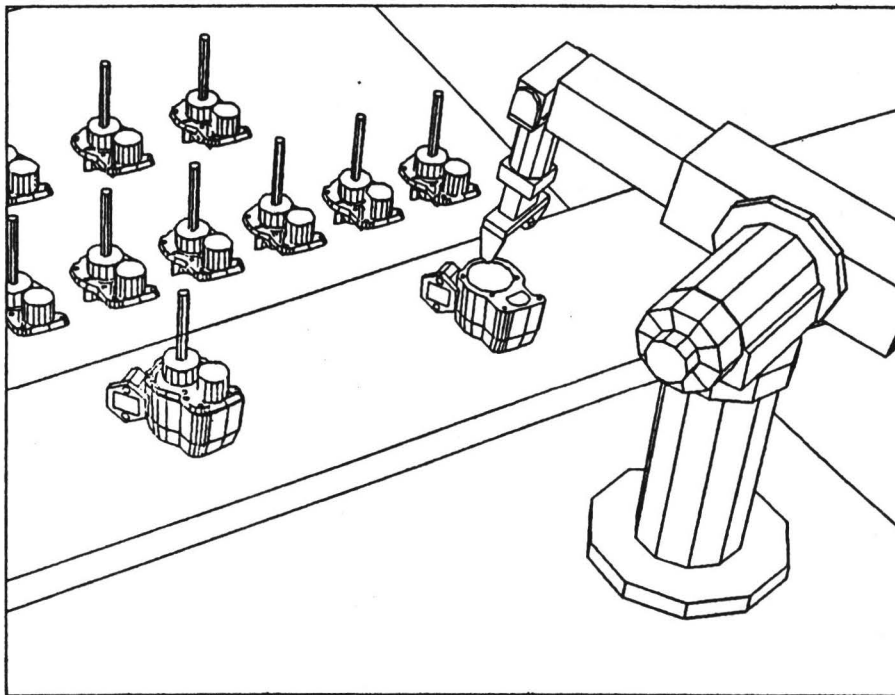


AN OVERVIEW OF AL, A PROGRAMMING SYSTEM FOR AUTOMATION

**Raphael Finkel, Russell Taylor, Robert Bolles,
Richard Paul*, Jerome Feldman***
Stanford Artificial Intelligence Laboratory
Stanford, California USA

June 3, 1975



* Jerome Feldman is now at the University of Rochester. Richard Paul is now at the Stanford Research Institute.

This research was supported in part by the National Science Foundation under contract No. GI-42906 and in part by the Advanced Research Projects Agency of the Office of Defense under Contract No. DAHC-15-73-C-0435. The views and conclusions in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the funding agencies.

ABSTRACT

AL is an high-level programming system for specification of manipulatory tasks such as assembly of an object from parts. AL includes an ALGOL-like source language, a translator for converting programs into runnable code, and a runtime system for controlling manipulators and other devices. The system includes advanced features for describing the motions of manipulators, for using sensory information, and for describing assembly algorithms in terms of common domain-specific primitives. This paper describes the design of AL, which is currently being implemented as a successor to the Stanford WAVE system.

AN OVERVIEW OF AL

This short paper cannot cover the subject of AL in depth; a complete discussion may be found in *AL, A Programming System for Automation*, Stanford Artificial Intelligence Laboratory Memo AIM-243, Stanford University Computer Science Department Report STAN-CS-74-456, by the authors of this paper.

INTRODUCTION

The development of robot manipulators such as the "Unimate" has led to the belief that these tools are in some way general-purpose devices and that they might be programmed like a computer. As a general-purpose programmable device, the robot manipulator provides a possible answer to the need for automation of assembly in batch manufacturing industries where small production runs rule out the use of special-purpose equipment.

We are implementing a system called AL for small scale batch manufacturing where setup time is the key factor. We rely on a symbolic database and previously-defined assembly primitives to minimize programming time. The system is capable of high-level planning and intelligent interpretation of user-defined primitives. The principal aim of this work is not to provide a factory floor programming system but rather to design a language which will be a tool for investigating the difficulty, necessary programming time, and feasibility of writing programs to control assembly operations.

