Interactive Generation of Object Models with a Manipulator

DAVID D. GROSSMAN AND RUSSELL H. TAYLOR

Abstract—**Manipulator programs in a high-level language consist of manipulation procedures and object model declarations. As higher level languages are developed, the procedures will shrink while the declarations will grow. This trend makes it desirable to develop means for automating the generation of these declarations. A system is proposed which would permit users to specify certain object models interactively, using the manipulator itself as a measuring tool in three dimensions. A preliminary version of the system has been tested.**

INTRODUCTION

Manipulator Languages and Object Models

IMPORTANT PROBLEM areas for current research on manipulators are function, ease of programming, and speed and reliability of execution. The domain of function is concerned with whether or not certain tasks are technically feasible, without regard to economic issues. For the real-world environment of industrial assembly, function has been demonstrated by assembling a water pump with the Stanford arm [1] and by similar experiments at other laboratories.

Since 1973, the main thrust of the Stanford effort has been directed at ease of programming. The approach taken was to design a new high-level language called AL [2] in which assembly programs would be much simpler to code than in any previously existing manipulator language. A summary of the AL language is given in the Appendix. The AL system began operation at the end of 1975.

Most other laboratories and companies have chosen to pursue a teaching by doing approach to manipulator programming instead of developing high-level languages [3], [4]. In the short run, programming by guiding is easier than programming in a formal language. In the long run, however, there is good reason to believe that the greater generality offered by a system incorporating formal languages will be desirable.

The ultimate goal of research on high-level manipulator languages is a language in which very few statements are needed to describe a highly complex assembly. For an object with N parts, perhaps a realistic goal would be to have a language in which the assembly can be described in about N statements.

In view of this goal, the level of manipulator languages is best measured not by the richness of their computer science content, but rather by the number of source statements required to code specific applications programs. Based on this criterion, AL is the highest level manipulator language developed to date, in spite of the fact that it lacks arrays, lists, string variables, formatted I/O, and so forth. In particular, AL is certainly higher in level than MANTRAN [5], WAVE [1], and ML [6], [7] though lower in level than the proposed AUTOPASS [8].

It is instructive to extrapolate from AL to an absurd ultimate high-level language. In this ultimate language, a program for assembling 1000 water pumps might have this form:

```
DECLARE WATER_PUMP etc.;
MAKE_PLAN(WATER_PUMP,
  ASSEMBLY_PLAN);
EXECUTE_PLAN(ASSEMBLY_PLAN,1000);
```

Such a program presumes that all the relevant artificial intelligence problems have been solved and embodied in MAKE_PLAN and all the manipulation problems have been solved and embodied in EXECUTE_PLAN. The important observation to make with respect to this absurd example is that as far as the user of such a language is concerned, all the effort of writing the program would be in expanding the "etc." in the first line.

The expansion of the "etc." constitutes the world model used by MAKE_PLAN. This world model is a complex data base, including such information as the specification of mechanical parts, component hierarchies, affixment relationships, geometric shapes, Cartesian transformations between features, material properties, and so forth. It is clear that the declaration of such a detailed world model would be a lengthy process, possibly requiring hundreds of lines of text.

The development of high level manipulation languages, therefore, will shift the problem from writing *procedures* to writing *declarations*. Already in AL a sufficiently high level has been reached that this new problem is becoming significant.

A rough measure of the importance of declarations as compared to procedures may be obtained by looking at sample AL programs [2] and taking the ratio of lines of code

Manuscript received June 21, 1976; revised October 16, 1977. This work was supported in part by the National Science Foundation under Contract NSF APR 74-01390-A02 and in part by a grant from the Alcoa Foundation.

The authors were with the Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, CA 94305. They are now with the Computer Sciences Department, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.

0018-9472/78/0900-0667$00.75 © 1978 IEEE

```
FRAME beam,bracket,bolt;
FRAME bracket_bore,beam_bore;
FRAME bolt_grasp,bracket_handle;

ASSERT FORM (TYPE,beam,object);
ASSERT FORM (GEOMED,beam,"beam.B3D");
ASSERT FORM (SUBPART,beam,beam_bore);
AFFIX beam_bore TO beam RIGIDLY
      AT TRANS(ROT(Y,90•DEG),VECTOR(0,1.5,6));

ASSERT FORM (TYPE,bracket,object);
ASSERT FORM (GEOMED,bracket,"bracket.B3D");
ASSERT FORM (SUBPART,bracket,bracket_bore);
ASSERT FORM (SUBPART,bracket,bracket_handle);
AFFIX bracket_bore TO bracket RIGIDLY
      AT TRANS(ROT(X,180•DEG),VECTOR(5.1,2,0));
AFFIX bracket_handle TO bracket RIGIDLY
      AT TRANS(ROT(X,180•DEG),NILVEC);

ASSERT FORM (TYPE,bolt,SHAFT);
ASSERT FORM (DIAMETER,bolt,0.5•CM);
ASSERT FORM (TOP_END,bolt,head_type_1);
ASSERT FORM (BOTTOM_END,bolt,tip_type_1);
ASSERT FORM (TYPE,tip_type_1,FLAT_END);

   . . .

bracket ← FRAME(NILROT,VECTOR(20,40,0));
beam ← FRAME(NILROT,VECTOR(10,60,0));
bolt ← FRAME(ROT(Y,180•DEG),VECTOR(30,50,5));
```

Fig. 1. Section of AL example.

```
FRAME beam,bracket,bolt;
FRAME bracket_bore,beam_bore;
FRAME bolt_grasp,bracket_handle;

ASSERT FORM (TYPE,beam,object);
ASSERT FORM (TYPE,bracket,object);

ASSERT FORM (SUBPART,beam,beam_bore);
ASSERT FORM (SUBPART,bracket,bracket_bore);
ASSERT FORM (SUBPART,bracket,bracket_handle);

AFFIX beam_bore TO beam RIGIDLY
      AT TRANS(ROT(Y,90•DEG),VECTOR(0,1.5,6));
AFFIX bracket_bore TO bracket RIGIDLY
      AT TRANS(ROT(X,180•DEG),VECTOR(5.1,2,0));
AFFIX bracket_handle TO bracket RIGIDLY
      AT TRANS(ROT(X,180•DEG),NILVEC);

bracket ← FRAME(NILROT,VECTOR(20,40,0));
beam ← FRAME(NILROT,VECTOR(10,60,0));
bolt ← FRAME(ROT(Y,180•DEG),VECTOR(30,50,5));
```

Fig. 2. Simplified AL example.

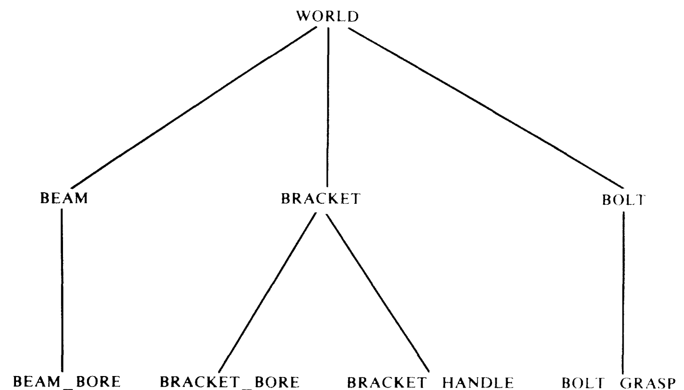

Fig. 3. Subpart hierarchy of AL example.

in the two categories. Neglecting comments, three low-level AL examples have declaration to procedure ratios of 1:2, 1:6, and 1:3. A very high-level AL example, however, has a declaration to procedure ratio of 6:1. Actually, the importance of declarations is even greater than these ratios would indicate, since the declarative statements tend to be far more difficult to code than the procedural statements. Even at the lowest level of AL, appreciable time can be spent in defining simple models for the initial locations of objects and their affixment structure.

