AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly

Abstract: This paper describes an experimental very high level programming system for computer controlled mechanical assembly, AUTOPASS (AUTOMASS (AUTOMASS

Introduction

Mechanical assembly is of major economic importance. Historically, all mechanical assembly operations were performed manually. As the requirements of assembly have grown, both in terms of complexity and volume, so has the desire to automate the process. At the present time, mechanical assembly may be divided into three classes on the basis of volume, measured in assemblies per year. At the very high volume end of the range, assembly is generally extensively automated, with specialized machines used for each step in an assembly operation and with automatic transfer of parts between machines. This mode of operation is characterized by high capital cost of the equipment, by long delays in design, delivery, and installation, and by difficulty in accommodating changes in the assembly operation. At the very low volume end of the range, assembly is carried out by hand and is characterized by high manpower costs and ease of change in the assembly operation, perhaps even to the extent that no two assemblies are the same. Between these extremes of volume is a region where human operators perform assembly operations either manually or by using hand tools, with jigs or fixtures used to hold pieces being worked on.

The automation of this intermediate region of the assembly volume spectrum is being addressed by the use of computer controlled manipulators. Such machines are capable of a number of motions that can move a gripping mechanism around the work space and use it to pick up parts and perform assembly operations on them. Machines of this class are already widely accepted in indus-

try [1]. However, a current topic for research and development is to increase the generality of application, in terms of the complexity of operations that can be performed reliably, by the use of sensory feedback from the assembly world. This feedback relies on tactile and force sensors in the gripping mechanism and the work area, and may extend to video and other remote sensing techniques.

Thus application areas for programmable general purpose mechanical assemblers are expected to lie in the region between very high volume dedicated assembly lines and custom assembled products. In this region it is believed that the flexibility of the general purpose assembler, in terms of its ability to switch rapidly between alternative programs, will offset its relative inefficiency in comparison to dedicated special purpose assembly machines. In our work in automation research, we have found that the major obstacle to the use of programmable general purpose mechanical assemblers is the difficulty and resultant high cost of programming a complex assembly. This kind of programming presents several basic problems: The three-dimensional world of assembly is spatially complex and difficult to visualize; the control of an intricate mechanical process is new to most programmers and requires an attention to detail unlike that found in the usual scientific or business computing. Another significant difference between traditional programming and mechanical assembly programming, which dictates a new approach, is that, whereas variables and program components in a digital computer are

- 9 1. ASM SUPPORT BRACKET
- 10 P/U AND POSITION THE NUT IN THE NEST OF THE FIXTURE.
- 11 1090037 NUT, CAR RET TAB QTY 01
- 12 P/U, ORIENT AND POSITION THE BRACKET INTO THE
- FIXTURE WITH ITS TAB OVER THE NUT.
- 13 1115191 BRKT ASM, RAIL SUPPORT QTY 01
- 14 P/U SCREW AND LOAD DRIVER.
- 15 1107379 STUD, CR TAB INTLK OTY 01
- 16 P/U, ORIENT AND POSITION THE INTERLOCK OVER THE BRACKET HOLE, WITH THE NOTCHED LUG UP.
- 17 1117637 INTERLOCK, CR + TAB QTY 01
- 18 P/U AIR DRIVER.
- 19 DRIVE SCREW TIGHT.
- 20 TORQUE 12.0 IN/LBS.
- 21 ASIDE AIR GUN.

Figure 1 Support bracket assembly description taken from an assembly sheet.

discrete and well-defined, the parts of an assembly and the assembler are imprecise. Assemblies and parts are subject to variations in size and position, assemblers have residual positional errors, and parts may slip in a gripper, so that sensory feedback must be used to carry out the assembly operations and to detect and correct errors.

Two main approaches to simplification of the programming of general purpose mechanical assemblers are in evidence. One is programming by showing, in which the machine is led through the assembly motions by the programmer, perhaps using a pushbutton control box, and the motions are stored for repetitive replay. Although this technique is very satisfactory for establishing the main motions of the assembly operation and can be executed by a trained assembly worker rather than a programmer, there are difficulties associated with the manner in which programs are to be edited and the manner in which sensory feedback and error recovery techniques are to be specified. In its basic form, that is, without extensive use of sensory feedback, this approach is already in use in industrial situations [2] and is being developed to handle a wider range of assembly specifications [3].

The other approach is through the use of textual programming, which may span a range from very high level automatic programming systems to low level machine languages. Consider the following example of an assembly task:

Screw the bracket and the interlock together. (1)

At this very high level of specification, the system has to recognize the items *bracket* and *interlock* and must interpret the *screw* action to be performed on them. The system must then generate a plan to achieve the desired goal. This problem could be approached by the use of

artificial intelligence (AI) problem solving systems [4–6], which would attempt to break the problem down into a series of subgoals to be achieved in sequence. This is a very complex problem, which we feel is not generally solvable by any existing AI techniques; however, other work is being performed that uses an AI approach to very high level languages for automatic assembly [7].

Consider now the description of the same example, which was taken from an existing industrial assembly instruction sheet and is shown in Fig. 1. In this case the problem has already been broken down into a sequence of subgoals, which are recognizable assembly steps. The assembly sequence has been specified together with the parts, tools, and fixtures to be used. However, explicit information is still missing, such as how the bracket is to be placed in the fixture, which hole in the bracket is to be used, and what is to be done with the screw. Since this form of high level language exists and assembly instruction sheets have already been written and optimized, it is clearly an attractive candidate for a formalized assembly language.

As an example of a lower level of assembly programming, Fig. 2 shows the code to implement statement 16 in Fig. 1, written in a hypothetical manipulator-level programming language. This code is manipulator directed; that is, it is concerned with specification of manipulator motions to achieve the desired assembly goal. The programmer now has to visualize the three-dimensional nature of the assembly operation, specify motions in terms of geometric variables, and analyze symbolic sensors. At an even lower level of manipulator control language, such as ML, which was described by Will and Grossman [8], the user programs directly in the manipulator's "motor space" and interprets absolute sensor data. The expansion of code between a manipulator-level language and ML is considerable, both in number of statements and complexity of numerical specification.

