The Cornell Robot System Design Report

Doug Campbell

TR 85-697
August 1985

Department of Computer Science
Cornell University
Ithaca, NY 14853

This work was supported in part by the National Science Foundation under grant ECS-8312096
ABSTRACT

This report describes a robot control system under development at Cornell University. The goal of the system is to demonstrate automatic generation of robot programs for mechanical assemblies that are specified by exploded diagrams. The structure and current capabilities of the system are discussed.
# TABLE OF CONTENTS

1 The Project .................................................................................................................. 2  
1.1 Project Goals ........................................................................................................... 2  
1.2 System Specifications ............................................................................................... 2  
  1.2.1 Paradigm of Operation .......................................................................................... 3  
  1.2.2 Exploded Diagrams .............................................................................................. 3  
  1.2.2.1 Objects ............................................................................................................. 3  
  1.2.2.2 Features ........................................................................................................... 4  
  1.2.2.3 Assembly Procedure Specifications .................................................................. 4  
  1.2.3 Assembly Procedures .......................................................................................... 4  
2 System Structure and Operation .................................................................................. 6  
  2.1 The Modules ........................................................................................................... 6  
    2.1.1 The Object Modeler ......................................................................................... 6  
    2.1.2 The Exploded Diagram Editor ........................................................................... 6  
    2.1.3 The Assembly Planner ...................................................................................... 7  
    2.1.4 The Assembly Procedure Manager ................................................................. 9  
    2.1.5 The Placement Planner ...................................................................................... 9  
    2.1.6 The Motion Planner .......................................................................................... 10  
    2.1.7 The Grip Planner ............................................................................................. 10  
  2.2 The Interfaces ......................................................................................................... 10  
    2.2.1 Objects ............................................................................................................. 10  
    2.2.2 Exploded Diagrams ........................................................................................... 11  
    2.2.3 Configurations ................................................................................................. 11  
    2.2.4 Conditions ....................................................................................................... 12  
    2.2.5 Object Motions ................................................................................................. 12  
    2.2.6 Robot Motions ................................................................................................... 12  
3 Current Capabilities .................................................................................................... 14  
  3.1 Current State of the System .................................................................................... 14  
  3.2 The Lincoln Log Environment ............................................................................... 15  
  3.3 A Detailed Example of Current Operation .............................................................. 16  
4 Future Work ................................................................................................................ 18  
  4.1 Current Efforts ........................................................................................................ 18  
  4.2 Future Goals .......................................................................................................... 18
CHAPTER 1

THE PROJECT

1.1. Project Goals

The goal of this project is to create a high-level, task oriented system for describing and generating robot assemblies. In short, we want a system that the non-technical user can use to quickly and easily generate a large class of robotic assemblies.

Today, many mechanical assemblies are being performed with robots. Most, however, are not. One reason for this is the lack of a simple, general method of programming robots. This project attempts to create a robot programming system that has both generality and simplicity. While previous projects [Lieberman 1977, Lozano-Perez 1977, Popplestone 1978, Taylor 1982] have made some progress toward this goal, we consider our project a more global, general, and ambitious attempt.

A general but complex method of robot programming is to explicitly program the robot at the standard programming-language level. This method is completely general because it allows full control of robot motions and sensory feedback. It is unacceptable in many industrial settings, however, because it is complicated, unreliable, and inflexible. There have been few high-level language facilities developed that allow high-level specifications of essential robot actions like grasping and pushing. Some progress has been made with high level specifications of motions [Mujtaba 1979] and though this is useful, it does not help with more complex issues like tactile sensing and reasoning about objects in space. Because of this, the act of programming robots in a standard high-level language is similar to programming general-purpose computers in machine language.

Another current method, which is simple but not general, requires assemblies to be specified using a teach pendant. Here, the robot is manually guided through every motion that it must perform for a particular assembly. It can then play back the sequence of motions to perform the assembly any number of times. This is unacceptable as a general method for the following reasons:

1. This method prohibits the generalized use of force feedback, visual feedback, or other sensory-guided decisions which have been shown to be useful and even necessary in many assemblies [Inoue 1974].
2. Small changes in a robot's environment, parts, or task may require substantial human-supervised re-teaching.
3. Teaching requires the existence of physical models or prototypes of the objects used in the assembly. These may not be available.

Our goal is to create a system driven by computer models of objects and actions that will make general use of sensory feedback and require little human effort to generate and modify complete robot assemblies.

1.2. System Specifications

Three major decisions were made in the initial specifications of the system. First, it was decided that the user would use exploded diagrams as the method of specifying and modifying assemblies. Exploded diagrams were chosen because they are a simple method of specifying
assemblies that are generally understood.