Since the motivation behind high-level manipulator languages is ease of programming, and since these languages are shifting the problem from procedures to declarations, it is natural to consider means for simplifying the generation of these declarations. This report proposes in some detail a prototype system which would semi-automate the process of specifying a large class of AL declarations. An initial version has been built and successfully tested. It is believed that a system of this type would be general enough to be applicable to other high-level manipulation languages also.

Attributes of AL World Models

The AL language has evolved considerably during implementation, and the specifications contained in the original publication [2] have become slightly obsolete. In order to keep the discussion on a concrete footing, therefore, a summary of AL is given in the Appendix to this paper, and it will be assumed that AL declarations take exactly the form shown there.

Fig. 1 shows a section of a very high-level AL example. All of the statements shown are considered to be declarations of the world model since they occur prior to the first motion of the manipulator.

The method of automating declarations to be described here is inappropriate for specifying information about the geometric shape of objects. Therefore, all statements which relate to geometric shape either in the form of Geomed [9] files or in the form of generic parts such as Shaft will be omitted from further consideration.

Fortunately, it is possible to write fairly high-level AL programs which do not need shape information, although in an ultimate high-level language, of course, geometric shape declarations will have to be provided. The eventual automation of shape declaration will some day come about through advanced interactive graphics and vision systems. It is important to realize, however, that while this major problem area is being bypassed in this discussion, the problems which remain are still substantial, and their practical solution would make it significantly easier to code AL programs.

The remaining statements may be rearranged according to their function as shown in Fig. 2. The net effect of these statements is to describe a tree in which the nodes represent physical objects and the arcs are relationships between them. The root of the tree is an implicit object which may be called "world," and all objects which are not subparts of anything else are subparts of world. The subpart hierarchy is shown in graphical form in Fig. 3.

The arcs have the implication of subpart or, equivalently, of subfeature. Each arc also must contain information concerning the relative transformation between the Cartesian frame of the parent and that of the son. For those arcs which emanate from world, these transformations are AL frames, while for all other arcs they are AL transes. Finally, each arc must specify whether the relationship is rigid, nonrigid, or independent.

```
FRAME(NILROT,VECTOR(20,40,0));
FRAME(NILROT,VECTOR(10,60,0));
FRAME(ROT(Y,180*DEG),VECTOR(30,50,5));

TRANS(ROT(Y,90*DEG),VECTOR(0,1.5,6));
TRANS(ROT(X,180*DEG),VECTOR(5.1,2,0));
TRANS(ROT(X,180*DEG),NILVEC);
```

(a)

```
FRAME(ROT(Z,60*DEG),VECTOR(40,30,0));
FRAME(ROT(Z,90*DEG),VECTOR(10,60,0));
FRAME(ROT(Z,90*DEG)*ROT(X,180*DEG),VECTOR(30,60,5));

TRANS(ROT(X,-90*DEG),VECTOR(5.1,0,15));
TRANS(ROT(X,180*DEG),VECTOR(5.1,5,0));
TRANS(ROT(X,180*DEG),VECTOR(0,5,5));
```

(b)

Fig. 4. (a) Intended frames and transes for AL example. (b) Actual frames and transes for AL example.

The principal difficulty for the programmer in specifying this tree lies in the symbolic definition of the frames and transes, which are summarized in Fig. 4(a). Some indication of the magnitude of the difficulty of accurately coding frames and transes is given by the fact that the figure in the AL publication [2] which purports to show the initial world defined by these declarations actually corresponds to the declarations given in Fig. 4(b).

Of these declarations, the third one is the only one which contains composite rotations. This example, however, involves objects with nice rectangular shapes, so that relative transformations between features are comparatively simple. In the real world of industrial parts, complex composite transformations will occur frequently, increasing the likelihood of programmer error, and vastly increasing the labor of accurate coding. The solution of this specific problem is the motivation behind the method of automating world model declarations proposed here.

Alternative Approaches to Generating the Models

Besides using textual definitions, there are at least three alternative approaches to the problem of generating AL models: graphics, vision, and pointing.

Graphics is concerned with building the required data structures with an interactive graphics system. There have been several publications describing graphics systems which have the potential to be applied to this problem [9]–[14]. The main difficulty with this approach is that it is impossible to use a graphics system to specify frames and transes without first specifying the shapes of all the objects. Even though the systems cited allow complex object shapes to be represented by unions and intersections of simpler shapes, the specification of an object's shape is bound to be more difficult than the specification of a few of its features.

Vision is concerned with showing the work station and the component parts to a camera, and having vision programs automatically or interactively generate the data structure. In the environment of complex industrial parts, a completely automatic vision system capable of constructing AL world models is far beyond the current state of the art. However, one can imagine an interactive vision system in which the user identifies features by simply pointing to them, either in the original scene or in a projected image of the scene. Suitable pointers for the original scene include arrays of light emitting diodes [15], a lighted wand [3], [16] or other easily identified stick, a laser [3], or even a specially marked glove. For pointing in a projected image of the scene, a movable cursor or some commercial digitizer could be used. In either case, accurately locating a feature on an object of unmodeled shape in three dimensions requires stereo vision or depth ranging, so that building such a system would not be a trivial undertaking.

Pointing, which is the method proposed here, involves an interactive system in which the data structure is built by using the manipulator and a special tool to point to the objects and features. The manipulator may be moved about manually, or under joystick or pushbutton control. It is possible to infer the position of the feature from the known joint angles of the manipulator. The accuracy of this method is inherently comparable to the accuracy with which the manipulator can perform the assembly itself.

Pointing with the manipulator is much easier than vision to implement, although vision would be somewhat more convenient to use. Aside from the fact that vision and pointing use different techniques to determine positions, the two approaches are essentially similar, and any system built for one approach could be adapted for the other.

Pointing with a Manipulator

Implicit Specification of Frames

The typical manipulator has six degrees of freedom, which allow it to be positioned at an arbitrary position and in an arbitrary orientation. Frames also have six degrees of freedom, corresponding to three directions of translation and three angles of rotation. It follows that if a single pointing of the manipulator is to imply a unique frame explicitly, there are no spare degrees of freedom. The absence of spare degrees of freedom makes it quite difficult to position the manipulator accurately.

This problem frequently arises in the context of manipulators which are programmed by guiding them through the motions. It is not hard to guide a manipulator manually to a good grasping position to pick a part out of a pallet. However, it can be quite difficult to guide it manually to a good *orientation*. As a result, when the gripper grasps the part it may rotate the part slightly, causing the part to bind in the pallet. The need for orientation accuracy becomes much more crucial when it is being used to define a world model, since any angular error may be multiplied by some long moment arm in the AL program, whereas in a guiding system the process of grasping is always spatially localized at the point where the angular error was made.

In order to avoid this difficulty, it is convenient to be able to define frames *implicitly* as well as *explicitly*. A suitable way of defining a frame implicitly is to use multiple pointings. The first pointing may define the *origin* of the frame, the second may define one *axis* of the frame, and the third may define one *plane* of the frame. In this manner, each pointing determines position only, and there is no need to have orientation accuracy.

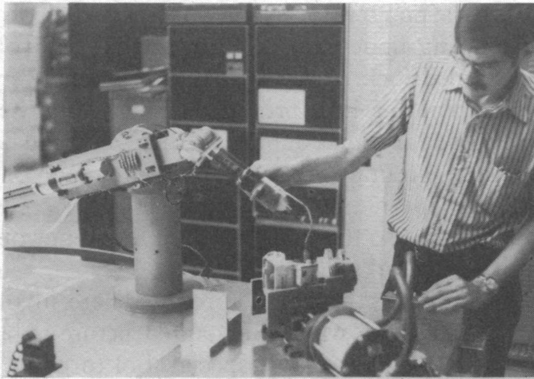

Fig. 5. Stanford arm with bendy pointer.