From the point of view of a user, the higher the level of language used, the easier the human programming process becomes. In the examples of four programming levels introduced above, the highest level (natural language input) is deemed infeasible to implement at present, and the lowest (ML) is clearly an extremely difficult level at which to program. The essential difference between the assembly instruction sheet of Fig. 1 and the manipulator-level language of Fig. 2 is that the former describes the assembly operations to be performed, whereas the latter describes manipulator motions to achieve an assembly operation. The user of a manipulator-level language must be prepared to program the necessary motions to accommodate the geometry of the assembly operations and the interpretation of sensor data. Even with the assistance of subroutines for performing common operations, the complexity of the coding task requires that he be an experienced programmer as well as one skilled in the art of manipulator control.

The assembly instruction sheet approach requires that the system understand the geometry of the assembly world. An assembly programming system working at this level must also include a means for specification of the geometry of parts, fixtures, manipulators, and other items of the assembly world. In both the case of specifying the geometry and of writing the assembly code, it is possible that the user be a designer or an assembly engineer rather than a programmer.

It is the purpose of this paper to demonstrate that a language close to the level of the assembly instruction sheet of Fig. 1 provides a natural interface for assembly or design engineers and that this level of language can be implemented. A design for such an experimental assembly-directed programming system, the AUTOmated Parts Assembly system (AUTOPASS), is presented.

Assembly-directed programming

A major goal of automatic programming systems is to permit the user to program in user domain terms [9, 10]. Assembly-directed programming enables him to specify assembly operations in a familiar manner, but to achieve the goal it is also necessary that he be able to reference assembly objects, such as parts, fixtures, and tools, in a familiar and natural manner. This latter requirement is pursued later.

The choice of the form of the language, as opposed to its level, may range from natural language to a conventional instruction set with operation code mnemonics. Natural language, because of its richness, ambiguity, and use of context to complete descriptions, is not a suitable form for stating assembly operations with adequate precision. A workable compromise is to use a formalized syntax with an English-like appearance. For example, verbs such as PLACE, TURN, and INSERT may be used for operation codes, with operand names such as bracket, interlock, and screw, and with qualifying phrases such as

with the tab.hole over the nut

In order to program in the language, the user must be familiar with its commands and syntax, but it is possible for one unfamiliar with the language to read and understand the code.

Choice of programming level

The choice of level for an assembly-directed language is a matter of great concern. As we have seen, too high a level leads to complex problem solving situations in which the system attempts to break a problem down into subproblems that it can solve. This approach can encounter difficulties when the achievement of one subgoal

```
/*This is an example of the kind of code which could be used to
    implement instruction 16 from the assembly instruction
    sheet: 'P/U, ORIENT AND POSITION THE INTERLOCK OVER
    THE BRACKET HOLE, WITH THE NOTCHED LUG UP
    The language used here is a hypothetical one, with features
    very much like those of PL/I and augmented by geometric
    variables, such as coordinate frames, geometric operators
    and by manipulator-specific operations.
    In the first section of this program the geometric variables
    are declared.
  {\tt DECLARE}\ (pallet, fixture, part position\, l, part position, grasp, goal,
            tab) COORDINATE FRAME.
            trajectory (6) 3-D POINT.
            (dx.dv) SCALAR:
/*In the following section of code the user defines the various
       Frames are defined in terms of other frames using the
      TRANSLATED and ROTATED operators.
    DEFINE goal AS fixture TRANSLATED (7.5,6.3,2.0);
    DEFINE partposition1 AS pallet TRANSLATED
      (0.6,21.9,3.0):
    DEFINE partposition AS partposition1 ROTATED (35,0,0);
    DEFINE grasp AS partposition TRANSLATED (0.5,19.0,-0.2);
    dy = 0.1;
/*The array trajectory consists of a set of point variables repre-
    senting points in space through which the origin of a frame
    (e.g., the manipulator gripper frame) will travel.
    The user has computed where the points are and is using
    the VECTOR constructor to initialize trajectory.
  trajectory(1) = VECTOR(28.5,30.6,15.1);
  trajectory(6) = VECTOR(...);
/*In this section of code the manipulator motions are defined.
  First open the gripper and re-orient the fingers
      OPEN FINGERS TO (.25):
      ROTATE GRIPPER TO (-90,0,0);
/*Re-define the manipulator coordinate frame to be 3 inches be-
    yond the tips of the fingers, then move it to the part grasping
    position and grip the part.
      DEFINE MANIPULATOR AS GRIPPER TRANSLATED
         (3.0.0.0):
      MOVE MANIPULATOR TO ORIGIN (grasp):
      CALL GRIPIT (3.5);
/*Move the hand (with the part) through the trajectory.
    This is a series of guarded moves using the logical sensor
    HIT which attains the value of TRUE if the force sensors
    feel contact during motion.
      i = 0:
      UNLESS HIT DO:
        IF i>6 THEN GOTO didihit;
        MOVE MANIPULATOR TO trajectory(i);
      END;
/*SUCCESS is a logical flag set to TRUE when the last requested
    motion has been successfully completed. Therefore, if
    SUCCESS is FALSE in this situation it means that a hit was
    sensed during the trajectory. The error recovery (not
    shown here) is specified beginning at label hitit.
      didihit: IF SUCCESS=FALSE THEN GOTO hitit;
/*Re-define the MANIPULATOR frame to be at the hole in the
    part. When the move is performed the frame of the hole will
    be aligned with the goal frame on the object held in the
    fixture.
      DEFINE MANIPULATOR AS MANIPULATOR TRANS-
      LATED (dx.dv.0):
      CALL SOFTMOVETO(ORIGIN(goal));
```

Figure 2 A manipulator-level program for support bracket example

makes achievement of a later one impossible. For example, if a part is to be fitted in one operation step, it may transpire that the fitting cannot take place because another part, placed in a separate step, is preventing the fitting operation from taking place. In general it is not possible to choose subgoals that are independent, and the system must iterate over the sequence of subgoals to achieve a valid overall solution.

