Stanford Artificial Intelligence Laboratory
Memo AIM-285
Computer Science Department
Report No. STAN-CS-76-568

August 1976

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
Computer Science Department
Stanford University
Stanford, California 94305

PROGRESS REPORT 3

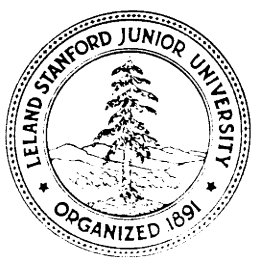
Covering The Period December 1, 1975 to July 31, 1976

EXPLORATORY STUDY

OF COMPUTER INTEGRATED ASSEMBLY SYSTEMS

by
T.O.Binford, D.D.Grossman, C.R.Liu, R.C.Bolles, R.A.Finkel,
M.S.Mujtaba, M.D.Roderick, B.E.Shimano, R.H.Taylor,
R.H.Goldman, J.P.Jarvis, V.D.Scheinman, T.A.Gafford

Prepared for:
NATIONAL SCIENCE FOUNDATION
WASHINGTON D.C. 20550



ABSTRACT

The Computer Integrated Assembly Systems project is concerned with developing the software technology of programmable assembly devices, including computer controlled manipulators and vision systems. A complete hardware system has been implemented that includes manipulators with tactile sensors and TV cameras, tools, fixtures, and auxiliary devices, a dedicated minicomputer, and a time-shared large computer equipped with graphic display terminals: An advanced software system called AL has been developed that can be used to program assembly applications. Research currently uhderway includes refinement of AL, development of improved languages and interactive programming techniques for assembly and vision, extension of computer vision to areas which are currently infeasible, geometric modeling of objects and constraints, assembly simulation, control algorithms, and adaptive methods of calibration.

TABLE OF CONTENTS

I. OVERVIEW

Thomas O. Binford

Artificial Intelligence Laboratory
Computer Science Department
Stanford University

The author is a Research Associate in the Computer Science Department and is a co-principal investigator on the Computer Integrated Assembly Systems Project.

INTRODUCTION

This report is the third in a sequence of reports summarizing research progress in Computer Integrated Assembly Systems. This project, supported by the National Science Found'ation, is concerned with. the software technology of programmable automation, including computer controlled manipulators and vision systems. The basic goal is the simplification of assembly and visual programming.

Prior to the period covered in this current report, a complete hardware system was implemented, including manipulators with tactile sensors and TV cameras, tools, fixtures, and auxiliary devices, a dedicated minicomputer, and a time-shared large computer equipped with graphic display terminals. An advanced software system called AL was developed as a research tool for studying problems in assembly automation.

During the past year, the AL system has been debugged, improved, and extensively documented. In the near future AL will be used to program some simple assembly applications examples. From a succession of applications, generic assembly routines can be identified and accumulated in a library. Additionally, a design review of AL has recently been started to identify its strengths and weaknesses. As a preliminary to this review, a questionnaire concerning the potential use of AL at other laboratories has been circulated, and responses are being received.

Work has begun on the classification of assemblies; assembly processes, and manipulators. Improved languages and interactive programming techniques for assembly and vision are also being studied. It is hoped that this work will lead to the extension of computer vision to areas that are currently infeasible, such as picking discrete parts out of a bin. Research has also included geometric modeling of objects and constraints, assembly simulation, control algorithms, and adaptive methods of calibration.

This overview offers a concise summary of recent progress by the varied research efforts that constitute the Computer Integrated Assembly Systems project. The progress report is divided into two main sections: work directly related to AL and assembly, and work on computer vision and modeling. It is through a comprehensive project of this sort that -prototype systems will eventually be developed for practical programmable assembly.

AL SYSTEM AND ASSEMBLY

Extensive experience has shown that the debugging process is vastly more time-consuming than people are willing to admit. In AL, this problem is accentuated because the system interacts with the real world in real time and because there are two computers and a language hierarchy. In response to these problems, Raphael Finkel has developed **ALAID,**

an interactive debugger that allows 'AL program execution to be monitored and that provides a means of patching AL programs to avoid the otherwise lengthy debugging loop. A LAID resides on both computers and partial recompilations are done on the planning machine, which maintains symbol table information. A by-product of the ALAID design is that it can be used to interface complex feedback routines on the planning machine to the runtime execution of AL programs. Finkel's work is described in the section *ALAID: An Interactive Debugger for 'AL.* In industry, as microprocessors spread, multi-machine hierarchies will be usual and debugging systems like ALAID will have wide potential application.

Within the AL run-time system, the speed of routines that transform between Cartesian space and joint-angle space is of considerable importance. Bruce Shimano has derived faster procedures for computing these transformations. Additionally, he has found, simpler procedures for calibrating force sensors, needed in those industrial applications that involve force-controlled compliant motions. Compliance using sensing is an essential means of coordinating two or more devices. Shimano explains his contributions in the section *Improvements in the AL Run-Time System.*

Russell Taylor has been studying ways to generate AL motion programs automatically from higher level task descriptions. A paradigm of progressive refinement is used to expand a single statement like "put peg in hole" into a succession of detailed steps necessary to get the job done. The task is non-trivial if attention is given to making the generated code rugged with respect to positioning errors. The work therefore requires an ability to maintain extensive planning information concerning the description of the semantics of the manipulator language, the definition of the task, the objects being manipulated, and the execution-time environment. Taylor discusses his approach in the section *Generating AL Programs from High Level Task Descriptions.*

Shahid Mujtaba has analyzed the automatic assembly of a pencil sharpener and compared the motion times for the Stanford Arm with those obtained by the technique of Methods Time Measurement for a person doing the same assembly. The task was also analyzed using assembly primitives developed at Draper Lab. For this task, the mechanical arm was considerably slower than the human. Identifying, the sources of this lethargy should make considerable speed improvements possible in the future. This work is discussed in the section *Case Study of Assembly of a Pencil Sharpener.*

In recent years, the manipulator hardware has stabilized, and the need for hardware construction has declined considerably. Nevertheless, certain hardware necessary for AL's operation is being developed. These additions, which are not described in this report, include hardware that provides increased PDP-11 memory for the run-time system, a second arm interface to facilitate testing AL's novel software capability of controlling two arms in simultaneous coordinated motion, a force-sensing wrist to provide greater accuracy in determining forces and torques than is possible by monitoring servo errors, and an improved gripper that should allow greater versatility in grasping small objects.

COMPUTER VISIONANDMODELING

Robert **Bolles** has 'shown that the execution time and memory size of his experimental program for visual inspection are almost practical, and that programming is simple. 'His recent work is concerned with establishing a firm mathematical basis for making verification decisions. A least-squares technique is used to combine available information and derive estimates for location and location accuracy of objects. Bayesian probability is used to determine necessary confidences within a sequential pattern recognition scheme. These well-known techniques are combined to answer various questions raised within a verification 'vision system. The work is described in the section *Mathematical Tools* ***for Verification*** *Vision.*

Michael **Roderick** has been investigating the possibility of reducing the sampling rate used by the run-time computer to control the Stanford **arm.** His analysis is based on the use of z-transforms, since **Laplace** transforms **are** not applicable for low sampling rates. Specific recommendations are derived for the lowest possible sampling rates at which the Stanford Arm might be controlled. Roderick's approach is described in the section ***Discrete*** *Control of* ***the*** *Arm.*

A previous progress report described POINTY, an interactive program for generating object models by manual positioning of the manipulator. During the course of writing applications programs, **Shahid** Mujtaba has found this technique to be an aid in reducing the **labor** of coding AL models. He has prepared a guide to the system entitled *POINTY User Manual,* useful for both training and reference purposes.

David Grossman has used a geometric modeling program to simulate discrete parts tolerancing, showing how manufacturing errors can' propagate until they affect the probability of successful assembly. The assembly of discrete parts is strongly influenced by imprecise components, imperfect fixtures and tools, and **inexact** measurements. Production engineers must choose among alternative ways to select individual tolerances in order to achieve minimum, **cost** while preserving product integrity. Grossman describes a comprehensive Monte Carlo method for systematically analyzing the stochastic implications of tolerances and related forms of imprecision. The **method** is **explained** in the section *Monte Carlo* ***Simulation*** of ***Tolerancing.*** This work is one example in which technology developed initially for programmable assembly is proving applicable in a much wider domain, particularly manual assembly.

II. ALAID: AN INTERATIVE DEBUGGER FOR AL

Raphael A. Finkel

Artificial Intelligence Laboratory
Computer Science Department
Stanford University

The author is currently an Assistant Professor in the Computer Science Department, University of Wisconsin, Madison, Wisconsin 53706. At the time this research was performed, he was a graduate student in the Computer Science Department at Stanford University.