Second, it was decided to make the system modular, with well-defined interfaces between the modules. The reason for any modular system is that a designer can work on a small section of the system without knowing or being concerned with the operations of the rest of it. This is especially desirable in our situation, since the fruits of other robotics research projects can be plugged in and used easily.

Third, it was decided that completely general automatic planning of fine, compliant motions is intractable (witness the difficulties in implementing even the simplest of such problems [Lozano-Perez 1984]), and thus would not be attempted by our system. Instead, we require some sensory feedback to be explicitly programmed. At first glance, this seems to defeat our goal of being able to specify assemblies easily. However, in any large assembly operation there are a small number of complex compliant motions which are repeated often. Examples are screwing, hammering, and so on. These procedures need only be programmed once, put into a library, and invoked where needed.

The specifics of these decisions and the paradigm of operation of the system are described in the following sections.

1.2.1. Paradigm of Operation

Three data structures are input by the system in order to generate a complete robot assembly program. One is an exploded diagram (see figure 2). This diagram contains specifications of parts, where they should be attached, and the name of an assembly procedure to attach them. The end user will input these diagrams.

Another input to the system is an initial configuration of objects in the robot’s workspace. This is also input by the end user.

The assembly procedures themselves are the third input to the system. These assembly procedures are programs that perform basic, generalized assembly tasks such as screwing and hammering. These procedures are resident in the system, and will not be input by the end user.

Given the exploded diagram, initial configuration, and assembly procedures, the system generates robot code to perform the assembly. Thus, the system can be thought of as an exploded diagram compiler, using a library of assembly procedures. The compiler transforms an exploded diagram of an assembly into the robot code required to perform it.

1.2.2. Exploded Diagrams

Our concept of an exploded diagram corresponds closely to that used in industry. Figure 1 shows a typical industrial-style exploded diagram of the assembly of a metal box. Figure 2 shows the information contained in an exploded diagram that might be used by our system to perform the same assembly. Both contain essentially the same information. Our diagram has a bit more information about exactly how to perform the screwing procedures. These parameters are those that a human might know or use intuitively, but must be specified to a robot.

There are three main components in both diagrams. They are the objects, the object features, and the lines between features that specify assembly procedures. These are each described below.

1.2.2.1. Objects

Our objects are computer representations of physical entities. The exact form of such a representation (and, indeed, a definition of what an object even is) is a current research topic at Cornell, and thus we are using the simplest representation possible until research results are obtained. Currently, our objects are named coordinate frames in Euclidean 3-space.
That is, they have three positional and three rotational degrees of freedom. This frame is only significant as a reference point for features that are defined on the object. That is, the object’s coordinate frame doesn’t have to be located anywhere specific on the object (or even on the object at all). This representation will become more elaborate and useful as research results become available.

1.2.2.2. Features

A feature is a coordinate frame relative to some object’s coordinate frame. In figure 2, the features are represented by groups of three arrows representing the origin of their coordinate frames. They are used as parameters to assembly procedures. The corresponding entities in figure 1 are the parts of the objects at the ends of the dotted lines, and they are less precisely located than we require with our system. A human intuitively knows that the ends of the dotted lines refer to the bolt shafts and holes, but the system requires exact knowledge of the locations and types of these features. The assembly procedures of the system assign meaning to these points. For example, the screwing procedures all learn that these feature locations correspond to the centers of threaded shafts and holes. How they learn this will be discussed under the section on the exploded diagram editor.

1.2.2.3. Assembly Procedure Specifications

The dotted lines in figure 1 represent an assembly operation. So do the dotted lines in our representation in figure 2. Our lines, however, explicitly name the assembly procedure to be performed, and include parameters to the procedure beyond the features on which it will operate. These extra parameters are obtained from the user by the exploded diagram editor.

1.2.3. Assembly Procedures

An assembly procedure is a program that performs a single class of assembly operation. These programs are robot-independent. They specify abstract motions and abstract sensory feedback that can be used by any sufficiently capable robot hardware. They are also environment-independent, in that they specify certain abstract conditions under which they will work.

These procedures accept parameters. The one essential type of parameter is the group of features on which the procedure will operate. These features are those at the ends of the dotted lines that name the procedure in the exploded diagram. Other parameters to assembly procedures determine how the procedure is to be performed and provide extra information about the environment. For example, in figure 2, the final torque of the screwing and the pitch of the screw threads are specified as parameters to each screwing procedure. The methods by which the procedures obtain these parameters are described under the section on the exploded diagram editor.