The Bendy Pointer

The manipulator extremity must be provided with some sort of sharp pointer so that it can be used as a precise measuring tool. The pointer must have a shape suitable for reaching into awkward places such as the inside of a screw hole, the interior of a box, and so forth. In order to make the shape of the pointer compatible with all kinds of unforseen obstructions, it is desirable to design a pointer which may be bent by the user into an arbitrary shape. Such a special device will be referred to as a "bendy pointer."

Whenever the user wishes, he may deform the bendy pointer into any new configuration which appears to be convenient for the next pointing operation. Having deformed the pointer, the user must calibrate its new end position by using the pointer to point to a standard fiducial mark at a known location in the laboratory. From the frame of the fiducial and the frame of the gripper, the system can infer the transformation which takes the gripper frame into the bendy pointer. The fiducial mark itself may also be moved around the work station and calibrated by grasping it by the manipulator gripper without the bendy pointer.

A possible refinement to the bendy pointer would be to provide it with some sort of terminal sensor. For example, if the object being measured were metallic, one could construct an electric circuit and measure current through the pointer. Another possibility would be to use some sort of infrared or fluidic device. Adding a sensor would improve the sensitivity of locating the pointer accurately, but it would also create problems of compatibility with being able to bend the device. For this reason, it will be assumed that there is no terminal sensor, and the user is required to eyeball the positioning.

One way of fabricating a bendy pointer is to have a 6-in length of wire coat hanger protruding from a metal block which is suitably shaped for grasping by the manipulator. Such a pointer is shown in Fig. 5. Another possibility would be to use a linkage made from two or three thin rods connected by ball joints with clamps. A bead and tendon chain might also be suitable. An alternative to the bendy pointer would be a tool set consisting of an assortment of rigid pointers of commonly useful shapes which could be quickly attached or detached. Whatever type of pointer is used, it must be reasonably rigid under gravity.

```
WORLD
    FIDUCIAL
    ARM
        POINTER
    BEAM
        BORE
    BRACKET
        BORE
        HANDLE
    BOLT
        GRASP
```

Fig. 6. Subpart hierarchy for hypothetical protocols.

Prior Art

The idea of guiding a manipulator to teach it a sequence of motions is well known and in common use at many laboratories and in several commercial products. However, adopting a guiding approach towards the generation of world models as proposed here has not been previously reported.

The idea of using a bendy pointer with a manipulator has not appeared in the literature, although this idea is related to the previous use of a rigid retractable pointer. Such a rigid pointer with a sensing capability was initially installed on the IBM Research manipulator in 1973 as a means for detecting touch with low inertia, a capability which was used, for example, to determine orientation of parts [17]. Subsequent to its installation, this sensing device was frequently used as a pointer to a set of fiducial marks in order to verify that the manipulator's calibration had not drifted.

Positioning a mechanical pointer in 3-d space is in some sense analogous to the action of moving and rotating a cursor in a 2-d graphics system. In a 3-d graphics system, the cursor has three translational and three rotational degrees of freedom. While there is no mathematical difficulty in moving a 3-d cursor, there is a substantial human problem in orienting the cursor or even in simply visualizing an oriented frame. One of the authors has dealt with this problem in the context of representing 3-d objects [10] and found it desirable to use implicitly defined frames rather than explicitly defined frames to specify how objects were to be attached; the actual coding of the implicit frame definition routines was done by Roger Evans, who subsequently coded similar routines into the ML language used with the IBM arm [6]. The latter routines are used to specify absolute frames, but there is no notion of subpart hierarchies with affixment relations, and no notion of relative transes.

Conversations with members of the Stanford Hand-Eye project indicate that several people had considered many if not most of these ideas back in 1973, but at that time the need to design and implement AL superseded the need for interactive world model generation.

Hypothetical Protocols

Initial State

This section presents hypothetical protocols between the proposed system and a user who wishes to generate the world model corresponding to the tree shown earlier in Fig. 3. The subpart hierarchy is summarized in Fig. 6. The reasons for including the fiducial mark, the arm, and the pointer in this hierarchy will become clear later on when

the topics of system architecture and display facilities are discussed. At the start, when the system is first turned on, the subpart hierarchy consists only of the objects shown in the first four lines of Fig. 6.

In the protocols which follow, system output is shown in upper case, user responses are shown in lower case, and explanatory material is contained in curly brackets {like this}. When the system is ready for further user input, it prompts the user with an asterisk {*}.

Protocol to Build Subpart Hierarchy

* new bracket {This means that a new object called bracket is to be added to the subpart hierarchy.}
* nonrigid {This specifies that bracket is nonrigidly affixed to world.}

* {Here the user moves the manipulator to point to the bracket frame. The details of the protocols for this type of operation are given in the next section.}

* $d:\leftarrow$bracket {This sets a context in which all future new objects are subparts of bracket until the context is changed. Here "d:" stands for "dad." When the system was first turned on, the initial context was $d:\leftarrow$ world.}
* new bore {These two steps add a new object called bore to the hierarchy as a subpart of bracket and cause bore to be affixed rigidly
* rigid to bracket.}

* {Here the user moves the manipulator to point to the bore frame. The details of the protocols for this type of operation are given in the next section.}

* new handle
* rigid

* {Here the user moves the manipulator to point to the handle frame.}

* $d:\leftarrow$world {All future new objects are subparts of world again.}

* new beam
* nonrigid

* {The user points to the beam frame.}

* $d:\leftarrow$beam {Future new objects are subparts of beam.}
* new bore {The system is capable of distinguishing beam bore from bracket bore.}

* rigid
* {The user points to the bore frame.}

* $d:\leftarrow$world
* new bolt
* nonrigid

* {The user points to the bolt frame.}

* $d:\leftarrow$bolt
* new grasp
* rigid

* {The user points to the grasp frame.}

*

Protocol to Point to Frame

{Suppose the user bends the pointer into a new shape. He now wishes to calibrate it.}

* setfid {This macro expands into "free; $a:\leftarrow$arm; $\leftarrow$fiducial; inverse; *; pointer:$\leftarrow a$:↑." The "free" releases the arm brakes so the user can move the manipulator manually so that the pointer points to the fiducial mark. When he has done so, he pushes the "stop arm" button to proceed. The remaining primitive stack operations compute the correct pointer transform and are described more fully in the next section.}
* record {This macro records the point. It expands into "free; $a:\leftarrow$ |pointer;." The user moves the arm to make the pointer touch the current frame origin and again pushes the "stop arm" button. The pointer location is then read and remembered.}
* record {This time the user records any other point on the z-axis of the current frame.}
* record {This time the user records any other point on zx-plane of the current frame. There would be other commands to select a different axis and plane.}
* construct {This computes a trans from the three recorded pointings.}
* |bracket:$\leftarrow a$: {This gives BRACKET the computed trans.}

Protocol to Move in a Frame

{The user may prefer to have the pointer moved by computer control rather than moving it himself manually. For example, suppose he is defining the frame of BRACKET_HOLE when he has already defined that of BRACKET.}

* free {He manually brings the pointer near the correct place, and pushes the "stop arm" button to proceed.}
* $r:\leftarrow$bracket {This says that the user wants to make motions with respect to the BRACKET frame.}
* $m:\leftarrow$pointer {This says that the frame which is to do the moving is the pointer. This is the main reason for including POINTER in the subpart hierarchy.}
* $dz - 0.01$ {This says move the pointer by a distance of 0.01 units in the minus z direction of the BRACKET frame. This macro expands into "$b:\leftarrow v$ 0 0 −0.01; d move; ↑." Again the commands will be explained later.}
* x 0 {This says move the pointer along the x axis of the BRACKET frame until it has $x = 0$ in that frame. Again, x is a macro.}