PHILOSOPHY OF DESIGN

DATA AND CONTROL STRUCTURES

The principal mode of input to AL is textual, as opposed to spoken or manual (joystick). There are levels of complexity which are much more readily transmitted from man to machine through an interface of

symbolic text. For example, simultaneous motions of two arms and termination and error conditions are more likely to be unambiguously described through the medium of text because a textual language can provide a consistent framework for such intuitive ideas. Non-textual forms of input for defining target locations and suggesting arm trajectories to avoid collisions are most useful when applied in conjunction with a program text which supplies the overall intent of the programmer. The supervisor level of AL is simple enough to allow natural teaching by showing; it should be easy to interface such devices as joysticks and vocal input into AL, although we do not intend to do so at present.

Experience with languages like SAIL and WAVE has shown that text macros are a useful feature; they reduce the amount of repetitive typing. AL has a general-purpose text macro system interfaced into the scanner and parser.

The datatypes available include those types necessary to refer to one-dimensional measures (like distance, time, mass) and three-dimensional measures (like directed distance, locations, orientations). Arithmetic operators are available not only for the standard scalar operations like multiplication and addition, but also for such operations as rotation and translation.

Provision is made for simultaneous execution of several processes. This allows calculation and arm motion to take place simultaneously; several manipulators can be in independent or coordinated motion.

MOTION SPECIFICATIONS

Experience with WAVE has shown that calculating trajectories for manipulators is desirable but time-consuming. Trajectory calculations, together with other calculations which need only be performed once, are done at compile time. This allocation of effort drastically reduces the computing load at execution time and eliminates wasteful recomputation every time a sequence of actions is executed.

A wide range of exceptional conditions can occur during the motion of a manipulator. Appropriate action must be taken as soon as any of these occurs, for example to start up a new concurrent process or to notify the user. Therefore, AL allows the flexible specification of conditions to be monitored during motions (and during execution of blocks of code in general) and what to do in the case that a tested condition occurs.

USE OF A PLANNING MODEL

Since locations are not known exactly during the planning of a trajectory, there is a clear distinction between planned values and runtime values. Planned values are used for trajectory calculation; at runtime, trajectories are modified if necessary to account for

any discrepancies. The planned values are therefore a database on which trajectory calculations are computed.

Assembly tasks require that one object be affixed to another. We model this by having a semantic attachment between objects. If two objects are affixed, and one moves, the second one should move accordingly, that is, its planning value should be properly modified. The planning model includes information on attachments of objects. The affixment concept carries over to the runtime system, which does the equivalent modifications of the actual values. This saves the user the tedious bookkeeping operations required to determine where an object is after its base has been moved.

More generally, the compiler maintains a wide variety of information about expected runtime states. This includes information like the accuracy within which the planning value is known, how heavy an object is, how many faces it has on which it can rest, how wide the fingers of an arm should open to grasp it. This information may come from several sources, including explicit assertions by the user and built-in knowledge about the system hardware. AL has a general framework for representing and using such knowledge.

USE OF DOMAIN-SPECIFIC KNOWLEDGE

The system will eventually have enough domain-specific knowledge to allow programs to be written in terms of common assembly operations, rather than exclusively in terms of detailed single motions. At the simplest level, this involves a library of common assembly macro-operations that can be conditionally expanded to perform particular subtasks. Beyond this, we foresee an interactive system that can take a "high level" description of an assembly algorithm and fill in many of the detailed decisions required to produce a consistent and efficient output program.

A user will be able to specify different parts of a task at various levels of detail. The system is designed to accept explicit advice telling exactly how some particular subtask is to be accomplished. This is especially important for early versions of AL, which are not likely to be very "smart" and will therefore require a fair amount of explicit help. Other parts can be described by assertions which specify prerequisites and effects. The system will then complete the program incorporating the advice and satisfying the other assertions. The system can show the user how it is filling in the details to produce an output program, and why. This is very important both for debugging and for explaining to the user any requests for advice that it must make.

THE RUNTIME SYSTEM

The calculation of trajectories is time-consuming but not time-critical; servoing of devices is time-critical but not especially time-consuming. Therefore, the

compiler is written in a high-level language, SAIL, which runs on a large timeshared computer (a PDP-10) and the runtime system is designed to run on a dedicated minicomputer (a PDP-11/45).