CHAPTER 1

BACKGROUND

The road to constructing working code in any programming language can be long and tedious. Several of the important milestones are these: 1) understanding the problem *(conceptualization),* 2) creating an algorithm to solve it *(design),* 3) writing that algorithm in a suitable programming language *(formalization),* 4) submitting the program to the scrutiny of the computer *(compilation),* 5) running the program (execution), 6) getting the program to do what was intended (debugging), 7) making sure that the program behaves under diverse conditions *(testing),* and 8) production runs of the-finished prdgram *(bliss).* These steps are not necessarily distinct; it often happens that conceptualization, design, and formalization are performed simultaneously, and the stages from formalization through debugging are often repeated several times.

The problem of successfully traversing this route is compounded in the particular case of arm code by several factors. The first obstacle is that the real world is less tractable than the highly controlled world of the computer. Any given strategy to accomplish a given task may fail, because the actual real-world result of the program may not be what the programmer desired. An effort to insert a pin in a hole may result in a jammed pin or a jarred workpiece.

Not only is the, world, recalcitrant, it also is complex. A programmer in a purely algebraic language can attempt to keep in mind the various states of his program at different places. State information like loop invariants provides enough environment so that reasonable code may be written. Not so in the realm of mechanical manipulation. Objects can be modelled, but only partially, and the extent to which models reflect the real objects is subject to design choices on the part of the programmer. He may only discover during debugging that his model is incomplete in a crucial way, that some important feature of an object has been omitted from consideration.

Design of appropriate error recovery routines depends greatly on what errors are encountered. It is a waste of effort to design the program to carefully detect and remedy an error like dropping a workpiece if in fact the arm never or rarely commits this error in practice. It is even 'more frustrating to fail to foresee the possibility that a screw hole is mispositioned if that error turns out to be frequent. Experience and sharpened intuition can slowly train a programmer in what sorts of errors to expect, but the learning process is full of necessary and clumsy-iterations through the debugging loop.

Another feature (some would no doubt consider it a bug) of the real world is that many actions are irreversible. In algebraic languages, most actions have inverse actions, and if it is

important to be able to back up, care can be taken to preserve either the state of the computation or a history of the actions that have been taken so that the code can be retried.. But the moving finger dips, and having smashed, moves on. Nor all our history lists nor glue shall lure it back to fix but half the wreck, nor all our work shall make it look like new. Even non-destructive actions, like putting a pin in a hole, cannot be reversed automatically, since there is no way to determine what forces ought to be applied during the backward motion; they are not closely related to the forces applied during the initial insertion.

In many programming languages, the debugging loop is exceptionally long. AL is no exception. In order to fix a known error, it is necessary to modify the source code and then to resubmit the program to the compiler. The compiler (which is fairly slow) produces an intermediate output, which, is finally loaded into the PDP-I 1. And then the pieces must, be reinitialized to their starting positions, and the race must be run up to the point where the failure occurred in order to test the fix. The failure may not be very reproducible, so the new code may not be easily tested. The effort that must be exerted to locate a bug in the first place can be immense; Making some small change and resubmitting the program in the hope that the change will make the bug more traceable is very tedious due to the long turn-around.

There are several debuggers for the PDP-1 1 machine; one typical example is 1 1-DDT, based on RUG, another similar debugger, and implemented by Jeff Rubin at Stanford [Binford 75]. DDT is a symbolic interactive debugger, but it has no knowledge of AL; its microscopic vision cannot see the forests of manipulator code built out of the trees of machine instructions. The AL runtime environment has been implemented and debugged with the assistance of 11-DDT, but the debugger is fairly useless for debugging manipulator programs, especially if the person using AL is not an expert on the implementation. The problem is mostly one of level; 11-DDT is a low-level debugger, and AL is a high-level language.

The intent of this report, which is taken from the third chapter of my thesis, is to examine the problems of preparing correct manipulator code and to suggest the design of a user interface that assists the programmer in fulfilling his function. This interface will have some of the flavor of a debugger and some of the flavor of an operating system. Although it is based specifically on the implementation of AL, the design of the interface is of interest for more general reasons: it provides added insight into control structures for operating mechanical devices under programmed computer control, and it proposes a uniform debugging and preparation technique that might find use in any large and complex programming environment. For this reason, this report may be read independently of the remainder of my thesis.

The discussion has two distinct flavors. On the one hand, philosophical issues dealing with debugging in general and arm code debugging in particular are treated in a rather abstract fashion. On the other hand, details of implementation are mentioned in an attempt to demonstrate how needed facilities can be obtained. This second type of discussion deals both with a preliminary test 'implementation currently running in the AL context and with extensions to it, both simple and complex.

CHAPTER 2

CLASSICAL INTERACTIVE DEBUGGING

Success is the mother of ***disaster.***
-- V. Cerf

The art of debugging programs has developed as a bastard son of the art of computer programming. The first debugging was done by staring at the code until a bug was found, or by inserting intermediate output statements to test hypotheses concerning the expected state of the computation. An entire generation of programmers became familiar with the core dump, either in raw machine representation (it is said that one can even come to love hexadecimal) or with some preliminary transcription into instructions or, more often, ASCII or EBCDIC tcx t rcpresen **tation.**

Programming in machine language has given rise to such interactive debuggers as DDT, which allow the user to interrupt his program, investigate it, make changes, and then allow it to continue. The fact that in machine language, program and data are represented by the same forms, namely, machine words, makes such debugging especially natural. Fundamental to such debugging is the concept of breakpoints, which are locations in the program of interest to the programmer during his debugging. When breakpoints are encountered, the interactive terminal is connected directly to the debugger, and execution of the program is suspended1 During this time, the user 'can examine the values of variables, set and remove breakpoints, and then allow the computation to continue.

High level languages like FORTRAN and ALGOL no longer maintain a unity of program and data, and debugging techniques either use post-mortem dumps and traces of procedure calls and variable. assignments, or they force the user to debug the generated machine code using a DDT-like debugger. Some student-oriented languages (SNOBOL and ALGOL W come to mind) provide the facility of optional post-mortem dumps that wind back through the **stack** of procedure calls and print out the values of all variables for each procedural **level.** These languages also allow the tracing of procedures, so that an examination of the program output will reveal the sequence in which control entered and exited procedures. In short, the debugging facilities allow examination of the path of control and the values of variables. (See **[Satterthwaite 75]** for a discussion of the debugging facilities of ALGOL W and a very good summary of the history of debugging.)

Interactive high-level languages, like LISP and SAIL, increase the user's control over his program. He can interrupt it at will and restart it. (This alone is a fantastic advantage over batch systems. Many are the times that there is no apparent failure of the program, but the

programmer suddenly remembers a mistake, and he can stop the program to fix it without wasting unnecessary computer time to complete a worthless computation.) The idea of a program **trace** has been strengthened to allow the user to do his examination whenever an interesting place is reached. All the features developed in DDT for machine language programs are incorporated **into LISP** debuggers.

LISP completes the circle; it is a high-level language that unifies program and data, so that a program can be self-aware; it has become natural to **write** LISP debuggers in LISP itself.

Once it has become clear why a program is failing, it is often useful to patch that error and let the program proceed. In this way the next failure can be found, and the patch can be tested. An interactive debugger allows not only ***examination*** but, also ***modification*** of the contents of the program. In a language like LISP, where there is no distinction between programs and data structures, the facility to modify data structures is extremely powerful, because it implies a power to modify the program itself. This power is reflected in the debugging packages found in most LISP implementations. (See, for example, **[Teitelman 74]**).

Flow of control is also a plaything in' the hands of interactive debuggers. If some bad code is to be avoided, the program can be made to jump over it. Special code to be executed for restorative purposes (initialization routines) can be executed directly from the debugger, and then control returned to the ailing program.

In most algebraic languages, the input syntax is compiled into a machine language representation, which is then executed. Interactive debugging of programs in these languages is made difficult by the fact that the programmer must be able to associate machine code to source code. . This process is made **easier** by such debuggers as DDT and RAID (the latter being **a** display version' of DDT, a standard debugger on the **PDP-10),** which can display memory cells in various modes, including symbolic instruction mode and various arithmetic modes. -Another feature that these debuggers offer is that they can refer symbolically to locations in memory; if the programmer has named a variable "felicity", then that name may be used during debugging as well. If a program label is "charity", then that is how the debugger will refer to that section of code. Thus, these debugging programs contain *disassemblers* that can take assembled code and recreate the source that spawned **it.** In order to allow the user to modify instructions, classical debuggers also include primitive assemblers **as** well.

The use of symbolic names for instructions (as opposed to numeric format), for labels, and for variables is a special case of a very important 'idea in debugging: the code that is being debugged- should be presented to the programmer in as close a way as possible to the code that he himself wrote. Unfortunately, DDT and RAID only work on the machine language level, a level at which most programs are not written. A significant effort in the direction of source-language debugging is the debugger **BAIL [Reiser** 751, which **is** used for debugging SAIL **[VanLehn 73]** programs. It is capable of displaying the exact text that is being executed (by maintaining cross-references into the source file) and takes commands that are a

subset of standard SAIL commands, especially procedure invocations. In this case, the disassembly process is made possible by keeping pointers to the source code, not by examination of the object code. BAIL contains a primitive compiler, in that it can parse some constructs of SAIL for the purpose of patching code. Other source-language debuggers also, exist; COPILOT [Swinehart 74] includes a sophisticated example. None of these can display macros in their expanded form.