Extensions

The proposed system might be extended in several possible ways. For example, pointing by vision could be

```
                         Affixment Editor

M:P: WORLD at T 0 0 0 0 0 0                          N: HANDLE
        *FIDUCIAL at T 0 0 0 41.3 4.8 0                 BRACKET
R:      +ARM at T 1.2 92.3 1.8 11.4 18.2 10.5           BOLT
           +POINTER at T 0 0 0 1.0 1.4 6.2              WORLD
        +BEAM at T 0 0 0 20.0 40.0 0
           *BORE at T 0 90.0 0 0 1.5 6.0             D: BRACKET
D:      +BRACKET at T 0 0 0 10.0 60.0 0                 BEAM
N:         *HANDLE at T 0 0 0 0 0 0                     WORLD
           *BORE at T 0 180.0 0 5.1 2.0 0
        +BOLT at T 0 180.0 0 30.0 50.0 5.0
           -GRASP at T 0 0 0 0 0 0

                         Arithmetic Section

A: STACK
     1: 3.14159
     2: V 1.0 0 1.5
     3: T 90.0 10.0 18.5 1.9 2.3 1.3

B: STACK
     1: V 0.1 0 0
     2: <empty>
     3: <empty>

                      Terminal Interface Section

*new handle
*d:←bracket
*rigid
*
```

Fig. 7. Typical display scene.

provided to augment the manipulator pointing capability. Provision could be made for the user to define other attributes of object models besides location and affixment. Perhaps generic shapes such as cuboids, cylinders, or machine screws could be specified in this fashion. If visualizing frames proved to be difficult for some users, some form of graphics capability could be added to display frames of the objects in the subpart hierarchy, or possibly even to display some of the generic shapes. A more speculative extension would be a means for keeping track of a succession of world models, since such a sequence could be interpreted as an entirely new way of specifying an assembly procedure.

Software Design Details

System Architecture

The pointing system contains three principal working modules: the affixment editor, arithmetic routines, and manipulator interface routines.

The affixment editor contains facilities for creating and modifying the subpart hierarchy. The principal data structures associated with this section are the tree used to represent the hierarchy and a set of stack-structured context pointers called "cursors."

The arithmetic section contains a full set of arithmetic operations for scalars, vectors, and transes, together with routines for modifying the location attributes of parts. The principal data structures associated with this section are a pair of stacks used to hold operands. Typically, one stack is used to hold values specifying incremental motions for the manipulator, while the other is used for computing new location values.

The manipulator interface contains facilities for moving the manipulator under either system or user control and for retrieving the current location of the manipulator for use by the rest of the system. For the latter purpose, the node ARM in the subpart hierarchy always contains the current location of the manipulator.

In addition, there are three service modules: a command interpreter, display facilities, and output facilities.

The command interpreter accepts commands typed in from the terminal and translates them into calls on appropriate routines. Most commands have mnemonic text names (e.g., "getdad" to move a cursor from a subpart node to its parent). In addition, many have single character abbreviations (e.g., " ∧ " for "getdad"). A macro facility for stringing together commonly occurring sequences of primitive operations is assumed to be present but will not be discussed in detail in this document, although definitions for some assumed built-in macros will be given.

The display routines update the screen of the user's terminal to reflect the current state of the affixment editor, arithmetic section, and manipulator interface.

The input/output facility contains routines for saving and restoring subpart hierarchies in files. In addition, it contains routines for translating a subpart tree into a text file of AL declarations.

Subsequent subparts of this section will describe the display routines and the primitive user commands for invoking the facilities provided by the various modules.

Display Facilities

The display routines are used to provide the user with an up-to-date picture of the current system state. To do this, the display screen is broken into an affixment editor region, an arithmetic region, and a terminal interface region, as is shown in Fig. 7.

In the affixment editor section, subpart relations are indicated by paragraphing. The type of affixment is indicated by preceding the part name by a "*" for rigid affixment, " + " for nonrigid affixment, and " − " for independent affixment. Cursor positions are indicated by placing the cursor name on the appropriate line. Finally, the location attributes for each node are indicated in the "at" expression. In the case of rigid or nonrigid affixment, relative position with respect to the parent is printed. For independent affixment, location with respect to the WORLD node is shown. In addition, the affixment editor section displays the contents of the two most recently referenced cursor stacks.

The arithmetic section displays the current contents of the top three locations in the stacks. Values are shown in the same format in which they would be read in from the terminal, i.e., scalars are merely typed out, vectors have V followed by three scalars, and transes have T followed by three rotation angles and three displacement values.

The terminal interface section is simply an area for the operating system's page printer to go, and shows the last few commands typed by the user. As characters are typed by the user, this is where they are put.

For efficiency, random access graphics could be used to update only those parts of the display that are changed from command to command. However, it should be pointed out that the design shown uses only text characters for typeout.

Thus, on reasonably fast terminals a possible alternative would be simply to type out the whole screen each time.

Affixment Editor

The affixment editor contains facilities for creating and modifying the subpart hierarchy, which is stored internally as a tree of part nodes. Each such node contains the following information.

PNAME—The print name of the node. For instance, "BORE." Notice that these names are not necessarily unique.

DAD—Pointer to the parent node. The special node WORLD acts as an ultimate ancestor.

SON—Pointer to the node most recently affixed to this one.

EBRO and YBRO—Pointers to the elder and younger brother nodes, respectively. For example, EBRO[N] points to the node that was SON[DAD[N]] when N was affixed to DAD[N].

HOWAFFIXED—An integer telling how the node is affixed to its parent. The three kinds of affixment are *rigid* (1), *nonrigid* (2), and *independent* (0). Rigid and nonrigid affixment have essentially the same meaning as given in AL. When a DAD and SON are rigidly affixed, the motion of either one also causes the other to move. Under nonrigid affixment, the DAD moves the SON, but the SON does not move the DAD. Independent is a special kind of affixment used for nodes that have a subpart relationship with their parents but whose locations are independently specified. Independent affixment is primarily useful as an intermediate step in editing a structure.

XF—A homogeneous transformation used to specify the location of the node. This is internally stored as a 4 by 4 matrix. If the affixment is rigid or nonrigid, XF specifies the location of the node with respect to its parent node. If the affixment is independent, XF specifies the location with respect to WORLD.

Primitive commands are provided for creating new nodes, deleting old ones, modifying the affixment links, copying structures, and various other useful editing functions. These commands will be discussed in a moment; further details have been published elsewhere [18].

A number of context pointers, called cursors, are used to specify some of the arguments to the editing primitives. Such cursors are stack structured and typically are named by a single character followed by a colon. The following cursors are included in the current design.

CURNODE (N:)—gives the node on which the user is currently working.

CURDAD (D:)—gives node to which subparts are to be affixed.

CURPATH (P:)—gives root node of current subtree for name recognition.

CURREF (R:)—gives the current motion reference frame. All motion commands are to be made with respect to the location frame of this node.

CURMOVE (M:)—gives the current motion frame. All motion commands actually move the location frame of this node.