The runtime system supports simultaneous execution of many processes. Several manipulators or devices might be running simultaneously, and each motor requires a separate process; several condition monitors might be active; several code segments (doing, perhaps, calculations) might be simultaneously active. Those processes which are dealing with real-time devices (joint servos and condition checkers) must be guaranteed service at regular intervals; the computation processes can fill in any time gaps.

The wide range of conceivable tasks implies that pure hardware servoing does not in general suffice. The reason for this is that hardware servoing restricts use to one of a small number of servo modes (typically position, velocity, or force), and has no provision for motions of accommodation or motions whose modes might change in midstream due to some software-detectable condition. Pure hardware servoing could not be readily modified to account for new feedback devices or methods. A philosophy of *software servoing* has these advantages: It is possible to program the manner in which feedback is to be used, to interface new types of sensors, to modify the servo while the arm is in motion, to supply the driving program with information concerning the success of the motion as well as to keep it up-to-date on the arm status. It also allows coordination of several arms, with one acting as a master and the others following. Hardware servoing would not save computation since the computer would need to perform an equivalent servo calculation in order to understand what the manipulator is doing.

GENERAL SYSTEM OUTLINE

HARDWARE

Currently two Stanford Electric Arms, built by Victor Scheinman [Scheinman], are available. They are called YELLOW and BLUE. Each has six joints and a hand that can open and close. The joints are controlled by electric motors; each joint has both position and velocity feedback. Motor drives are sent from the computer to the arm via a digital-to-analog converter (D-to-A); feedback signals are routed through an analog-to-digital converter (A-to-D) back to the computer.

Various other devices are designed and implemented as needed. We use tools, jigs and special markings for several purposes: to render a task possible (an example is the arm itself), to improve efficiency (a mechanical screwdriver), and to overcome some of our sensory and mechanical limitations (a screw dispenser).

SOFTWARE

See figure 1 for a picture of the system.

The SUPERVISOR is the top level of AL. It runs on the timesharing computer and provides an interface between the user and the other parts of the system: 1) listening to the user's console and interpreting simple command language input; 2) controlling the compiler, starting it and relaying its error messages back to the user; 3) signalling the loader when it is necessary to place compiled code into the mini; 4) handling the runtime interface to the mini.

The USER sits at a console and makes requests of AL. These fall into several categories: compilation, loading, execution of programs, debugging of code, requesting of status information, asking for immediate arm motion, saving and restoring the state of the world at safe points, requesting explanation of certain compiler decisions. There are two different consoles at which a user can sit: one is connected to the timesharing computer, through which he can speak to the supervisor and all the parts of AL residing on the timesharing computer; the other is connected to the mini, and through it the user can investigate the runtime system and cause modifications.

The COMPILER reads AL programs from files (or, optionally, directly from the user's console) and produces load modules. The compiler is divided into three phases: The PARSER, which produces parse trees of the program, the EXPANDER, which expands those parse trees by replacing high-level primitives with low-level primitives, and the TRAJECTORY CALCULATOR and CODE GENERATOR, which creates the output files.

The LOADER takes the load modules prepared by the compiler and enters them into the mini's runtime system. Address relocation and linking are done at this time. The loader also sets up the data area in the runtime interface in the timesharing computer; these data include output strings, procedure linkages, and information necessary for diagnostic purposes during runtime. Loading is often done in a partially incremental fashion, installing new code following previously loaded code.

The RUNTIME INTERFACE, which resides in the timesharing computer, is charged with initiating the mini program, fielding procedure calls from the running program to procedures on the timesharing machine, returning values from these procedures, and fetching values from the mini for debugging purposes. The interface has the power to interrupt the execution of the program and to modify the status of the runtime system, for example, by patching in additional programs or modifying the values of some variables. This allows the user to control the program through the timesharing computer.