Style of debugging varies from person to person. A common technique is to proceed the program until a fatal error occurs (like a memory reference trap). By examining the failing instruction it is often possible to deduce what data are wrong; these data are then examined. If they are indeed wrong, an attempt is made to localize the bug by installing a breakpoint at some place where it is expected that the data are still right. The program is then restarted or backed up to a safe place and allowed to proceed. When the breakpoint is encountered, the suspect data are examined. If they already are in error, an earlier breakpoint is installed and the process is repeated. If they are still good, single stepping is employed to see where they go wrong. One powerful technique is to use a procedure that checks the consistency of the World, and to call this procedure from the debugger at each of the test breakpoints. [Charles Simonyi, personal communication] Procedure calls that are expected not to be relevant to the problem are executed, as a single step. Finally the error is localized and the programmer convinces, himself that his code in fact makes a mistake. It is surprising how resistant the human mind is to the suggestion that a perfectly straightforward piece of code might fail under some circumstances. Once the error is found, it is fixed in place if at all possible (and it often is possible if the debugger is capable of patching one piece of code in place of another) and tested by backing up once again and seeing if the same fatal error occurs as before.

A more cautious technique of debugging is to step through all, new pieces of code one instruction at a time in order to make sure they do not fail. The methods for determining sources-of error and fixingthem are similar to the outline above. This method works well if the failing program is not heavily context dependent, 'or if the error is so severe that the program never works. A later stage of debugging deals with programs that fail only occasionally; stepping through such programs to find bugs is tedious and generally unproductive. In these cases, debugging often proceeds by setting breakpoints and trying to figure out the exact situation that causes failure, then reproducing the failure at will until the source of the error has been tracked down.

In summary, what we might call classical interactive debugging has these interlocking features:

1. Symbols are used extensively.
2. Both examination and modification are possible.
3. These constructs are available for examination and modification:
   - program labels
   - program code
   - flow of control
   - values of variables (data structures)

4. Backing up (or restarting) and patching are typical operations.

The realm of examination includes such abilities as searching for code or data having a particular form, displaying instructions and data, setting *breakpoints or traces on code or variable-reference so that the flow of control and history of variables may be traced, and single-stepping to execute only one instruction or procedure call at a time. Modification involves depositing replacement code or data, zeroing large blocks of data, interrupting execution, restarting execution at arbitrary places, inserting new labels or moving old ones (although no debugger is yet capable of changing all old references to a label to correspond to the new meaning, with the possible exception of SNOBOL), and directly executing instructions from the debugger, especially procedure invocations.

CHAPTER 3

INTERACTIVE ARM CODE DEBUGGING

The fact that AL is a real-time language for control of real-world devices in an environment of multiple processes residing on several machines presents some unique problems for the design and implementation of a debugger. The purpose of this chapter is to discuss debugging issues raised by manipulator programming in general and by the AL language in particular. Some conclusions will be reached not only for the design of a debugging package for AL, but also for the implementation of AL so that it might be easier to debug. These conclusions can be generalized' beyond the context of debugging to the larger issue of preparation of workable code in general and arm code in particular.

Section 1
Block Structure

AL is a block-structured language. In most ways, it conforms to the scope rules standard in such languages. As a block is entered, all variables for that block are declared and room made for them in the environment of the current process. These variables include special ones for condition monitors, force feedback, events, and calculators. As control exits from a block, each of these variables is released and its space reclaimed. An attempt is made to sever any connection between these variables and the state of the computation outside the block: variables and expressions are unlinked from any graph-structural relations, condition monitors are awakened to tell them to disappear, and events are returned to the kernel, which awakens any process waiting on them with a failure indication.

Due to details of implementation, block structure is violated in a few ways. Changers can be applied to a global variable in such a way that after control leaves the block in which the changer was applied, the global variable still has the changer associated with it; this anomaly should perhaps be considered more of an implementation bug than a part of the design of the language.

During the course. of debugging, it often happens that control must be forced to exit from a block or transferred into the middle of another block: This facility is, most easily implemented by associating information on variables that must be created and destroyed with each block entrance and exit. A premature block exit is then simulated by jumping to the end of the

block where the exit. code is to be found. 'If the programmer wishes to make a wild jump from one part of the program to another, it is only necessary to carefully close all the blocks that lead' out to the first common ancestor of the current locus of execution and the desired one, and then to enter all blocks that lead in' to the specified place. Once the appropriate block has been entered, a direct jump to the-indicated code should work.

Section 2
Parallelism

The first problem that AL presents is its parallelism. The source language itself allows the user to split control explicitly into several threads of-execution. These threads are treated as separate processes in the running program. Condition monitors are implicitly also understood to refer to processes that have a special scheduling priority. Even less explicit is the use of processes to implement joint servos and force feedback variables. Parallelism in AL is therefore only partially a result of the explicit nature of the language; any implications such simultaneity may have for the debugger are equally valid for any language supporting concurrency (like SAIL, or concurrent PASCAL). Other parallelism derives from the real-time aspect of AL: condition monitors are intended specifically for rapid response to real-time feedback. Still other parallelism is due to implementation decisions taken in the coding of AL: Both servoing and force monitoring make use of the process structure available in the runtime environment. Each joint is separately treated by a software servo; to read forces on. the arm it is necessary occasionally to recompute some configuration-dependent information. Force calculation is readily scheduled as an infrequent concurrent process, while, servos are frequently executed.

During a debugging session, one would like to attack problems in one thread of execution without interfering with other threads. This consideration is especially important if one of the threads is causing an arm to move, and a bug has been discovered in a different and independent piece of code. It is not a good idea to throttle the machine by debugging at high machine priority while waiting for interactive input (unlike 1 I-DDT, which assumes control of machine interrupts). The problem of non-interference has been discussed by D. Swinehart [Swinehart 74]; his interest is to allow the debugger to oversee and report on the state of continuing computations without interfering with them. The method he employs is to make the debugger itself a process Just like the others.

The debugging process is a window into the inner secrets of any other process it chooses to examine. It can link itself into the data structures of that process and therefore it has access to variables and code local to the object of its scrutiny. If variables are held in common among several processes, then the data structures within any one of them point to all the global variables. Of course, there is a problem of naming, since different variables may be

called the same thing in different blocks and in different threads. This problem of non-unique naming is fairly well understood; standard PDP-IO DDT (and RAID) have separate symbol tables for separate blocks, and they maintain a *current* block. If a variable is requested, it is sought first in the current block, and then in successively more global blocks. Commands are provided to change the context by moving to other blocks. This separate symbol table idea can be generalized to the domain of concurrent processes; each one has its own symbol table that associates internal names (those that the process itself uses when making reference 'to a variable) with **programmer-defined** names. Along with a current process currently under scrutiny by the debugger is a symbol **table** that dictates how that process names its variables.

, The naming problem takes a different form with regard to names of processes themselves. Somehow it must be possible to point to a process and sic **[sic]** the debugger on it. Naming problems are exacerbated by the fact that processes come and go during the execution of the program. It is necessary to be able to name a process that does not yet exist in such a way that when it does exist, the debugger will use its symbols.

There may be some use to structuring the set of processes that have been examined so that one may return to a previous one. A stack of processes that have been under examination is one such technique, but it suffers from the. fact that the order in which one examines . processes during debugging is often independent of the actual structure of processes in the program. Therefore, stacking the processes violates the dictum of naturalness, which implies - that the debugging structures should **be** the same, as the programming structures. A more 'attractive alternative to stacking is to name processes as parents, siblings, and offspring of other processes. If the current process splits into three (for a **COBEGIN** nest), then each of the three can be considered an offspring of the current process and siblings to each other. It , ' ' is, however, unclear to what extent it is necessary to provide such process structure during the debugging phase; **what** is clear is that each process that the user may wish to examine must be accessible, either by an explicit name or implicitly in relation to other processes.

It is conceptually easiest for the debugger always to be pointed at one particular subject process, known as the *current process.* All commands to examine or modify data or control structures **will** apply only to that process. If another process should encounter a breakpoint, it 'will send the debugger ,a message and wait, but **will** not be automatically connected as the current process. This concept can be fruitfully generalized to the idea of *'current context,* which **is** a related set of processes, all of which are under scrutiny. If the context includes only one process, and it splits into several new ones, then each of the offspring are also part of the same context; no explicit switch is necessary to examine them. Contexts are related to each other according to the lexical pattern of the program; it makes sense to switch to the next more general context (the parent) or to one of several more specific contexts (the offspring).

