi}$ is obtained by multiplying the object's center of mass coordinate vector by the object's homogeneous coordinate transformation matrix describing its position in space. To obtain the actual location of the center of mass of the object, its $\mathbf{r}_G$ vector must be multiplied by the object's homogeneous coordinate transform matrix. It is important not to add the fourth component of the 4×1 vectors which is always 1 and is included only for compatibility when multiplying the vector with a 4×4 homogeneous coordinate transform matrix.

The equation for computing the rotational inertia of the resultant object is not so straightforward but is still easily evaluated by a computer. The computation begins with the evaluation of the rotational inertia of the individual component objects in their given rotation. Let x', y', z' be the new coordinate system and x'', y'', z'' the old coordinate system, and let $l_{p'q''}$ be the angle between the axis p' of x', y', z' and q'' of x'', y'', z''. Then

$$\begin{aligned} I_{p'p'} &= l^2_{p'x''} I_{x''x''} + l^2_{p'y''} I_{y''y''} + l^2_{p'z''} I_{z''z''} - \\ &\quad 2l_{p'x''} l_{p'y''} I_{x''y''} - 2l_{p'y''} l_{p'z''} I_{y''z''} - 2l_{p'x''} l_{p'z''} I_{x''z''} \end{aligned}$$

where $p' = x', y', or z'$. Following this, the rotational inertia for the translated component objects can be computed using the parallel axis theorem where x, y, z are the coordinates $\mathbf{r}_G$ of the center of mass of the resultant object

$$I_{xx_i} = I_{x'x'_i} + m_i((y' - y)^2 + (z' - z)^2)$$

$$I_{yy_i} = I_{y'y'_i} + m_i((x' - x)^2 + (z' - z)^2)$$

$$I_{zz_i} = I_{z'z'_i} + m_i((x' - x)^2 + (y' - y)^2)$$

Finally, the rotational inertia of the compound object is computed simply as

$$I_{xx} = \sum_{i=1}^{2} I_{xx_i} = I_{xx_1} + I_{xx_2}$$

$$I_{yy} = \sum_{i=1}^{2} I_{yy_i} = I_{yy_1} + I_{yy_2}$$

$$I_{zz} = \sum_{i=1}^{2} I_{zz_i} = I_{zz_1} + I_{zz_2}$$

7. **Part-of**—This field holds the name of the object to which the given object belongs. This is used yo create a doubly linked tree structure, thus allowing searches up and down a tree.

8. **Constituent Objects**—This field holds the name and type of the two objects that constitute the given object, as well as the type of mechanical relationship that holds between the two constituent objects.

9. **List of Faces**—In the case of primitive objects, the constituent object field is replaced by the following field which contains a list of all the faces that make up the primitive object. Since the top of the tower is a primitive, it has a list of faces field rather than a constituent objects field.

The top cylinder of the tower is thus defined by the statement shown in figure 5.1 where `top` is a symbol whose value is the structure created by the `make-primitive` function.

5.3.2 LISP Data Structures for Nominal Faces

The faces of an object also have to be described. Each node representing a primitive object points to all the nodes representing its faces.

A face node is also a LISP symbol whose value is a structure with the following fields:

1. **Name**—The face's name.

```
(setf top
      (make-primitive
       :name 'top
       :type 'primitive
       :geometric-class 'solid-cylinder
       :functional-class 'sub-component
       :configuration (make-array '(4 4) :initial-contents
                                  '((1.0 0.0 0.0 0.0)
                                    (0.0 1.0 0.0 0.0)
                                    (0.0 0.0 1.0 0.0)
                                    (0.0 0.0 0.0 1.0)))
       :mass 7.85
       :center-of-mass (make-array '(4 1) :initial-contents
                                   '((0.0) (0.0) (0.5) (1.0)))
       :moment-of-inertia (make-array '(4 1) :initial-contents
                                      '((1.15) (1.15) (0.98) (1.0)))
       :part-of nil
       :list-of-faces '(top.top top.bottom top.side)))
```

Figure 5.1: Definition of the top of the tower

2. **Type**—This field describes the geometry of the surface in symbolic terms. The tower is constituted from three types of surfaces: rectangles, disks, and cylinders. Again, this information can be used by the Planner to draw inferences.

3. **Analytic Description**—This field contains the description of the surface in analytic terms. The section on boundary representations discussed three ways of describing surfaces analytically: polygonal approximation, quadrics, and bicubics. While bicubic surface patches probably offer the most versatile boundary representation scheme, quadrics offer much greater simplicity and elegance, and better describe the essential topology of an object. For this reason, quadrics are used to demonstrate the feasibility of the proposed approach. The field thus contains a 4×4 matrix as described on page 80.

4. **In/Out Flag**—In order to determined whether a point is inside an object (in the material) or in free space, each surface has a flag associated to it that indicates whether its normal is pointing into material or into free space.

5. **Surface Type**—Used as an index to look up the coefficient of friction in a friction matrix. Friction is discussed in section 4.4.

6. **List of Edges**—Surfaces can be either "closed", "semi-closed" or "open". A closed surface, such as a sphere, has no bounding edges. An open surface, such as a square or a half-sphere, has bounding edges all around. A semi-closed surface, such as a cylinder, is closed in some directions but open in other directions. Since quadrics yield infinite open and semi-open surfaces, these surfaces must have edges attached to them to bound them. These edges are either straight lines, circles, ellipses, or non planar curves described by a parametric polynomial equation of degree 4.

7. **Part-of**—This field contains the name of the primitive solid that contains the surface. It is used to form a doubly linked graph.

8. **VFace**—This field holds the name of the VFace nodes that constitute the surface, usually a single one. It links the nominal information about objects to the variational graph that holds the tolerance information.

```
(setf top.top
      (make-disk
       :name 'top.top
       :type 'disk
       :radius 0.5
       :center-point (make-array '(4 1) :initial-contents
                                 '((0.0) (0.0) (1.0) (1.0)))
       :analytic-description (make-array '(4 4) :initial-contents
                                         '((0.0 0.0 0.0 0.0)
                                           (0.0 0.0 0.0 0.0)
                                           (0.0 0.0 0.0 0.5)
                                           (0.0 0.0 0.5 -1.0)))
       :in-out-flag 'out
       :surface-type nil
       :list-of-edges '(top.edge1)
       :part-of 'top
       :vface 'top.vtop))
```

Figure 5.2: Definition of the top surface of the tower's top

The definition of a typical surface, the top face of the tower's top component, `top.top`, is illustrated in figure 5.2.

Certain types of surfaces may have extra fields. For instance, the `top.top` surface, being a disk, has a center-point field and a radius field indicating both the position of the center of the disk and the radius of the disk. This information is not available from the analytic description of the surface which represents an infinite plane.

5.3.3 LISP Data Structures for Nominal Edges

Edges are described in geometric and parametric forms. The geometric description allows the Planner to determine the essential topology of an edge quickly, while the parametric description is used to generate points on the edge in terms of its parameter. An edge node is also a LISP symbol whose value is a structure with the following fields:

```
(setf top.edge2
      (make-circle
       :name 'top.edge2
       :type 'circle
       :radius 0.5
       :center-point (make-array '(4 1) :initial-contents
                                 '((0.0) (0.0) (0.0) (1.0)))
       :part-of '(top.bottom top.side)
       :parametric-description '(lambda (s)
                                 (make-array '(4 1) :initial-contents
                                             (list
                                              (list
                                               (times 0.5
                                                      (cos (times s two-pi))))
                                              (list
                                               (times 0.5
                                                      (sin (times s two-pi))))
                                              '(0)
                                              '(1))))))
```

Figure 5.3: Definition of the edge of the top surface of the top component of the tower

1. **Name**—The name of the edge.

2. **Type**—A symbol that describes the geometry of the edge. Edges can be either straight lines, circles, ellipses, or general, the latter designating a parametric polynomial description of degree 4.

3. **Parametric Description**—A function of one parameter that returns a 4×1 matrix representing a point on the edge as a function of the parameter's value, which should be in the range of 0 to 1.

4. **Part-of**—The name of the faces or faces the edge belongs to.

The definition of a typical edge, the edge bounding the top face of the top component of the tower, `top.edge2`, is shown in figure 5.3.

Certain types of edges may have extra fields that hold more information about the geometry of the curve. In the case of `top.edge2`, a circle, its center point and radius are also stored to define a particular instance of a circle. However, the orientation of the circle may only be derived from its parametric description.

5.3.4 LISP Data Structures for Variational Graphs

To build a variational graph, a few types of nodes, implemented as LISP structures, must be defined:

1. **VFaces**—A VFace node has several fields:
   (a) A name field that holds the name of the node.
   (b) A part-of field that holds the names of nominal faces nodes. These nodes contain the description of the VFace. This field just points back to the NFace nodes, forming double links.
   (c) A field that holds a list of the VFace's bounding VEdges.
   (d) A field that holds the name of the FaceOp node, if any, to which the VFace belongs.
   (e) A field that holds a list of tolerances.
2. **FaceOps**—FaceOp nodes are unions of VFaces. They contain the following fields:
   (a) A name field that holds the name of the node.
   (b) A part-of field that may hold the name of another FaceOp node. This allows for the hierarchy of FaceOps, and the hierarchy of patterns of features.
   (c) A field that holds the names of all the constituent VFaces.
   (d) A field that holds a tolerance list.
3. **VEdges**—A VEdge node represents part of the boundary of a VFace. It thus corresponds to a nominal edge node. Like VFace nodes, this node has a name

field holds the name of the nominal edge node that contains the description of the VEdge, a part-of field that points back to the VFace forming a double link, a field that holds the name of the EdgeOp node to which the VEdge belongs, if any, and a field that holds a list of tolerances.

4. **EdgeOps**—A EdgeOp node represents the union of VEdges. Like the FaceOp node, it contains a field holding its name, a field holding the list of constituent VEdges, a field holding the name of a superior EdgeOp node to form hierarchies of edge patterns, and a tolerance list.

Tolerances are unnamed LISP structures. Each one has a field holding the type of the tolerance, and other fields holding appropriate information, as described in figure 4.10. Of particular interest are the fields that point to datums.

A datum is simply the name of some nominal feature. For instance, when measuring positions relative to a face or set of faces, the reference datum field of the tolerance structure simply holds the name of the nodes describing the nominal face. When measuring positions relative to measured entities, the appropriate entities must exist as named nominal features, either planes, lines, or vertices.

5.4 Proof of Concept

This section seeks to demonstrate the feasibility of using a hybrid CSG/B-rep representation scheme. This is done by creating the tower object of figure 3.5, a compound CSG object, from its two primitive sub-components: the base, a cube, and the top, a cylinder.

The demonstration consists of an annotated transcript of a Franz LISP session. It begins by positioning the base to show that an object's description can be obtained for any configuration. To be as general as possible and yet remain simple, the base is first rotated about the x axis by 90 degrees, then rotated about the z axis by 45 degrees, and finally translated along the x and y axes by one unit. The initial and final position of the base is illustrated in figure 5.4. Following this, the cylinder is moved on top of the base. Finally, the tower object is created as a compound object, and its mass and center of mass computed. The demonstration

terminates with the description of the tower and all its constituent entities: the two primitives, their faces and edges, all in their new configuration.

The complete definitions of the base and top in their original position appear in Appendix C, along with the LISP code to manipulate the data structures. The reader can appreciate from that Appendix the large amount of information required to define the two primitives and the code to manipulate it. Adding tolerancing information would increase substantially the amount of information and code required, a further burden for the reader. Consequently, tolerancing information has not been included.

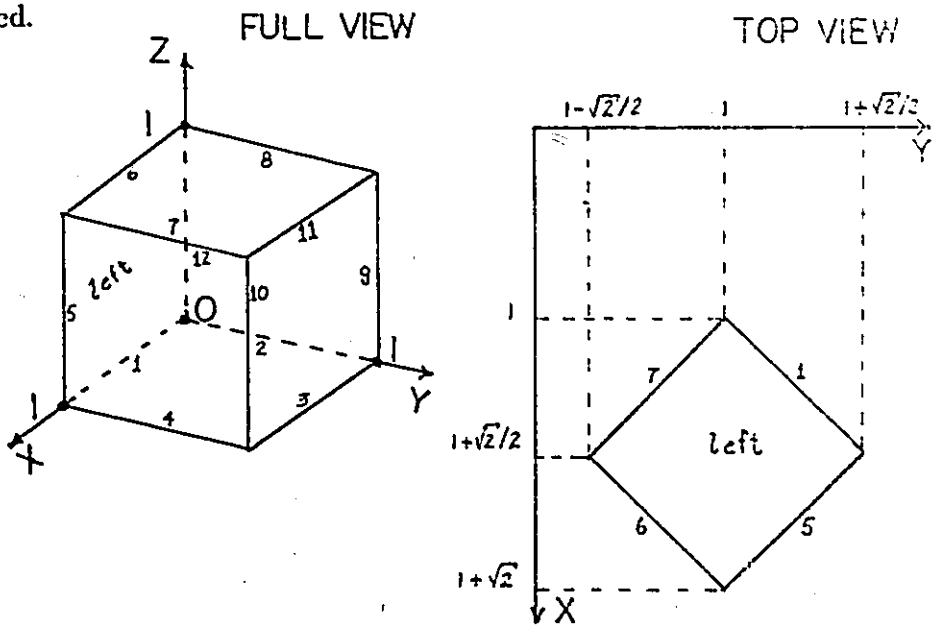

Figure 5.4: Motion of the base object

The transcript begins on the next page.