Operation	*Typein*	*Description*
Set cursor	**c:**←<node spec>	Pushes down the old value of cursor **C:**, then sets the new value <node spec>, which may be a node name or a cursor name. For instance, "**n:**←beam" followed by "**d:**←**n:**" would set the current value of both **N:** and **D:** to BEAM.
Pop cursor	**c:**↑	Restores the previous value of **C:**. May be combined with assignment, as in "**d:**←**n:**↑", which pops **N:** into **D:**, while pushing down the previous value of **D:**.
Exchange cursor	**c:**↔	Exchanges the current and previous values of **C:**.
Roll up cursor	**c:**∪	Pops cursor **C:** and places the popped-off value at the *bottom* of the stack for **C:**.
Roll down cursor	**c:**∩	Removes the *bottom* value from the stack for **C:** and pushes it onto **C:**.
Get father	**c:**∧	Replaces **C:** with DAD[**C:**]. Does not save previous value.
Get son	**c:**∨	Replaces **C:** with SON[**C:**]. Does not save previous value.
Get brothers	**c:**< and **c:**>	Replace **C:** with YBRO[**C:**] and EBRO[**C:**], respectively. Do not save previous values.

Fig. 8. Cursor operations.

CURTOP (T:)—gives the root node of the displayed subtree.

CURKILL (K:)—gives the root node of the most recently deleted subtree.

As mentioned before, cursors are stack structured. This means that typically whenever a new value is assigned to a cursor, the old value is saved away (to a depth of four) where it can be recalled if need be. This facility provides a nice way for a user to recover from errors, as well as to suspend temporarily one editing sequence and go do something else.

The operations shown in Fig. 8 are supplied for modifying the contents of cursors. Here C: has been used to stand for any cursor name.

An additional property of the cursor commands is that they remember the last cursor operated on. If a cursor designation is left off of a command, then the last cursor specified will be used. For instance, "*n*: ∧ ∧ >" will move the cursor *N*: so that it points at its great uncle.

Earlier, it was stated that node names need not be unique. Ambiguities may be resolved by naming an ancestor node whose subtree contains only one node with the name in question. In Fig. 7, for example, to distinguish the two BORE nodes, the user could refer to them as BEAM . BORE and BRACKET . BORE. Alternatively, if the user wishes to work for an extended period in one subtree, he can position the cursor P: to supply a default ancestor node. Thus, the sequence

```
p:←bracket
n:←bore
<
```

would place the cursor *N*: as shown in Fig. 7.

It is still possible to have inherently ambiguous namings, and such situations arise quite naturally as an intermediate state in cases where a previously defined structure is being copied and modified. Therefore, ambiguously named nodes are not outlawed. If a user wished to point at such nodes, he

Operation	*Typein*	*Description*
Make new node	new <name>	Creates a new node with the specified name and affixes it as an independent subnode of WORLD. Sets **N:** to point at the new node. The old value of **N:** is pushed.
Affixment	<affixtype>	Affixes the node pointed to by **N:** to that pointed to by **D:** in the indicated manner. Updates links and location values appropriately. (Discussed further in text)
Kill structure	kill <node spec>	Deletes the subtree rooted at the named node. Sets **K:** to point at the deleted structure. The previous value of **K:** is pushed. If <node spec> is omitted, then **n:** is assumed.
Unkill	**k:↑**	Popping a node off the **K:** stack restores the subtree rooted there to life. The restored structure is linked back onto its parent in the same way as during its previous existence.
Copy structure	copy <node spec>	Copies the subtree rooted at <node spec> and sets **N:** to point at the copy. The previous value of **N:** is pushed. If <node spec> is omitted, then **n:** is assumed.
Merge structure	merge	Unlinks all subparts from the node pointed to by **N:** and makes the corresponding affixments to the node pointed to by **D:**. Does not change **N:** and **D:**

Fig. 9. Editing operations.

Operation	*Typein*	*Description*
Fetch value	s:←<value>	Pushes <value> onto the named stack. <value> may be a literal (described in text), a register name (as in "**a:←b:**" or "**a:←o:↑**") or an absolute or relative node location.
Relative location	s:←@<node spec>	Pushes the XF attribute of the specified node onto the specified stack.
Absolute location	s:← | <node spec>	Pushes the location of the specified node with respect to WORLD.
Pop	s:↑	Pops the previous contents of **S:** into **S:**.
Edit	<lhs>	Places the string "<lhs>←<value>" into the system line editor. (Described in text)
Exchange	s:↔	Exchange the top two elements of **S:**
Store relative	@<node>←<value>	Copy the specified value into the XF attribute of the specified node.
Store absolute	| <node>←<value>	Modify the affixment structure so that the absolute location of the specidied node has the specified value.

Fig. 10. Elementary stack operations.

must get to a nearby node and then use cursor moving operations to get the rest of the way.

The primitives shown in Fig. 9 are d to edit the tree structure. The affixment operation is of three kinds: rigid, nonrigid, and independent. In the first two cases, the XF attribute of the node must be set to the current relative position of the node with respect to its new parent, while in the last case, the XF must be set to the current absolute location of the node, i.e., its location relative to WORLD.

One side effect of the way the kill command is defined is that storage for a deleted structure will not be reclaimed until the structure falls out the bottom of the K: stack (i.e., after four deletions). If this should ever get to be a problem, then a "purge the kill stack" operation is easy to add. (In fact, repeating "k:←k:" three times will flush all but the most recently killed object).

The copy command is useful for replicating subpart hierarchies in order to describe mating parts. For example, if a box, gasket, and lid all have holes in corresponding locations, the structure for the box and its holes can simply be copied twice. The merge command provides a means to move such structures around and to perform editing operations on sets of nodes.

The arithmetic section provides the user with a means for performing computations on scalars, vectors, and transes. All operations are performed in Polish on one of two operand stacks, and the results are stored back in the stack on which the operation was performed. In addition, facilities are provided for the user to enter values from the terminal and to retrieve and modify the location attributes of nodes in the affixment structure. Since the current value of ARM and POINTER always give the positions of the manipulator and the pointer, respectively, this means that the user can use the results of pointing operations to define relative or absolute part locations.

The arithmetic section contains two functionally identical 100-element operand stacks, called A: and B:. A third and somewhat shorter stack, called O: (for "oops"), is provided to facilitate error recovery. Any time a value is popped off one of the operand stacks, it is pushed onto O:. Thus, if a careless user makes a mistake and invokes the wrong operator, he can get his operands back by popping them off O:. As with all stacks in the system, overpushing causes the oldest element to fall out the bottom.

The elementary stack operations shown in Fig. 10 are provided for manipulating values. Here, S: stands for any arithmetic stack. The format used for type-in of literal values is the same as that used by the display routines to type them out.

The edit command is intended to facilitate modification of values that may be a little off, due to inaccuracies in pointing. Essentially, the line editor is a facility maintained by the time sharing system that allows a user to perform local editing operations on text that has been typed ahead but not yet activated [19]. (Some "smart" terminals offer similar local editing of text input.) One principal advantage of such a facility is that it provides users with an efficient, flexible, and uniform set of editing conventions. For instance, suppose that the current location of HANDLE with respect to BRACKET is "T 0 179.3 1.8 0 0 0.01." One "knows" that this just cannot be quite right, since rotation angles are assumed, for the moment, to come out to even fives of degrees. Also, one strongly suspects that the final displacement value ought to be 0. The value can be patched up by the following command sequence:

edit @ handle	{This loads the line editor with the text: "@ handle:← t 0 179.3 1.8 0 0 0.01" and puts the user in local editing mode.}
@ handle←t 0 180 0 0 0 0	{The user has edited the line to the form shown here and activated by typing carriage return.}