Parallelism makes the problem of *backing* up especially difficult. One seldom discovers a bug until it has caused incorrect actions; the debugger must assist the user in restoring the world to the state it had before the bug struck in order to track it down and try repaired code. The

essential ability is to have access to all the state of the machine that defines the world at any point in the execution and to be able to modify it.

First, the *point of execution* must be defined in the presence of parallelism; it refers to a point along each of the many threads that may be active. That is, the context in which execution is understood is the most general context of all: the outermost block. This definition is more restrictive than is strictly necessary, since backing up may only be needed with respect to one or several threads, that is, within a more specific context. However, there is no guarantee that a given context will contain any active processes unless that context is the most general one.

Next, a debugger that deals with parallel processes must be able to observe and manipulate all synchronization control between them. If one thread has produced a signal that another thread will eventually await, and a bug strikes, it must. be possible to back up past the signalling of the event. This consideration forces the debugger to keep track of what events are signalled and awaited. If the event has been successfully awaited by another process, then to back up the first requires that the second one be backed up at least to the point at which the event was awaited. This consideration forces the debugger to hold a context large enough to include all processes that use any important events.

An alternative that has shown some success when signals and waits, are paired into a *synch* command involves signals that have short lifetimes. If no process has received a signal after' a period of time, the signal disappears, and the signalling process repeats the signal. Backing past such a piece of code is easy, since its effect, is transitory. This solution cannot be generalized to the types of events present in most parallel structures, including those found in AL.

Section 3
Levels of Detail

Another problem presented by AL is the fact that the code and data exist at several levels of detail. Each statement in the source language is translated to a set of pseudo-operations for the target machine; that machine is implemented in PDP-11 assembler language, in which each pseudo-operation is expanded to a hand-coded procedure. Variables are used to clothe many-disparate entities, including program variables, and, expressions, which are in many ways symmetric to program variables. Condition monitors are also implemented as a funny kind of variable, the value of which determines the state of the monitor and the code that it executes. Force feedback is implemented through yet more complicated variables.

The question is at what level the user would like to carry on his extermination activities. For errors in logic of his program, he would most like to work in terms of the source language. If

he wants to know the current status of the affixment structure, information not available in the source language, he might want to track through the graph structure, assuming that he understands how it is put together. If he is trying to implement a new pseudo-operation, he may want to work at the level of machine instructions. In general, the debugger has the responsibility to make accessible all information and power that the user will need in a form that he can understand. This implies that there should be commands for examining graph structure, even if it is not subject to scrutiny in the source language. The instructions that are being executed should be visible either in source formalism, pseudo-code, or machine, code, at the desire of the user.

Part of this problem would disappear if the source formalism were directly interpreted in the target machine. This is not an absurd idea, although the implementation chosen did not follow that direction. Debuggers for LISP (See [Teitelman 74] for example) take full advantage of the fact that all code in LISP is representable within the data structures native to the language. Therefore there are no hidden structures' except for compiled routines, which are usually not used unless they are assumed to be bug-free.

Short of implementing AL in AL, some steps could be taken to add features to the language that allow examination of structures. One such feature would allow the program to discover if one frame is attached to another. It may happen that such investigatory functions would be useful in their own right as parts of programs, independently of debugging strategies. Another suggestion that leads still farther into the LISP-like realm is to make the debugger homoiconic with the language, that is, let all debugging commands be available as statements in the language. Then extend the debugger so that any statement in the language can be given to it to execute.

The same argument holds at the level of the pseudo-code; not only should statements of the source language be directly executable from the debugger, but pseudo-operations should also be accessible. These' operations are useful for performing only a part of a full-fledged statement. For example, to cause a' new variable to exist, it is most convenient to execute the pseudo-code that creates variables.

The problem of unusual data types is related to the level-of-detail problem. Not only does AL have algebraic types (scalar, vector, transform), it also has control types (events, expressions, condition monitors, force feedback variables, motion tables), each of which has a peculiar representation of its own. How is a motion table to be displayed so that the user can see its destination, initial point, and what clauses have been associated to it? This problem requires a disassembler with some rather special knowledge of not only the code, but also the data structures involved in the runtime implementation. The strange format used for such things as force feedback variables, condition monitors, and motion tables can also pose problems for modifying their information. According to the philosophy of representing constructs in source formalism wherever possible, the way for the user to enter a correction to these structures would be to restate his source code, and let the debugger regenerate the proper forms. In this sense, the debugger should have the full power of the compiler.

Yet another suggestion with regard to language design can be' made in regard to these complex data structures. In order to easily point the debugger at any object, it is most convenient to somehow label that object. *Variables automatically have names; that is why it is so easy to refer to them during debugging. We have already seen that the fact that processes do not have explicit names Causes problems in pointing the debugger at a particular process. The same consideration carries over to such cumbersome structures as motion tables, not to mention statements. A reasonable suggestion is to associate variables of the appropriate type with each of the control and data structures used by the AL language. Not only would this association facilitate debugging, but it would also render the concepts represented by the structures. more flexible. For example; if motions are values that can be named by variables, it becomes natural to consider composition and extraction operators that can act on this datatype. Even some "arithmetic" operators may not be out of the question; perhaps a scalar multiplied by a trajectory changes the overall timing of the motion. This concept of increasing flexibility by explicit naming also arises in the context of limitations to the language. The fact that the same suggestion arises naturally in the context of debugging supports the hypothesis that debugging and programming are very similar activities that are carried out in the same domain and require identical structures.

Section 4
Multiprocessor Environment

The best way to gain the full power of the compiler without actually writing one for the execution system is to use the one that already exists. As we have seen, the trajectory calculation problem is hard enough to warrant using a larger computer for at least that stage of the compilation. This desire leads to the need for a linkage between the target machine and the compiler that will allow parts of programs to be recompiled'and reloaded during the debugging phase. If the input formalism to the debugger is to be the same as the statement formalism of AL, then for every debugging request it is necessary to compile the code; implied in the request and make the resulting pseudo-code available to the debugging process.

The idea of a two-machine link in which one machine can monitor and control the other is found in some recent computers. The best example is the DEC KL-10, which uses a PDP-11/40 to monitor and control the PDP-10 main machine. Another example is found in the CDC 6600 computer, in which a set of peripheral processors each has access to the main memory and can cause interrupts in the main processor, although there is little communication in the reverse direction, and the several peripheral processors cannot pass information among themselves. The concept we wish to develop in the context of debugging is slightly different: Two machines, with overlapping but not identical capabilities, cooperate to solve problems, where the problems may originate on either machine, and the solution may be found on

either machine.

This concept is a generalization of that found in the XNET debugger [Beeler 76], which allows programs in a PDP-I 1 to be debugged across a network. XNET works with a skeletal debugger in the PDP-11 capable of handling a small set of examination and deposit operations (not in a symbolic fashion) and a sophisticated symbolic debugger in a larger remote machine. Thi two machines are linked by a communications protocol described in [Mader 74]. The concept mentioned above generalizes XNET-type interactions by allowing the debugging to proceed under direction from either machine, with symmetric question-asking potentials in the two computers.

Linking two machines has many delightful properties beyond the ability to compile and recompile. Firstly, it allows the debugging to take place from either machine. Information can be distributed in such a way that the information necessary for the response to some queries is immediately available in the *host* machine, that is, the one with which the programmer is directly communicating. At other tjmes, the host computer will need to request information from 'or perform actions on data' that is only available on the *remote* machine, that is, the one not currently discussing the knotty issues of bug control with the human guide. In these cases, the host will send a request to the remote machine in exactly the format that would be used by the human if he were working from that machine. In fact, there is no particular reason why several people cannot be simultaneously debugging from different portals, each with his own debugging process and his own host machine.

A second feature of' linking the compiler to the execution machine is that it provides a mechanism whereby the entire execution can be controlled.by a supervisory program residing on either machine. Instead of going through the standard stages of writing code, compiling it, loading, and then trying it all out, using a different program for each stop on. one machine or the other, a unified command structure can control the entire program preparation endeavor.

A third. happy' result of the link is that complex forms of feedback (for example, visual feedback) can be interfaced to running manipulator programs across this link. The running program can wait for a picture to be taken and processed on the large machine and the 'results of this exercise can be translated into new values to be deposited in variables in the running program. This interfacing of high-leuel or computationally expensive feedback is easily obtained by, using a program to emit the commands that the human usually would feed into the debugger. That same program would be in charge of controlling the television camera or other feedback device. If the debugging commands are the same as the source language statements, the debugger is also an ideal hook on which to hang strategist programs that reduce abstract task descriptions into AL programs.