Annotated transcript of LISP session in which the tover object was created

All comments in the transcript are preceded by a semicolon.

```
; Once loaded, Franz LISP prints its prompt

Franz Lisp, Opus 38.79

; Now the compiled support code, stored in 'common3.o' is loaded using the fast
; load function

-> (fasl '~/lisp/common3)

; Franz responds with the full path name of the file, and with t, for true,
; once the file has been loaded successfully.

[fasl /usr/users/gauthier/lisp/common3]
t

; Now the source code containing the code and data for the World Model is
; loaded.

-> (load '~/lisp/wm)

; Franz again responds with the full path name of the file, and the full file
; name, once the file has been loaded and processed successfully. Once this
; is done, the two primitive objects with their faces and edges have been
; defined.

[load /usr/users/gauthier/lisp/wm.l]
t

; Now the base primitive is rotated about the x axis by 90 degrees, then
; rotated about the z axis by 45 degrees, and finally translated by 1 unit along
; the x and y axes. Typically, this command would be issued by the Planner once
; it has derived a complete and successful partial plan for a TLI to update the
; World Model.

-> (move base 1.0 1.0 0.0 90.0 0.0 45.0)

; Franz responds with the result of the move function which is the internal
; representation of the structure representing the base. The configuration
; field of the object, the first term of the list, has been updated to reflect
; the new configuration of the base.

(array[16] 10.0 nil primitive array[4] nil (base.base base.front base.top
 base.back base.left base.right))

; Now the top is positioned on the base. The parameters of the move would again
; be provided by the Planner, and the operation performed once a successful
; partial plan had been found.

-> (move top 1.707106 1.0 1.0 0.0 0.0 0.0)

; Again, Franz responds with the result of the move operation.

(array[16] 7.85 nil primitive array[4] nil (top.top top.bottom top.side))
```

```
; Now the tower object is created by the rigid-attach function. The name of the
; object to create, as well as the description of the two objects to be rigidly
; attached, are passed as parameters to the function. The name of the object to
; create must be passed to update the part-of fields of the two constituent
; objects, and to fill the name field of the new object. The setf function
; updates the value of the symbol tower to the newly created structure. Note
; that the value of the symbol tower CANNOT be modified within the function
; since side effects of this sort are prohibited in LISP.

-> (setf tower (rigid-attach 'tower base top))

; Franz responds with the value bound to the symbol tower, which is the
; internal representation of the newly created structure.

(compound tower compound general unknown array[16] 17.85 nil compound array[4]
 (0.0 0.0 0.0) (rigid base primitive top primitive))

; Now Franz is asked to describe the newly created compound object, and all its
; constituent elements in their new configuration.

-> (describe tower :all)

; Description of compound object tower

Description of solid: tower
                name:                   tower
                type:                   compound
                geometric class:        general
                functional class:       unknown
                part of:                nil
                mass:                   17.85
                center of mass:
                        x-coordinate:   1.707106437639522
                        y-coordinate:   1.0
                        z-coordinate:   0.9397759103641457
                constituent objects:    (rigid base primitive top primitive)

; Description of first constituent solid

Description of solid: base
                name:                   base
                type:                   primitive
                geometric class:        cube
                functional class:       sub-component
                part of:                tower
                mass:                   10.0
                center of mass:
                        x-coordinate:   1.707106781186548
                        y-coordinate:   1.0
                        z-coordinate:   0.5
                list of faces:          base.base
                                        base.front
                                        base.top
                                        base.back
                                        base.left
                                        base.right
```

```
; Description of the 6 faces making up the base

        Description of face: base.base
                name:                       base.base
                type:                       rectangle
                length:                     1.0
                width:                      1.0
                analytic description:       A = 0.0
                                            B = 0.0
                                            C = 0.0
                                            F = 0.0
                                            G = 0.0
                                            H = 0.0
                                            P = 0.7071067811865475
                                            Q = -0.7071067811865475
                                            R = 5.721188726109832e-18
                                            D = 1.962615573354719e-17
                in-out-flag:                in
                is part of:                 base
                has vface:                  base.vbase
         the face has 4 edges:              base.edge1 base.edge2
                                            base.edge3 base.edge4

        Description of face: base.front
                name:                       base.front
                type:                       rectangle
                length:                     1.0
                width:                      1.0
                analytic description:       A = 0.0
                                            B = 0.0
                                            C = 0.0
                                            F = 0.0
                                            G = 0.0
                                            H = 0.0
                                            P = 0.3535533905932738
                                            Q = 0.3535533905932738
                                            R = 0.0
                                            D = -2.414213562373095
                in-out-flag:                in
                is part of:                 base
                has vface:                  base.front
         the face has 4 edges:              base.edge4 base.edge5
                                            base.edge12 base.edge10
```

```
Description of face: base.top
        name:                   base.top
        type:                   rectangle
        length:                 1.0
        width:                  1.0
        analytic description:   A = 0.0
                                B = 0.0
                                C = 0.0
                                F = 0.0
                                G = 0.0
                                H = 0.0
                                P = 0.3535533905932738
                                Q = -0.3535533905932738
                                R = 2.860594363054916e-18
                                D = -1.0
        in-out-flag:            in
        is part of:             base
        has vface:              base.vtop
 the face has 4 edges:          base.edge6 base.edge8
                                base.edge11 base.edge12

Description of face: base.back
        name:                   base.back
        type:                   rectangle
        length:                 1.0
        width:                  1.0
        analytic description:   A = 0.0
                                B = 0.0
                                C = 0.0
                                F = 0.0
                                G = 0.0
                                H = 0.0
                                P = 0.7071067811865475
                                Q = 0.7071067811865475
                                R = 0.0
                                D = -2.82842712474619
        in-out-flag:            in
        is part of:             base
        has vface:              base.vback
 the face has 4 edges:          base.edge2 base.edge7
                                base.edge8 base.edge9
```

```
Description of face: base.left
        name:                   base.left
        type:                   rectangle
        length:                 1.0
        width:                  1.0
        analytic description:   A = 0.0
                                B = 0.0
                                C = 0.0
                                F = 0.0
                                G = 0.0
                                H = 0.0
                                P = 0.0
                                Q = 0.0
                                R = 1.0
                                D = 0.0
        in-out-flag:            in
        is part of:             base
        has vface:              base.vleft
 the face has 4 edges:          base.edge1 base.edge5
                                base.edge6 base.edge7
Description of face: base.right
        name:                   base.right
        type:                   rectangle
        length:                 1.0
        width:                  1.0
        analytic description:   A = 0.0
                                B = 0.0
                                C = 0.0
                                F = 0.0
                                G = 0.0
                                H = 0.0
                                P = 0.0
                                Q = 0.0
                                R = 0.5
                                D = -1.0
        in-out-flag:            in
        is part of:             base
        has vface:              base.vright
 the face has 4 edges:          base.edge3 base.edge9
                                base.edge10 base.edge11

; Description of the 12 edges of the base

            Description of edge: base.edge9
             name:                  base.edge9
             type:                  line
             part of:               (base.right base.back)
             end point 1 is at:
                    x-position:     1.707106781186548
                    y-position:     0.2928932188134525
                    z-position:     1.0
             end point 2 is at:
                    x-position:     1.0
                    y-position:     1.0
                    z-position:     1.0
```

```
Description of edge: base.edge7
 name:                    base.edge7
 type:                    line
 part of:                 (base.left base.back)
 end point 1 is at:
        x-position:       1.0
        y-position:       1.0
        z-position:       0.0
 end point 2 is at:
        x-position:       1.707106781186548
        y-position:       0.2928932188134525
        z-position:       5.721188726109832e-18

Description of edge: base.edge11
 name:                    base.edge11
 type:                    line
 part of:                 (base.right base.top)
 end point 1 is at:
        x-position:       1.707106781186548
        y-position:       0.2928932188134525
        z-position:       1.0
 end point 2 is at:
        x-position:       2.414213562373095
        y-position:       1.0
        z-position:       1.0

Description of edge: base.edge8
 name:                    base.edge8
 type:                    line
 part of:                 (base.top base.back)
 end point 1 is at:
        x-position:       1.707106781186548
        y-position:       0.2928932188134525
        z-position:       5.721188726109832e-18
 end point 2 is at:
        x-position:       1.707106781186548
        y-position:       0.2928932188134525
        z-position:       1.0

Description of edge: base.edge6
 name:                    base.edge6
 type:                    line
 part of:                 (base.left base.top)
 end point 1 is at:
        x-position:       2.414213562373095
        y-position:       1.0
        z-position:       5.721188726109832e-18
 end point 2 is at:
        x-position:       1.707106781186548
        y-position:       0.2928932188134525
        z-position:       5.721188726109832e-18
```

```
Description of edge: base.edge10
 name:                    base.edge10
 type:                    line
 part of:                 (base.right base.front)
 end point 1 is at:
         x-position:      1.707106781186548
         y-position:      1.707106781186548
         z-position:      1.0
 end point 2 is at:
         x-position:      2.414213562373095
         y-position:      1.0
         z-position:      1.0

Description of edge: base.edge12
 name:                    base.edge12
 type:                    line
 part of:                 (base.front base.top)
 end point 1 is at:
         x-position:      2.414213562373095
         y-position:      1.0
         z-position:      5.721188726109832e-18
 end point 2 is at:
         x-position:      2.414213562373095
         y-position:      1.0
         z-position:      1.0

Description of edge: base.edge5
 name:                    base.edge5
 type:                    line
 part of:                 (base.front base.left)
 end point 1 is at:
         x-position:      1.707106781186548
         y-position:      1.707106781186548
         z-position:      0.0
 end point 2 is at:
         x-position:      2.414213562373095
         y-position:      1.0
         z-position:      5.721188726109832e-18

Description of edge: base.edge4
 name:                    base.edge4
 type:                    line
 part of:                 (base.base base.front)
 end point 1 is at:
         x-position:      1.707106781186548
         y-position:      1.707106781186548
         z-position:      0.0
 end point 2 is at:
         x-position:      1.707106781186548
         y-position:      1.707106781186548
         z-position:      1.0
```

```
Description of edge: base.edge3
 name:                    base.edge3
 type:                    line
 part of:                 (base.base base.right)
 end point 1 is at:
        x-position:       1.0
        y-position:       1.0
        z-position:       1.0
 end point 2 is at:
        x-position:       1.707106781186548
        y-position:       1.707106781186548
        z-position:       1.0

Description of edge: base.edge2
 name:                    base.edge2
 type:                    line
 part of:                 (base.base base.back)
 end point 1 is at:
        x-position:       1.0
        y-position:       1.0
        z-position:       0.0
 end point 2 is at:
        x-position:       1.0
        y-position:       1.0
        z-position:       1.0

Description of edge: base.edge1
 name:                    base.edge1
 type:                    line
 part of:                 (base.base base.left)
 end point 1 is at:
        x-position:       1.0
        y-position:       1.0
        z-position:       0.0
 end point 2 is at:
        x-position:       1.707106781186548
        y-position:       1.707106781186548
        z-position:       0.0

; Description of second constituent solid

Description of solid: top
 name:                    top
 type:                    primitive
 geometric class:         solid-cylinder
 functional class:        sub-component
 part of:                 tower
 mass:                    7.85
 center of mass:
        x-coordinate:     1.707106
        y-coordinate:     1.0
        z-coordinate:     1.5
 list of faces:           top.top
                          top.bottom
                          top.side
```

```
; Description of the 3 faces of the cylinder

        Description of face: top.top
                name:                   top.top
                type:                   disk
                radius                  0.5
                center-point            (1.707106 1.0 2.0)
                analytic description:   A = 0.0
                                        B = 0.0
                                        C = 0.0
                                        F = 0.0
                                        G = 0.0
                                        H = 0.0
                                        P = 0.0
                                        Q = 0.0
                                        R = 0.5
                                        D = -2.0
                in-out-flag:            out
                is part of:             top
                has vface:              top.vtop
         the face has 1 edges:          top.edge1

        Description of face: top.bottom
                name:                   top.bottom
                type:                   disk
                radius                  0.5
                center-point            (1.707106 1.0 1.0)
                analytic description:   A = 0.0
                                        B = 0.0
                                        C = 0.0
                                        F = 0.0
                                        G = 0.0
                                        H = 0.0
                                        P = 0.0
                                        Q = 0.0
                                        R = 0.5
                                        D = -1.0
                in-out-flag:            in
                is part of:             top
                has vface:              top.vbottom
         the face has 1 edges:          top.edge2
```

```
Description of face: top.side
        name:                   top.side
        type:                   cylinder
        height:                 1.0
        radius:                 0.5
        analytic description:   A = 1.0
                                B = 1.0
                                C = 0.0
                                F = 0.0
                                G = 0.0
                                H = 0.0
                                P = -1.707106
                                Q = -1.0
                                R = 0.0
                                D = 2.914210895236
        in-out-flag:            out
        is part of:             top
        has vface:              top.vside
 the face has 2 edges:          top.edge1
                                top.edge2

; Description of the 2 edges of the cylinder

            Description of edge: top.edge2
             name:                  top.edge2
             type:                  circle
             part of:               (top.bottom top.side)
             circle radius is:      0.5
             circle center is at:
                    x-position:     1.707106
                    y-position:     1.0
                    z-position:     1.0
             circle passes through  (2.207106 1.0 1.0)
                                    (1.207106 1.0 1.0)

            Description of edge: top.edge1
             name:                  top.edge1
             type:                  circle
             part of:               (top.top top.side)
             circle radius is:      0.5
             circle center is at:
                    x-position:     1.707106
                    y-position:     1.0
                    z-position:     2.0
             circle passes through  (2.207106 1.0 2.0)
                                    (1.207106 1.0 2.0)

; Describe returns nil when done

nil
```

5.5 Conclusion

In this chapter, the implementation of the World Model was discussed, and a sample one created and manipulated to prove the feasibility of using a hybrid CSG/B-rep representation scheme. While quadrics were used to model surfaces, no claim is made that this is the best approach. In fact, as stated in chapter 6, most planning algorithms work on polyhedral models of objects. However, a polyhedral model can be built relatively simply from a quadric description, while the converse is impossible: essential topological information is lost when going from quadrics to polygons. The modeling of surfaces is currently an area of intense research and much more work must be done before a definitely superior scheme can be identified.

The World Model maintains both symbolic and analytic information about entities. The symbolic information conveys some information about the essential topology of an entity that the Planner can use to draw inferences about geometric problems with a minimum of computation. The analytic information allows the Planner to do extensive geometric, kinematic, and dynamic analyses about objects and motions.

Much research remains to be done in the area of modeling objects whose shape changes under external forces, such as mechanisms, objects composed of rigid members connected by movable links, and such objects as thin flexible sheets, cloth, etc.

Chapter 6

Planner

6.1 Introduction

As discussed in chapter 3, the Planner is made up of five modules: the Parser, the Main Planner, and the three specialized planning modules that seek to produce an executable robot program. The architecture of the Planner, reproduced in figure 6.1, was motivated by the need to decouple the specialized planning modules from one another, so that any one of them may be easily replaced with another to test a different algorithm. This decoupling is possible because the specialized planning modules interact together by passing constraints back and forth at the top level.

The Planner uses the method of symbolic constraint propagation as proposed by Brooks [6] and discussed briefly in section 6.3. An introduction to the use of constraint propagation can be found in [76,75]. More details on the technique can be found in several papers on specific planners, such as CONSTRAINTS [66], EL [64], SIPE [74], and ACRONYM [8,5]. Symbolic constraint propagation allows the Planner to propagate constraints on the values of the planning variables both in the forward and backward direction. The backward propagation of constraints can save substantial search time over a "blind" depth-first search by constraining choices at a given point in the search.

Since the purpose of the overall system is to allow experimentation with various path planning, fine motion synthesis, and grasp planning algorithms, the implementation of the various planning modules is not discussed. However, the chapter

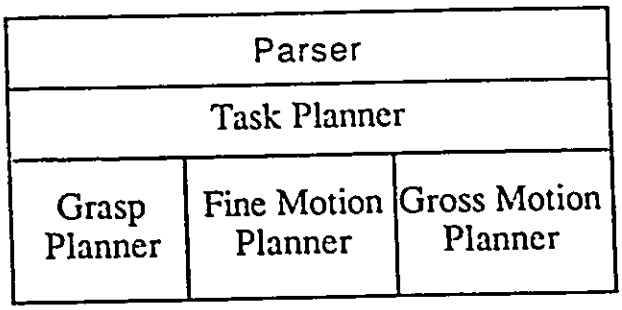

Figure 6.1: Logical architecture of the Planner

provides a review of the existing planning techniques and describes the current state of the art.

The chapter is divided into five sections. The first three look at gross motion, fine motion, and grasp planning. The fourth section discusses the implementation of the Parser using some pseudo code to clarify the discussion. The last section discusses the Main Planner and describes a typical planning algorithm in detail.

6.2 Gross Motion Planner

The function of the Gross Motion Planner is to generate collision free paths between two regions of space using only *pure position control*. Compliant and guarded motions are not allowed in gross motion planning. In some cases, a path may exist if compliant and guarded motions were allowed along some of its segments. However, this would slow down the operation of the robot drastically and reduce its productivity. To maintain productivity, the system requires that paths connecting "distant" regions of space be connected using pure position control. Consequently, the Gross Motion Planner thus outputs a sequence of robot motions that will navigate a payload and the robot's members through obstacles without touching anything. If such a path cannot be found, the planner signals its failure and may output constraints that may help other modules to derive alternatives. The Gross Motion Planner can then be reinvoked in an attempt to find a path under new conditions.

The path planning problem, generally referred to as the FindPath problem, has two essential components:

1. the representation of the physical and geometric constraints, and

2. the search for a solution that satisfies all the constraints simultaneously.

Essentially, the constraints represent configurations of the robot that are prohibited either because they would cause collisions or because of other mechanical constraints, such as extending too far with a heavy payload. An ideal solution would completely characterize the constraints and provide a complete algorithm for that characterization, i.e. find a path in all cases where one exists in the given model. Schwartz and Sharir have proven the existence of a representation-complete and search-complete algorithm to solve the FindPath problem given an arbitrary number of moving and stationary bodies. The time taken by their algorithm to find a solution is a polynomial function of the total number of smooth surfaces in the model and of the maximal degree of the equations defining them, and an exponential function of the number of degrees of freedom of the system of bodies. In the case where the only moving object is a six degree of freedom robot, the algorithm is $O(n^{64})$, where n is the polynomial function of the number of faces in the model and the maximal degree of the equation defining them [61,6,27].

It is only in the last two or three years that representation-complete, search-complete algorithms have been implemented [18,40,12]. This section briefly reviews some of the algorithms proposed. The algorithms are characterized according to the way they model free space.

6.2.1 Finding a Path through Configuration Space

The key idea of the configuration space approach is to provide a representation of free space in which the problem of finding a collision free path for a three-dimensional object through 3-D space is simplified to the problem of finding a collision free path for a *point* through a n-dimensional space, where n is the number of degrees of freedom of the overall system of bodies.

The use of configuration space was initially proposed by Udupa, and further developed by Tomás Lozano-Pérez [39,41,40]. A few years ago, Lozano-Pérez proposed an algorithm based on configuration space that applied only to Cartesian robots [41]. Later, that algorithm was generalized to any type of robot [40]. This latter algorithm is briefly described below for a revolute robot.

The algorithm maps solids modeled as the union of finite, rigid, possibly overlapping 3-D polyhedrons into n-dimensional manifolds whose boundaries are $(n-1)$-dimensional "surfaces", where n is the number of degrees of freedom of the robot, if the robot is the only moving object. In the case of a six degree of freedom revolute robot, the resultant space is six-dimensional, each axis describing the angle of a joint. Such a space is called a *configuration space* or *C-space*. The robot itself and its *rigid* payload are reduced to a single point in the C-space. The problem of finding a path through 3-D space is reduced to the problem of navigating the point representing the robot through its C-space. That point is allowed to move parallel to $(n-1)$-dimensional "surfaces", or along 1-dimensional to $(n-2)$-dimensional "surface" intersections, or to jump between n-dimensional obstacles.

To build the representation of free space, the algorithm begins by approximating all the obstacles in C-space. An obstacle is nothing more that an area of C-space the robot cannot occupy without causing a collision. This *obstacle space* is built from the union of $(n-1)$-dimensional *slice projections*. Each $(n-1)$-dimensional slice projection represents the possible configuration of the robot with one joint fixed at a certain angle. Each $(n-1)$-dimensional slice projection is recursively defined as the union of $(n-2)$-dimensional slice projection, and so on until a union of linear ranges is obtained. The resultant C-space approximation for the case of a 2 degree of freedom robot is shown in figure 6.2.

Once the C-space of the robot is determined, its free space is defined simply as the union of the same slice projections, except that the linear ranges are exchanged for their complement. Since the work envelope of the robot is finite, the complement of every linear range is also finite. To reduce the storage requirements and computation time, the free space is represented as a set of partially overlapping rectangular regions, each region being built from adjacent linear ranges. A region graph is then built to represent the connectivity of the various regions up to dimension n. The search for a path then reduces to a search through the region graph using the A^* search algorithm, a modified branch-and-bound search with underestimating of the remaining distance that produces an optimal path according to some metric (in fact, the minimal joint motions). The A^* algorithm is a well known search algorithm in artificial intelligence and is described in [76,50]. Because of the

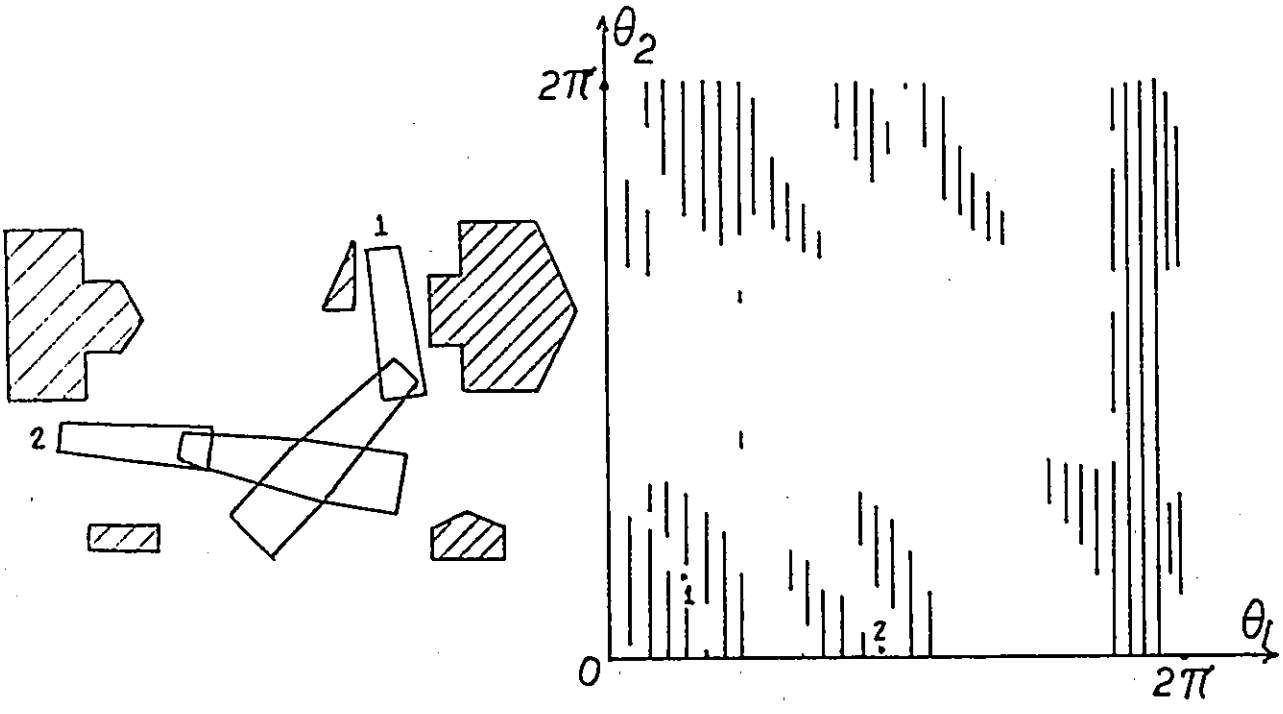

Figure 6.2: Resultant C-space (b) for a 2 degree of freedom robot in a cluttered environment (a)

Adapted from [40, page 6]

rules controlling the motion of the point through C-space, the resultant path tends to be very ragged. Some form of postprocessing to smooth out the robot motion is reported to be under investigation [40].