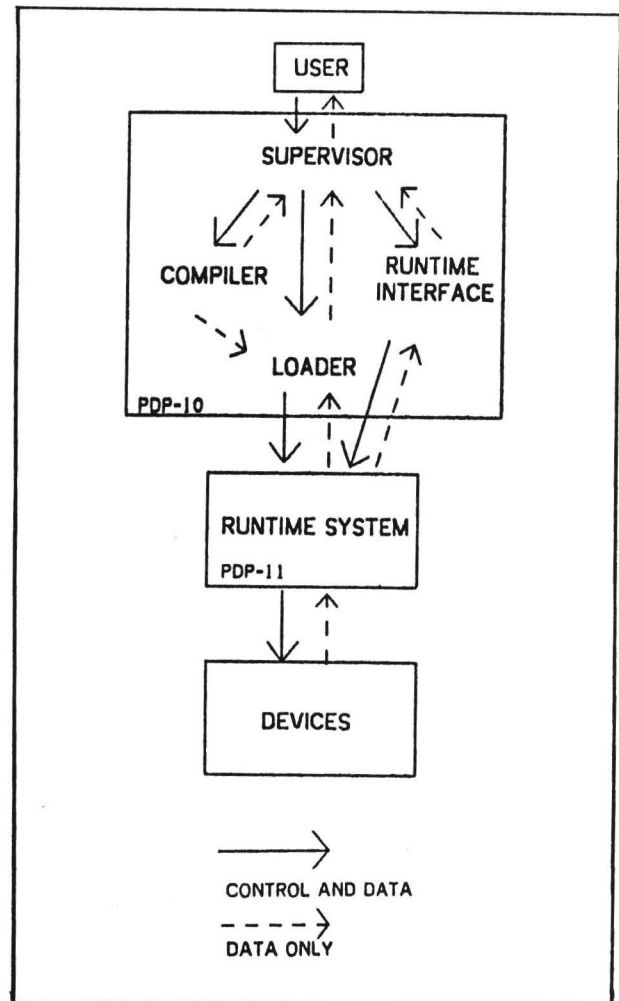


FIGURE 1.

The RUNTIME SYSTEM is the set of programs which reside in the mini. This system includes kernel programs for time-slice cpu sharing and process control and a set of dynamically created processes. These are of three basic types: a) An INTERPRETER examines the code prepared by the compiler and executes the numeric computations requested. When a move is to be started, the interpreter creates a servo for each joint and waits until all these servos are finished. b) A SERVO handles the motion of one moving joint. c) A CONDITION-MONITOR repeatedly examines certain conditions (whatever the programmer has specified). If it should discover that its condition has occurred, it creates an interpreter to take appropriate action. The runtime system also includes routines for communication with the runtime interface in the timesharing computer.

AN EXAMPLE

A simple task dealing with the objects shown in Figure 2 illustrates some of the features of AL:

1. Pick up the bracket with the YELLOW arm and position it next to the beam so that the holes line up,
2. Pick up the bolt with the BLUE arm,
3. Fasten the bracket to the beam by inserting the bolt in the holes,
4. Return the arms to their park positions.

Some of the features to be demonstrated are: the affix structure, reference frames, dimensions, and multi-processing. We demonstrate these capabilities by presenting a highly commented AL program to accomplish the task stated above. Except for the missing macro bodies, this program is complete; it could perform as indicated.

AL is a multi-level programming language; at one extreme the user can write detailed, system-like programs and at the other he can describe the tasks and any partial ordering among them and let the system determine necessary details. Our example is written in an intermediate level. In particular, it assumes that there are several general-purpose macros and routines which understand how to GRASP and RELEASE things and carry out a NORMAL_SEARCH to insert something into a hole.

Capitalized words in the example are key words within AL. Lower case words are user-defined identifiers. Comments are surrounded by curly brackets.

whole_task: BEGIN

{First declare the necessary FRAMES and describe how they are initially related. A FRAME is a coordinate system. It has two components: the location of the origin (a distance VECTOR) and the orientation of the axes (a ROT). Frames are typically used to describe objects and important features of objects. There are several predeclared frames in AL. STATION is the frame which represents the work station's frame of reference. Each hand available to the system also has a frame variable, whose value (continually updated) is the position of that hand. Currently there are two such frames: YELLOW and BLUE.}

The attach structure representing the initial world is shown in Figure 3. The arrows indicate how the movement of a frame affects other frames. If a frame at the tail of an arrow is moved (by the arm, visually updated, etc.) the frame at the head of the arrow will be automatically updated. The double arrows are the results of RIGIDLY AFFIXING one frame to another.}