Section 5
Side' Effects

AL is designed with two independent types of programmable side effect: condition monitors and affixment structures. The ability to associate side effects with situations that are suspected of leading to bugs is very useful in debugging, but the fact that hidden responses are taking place can be very confusing:

How can condition monitors be used to assist debugging? If it is suspected that some code is failing because the arm is never feeling a desired force, a condition monitor can be designed to stop the arm after some period of motion as if the force had been felt, and the program can be continued. In this case, a structure is used to simulate a desired result so that other errors can be found. The condition monitor can also be used to scan for the presence of a bad condition; for example, suppose that an erroneous signal is being emitted on an event somewhere and it is not possible to figure out who the culprit is. A condition monitor that waits for that signal can then immediately stop execution and let the programmer poke around and find out what is happening. Tracing evanescent situations is the most efficient debugging use of condition monitors. A monitor can be used to trace the forces on an arm, so that the programmer can figure out what a reasonable threshold might be for stopping the arm. Condition monitors are also capable of testing variables and complaining if the values are bad, but variable testing is the special forte of affixment structures. A special changer can be associated with a suspect variable; whenever its value is changed, the changer can either trace the current value or it can make a validity check and complain if the value.is bad.

Both of these programmable side effects find debugging use, therefore, in testing out suspicions and running traces during the execution of the program. In this sense they are very like output statements with which a programmer peppers his code in order to get some flavor of how the ingredients of his' program are interacting. In order to use this approach it is necessary to have some suspicions, to modify the program in such a way'that the suspicions can be tested, and to then try out the code anew. Interactive debugging usually does not follow such a tracing paradigm; the debugger itself is used to do the necessary tracing. If we allow the debugger the same power as the source language, there is no reason why a changer could not be patched into the graph structure for a variable, or a condition monitor spawned by the debugging process to perform the desired tracing. In this way, the power of the side effects can be brought under the control of the debugger.

Side effects, although easily brought into the house as pets, are not so easily domesticated. Those' instances wherechangers, calculators, and condition monitors influence the values of important variables or arm motions can be difficult to perceive. Mistakes in affixing frames

can cause particularly opaque results. All that can be observed is that the arm goes to the wrong place.' But' how the destination frame managed to get such a value may be a complete mystery. Condition, monitors cause a new flow of control to temporarily exist, and during that time, 'commands that 'move arms may be encountered. In fact, it is possible to program changers so that a side effect of assignment into a particular variable is to park an arm!

The debugger must help the programmer figure out the causes for all observable behavior. If the arm starts to move unexpectedly, the user must be able to halt it and find the source code that 'is is causing the motion. That means that the processes that implement condition monitors and changers must be subject to the same scrutiny as all other, garden-variety threads of control. In particular, these special processes have unusual states that the debugger should be able to influence. For example, a condition monitor has these states: inactive, active but waiting for the next checking time, busy checking, executing the conclusion, and uncreated (its block is not being executed). The debugger should be able to put a *hold* on the monitor so that it does not leave its present state, and then be able to force it into some other state; in this way, it is possible to test out the conclusion of a condition monitor without having to actually create the condition that normally triggers it.

CHAPTER 4

DESIGN OF A DEBUGGER

What's in a name? That which we call a rose
By any other name would smell as sweet.
Shakespeare, *Romeo and Juliet,* II:ii,43

This chapter describes some of the design of a debugger for AL. Following the pattern of RAID, AID, and BAIL, this debugger is termed *ALAID*, although other suggestions [Cerf, personal communication] include TRYAL, ADDALD, DEBAL, and even ALDEBERAN (AL DEBugging Execution Arm Environment).

ALAID is an attempt to meet the special needs of an arm code debugger. Its driving principles are these: 1) The link between the two computers allows a partitioning of planning and runtime information. 2) Debugging should proceed equally well from either machine; they should be symmetric as far as possible. 3) Debugging should be possible without the link insofar as the necessary information is available on the machine used. 4) Debugging should consist of symbolic examination and modification of data, program, and control flow. 5) The debugger should be usable as a top-level command structure for the system composed of the compiler, the runtime, and the debugging package. 6) Insofar as possible, all structures of AL code should be available for examination and modification under formalisms present in the source language.

The purpose of this chapter is to discuss the structures needed to implement such a debugger. This treatment is heavily based on the nature of the current AL implementation at use at Stanford, both its software and its hardware. The ideas, however, are generally applicable to any AL implementation and in part to any programming language implementation.

The state of ALAID at the moment is fairly primitive; it resides on the two machines and can start up the interpreter, examine and set arithmetic variables, signal/wait events, and cause the runtime system to enter 1 l-DDT. These initial facilities alone make the current ALAID implementation quite useful for testing out new runtime routines, AL programs, and high-level feedback that requires the PDP-10. Many of the concepts introduced here are therefore ideas for extensions to ALAID and design strategies for accomplishing their implementation.

Section 1
The Link Between Machines

ALAID is intended for the interactive debugging of a program that has been compiled on one machine and is being executed on another. (The XNET debugger [Beeler 76] also operates in a multi-machine environment.) The compilation phase not only transforms the input code into a form acceptable to the target machine, but also develops a planning model of the values of variables throughout the program. Furthermore, it creates a symbol table associating the printing names of variables with level-offset pairs. One might expect the compilation phase to emit some of this state information into the output code, so that the debugger can reside entirely on the target machine. However, since the compilation machine would be needed anyway if the programmer decides to rewrite a section of code, it seems reasonable that the compiler (or 'at least some of its tables) remain available during a debugging run in order to assist in associating names and planning values to variables and to help in making patches. In this way, the runtime environment does not become cluttered with too much information that is not directly needed during the execution of a program; such information can be found in the other machine.

In order to make use of several machines, it is necessary to have a form of communication between them. In this case of AL, the compiler process resides on the PDP-10, running under a timesharing multiuser system, and the target machine resides on the PDP-11, running under the kernel. The purpose of the *link* is to provide an efficient communication path between these machines so that each machine can appear to be both *a* debugging user and a source of information from the point of view of the other machine. This link has been implemented as described below.

1. PROTOCOL FOR THE LINK

The actual link implementation uses the hardware interface that connects our PDP-10 with the 'PDP-11. That interface allows either processor to generate interrupts on the other, and the PDP-10 can read and write the PDP-11 unibus. In this way, information in the memory df the PDP-11 is available to both processors.

In the following discussion, it is assumed that the interface is a foolproof channel; all communications in both directions reach their destinations. Much work has been done in computer networking to provide for noisy channels; most methods developed in such environments involve positive acknowledgement, retransmission, and sequence numbers. These techniques could be employed in the ALAID context as well if the channel were unreliable; since the present hardware is no less reliable than the processors themselves, problems of' inaccurate transmission have been ignored. The current link is also less powerful

than a full-fledged network-based communcations protocol; the two machines must be physically connected by the PDP-11 unibus. These restrictions are not fundamental to the idea of the two-machine link.

In order to avoid conflict in the allocation of this memory for communications, two fixed blocks of memory, called *noteboxes*, are reserved at all times. One is for the PDP- 10 to send little notes to the PDP-1 I, and the other is for the PDP-11 to send notes to the PDP-10. As a sign that the note has been received and to clear the notebox for further communication, the receiving processor overwrites the first word of the note, setting it to zero. The traffic in notes provides for agreements on the use of the larger memory for more substantial *messages*; all actual allocation is treated by the PDP-11, which honors requests for message space both from its own processes and from foreign notes.

There are very few note types required to maintain the link, since the allocation of message buffers is the prime activity at this level. Allocation is non-symmetrically handled by the PDP-11, so the two processors send different kinds of notes to each other. Each note is at most three words long; the noteboxes occupy very little space. The first word identifies the type of note, and the second two words provide room for arguments to the requests and responses.

2. *NOTES FROM THE PDP-IO TO THE PDP-II*

These are the note types that the PDP-10 can place in the PDP-1 I's notebox:

note type CETBUF **<s>**

Allocate *a* message buffer s bytes long. The expected response is the BUFALC note.

note type USEBUF <a>

The buffer that starts at address a is a message. Look at it, act on it, and then reclaim the message buffer.

note type RELBUF **<a>**

The PDP-11 sent the PDP-IO a message at address a. The PDP-10 has looked at it and is finished with it.

3. *NOTES FROM THE PDP-II TO THE PDP-JO*

These are the note types that the PDP-11 can place in the PDP-10's notebox:

note type BUFALC <s, a>

A **requested buffer** has been allocated for the PDP-10's use. It has size s (bytes) and is at address a.

note type TAKBUF <s>

The buffer that starts at' address a is a message for **the PDP-10,** which should look at it and act **on** it.

4. *MESSAGE BUFFERS*

Once the dickering between processors has established room for a message, the responsible processor fills the **given** area (known as a ***message buffer).*** There are several kinds of messages. The first is a request; which is either a query for information or a directive to be obeyed. The second message type is an ***answer,*** which either contains information queried by some other message or indicates that some directive has been carried out. A third type of message is the ***tidbit,*** **which** is information possibly (but not necessarily) interesting to the destination processor, and which may be ignored; it is never acknowledged.