The language in which assembly procedures are specified is a current research topic. Though we are expecting it to be a high-level language, we do not expect the non-technical user to be able to generate correct assembly procedures with it. Instead, a large library of the most common assembly procedures will become a resident part of the system. The average user will then view these procedures as black boxes, and will be able to invoke them where needed. The system will check that the conditions are amenable to each invocation of an assembly procedure. For example, the system would notice and prohibit an attempt to screw a bolt into a hole that was covered by some other object. The methods by which these checks are done are described in the sections on the assembly planner and the assembly procedure manager.
CHAPTER 2

SYSTEM STRUCTURE AND OPERATION

2.1. The Modules

The modules of our system and the main flow of information between them are shown in figure 3. These modules are designed to represent the basic components required for automatic assembly programming. Each one has the potential to be a large research project, and many are being studied by the Cornell Robotics Group.

To define a module one defines the interface that the module has with the rest of the system. Once the interface definition is complete, the required functions of the modules have been determined, and work on implementing the modules can progress. As an aid to intuition, however, we present the functions of the modules first, without regard to the specifics of the interfaces. In this discussion, we will follow the specification and compilation of the assembly of the metal cabinet specified in figure 2.

2.1.1. The Object Modeler

The object modeler allows users to create models of the objects to be assembled. In our example from figure 2, the box body, the box lid, and a bolt have to be modeled. These models will be kept in some sort of library for use by the exploded diagram editor.

The definition of an object model is a current research topic at Cornell. Currently our object models are sets of named features, which are coordinate frames in space, and their relative positions. In the future, we expect this definition to grow. For example, some sort of volume model will be required. There is much current research going on in this area at Cornell, and we expect a volume modeler to appear soon.

2.1.2. The Exploded Diagram Editor

The entire specification and modification of assemblies is done in the exploded diagram editor. It allows the user to take object models from the object modeler and assembly procedures from the assembly procedure manager, and use them to create exploded diagrams.

To draw "lines" between objects in the diagram, the user first specifies the name of the assembly procedure that will perform the operation. When an assembly procedure is specified, the procedure itself begins to prompt the user for parameters. Features of the objects that it must know about are entered by graphically creating coordinate frames on the objects involved.

For example, suppose the user wants to draw a line from a bolt to a bolt hole to specify one of the screwing operations in figure 2. The user first finds the name of the screwing procedure in the assembly procedure manager, and requests it. Once the user has selected this procedure, the assembly procedure's parameter inquiry code is run. This code prompts for the following information:
1. The location of the bolt shaft
2. The location of the bolt hole
3. The location of any holes of objects that will be fastened between them
4. The final torque to be applied
5. The pitch of the threads

The first three items are input by graphically creating coordinate frames on the bolt shaft, the bolt hole, and the cover hole. For example, to specify the location of the bolt shaft, the user might be asked to place a coordinate frame centered on the open end of that feature so that the frame's Z-axis points out of the open end of the shaft, and its X and Y axes lie in the plane of the end of the shaft (figure 4). In this way, the screwing procedure now can associate meanings with the feature coordinate frames. The torque is input as a numerical parameter (with a default given), and the pitch of the threads is input similarly. The assembly procedure manager then stores this information away for later use in the planning process.

This entire operation will not be necessary for each screwing operation. The editor will be able to make "copies" of assembly procedure parameters, making local changes to them for each invocation. Also, as object models become more sophisticated, object-inherent parameters such as pitch of threads may be able to be determined automatically from the object model itself.

A partial ordering on the assembly procedures may be supplied by the user. The system will try to infer the ordering constraints if they are not specified, but specifying them explicitly is sometimes useful when the system cannot know about certain temporal issues. For example, it would be wise to specify to the system that a piece of metal must be bent after it is put in the furnace, and not before.

Initial conditions of the objects will also be input by the exploded diagram editor. These initial conditions are represented by a configuration, which will be described later. In the future we plan to allow the user to specify the arrival of parts in the robot's workspace by reference to a procedure that notifies the robot when and where the part arrives.

2.1.3. The Assembly Planner

Once an exploded diagram and an initial configuration are complete, they are sent to the assembly planner for compilation. The assembly planner's task is to oversee the operations of the placement planner, motion planner, assembly procedure manager, and grip planner during the planning phase. It must choose feasible assembly orderings, plan them, and generate code to perform them.

At present, the assembly planner uses a simple heuristic to plan assemblies. It first picks an ordering of assembly procedures based on the partial ordering constraints provided to it. Then for each assembly procedure in the exploded diagram it performs the following six steps.

A. Have the placement planner establish the assembly procedure's precondition.
B. Have the assembly procedure plan the motions required to perform the operation.
C. Have the grip planner plan a grip of the objects involved.
D. Have the grip planner plan ungrips, if necessary, from the previous operation.
E. Have the motion planner plan gross motions from the end of the previous operation to the grips of this operation.
F. Have the motion planner plan gross motion from the gripped objects to the starting positions of the assembly procedure.