In cases where the thing being edited is a stack register, the

Operation	*Typein*	*Description*
Binary operation	s:+ - * /	Computes **S:**[1] <op> **S:**[0] and replaces the elements with the result. If both elements are scalars, then <op> has the "usual" meaning. If one is a scalar and the other is a vector, performs <op> element-wise. If both are vectors, then "+" and "-" give vector sum and difference, and "*" gives vector inner product. If both are transes, then "*" gives product. If **S:**[0] is a vector and **S:**[1] is a trans, then "*" gives T*v. Otherwise, <op> is undefined.
Cross product	s:cross	Computes **S:**[0] × **S:**[1].
Unary minus	s:neg	Computes 0-**S:**[0], providing it is defined.
Inverse	s:inverse	Computes **S:**$[0]^{-1}$, provided **S:**[0] is a trans.
Magnitude	s:magn	Computes the square root of **S:**[0]***S:**[0].
Displacement	s:dpart	Requires that **S:**[0] be a trans. Computes a vector equal to the displacement. E.g., for **S:**[0]= T 0 180 30 1 3 5, computes V 1 3 5.
Rotation	s:rpart	Requires **S:**[0] to be a trans. Produces a trans with zero displacement part but the same rotation as **S:**[0] E.g., for **S:**[0] = T 0 180 0 1 3 5, computes T 0 180 0 0 0 0.
Vector trans	s:vtrans	Converts **S:**[0] into a trans with zero rotation and **S:**[0] as the displacement. E.g., for **S:**[0] = V 1 3 5, computes T 0 0 0 1 3 5.
Construct trans	s:construct	Constructs a trans T from vectors **S:**[0], **S:**[1], and **S:**[2], such that T^{-1}***S:**[2]= (V 0 0 0), T^{-1}***S:**[1]= (V 0 0 z) for some z>0, and T^{-0}***S:**[1]= (V x 0 z) for some x,z with x>0.

Fig. 11. Arithmetic operations.

Operation	*Typein*	*Description*
Absolute move	s:amove	Requires that **S:**[0] be a trans. Moves the manipulator so that (absolute position of **M:**) = (absolute position of **R:**)***S:**[0]
Diff. motion	s:dmove	Moves the manipulator so that the absolute location of **M:** is changed by (absolute position of **R:**)***S:**[0].
Free arm	free	Releases brakes on manipulator joints, which can then be positioned manually. Completion of positioning is signalled by typing any activation character at the terminal or by means of a button on the manipulator.
Joystick	joy	Places the manipulator under control of a joystick.

Fig. 12. Manipulator operations.

Operation	*Typein*	*Description*
Save structure	save <file name>	Saves the structure pointed to by **N:** the file specified by <file name>. Has no effect on the affixment editor data structures.
Restore structure	load <file name>	Reads the structure stored the named file into the affixment editor's menory. Causes **N:** to point at the newly read structure, which is affixed as an independent subpart of WORLD. The previous value of **N:** is pushed.
AL code emission	al <file name>	Translates the structure pointed to by **N:** into a corresponding set of AL declarations and writes the text into the specified file. Has no effect on the affixment editor data structures.

Fig. 13. Input/output commands.

stack value is *popped* into the line editor without affecting the O: stack.

Except for nodes whose immediate parent is WORLD or which are independently affixed, absolute locations are not directly available in the affixment editor's data structure. However, the required computation is very straightforward.

A somewhat similar problem arises when storing away the absolute location of a node, since nonindependent nodes keep the relative location with respect to their parent node. Here, the effect desired in asserting that a node N has a new absolute location X is implicitly to update the absolute location of all nonindependent descendents of N. Also, if N is rigidly affixed to its parent, then the absolute location of the latter must also be updated.

The arithmetic operations which are provided are shown in Fig. 11. Here, *S*: is used to stand for either A: or B:, S:[0] refers to the top element of stack S:, S:[1] refers to the next element down, and so forth.

All these operations pop their operands off the stack, perform the computation, and then push the result. As with cursors, the name of the last arithmetic stack referred to is remembered and will be used as a default in subsequent stack operations.

Manipulator Interface

The manipulator interface provides the user with facilities for making controlled motions of the manipulator. The principal data structures employed are the nodes ARM and POINTER provided in the affixment data structure, and the cursors M: and R:, which are used to request computer-controlled motions of the manipulator. The primitive commands involved are given in Fig. 12.

As was mentioned earlier, the current location of the manipulator is assumed to be always present in ARM. In actual practice, it is rather inconvenient to do this while the manipulator is being moved. Therefore, it is assumed that the value is only updated at times when the system is at a convenient place, such as at the top of its command interpretation loop. Such a policy is unlikely to cause any difficulties for the user.

It is unlikely that the joy command would be implemented at Stanford, where manual positioning is used instead. The command is included to give some indication of where a joystick would fit into such a system.

Input/Output Facility

This module contains routines for saving subpart trees onto a file and for reading saved structures back into the affixment editor. This allows the user to spread his work over several terminal sessions without having to recreate the entire structure each time. In addition, a routine is provided to translate a subpart tree into AL declarations, using the hierarchy to produce unambiguous names. The input/output commands are given in Fig. 13.

Implementation Strategy

The proposed pointing system could be implemented in a minicomputer with at most 16k words of memory. However, at the Stanford University Artificial Intelligence Laboratory it is substantially easier to implement, modify, and use a pointing system on the large time-sharing system. The only

portion of the design which inherently assumes that this system would be used is the line-edit feature in the arithmetic section.

A preliminary version of the system has been implemented in SAIL [20] using record structures for the subpart hierarchy, and using available display primitives. The preliminary version, called POINTY, uses a symbolic debugger, BAIL [21], for command scanning. With BAIL, instead of the command syntax described earlier, users type SAIL procedure calls which are then interpreted.

The authors do not wish to be accused of succumbing to the *Mikado* syndrome which consists of saying that since the implementation is as good as done, for all practical purposes it is done, and if it's done, why not say so? [22]. On the other hand, it should be stressed that whether or not the full system proposed here is actually ever implemented, tests with the preliminary version demonstrate the feasibility of generating object models in a reasonably simple manner. What otherwise might have been a serious drawback to the use of high-level languages for manipulation has been shown, therefore, not to be a serious problem at all.

Conclusion

This paper has addressed the topic of generating object models for programs in a high-level manipulator language. An interactive system was proposed in which the manipulator itself is used as a measuring tool in three dimensions. One component in this system is a bendy pointer whose deformation may be calibrated against a known fiducial mark. Hypothetical protocols involving such a system were presented, as well as details about the design of the underlying software.

Such a system would materially speed up the process of generating world models for AL programs. Most of the proposed system could be adapted for other methods of pointing or for other high-level manipulation languages. The system could be expanded to permit additional descriptive data about objects, or to keep track of a sequence of world models as a new means of specifying an assembly procedure.

A preliminary version of the system has been implemented and tested. This preliminary system demonstrates that specifying object models can be a much easier process than might otherwise have been believed.

Appendix: A Subset of AL

The summary of AL which appears in this appendix is a highly condensed version of a portion of the material contained in [2]. As such, it represents the research of the authors of [2] plus B. Shimano. This material is included in this paper for completeness only, since the cited publication is somewhat out of date and since the material is otherwise not available in any technical journal.