Each message buffer **holds** the contents of the message along with a few header words that indicate the nature of the communication:

buf fer header MESID

This entry **is** the communication number of the message. Answers to requests have the same number, as the request; in that way, when a processor receives an answer it can use the communication number to determine which process within its domain made the request and is awaiting the response.

buff et header MESTYP

This field distinguishes whether the message **is** an answer, a query, or a tidbit, and whether it comes from the PDP-10 or from the PDP-I 1. When a message arrives at a processor, this information is used to decide what to do with the message.

buff& header MESLTH

The length in bytes of the message.

5. A SAMPLE DIALOG BETWEEN MACHINES

Suppose that the user has asked the PDP-10 for the value of 'some variable. The communication between the machines might look like this:

```
GETBUF 50                    from PDP-10
BUFALC 50.20436              from POP-11
<message at 20436: type request. SHOW VALUE .SNAME var1>
USEBUF 20436                 from POP-10
<PDP-11 gets value, makes an answer at 22312: type answer. SCALAR 3.0>
TAKBUF 22312                 from POP-11
<PDP-10 gets answer, reports It to user>
RELBUF 22312                 from PDP-10
```

6. ROUTINES RESIDENT ON THE TWO MACHINES

Even though symmetry of form between the two processors is 'desired, each has a different regime in force that constrains the implementation. The **runtime** system is under the tutelage of the kernel, which has control over the various processes and scheduling. The part of **ALAID** that resides on the **PDP-10** must be compatible with the compiler, so it has been written in SAIL (the language in which the compiler is written) using the SAIL Process mechanisms.

The most primitive routines on each processor are those capable of receiving and sending notes. The implementation could make use of the interrupts that each machine can generate on the other, but at present this is not done; the receiver on the PDP-11 sleeps for a short time between checks of the **notebox,** and the receiver on the PDP-10 is a SAIL process that is explicitly called when communication is expected. When the receiver gets a note, it makes a copy and zeroes out the first word of the **notebox** as a sign that the note has been seen. When the sender 'is asked to transmit a note, it waits until the proper **notebox** is free (its first word is zero) and then dumps the note in place, putting the first word in last.

These routines are under the **control** of a process' known as the server, which perpetually loops,. calling the receiver to get a note and then deciding how to treat that note on the basis of its type. For example, the server on the PDP-11 must interpret RELBUF, GETBUF, and USEBUF. The first involves only the free storage allocator; the second also **must** call the sender to inform the PDP-10 that a buffer has been allocated. The third note, USEBUF, implies that a message has arrived, in which case the routine ***treatmessage*** is called to handle it.

Treating a message involves different actions for different kinds of messages. If the message is a request, a new process is started to **handle** it and eventually to return an answer. If the message is an answer, then it is made available to whatever process sent the corresponding request. The correspondence between process and request message number is kept in a list

that is modified whenever a request is sent or an answer is received across the link. If the message is a tidbit, then its contents are directed to a tidbit handler. The usual response to a tidbit is to print it for the user to see. (A process that has encountered a breakpoint makes its plight known by passing tidbits.)

Just as there is only one server on each machine, there is likewise only one requester. Sending a rquest involves adding the name of the requesting process to the waiting list and sending notes across the link to agree on the transfer of the message. The process that asked for the transfer is suspended until an answer arrives.

7. FLOW OF INFORMATION

To demonstrate the manner in which information is passed among the various pieces of **ALAID,** consider the user request to place a breakpoint at a particular point in the code. Assume that the user is communicating directly with the PDP-IO, and that he uses the name of a program label to identify the spot. The request looks like this:

```
BREAK SADDRESS L1
```

Now the PDP-10 cannot itself insert breakpoints, so it passes the entire request to the PDP-11. The PDP-11 cannot interpret the label, since it has no symbol table. Therefore it must ask the PDP-10 to identify the location in its code:

```
CONVERT PADDRESS "SADDRESS L1"
```

The appropriate symbol table resides in the PDP-10, so it takes this request and finds that **L1** is at location 132012. It then responds to the PDP-11:

```
PADDRESS 1320 12
```

Now that the label has been resolved, the PDP-11 proceeds to place the breakpoint. Having finished its task, it responds to the PDP-IO:

```
DONE
```

Some tasks rquire more communication than this simple example demonstrates. If the user wishes to assign a value to a variable, then the variable must be sought in a symbol table, the value must be evaluated (possibly involving compiling code and running it, or else by repeated requests for the values of variables), and finally the assignment can be made.

These examples point out the various *portals* to which each **ALAID** member must respond. One portal is the user: In the example above, this portal is In use on the PDP-10 member only. Each of the two members in the example has a portal devoted to the other member. In addition, **ALAID** can be used as an interface into AL from another program; in this case, a portal is 'devoted to that program. One example is when the executing AL program encounters the breakpoint set in the scenario above. It then informs the PDP-11 member of **ALAID** through its own portal; **ALAID** then sends a tidbit to the other member.

More than one portal can be in use simultaneously, as we have seen; the PDP-IO member is active on two portals during the breakpoint-setting example. In fact, the program portal can be thought of as a potentially infinite set, with one portal for each process currently active.

This would be the case when more than one process has hit a breakpoint.

Given the plurality of portals, effort must be exerted to direct information to the right place. Queries, are signed by the originator, and responses carry the same signature, so it is straightforward to direct the answers back to the proper portal. A harder question is to decide where to send a query that cannot be answered locally. In the case of a two-member **ALAID,** the usual answer is to send the query to the other member. If the query itself can be answered, but to complete the answer requires some information that is not available, then a query can be formed to get that information from the other member. If that query fails, then the user can be asked; he is also a member of the **ALAID** community, although perhaps not in such good standing. In that case, care must be. taken to distinguish between the user's answers and further queries that he might make. In this way, portals can be characterized by their ability to make and respond to queries. The portal that connects **ALAID** with a program is usually useful only for transmitting queries into **ALAID** and answers back out. The user is also usually a questioner, but in some situations he can be asked queries directly. The other members have both properties of originating and responding to requests. For the sake of completeness, one can imagine a portal connected only to symbol tables; such a portal can be queried but will never generate a request.

Given the multiplicity of ways to direct queries that cannot be directly answered, how can **ALAID** know that a query cannot be answered at all? A simple approach for two-member **ALAID** is to try only the other member, and if he cannot help, then the request is unanswerable. If the impossible part is only .a subset of the whole request, it can be useful to report back that this particular part was the bottleneck. Otherwise, a simple failure return suffices. More complicated situations' ensue if there is more than one portal to, which the request can be directed, or if there are several ways to subdivide the query into easier questions. Then the process of finding an answer. has the appearance of a depth-first tree search; every node represents a subquery, and has a set of alternative strategies, each of which leads to a collection of other, nodes that must be successfully treated for that strategy to work.

In order to prevent the search from becoming circular, each query must contain some history that tells which members of **ALAID** have tried to answer the question and have failed. The initial history list indicates the originator of the query. If a member ever sees a query that he has seen before, he immediately returns failure. In fact, queries should never be sent to any member already on the history list. If a query generates a **subquery** that is somehow equivalent, then circularity is still possible, because the **subquery** does not share the history list of its parent, but can perhaps spawn another **subquery** that has exactly the same form as the original one. The only way to avoid this problem is to disallow equivalent subqueries or to give them the same history list as the original queries.

Some frequently used information might be duplicated in several members. In this case, any one that receives a query. that requires, that information can service it. As long as the information is static, there is no danger that inconsistencies will arise between the several. repositories. If, on the other hand, the information is subject to fluctuation (for example, the

current context), then some method must be developed to keep the various versions as consistent as possible. Tidbits can be sent to all interested parties to inform them of a change in their data base; the issues reside in who should initiate these tidbits and to whom they should be sent.

If every piece of shared data has an owner, then only that owner should be allowed to make a change to the data, and when he does, a tidbit should be sent to the other members to inform them of the change. That means that a request whose effect is to change information that has duplicate copies must be directed to the member that owns that information; queries that only investigate the information can be serviced from any of the sharing members. An alternate technique is to allow any of the sharing members to make modifications, but to force them to send tidbits whenever so doing. The disadvantage of this strategy is that it is not possible to be sure of getting the most recent value of such information, because one of the members may have changed it without yet informing the rest of the community. In the first approach, a good value can always be generated by asking the owner specifically.

Tidbits advising that multiply copied information must be updated could be broadcast to the entire community of members, or each type of duplicated information could include pointers linking it to the various members who have copies.

8. GENERALIZATIONS OF THE LINK

The idea, of connecting two processors together to share in the work of debugging has some obvious generalizations. The link that has been described is designed to share information structures in a domain that naturally divides labor between two processors. Cases in which more than two processors are in use are becoming more frequent; the ARPA network of computers presents an environment in which cooperation among many machines may be used in the execution of a single algorithm. Furthermore, the decreasing cost of processors seems to be creating a trend to divide complex algorithms among several, perhaps many, machines. The natural question is how well the two-machine link can be extended to treat many machines.