The order in which the operations are planned is not the order in which they will be performed. This allows highly constrained planning to be done before loosely constrained planning, increasing chances of a successful plan. Figures 5 and 6 show the assembly order and the planning order respectively. The lettered steps correspond to planning steps, and the numbered steps correspond to the actual stages of assembly. The details of each planning
operation are given in figure 6.

A. Establish assembly procedure's precondition

The assembly procedure has already stored away its parameters in the assembly graph. From these parameters, it now generates a precondition that must be met for the operation to proceed. Preconditions are boolean expressions on relative positions of objects, relative orientations of objects, free space around objects, grippedness of objects, and stability of objects. For our example of screwing the bolt, the precondition might contain seven conjunctive clauses that state:

1. The end of the bolt shaft must be 1mm above the box cover's hole.
2. The bolt shaft must be aligned with the box cover's hole.
3. The box cover's hole must be aligned with the bolt hole.
4. The box cover's hole and the bolt hole must be touching (spatially next to each other).
5. There must be free space around the bolt shaft equal to the maximum radius of the bolt head.
6. The bolt must be gripped by a robot manipulator.
7. The bolt hole must be stable (to within some force rectangle in 6-space).

The placement planner is now called upon to choose a configuration of objects in the workspace that satisfy this condition. In our example, the placement planner puts the bolt in the obvious position just above the hole.

B. Plan the assembly operation

With the precondition of the assembly operation established, the assembly procedure now generates the code to perform the operation. All physical motions are described as pure object motions with no reference to a robot. That is, the objects are moved about "by magic" until we know what robot manipulators will be moving them. In our example, the bolt will be moved in a downward spiraling path into the bolt hole. Later, when the grip of a robot manipulator is determined on the bolt, the motions of the bolt can easily be translated into motions of the robot manipulator that is gripping it (see figure 7).

C. Plan the grips

Now that the assembly operation motions have been determined, a grip of the manipulated objects can be determined. The grip is determined after the assembly operation motions are determined because the assembly operation motions are fixed, and cannot be changed. Thus, the grip planner must choose a grip that will work in the context of the assembly operation motions. In our example, the grip planner must find a grip on the bolt that will cause no collisions with other objects when the bolt is moved downward into its hole (see figure 7).

The grip planner produces code for gripping the relevant objects and a requested configuration of the robot manipulators at the start of its gripping routines. This robot configuration must be attained before the grip planner's code can run to grip the objects.

D. Plan the ungrips

The previous assembly operation may have left the robot gripping some objects that we don't wish to manipulate now. Thus, the grip planner must find safe ways to ungrip these objects from the configuration at the end of the last operation. It plans these ungrips and returns a new configuration of the workspace that reflects the status after its ungrip actions. In our example, the robot may be holding onto a bolt that had been screwed in at the previous operation, and would have to be ungripped.
E. Motion plan ungrip to grip
The motion planner is now called to get the robot manipulators from their ungrip configurations to their configurations at the start of the gripping routine. These configurations were created during steps C and D. In our case, this would involve moving the gripper from its position after it ungripped the previous bolt to the position from which it could begin its gripping routine for the next bolt.

F. Motion plan grip to assembly operation
The motion planner is now called to move the robot manipulators with their gripped objects to the configuration determined in step A by the placement planner. This configuration is the one that satisfies the assembly procedure’s precondition and allows the operation to proceed. In our example, the gripped bolt would be moved to a position just above the box lid’s hole.

When this sequence of steps has been performed for each assembly operation in the exploded diagram, the robot code for the entire assembly is generated and ready to run.

There are lots of complicated research issues that must be addressed for this scheme to be successfully implemented in generality. As this system is a testbed for research, however, this situation exactly meets our requirements. We feel that many results of robotics research can be directly and easily applied and tested in this setting.

2.1.4. The Assembly Procedure Manager
The assembly procedure manager is the librarian for all of the assembly procedures. It keeps track of each assembly procedure’s three parts. These parts are:

1. Parameter Inquiry Code
The parameter inquiry code is invoked whenever an assembly procedure is used in the exploded diagram editor. Its task is to accumulate enough information about each instantiation so that it can correctly plan the assembly in some future context. An example of some possible queries from the screwing procedure is given in the section on the exploded diagram editor.

2. Precondition Generation Code
The precondition generation code generates a precondition based on the parameters that have been received from the exploded diagram editor. These conditions are boolean expressions on relative positions of objects, relative orientations of objects, free space around objects, grippedness of objects, and stability of objects. An example of a precondition is given in the section on the assembly planner.

3. Planning Code
The planning code of an assembly procedure generates the actual object motions to perform the operation. It is run when the system knows the configuration of objects in the workspace at the time the operation is performed. This configuration is passed to the planning code, and the planning code then generates the object motions to perform the operation in this situation. These object motions are later translated into robot motions.