While the algorithm is conceptually simple and elegant, the amount of computation required for a good approximation of the free space of a six degree of freedom robot is staggering. Lozano-Pérez has proposed to reduce the computational load by computing free space only in the vicinity of the manipulator, and by restricting the number of degrees of freedom of the robot [40].

6.2.2 Voronoi Diagrams

The use of Voronoi diagrams to solve the FindPath problem has recently attracted some attention. The Voronoi diagram of a set of stationary obstacles characterizes its free space as a set of trajectories that are maximally clear of those obstacles according to some metric. The search for a collision free path then reduces to a search for a path connecting the initial position of the manipulator to a point on the Voronoi diagram, the final position of the manipulator to a point on the Voronoi diagram, and the two points on the Voronoi diagram. Voronoi diagrams in themselves do not produce collision free paths, but narrow down the search for such paths to a set of possible trajectories.

Voronoi diagrams are formally defined below, and a small two-dimensional example is shown. All definitions are taken from [63].

1. **Distance**—The distance (not necessarily Euclidean) between two points e_i and e_j is denoted $d(e_i, e_j)$. The distance between a point e_i and a set of points X is defined as

$$d(e_i, X) = min\,\{d(e_i, e_j); e_j \in X\}$$

2. **Bisectors**—The bisector $B(e_i, e_j)$ of two points e_i and e_j is the locus of points equidistant from e_i and e_j, i.e.

$$B(e_i, e_j) = \{x \mid d(x, e_i) = d(x, e_j)\}$$

Similarly, the bisector $B(X, Y)$ of two sets of points is the locus of points equidistant from X and Y, i.e.

$$B(X, Y) = \{x \mid d(x, X) = d(x, Y)\}$$

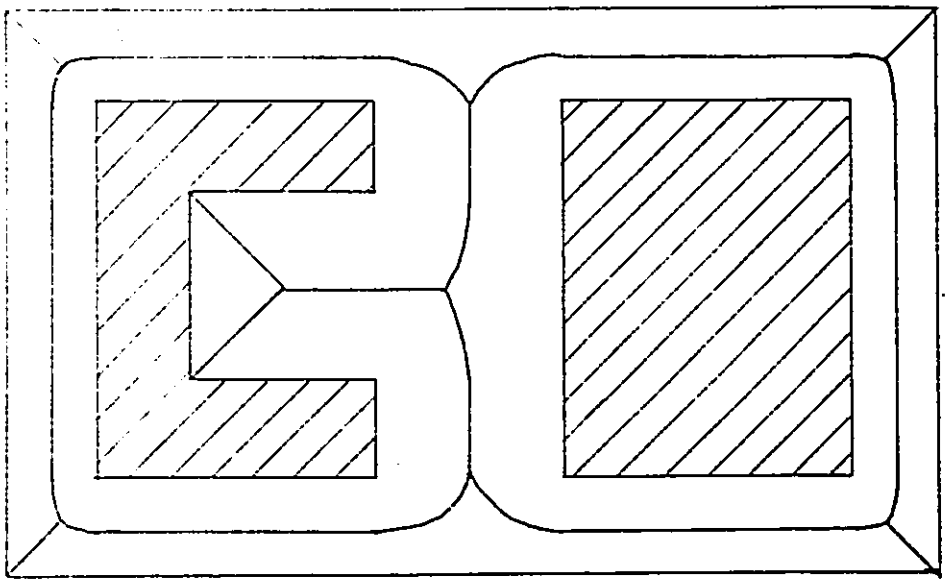

Figure 6.3: Voronoi diagram of two 2-D objects in a rectangular space

3. The open half-plane $h(e_i, e_j)$ is the set of points that lie closer to the point e_i than to the point e_j. The complement of $h(e_i, e_j)$ is denoted $\overline{h}(e_i, e_j)$. Note that $\mathrm{bd}(\overline{h}(e_i, e_j)) = B(e_i, e_j)$.

4. Let S_1 and S_2 be two sets of points. The Voronoi region $V(S_1, S_2)$ of S_1 with respect to S_2 is the set of all points closer to S_1 than to S_2, i.e.

$$V(S_1, S_2) = \cup_{e_i \in S_1}(\cap_{e_j \in S_2} h(e_i, e_j))$$

5. The Voronoi diagram VOD(S) of a set of points $S = \{e_i\}$ is given by

$$\cup_{e_i \in S} V(e_i, S - e_i)$$

It is customary in robotics research to let

$$\mathrm{VOD}(S) = \mathrm{bd}(\cup_{e_i \in S} V(e_i, S - e_i))$$

Figure 6.3 illustrates the Voronoi diagram of two 2-D polygons enclosed in the rectangular work envelope of a robot manipulator.

John Canny and Bruce R. Donald have proposed to use Voronoi diagrams to find collision free paths. They first build the C-space representation of a manipulator

to account for collisions, and then derive the Voronoi diagram of the resultant n-dimensional set of obstacles [12]. They then look for a path that is maximally clear of obstacles along the diagram, unlike Lozano-Pérez which looks for the path of minimum joint motions.

6.2.3 Generalized Cones

Brooks reduced the complexity of the FindPath problem by deliberately restraining the number of degrees of freedom of a robot from six to four [6]. Brooks justifies his approach by pointing out that, during gross motion planning, the Planner computes collision free trajectories for pick and place operations—picking an object up in one place and putting it down somewhere else—and that, in most cases, the only reorientation of the object will be about its vertical axis, or perhaps at the end of the gross motion using pure wrist motions. Arbitrary reorientation of the manipulator to avoid collisions is only necessary when the object is much larger than the manipulator or when the environment is extremely cluttered. Consequently, four degrees of freedom are sufficient for this type of motion. Furthermore, he argues that four degrees of freedom are also often sufficient for fine motions, referring to Nevins and Whitney's work [48]. Typically, in an insertion task, there is a direction of insertion and the only rotations are about a vector in that direction. Thus three translational degrees of freedom and one rotational degree of freedom suffice.

Brooks also states that in cases where more than four degrees of freedom are required for fine motions, six degrees of freedom are generally not enough: any motion of the first few joints of the robot manipulator must be very accurate to maintain a certain degree of accuracy at the end effector. Instead of building highly accurate robots, it may be preferable to build wrists with six degrees of freedom, effectively providing a "mini" robot at the end of a movable arm.

Brooks's algorithm represents free space as overlapping generalized sweeps, also called generalized cones, with straight line spines. These generalized cones constitute *freeways*. Each point on a spine is associated to a set of constraints, the legal orientations of the load. Since each member of the robot and the payload have different motion constraints, free space must be characterized for each, i.e. a freeway must be defined for each. A path for each robot member is then searched for along

the spines, and can change spines only where they intersect. When transferring spines, the orientation of the object must be legal for both spines.

Brook's tested his algorithm using the ACRONYM image-recognition system that models objects as generalized cylinders (or cones). The system acquired a scene using a vision system and built a polygonal model of it. Then, simple polygonal representations of freeways in the model were synthesized.

6.2.4 Other schemes

Many other techniques have been proposed to compute collision free trajectories through space. One approach, the use of artificial potential fields, uses a local characterization of free space rather than a global one [27]. In this scheme, a path is searched for along the lines of least potentials. This method has a few drawbacks and seems to be applied only to the computations of paths on-line. Necsulescu discusses the use of artificial potential fields to compute safe paths in the case of an emergency, such as when someone steps into the work envelope of the robot [47].

Another approach was proposed by Hasegawa. His algorithm plans paths for a six degree of freedom revolute robot using a 3-D representation of free space rather than a n-dimensional configuration space [27]. The method is reminiscent of Lozano-Pérez's in it's quantization of joint angles to derive an approximation of free space in an acceptable time. First, the *Principal Function Space* or *PFS* is built from the collision free configurations of the arm, which has three degrees of freedom. Since Hasegawa quantizes each joint's rotation into 64 steps, the PFS is a $64 \times 64 \times 64$ array. Next, an approximation of *Full Free Space* or *FFS* is built. The FFS is defined as the space in which the robot can take any configuration without collisions. To build the FFS, the space swept out by the wrist and its load, which collectively have three degrees of freedom, is modeled for each point or element in the PFS. Those free space cells of the PFS for which the full space swept by the wrist is collision free become free space cells in the FFS. Then a connectivity analysis is performed on the FFS and PFS: all neighboring free space cells are labelled identically, thus yielding a set of spatial regions. The problem of finding a collision free path from an initial position to a goal position then reduces to the problem of finding three collision free paths in smaller spaces: one path from

the initial position to the FFS, one path from the goal position to the FFS, and one path through the FFS. This last path is trivial to compute since, by definition, the robot may assume any configuration without causing collisions. Obviously, the larger the FFS regions are, the simpler the problem. The other two collision free paths must lie in the PFS. To find these paths, two geodesic domes centered on the initial and goal point are built. A geodesic dome is a polygonal approximation of a sphere using triangles as faces. Each facet defines the base of a pyramid with its vertex at the center of the dome. To each pyramid is associated a depth value . This value indicates the distance between the center of the dome and an obstacle in the direction of the principal axis of the pyramid. Hasegawa then describes an algorithm to find a path using only translations along the principal axes of the pyramids and rotations about the center of the dome from one pyramid to another. Hasegawa also describes how to partition a path into a number of smaller paths to which the described procedure can be applied. As can be seen, the proposed algorithm isn't as conceptually simple as Lozano-Pérez's or Canny's, nor as elegant.

Finally, some work has also been done in the planning of time optimal paths [60] and in planning paths for multiple robots [33,22].

By carefully engineering the layout of a work cell and the assembly operations, the FindPath problem can be trivialized. For instance, by using an overhead Cartesian robot, it is possible to grasp an object, lift it above all other obstacles, and then lower it in position. However, the FindPath problem cannot be trivialized for fine motions, hence the wide interest in the topic.

6.3 Fine Motion Planner

The function of the Fine Motion Planner is to generate guarded and compliant motions to achieve successful assembly operations. Guarded and compliant motions were briefly defined in chapter 2. Both utilize hybrid position/force control. A guarded motion consists of a commanded velocity—speed and direction—of the manipulator and a termination *predicate* or test. The general form of the termination predicate is a combination of force, torque, position and time values. Typically, a guarded motion is used to approach and contact a surface, in which case motion

terminates when (1) force/torque sensors signal contact, or (2) position sensors indicate the manipulator is out of the region of space in which contact should have occurred—the robot shouldn't move across the entire work space until it bumps something—or (3) a timer signals that something should have happened by that time. A compliant motion consists of a commanded contact force in a given direction as well as an initial commanded velocity in a given direction. However, during a compliant motion, the robot must modify its path to maintain the commanded contact force. A compliant motion also has a termination predicate, just as a guarded motion. Compliant motions are used to follow a surface with a constant contact force, and to detect surface features as specified by their termination predicate. For example, to find a hole, a guarded motion can position the end effector or its load on a surface where the hole is presumed to be, then a compliant motion initiated. The hole is found whenever the contact force cannot be maintained in a given region of space, and the position and torque sensory data is consistent with the discovery of the hole.

The Fine Motion Planner should produce and output a sequence of guarded and compliant motions, as well as the regions of space from which motion may be initiated with guarantee of success. The exact sequence of guarded and compliant motions required to accomplish an assembly task depends on:

1. the geometry of the robot, payload and objects before motion is initiated,
2. the desired relationship between objects at the end of the sequence of motions,
3. the uncertainty in the dimensions and locations of features of objects due to tolerances, and
4. the uncertainty in the position and velocity of the robot due to its limited accuracy.

Consequently, the generation of fine motions requires an analysis of the geometry of a problem, the uncertainties involved, the robot's dynamics, and the friction between surfaces. Given that fine motions are crucial in robotic assembly and the prime reason task-level programmed systems are desirable, it is somewhat surprising that the topic hasn't attracted the attention that gross motion planning has.

Taylor was one of the first to study the problem of fine motion generation based on an analysis of the uncertainties in the environment [43]. Taylor's system was provided with a number of *strategy skeletons*, AL procedures that consist of fine motions and sensing operations to solve a particular assembly task under a given set of geometric constraints. Taylor's idea was to propagate the uncertainties in numerical terms through models of the assembly tasks, analyze the results, and then select an appropriate skeleton and instantiate the skeleton's parameters.

AUTOPASS also proposed to use libraries of sensory feedback routines to implement fine motions [37]. How a particular procedure was to be selected was not discussed.

Lozano-Pérez's LAMA system was similar to Taylor's except that, following analysis, the skeletons could be modified to introduce sensing operations to reduce the uncertainties in the environment [44].

Brooks extended the idea of Taylor by representing and analyzing constraints in symbolic rather than numeric terms [7]. The constraints could be propagated forward to estimate uncertainties in particular operations, or backwards to constrain the value of plan variables to values that guarantee success of the partial plan. When no such values could be found, the system could introduce sensing operations in an attempt to reduce the uncertainties sufficiently to guarantee success.

Lozano-Pérez's ATLAS system maintained the idea of a library of skeletons, but proposed a fine motion synthesizer that could, in theory, produce a sequence of fine motions for any task, geometry, and uncertainties [42]. The system described had a skeleton only for the ubiquitous peg-in-hole task.

The first attempts at developing an actual theory from which fine motion synthesizers could be designed were made by Mason, who formalized the notion of active compliance and extended it to C-space [46], and by Lozano-Pérez, Mason, and Taylor [43] who proposed the use of *pre-images* for fine motion synthesis. The method of pre-images completely abandons the idea of prestored skeletons; all fine motions are synthesized from the geometric and physical constraints of the assembly task. The idea behind the pre-image method is to compute all possible initial configurations from which a sequence of guarded motions is guaranteed to achieve the desired task given uncertainties in the initial configuration of the robot and its commanded velocity, the tolerances on parts, friction, and the robot's dynamics.

These initial configurations constitute the pre-images of a goal. If such pre-images cannot be found, compliant motions are introduced and the process repeated. Compliant motions yield larger pre-images than guarded motions. If no pre-image can be found, the problem is assumed to be unsolvable. In order to avoid the computation of solid intersections to detect collisions, all computations are done in the C-space of the manipulator. Backchaining is used to derive a sequence of pre-images until one is attainable without the use of fine motions. The proposed method was never implemented. In fact, Erdmann [23] showed that the general form of pre-images is not computable. However, he proposed to substitute pre-images with backprojections which are computable, and showed that under some conditions, backprojections approximate pre-images. He also suggested a method to represent friction in C-space.

6.4 Grasp Planner

Very little work has been done on grasp planning, perhaps because the problem is conceptually simpler that gross or fine motion planning. The Grasp Planner's function is to select grasp configurations consistent with the end effector's geometry and configuration, the forces and torques exerted during motion, and the goal configuration as defined by the fine motion planner.

Lozano-Pérez's LAMA system is perhaps the only proposal that gave more than passing consideration to grasp planning [44]. In an initial phase, LAMA's grasp planner would make a list of grasp configurations ordered according to stability and chances of slipping, and in a latter phase, once some plan variables are instantiated, select a particular grasp configuration from the set.

Note that the Grasp Planner's function is to select a grasp configuration which becomes a goal to a motion planner; no motions are generated by the Grasp Planner itself. The motions required to grasp the object will be determined by the Gross and Fine Motion Planners.

```
procedure plan ();
  begin
    *plan* <- nil;
    tli <- task-level-program;
    if tli = create-statement
      then begin
        tli <- add-instruction (tli initial-state);
        parse (tli)
      end
      else error ("First instruction must be a create statement")
    end;
  end;
```

Figure 6.4: Translate procedure pseudo code

6.5 Parser

The Parser's purpose is to verify the syntax of each task-level instruction or TLI, select the appropriate subgoal template associated to the TLI, setup the final goal in a "standard" representation, and invoke the Main Planner with the subgoal template and final goal as parameters.

The Parser is made up of a main procedure called **parse** and several other procedures, one for each task-level verb. These in turn may be made up of several other procedures. The Parser is implemented in this way to provide for the extension of the language with virtually no rewriting of existing code. A new instruction can be added at any time by adding its specific parser and its subgoal template to the existing system. The actual implementation of the Parser is outlined below, using skeleton code to illustrate the discussion. This pseudo code is given in a Pascal- or C-like style rather than in a LISP style for the benefit of the reader unfamiliar with LISP. When necessary, deviations between the LISP implementation and that illustrated in the pseudo code are explained in detail. The entire top level LISP code of the Parser appears in figures 6.8 and 6.9.

The conversion of a task-level program is initiated by a call to the `plan` procedure, illustrated in figure 6.4. This procedure is not part of the Parser itself, but is

```
procedure parse (tli);
  begin
    case get-verb (tli) of
      create: parse-create (tli);
      mate: parse-mate (tli);
      ⋮
      otherwise: error ("Non existent task-level instruction specified")
    end;
  end;
```

Figure 6.5: Parse procedure pseudo code

discussed here nonetheless for completeness. The `plan` procedure begins by initializing the `*plan*` variable to `nil`—the empty list. The `*plan*` variable is a global variable (it is customary to bracket global variables with asterisks in LISP) which will accumulate the successful partial plans produced during planning. The TLI variable is also initialized to the entire task-level program, which should be a single `create` statement. If it is, then a special instruction must be added as the last instruction of the task-level program to re-establish the initial state of the system. This is necessary if the resultant executable program is to be run several times in succession. Once this is done, the augmented program is passed on to `parse`.