The two issues raised in the previous section, directionality and circularity, pose the major problems to extending ALA ID to more members. The circularity issue becomes worse with many processors, since one question can generate several subordinate questions, each of which can be directed to a different machine.

The directionality issue is the more severe. Each member could have a list associating types of queries and members that are capable of responding to them. If a query is not found on that list, then it might be sent to each of the members in turn under the hope that the current recognition list is out of date. If a member returns failure from a query, it must also indicate what other members have been asked to look at the same query; otherwise, there could be wasted effort involved in sending the same query back to members that have already seen it. A spanning tree that contains all the members could be used to direct queries; it would be

illegal to pass a query along any link not in the tree or back across the link by which it arrived. In this way, it is also possible to reduce the interconnectedness of the entire graph of members.

It may be unreasonable that each processor should be able to recognize all the possible questions. In this case, a set of *clearinghouse* processors can keep track of who will answer a given query. So the first action on receiving a query that cannot be immediately answered is to query the clearinghouse for the name of the members to which to direct the original query. Thus a legal **datatype** of the response language includes member names.

What does it mean to have many processors running on one algorithm? In the case of AL, one is an execution processor, the other is a planning processor. The most general case is to have many execution and many planning processors. In AL, the two processors have very restricted normal communication. The general case may include many channels of communication among the execution machines. This communication could be handled exclusively through **ALAID** on each machine through the program portal. In this way, the communication link that is already present would be put to good use.

How to connect the various members together is a well-studied problem. **ALAID** takes some advantage of the interface between **PDP-10** and PDP-11. The requirement for many-way connection is that any machine must be capable of getting messages to any other machine. One. common memory (with one non-reentrant process for parceling out memory) will work, but short of that, especially for more than 10 or so processors (at which point such linking may get too expensive) a **new** kind of message could be used, the *transit* message. It is not intended for the recipient, but should be handed on. At this point we are getting into networking, not debugging issues. As long as any processor can get, a message to any other one in such a way that the sender is identified, it suffices.

One generalization of treating each processor as an indivisible entity, is to treat each process as an entity. The **ALAID** member on each process could be shared among all the processes within a single processor, giving rise to a two-level hierarchy of **ALAID** communication. Once a hierarchy of processes is introduced, it can be used as a general ordering technique for the entire cooperative computation, with higher level processes charged with parcelling out tasks to inferior ones and directing communcations between those lower processes and the outside world. Using **ALAID-based** communication might be fruitful in this domain; much further research is warranted to investigate these issues.

Another realm for further work is the dynamic redistribution of information. Space constraints may force one Processor to relieve itself of some information of secondary importance by giving it to another processor to store. Frequent references to some data may imply that thosedata should be made-more accessible by copying them or moving them to the requesting processor. To identify these situations, set up new processors with **ALAID** members, and transmit the information is a problem of some difficulty whose solution may prove quite powerful.

Section 2
Symbol Tables

As we have seen, debugging is an activity that requires repeated examination and modification of a test program. The link that has been described in such detail is fundamental to making information available to the various parts of **ALAID**. In particular, symbol tables residing on the PDP-10 provide correspondence between symbolic entities in the source code and physical entities in the object code. A symbol table can be pictured as a memory of bindings kept so that the decisions made in binding can be quickly simulated without being explicitly recomputed. In this way, all the information that goes into the decisions can be discarded; the result itself is distilled for future reference. Three examples of symbol tables will be discussed here: variables, processes, and code. Each of these symbol tables must work in both directions: the PDP-11 representation must be resolved into PDP- 10 representation and vice-versa.

1. VARIABLES

The **runtime** representation of variables is based on their lexical level within the source program and the order of declaration within that level. A new lexical level is started for each COBECIN and PROCEDURE (although procedures are not fully implemented); a slightly different design might have counted each BEGIN in the lexical level count. , Thus each variable is identified (although not uniquely) by level and ***offset***. Typical values for the level are 0, 1, and 2, and typical offsets are even octal numbers from 10 to 100. In the **runtime** system, **these two** quantities are combined into one **16-bit** word with the level in the left 8 bits and **the** offset in the right 8 bits. In order to make a level-offset pair uniquely refer to a variable, it is necessary to know which of several parallel blocks, that is, which context, contains it uniquely. For example, consider this piece of code:

```
BEGIN (level 0) (name 1)
  SCALAR S1; (offset 10)
    COBEGIN
      BEGIN (level 1) (name 2)
        SCALAR S2; (offset 10)
        SCALAR S3; (offset 12)
        END;

      BEGIN (level one) (name 3)
        SCALAR S4; (offset' 10)
        END
      END
  SCALAR S5; (offset 12)
    END
```

and S4, because they are each the first variable in the first level. During execution, there is no possible conflict, because two different interpreters are active; the one that has access to **S2** cannot see S4, and vice-versa. Therefore a third datum is necessary to distinguish variables: the *name* of **the** interpreter, or, equivalently, a context in which the variable is unambiguous. Variable S2 is fully described by the triple **(2,1,10),** which gives the name, the level, and the offset; variable **S3** is **(3,1,10).** During execution, the name is always implicit, and no code need be generated. However, if **ALAID** wishes to access a variable, it must specify the name as well, either explicitly or implicitly. The current context gives a partial implicit specification; if it contains only one interpreter, no interpreter name is necessary. If there are several interpreters, then those variables to which each has access need no interpreter names; the others do.

The symbol table for variables, which is used to make correspondences in both directions, is structured according to interpreters. This structure implicitly includes the interpreter name in each entry. The individual entries include source code name, internal compiler name (which is different), and the level-offset pair. Search in the table for **level-offset** pairs is conducted first in the top-level interpreter of the current context, and then, if necessary, in each of the next lower-level interpreters. The result of the search is either a level-offset if that will suffice, or a name-level-offset if necessary, or an error condition: found more **than** once, in which case the context is not sufficiently specific. If the variable is not found at the current interpreter, then surrounding contexts are searched until the variable is found. To find the source- name for a given level-bffset pair, if the level is deeper. than the current context, then **there** is no **unique** solution.' Otherwise, it is fairly easy to find the source name.

Once a name-level-offset triple has been found, to find the value of that variable on the PDP-**11** requires that all interpreters be accessible by name. This is accomplished by keeping a linked list of all interpreters (a tree structure would be more efficient if there are many) and providing a special-purpose pseudo-op **INAME** that causes the interpreter that executes it to **assume** a new name.

2. *PROCESSES*

The previous discussion shows that it is necessary to identify processes in order to properly access variables. Each process is given a unique name by the compiler. An obvious extension is to allow the user to assign process names himself; this will allow greater ease in setting the context during a debugging session. A simple pairing of user-given names and compiler-generated names suffices. (The actual implementation might use hash coding, although the total number of processes is likely to be small enough so that even linear tables are adequately efficient.)

3. *CODE*

The organization of code on the two machines is quite different. The basic unit in the source language is the statement, which is compiled into a stream of pseudo-operations. There are four different naming techniques that can be partially interconverted: source code, source labels, pseudo-code, and addresses on the runtime machine. Let us restrict our attention to the problem of determining the current statement in the source language given the. runtime address.

If the entire source program is kept in the PDP-IO memory (this is the case for the current implementation), then markers can be emitted along with the code that refer back to statement names that the compiler understands. In this way, the pseudo-code can be made. self-descriptive, and to find the source code from an address on the PDP-11, one need only go back in the pseudo-code until a, marker is 'encountered. The price for this method is the space occupied in the pseudo-code for the marks, which might amount to about 25 percent of the total code, not counting trajectory files and constants. (If these are also taken into account, then the expetise of the marks is only about five percent.)

A related technique is for the compiler to emit a separate symbol table that coordinates pseudo-code addresses with source language statements. Such a table would be searched by a binary chop method. About the same space requirements would be necessary in this case; the advantage is that the symbol table is separated from the objects it is describing, and it can therefore be moved to the other machine. The trouble with this method is that during compilation, no information is maintained about the location where the code will be placed, and, furthermore, it is hoped that AL will soon become capable of compiling relocatable modules. The implication of this observation is that the symbol table must be manipulated by the loader to convert relocatable addresses into actual addresses.

The best solution is to combine these two approaches. The relocatable output should contain marks that relate the code to source statements, and the loader should remove these marks, constructing a symbol table in the process. In this way, the binding of the symbol table takes place at the time that all necessary information is available, and at that point extraneous information (the marks) can be discarded. This solution also lends itself to the problem of

finding the pseudo-code location of a given source-language statement.