2.1.5. The Placement Planner
The placement planner’s job is mainly to establish preconditions. It must arrange objects in such a way that preconditions of assembly procedures are established while ensuring that the resultant configuration of objects is stable. For example, to establish the precondition for one of the bolting procedures, it might have to place the box lid on top of the box in the obvious way. It must know not to place the box on its side and then put the lid on end next to it, as this would not be a stable configuration.
Research is currently underway at Cornell to determine the stability of configurations of objects. This will be essential to a successful placement planner.

2.1.6. The Motion Planner

The motion planner is used to plan large-scale motions of robots and their gripped objects through a cluttered workspace. Formally, its parameters are two configurations of the robot workspace, one the starting configuration, and one the desired configuration. It then returns the robot motions necessary to achieve the change.

2.1.7. The Grip Planner

The grip planner has two tasks. One is to determine and perform grips that will work in the context of specific assemblies. The other is to perform safe ungrips of objects.

To plan a grip on an object, the grip planner takes as parameters the motions that the object will make in the ensuing assembly operation, and the configuration of surrounding objects when these motions are performed. The grip planner then determines a grip that will be safe in the context of these assembly motions. After that, it determines how to grip the object with such a grip from its current configuration in the robot workspace.

For example, to screw in a bolt, the grip planner would be required to grip the bolt in such a way that its grip would not cause a collision in the process of screwing in the bolt. One safe grip would be to grab the bolt sides with the tips of its fingers (see figure 7). The grip planner would then have to determine how obtain this grip on the bolt from its current configuration. If the bolt started out upside-down on the table (see figure 5), then the grip planner would have to figure out how to turn the bolt over before gripping it correctly.

2.2. The Interfaces

This section formally describes the interfaces between the modules of the system.

2.2.1. Objects

An object is currently a list of named features. Each feature consists of a sextuple representing the \((x,y,z,\theta,\phi,\psi)\) coordinate transform of this feature relative to its containing object's origin, a reference to its containing object, and a list of named values. These named values can contain information of the sort \((name = \text{pitch}, value = 17)\) to represent the fact that this feature has pitch 17. These names and values are used by assembly procedures as parameters to their attachment routines. They are generated either from the object modeler or from the assembly procedure's parameter query section.

\[
Object = \{ \begin{array}{l}
\{ name_1 \ feature_1 \\
\{ name_2 \ feature_2 \\
\ldots \\
\{ name_n \ feature_n \\
\end{array}
\]

\[
Feature = \{ \begin{array}{l}
(\{ \begin{array}{l}
(x,y,z,\theta,\phi,\psi) \\
object \\
name_1 \ value_1 \\
name_2 \ value_2 \\
\ldots \\
name_n \ value_n \\
\end{array}\end{array}
\}
\]
2.2.2. Exploded Diagrams

At the highest level, an exploded diagram is represented as a list of assembly procedure instantiations, corresponding to the lines in the diagram. Each instantiation consists of the name of the assembly procedure to perform the operation, a list of \((name, value)\) pair parameters, and a list of other assembly procedure instantiations. This last list represents a adjacency list that implements a DAG (directed, acyclic graph) of the procedures to specify a partial ordering. The vertices of the DAG are assembly procedure instantiations, and an edge from \(a\) to \(b\) means that \(a\) must be performed before \(b\).

The \((name, value)\) pairs are the parameters to each procedure instantiation. Features of the objects to be attached are listed here. For example, one \((name, value)\) pair might be \((name = \text{slot-center}, value = \text{log-upper-left-slot})\) where "log-upper-left-slot" is a name of an object's feature name. That is, it's a reference to a reference to a feature. The reason for double indirection will be given in the section on configurations. The list of \((name, value)\) pairs also contains parameters of the sort \(name = \text{torque}, value = 500\) to represent the fact that this attachment should be performed to a torque of 500. These names and values are generated from the assembly procedure's parameter query section.

\[
\text{Exploded-diagram} = \{ \text{assembly-node}_1 \} \cup \{ \text{assembly-node}_2 \} \cup \cdots \cup \{ \text{assembly-node}_n \}
\]

\[
\text{Assembly-node} = \{ \text{assembly-proc-name} \} \cup \{ \text{name}_1 \ \text{value}_1 \} \cup \{ \text{name}_2 \ \text{value}_2 \} \cup \cdots \cup \{ \text{name}_n \ \text{value}_n \} \cup \{ \text{assembly-node}_1 \} \cup \{ \text{assembly-node}_2 \} \cup \cdots \cup \{ \text{assembly-node}_m \}
\]

2.2.3. Configurations

Configurations perform two sets of bindings. They bind objects to locations and features to assembly procedure parameters.