The `parse` procedure, shown in figure 6.5, verifies the validity of each instruction verb and then invokes the specific parser for that instruction. Note that the pseudo code shows a case statement in which the appropriate parser for the specific instruction is selected. Such an approach does not permit extension of the language without modification of the existing code, but does provide a simple illustration of the concept involved. In LISP, the name of the parser to call would first be "built" from `parse-` and the instruction verb, and then "funcalled" with its argument using the statement:

```
(funcall (concat 'parse- (get-instruction-verb tli)) tli)
```

To insure that this is a valid function call, it is necessary to verify before hand that the instruction verb is valid. The simplest way to do this is to keep all the valid

```
procedure parse-create (tli);
  begin
    name <- get-name-of-object-to-create (tli);
    if exists-already (name)
        then error (name "is already defined");
    list-of-instruction <- get-body-of-create (tli);
    foreach instruction in list-of-instruction do
      begin
        make-name (new-name);
        bind-name (new-name instruction)
      end;
    instruction <- last-instruction (list-of-instruction);
    if not special (instruction)
      then bind-name (name instruction)
      else begin
        instruction <- one-before-last-instruction (list-of-instructions);
        bind-name (name instruction)
      end;
    foreach instruction in list-of-instruction do
      parse (instruction);
  end;
```

Figure 6.6: Parse-create procedure pseudo code

instruction verbs in a single list, `*inst-verb-list*`, and verify that the current instruction verb is a member of the list, i.e.

```
(member (get-instruction-verb tli) *inst-verb-list*)
```

The entire `parse` function in LISP is shown in figure 6.8 and figure 6.9 where `inst-verb` is set to the instruction verb of the task-level instruction once for the sake of performance. The `error` function causes an error trap so that the `funcall` does not get executed unless the programmer invokes the debugger and forces program continuation, presumably after correcting the error. In a "production" version of the Planner, the Planner would simply stop and return control to the calling program.

The `parse-create` procedure, illustrated in figure 6.6, first verifies if the name specified in the `create` statement clashes with an existing name—all object names

```
procedure parse-mate (tli);
  begin
    if not mate-parameters-ok (tli)
      then error ("Mate instruction parameters are incorrect");
    goal <- build-goal-representation (tli);
    return (mate-subgoal-template goal)
  end;
```

Figure 6.7: Parse-mate procedure pseudo code

must be unique. If so, it signals the error. Otherwise, it processes every task-level instruction in the body of the `create`, assigning a new name to every object that will be created by every instruction. The name of the object to be created is assigned to the last instruction of the `create`, unless that instruction is the special one that restores the initial state of the system, in which case the name is bound the next to last instruction. Finally, the `parser` procedure is invoked recursively to process each instruction in the body of the `create`.

A typical individual instruction parser is illustrated in figure 6.7. This parser verifies the type of its arguments, insures that the objects they represent actually exist, builds a standard representation of the final goal and returns it along with the appropriate subgoal templates. The `parser` will then pass these values on to the Main Planner.

6.6 Main Planner

The Main Planner brings all of the previous elements together. It is simply a finite state machine that drives the planning process, invoking the planning modules as necessary and thus controlling the flow of information between them. It also accumulates the successful partial plans into a complete plan. As such, it does not perform any analysis whatsoever beyond determining which planning module to call next.

The sequence of invocations of the various planning modules is determined by the subgoal template associated to each instruction. Consequently, there is no fixed

```
(defun translate ()
  (setf *plan* nil)
  (setf TLI task-level-program)
  (if (create-statement-p TLI)
      then (parse TLI)
      else (error "First instruction must be a create statement")))


(defun parse (TLI)
  (setf inst-verb (get-instruction-verb TLI))
  (if (member inst-verb inst-verb-list)
      then (funcall (concat 'parse- inst-verb) TLI))
      else (error "Non-existent task-level instruction specified")))


(defun parse-create (TLI)
  (setf list-of-instruction (reverse (get-create-body TLI))
  (setf name (get-name-of-object-to-create TLI))
  (when (exists-already-p name)
        (error name "is already defined"))
  (setf list-new-instructions nil)
  (setf list-new-instruction
        (cond ((special-instruction-p (first list-of-instructions))
               (cons (bind name (second list-of-instructions)))
               (setf list-of-instruction (rest list-of-instructions)))
              (t
               (cons (bind name (first list-of-instructions)))))
               list-new-instructions)
   (do ((list-of-instructions (rest list-of-instructions)
                              (rest list-of-instructions)))
       ((endp list-of-instructions) (setf list-new-instructions
                                          (reverse list-new-instructions))
       (setf list-new-instructions
             (cons (bind (gensym) (first list-of-instructions))
                   list-new-instructions)))
   (do ((list-of-instructions list-new-instructions
                              (rest list-of-instructions)))
       ((endp list-of-instructions))
       (parse (first list-of-instructions))))
```


Figure 6.8: Parser top-level LISP code: part 1

```
(defun parse-mate (TLI)
  (unless (mate-parameters-ok TLI)
          (error "Mate parameters are incorrect))
  (setf goal (build-goal-representation TLI))
  (add-goal goal mate-subgoal-template))
```

Figure 6.9: Parser top-level LISP code: part 2

planning algorithm coded into the planner.

A typical algorithm for an assembly statement not requiring a tool change is described below:

1. Call the Fine Motion Planner with the set of constraints that must be satisfied in the goal state. The Fine Motion Planner can either:

   (a) produce a sequence of fine motions to achieve the goal and return a region of space R_3 from which the sequence of motions is guaranteed to succeed under normal conditions,

   (b) produce a set of constraints—maybe none—on the configuration of the object, or

   (c) fail with or without giving reasons.

   The Fine Motion Planner can assume any configuration for the payload that produces solutions. These configurations constitute sets of constraints that will be passed on to the Grasp Planner. The selected grasp configuration is then used by the Gross Motion Planner to find a path.

   The Fine Motion Planner *must* memorize those plans it generated before so that when it is reinvoked, it doesn't repeat itself.

   If the Fine Motion Planner fails to find a solution, planning stops.

2. Call the Grasp Planner to generate a grasp configuration consistent with the fine motions generated in step 1. The Grasp Planner can either:

   (a) produce a grasp configuration, or

(b) fail with or without a set of constraints.

If none can be found, the Planner backs up to step 1 and generates another set of fine motions with, hopefully, a different set of grasping constraints. The constraints provided by the Grasp Planner could be used by the Fine Motion Planner to guide the search.

3. Call the Fine Motion Planner to generate a sequence of fine motions to grasp the object. The Fine Motion Planner can either

   (a) produce a sequence of fine motions to achieve the grasp and return a region of space R_1 from which the sequence of motions is guaranteed to succeed under normal conditions, or

   (b) fail with or without producing constraints.

   If the space around the object to be picked up is uncluttered, the problem is trivial; the Fine Motion Planner only needs to open the robot's gripper wide enough and move the end effector into position, possibly using only pure position control. However, the problem is really one of fine motion planning rather than one of gross motion planning; the geometry of the current task and final goal determines the motions rather than the risks of collisions. Furthermore, pure position control is a special case of a guarded motion, as defined in section 6.3, in which the termination predicate is a position.

   In case the Fine Motion Planner fails, the Planner backs up to step 2 and tries to find a different grasp configuration. Any constraints produced at this level may be used by the Grasp Planner to select a grasp configuration.

4. Call the Fine Motion Planner to generate a sequence of fine motions to take the grasped object away from its initial position. The Fine Motion Planner can either:

   (a) generate a sequence of fine motions away from the grasped object's initial position and produce a region of space R_2 from which the motion is guaranteed to succeed, or

(b) fail with or without producing constraints.

Again, the problem may be trivial if space is uncluttered. If the Fine Motion Planner fails, the Planner backs up to step 3. While in most cases this operation should succeed if the previous one did, it is possible to contrive situations where this is not true. One such situation occurs virtually everyday when one tries to retrieve the contents of a tight jean pocket; as long as one's hand clutches the content of the pocket, it cannot be removed! Hasegawa shows a few more realistic situations where similar problems could occur [27].

5. Call the Gross Motion Planner to generate collision free paths connecting the initial position of the robot, R_0, to R_1. If such a path cannot be found, back up to step 3

6. Call the Gross Motion Planner to generate collision free paths connecting R_2 to R_3. If such a path cannot be found, back up to step 4.

7. Call the Fine Motion Planner to ungrasp the held part. Again, this may be trivial if the surrounding space is uncluttered. This should not fail! If the robot and its payload did not bump into anything on the way to the goal, then the robot should be able to retrace its tracks. Again, pathological situations could be contrived where this does not hold. For example, the robot may hold a box's lid much as a waiter holds his tray, position the lid on the box, which provides a cutout for the arm, and then find that its gripper will not pass through the cutout. If such situations are possible, this step should become step 2, in which case a viable fine motion strategy would be devised before wasting time planning other steps by backtracking immediately to step 1.

8. Sort the resultant partial plan and add the corresponding TLI's to it. Each section of the partial plan must also be associated with information about the subgoal it is supposed to achieve.

9. Add the complete partial plan to the list of plans.

10. Update the World Model.

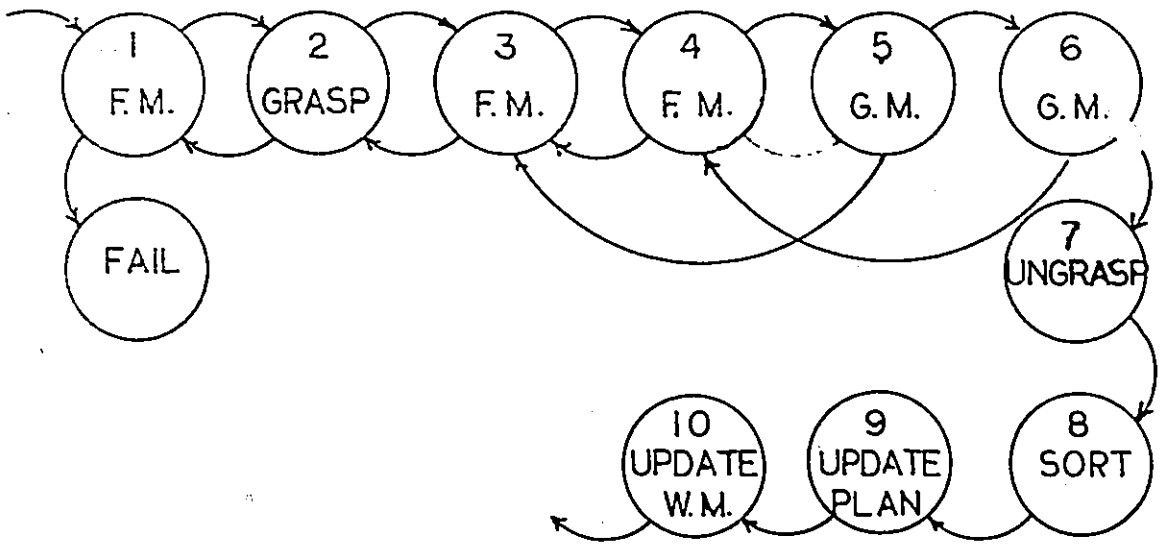

Figure 6.10: Finite state machine representation of planning algorithm outlined in the text

The corresponding finite state machine is illustrated in figure 6.10.

The above algorithm resembles PROLOG in its use of backtracking. To implement backtracking, all planning variables must be maintained in a special environment in order not to modify the World Model and to be able to undo previous planning steps. Such environments are easily implemented in LISP using association lists or a-list: lists consisting of pairs made up of a variable and its value. In fact, many LISP interpreters maintain their variables in such lists. By deleting the segments of the a-list corresponding to the work to undo, the previous values of variables are automatically restored. The implementation of PROLOG-like systems in LISP is described in [75]. A more general discussion can be found in [10].

The Planner differs from Prolog in that information is propagated both forwards and backwards as full sets of numeric and symbolic constraints, thus potentially reducing the search time drastically.

It can be deduced from the description in the text and from figure 6.10 that the entire planning process is driven by the goal, as it should be. Once a sequence of fine motions has been found that achieves the stated goal, the Planner looks for a compatible grasp configuration and for ways of grasping the component. These are activities that are difficult for human programmers. Finding collision free paths to

connect the various regions of space is secondary to fine motion planning and grasp planning, and can be easily accomplished by humans in uncluttered environments. In fact, the gross motion planning problem could be left to humans. By using a graphical simulator, an operator could devise paths which would be integrated into the executable robot program.

6.7 Conclusion

While Schwartz and Sharir have shown that the gross motion planning problem is solvable, the general algorithms proposed so far are still intractable for robots with six degrees of freedom. The introduction of more than one moving robot in the work cell increases the complexity of the problem exponentially in the number of added degrees of freedom. This fact alone supports The restriction imposed to a single robot in the cell. Furthermore, any mechanism (object with movable links) also increases the number of degrees of freedom of the problem and its complexity. Because of the inability to deal effectively with the added degrees of freedom brought about by mechanisms, representing and reasoning about them has been neglected so far in the literature. Given the inherent computational complexity of the problem, it is likely that humans will continue to be better than machines at solving the FindPath problems in the foreseeable future.

Effective generation of fine motions requires a fundamental understanding of the underlying physical processes involved in manipulating objects, as well as the means to deal with the uncertainties stemming from part tolerances and the finite accuracy of the manipulator. Given that research in this field began only in the last decade or so, it is not surprising that no general theory has yet emerged. What is a bit surprising is that most efforts have concentrated on solving the FindPath problem, which, as discussed in the previous section, can often be trivialized while the fine motion synthesis problem cannot. Perhaps the lack of basic analytical tools, and the need for some knowledge in the fields of statics and dynamics, computational geometry, control theory, computer science and artificial intelligence, coupled with the inherent complexity of the problem, have daunted more than a few researchers. What is certain is that much work remains to be done. With the exception of Requicha [56,57], no one, to the best knowledge of the author, has analyzed the

problem of representing tolerances on parts or reasoning about them, although both Taylor and Brooks used position tolerance data in their work [7].

Similarly, little work exists on the representation and interpretation of geometric constraints in assembly. Popplestone *et al.* in [54] described a way to extract expressions defining the positions of bodies in the goal state from a specification of that state using constraints similar to those proposed in the task-level language of chapter 3. Specifically, Popplestone could deal with plane faces, solid cylinders, cylindrical holes, spherical faces, edges and vertices, and the `against`, `fits` and `coplanar` constraints. Two entities are `against` one another if they just touch. A cylinder `fits` a hole if their longitudinal axes are parallel. Finally, two planar faces are `coplanar` if they lie in the same plane with their normals pointing in the same direction.

Grasp planning also hasn't attracted much attention. However, it seems that much of the work done in characterizing and reasoning about geometric and physical constraints in fine motion synthesis is directly applicable to grasp planning. After all, both problems deal with the analysis of geometries and robot dynamics.

From the previous exposé, it is obvious that the problems of planning for robotic assembly are far from having been solved. The proposed test-bed will allow further research to be performed, and developed algorithms and heuristics to be tested.

Chapter 7

Conclusion

This thesis investigated the feasibility of implementing a task-level programmed robotic assembly test-bed which allows experimentation with various planning algorithms, and described that implementation. The actual implementation of this test-bed was not the topic of the thesis, considering the large amount of work involved. However, the thesis did outline the implementation of the Planner and World Model and justified every decision made.

Further Research

The thesis revealed that, while much work has been done in computer graphics and CAD system design on representing solids, very little attention has been paid to the topic by researchers in robotics. Particularly neglected is the representation and interpretation of tolerancing information; with the exception of Taylor and Brooks who only manipulated tolerance values, no one, to the best knowledge of the author, has considered how to represent tolerance information in automatic reasoning systems.

As far as representing nominal solids is concerned, the MIT research groups appear to have used polygonal modeling of solids exclusively. Why this is so is not clear. However, since the time bound on planning algorithms is a polynomial function of the degree of the equation describing faces, the polygonal representation offers a definite advantage when modeling simple shapes since it uses only linear equations. Perhaps also their choice was motivated by the ability of their research

machines to display polygonal models and simulate assembly operations graphically. Whatever the reasons, since the time bound on the FindPath algorithm is also a polynomial function of the number of faces, when accurate modeling of curved surfaces is required, it is not certain that polygonal approximations are superior to other schemes. The effects of the increase in the number of planar faces over quadric or bicubic faces to model curved surfaces may well overshadow the advantage of the linear degree of their description. This is especially true when comparing quadrics of degree two to polygons of degree one (bicubics are of total degree six). The best approach might be to model solids with quadric surfaces, and to convert these quadrics into polygons when coarse approximations of solids are sufficient. Such an approach would combine the advantage of both representation schemes without increasing processing time substantially, given the complexity and time bound of the planning problem.

The thesis also showed that current algorithms to solve the FindPath problem still require enormous amounts of processing time to solve the problem in a cluttered 3-D space. There is therefore ample room for further research in this area, which only reinforces the need for a test-bed to actually try out these ideas, if only through simulation.

Much work remains to be done in the areas of fine motion synthesis and grasp planning, especially in the derivation, representation and interpretation of geometric and physical constraints. However, it is becoming increasingly clear that a true general-purpose task-level programmed robotic assembly system will be difficult to realize in the foreseeable future. Some scheme is required to reduce the complexity of the problem. One such approach would be to endow CAD systems with the ability to analyze the interactions of part geometries and robot capabilities during fine motions, thus driving the part design process in terms of the assembly tasks and the robots performing them, rather than driving the assembly tasks from the part designs. Information produced by the fine motion analyzer could then be used in the actual fine motion synthesis phase, thus restraining the search even more, possibly to a single sequence of fine motions.

Appendix A

Formal Set Definitions

This appendix formally defines some of the terms used in chapter 4. These definitions where collected from the following sources: [55], [65], [73], [38], [31] and [68], and re-written in a consistent manner.

1. Topologies and Open Sets

Let X be a non-empty set. A class (set of sets) $\mathcal{T}$ of subsets of X constitutes a *topology* on X if and only if $\mathcal{T}$ satisfies the following axioms:

O_1 $X \in \mathcal{T}$ and $\emptyset \in \mathcal{T}$.