The segmentation of an assembly problem such as (1) into subtasks (Fig. 1) is an operation with which an industrial assembly engineer is familiar. His normal role of planning for manual assembly involves solution of similar problems, for example, generation of insertion sequences for parts. If the language level is set too low, then the user may have to specify more steps, with correspondingly more attention to detail, than is convenient. For example, statement 16 in Fig. 1:

P/U, ORIENT AND POSITION THE INTERLOCK OVER THE BRACKET WITH THE NOTCHED LUG UP.

could be programmed as a sequence

GRASP interlock AT POINT x

MOVE interlock TO POINT y OVER fixture
LOWER interlock TO CONTACT bracket
RELEASE interlock

In this example the user specifies a grasping point x, an intermediate point y over the *fixture*, the method of lowering the *interlock* onto the *bracket*, and when to open the gripper. However, the specification of these extra steps does not generally simplify the task of generating manipulator motions. The sequence of operations is well understood and can be incorporated in the system; the choice of a grasping point and an approach point can in principle be derived from a geometric model (though it is possible that the compiler will fail to find a solution to a problem); and the lowering into position can be accomplished by selection of library routines.

An examination of low-level manipulator code that we have written shows that the code is generally a straight line sequence of manipulator motions. Conditional branching based on sensor analysis occurs within operations such as *lower to contact*, and wider ranging branching for error recovery, after, for example, dropping a part, occurs between routines at the level of the statements of Fig. 1.

The AUTOPASS user plans the overall assembly operation as a sequence of high level assembly statements, each involving operations on the level of positioning one part or inserting one screw. The AUTOPASS statements have been designed to enable the user to communicate the assembly process to the system in a natural and convenient manner. The statements are English-like in appearance with verbs such as PLACE, TURN, INSERT;

qualifiers such as UNTIL TORQUE IS 5 FT LB; and objects and features of objects with user defined names such as 10-32 screw and side-bracket-top-surface. However, the statements are not in a natural language but have a formalized syntax with precisely defined semantics for all permissible verbs and qualifiers. The problem domain has been sufficiently constrained that each assembly statement can invoke a prestored template that describes a sequence of utility routines to be used in the generation of the implementation plan. The calling of some of these routines may be optional, depending on the state of the assembly world, and the individual routines may require a process similar to problem solving in a restricted domain, as do the grasping and collision avoidance routines. The major utility functions, many of which are commonly used by the semantic modules of several statements, are discussed later. If the compiler is unable to find solutions to one or more of the basic steps specified by the template for the command, the user would be asked to re-specify the operator, or perhaps use lower level operations of the form given in (2).

Representation of the assembly world

(2)

Assembly directed statements require that the system understand the nature of the assembly world, in terms of both its geometric and physical properties. One possible approach would be to use the real world itself; however, inference of the state of the world from sensory input is beyond the present state of the art in terms of both the processing of sensory data and the deduction of assembly relationships among objects. Problems still remain when the real world is to be used to predict the effects of manipulator motions, such as in calculating a collision-free trajectory. Either the system must operate iteratively, sensing changes in the world after every small motion, or an internal representation derived from the real world may be used.

AUTOPASS represents the real world by a data base called the world model. The representation of both geometric information, such as the shape of an object and its location in the assembly world, and physical information, such as stability of objects and support relationships (e.g., will a part fall over if another part is removed) and attachment relationships between objects (e.g., if the interlock is removed from the fixture, will the bracket be removed also), is conceptually straightforward. However, major difficulties arise in the recognition of changes in the geometric and physical states of the system as a result of manipulator actions. Changes in the geometry are readily understood, but changes in physical relationships are much harder to recognize. For example, in the program given in Fig. 1, how does the system recognize that inserting a screw into a hole attaches all four objects together? The approach taken in AUTOPASS is to place on the user the responsibility for specifying physically realizable operations, for example, that parts are placed in stable positions, and for informing the system of changes in attachment relationships.

At the start of compilation of an AUTOPASS statement, the world model represents the current state of the assembly world. During the course of compilation of the statement the data base provides information for such operations as choice of the grasp point on an object and generation of manipulator trajectories. At the end of compilation of the statement, the world model is updated to reflect the changes introduced by the successful compilation of the statement.

The use of an internal representation, the world model, enables the compiler to simulate the expected run time world at compile time. Compilation may be performed using a suitably large processor system, and the object code generated may be run on a relatively small manipulator control system. In a production environment, the object code would be expected to be run repetitively for extended periods without modification.

The world model data base may have been generated in a declarative phase, before compilation is begun, using a geometric design processor [11]; in a fully integrated computer aided design and computer aided manufacturing system (CAD/CAM), much of the world model information would already be available in digital form. Symbolic naming of assembly objects and features of objects, required by the English-like form of AUTO-PASS, may be obtained by permitting suitable naming operations to be performed on the world model, either in an initial declaration phase or dynamically by declaration statements during execution.

User interface

We have already discussed some requirements for an interface suited to users with limited programming experience. A further area of considerable importance arises from the nature of assembly-directed programming. The language statements are goal-directed in that they specify the required result of an assembly step rather than the method by which the manipulator is to execute the step. In common with all goal-directed languages, the problem arises of assuring that the system's model of the problem matches that intended by the user. In the case of a conventional business or scientific programming problem, the consequences of an error in program behavior may be contained by execution on suitable trial data. In the case of a manipulator, the consequences could well be physical damage to the machine and the assembly work area, so that ensuring model correctness becomes of greater concern.

The AUTOPASS compiler interacts with the user at two levels. At the higher level, statement syntax is checked, operands are evaluated, and general feasibility checks are carried out. Feasibility checks might include: Can the specified object be moved; is there a hand available for the task (if not, can one be freed); are specified features on the expected object? At this level, specification changes may be made by the user without any sizable investment in compilation. At the lower level, the compiler attempts to generate object code. The compiler outlines the steps it is going to take and, as it proceeds, indicates choices it has made and results it has obtained, both at a relatively high level. At any time the user may terminate the compilation and back up to the start of the statement and re-specify it. Similarly, if the system is unable to complete a step in the compilation, the system asks the user for help. The user may then assist the system by giving an alternative and/or more detailed specification of the operation. Interaction with the user on a statement-by-statement basis allows the user to start out with the highest level of statement for a given operation and, if requested, provide the system with additional information [9]. In extreme cases, he may have to respecify the operation as a sequence of lower level, assembly-directed, primitive operations, such as in (2).