Binding objects to locations is straightforward. A list of \((object, location)\) pairs is maintained, with each location being an \((x, y, z, \theta, \phi, \psi)\) coordinate transform of the object with respect to a global coordinate frame.

Binding features to assembly procedure parameters involves matching names in the value half of the \((name, value)\) pairs of assembly nodes with the names of features of objects in the object bindings of the configuration. This might seem at first a roundabout way of referencing a feature from an assembly operation. There are no simpler alternatives available, however. If we were to have assembly operations directly reference features, we could not maintain multiple configurations with just one exploded diagram, which is necessary. For example, one configuration is needed to represent the world state just prior to the assembly operation, while another is needed to plan the grip of the objects from an entirely different world state. Adding a level of indirection by specifying names of features instead of directly referencing them is still insufficient. This is because features of subobjects may become features of a larger object as a result of some assembly operation, and it would be inconvenient to require the names of the features of the new object to equate with their corresponding names on the subobjects.
Configuration = \[
\quad \begin{aligned}
&= \{ \text{object}_1(x_1, y_1, z_1, \theta_1, \phi_1, \psi_1) \\
&\quad \text{object}_2(x_2, y_2, z_2, \theta_2, \phi_2, \psi_2) \\
&\quad \quad \quad \quad \quad \quad \quad \quad \cdots \\
&\quad \quad \quad \quad \quad \quad \quad \quad \text{object}_n(x_n, y_n, z_n, \theta_n, \phi_n, \psi_n) \\
&\quad \text{parameter-name}_1 \text{ feature-name}_1 \\
&\quad \text{parameter-name}_2 \text{ feature-name}_2 \\
&\quad \quad \quad \quad \quad \quad \quad \quad \cdots \\
&\quad \text{parameter-name}_n \text{ feature-name}_n
\end{aligned}
\]

2.2.4. Conditions

Conditions are boolean expressions. They are used to express the allowable relative states of features before an assembly procedure can be invoked. The elements in the boolean expression are of the following types:
1. (positioned feature$_1$ feature$_2$ (x, y, z, \theta, \phi, \psi))
   Specifies that feature$_1$ must be at the specified position relative to feature$_2$.
2. (collinear point$_1$, point$_2$, point$_3$, point$_4$, \ldots, point$_n$, angle)
   Specifies that 3 to 4 points on features must be ordered collinearly, with an optional specification of the rotational deviation of the segments defined by (point$_2$, point$_4$) and (point$_4$, point$_n$) about the line of collinearity.
3. (inside point volume)
   Specifies that point on a feature must be inside volume relative to some feature.
4. (empty volume, object$_1$, \ldots, object$_n$)
   Specifies that no object intersects volume relative to some feature except for objects 1 through n.

We have found these requirements sufficient to specify preconditions for a large class of assembly operations. Conditions have not yet been used to their full capabilities, so this class of restrictions has not yet been fully tested.

2.2.5. Object Motions

Object motions are currently only specified by the final (x, y, z, \theta, \phi, \psi) world location of the object's origin. We intend to implement representations for straight line and rotation motions, and force-guided motions.

2.2.6. Robot Motions

A robot motion is a list of sets of concurrent robot actions. Each robot action specifies an actuator name and a value. The intent is that the named robot actuator must attain the new value. Since we plan to have the complete dynamic model of each robot that the system plans assemblies for, the system can determine what actuators can and should be controlled to attain any desired configuration. In this way our system is robot independent. The modules that must use the robot model are the motion planner, the grip planner, and the ungrip planner.

Robot-motions = \[
\quad \begin{aligned}
&= \{ \text{concurrent-robot-action}_1 \\
&\quad \text{concurrent-robot-action}_2 \\
&\quad \quad \quad \quad \quad \quad \quad \quad \cdots \\
&\quad \quad \quad \quad \quad \quad \quad \quad \text{concurrent-robot-action}_n
\end{aligned}
\]
Concurrent-robot-motion = \{ \begin{array}{c}
actuator_1 \ \text{value}_1 \\
actuator_2 \ \text{value}_2 \\
\ldots \\
actuator_n \ \text{value}_n
\end{array} \}
CHAPTER 3

CURRENT CAPABILITIES

3.1. Current State of the System

This system was designed to evolve. As more and more research bears fruit, we will continue to upgrade its capabilities and performance. We felt it desirable to get the entire framework implemented even before many research results were available for two reasons. First, we could test the feasibility of our framework. Second, without the entire framework, it would be difficult to test the usefulness of any module, since the final assembly could never be attained.

The design and implementation of the framework with very simple modules ended up providing interesting insight into the capabilities of the framework itself. The current implementation was designed with the following restrictions:

1. There are no volume models of objects. Our group is currently doing research to determine the most appropriate object representation, and until some results are available, we have no representation of the volume occupied by an object.
2. No sensory feedback is available. Force sensing hardware has been installed, but the controlling software is not yet integrated with the system.