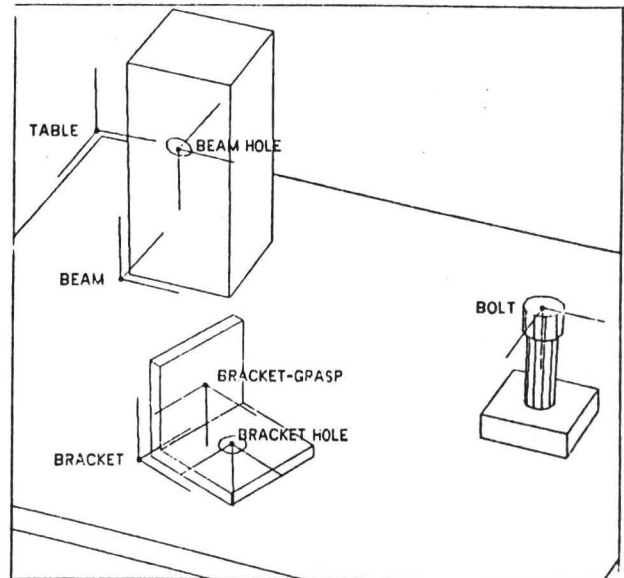


FIGURE 2.

```
FRAME beam, beam_hole;      FRAME bolt;
FRAME bracket, bracket_hole, bracket_grasp;
beam ← FRAME(ROT(Z, 90*DEG), VECTOR(10,6,0));
```

{The beam is expected to be positioned at (10,6,0) in the station's coordinate system (the default unit for distance measurements is centimeters) and rotated 90 degrees about the station's Z vector. AL knows about dimensions like DEG for degrees. Dimensions are adjuncts to variable types; new ones can be defined in terms of old ones.}

```
beam_hole ← beam*TRANS(ROT(X, 90*DEG),
                        VECTOR(3,0,7));
```

{The TRANS represents the position of the beam_hole with respect to the beam. The premultiplication by the frame beam positions the beam_hole in the station's coordinate system.}

```
AFFIX beam_hole TO beam;
```

{As shown in figure 3}

```
ASSERT FORM(DEPROACH, beam_hole,
            TRANS(NILROT, VECTOR(0,0,-3)));
```

{Trajectories consist of a path from the current position, through a departure point, possibly through some via points, through an approach point, and finally to the destination. The primary use of via points is to avoid collisions during the motion. All FRAMES have a DEPROACH TRANS associated with them. TRANSes are essentially the same as FRAMES. Whenever leaving (or moving to) a FRAME the standard departure (or approach) used is that

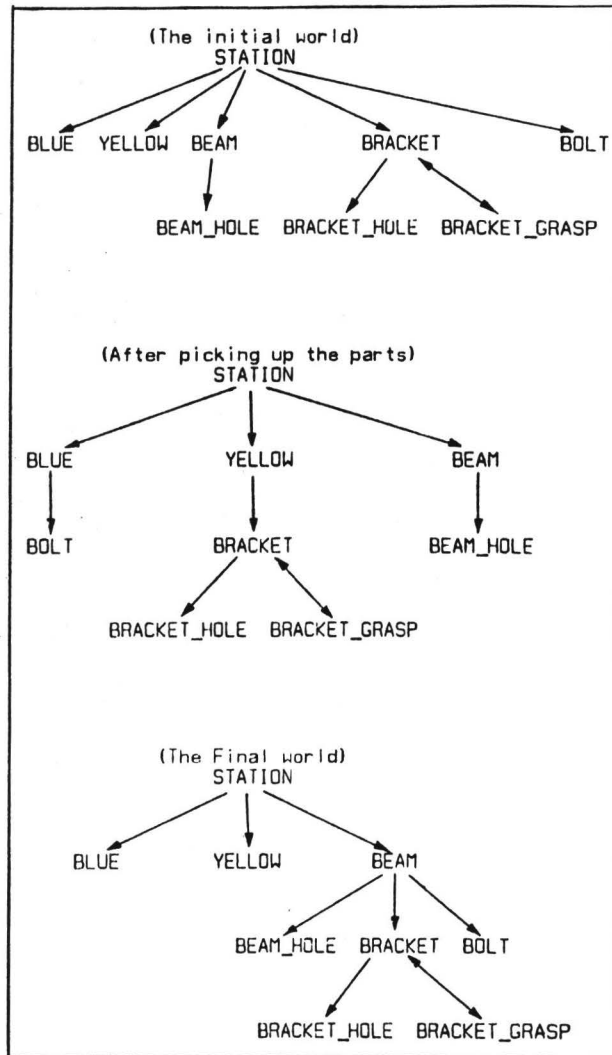


FIGURE 3.