Performing compilation on a statement-by-statement basis using an internal representation of the expected run time assembly world state permits an effective and natural level of interaction with the user. However, problems arise with the use of conditional branching in the program, as a statement in the code can, in principle, be reached with different states of the world model. In order to handle these problems, constraints are placed on the branching that a user may specify in his program. Although severe in a general programming sense, it seems that apart from error recovery there is little need for complicated branching paths in an assembly program, and in practice the user may not be affected. In all cases the constraints are intended to permit the compiler to know at compile time the state of the world model at the start of execution of a statement. Iteration and subroutine calls are permitted provided that the compiler can expand the code and simulate it as a single straight line stream. Conditional branches are permitted provided the paths of the branch rejoin with equivalent world model states; equivalence is interpreted to mean states that differ nonsignificantly, such as the use of parts of differing colors or the incorporation of minor differences between versions of an assembly. It is hoped that any practical assembly can readily be partitioned so that these constraints are not a serious inconvenience.

Another area of concern arising from the one-statement-at-a-time compilation is that of parallel code. For any manipulator-level language to be useful as the target

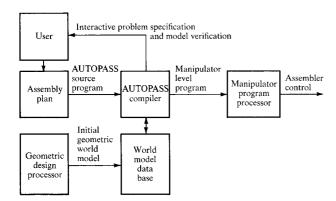


Figure 3 Block diagram of AUTOPASS system.

of AUTOPASS, it must permit the specification of instruction streams that are to be executed in parallel. In cases where there is more than one manipulator, this parallelism may include parallel assembly operations. Although explicit synchronization points may be included in the code, the compiler cannot guarantee synchronization of the manipulators between code synchronization points, and the possibility of collision arises. The approach adopted for AUTOPASS is to require that the manipulators be kept sufficiently far apart that spatial synchronization may safely be independent of time synchronization.

AUTOPASS system

The overall structure of the AUTOPASS system is shown in Fig. 3. The user generates an assembly plan, codes it as an AUTOPASS source program, and also provides an initial geometric model of the assembly world using a geometric design processor. The AUTOPASS compiler processes the source program one statement at a time. The compiler interacts with the world model data base for information on the geometric and physical relationships involved in the assembly operations and with the user to ensure correctness of specification of the assembly plan. The target code consists of procedures in a manipulator-level language. AUTOPASS is embedded in a subset of PL/I and offers the user many of the control and data type facilities of that language. The present status of the work is that the AUTOPASS language has been defined and a complete implementation of the compiler outlined. A translator-writing system employing De-Remer's method for SLR(1) grammars [12] has been used to generate the parsing phase of the compiler. The world model has been designed and the geometric design processor has been implemented.

The world model is a geometric data base in which are represented the geometric structure of objects, the spatial positions and relationships among objects, and the assembly or attachment relationships between objects. The structure of the model representation is a graph in which each vertex represents an object component, an object, or an assembly, and may have a symbolic name. The edges are directed and can indicate four kinds of relationships: part-of, attachment, constraint, and assembly-component.

Each object is modeled internally as a polyhedron, giving the system uniform geometric structures for which interference and other algorithms can be conveniently developed. The polyhedral description is a set of vertex, edge, and surface list structures accessed by a pointer at the object vertex. The polyhedron is created by a geometric construction program in the geometric design processor. Objects are constructed by combining primitive volumes using a parametric description [11]. Since the polyhedral representation is necessarily an approximation to primitives with curved surfaces, a parameter indicating the degree of accuracy of the approximation may be specified. Symbolic names are assigned by the user to any vertex in the graph, thus allowing him to refer to parts and their component volume pieces in the program. The user may name sets of lines, surfaces, and points of an object's polyhedron so that they may be referenced in an AUTOPASS statement.

As an example of a world model, Fig. 4 shows the state of the world at the end of the assembly of Fig. 1. Note that the nodes contain other information not shown in the figure, such as coordinate transformations giving the location of the object and physical properties.

The initial state of the world model is provided by the geometric design processor. This subsystem enables the user to specify parts, fixtures, and the manipulator by designing with parametric primitive volumes, such as a cuboid with given sides or a cone with given height and angle, using an interactive graphics terminal. It also allows him to specify relative positions, types of edge relationships, and names for objects and features of objects.

AUTOPASS language

The AUTOPASS language is embedded in PL/I and offers the user many of the control and data facilities of that language. However, it is generally not necessary that he be familiar with more than the AUTOPASS commands and a very small subset of PL/I control and data type operations, such as conditional branching and DO groups.

AUTOPASS language statements may be divided into two classes, assembly related and miscellaneous. Assembly related statements are concerned with specification of assembly operations and have been divided into three groups: state change, tool, and fastener. Miscellaneous statements are used for such operations as specification of control flow, declaration of geometric variables

and assignment of numerical values to them, and description of inspection operations. The Appendix contains a condensed syntax of the language, including a complete listing of assembly related statements and a partial listing of miscellaneous statements.

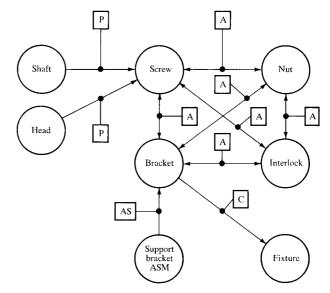
Control of program flow is provided by the branching and looping facilities of PL/I. The program may branch conditionally on, for example, the assembly model number, allowing different features on different models; looping and indexing may be used to specify repetitive operations such as installation of screws holding a cover plate; subroutines may be used for common operations. However, one-statement-at-a-time compilation with an internal representation of the assembly world places restrictions on the use of branching in the control flow, as discussed earlier.