O_2 The union of any number of sets in $\mathcal{T}$ belongs to $\mathcal{T}$.

O_3 The intersection of any finite number of sets in $\mathcal{T}$ belongs to $\mathcal{T}$.

The members of $\mathcal{T}$ are called *$\mathcal{T}$-open sets* or simply *open sets*. The pair $(X,\mathcal{T})$ constitutes a *topological space*.
If $\mathcal{T}$ is the only topology on X which is under consideration, then the shorter term "topological space X" can be used instead of the term "topological space $(X,\mathcal{T})$ without any ambiguity. Similarly, the term "open sets in X" can be used instead of "open sets in $(X,\mathcal{T})$. This also applies to closed sets.

Local characterization of open sets

The above definition constitutes a *global characterization* of open sets. To test whether a set is open or not, one must test the entire class T. A *local characterization* of open sets would allow one to test the set of interest only. That local characterization is:
Let X be a topological space. A subset A of X is open in X if and only if for each $x \in A$ there is some open set G_x such that $x \in G_x$ and $G_x \subset A$. In other words, A is open if and only if each of its points can be surrounded with an open set G_x which is itself contained in A.

2. Accumulation Points

Let X be a topological space. A point $x \in X$ is an *accumulation point* or *limit point* of a subset A of X if and only if every open set G containing x contains a point of A different from x. The set of accumulation points of A, denoted $\dot{A}$, is called the *derived set* of A. Note that x need not be an element of A.

3. Closed Sets

Let X be a topological space. A subset A of X is a *closed set* if and only if its complement A^c is an open set. The class of closed subsets of X have the following properties:

C_1 X and $\emptyset$ are closed sets.

C_2 The intersection of any number of closed sets is a closed set.

C_3 The union of any finite number of closed sets is a closed set.

A subset A of a topological space X can be locally characterized as closed if and only if A contains each of its accumulation points, i.e. $\dot{A} \subset A$.

4. Interior Points and Largest Open Sets

Let A be a subset of a topological space X. A point $x \in A$ is called an *interior point* of A if x belongs to an open set G contained in A, i.e. $x \in G \subset A$, where G

is open. The interior of A, the set of all interior points of A, is denoted $int(A)$ or A°. The $int(A)$ constitutes the *largest open set in A.*
An open set G is the largest open set of A if G is contained in A and whenever H is an open set contained is A then $H \subset G$.

5. Exterior Points

Let A be a subset of a topological space X. The *exterior* of A is the interior of the complement of A, i.e. $ext(A) = int(A^c)$.

6. Closure of a Set and Smallest Closed Sets

Let A be a subset of a topological space X. The closure of A, denoted $\overline{A}$, is the *smallest closed set of A.*
A closed set F is the smallest closed set of A if F contains A and whenever E is a closed set containing A, then $F \subset E$.

7. Boundary Points

Let A be a subset of a topological space X. The *boundary* of A, denoted $bd(A)$, is the set of points which do not belong to the interior or exterior of A. Also,

$$bd(A) = \overline{A} - A^\circ$$

8. Regular Spaces and Regular Sets

Let X be a topological space. X is a *regular* space if whenever F is a closed set in X and $x \in X - F$ there are open sets G and H in X such that $x \in G$, $F \subset H$, and $G \cap H = \emptyset$.
An open set G is a *open regular set* if $G = \overline{G}^\circ$.
A closed set H is a *closed regular set* if $H = \overline{H^\circ}$. This can be abbreviated to $H = rH$, where r is called the *regularization operator.*

9. Metrics

Let X be a non-empty set. A real-valued function d defined on $X \times X$ is called a *metric* or *distance function* on X if and only if it satisfies, for every $a, b, c \in X$, the following axioms:

$\mathbf{M_1}$ $d(a,b) \geq 0$ and $d(a,a) = 0$.

$\mathbf{M_2}$ $d(a,b) = d(b,a)$ (Symmetry).

$\mathbf{M_3}$ $d(a,c) =\leq d(a,b) + d(b,c)$ (Triangle Inequality).

$\mathbf{M_4}$ If $a \neq b$, then $d(a,b) > 0$.

10. Bounded set

The *diameter* of an non-empty subset A of X is denoted and defined by:

$$d(A) = \sup\{d(a,a') : a, a' \in A\}$$

The function *sup* returns the *least upper bound* or *supremum* of its argument. In this case, it returns the largest real-value of $d(a,a')$, i.e. the largest distance between two points of A.
If $d(A) < \infty$, then A is said to be *bounded*, otherwise, it is *unbounded*.

11. Semianalytic

A function $f : R^3 \mapsto R^1$ is *analytic* if $f(x,y,z)$ can be expanded into a convergent series about each point of its domain.
A set is *semianalytic* if it can be expressed as a finite Boolean combination of sets of the form

$$(x,y,z) : f_i(x,y,z) \leq 0$$

where the f_i are analytic, using the standard set operators $\cap, \cup$ and $-$.

Relation of definitions to Representation Schemes

1. When modeling a solid in three-dimensional Euclidean space R^3, we are obviously interested in the solids *interior* and in its surface or *boundary*, which have, in this special case, their "everyday" meaning. From the above definitions, it is clear that a solid must be modeled by a closed set, i.e. a set that includes its interior *and* boundary.

2. Furthermore, since each object is of finite size, bounded sets are to be used in modeling.

3. Regular sets prevent the representation of zero-, one- or two-dimensional objects, unless the point, line or plane is adjacent to the interior of a solid. In other words, the boundary of the solid *must* enclose the interior of the solid; dangling faces, edges, or points are not allowed. This follows from the definition of a closed regular set, $H = \overline{H^\circ}$, which states that such a set is equal to its interior plus the boundary of that interior. The definition does however allow references to object surfaces, edges or vertices.

Appendix B

ANSI Y14.5 Geometric Tolerances

This Appendix briefly defines the ANSI Y14.5 standard geometric tolerances. Its purpose is to clarify some notions for unfamiliar readers. It does not pretend to cover all aspects of the standard.

Geometric tolerances can be divided into three groups: position (or location), form, and runout. Some of these are intrinsic while others are extrinsic, i.e. they depend on some external reference plane, line or point called a datum. The various types of tolerances with their associated ANSI Y14.5 symbols are shown in figure B.1 and their interpretation defined in this Appendix. Note that size tolerances are not defined in the ANSI Y14.5 standard.

Size and some form tolerances are intrinsic: they depend only on the feature being toleranced. Size tolerances limit the variation in dimensions of a feature, while form tolerances constrain surface variations to lie within certain limits. Position tolerances, other form tolerances, and runout tolerances are extrinsic or relative to other features and require the specification of references or *datums*. Position tolerances define the acceptable variations in position *and* orientation of a feature, while runouts constrain the relationship of one or more components to a datum axis. Features controlled by runout tolerances include surfaces constructed around a datum axis or at right angles to a datum axis.

A datum is any coordinate system or reference plane, axis or point constructed from nominal faces, edges or vertices of a nominal object, that is used as a base for establishing the true shape of a part. *Measured entities* do not correspond to features but are computed from them. For example, the central axis of a cylinder

		Characteristic	Symbol
Individual Features	Form Tolerances	Straightness	—
		Flatness	▱
		Roundness	○
		Cylindricity	⌭
Individual or Related Features		Profile of a line	⌒
		Profile of a surface	⌓
Related Features		Angularity	∠
		Perpendicularity	⊥
		Parallelism	∥
	Location Tolerances	Position	⌖
		Concentricity	◎
		Symmetry	⌯
	Runout Tolerances	Circular	↗
		Total	⌰

Figure B.1: ANSI Y14.5 geometric tolerances and associated symbols

is derived from its nominal diameter instead of being specified directly. Measured centerplanes, axes, and centers are associated with symmetric features and can be used as datums to locate the symmetric features, such as cylindrical holes. In such cases, the symmetry axis of the feature is positioned relative to some other datum and the feature itself positioned relative to its measured entity.

In ANSI Y14.5, nominal values are written in rectangular boxes without any other symbols, while tolerances are indicated by boxes containing, from left to right, the geometric tolerance's symbol, the list of reference datums, if any, in order of importance, and the value of the tolerance. Datums are identified by boxes containing uppercase letters only, usually from A to Z and from AA to ZZ if more are required, attached to the datum feature.

The following sections define all of the ANSI Y14.5 geometric tolerances, illustrating some of the most important ones for robotics.

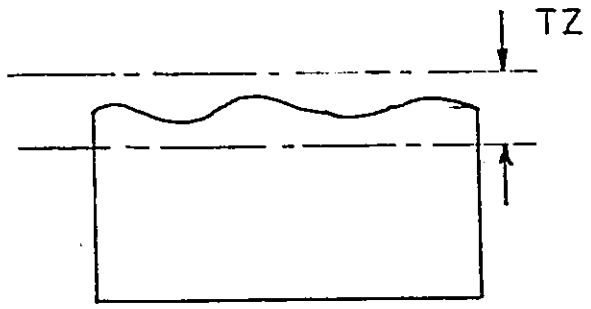

Figure B.2: Flatness

B.1 Intrinsic Form Tolerances

Intrinsic form tolerances constrain the shape of a surface without any reference to external datums.

B.1.1 Flatness

Flatness refers to the condition of a surface where all of the points that make up the surface lie in a single plane.

A flatness tolerance defines a tolerance zone consisting of two parallel planes separated by a distance equal to the specified tolerance between which the entire surface must lie. See figure B.2.

B.1.2 Straightness

Straightness refers to the condition in which the longitudinal elements of a surface lie in straight lines.

A straightness tolerance specifies a tolerance zone of uniform thickness along a straight line, within which all points of the considered line must lie. See figure B.3.

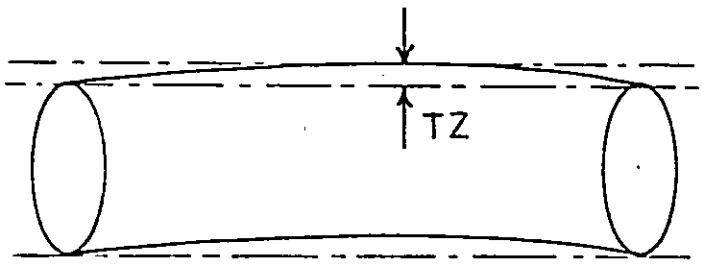

Figure B.3: Straightness

B.1.3 Roundness

Roundness refers to the condition of a surface of revolution such as a cylinder, cone, or sphere, where all points of the surface intersected by any plane perpendicular to a symmetry axis (cylinder, cone) or passing through a common center (sphere) form a circle.

A roundness tolerance specifies a tolerance zone consisting of two concentric circles separated by a distance equal to the specified tolerance in which every point in any cross section perpendicular to a reference axis must lie. See figure B.4.

B.1.4 Cylindricity

Cylindricity refers to the condition of a surface of revolution in which a solid figure, described by the edge of a rectangle, is rotated around its parallel edge and this edge becomes the axis of the formed cylinder.

A cylindricity tolerance specifies a tolerance zone consisting of two concentric cylinders separated by a distance equal to the tolerance specified between which the surface must lie. See figure B.5.

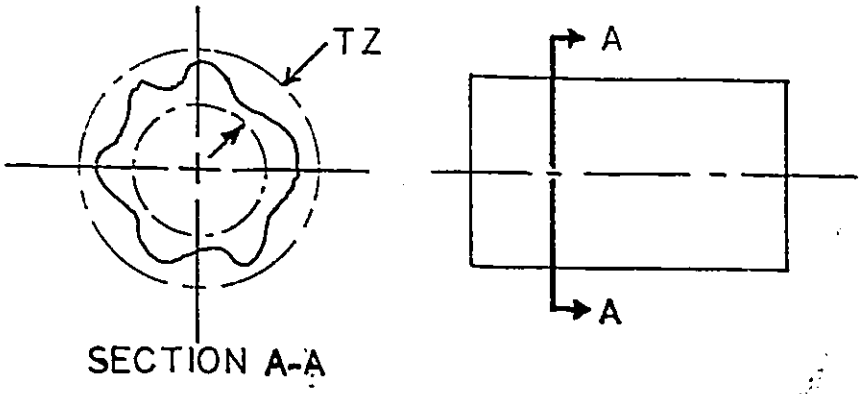

Figure B.4: Roundness

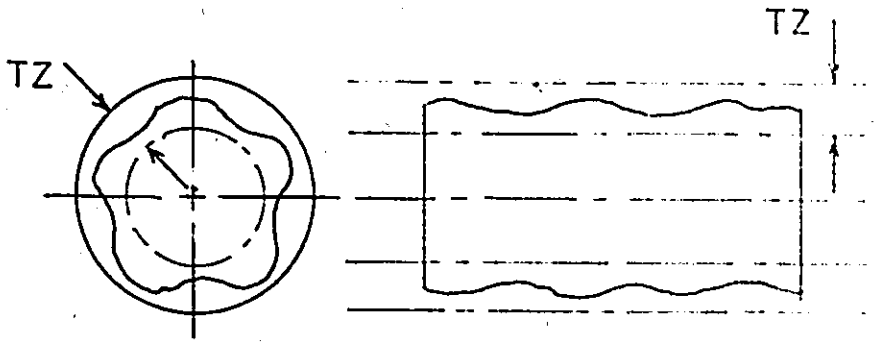

Figure B.5: Cylindricity

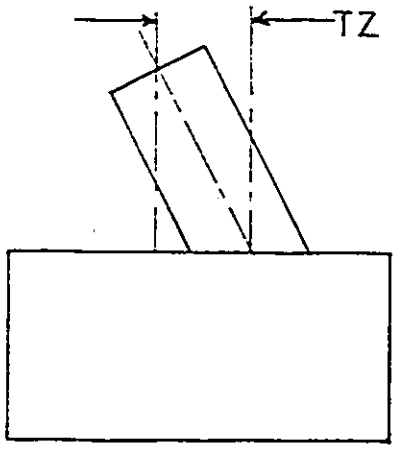

Figure B.6: Perpendicularity

B.2 Extrinsic Form Tolerances

Extrinsic form tolerances define the tolerance on the orientation of one feature relative to another.

B.2.1 Perpendicularity

Perpendicularity refers to the condition of a surface, axis, or line which lies at right angle to a datum plane or axis.

A perpendicularity tolerance applied to a surface with respect to a datum plane or axis defines a tolerance zone consisting of two parallel planes perpendicular to the datum plane or datum axis and separated by a distance equal to the specified tolerance between which the surface must lie.

A perpendicularity tolerance applied to an axis with respect to a datum plane defines a cylindrical tolerance zone perpendicular to the datum plane whose radius is equal to the specified tolerance in which the axis must lie. See figure B.6.

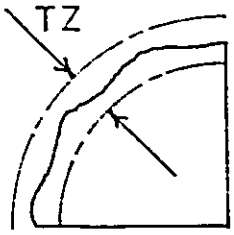

Figure B.7: Profile

B.2.2 Angularity

Angularity refers to the condition of a surface which lies at an angle other than 90° to a datum plane or axis.

An angularity tolerance defines a tolerance zone consisting of two parallel planes lying at the specified angle relative to the datum plane or axis and separated by a distance equal to the specified tolerance. See perpendicularity.

B.2.3 Parallelism

Parallelism refers to the condition of a surface, axis, or line which lies parallel from a datum plane or axis.

A parallelism tolerance applied to a surface with respect to a datum plane defines a tolerance zone consisting of two parallel planes parallel to the datum plane and separated by a distance equal to the specified tolerance between which the surface must lie.

A parallelism tolerance applied to an axis with respect to a datum axis defines a cylindrical tolerance zone parallel to the datum plane whose radius is equal to the specified tolerance in which the axis must lie.

B.2.4 Profile

Profile refers to the amount of variation that can be tolerated along a line or surface.

A profile tolerance defines a tolerance zone of width equal to the specified tolerance always measured normal to the surface of line within which the surface or line must lie. See figure B.7.

B.3 Runout tolerances

B.3.1 Runout

Circular runout refers to the condition of perfect form and axial alignment of two or more circular elements of surfaces of revolution such as cylinders, cones, or contours generated about a common axis.

A circular runout tolerance controls the relationship of two or more features along a circular contour within the allowable errors of concentricity, perpendicularity, and alignment of the features. It also controls variations in roundness, straightness, flatness, angularity, and parallelism of individual surfaces. In essence, circular runout establishes composite-form control of those features of a part having a common axis.

B.3.2 Total Runout

Total runout refers to the condition of perfect form and axial alignment of two or more surfaces of revolution such as cylinders, cones, or contours generated about a common axis.

A total runout tolerance controls the relationship of two or more features within the allowable errors of concentricity, perpendicularity, and alignment of the features. It also controls variations in roundness, straightness, flatness, angularity, and parallelism of individual surfaces. In essence, total runout establishes composite-form control of those features of a part having a common axis.

Total runout is usually implied if "total" or "circular" is not specified. While circular runout applies to circular elements cut out of volumes of revolution by a plane a right angle to a datum axis, total runout applies to the entire surface.

B.4 Position Tolerances

Position tolerances constrain the position and orientation—angularity—of surfaces, centerplanes, or center axes relative to other features. Position tolerances are divided into MMC, RFS, concentricity and symmetry.

B.4.1 MMC Position Tolerances

MMC stands for Maximum Material Condition and refers to the limits of size of a feature that leads to the use of the maximum amount of material. In the case of a solid, MMC refers to the maximum allowed dimensions of the solid, while for a hole it refers to the minimum allowed dimensions. The opposite of MMC is LMC which stands for Least Material Condition (LMC) and refers to the limits of size of a feature that leads to the use of the minimum amount of material.

An MMC position tolerance applied to a volume of revolution such as a cylinder or cone defines a cylindrical tolerance zone whose *minimum* radius is equal to the specified tolerance in which the symmetry axis of the feature must lie.

An MMC position tolerance applied to a non-cylindrical feature defines two parallel planes separated by a *minimum* distance equal to the specified tolerance in which the centerplane of the feature must lie.

MMC position tolerances are used to constrain the position of mating components such as pin-hole or tab-slot pairs. As the dimensions of the feature goes from MMC to LMC, i.e. as the pin or tab gets smaller, or the hole or slot gets larger, the error on position can be made greater while still maintaining sufficient clearing for insertion. For instance, if a hole is found to be 0.001 inch larger than the minimum tolerated, its position tolerance increases by 0.001 inch.