These restrictions imply other restrictions. Most importantly, no real motion planning can occur. Without volume models of objects, it is impossible to avoid collisions. Also, general automatic gripping cannot be done without volume models.

With these restrictions, it might seem that nothing can be done. It turns out that abnormal use of the assembly procedures can bypass some of these restrictions to allow assemblies in restricted environments. For example, gripping a specific object can be coded as an assembly procedure. This is not the way that the system is designed to operate, but it at least allows us to demonstrate some real-world assemblies.

The current system’s modules have been implemented to the following degrees:

The Object Modeler
No volume models have yet been implemented, but feature coordinate frames are used extensively. Object features must currently be input as lisp S-expressions.

The Exploded Diagram Editor
A special-case editor of Lincoln log assemblies has been implemented. This is described in the next section.

The Assembly Planner
An assembly planner has been implemented that coordinates all assembly planning activity. It coordinates the operations of the modules as described in section 2.1.3. It cannot yet handle error recovery in the planning.

The Assembly Procedure Manager
Two assembly procedures have been coded. One is a simple placement procedure that puts a feature of an object at a specified place in the workspace. The other is a stacking procedure that matches up one or more pairs of features and places them so that all pairs are touching. The preconditions generated are always fixed relative position requirements. The parameter query sections have not yet been implemented.
The Placement Planner
   No implementation.

The Grip Planner
   A simple grip planner has been implemented. It moves a two-fingered gripper just
   above the object to be gripped, moves it down to the center of the object, closes the two
   fingers, and moves the object up off the table a few millimeters. Ungripping involves
   opening the fingers and moving up.

The Motion Planner
   Our motion planner currently uses a simple heuristic. It finds the manipulators and
   objects that must change position, it moves them straight up high in the air above the
   workspace, moves them to their new (x,y) coordinates, and then lowers them to their
   new z coordinate. There are clearly a large class of problems on which this heuristic
   will fail, and though there are better solutions [Brooks 1983, Lozano-Perez 1979] it
   remains an open research problem. Recent results [Hopcroft 1984] indicate that no com-
   pletely general solution is tractable, so heuristics are likely to be necessary.

These modules have been implemented in GLISP, a portable set of LISP definitions that
provide an object-oriented programming environment. They currently run on Franz Lisp on a
VAX 750 under Berkeley 4.2, and on a Xerox Dandelion workstation running Interlisp. The
robot used is a Unimation Puma 560.

3.2. The Lincoln Log Environment

   Our goal with the Lincoln log environment was to get the framework of the system to
   the point that it could build arbitrary Lincoln log structures. This goal has been attained.
   The system currently is able to build arbitrary Lincoln log structures to within the limits of
   the current user interface to specify them.

   The models of the Lincoln logs themselves have been input to the system as lisp S-
   expressions of the correct type. The models consist of a coordinate frame which is at the ori-
   gin of the Lincoln log (representing the object), and coordinate frames for each slot, relative
   to the origin (representing the features). These frames are shown in figure 8.

   A custom exploded diagram editor has been implemented for Lincoln log assemblies. It
   displays logs and allows the user to draw lines between the slots indicating attachment (figure
   9). This method is not completely general, since the user cannot specify the rotational ori-
   entation of a connection. When the user is satisfied with the diagram, he requests the system to
   compile it. This user interface checks the validity of the diagram, and if it finds a consistent
   global assembly, it sends the exploded diagram to the assembly planner.

   Checking the validity of a Lincoln log assembly involves checking the stability and con-
   nectedness of the result. Both checks view the exploded diagram of the assembly as a kind of
   graph. The logs correspond to nodes, and the attachment lines correspond to edges. This is
   not exactly a graph, however, in that each edge into a log is attributed by the identifier of the
   slot that this attachment involves.

   Stability is checked by traversing this graph and associating a height and vertical orienta-
   tion with each log. Once a base log and its orientation are determined, this can be done
   easily. Each log with a slot connected to an upper slot of a log of height h is of height h + 1.
   Similarly, each log with a slot connected to a lower slot of a log of height h is of height h - 1.
   The orientation of the new log is determined by the identifications of the slots involved.
   Attempting to assign two different heights or orientations to a log or to assign a negative
   height implies an unstable or invalid assembly. Having successfully assigned all heights and
   orientations, stability is ensured if each log of height > 0 has attachments to at least one bot-
   tom slot on each side of its center. We don't know a priori which log(n) may be base logs, so
   we perform the stability check with each log in the graph as a base log in both possible
orientations until either a stable configuration is confirmed or all possibilities are eliminated.