Within each group of assembly related statements there is at least one very general statement with many qualifier options. These general statements permit specification at the highest level of the most complicated operations covered by the group. The other members of a group are generally more specific statements with fewer, more directed, qualifiers. For example, in the case of state change statements, the PLACE statement represents the very general statement for an action which could be specified as a sequence of more specific statements, such as

GRASP LIFT MOVE LOWER RELEASE

It is expected that the user will initially specify operations at a high level and with as few qualifiers as possible. In the course of interactive compilation of the statement, the compiler may find that the statement is either ambiguous, and ask for further qualifiers, or not compilable, and ask for an alternative, narrower, specification of the operation.

The AUTOPASS language permits the use of multiple manipulators. The manipulators available for performing assembly operations, together with those of their motions that may be used, may be declared dynamically in the program. Further, each of the assembly related statements may be preceded by a qualifying hand specification indicating which manipulator is to be used for the operation. The use of these hand specifications is optional. If not given, the system decides which manipulator to use based on the current operation alone. Such a localized decision procedure may produce conflicts with later statements, and the hand specification may be included to enable the user to include in his assembly plan the use of specific manipulators. A number of statement types



- Attachment: may be rigid (objects do not move with respect to each other), non-rigid (objects are attached, but have limited relative movement, e.g., a joint), or conditional (objects are attached only under certain conditions).
- Constraint: may be translational or rotating; described by direction or axis vector and force threshold.
- As Assembly: indicates that an object (and each object attached to it) is a component of an assembly.
- Part-of: indicates a component of a rigid part. (Not all components and part-of relationships are shown above.)

Figure 4 World model for support bracket assembly.

include a *then-hold* option, which commands the manipulator to remain in position at the end of the statement, thereby permitting one hand to hold a part in position while another performs a related task.

One consequence of having multiple manipulators is the possibility of parallel and cooperative operations. AUTOPASS permits specification of parallel motions by means of an IN PARALLEL DO construction. However, since the compiler works one statement at a time, and the individual statements refer to only one manipulator, it is the user's responsibility to ensure that the arms are sufficiently far apart that spatial synchronization may safely be independent of time synchronization. Eventually the problem will be solved by the introduction of constructs to allow specification of spatial and time synchronization for continuous paths.

State change statements

State change statements allow description of assembly operations such as placement and adjustment of parts and motions of the manipulators.

The general purpose state change statement is PLACE with first level of syntax as follows:

- PLACE (object) (prepositional-phrase) (object) (graspingphrase) (final-condition-phrase) (constraintphrase) (then-hold)
- (object) is a fully qualified symbolic name for an object, generally a part.
- (prepositional-phrase) may be IN or ON; the semantic difference between them gives the compiler advice on the type of operation being performed. ON implies a relatively open goal position, whereas IN implies that the goal position is enclosed.
- (grasping-phrase) provides an optional specification of how the fingers are to be positioned for grasping the object. The phrase may be used to advise the compiler where and how hard to grasp a part that is difficult to hold or to avoid gripper caused collisions in later commands. When not specified, the compiler uses its knowledge of the geometry and material of the part to calculate a suitable gripping force and searches for a suitable gripping point in conjunction with its trajectory search computations.
- (constraint-phrase) provides an optional specification of constraints to be met during the execution of the command. These include position, orientation, force, duration of execution of the command, linear and angular velocities, accelerations, and jerks.
- (then-hold) provides an option to specify that the hand remain in position on the completion of the command.

Tool statements

Tools are widely used in assembly operations; in particular, they are used to make adjustments and install fasteners. Individual tool types have individual semantics, which are programmed into the compiler and will have to be extended when new tool types are added to the system. The compiler's semantic routines know where the tool is kept, what accessories it uses, what are its modes of operation, and what sensory feedback is necessary for it to execute its operations.

- OPERATE (tool) (load-list-option) (at-position-option) (attachment-option) (final-condition-phrase) (tool-parameter-list-option) (then-hold)
- (tool) specifies the tool to be used and causes its semantic routines to be called.
- (load-list-option) allows specification of accessories to be used; for example, screwdriver blade or nut socket. It is optional to cover cases in which the tool has no accessories or the tool has already been loaded by previous commands.

- (at-position-option) allows specification of where the tool is to be operated. The option allows for the tool already being in position.
- (attachment-option) allows specification of new attachments caused by the command.
- (tool-parameter-list-option) allows specification of tool operation parameters such as direction of rotation and speed.

Fastener statements

Fasteners represent a highly developed technology in mechanical assembly and can be expected to be the subject of a significant fraction of assembly operations. Fasteners come in many types, shapes, and sizes, are generally small relative to the parts they are fastening, and are generally installed with tools, either hand-held or at fixed work stations. It is expected that manipulators will work in a similar manner, either holding and operating tools or moving the assembly to a work station; for the present the fastener statements of the language cover explicitly the case of fasteners being applied with machine-held tools. Fasteners may be single sided, that is, applied and fastened from one side of the workpiece, or double sided, when the fastener has two parts applied to opposing sides of the workpiece; both cases are covered by the language.

In view of the important role of fasteners in assembly, and the frequency with which they are used, the language provides the user with fastener statements that enable him to describe a fastening operation in terms of the fasteners to be used rather than the tools used to apply the fasteners. As with tool statements, the implementations of fastener statements are highly tool dependent and require special semantic routines for each tool type; in general, these routines make considerable use of sensory feedback.

The general fastener statement is ATTACH with first level of syntax

ATTACH (fastener) (second-fastener-option) TO (at-position) (side-option) (attachment-option) (final-condition-phrase) (using-option)

where the qualifying phrases allow the following specifications:

(fastener) the fastener to be used, e.g., a clip.

(second-fastener-option) the second fastener in a double sided fastener. The second fastener is assumed to be already in position, for example, a nut being held in position by another hand. This phrase informs the compiler of fastener attachments, for example, that a screw is being attached to a nut.

(at-position) where the fastener is to be applied.

(side-option) from which side of the workpiece the fastener is to be applied.

(attachment-option) as for OPERATE statement.