B.4.2 RFS Position Tolerances

RFS stands for Regardless of Feature Size. An RFS position tolerance applied to a volume of revolution such as a cylinder, cone, or sphere defines a cylindrical tolerance zone whose *fixed* radius is equal to the specified tolerance in which the symmetry axis of the feature must lie.

An RFS position tolerance applied to a non-cylindrical feature defines two parallel planes separated by a *fixed* distance equal to the specified tolerance in which the centerplane of the feature must lie.

B.4.3 Composite Position Tolerances

Composite position tolerances are used to position patterns of features, such as holes equally spaced on a circle. Two position tolerances are then specified: the first locating the pattern itself while the second locates the individual features composing the pattern.

B.4.4 Concentricity

Concentricity refers to the condition of surfaces of revolution, such as cylinders, cones, or spheres, wherein they have a common axis of symmetry.

A concentricity tolerance defines a cylindrical tolerance zone whose radius is equal to the specified tolerance within which the axis of symmetry of each feature must lie.

B.4.5 Symmetry

Symmetry refers to the condition in which a part or a feature has the same contour and size on opposite sides of a central plane or a condition in which a feature in symmetrically disposed about the central plane of a datum feature.

A symmetry tolerance defines two parallel planes equally disposed about the datum plane and separated by a distance equal to the specified tolerance in which the centerplane must lie.