Connectedness of a Lincoln log assembly is checked by traversing the assembly graph and assigning horizontal locations to the endpoints of each log. Since the slots of Lincoln logs are equally spaced, we can assign \((x,y)\) integer coordinates to each slot. For example, suppose a two-slot log \(A\) with slots at \((0,0)\) and \((0,1)\) is connected to a similar log \(B\), the connection being at \((0,0)\). Then log \(B\)'s other slot must be at either \((1,0)\) or \((-1,0)\). We can then search the graph, recursively choosing both possibilities for each new log encountered, until a valid connectedness is achieved, or all possibilities are exhausted. Attempting to assign two different \((x,y)\) coordinates to a slot indicates an invalid coordinate assignment.

Once an assembly has been verified for stability and connectedness, the exploded diagram is attributed with assembly operation ordering constraints. These ordering constraints specify that no log of height \(h\) can be stacked until the logs of height \(h-1\) that it will be attached to have been stacked.

The exploded diagram editor is the only module of the system that has any Lincoln log specific code. Once the diagram is passed to the assembly planner, the system behaves with its full (currently limited) generality. Robot code is generated which, when run, assembles the requested structure.

3.3. A Detailed Example of Current Operation

This section will describe the actual algorithms that are used to plan the log stack assembly operation. The steps are labeled A - F, corresponding to the steps defined in section 2.1.3 and in figures 5 and 6.

Throughout this example, we assume that every object's origin and features can be absolutely located in the robot's workspace. This is because a configuration that maps objects to absolute positions is always available. The mapping requires one \(4\times4\) matrix multiplication to determine an object origin's absolute position, and two such multiplications to determine a feature's absolute position.

A. Establish assembly procedure's precondition
Preconditions for log stacking are always strict relative position requirements. Given a set of pairs of slots to be stacked, the system calculates positions 5mm directly above the slots to be stacked upon. These will be the starting positions of the slots to be stacked. The relative position of the origin of the stacking log can then be calculated from the relative location of just one of the log's slots by an inverse coordinate frame transformation of the slot's position relative to the origin.

B. Plan the assembly operation
The assembly operation for stacking logs involves moving the upper log down 5mm in a straight line.

C. Plan the grips
Since each log's origin has been placed at its grip position (according to our current conventions), the robot gripper can obtain a grip by positioning itself at the log's coordinate frame origin and closing its fingers. The position of the robot to start the grip is 2cm above the grip position, and the grip operation moves down 2cm, closes fingers, and moves up 2cm.

D. Plan the ungrips
Ungripping is performed by opening the fingers and moving up 2cm.

E. Motion plan ungrip to grip
The motion planning algorithm is described in section 3.1. The robot hand is motion planned from the ungrip of the previous operation to the position from which the grip routine can run. This new position is 2cm above the origin of the new log.
F. Motion plan grip to assembly operation

The motion planning algorithm is described in section 3.1. The robot hand, with the log gripped, is motion planned from the grip end position to the log stack start position calculated in step A.

These algorithms are not intended to be general or novel. The purpose of these algorithms was to demonstrate a minimum operational system using the formalisms defined. Now, with a minimal system operational, more general and sophisticated algorithms can be designed and tested.
CHAPTER 4

FUTURE WORK

4.1. Current Efforts

Force sensing hardware has been added to the robot available to us, and work is underway to integrate this dimension of the system. The most fundamental formalism change is to the specification of robot motions, which will now contain provision for force-guided motions. Primitives are being considered that maintain forces until at the boundaries of specified force-and position 6-space rectangles. Temporal constraints will also be included to prevent infinite looping.

4.2. Future Goals

This scope of this system is so large that there are always many areas in which to progress. The robotics group at Cornell, while interested in all aspects of robotics, are currently making specific strides in the studies of stability, object modeling, motion planning, and graphical user interfaces.
ACKNOWLEDGEMENTS

Aliza Wechsler provided valuable help with the implementation of the system.

Many helpful suggestions and comments on the system's structure were offered by Lee Barford, Alan Demers, Lane Hemachandra, John Hopcroft, Greg Johnson, Dean Kraft, Richard Palmer, Aliza Wechsler, and many others.

Lane Hemachandra provided excellent writing suggestions by reviewing previous editions of this paper.

Special thanks to John Hopcroft, the founder of the Cornell Robotics Project, who made useful suggestions about this project and this report, and who made the entire effort possible.
CHAPTER 6

REFERENCES


Figure 2

Procedure: Screw
Pitch: 12
Torque: 17
Figure 3
A. Establish Assembly Procedure’s Precondition
B. Perform Operation
C. Plan Grip
D. Plan Ungrip
E. Motion Plan to Grip
F. Motion Plan to start of Assembly Procedure

Figure 6
Motion "by magic"

Motion with robot attached

Figure 7