(final-condition-phrase) as for PLACE statement, though only a subset of the conditions is applicable for any given fastener.

(using-option) the tool to be used. Generally (fastener) indicates to the system which tool is to be used; this phrase covers ambiguous cases.

Miscellaneous statements

An important goal in automated mechanical assembly is high reliability of execution of assembly operations. Error testing and optional execution of recovery routines are provided by the VERIFY statement. Note that although the final-condition phrase of a PLACE statement implies inspection to ensure that the command has indeed been executed correctly, the system only generates its own corrective actions by minor modifications to the command being executed. In general, the compiler is designed to produce manipulator sequences that are inherently reliable and allow recovery from foreseen errors. For example, the manipulator operations to fetch a screw from a feeder make a test to ensure that a screw has indeed been collected and, if necessary, retry the operation.

AUTOPASS includes a number of statement types for declaration of names and assertion of relationships. The principal types are

declarations of manipulator names and characteristics. These may be used to name manipulators and to define sets of motions that are permitted.

declarations of spatial features. These commands may be used to define new coordinate frames and features such as lines and surfaces on an object.

assignments to symbolic geometric variables, for example, giving numeric values to points and vectors.

assertions of assembly relationships, giving symbolic names to assemblies so that they can subsequently be referenced as single objects.

assertions of attachment relationships, informing the compiler of attachment relationships that cannot be derived by semantic routines.

The semantics of attachment relationships include rules for updating relationships after a change of state of the world model, for example, when a part is removed from an assembly. Further assertion statements allow explicit cancellation of previously defined attachment relationships.

- 1. OPERATE nutfeeder WITH car-ret-tab-nut AT fixture.nest
- 2. PLACE bracket IN fixture SUCH THAT bracket.bottom CONTACTS car-ret-tab-nut.top

AND bracket.hole IS ALIGNED WITH fixture.nest

- 3. PLACE interlock ON bracket SUCH THAT interlock.hole IS ALIGNED WITH bracket.hole AND interlock.base CONTACTS bracket.top
- 4. DRIVE IN car-ret-intlk-stud INTO car-ret-tab-nut AT interlock.hole SUCH THAT TORQUE IS EQ 12.0 IN-LBS USING air-driver ATTACHING bracket AND interlock
- 5. NAME bracket interlock car-ret-intlk-stud car-ret-tab-nut ASSEMBLY support-bracket

Figure 5 AUTOPASS program for support bracket assembly.

AUTOPASS example

To illustrate the use of the AUTOPASS language, Fig. 5 shows the AUTOPASS program to perform the bracket and interlock assembly operation of Fig. 1. The intent of the example is to show the closeness of the level of the AUTOPASS representation to that of the assembly instruction sheet and that, within the restricted domain and rigid semantics of the language, the problem specification is sufficiently complete for the compiler to generate correct manipulator code.

1. OPERATE is a tool command that involves a routine associated with the type of tool named in the operand. In this case the nutfeeder is a device that can pick up a nut from its holder (feeder) and then place it somewhere else. The final position of the nut (called car-ret-tab-nut in the example) is specified as fixture.nest, which is a subpart of the fixture base. The sequence of steps generated by this operation consists of moving the hand to a position near the nutfeeder, grasping the tool, moving to a position of approach to the nut (in its feeder), operating the tool to pick up the nut, moving to the target position, operating the tool to release the nut. The tool is replaced in its holder when the compiler is satisfied that it is no longer needed; in this case it occurs at the start of the next command.

Note that several of the operations required of the compiler in generating this sequence are common to many of the other high level assembly commands. In particular, trajectory planning and object avoidance are frequently needed functions in the compiler. A function that is not common to other high level commands is the module that has the semantics for the *nutfeeder* tool; there is a different semantic routine for each tool.

The changes that occur in the world model at the end of compilation of this statement are: The coordinate transform at the nut object vertex is changed to indicate it is now in the *fixture.nest*; a new nut takes its place in the feeder and appears as a new vertex in the world

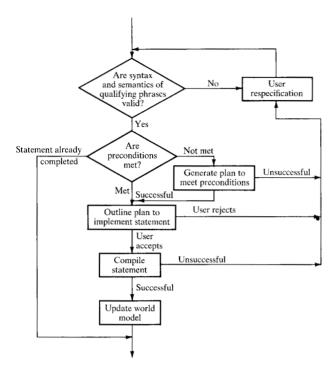


Figure 6 Flowchart of assembly related statement portion of AUTOPASS compiler.

model; the list of constraints for the nut, at the nut vertex, is modified by removing the previous constraints of the nut holder and replacing it with the list of surface-to-surface constraints between the nut and the nest surfaces. The computation of new constraints must be done each time a state change operation is compiled.

2. PLACE is the general movement statement of the state change class of assembly statements. The general sequence of operations that it generates is: move the hand to a position that permits pickup of the object, grasp the object, move along a clear trajectory to an approach position for putting the object in the target place, put it in place.

The spatial position and orientation of the object in the target place is determined by the combination of final conditions listed in the SUCH THAT phrases. The qualifiers indicate which way up the bracket should be (bracket.bottom contacts car-ret-intlk-nut.top), and the alignment of two axes (interlock.hole WITH fixture.nest). In this case the alignment is not to be measured directly but is inferred from the dimensions of the objects. Depending on the geometry of fixture and bracket, the compiler might find the final condition specification incomplete (orientation about the axis is not specified) and would then ask for more qualifiers. Note that interlock.hole and fixture.nest are both assumed to have natural axes; if this proves to be false, the compiler asks for

further specification. In general a library of force feedback routines is used to ensure that spatial final conditions have been met.

In addition to the general operations of trajectory calculation and collision avoidance, the PLACE command also requires a functional module to find possible pickup points on the object. This module must interact with the trajectory calculator since the hand orientation at pickup may render a predicted trajectory infeasible at its final position. After this state change, the transform for bracket is updated and the bracket vertex now has constraint edges showing its relations relative to its new location in the fixture.