Appendix C

Sample World Model

```
;;; Code to manipulate the World Model and describe it.


;; Need tabs because the ~t directive of the format function hasn't been
;; implemented. two-pi is used to convert degrees into radians. They are
;; declared special. :all and :nominal are flags bound to themselves. Identity
;; is useful and saves time.

(defvar $tab$ (ascii 9))
(defvar two-pi (* 8 (atan 1)))
(defvar :all ':all)
(defvar :nominal ':nominal)
(defvar identity (make-identity 4))

;; Functions to move an object through space

(defun move (obj x-pos y-pos z-pos x-rot y-rot z-rot)
  ;; Move an object by changing its configuration matrix
  (setf (solid-configuration obj) (mult (make-hctm x-pos y-pos z-pos
                                                   x-rot y-rot z-rot)
                                        (solid-configuration obj))))


(defun make-hctm (x-pos y-pos z-pos x-rot y-rot z-rot)
  ;; This function builds a homogeneous coordinate transform matrix out its
  ;; parameters. It performs the x rotation, then the y rotation, then the
  ;; z rotation, then the translation.
  (mult (translate x-pos y-pos z-pos)
        (rotate 'Z z-rot)
        (rotate 'Y y-rot)
        (rotate 'X x-rot)))


;; Rotate and translate are used to generate homogeneous coordinate transform
;; matrices. They are used by the move function.

(defun rotate (axis angle)
  ;; This function generates a homogeneous coordinate transform matrix for a
  ;; single rotation about the x, y or z axis, where the angle is expressed in
  ;; degrees.

  ;; Change angle from degrees to radians
  (let ((angle (times (quotient angle 45) (atan 1))))
    (caseq axis
        (X (make-array '(4 4) :initial-contents
                       (list '(1.0  0.0  0.0  0.0)
                             (list  0.0 (cos angle) (- (sin angle)) 0.0)
                             (list  0.0 (sin angle)    (cos angle)  0.0)
                             '(0.0  0.0  0.0  1.0))))
        (Y (make-array '(4 4) :initial-contents
                       (list (list (cos angle) 0.0 (sin angle) 0.0)
                             '(0.0  1.0  0.0  0.0)
                             (list (- (sin angle)) 0.0 (cos angle) 0.0)
                             '(0.0  0.0  0.0  1.0))))
        (Z (make-array '(4 4) :initial-contents
                       (list (list (cos angle) (- (sin angle)) 0.0  0.0)
                             (list (sin angle)    (cos angle)  0.0  0.0)
                             '(0.0  0.0  1.0  0.0)
                             '(0.0  0.0  0.0  1.0))))
        (t (error "Unknown axis:" axis)))))
```

```
(defun translate (x y z)
  ;; This function generates a homogeneous coordinate transform matrix for a
  ;; translation along the x, y, and z axes.
  (if (and (floatp x)
           (floatp y)
           (floatp z))
      then (make-array '(4 4) :initial-contents
                       (list (list 1.0 0.0 0.0 x)
                             (list 0.0 1.0 0.0 y)
                             (list 0.0 0.0 1.0 z)
                                '(0.0 0.0 0.0 1.0)))
      else (error "Arguments to translate must be flonums:" x y z)))


;; These functions create new objects.

(defun rigid-attach (new-obj obj1 obj2)
  ;; This function creates a new object with name new-obj which results from
  ;; the rigid attachment of obj1 and obj2 in their specified position.
  ;; To keep the result, don't forget to setf a symbol to it.

  ;; Verify that obj1 and obj2 are solids and new-obj a symbol
  (unless (and (symbolp new-obj)
               (solid-p obj1)
               (solid-p obj2))
          (error "First argument to rigid-attach must be a symbol"
                 "Two remaining arguments must be solids"))

  ;; Verify that obj1 and obj2 are solids that touch without intersecting.
  (unless (just-touching obj1 obj2)
          (error "Attempting tc attach to disjoint or intersecting solids"))

  ;; If o.k. make the new object. The object type depends on the type of the
  ;; component objects and their relationship (rigid). If any object is an
  ;; aggregate, resultant object must be an aggregate. If any object is a
  ;; mechanism (but not an aggregate), resultant object must be a mechanism.
  ;; Otherwise, resultant object is a compound object.

  (cond ((or (aggregate-p obj1)
             (aggregate-p obj2))
         (make-new-aggregate new-obj obj1 obj2))
        ((or (mechanism-p obj1)
             (mechanism-p obj2))
         (make-new-mechanism new-obj obj1 obj2))
        (t
         ;; Make a new compound object and update the part-of field of the two
         ;; constituent objects to point to the new one. Return the new object.
         (progl
           (make-new-compound new-obj obj1 obj2)
           (setf (solid-part-of obj1) new-obj)
           (setf (solid-part-of obj2) new-obj)))))
```

```
(defun make-new-compound (new-obj obj1 obj2)
  ;; This function makes a compund object from two other objects which can be
  ;; compounds or primitives. It has to be a macro if new-obj's is to retain
  ;; its new value outside of the function.
  (make-compound
   :name                new-obj
   :type                'compound
   :geometric-class     (get-geometric-class  new-obj)
   :functional-class    (get-functional-class new-obj)
   :configuration       (make-identity 4)
   :part-of             nil
   :mass                (compute-mass  obj1 obj2)
   :center-of-mass      (compute-c-o-m obj1 obj2)
   :rotational-inertia  (compute-r-i obj1 obj2)
   :constituent-objects (list 'rigid
                              (solid-name obj1)
                              (solid-type obj1)
                              (solid-name obj2)
                              (solid-type obj2))))


(defun make-new-mechanism (new-obj obj1 obj2)
  ;; This function makes a mechanism. It hasn't been implemented yet!
  (error "make-new-mechanism not yet implemented"))


(defun make-new-aggregate (new-obj obj1 obj2)
  ;; This function makes an aggregate. It hasn't been implemented yet!
  (error "make-new-aggregate not yet implemented"))


;; These functions compute some of the properties of objects

(defun get-geometric-class (new-obj)
  ;; This function returns the geometric class of an object based on
  ;; information supplied by the Planner stored in the symbol property list, or
  ;; based on an analysis of the geometric classes of the constituent objects.
  ;; Currently, if the Planner hasn't specified anything, it returns "general".
  (cond ((get new-obj 'geometric-class))
        (t 'general)))

(defun get-functional-class (new-obj)
  ;; This function returns the functioanl class of an object based on
  ;; information supplied by the Planner stored in the symbol property list.
  ;; Currently, if the Planner hasn't specified anything, it returns "unknown".
  (cond ((get new-obj 'functional-class))
        (t 'unknown)))

(defun compute-mass (obj1 obj2)
  ;; Return the sum of the masses of obj1 and obj2, a scalar.
  (sum (solid-mass obj1) (solid-mass obj2)))
```

```
(defun compute-c-o-m (obj1 obj2)
  ;; Return the center of mass of obj1 and obj2, a 4 x 1 vector.
  (let ((com1 (firstn 3 (listarray (solid-center-of-mass obj1))))
        (com2 (firstn 3 (listarray (solid-center-of-mass obj2))))
        (m1 (solid-mass obj1))
        (m2 (solid-mass obj2))
        (m  (compute-mass obj1 obj2))
        (com (make-array '(4 1))))
    ;; Watch this...
    (fillarray com (append (mapcar '(lambda (x) (quotient x m))
                                   (mapcar 'sum
                                           (mapcar '(lambda (x) (times x m1))
                                                   com1)
                                           (mapcar '(lambda (x) (times x m2))
                                                   com2)))
                           '(1)))
    ;; Get it
    com))

(defun compute-r-i (obj1 obj2)
  ;; Return the rotational inertia of ob1 and obj2.
  ;; Implement it if enough time...
  ;; Right now, just return a vector of "dummy" values.
     (make-array '(4 1) :initial-contents '((0.0) (0.0) (0.0) (1.0))))


(defun get-configuration (obj)
  ;; This function returns a homogeneous coordinate transform matrix that
  ;; represents the configuration of the object.

  ;; Build a list of all the configuration matrices from the root node down to
  ;; obj, then multiply them all together to get the position of obj in the
  ;; world coordinate system. Faces and Edges don't have a coordinate transform
  ;; matrix
  (do ((part-of (part-of obj) (part-of (eval part-of)))
       (list-of-hctm (list (obj-configuration obj))))
      ((null part-of) (eval (cons 'mult list-of-hctm)))
      (setq list-of-hctm (cons (obj-configuration (eval part-of))
                               list-of-hctm))))


(defun part-of (obj)
  ;; This returns the part-of field of any object. It just selects the
  ;; appropriate procedure (where's object-oriented stuff when you need it...)
  ;; In the case of an edge, one of its parent faces is selected.
  (cond ((solid-p obj) (solid-part-of obj))
        ((face-p obj)  (face-part-of obj))
        ((edge-p obj)  (first (edge-part-of obj)))
        (t (error "Unknown object type to part-of"))))


(defun obj-configuration (obj)
  ;; This is another kludge to get object-oriented type methods. It selects the
  ;; appropriate value to return in terms of the type of the argument.
  (cond ((solid-p obj) (solid-configuration obj))
        ((face-p obj)  identity)
        ((edge-p obj)  identity)))
```

```
(defun solid-center-of-mass (obj)
  ;; Another kludge around the lack of methods...
  (cond ((primitive-p obj)
         (mult (get-configuration obj) (primitive-center-of-mass obj)))
        ((compound-p obj)
         (mul (get-configuration obj) (compound-center-of-mass obj)))
        (t
         (error "That object doen't have a center of mass"))))


(defun just-touching (obj1 obj2)
  ;; This function verifies that two objects are just in contact.
  ;; Implement it if there is enough time.
  t)


(defun primitive-list-of-edges (prim)
  ;; This function return a list of the edges of the primitive without any
  ;; repetition.
  (do ((faces (primitive-list-of-faces prim) (rest faces))
       (list-of-edges nil))
      ((endp faces) list-of-edges)
      (do ((edges (face-list-of-edges (eval (first faces))) (rest edges)))
          ((endp edges))
          (pushnew (first edges) list-of-edges))))

;; These two are useful until the appropriate structures are defined.

(defun aggregate-p (obj)
  nil)


(defun mechanism-p (obj)
  nil)


;; These functions are used to output the description of objects in readable
;; format.

(defun describe-edge (edge hctm)
  ;; Describe the name, type, geometry, position and rotation of an edge
  ;; ~T control not implemented in format! Use the special variable $tab$ to
  ;; move the cursor to the next tab stop. ~a must be used to output the tab
  (unless (edge-p edge)
          (error "Parameter to describe edge must be an edge"))
  (describe-edge-common edge)
  (caseq (edge-type edge)
         (line (describe-line edge hctm))
         (circle (describe-circle edge hctm))
         (t
          (format t "~%~% Unknown edge type: ~a" edge)
          (format t "~%~%"))))
```

```
(defun describe-edge-common (edge)
  (format t "~%~a~aDescription of edge: ~a" $tab$ $tab$ (edge-name edge))
  (format t "~%~a~a name: ~a~a~a~a" $tab$
          $tab$ $tab$ $tab$ $tab$ (edge-name edge))
  (format t "~%~a~a type: ~a~a~a~a" $tab$
          $tab$ $tab$ $tab$ $tab$ (edge-type edge))
  (format t "~%~a~a part of: ~a~a~a"
          $tab$ $tab$ $tab$ $tab$ (edge-part-of edge)))

(defun describe-line (line hctm)
  (let ((end-point-1 (listarray (mult hctm (line-endpoint1
                                            line))))
        (end-point-2 (listarray (mult hctm (line-endpoint2
                                            line)))))
    (format t "~%~a~a end point 1 is at:" $tab$ $tab$)
    (format t "~%~a~a~a x-position: ~a~a"
            $tab$ $tab$ $tab$ $tab$ (first end-point-1))
    (format t "~%~a~a~a y-position: ~a~a"
            $tab$ $tab$ $tab$ $tab$ (second end-point-1))
    (format t "~%~a~a~a z-position: ~a~a"
            $tab$ $tab$ $tab$ $tab$ (third end-point-1))
    (format t "~%~a~a end point 2 is at:" $tab$ $tab$)
    (format t "~%~a~a~a x-position: ~a~a"
            $tab$ $tab$ $tab$ $tab$ (first end-point-2))
    (format t "~%~a~a~a y-position: ~a~a"
            $tab$ $tab$ $tab$ $tab$ (second end-point-2))
    (format t "~%~a~a~a z-position: ~a~a ~%"
            $tab$ $tab$ $tab$ $tab$ (third end-point-2))))


(defun describe-circle (circle hctm)
  (let ((center-point (listarray (mult hctm (circle-center-point edge))))
        (point1 (listarray (mult hctm (funcall (circle-parametric-description
                                                circle) 0))))
        (point2 (listarray (mult hctm (funcall (circle-parametric-description
                                                circle) .5)))))
    (format t "~%~a~a circle radius is: ~a~a"
            $tab$ $tab$ $tab$ (circle-radius edge))
    (format t "~%~a~a circle center is at:" $tab$ $tab$ )
    (format t "~%~a~a~a x-position: ~a~a"
            $tab$ $tab$ $tab$ $tab$ (first center-point))
    (format t "~%~a~a~a y-position: ~a~a"
            $tab$ $tab$ $tab$ $tab$ (second center-point))
    (format t "~%~a~a~a z-position: ~a~a"
            $tab$ $tab$ $tab$ $tab$ (third center-point))
    (format t "~%~a~a circle passes through ~a~a"
            $tab$ $tab$ $tab$ (firstn 3 point1))
    (format t "~%~a~a~a~a~a~a ~%"
            $tab$ $tab$ $tab$ $tab$ $tab$ (firstn 3 point2))))


(defun describe-face (face hctm)
  (unless (face-p face)
          (error "Argument to describe-face must be a face"))
  (describe-face-common face)
  (caseq (face-type face)
         (cylinder  (describe-cylinder face hctm))
         (disk      (describe-disk face hctm))
         (rectangle (describe-rectangle face hctm))
         (t (format t "~%~% Unknown face type ~%~%"))))
```

```
(defun describe-face-common (face)
  (format t "~%~aDescription of face: ~a" $tab$ (face-name face))
  (format t "~%~a~a name: ~a~a~a~a"
          $tab$ $tab$ $tab$ $tab$ $tab$ (face-name face))
  (format t "~%~a~a type: ~a~a~a~a"
          $tab$ $tab$ $tab$ $tab$ $tab$ (face-type face)))


(defun describe-rectangle (rect hctm)
  (format t "~%~a~a length: ~a~a~a"
          $tab$ $tab$ $tab$ $tab$ (rectangle-length rect))
  (format t "~%~a~a width: ~a~a~a"
          $tab$ $tab$ $tab$ $tab$ (rectangle-width rect))
  (describe-analytic rect hctm))


(defun describe-disk (disk hctm)
  (format t "~%~a~a radius ~a~a~a"
          $tab$ $tab$ $tab$ $tab$ (disk-radius disk))
  (format t "~%~a~a center-point ~a~a~a"
          $tab$ $tab$ $tab$ $tab$
          (firstn 3 (listarray (mult hctm (disk-center-point disk)))))
  (describe-analytic disk hctm))


(defun describe-cylinder (cyl hctm)
  (format t "~%~a~a height: ~a~a~a"
          $tab$ $tab$ $tab$ $tab$ (cylinder-length cyl))
  (format t "~%~a~a radius: ~a~a~a"
          $tab$ $tab$ $tab$ $tab$ (cylinder-radius cyl))
  (describe-analytic cyl hctm))


(defun describe-analytic (face hctm)
  (let* ((invhctm (inverse hctm))
         (new-description (listarray (mult (transpose invhctm)
                                           (face-analytic-description face)
                                           invhctm))))
    (format t "~%~a~a analytic description:  A = ~a" $tab$ $tab$
            (nth 0 new-description))
    (format t "~%~a~a~a~a~aB = ~a"
            $tab$ $tab$ $tab$ $tab$ $tab$ (nth 5 new-description))
    (format t "~%~a~a~a~a~aC = ~a"
            $tab$ $tab$ $tab$ $tab$ $tab$ (nth 10 new-description))
    (format t "~%~a~a~a~a~aF = ~a"
            $tab$ $tab$ $tab$ $tab$ $tab$ (nth 1 new-description))
    (format t "~%~a~a~a~a~aG = ~a"
            $tab$ $tab$ $tab$ $tab$ $tab$ (nth 2 new-description))
    (format t "~%~a~a~a~a~aH = ~a"
            $tab$ $tab$ $tab$ $tab$ $tab$ (nth 6 new-description))
    (format t "~%~a~a~a~a~aP = ~a"
            $tab$ $tab$ $tab$ $tab$ $tab$ (nth 3 new-description))
    (format t "~%~a~a~a~a~aQ = ~a"
            $tab$ $tab$ $tab$ $tab$ $tab$ (nth 7 new-description))
    (format t "~%~a~a~a~a~aR = ~a"
            $tab$ $tab$ $tab$ $tab$ $tab$ (nth 11 new-description))
    (format t "~%~a~a~a~a~aD = ~a"
            $tab$ $tab$ $tab$ $tab$ $tab$ (nth 15 new-description))
    (format t "~%~a~a in-out-flag: ~a~a~a"
            $tab$ $tab$ $tab$ $tab$ (face-in-out-flag face))
```

```
    (format t "~%~a~a is part of: ~a~a~a"
            $tab$ $tab$ $tab$ $tab$ (face-part-of face))
    (format t "~%~a~a has vface: ~a~a~a"
            $tab$ $tab$ $tab$ $tab$ (face-vface face))
    (format t "~%~a~a the face has ~a edges:~a~a~a"
            $tab$ (length (face-list-of-edges face))
            $tab$ $tab$ (first (face-list-of-edges face)))
    (print-list (rest (face-list-of-edges face)))
    (terpri)))


(defun describe-solid (solid hctm)
  ;; Print out a description of a solid
  (unless (solid-p solid)
          (error "Argument to describe-solid must be a solid"))
  (describe-solid-common solid)
  (caseq (solid-type solid)
         (primitive (describe-primitive solid hctm))
         (compound  (describe-compound solid hctm))
         (t (error "That type of object is not supported yet"))))


(defun describe-solid-common (solid)
  (format t "~%Description of solid: ~a" (solid-name solid))
  (format t "~%~a~a name: ~a~a~a~a"
          $tab$ $tab$ $tab$ $tab$ $tab$ (solid-name solid))
  (format t "~%~a~a type: ~a~a~a~a"
          $tab$ $tab$ $tab$ $tab$ $tab$ (solid-type solid))
  (format t "~%~a~a geometric class: ~a~a"
          $tab$ $tab$ $tab$ (solid-geometric-class solid))
  (format t "~%~a~a functional class: ~a~a"
          $tab$ $tab$ $tab$ (solid-functional-class solid))
  (format t "~%~a~a part of: ~a~a~a"
          $tab$ $tab$ $tab$ $tab$ (solid-part-of solid))
  (format t "~%~a~a mass: ~a~a~a~a"
          $tab$ $tab$ $tab$ $tab$ $tab$ (solid-mass solid)))


(defun describe-primitive (prim hctm)
  (let ((center-of-mass
         (listarray (mult hctm (primitive-center-of-mass prim)))))
    ;; Put stuff here to compute rotational inertia
    (format t "~%~a~a center of mass:" $tab$ $tab$)
    (format t "~%~a~a~a x-coordinate: ~a~a"
            $tab$ $tab$ $tab$ $tab$ (first center-of-mass))
    (format t "~%~a~a~a y-coordinate: ~a~a"
            $tab$ $tab$ $tab$ $tab$ (second center-of-mass))
    (format t "~%~a~a~a z-coordinate: ~a~a"
            $tab$ $tab$ $tab$ $tab$ (third center-of-mass))
    (format t "~%~a~a list of faces: ~a~a"
            $tab$ $tab$ $tab$
            (first (primitive-list-of-faces prim)))
    (print-list (rest (primitive-list-of-faces prim)))
    (terpri)))

(defun print-list (list)
  ;; This function prints out a list of elements, one per line.
  (do ((list list (rest list)))
      ((endp list))
      (format t "~%~a~a~a~a~a~a~a"
              $tab$ $tab$ $tab$ $tab$ $tab$ (first list))))
```

```
(defun describe-compound (comp hctm)
  (let ((center-of-mass (listarray (mult hctm (compound-center-of-mass
                                                solid)))))
    ;; Put stuff here to compute rotational inertia
    (format t "~%~a~a center of mass:" $tab$ $tab$)
    (format t "~%~a~a~a x-coordinate: ~a~a"
            $tab$ $tab$ $tab$ $tab$ (first center-of-mass))
    (format t "~%~a~a~a y-coordinate: ~a~a"
            $tab$ $tab$ $tab$ $tab$ (second center-of-mass))
    (format t "~%~a~a~a z-coordinate: ~a~a"
            $tab$ $tab$ $tab$ $tab$ (third center-of-mass))
    (format t "~%~a~a constituent objects: ~a~a~%"
            $tab$ $tab$ $tab$
            (compound-constituent-objects solid))))

(defun describe (obj &rest flags)
  ;; This function prints a description of an object, and if all is non nil, a
  ;; description of all its components. If the :all flag is specified, print
  ;; the description of every constituent object. If :nominal is specified,
  ;; print the description of the object in its local coordinate system, else
  ;; print its description in the world coordinate system.

  ;; Verify that object is valid
  (unless (or (solid-p obj)
              (face-p obj)
              (edge-p obj))
          (error "Unknown object type to describe"))
  ;; Get flag values
  (let* ((all (first (memq :all flags)))
         (nom (first (memq :nominal flags)))
         (hctm (if nom
                   then identity
                   else (get-configuration obj))))
    (cond ((solid-p obj)
           (describe-solid obj hctm)
           (when all (cond ((compound-p obj)
                            ;; Describe every solid making up the compound
                            ;; object
                            (do ((constituent-objects
                                  (list
                                   (second (compound-constituent-objects obj))
                                   (fourth (compound-constituent-objects obj)))
                                  (rest constituent-objects)))
                                ((endp constituent-objects))
                                (describe (eval (first constituent-objects))
                                          all nom)))

                           ((primitive-p obj)

                            ;; Describe every face making up the solid
                            (do ((list-of-faces (primitive-list-of-faces obj)
                                                (rest list-of-faces)))
                                ((endp list-of-faces))
                                (describe-face (eval (first list-of-faces))
                                               hctm))
                            ;; Describe every edge making up the solid
                            (do ((list-of-edges (primitive-list-of-edges obj)
                                                (rest list-of-edges)))
                                ((endp list-of-edges))
                                (describe-edge (eval (first list-of-edges))
                                               hctm))))))))
```

```
          ((face-p obj)
           (describe-face obj hctm)
           (when all
                 ;; Print out the description of every edge making up the face
               (do ((edges (face-list-of-edges obj) (rest edges)))
                   ((endp edges) t)
                   (describe-edge (eval (first edges)) hctm))))

          ((edge-p obj)
           (describe-edge obj hctm)))))

;; Definition of structures of the World Model

(defun edge-p (edge)
  (or (line-p edge)
      (circle-p edge)))

(defstruct (edge (:predicate nil))
  name                                  ;name of the edge
  type                                  ;type of the line
  part-of)                              ;list of faces the edge belongs to


(defstruct (line (:include edge))
  ;; geometric description of a line
  endpoint1                             ;4 x 1 vector locating one end point
  endpoint2                             ;4 x 1 vector locating other end point
  ;; parametric description
  parametric-description)               ;function of one variable that returns
                                        ;a 4 x 1 vector of points on the line

(defstruct (circle (:include edge))
  ;; geometric description of a circle
  center-point                          ;4 x 1 vector
  radius                                ;4 x 1 vector
  ;; parametric equation of the circle
  parametric-description)               ;function of one variable that returns
                                        ;4 x 1 vector of points on the circle

(defun face-p (face)                    ;Only rectangle, cylinder and disk
  (or (rectangle-p face)                ;are currently supported
      (cylinder-p face)
      (disk-p face)))

(defstruct (face (:predicate nil))
  name                                  ;name of face
  type                                  ;type of face
  analytic-description                  ;4 x 4 matrix
  in-out-flag                           ;material side flag
  surface-type                          ;description of material and surface
                                        ;finish
  list-of-edges                         ;list of edges of the face, if any
  part-of                               ;name of solid the face belongs to
  vface)                                ;name of corresponding VFace

(defstruct (rectangle (:include face))
  length
  width)
```

```
(defstruct (disk (:include face))
  radius
  center-point)

(defstruct (cylinder (:include face))
  length
  radius)

(defun solid-p (object)
  (or (primitive-p object)
      (compound-p object)))


(defstruct (solid (:predicate nil))
  name                                  ;name of solid
  type                                  ;type of solid
  geometric-class                       ;description of form
  functional-class                      ;description of function
  configuration                         ;4 x 4 hctm matrix
  mass                                  ;mass of object
  part-of)                              ;name of node immediately above this
                                        ;one in CSG tree



(defstruct (primitive (:include solid))
  (type 'primitive)                     ;type of the solid
  center-of-mass                        ;coordinates of center of mass
  rotational-inertia                    ;rotational inertia relative to master
                                        ;coordinate system
  list-of-faces)                        ;list of the faces making up the solid

(defstruct (compound (:include solid))
  (type 'compound)                              ;type of the solid
  center-of-mass                        ;coordinates of center of mass
  rotational-inertia                    ;rotational inertia relative to master
                                        ;coordinate system
  constituent-objects)                  ;list of the names and types of the two
                                        ;nodes immediate below this one and
                                        ;their mechanical relationship

;;; Description of block edges

(setf base.edge1
      (make-line
       :name 'base.edge1
       :type 'line
       :part-of '(base.base base.left)
       :endpoint1 (make-array '(4 1) :initial-contents
                              '((0.0) (0.0) (0.0) (1.0)))
       :endpoint2 (make-array '(4 1) :initial-contents
                              '((1.0) (0.0) (0.0) (1.0)))
       :parametric-description '(lambda (s)
                                  (make-array '(4 1) :initial-contents
                                              (list (list s)
                                                    '(0.0)
                                                    '(0.0)
                                                    '(1.0))))))
```

```
(setf base.edge2
      (make-line
       :name 'base.edge2
       :type 'line
       :part-of '(base.base base.back)
       :endpoint1 (make-array '(4 1) :initial-contents
                              '((0.0) (0.0) (0.0) (1.0)))
       :endpoint2 (make-array '(4 1) :initial-contents
                              '((0.0) (1.0) (0.0) (1.0)))
       :parametric-description '(lambda (s)
                                  (make-array '(4 1) :initial-contents
                                              (list '(0.0)
                                                    (list s)
                                                   '(0.0)
                                                   '(1.0))))))

(setf base.edge3
      (make-line
       :name 'base.edge3
       :type 'line
       :part-of '(base.base base.right)
       :endpoint1 (make-array '(4 1) :initial-contents
                              '((0.0) (1.0) (0.0) (1.0)))
       :endpoint2 (make-array '(4 1) :initial-contents
                              '((1.0) (1.0) (0.0) (1.0)))
       :parametric-description '(lambda (s)
                                  (make-array '(4 1) :initial-contents
                                              (list  (list s)
                                                    '(1.0)
                                                    '(0.0)
                                                    '(1.0))))))
(setf base.edge4
      (make-line
       :name 'base.edge4
       :type 'line
       :part-of '(base.base base.front)
       :endpoint1 (make-array '(4 1) :initial-contents
                              '((1.0) (0.0) (0.0) (1.0)))
       :endpoint2 (make-array '(4 1) :initial-contents
                              '((1.0) (1.0) (0.0) (1.0)))
       :parametric-description '(lambda (s)
                                  (make-array '(4 1) :initial-contents
                                              (list '(1.0)
                                                    (list s)
                                                   '(0.0)
                                                   '(1.0))))))
(setf base.edge5
      (make-line
       :name 'base.edge5
       :type 'line
       :part-of '(base.front base.left)
       :endpoint1 (make-array '(4 1) :initial-contents
                              '((1.0) (0.0) (0.0) (1.0)))
       :endpoint2 (make-array '(4 1) :initial-contents
                              '((1.0) (0.0) (1.0) (1.0)))
       :parametric-description '(lambda (s)
                                  (make-array '(4 1) :initial-contents
                                              (list '(1.0)
                                                    '(0.0)
                                                     (list s)
                                                    '(1.0))))))
```

```
(setf base.edge6
      (make-line
       :name 'base.edge6
       :type 'line
       :part-of '(base.left base.top)
       :endpoint1 (make-array '(4 1) :initial-contents
                              '((1.0) (0.0) (1.0) (1.0)))
       :endpoint2 (make-array '(4 1) :initial-contents
                              '((0.0) (0.0) (1.0) (1.0)))
       :parametric-description '(lambda (s)
                                  (make-array '(4 1) :initial-contents
                                              (list  (list s)
                                                     '(0.0)
                                                     '(1.0)
                                                     '(1.0))))))
(setf base.edge7
      (make-line
       :name 'base.edge7
       :type 'line
       :part-of '(base.left base.back)
       :endpoint1 (make-array '(4 1) :initial-contents
                              '((0.0) (0.0) (0.0) (1.0)))
       :endpoint2 (make-array '(4 1) :initial-contents
                              '((0.0) (0.0) (1.0) (1.0)))
       :parametric-description '(lambda (s)
                                  (make-array '(4 1) :initial-contents
                                              (list '(0.0)
                                                    '(0.0)
                                                     (list s)
                                                    '(1.0))))))
(setf base.edge8
      (make-line
       :name 'base.edge8
       :type 'line
       :part-of '(base.top base.back)
       :endpoint1 (make-array '(4 1) :initial-contents
                              '((0.0) (0.0) (1.0) (1.0)))
       :endpoint2 (make-array '(4 1) :initial-contents
                              '((0.0) (1.0) (1.0) (1.0)))
       :parametric-description '(lambda (s)
                                  (make-array '(4 1) :initial-contents
                                              (list '(0.0)
                                                     (list s)
                                                    '(1.0)
                                                    '(1.0))))))
(setf base.edge9
      (make-line
       :name 'base.edge9
       :type 'line
       :part-of '(base.right base.back)
       :endpoint1 (make-array '(4 1) :initial-contents
                              '((0.0) (1.0) (1.0) (1.0)))
       :endpoint2 (make-array '(4 1) :initial-contents
                              '((0.0) (1.0) (0.0) (1.0)))
       :parametric-description '(lambda (s)
                                  (make-array '(4 1) :initial-contents
                                              (list '(0.0)
                                                    '(1.0)
                                                     (list s)
                                                    '(1.0))))))
```

```
(setf base.edge10
      (make-line
       :name 'base.edge10
       :type 'line
       :part-of '(base.right base.front)
       :endpoint1 (make-array '(4 1) :initial-contents
                              '((1.0) (1.0) (0.0) (1.0)))
       :endpoint2 (make-array '(4 1) :initial-contents
                              '((1.0) (1.0) (1.0) (1.0)))
       :parametric-description '(lambda (s)
                                  (make-array '(4 1) :initial-contents
                                              (list '(1.0)
                                                    '(1.0)
                                                     (list s)
                                                    '(1.0))))))
(setf base.edge11
      (make-line
       :name 'base.edge11
       :type 'line
       :part-of '(base.right base.top)
       :endpoint1 (make-array '(4 1) :initial-contents
                              '((0.0) (1.0) (1.0) (1.0)))
       :endpoint2 (make-array '(4 1) :initial-contents
                              '((1.0) (1.0) (1.0) (1.0)))
       :parametric-description '(lambda (s)
                                  (make-array '(4 1) :initial-contents
                                              (list  (list s)
                                                    '(1.0)
                                                    '(1.0)
                                                    '(1.0))))))
(setf base.edge12
      (make-line
       :name 'base.edge12
       :type 'line
       :part-of '(base.front base.top)
       :endpoint1 (make-array '(4 1) :initial-contents
                              '((1.0) (0.0) (1.0) (1.0)))
       :endpoint2 (make-array '(4 1) :initial-contents
                              '((1.0) (1.0) (1.0) (1.0)))
       :parametric-description '(lambda (s)
                                  (make-array '(4 1) :initial-contents
                                              (list '(1.0)
                                                     (list s)
                                                    '(1.0)
                                                    '(1.0))))))
```

```
;;; Definition of the faces of the block

(setf base.base
      (make-rectangle
       :name 'base.base
       :type 'rectangle
       :length 1.0
       :width 1.0
       :analytic-description (make-array '(4 4) :initial-contents
                                         '((0.0  0.0  0.0  0.0)
                                           (0.0  0.0  0.0  0.0)
                                           (0.0  0.0  0.0  1.0)
                                           (0.0  0.0  1.0  0.0)))
       :in-out-flag 'in
       :surface-type nil
       :list-of-edges '(base.edge1 base.edge2 base.edge3 base.edge4)
       :part-of 'base
       :vface 'base.vbase))


(setf base.front
      (make-rectangle
       :name 'base.front
       :type 'rectangle
       :length 1.0
       :width 1.0
       :analytic-description (make-array '(4 4) :initial-contents
                                         '((0.0  0.0  0.0  0.5)
                                           (0.0  0.0  0.0  0.0)
                                           (0.0  0.0  0.0  0.0)
                                           (0.5  0.0  0.0 -1.0)))
       :in-out-flag 'in
       :surface-type nil
       :list-of-edges '(base.edge4 base.edge5 base.edge12 base.edge10)
       :part-of 'base
       :vface 'base.front))

(setf base.top
      (make-rectangle
       :name 'base.top
       :type 'rectangle
       :length 1.0
       :width 1.0
       :analytic-description (make-array '(4 4) :initial-contents
                                         '((0.0  0.0  0.0  0.0)
                                           (0.0  0.0  0.0  0.0)
                                           (0.0  0.0  0.0  0.5)
                                           (0.0  0.0  0.5 -1.0)))
       :in-out-flag 'in
       :surface-type nil
       :list-of-edges '(base.edge6 base.edge8 base.edge11 base.edge12)
       :part-of 'base
       :vface 'base.vtop))
```

```
(setf base.back
      (make-rectangle
       :name 'base.back
       :type 'rectangle
       :length 1.0
       :width 1.0
       :analytic-description (make-array '(4 4) :initial-contents
                                         '((0.0  0.0  0.0  1.0)
                                           (0.0  0.0  0.0  0.0)
                                           (0.0  0.0  0.0  0.0)
                                           (1.0  0.0  0.0  0.0)))
       :in-out-flag 'in
       :surface-type nil
       :list-of-edges '(base.edge2 base.edge7 base.edge8 base.edge9)
       :part-of 'base
       :vface 'base.vback))

(setf base.left
      (make-rectangle
       :name 'base.left
       :type 'rectangle
       :length 1.0
       :width 1.0
       :analytic-description (make-array '(4 4) :initial-contents
                                         '((0.0  0.0  0.0  0.0)
                                           (0.0  0.0  0.0  1.0)
                                           (0.0  0.0  0.0  0.0)
                                           (0.0  1.0  0.0  0.0)))
       :in-out-flag 'in
       :surface-type nil
       :list-of-edges '(base.edge1 base.edge5 base.edge6 base.edge7)
       :part-of 'base
       :vface 'base.vleft))

(setf base.right
      (make-rectangle
       :name 'base.right
       :type 'rectangle
       :length 1.0
       :width 1.0
       :analytic-description (make-array '(4 4) :initial-contents
                                         '((0.0  0.0  0.0  0.0)
                                           (0.0  0.0  0.0  0.5)
                                           (0.0  0.0  0.0  0.0)
                                           (0.0  0.5  0.0 -1.0)))
       :in-out-flag 'in
       :surface-type nil
       :list-of-edges '(base.edge3 base.edge9 base.edge10 base.edge11)
       :part-of 'base
       :vface 'base.vright))
```

```
;;; Description of block
(setf base
      (make-primitive
       :name 'base
       :type 'primitive
       :geometric-class 'cube
       :functional-class 'sub-component
       :configuration (make-array '(4 4) :initial-contents
                                  '((1.0  0.0  0.0  0.0)
                                    (0.0  1.0  0.0  0.0)
                                    (0.0  0.0  1.0  0.0)
                                    (0.0  0.0  0.0  1.0)))
       :mass 10.0
       :center-of-mass (make-array '(4 1) :initial-contents
                                  '((0.5) (0.5) (0.5) (1.0)))
       :moment-of-inertia (make-array '(4 1) :initial-contents
                                     '((1.67) (1.67) (1.67) (1.0)))
       :part-of nil
       :list-of-faces '(base.base base.front base.top
                        base.back base.left  base.right)))

;;; Description of edges of cylinder
(setf top.edge1
      (make-circle
       :name 'top.edge1
       :type 'circle
       :radius 0.5
       :center-point (make-array '(4 1) :initial-contents
                                 '((0.0) (0.0) (1.0) (1.0)))
       :part-of '(top.top top.side)
       :parametric-description '(lambda (s)
                                  (make-array '(4 1) :initial-contents
                                              (list
                                               (list
                                                (times 0.5
                                                       (cos (times s two-pi))))
                                               (list
                                                (times 0.5
                                                       (sin (times s two-pi))))
                                               '(1)
                                               '(1))))))

(setf top.edge2
      (make-circle
       :name 'top.edge2
       :type 'circle
       :radius 0.5
       :center-point (make-array '(4 1) :initial-contents
                                 '((0.0) (0.0) (0.0) (1.0)))
       :part-of '(top.bottom top.side)
       :parametric-description '(lambda (s)
                                  (make-array '(4 1) :initial-contents
                                              (list
                                               (list
                                                (times 0.5
                                                       (cos (times s two-pi))))
                                               (list
                                                (times 0.5
                                                       (sin (times s two-pi))))
                                               '(0)
                                               '(1))))))
```

```
;;; Description of faces of cylinder

(setf top.top
      (make-disk
       :name 'top.top
       :type 'disk
       :radius 0.5
       :center-point (make-array '(4 1) :initial-contents
                                 '((0.0) (0.0) (1.0) (1.0)))
       :analytic-description (make-array '(4 4) :initial-contents
                                         '((0.0  0.0  0.0  0.0)
                                           (0.0  0.0  0.0  0.0)
                                           (0.0  0.0  0.0  0.5)
                                           (0.0  0.0  0.5 -1.0)))
       :in-out-flag 'out
       :surface-type nil
       :list-of-edges '(top.edge1)
       :part-of 'top
       :vface 'top.vtop))

(setf top.bottom
      (make-disk
       :name 'top.bottom
       :type 'disk
       :radius 0.5
       :center-point (make-array '(4 1) :initial-contents
                                 '((0.0) (0.0) (0.0) (1.0)))
       :analytic-description (make-array '(4 4) :initial-contents
                                         '((0.0  0.0  0.0  0.0)
                                           (0.0  0.0  0.0  0.0)
                                           (0.0  0.0  0.0  0.5)
                                           (0.0  0.0  0.5  0.0)))
       :in-out-flag 'in
       :surface-type nil
       :list-of-edges '(top.edge2)
       :part-of 'top
       :vface 'top.vbottom))

(setf top.side
      (make-cylinder
       :name 'top.side
       :type 'cylinder
       :radius 0.5
       :length 1.0
       :analytic-description (make-array '(4 4) :initial-contents
                                         '((1.0  0.0  0.0  0.0)
                                           (0.0  1.0  0.0  0.0)
                                           (0.0  0.0  0.0  0.0)
                                           (0.0  0.0  0.0 -1.0)))
       :in-out-flag 'out
       :surface-type nil
       :list-of-edges '(top.edge1 top.edge2)
       :part-of 'top
       :vface 'top.vside))

;;; Description of cylinder
```

```
(setf top
      (make-primitive
       :name 'top
       :type 'primitive
       :geometric-class 'solid-cylinder
       :functional-class 'sub-component
       :configuration (make-array '(4 4) :initial-contents
                                  '((1.0  0.0  0.0  0.0)
                                    (0.0  1.0  0.0  0.0)
                                    (0.0  0.0  1.0  0.0)
                                    (0.0  0.0  0.0  1.0)))
       :mass 7.85
       :center-of-mass (make-array '(4 1) :initial-contents
                                   '((0.0) (0.0) (0.5) (1.0)))
       :moment-of-inertia (make-array '(4 1) :initial-contents
                                      '((1.15) (1.15) (0.98) (1.0)))
       :part-of nil
       :list-of-faces '(top.top top.bottom top.side)))
```