AL has a basic block structure which looks like this:

```
label: BEGIN
       body of block;
       END label;
```

The body of the block may consist of other blocks, AL commands, invocations of library routines or macros, and calls to procedures. Comments may appear anywhere, delimited by curly brackets. The steps in a block of this form are executed in the order of their appearance. If time ordering is not important, and concurrency is permissible, there is a different block structure:

```
label: COBEGIN
       body of block;
       COEND label;
```

Users may effectively expand the set of AL commands by coding routines in the following manner:

```
ROUTINE routine name(parameter_list)
  body of routine;
```

Such routines may be saved on disk or retrieved from disk by the respective commands

```
SAVE new_old_flag routine_name_list ON file_id;
RETRIEVE new_old_flag routine_name_list
  FROM file_id
```

Macros are defined in the following way:

```
DEFINE macro name(parameter list) = "body of macro";
```

Procedures are defined as follows:

```
optional type PROCEDURE proc_name
  (parameter_list)
  body of procedure;
```

Variables are declared in the following manner

```
dimension_type data_type name_list;
```

The allowed data types are EVENT, SCALAR, VECTOR, ROT, FRAME, PLANE, and TRANS. Events may not have a dimension type, while it is optional for the other data types. A FRAME is a position and orientation in space, while a TRANS is a transformation taking one FRAME into another. Built in dimensions are TIME, MASS, and ANGLE, but this list may be expanded by the use of the statement

```
DIMENSION new_dimension_type
   = old_dimension_type_expression;
```

Whenever a variable is used, its unit of measure may be specified by coding

```
variable_name * unit_name.
```

Built in unit names are CM, GM, SEC, and DEG, but this list may be expanded by the use a macro of the form

```
DEFINE new_unit_name
   = "(old_unit_name_expression)";
```

In addition to the usual scalar arithmetic operators (+ − */), there are operations relating to other data types, as summarized in Fig. 14(a). There is also a set of keyword operations as listed in Fig. 14(b). The keywords NILVEC

Operation	*Description*
vector . vector	dot product of two vectors
plane . vector	distance from vector to plane
vector . plane	distance from vector to plane
scalar * vector	dilation of a vector
vector / scalar	contraction of a vector
vector + vector	sum of two vectors
vector - vector	difference of two vectors
rot * vector	rotation of a vector
trans * vector	transformation of a vector
rot * rot	right first composition of two rotations
frame + vector	translation of a frame
frame * rot	rotation of a frame
frame * trans	transformation of a frame
plane + vector	translation of a plane
frame → frame	trans that maps one frame into another
trans*trans	right first composition of transformations

(a)

Operation	*Description*
VECTOR(scalar1,scalar2,scalar3)	construct vector
ROT(vector_axis,scalar_angle)	construct rotation
FRAME(rotation,vector_translation)	construct frame
TRANS(rotation,vector_translation)	construct trans
ABS(vector)	length of vector
ORIENT(frame)	rotation part of frame
LOC(frame)	translation part of frame
NORMAL(plane)	normal vector to plane
vector WRT frame	same as ORIENT(frame)*vector

(b)

Fig. 14. (a) Basic AL operations. (b) AL keyword operations.

and NILROT stand for a null vector and a null rotation.

The two types of loop provided are

FOR scalar_var ← scalar_expr STEP scalar_expr
  UNTIL scalar_expr DO statement
WHILE condition DO statement.

The condition is a boolean expression involving the operators $<$, $>$, $\leq$, $\geq$, $=$, and $\neq$, the logical connectives $\wedge$, $\vee$, and $\neg$, and the logical constants TRUE and FALSE. The conditional statement has the form

IF condition THEN statement ELSE statement

where statement may be a block. There is also a conditional expression

IF condition THEN expression ELSE expression

which may be used wherever an expression would otherwise be used.

The basic motion command has the form

MOVE object_frame TO goal_frame
  via_clauses
  on_condition_clauses
  with_requirement_clauses.

The particular object frames, YELLOW and BLUE, are the frames of the yellow and blue grippers of the yellow and blue manipulators. The particular goal frame ⊗ (pronounced grinch) means the "current position." Within the scope of a move, the STOP command terminates the motion immediately, while the ABORT command at any point in the program terminates execution immediately.

Via clauses are used to specify intermediate frames along the trajectory as well as further requirements to be imposed or actions to be taken at these points. Trajectories are generated which pass smoothly through the intermediate points. There are five types of via clauses:

TRACING parameterized_frame
  FOR parameter_loop_specification
VIA intermediate_frame_list
VIA intermediate_frame where_requirement_clause
VIA intermediate_frame THEN statement
DIRECTLY.

The first type essentially specifies an entire intermediate trajectory, while the second type is used to specify a discrete list of intermediate frames. Instead of a single list, it is possible to specify a sequence of intermediate frames by a succession of commands of types 3 and 4. The syntax of the where requirement clauses is discussed below. Even if via clauses of types 1 through 4 are absent, two default intermediate frames called DEPARTURE and APPROACH are assumed, unless DIRECTLY is specified.

The on condition clauses have the form

label: defer_control ON on_condition DO statement.

The label is optional. Defer control may be missing, in which case the on condition is initially enabled, or it may be DEFER, in which case it is initially disabled. The statement portion of the clause may be a block, inside of which the commands

ENABLE label
DISABLE label

may be used to enable or disable other on condition clauses pertaining to the same motion. The ENABLE command without any label reenables the current on condition clause, which is otherwise automatically disabled the first time it triggers.

Inside the on condition several keywords may appear: DURATION is the elapsed time in seconds since the start of the motion. ARRIVAL is 1 or 0 depending on whether or not the goal frame has been reached. FORCE(direction_vector) is the scalar force along the specified direction. TORQUE(axis vector) is the scalar torque about the specified axis. SQUEEZE is the scalar squeezing force of the gripper. OPENING is the scalar distance between the fingers. Inside on conditions these keywords may not appear in expressions, either boolean or arithmetic. FORCE, TORQUE, SQUEEZE, and OPENING are defined globally, and they may appear in expressions outside of on conditions.

Inside the statement portion of the on condition clause, the commands

CRITICAL
UNCRITICAL

may delimit the beginning of code which is time critical or not. The default is uncritical.

The with and where requirement clauses have the forms

WITH requirement
WHERE requirement.

The difference between the two is the scope of the requirement. WITH pertains to an entire motion, while WHERE

pertains to a particular via point along a motion. The same set of keywords may be used in the requirement as in the on conditions discussed above.

A special form of the move command is used for straight line constant velocity motion:

```
MOVE object_frame
WITH VELOCITY = vector
  THROUGH goal_frame
  FOR DISTANCE = scalar_distance
  on_condition_clauses.
```

AL provides several means for synchronizing processes. Synchronization of statements in a cobegin block may be done with the statements

```
SIGNAL event_variable
WAIT event_variable.
```

It is possible to impose time order on labeled statements by the command

```
PREREQUISITE OF label_1 IS label_2.
```

Weak synchronization of two arms may be achieved by using

```
[left_argument: right_argument]
```

throughout a motion command, wherever a single argument would otherwise have been used. In this way, goals and via points may be coordinated. Strong synchrony which coordinates the entire trajectories of two arms is accomplished by a new clause

```
COORDINATING frame_equation
```

where the equation specifies the way in which the frames of the two arms are to be related.

There are two special motion commands for searching and centering. A search is a means of specifying repeated action in a spiral. The syntax is

```
SEARCH object_frame
  ACROSS plane
  WITH INCREMENT = scalar_distance
  REPEATING statement
  on_condition clauses
```

The repeated statement may be a block, which in turn may contain motion commands. The command

```
CENTER object_frame
  on_condition_clauses
```

causes the gripper to close on an object while the arm accommodates to its location.

Devices like the electric screwdriver have device names. These devices may be operated by the command

```
OPERATE device_name
  on_condition_clauses
  with_requirement_clauses.
```

Aside from the CENTER command, the grippers are considered to be devices called YFINGERS and BFINGERS. In the on conditions for the grippers, the keyword OPENING may be used to continually refer to the distance between the fingers.

At any point in the assembly sequence, the program may specify that two frames are affixed. The relevant command is

```
AFFIX part_frame TO object_frame
  BY relative_trans
  rigidity_specification.
```

The relative transformation is object_frame → part_frame. If the by clause is present, this transformation is given the designated name. Otherwise, the system invents a temporary trans variable for it. If the rigidity specification is RIGIDLY, then subsequent motion of either the part frame or the object frame will cause the other one to be updated. If the rigidity specification is absent, the motion of the object frame carries the part frame along, but the reverse is not true. Affixment is undone by the command

```
UNFIX part_frame FROM object_frame.
```

It should be noted that neither AFFIX nor UNFIX involve manipulator actions; they simply modify affixment graph structures.