- 3. In PLACE interlock ON bracket, ON indicates to the compiler that this is a placement operation in open surroundings, whereas in statement 2 IN indicated some measure of enclosure. This information is used by the compiler in analyzing the required final position of an object and the approach trajectory. Again, orientation is not specified, the compiler queries the user to see if it is important and, if not, is free to choose any orientation that enables it to meet other conditions. Removal of a requirement such as orientation makes the task of selecting a pickup point and trajectory easier, as the extents of the required manipulator motions are less likely to reach the allowable limits in motor space, as defined in a table of motor constraints.
- 4. The parts are now joined by driving a screw, car-retintlk-stud, through the aligned holes into the nut. The DRIVE command indicates that a driving tool (airdriver) is to be used to apply a fastener. The IN phrase indicates that the fastener is going into the object (as opposed to being taken out). The first and second operands form a fastener pair. The semantics are that the second fastener is in place and that the first one is to participate actively in the operation. The next operand is the position where the fastener is to be applied, followed by a final condition phrase giving the termination condition, and finally a phrase with a list of objects that will be attached by this operation. Driving the screw in causes several modifications to the model. Attachment relationships among the vertices of bracket, interlock, car-ret-intlk-stud, and car-ret-intlk-nut are created because of the explicit attachment phrase and because of the implicit attachment due to any fastener type statement. To maintain consistency all the constraint relationships among the objects involved are added to their constraint lists.
- NAME defines the listed objects as an object of type assembly called *support-bracket* and creates a new node in the world model with pointers to the compo-

nent objects. The purpose of this statement is to allow the programmer to name the assembly as an entity which can be used as an operand in assembly statements later in the program.

The above discussion indicates the manner in which the AUTOPASS system is able to reduce the assemblydirected statements of the language to a sequence of independent, unambiguous steps to be entered into the manipulator motion generation phase of the compiler. The operation of the compiler is described further in the next section.

AUTOPASS compiler

The compiler flow chart for assembly related statement processing only is shown in Fig. 6. (Note that this does not include the portion of the compiler that deals with declarations, program control statements, arithmetic statements, etc.) The basic mode of operation of the compiler is to invoke a prestored template for each command type. This template defines preconditions that have to be satisfied before execution of the statement can begin, a network of calls to utility routines (many of which are used by other commands), and a set of postconditions that will exist at the end of successful compilation of the statement and will be added to the world model.

As an example, consider the compilation process for statement 3 in Fig. 5:

PLACE interlock ON bracket . . .

First the overall syntax of the command is checked and the operands evaluated. The world model is used to determine the semantic "correctness" of the statement: are the operands proper types of objects and can the first operand, *interlock*, be moved (perhaps it is rigidly attached to the world coordinate frame)? If these tests fail, the user is informed and asked to modify the command.

The preconditions for the statement type are investigated. In the case of a PLACE statement, a gripper must be free or already holding the second operand. If it is not free, the system tries to free it. For example, a tool it is holding can be replaced or a part being grasped may be released. The former of these possibilities can be satisfied by an operation similar to that of executing the motion portion of a PLACE command.

The system now outlines to the user the general sequence of operations it will follow to implement the statement. In this case the principal elements of the sequence are the evaluation of the goal position defined by the final condition phrases of the statement and the generation of a plan for a trajectory to move the *interlock* from its known current position to the goal position. The

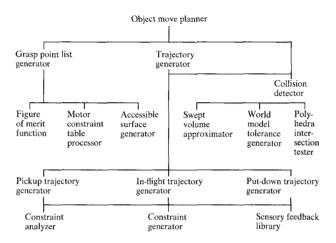


Figure 7 Functional module hierarchy of object move planner.

goal position is evaluated by using the world model to simulate the position of the *interlock* on the *bracket*, iterating if necessary to satisfy all the conditions. In this case the statement is incomplete in that rotational orientation is not specified; the compiler requests clarification and is informed that rotational orientation is not important.

The hierarchy of functional modules used in planning the movement of an object is shown in Fig. 7.

The grasp point list generator produces an ordered list of grasping points on a part that are within the reach of the manipulator. Elements of the list are grasping point regions and are derived by inspection of the world model. The accessible surface generator eliminates from consideration as potential grasping surfaces those parts of surfaces that are concealed by other items, such as surface areas in contact with the worktable. The regions are specified in terms of the extent of the region on the part and ranges of gripping angles. Motor positions are checked against a table of motor position constraints to ensure that only feasible motor positions are returned.

The list is ordered by figure of merit, the value of a function that takes into account such factors as the distance from the center of gravity, the size and shape of the surfaces, and perhaps even the mechanical properties of the material and the surface texture. List elements with high figures of merit are likely to be characterized by opposing plane parallel surfaces large enough to be held by the gripping surfaces of the fingers, whereas lower ranking elements might be for grasping a round pin. In cases in which the GRASPING option is used or the THEN HOLD POSITION is already in effect, the grasp point list generator returns the single element list of the specified grasping position.

The trajectory generator module is given a grasping point and has responsibility for generating a collisionfree trajectory from the current position of hand and part to the goal position. The trajectory is divided into three stages—pickup, in-flight, and put-down—each handled by a separate module. All three modules must ensure that constraints specified by SUBJECT TO constraint phrases are met, or at least can still be met by other functional modules; the put-down module must also ensure that final conditions from a SUCH THAT phrase are met.

The pickup module is given an initial hand clear position near the grasping point, the grasping point itself, and an in-flight trajectory start point, and it has to find a collision-free path for the hand between these points and the nearby grasping point. It uses the constraint analyzer module to provide information on constraints on the part at the current position and the collision detector to ensure that the motions, although small, do not cause any collisions. A library of sensory feedback routines is used to handle world model and part tolerances at execution time.

The in-flight trajectory planner is given start and end points for the trajectory and has to find a collision-free path between them. For the present it is assumed that the world is relatively uncluttered and that collision-free trajectory planning is dealing with rather large clearances around objects in the world. Apart from simplifying the planning strategies, this also makes the trajectory relatively insensitive to geometric tolerances on parts of the world. The output trajectory is a list of intermediate manipulator positions defining a space path in straight line sections and tolerances defining a band about the space path. Run-time modules convert the list into machine control commands to give a smooth trajectory through the positions. The points and tolerances are chosen so that the trajectory is independent of worst case run-time tolerances on the world state.