FRAME's DEPROACH TRANS. The station has a DEPROACH TRANS which is three inches above it. Whenever one FRAME is AFFIXed to another, by default the former takes on the latter's DEPROACH. The result of this ASSERT is that the arms will approach the beam hole from the side instead of from above.

```

bracket ← FRAME(ROT(Z,45*DEG),
    VECTOR(20,14,0));
bracket_hole ← bracket*
    TRANS(ROT(X,180*DEG),VECTOR(3,3,0));
AFFIX bracket_hole TO bracket;
bracket_grasp ← bracket*
    TRANS(ROT(X,180*DEG),VECTOR(0,3,3));
AFFIX bracket_grasp TO bracket RIGIDLY;

```

{The RIGID AFFIXment insures that a change of bracket_grasp will automatically change bracket, which in turn will automatically change bracket_hole. This is quite convenient if the position of the whole 'object' is being updated by one grasping position (ie. bracket_grasp).}

```

bolt ← FRAME(ROT(Z,90*DEG)*
    ROT(X,180*DEG),VECTOR(16,30,0));

```

{The rotation portion of the FRAME has been specified as a composition of two primitive rotations.}

```

DEFINE OZ = "72.007789*DYNES";
DEFINE INCHES = "2.54*CM";

```

{Some of the standard macros are defined next.}

```

DEFINE grasp
    (TRANS special_departure,special_approach;
    FRAME ATOM the_arm(DEFAULT YELLOW);
    FRAME object,grasp_point,
        thing_object_affixed_to;
    DISTANCE SCALAR opening_before_departure,
        opening_for_approach(DEFAULT 15*CM),
        thickness(DEFAULT 0.1*INCHES))
    = "{body of macro goes here}";

```

{The expansion of such a macro can depend upon the supplied arguments, the DEFAULT arguments, and any values in the current planning model.}

```

DEFINE release
    (FRAME ATOM the_arm(DEFAULT YELLOW);
    FRAME the_object,the_new_parent;
    DISTANCE SCALAR the_opening
        (DEFAULT 15*CM))
    = "{body of macro goes here}";

```

```

DEFINE normal_search
    (FRAME ATOM the_arm(DEFAULT YELLOW);
    DISTANCE SCALAR increment(DEFAULT .3*CM),
    distance_fwd;
    FORCE SCALAR stopping_force;
    SCALAR number_of_tries(DEFAULT 9))
    = "{body of macro goes here}";

```

{This would include some automatic error recovery and a call to the operator if something drastic goes wrong.}

COBEGIN

{This COBEGIN-COEND construction describes two independent subtasks (one for YELLOW and one for BLUE) which can be executed in any order determined by the runtime system; in parallel or serially. This, of course, assumes that the two arms work in completely separate parts of the workstation so there is no possibility of a collision.}

ypickup: BEGIN {pick up the bracket with yellow}
grasp(object=bracket,grasp_point=bracket_grasp,
opening_for_approach=3*CM);

{Only the necessary parameters need to be specified. By default the YELLOW arm will be used and there will be no special approaches or departures. One effect of grasp is to AFFIX the object to the arm.}

MOVE bracket_hole TO beam_hole +
VECTOR(0,0,-3) WRT beam_hole;

{The YELLOW arm (since it holds the bracket to which bracket_hole is AFFIXed) positions itself so that the bracket_hole lines up with the beam_hole, but is .3 cm away from the beam_hole. The WRT operator is one way of describing a vector within a frame of reference other than the station's.}

MOVE YELLOW TO * +
VECTOR(0,0,.6) WRT beam_hole
ON FORCE(Z WRT beam_hole)>50*OZ DO
STOP YELLOW
ON ARRIVAL DO
ABORT("ERROR* bracket went too far");

*{The * represents the current position of the arm. The arm moves .6 cm in Z relative to the beam_hole's frame. The purpose of this move is to push the bracket up against the beam. If the beam is there, the arm will sense an opposing force. If not, the arm will succeed in moving forward the prescribed .6 cm. In order to check for these possibilities, two condition monitors have been included with the MOVE statement. The first one monitors the force and stops the arm if the force exceeds 50 ounces (which means everything is ok). The second one is an interrupt type condition. If the arm successfully carries out the complete MOVE, this monitor is awakened, the message is printed, and control is given to an operator.}*