Associated with the frame of each object is a deproach frame, through which the manipulator moves during motion approaching or departing from the specified object. Deproach frames may be set by the command

```
ASSERT FORM
  (DEPROACH, frame_name, relative_trans).
```

If a deproach has not been specified, the system searches through the affixment graph to determine a reasonable deproach. The set of READ and WRITE commands has not yet been fully specified.

Acknowledgment

The research reported here was performed as a part of the Computer Integrated Assembly project headed by T. Binford. It was therefore heavily dependent on the prior work of V. Scheinman, R. Paul, R. Finkel, and B. Shimano.

One of the authors (Grossman) also wishes to acknowledge enlightened management at IBM and at the Stanford Artificial Intelligence Laboratory for making possible his sabbatical year at SAIL. Also, the other author (Taylor) would like to express his appreciation for fellowship support from the Alcoa Foundation during the period in which this work was performed.

References

[1] R. Bolles and R. Paul, "The use of sensory feedback in a programmable assembly system," Stanford Artificial Intelligence Lab. Memo AIM-220 and Stanford Univ. Computer Science Report STAN-CS-396, Oct. 1973.

[2] R. Finkel, R. Taylor, R. Bolles, R. Paul, and J. Feldman, "AL, A programming system for automation," Stanford Artificial Intelligence Lab. Memo AIM-243, and Stanford Univ. Computer Science Report STAN-CS-74-456, Nov. 1974.

[3] N. J. Nilsson, Ed., "Artificial intelligence—Research and applications," Stanford Research Institute Project 3805 Progress ep., p. 16, May 1975.

[4] J. Nevins *et al.*, "Exploratory research in industrial modular as-

sembly," 3rd Progress Rep., No. R-921, Charles Stark Draper Lab., Inc., Cambridge, MA, Aug. 1975.
[5] D. J. Barber, "Mantran: A symbolic language for supervisory control of an intelligent remote manipulator," MIT Engineering Projects Lab. Rep. 70283-3, June 1967.
[6] P. M. Will, "Computer controlled mechanical assembly," in *Proc. 5th Int. Symp. Industrial Robots*, IIT Research Inst., Sept. 1975.
[7] P. M. Will and D. D. Grossman, "An experimental system for computer controlled mechanical assembly," *IEEE Trans. Comput.*, vol. C-24, no. 9, Sept. 1975.
[8] L. I. Lieberman and M. A. Wesley, "Autopass, An automatic programming system for computer controlled mechanical assembly," *IBM J. Research and Development*, vol. 21, no. 4, July 1977.
[9] B. G. Baumgart, "Geomed," Stanford Artificial Intelligence Lab. Memo AIM-232 and Stanford Univ. Computer Science Rep. STAN-CS-414, May 1974.
[10] D. D. Grossman, "Procedural representation of three-dimensional objects," *IBM J. Research and Development*, vol. 20, no. 6, Nov. 1976.
[11] I. C. Braid, *Designing With Volumes*. Cambridge, England: Cantab Press, 1974.
[12] ——, "The synthesis of solids bounded by many faces," *Commun. ACM*, vol. 18, no. 4, p. 209, Apr. 1975.
[13] "An introduction to PADL," Production Automation technical memorandum 22, Univ. Rochester, Dec. 1974.
[14] A. A. G. Requicha, N. M. Samuel, and H. B. Voelcker, "Part and assembly description languages—II," Production Automation technical memorandum TM-20a, Univ. Rochester, Nov. 1974.
[15] Robert P. Burton, "Real-time measurement of multiple three-dimensional positions," Univ. Utah, rep. UTEC-CSc-72-122, June 1973.
[16] D. A. Seres, R. Kelley, and J. R. Birk, "Visual robot instruction," in *Proc. 5th Int. Symp. Industrial Robots*, IIT Research Institute, Chicago, IL, Sept. 1975.
[17] D. D. Grossman and M. W. Blasgen, "Orienting mechanical parts with a computer controlled manipulator," *IEEE Trans. Syst., Man, Cybern.*, Sept. 1975.
[18] D. D. Grossman and R. H. Taylor, "Interactive generation of object models with a manipulator," Stanford Artificial Intelligence Lab. memo AIM-274, and Stanford Univ. Computer Science rep. STAN-CS-75-536, Dec. 1975.
[19] B. Harvey, "Monitor command manual," Stanford Artificial Intelligence Lab. operating note 54.3, Dec. 1973.
[20] K. A. VanLehn, Ed., "SAIL user manual," Stanford Artificial Intelligence Lab. memo AIM-204 and Stanford Univ. Computer Science rep. STAN-CS-73-373, July 1973, revised Oct. 1975.
[21] J. F. Reiser, "BAIL—A debugger for SAIL," Stanford Artificial Intelligence Lab. memo AIM-270 and Stanford Univ. Computer Science rep. STAN-CS-75-523, Oct. 1975.
[22] W. S. Gilbert and A. S. Sullivan, *The Mikado*, Act II, Mar. 1885.

Effectiveness of Basic Display Augmentation in Vehicular Control by Visual Field Cues

ARTHUR J. GRUNWALD AND S. J. MERHAV, MEMBER, IEEE

***Abstract*—The effectiveness of different basic display augmentation concepts (fixed reticle, velocity vector, and predicted future vehicle path) for remotely piloted vehicles (RPV) controlled by a vehicle mounted television camera is investigated. The task is lateral manual control of a low flying RPV along a straight reference line in the presence of random side gusts. The man-machine system and the visual interface are modeled as a linear time-invariant system. Minimization of a quadratic performance criterion is assumed to underly the control strategy of a well-trained human operator. The solution for the optimal feedback matrix enables the explicit computation of the variances of lateral deviation and directional error of the vehicle and of the control force that are used as performance measures. These variances are initially calculated with assumed values of human operator parameters such as weighting coefficients and noise levels. In particular, it is investigated whether and to what extent the human operator actually utilizes the display of information representing higher order state components such as lateral velocity and acceleration. The results show that the effectiveness of the display aids strongly depends on the vehicle dynamics and the spectrum of the disturbance. A velocity vector reticle is very effective for fast vehicle dynamics and rather ineffective for slow vehicle dynamics. On the other hand, a future vehicle path reticle is very effective for slow vehicle dynamics but less effective for fast vehicle dynamics and fast disturbances. The analytical results obtained are then validated by means of a specially developed flight simulator. The experimental values of variances in deviation, directional error, and control effort are matched with the results of the analytical model. This matching process yields the following results. 1) The analytical model proves to be a convincing representation of the actual man-machine system of visual field control. 2) Important parameters in the human operator model involved, can be determined with good accuracy. 3) The relative advantage of different fundamental display aids can be evaluated in relation with the disturbance statistics and vehicle dynamics.**

Manuscript received January 13, 1977; revised November 18, 1977, and May 8, 1978. This work was supported by the Department of Research and Development, Ministry of Defense, Israel. This paper is based on a dissertation submitted to the Department of Aeronautical Engineering, Technion—Israel Institute of Technology, Haifa, Israel in partial fulfillment of the requirements for the Ph.D. degree.

A. J. Grunwald is with the Department of Aeronautical Engineering, Technion—Israel Institute of Technology, Haifa, Israel, and NASA Langley Research Center, VA.
S. J. Merhav is with the Department of Aeronautical Engineering, Technion—Israel Institute of Technology, Haifa, Israel.

I. Introduction

THE CONTROL of remotely piloted vehicles (RPV) is a special case of visual field control (VFC). The visual field (VF) for RPV control is the monitor image of a vehicle

0018-9472/78/0900-0679$00.75 © 1978 IEEE