The collision detector module is used iteratively. Given a trajectory points list, it returns a list of collision points with the extent and direction of the collision; this information is used in refining proposed trajectories. It can be requested to work at several modes of accuracy using different degrees of complexity and accuracy in modeling the volume swept by the motion.

The put-down module takes the part from the end of the in-flight trajectory. This is done by use of a library of sensory feedback routines to ensure that the basic operation is completed and that any final conditions specified have been met.

After a successful implementation of the command has been generated, post-conditions in the state of the world model are updated.

Summary

A very high level programming language for mechanical assembly has been defined that reduces the amount of

effort required to program a new application. The level of the language has been chosen to enable the user to plan the overall assembly, thereby avoiding the need for the system to use artificial intelligence planning techniques, and to enable the system to generate the details of the manipulator motions in the real world. The language is directed towards description of assembly operations rather than manipulator motions. It allows the user to specify an assembly procedure in much the same way he would compose an assembly instruction list for manual use. The user decides in which order parts are assembled, which tools are used, and the general positioning of these objects in the work space. The AUTOPASS compiler transforms this assembly procedure specification into a program that directs a mechanical assembler through the necessary motions to execute the assembly process.

The compiler generates motion commands by using a geometric data base called the world model to simulate at compilation time the expected run-time world. During the compilation process the user interacts with the compiler to resolve any ambiguities detected by the compiler in the problem specification. It is expected that this work will lead to economical ways of programming mechanical assemblers in batch assembly environments.

Appendix: Syntax of AUTOPASS high level statements

In the following statement lists, reserved words are shown in UPPERCASE letters and optional phrases in *italics*.

• State change statements

PLACE object1 ON object2 grasping final-conditions constraints then-hold

INSERT object IN receptor position sensor then-hold EXTRACT object distance sensor

LIFT object distance

LOWER object ONTO surface sensor then-hold
LOWER object distance sensor then-hold
SLIDE object ON surface slide-termination then-hold
PUSH object direction UNTIL final-condition then-hold
ORIENT object SUCH THAT positional-condition sensor then-hold

TURN rotor turning-condition rotation-axis then-hold GRASP object grasp-position hand-position grasping-force MOVE spatial-feature final-condition MOVE spatial-feature TO position final-condition MOVE spatial-feature motion-specification final-condition RELEASE

• Tool statements

OPERATE tool load-list target-position attachment final-condition tool-parameter-list then-hold

CLAMP locking-device SUCH THAT final-condition
UNCLAMP locking-device SUCH THAT final-condition
LOAD tool load-list
UNLOAD tool load-list
FETCH tool from-holder
REPLACE tool to-holder
SWITCH tool ON|OFF
LOCK locking-device attachment
UNLOCK locking-device release

• Fastener statements

ATTACH fastener second-fastener TO target-position side attachment final-condition

DRIVE IN drive-fastener target-position final-condition using-driver attachment driver-parameter-list

RIVET object-list target-position side attachment FASTEN object1 TO object2 more-objects WITH fastener target-position final-condition

UNFASTEN fastener-list source-position release targetposition

Note Any statements in the above three classes may be preceded by a qualifying hand specification:

• Miscellaneous statements (partial listing)

VERIFY inspection-condition inspection-action-list

OPEN STATE OF locking-device IS final-condition-list

CLOSED STATE OF locking-device IS final-condition-list

NAME object-list ASSEMBLY assembly-name

END

Acknowledgments

WITH hand-name . . .

The authors thank P. M. Will, D. D. Grossman, H. A. Panissidi, and M. A. Lavin for helpful discussions in the course of this work, J. A. Darringer for assistance in the preparation of a grammar and a parser for the language, and P. Goldberg for reviewing an early draft of this paper and for making constructive suggestions.

References

- Proceedings of the 5th International Symposium on Industrial Robots, IIT Research Institute, Chicago, Illinois, September 1975.
- J. F. Engleberger, "Economic and Sociological Impact of Industrial Robots," Proceedings First National Symposium on Industrial Robots, Chicago, Illinois, April 1970, p. 7.
- on Industrial Robots, Chicago, Illinois, April 1970, p. 7.
 3. C. A. Rosen and D. Nitzan, "Some Developments in Programmable Automation," Proceedings IEEE Intercon 75, New York, April 1975.
- D. G. Bobrow and B. Raphael, "New Programming Languages for Artificial Intelligence Research," Computing Surv. 6, 153 (1974).
- C. Hewitt, "Description and Theoretical Analysis (Using Schemata) of Planner: A Language for Proving Theorems and Manipulating Models in a Robot," AI Memo No. 251, MIT Artificial Intelligence Laboratory, 545 Technology Square, Cambridge, MA, April 1972.
- S. E. Fahlman, "A Planning System for Robot Construction Tasks," Artificial Intelligence 5, 1 (1974).
- R. Finkel, R. Taylor, R. Bolles, R. Paul, and J. Feldman, "AL, A Programming System for Automation," Stanford Artificial Intelligence Laboratory Memo. AIM-243, STAN-CS-74-456, Stanford University, November 1974.
- 8. P. M. Will and D. D. Grossman, "An Experimental System for Computer Controlled Mechanical Assembly," *IEEE Trans. Comput.* C-24, 879 (1975).
- R. M. Balzer, "Automatic Programming," Technical Report RR-73-1, U. S. C. Information Sciences Institute, Marina del Rey, CA, 1972.
- G. E. Heidorn, "Automatic Programming Through Natural Language Dialogue: A Survey," *IBM J. Res. Develop.* 20, 302 (1976).
- D. D. Grossman, "Procedural Representation of Threedimensional Objects," IBM J. Res. Develop. 20, 582 (1976).
- 12. F. L. DeRemer, "Simple LR(k) Grammars," Commun. ACM 14, 453 (1971).

Received May 3, 1976

The authors are located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.