Bibliography

[1] Robert U. Ayres and Steven M. Miller. *Robotics, Applications and Social Implications*. Ballinger Publishing Company, Cambridge MA, 1983.

[2] Paul J. Besl and Ramesh C. Jain. Three-dimensional object recognition. *Computing Surveys*, 17(1):75–145, March 1985.

[3] Bir Bhanu and Chih-Cheng Ho. CAD-based 3-D object representation for robot vision. *Computer*, 20(8):19–35, August 1987.

[4] John W. Boyse and Jack E. Gilchrist. GMSolid: interactive modeling for design and analysis of solids. *IEEE Computer Graphics and Applications*, 2(2):27–40, March 1982.

[5] Rodnay A. Brooks. Model-based three-dimensional interpretations of two-dimensional images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(2):140–150, March 1983.

[6] Rodnay A. Brooks. Planning collision-free motions for pick-and-place operations. *The International Journal of Robotics Research*, 2(4):19–44, 1983.

[7] Rodnay A. Brooks. Symbolic error analysis and robot planning. *The International Journal of Robotics Research*, 1(4):29–68, 1982.

[8] Rodnay A. Brooks. Symbolic reasoning among 3-D models and 2-D images. *Artificial Intelligence*, 17(1–3):285–348, August 1981.

[9] Curtis W. Brown. DMIS—a quality interface. In *Proceedings of MAPL88*, pages 99–106, National Research Council of Canada, Winnipeg, Canada, June 20–21 1988.

[10] J. A. Campbell, editor. *Implementations of PROLOG.* Ellis Horwood Limited, Chichester England, 1984.

[11] John Canny. Collision detection for moving polyhedra. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(2):200–209, 1986.

[12] John Canny and Bruce R. Donald. *Simplified Voronoi Diagrams.* AI Memo AI-957, MIT Artificial Intelligence Laboratory, Cambridge MA, April 1987.

[13] Malcolm S. Casale. Free-form solid modeling with trimmed surface patches. *IEEE Computer Graphics and Applications*, 7(1):33–43, January 1987.

[14] Malcolm S. Casale and Edward L. Stanton. An overview of analytic modeling. *IEEE Computer Graphics and Applications*, 5(2):45–56, February 1985.

[15] Richard B. Chase and Nicholas J. Aquilano. *Production and Operations Management: A Life Cycle Approach.* Richard D. Irwin Inc., Homewood IL, third edition, 1981.

[16] E. N. Coleman and R. Jain. Obtaining shape from textured and specular surfaces using four-source photometry. *Computer Graphics Image Processing*, 18(4):309–328, April 1982.

[17] Arthur J. Critchlow. *Introduction to robotics.* Macmillan Publishing Company, New-York NY, 1985.

[18] Bruce R. Donald. A search algorithm for motion planning with six degrees of freedom. *Artificial Intelligence*, 31:295–353, 1987.

[19] M. Dowson. A note on micro-planner. In J. A. Campbell, editor, *Implementations of PROLOG*, pages 19–27, Ellis Horwood Limited, Chicester England, 1984.

[20] Bruno Dufay and Jean-Claude Latombe. An approach to automatic robot programming based on inductive learning. *International Journal of Robotic Research*, 3(4):3–20, Winter 1984.

[21] Joseph F. Engelberger. *Robotics in Practice: Management and Applications of Industrial Robots.* American Management Association, New-York NY, 1980.

[22] Michael Erdmann and Tomás Lozano-Pérez. *On Multiple Moving Objects.* AI Memo AI-883, MIT Artificial Intelligence Laboratory, Cambridge MA, May 1986.

[23] Michael Andreas Erdmann. *On Motion Planning with Uncertainty.* Technical Report TR-810, MIT Artificial Intelligence Laboratory, Cambridge MA, 1984.

[24] J.D. Foley and A. Van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley Publishing Company, Reading, MA, 1982.

[25] W. E. L. Grimson. *A Computer Implementation of a Theory of Human Stereo Vision.* AI Memo AI-565, MIT Artificial Intelligence Laboratory, Cambridge MA, January 1980.

[26] Mikell P. Groover and Emory W. Zimmers, Jr. *CAD/CAM: Computer-Aided Design and Manufacturing.* Prentice-Hall, Englewood Cliff NJ, 1984.

[27] Tsutomu Hasegawa and Hajime Terasaki. Collision avoidance: divide-and-conquer approach by space characterization and intermediate goals. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(3):337–347, May/June 1988.

[28] Robin Hillyard. The build group of solid modelers. *IEEE Computer Graphics and Applications*, 2(2):43–52, March 1982.

[29] Katsushi Ikeuchi and Berthold K.P. Horn. Numerical shape from shading and occluding boundaries. *Artificial Intelligence*, 17(1–3):141–184, August 1981.

[30] R. Jain. Dynamic scene analysis. In A. Rosenfeld and L. Kanal, editors, *Progress in Pattern Recognition*, North-Holland, Amsterdam, The Netherlands, 1983.

[31] Thomas Jech. *Set Theory.* Academic Press, New-York NY, 1978.

[32] Takeo Kanade. Recovery of the three-dimensional shape of an object from a single view. *Artificial Intelligence*, 17(1–3):409–460, August 1981.

[33] Kamal Kant and Steven W. Zucker. Toward efficient trajectory planning: the path-velocity decomposition. *The International Journal of Robotics Research*, 5(3):72–89, 1986.

[34] Fumihiko Kimura. Geomap-III: designing solids with free-form surfaces. *IEEE Computer Graphics and Applications*, 4(6):58–71, June 1984.

[35] M. Krieger, J.-P. Lachance, and R. Harvey. Process activity language PAL for planning and coordination of FMS. In *Proceedings of MAPL 88*, pages 127–135, National Research Council of Canada, Winnipeg, Canada, June 20–21 1988.

[36] Jerome C. Lange. *Design Dimensioning with Computer Graphics Applications*. Marcel Dekker Inc., New-York NY, 1984.

[37] L.I. Lieberman and M.A. Wesley. AUTOPASS: an automatic programming system for computer controlled mechanical assembly. *IBM Journal of Research and Development*, 21(4):321–333, 1977.

[38] Seymour Lipschutz. *Theory and Problems of General Topology*. Schaum Publishing Co., New-York NY, 1965.

[39] Tomás Lozano-Pérez. Automatic planning of manipulator transfer movements. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(10):681–698, 1981.

[40] Tomás Lozano-Pérez. *A Simple Motion Planning Algorithm for General Robot Manipulators*. Memo AIM-896, MIT AI Laboratory, Cambridge MA, June 1986.

[41] Tomás Lozano-Pérez. Spatial planning: a configuration space approach. *IEEE Transactions on Computers*, 32(2):108–120, 1983.

[42] Tomás Lozano-Pérez and Rodney A. Brooks. An approach to automatic robot programming. In *Proceedings of Solid Modeling by Computers: from Theory to Applications*, pages 293–328, Warren MI, September 1983.

[43] Tomás Lozano-Pérez, Matthew T. Mason, and Russell H. Taylor. Automatic synthesis of fine motion strategies for robots. *The International Journal of Robotics Research*, 3(1):3–24, 1984.

[44] Tomás Lozano-Pérez and Patrick H. Winston. LAMA: a language for automatic mechanical assembly. In *Proceedings of the Fifth International Conference on Artificial Intelligence*, pages 710–716, 1977. Reprinted in part in P. H. Winston and R. H. Brown, eds. *Artificial Intelligence: An MIT Perspective*, MIT Press. Cambridge, MA. 1980.

[45] Oded Zvi Maimon. *Activity Controller for a Multiple Robot Assembly Cell.* PhD thesis, Purdue University, 1984. Available from University Microfilms International, 300 N. Zeeb Road, Ann Arbor MI, 48106.

[46] Matthew T. Mason. Compliance and force control for computer controlled manipulators. *IEEE Transactions on System, Man, and Cybernetics*, 11(6):418–432, June 1981.

[47] D. S. Necsulescu. Robot emergency control using artificial potential field approach. Copy of a paper provided to the author by Dr. Necsulescu.

[48] James L. Nevins and Daniel E. Whitney. Computer-controlled assembly. *Scientific American*, 238(2):62–74, February 1978.

[49] William M. Newman and Robert F. Sproull. *Principles of Interactrive Computer Graphics.* McGraw-Hill, 1979.

[50] Nils J. Nilsson. *Problem-Solving methods in Artificial Intelligence.* McGraw-Hill, 1971.

[51] P. Orban, G. The, and B. Van den Berg. A robot based CMM - feasibility study. In *Proceedings of MAPL88*, pages 87–91, National Research Council of Canada, Winnipeg, Canada, June 20–21 1988.

[52] P. Piché, M. Krieger, C. Gauthier, and E. Petriu. PAD: a new tool for FMS design. In *Proceedings IECON 86*, pages 799–804, Milwaukee WI, 1986.

[53] Robin J. Popplestone, A. Patricia Ambler, and Ilona M. Bellos. RAPT: a language for describing assemblies. *Industrial Robot*, 5(3), September 1978.

[54] Robin J. Popplestone, Patricia Ambler, and Ilona M. Bellos. An interpreter for a language for describing assemblies. In Michael Brady, editor, *Robot Motion*, pages 537–568, MIT Press, Cambridge MA, 1982.

[55] Aristides A. G. Requicha. Representations for rigid solids: theory, methods, and systems. *Computing Surveys*, 12(4):437–464, December 1980.

[56] Aristides A. G. Requicha. Towards a theory of geometric tolerancing. *The International Journal of Robotics Research*, 2(4):45–60, Winter 1983.

[57] Aristides A.G. Requicha and Stephen C. Chan. Representation of geometric features, tolerances, and attributes in solid modelers based on constructive geometry. *IEEE Journal of Robotics and Automation*, RA-2(3):156–166, September 1986.

[58] Charles Rich and Richard C. Waters. Automatic programming: myths and prospects. *IEEE Computer*, 21(8):40–51, August 1988.

[59] J.R. Rossignac and A.A.G. Requicha. Offsetting operations in solid modeling. *Computer Aided Geometric Design*, 3(2):129–148, August 1986.

[60] Gideon Sahar and John M. Hollerbach. Planning of minimum-time trajectories for robot arms. *The International Journal of Robotics Research*, 5(3):90–100, 1986.

[61] Jacob T. Schwartz and Micha Sharir. On the piano mover's problem: III. coordinating the motion of several independent bodies: the special case of circular bodies moving amidst polygonal barriers. *The International Journal of Robotics Research*, 2(3):46–75, 1983.

[62] Mark Segal and Carlo H. Sequin. Partitioning polyhedral objects into nonintersecting parts. *IEEE Computer Graphics and Applications*, 8(8):53–67, January 1988.

[63] Vijay Srinivasan and Lee R. Nackman. Voronoi diagram for multiply-connected polygonal domains I: algorithm. *IBM Journal of Research and Development*, 31(3):361–372, May 1987.

[64] Richard M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[65] Donald L. Stancl and Mildred L. Stancl. *Real Analysis with Point-Set Topology*. Marcel Dekker Inc., New-York NY, 1987.

[66] Gerald Jay Sussman and Guy Lewis Steele Jr. CONSTRAINTS—a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14:1–39, 1980.

[67] R.H. Taylor, P.D. Summers, and J.M. Meyer. AML: a manufacturing language. *The International Journal of Robotics Research*, 1(3):19–41, 1982.

[68] Robert Bruce Tilove. Set membership classification: a unified approach to geometric intersection problems. *IEEE transactions on Computers*, 29(10):874–883, 1980.

[69] Joshua U. Turner. Accurate solid modeling using polyhedral approximations. *IEEE Computer Graphics and Applications*, 8(3):14–28, May 1988.

[70] S. Ullman. *The Interpretation of Visual Motion*. MIT Press, Cambridge, MA, 1979.

[71] Philippe Villers. Intelligent robots: moving towards megassembly. In Patrick H. Winston and Karen A. Predergast, editors, *The AI Business*, pages 205–222, The MIT Press, Cambridge MA, 1984.

[72] John Voelker. Automating software: proceed with caution. *IEEE Spectrum*, 25(7):25–27, July 1988.

[73] Lewis E. Ward. *Topology*. Marcel Dekker Inc, New-York NY, 1972.

[74] David E. Wilkins. Domain-independent planning: representation and plan generation. *Artificial Intelligence*, 22:269–301, 1984.

[75] P. H. Winston and B. K. P. Horn. *LISP*. Addison-Wesley Publishing Company, Reading MA, third edition, 1989.

[76] Patrick Henry Winston. *Artificial Intelligence*. Addison-Wesley Publishing Company, Reading MA, second edition, 1984.

[77] Andrew P. Witkin. Recovering surface shape and orientation from texture. *Artificial Intelligence*, 17(1–3):17–45, August 1981.

[78] Robert J. Woodham. Analyzing images of curved surfaces. *Artificial Intelligence*, 17(1–3):117–140, August 1981.