END ypickup;
bpickup: BEGIN {pick up the bolt with blue}
grasp(the_arm=BLUE,the_object=bolt,grasp_point=bolt,
opening_for_approach=3*CM);
END bpickup;
COEND;

{Figure 3 shows the world after the parts have been picked up.}

MOVE bolt TO beam_hole +
VECTOR(0,0,-5.3) WRT beam_hole;

{The BLUE arm positions itself so that the bolt is lined up with the beam_hole and its tip is .3 cm away from the outside of the bracket_hole.}

normal_search(BLUE, 2*CM, 1.6*CM, 60*OZ, 9);

{This pushes the bolt through the bracket_hole and partly into the beam_hole. If control continues past this statement the bolt is assumed to be partly in the hole.}

MOVE BLUE TO * +
FRAME(ROT(Z,90*DEG),VECTOR(0,0,4))
ON FORCE(Z WRT BLUE) > 60*OZ DO
STOP BLUE;

{This pushes and twists the bolt into the hole. When the force exceeds 60 ounces, the bolt is assumed to be completely seated in the hole. There is no check to make sure the bolt seats properly.}

COBEGIN
parky: BEGIN {release the bracket and park}
release(the_object=bracket,
the_opening=3*CM,the_new_parent=beam);
MOVE YELLOW TO YPARK;
END parky;
parkb: BEGIN {release the bolt and park}
release(the_arm=BLUE,the_object=bolt,
the_opening=3*CM,the_new_parent=beam);
MOVE BLUE TO BPARK;
END parkb;
COEND;
END whole_task

CONCLUSIONS

AL is important for several reasons. It shows what sort of considerations are necessary for flexible control of mechanical manipulation. It demonstrates the feasibility of programmable assembly. It provides a research tool for investigation of new modes of software servoing, assembly primitives, arm-control primitives, and interactive real-time real-world systems.

AL is currently limited by the lack of certain features which would make it more competent. Many of these have to do with the fact that feedback is used only in a threshold way; either a monitor triggers or it does not. Fine control of the arm would be enhanced by more sensitive force-sensing elements on the hand and a means of programming accommodating, non-threshold response to this sensory input. Visual feedback should be implemented to provide better positioning capability, error detection, and error recovery. Moving assembly lines imply that AL should be able to understand motions which it does not cause directly through manipulation; objects should have a dynamic capability. Collision detection and avoidance remain difficult issues. AL would be more error-free if the trajectory calculator could ensure that the arms never interfere with each other or with objects in the current world.

BIBLIOGRAPHY

- [Bolles and Paul] R. C. Bolles, R. Paul, *The Use of Sensory Feedback in a Programmable Assembly System*, Stanford Artificial Intelligence Project, Memo No. 220, October 1973.
- [Finkel] R. Finkel, R. Taylor, R. Bolles, R. Paul, J. Feldman, *AL, A Programming System for Automation*, Stanford Artificial Intelligence Project, Memo No. 243, November 1974.
- [Inoue] H. Inoue, *Force Feedback in Precise Assembly Tasks*, Massachusetts Institute of Technology A. I. Memo No. 308, August 1974.
- [Nevins] J. L. Nevins, D. E. Whitney, S. N. Simunovic, *System Architecture for Assembly Machines*, The Charles Stark Draper Laboratory, Inc., Memo No. R-764, November 1973.
- [Paul] R. P. Paul, *Modelling, Trajectory Calculation and Servoing of a Computer Controlled Arm*, Stanford Artificial Intelligence Project, Memo No. 177, March 1973.
- [Rosen] C. Rosen, et. al., *Exploratory Research in Advanced Automation*, Stanford Research Institute Report, December 1973.
- [Scheinman] V. D. Scheinman, *Design of a Computer Manipulator*, Stanford Artificial Intelligence Project, Memo No. 92, June 1969.
- [Will] Peter M. Will and David D. Grossman *An Experimental System for Computer Controlled Mechanical Assembly*, IBM Research Report RC 4922, Yorktown Heights, New York, July, 1974.