Section 3
Control Over AL

Various programs must be applied in order to achieve execution of an AL program. One of the purposes of **ALAID** is to control the compilation, loading, and execution process so that a unified, face is presented to the user. The ideas in this section lay a groundwork for such a facility; these concepts have not been implemented in the current version of **ALAID.**

The primary unit of compilation is the *module.* It is-one statement long, and is self-contained. In general, the statement is a substantial program, but it can be very short as well. To refer to variables that are not in that module, a ***global*** **declaration** is given. The planning values for all global variables starts as "undefined", so assertions are necessary before these variables can be used. The output of a compilation is a load module that has symbol table, linking, and planning model information. in the form of decorated parse trees.

A linking loader is invoked to take this load module and insert it into the current **runtime** system. This loader is one of the resident programs on the PDP-10; symbol tables on the PDP-IO are referenced and modified during the loading process. Direct memory access on the* PDP- 11 is used to actually put the program in place.

One useful concept is ***unloading,*** which takes the **current** set of modules on the PDP-11 and packages them into a single load module for future reference. In this way programs that are constructed piecemeal can be combined together into larger modules. The source code for the various modules currently resident on the PDP-10 might be in part available on the PDP-10; a similar process to unloading creates a source file that combines the various modules together into one program.

Together, these facilities allow programming by experimentation. Routines are written and tested until they seem to work, and then they are embedded. in larger drivers. A legal statement in the source language would be "MODULE <name>" that refers to a previously compiled module. The compiler could either read the source code back in and compile it again,-at some cost of duplication of effort, or recover the decorated parse trees from the compiled file.

The modules currently' loaded in the **PDP-11** are therefore a dynamic set; new ones can be added (patched in), and old ones can be removed. A simple symbol table keeps track **of** where each module begins in core, **where** it ends, and where it is referenced (which should only be in one place, since procedures are not yet available). To remove a module, its

physical space is reclaimed, and' the place where it is referenced is patched to give an error should it ever be called.

While programs are being written in this experimental mode, it is useful to be able to manually move the arm, read the position with ALAID, and use the frame value as a constant in the program. A' simple facility that allows the result of a previous query to be embedded in a new query will allow the arm position to be embedded in the program under construction. Each snippet of program that the user constructs is remembered as a module, and together the modules can be assembled into a'working program, then stored for future reference.

CHAPTER 5

COMMANDS FOR DEBUGGING

This chapter demonstrates a tentative subset of the commands to be available in a full version of **ALAID**. Some of these commands have been implemented in the first preliminary version; others are proposed.

In his' work on **COPILOT** [Swinehart 74], Daniel Swinehart gave great attention to the use of East video displays for showing the state of a multi-process job. In addition to standard debugging and control commands, he includes -a set of display-oriented commands to distribute the limited screen among the various data that could be shown and to point to objects of interest by moving cursors: The display orientation of COPILOT could form a useful base for **ALAID**, and the commands listed in this chapter would be enriched by the addition of display features. It is likely that such facilities would be available only on the **ALAID** member that resides on the larger machine, since space is at a premium on the small machine. Therefore, rapid redisplay of changing status may not be possible in the **ALAID** environment, but even occasional redisplay would be useful.

The commands are divided into functional groups by the entities they deal with: internal state of **ALAID**, data structures, control structures, control flow, and advanced commands. Each group has three sections. First, the set of relevant **typeout** modes is introduced. These modes dictate which of several equivalent forms output is to take. Next, commands for examination are listed, each with a brief description. Last, commands for modification are listed, again with descriptions.

1. TYPEIN MODES

Many of the commands require specification of variables or code. For example, in order to ask for the current value of a variable, one needs to name that variable. In general, there are several alternate formalisms. Some can be immediately recognized in the PDP-1 1, and others require the symbol tables that reside in the PDP-10. Whenever alternatives exist, the user should preface his **typein** with an identifying word that indicates what type he is using. (This is a simple way of restricting the input syntax to avoid complex type determination. It is not necessary to the ideas behind the debugger.) In the following discussion, **typein** modes will be introduced as needed. As an example, to refer to the variable "creativity" by its source-language name, one **would** type

```
SNAHE creativity
```

Section 1
Internal State of ALAID

The internal state of **ALAID** consists of the current context and a set of ***modes*** with which various data are printed. The context is a thread of execution, possibly containing other threads within it as subprocesses. Contexts are used to disambiguate the meaning of variable names and to select processes for interaction. **Typeout** modes dictate the format in which variables and code are displayed. Commands that affect execution (like halting and jumping commands) influence all active processes in the current context; therefore one should be careful to distinguish contexts and threads.

This section will discuss the way modes are set; the appropriate **typeout** modes will be discussed in the sections in which they arise. Once a mode has been set it is permanent until reset. (Most versions of DDT have both permanent and temporary modes. RAID associates a mode for every one of the twenty or so variables that can be concurrently displayed.) Every time a query is answered in some mode, the name of the mode prefaces the result. The purpose of this is to make all output self-identifying, so that the result of one query can be used as the input to the next. For example, the result of a query for the current locus of control might be:

```
PADDRESS 132024,
```

or it might be

```
SCODE "HOVE BARM TO BPARK VI A BP1"
```

1. TYPEOVT MODES.

Contexts can be displayed by the code that starts up the thread of execution (CONTEXT-BY-CODE mode). That code can be named by location or 'by contents. Locations in the control store can be referred to either by octal location in the PDP-11 . (PADDRESS mode) or by labels and offsets in the source code (SADDRESS mode). The contents of the control store can be shown either as pseudo-instructions (PCODE mode) or by the source code that generated them (SCODE mode). **Each process** has a compiler-generated 'identifier. The identifier **associated with** the **top-level** thread of a context can be used to identify the context (CONTEXT-BY-IDENTIFIER mode).

2. EXAMINATION

SHOW CONTEXT

The current context is displayed. For example:

```
CONTEXT-BY-CODE  SADDRESS  LAB3
```

SHOW MODES

Each of the current **modes in** effect is listed. The **only typeout** mode in which this list can be printed **is** LIST mode. For example:

```
LIST
        CONTEXT- BY IDENTIFIER
        SADDRESS
```

3. MODIFICATION

SET CONTEXT <thread name> .

The thread can be currently in execution or not. If not, then no information will be available for variables local to that thread. The named thread can include many subthreads; **only** those variables in active subthreads may **be** accessed. The **name** of a thread can be given by the same modes used for showing the context.

MOVE **CONTEXT <list of codes>**

The context is to **be** changed from the current thread. One legal code is "UP **n**", where n is a positive integer. This code moves the context to the surrounding thread n levels more global. Another code is "ACROSS n", where n is any integer. The context is to be moved to ,a sibling thread, either forwards **(n>0)** or backwards **(n<0).** The last code is "DOWN **n", which moves down one** level only, to the nth daughter thread. An abbreviation for "DOWN n DOWN **m . .." is** "DOWN n, m, . ..".

SET MODE <mode specifier>

The **typeout** mode is set to the one given in the command.

Section **2**

Data Structures

1. TYPEOVT MODES

All arithmetic quantities are displayed according to their type, which is built into the **runtime** data structure. That is, vectors will always be typed as three numbers. However, there is some flexibility in typing rotations (and therefore frames and transforms, which have rotation components). One mode (ROT mode) reduces the rotation to one swivel about one axis, and reports the'rotation the **same** way the source language accepts them:

```
ROT( vector ,angle)
```

The other **mode** (EULER mode) reduces the rotation to up to three rotations about cardinal axes. This mode is far easier for the human to understand.

Non-arithmetic quantities include expressions and events. Expressions are printed as code; the relevant modes are PADDRESS, PCODE, SADDRESS, and SCODE, as described above.

Variables can be named as they are called in the source language (SNAME mode) or as they are translated for the pseudo code (PNAME mode).

2. EXAMINATION

SHOW VALUE <variable name>

The variable must be available in the current context. The name can either be in SNAME or in PNAME modes:,

```
SHOW VALUE SNAME brrm
SHOW VALUE PNAME 32
```

EVALUATE <expression>

The expression, which, is given in the source language (SCODE mode) is evaluated in the current context, and the value is returned. With this command ALAID has the full power of the source language to investigate data structures.

3. MODIFICATION

SET VALUE <variable name> <expression>

The variable name is given as for SHOW VALUE. The expression can be an expression variable (in SNAME or PNAME modes) or a source-language expression (in SCODE mode). The facility for executing source language statements (to be discussed in detail below) can also be used to set values:

```
EXECUTE SCODE "<variable> ← <expression>"
```

Section 3
Control Structures

1. EXAMINATION

SHOW CODE **<address>**

The address can be in SADDRESS or PADDRESS format; the code that is displayed will be in SCODE or in PCODE depending on the current typeout mode. Thus the SCODE corresponding to a PADDRESS can be displayed. If the PCODE is in the middle of a single SCODE statement, the SCODE displayed will be annotated *in progress.*

2. MODIFICATION

SET CODE <address> <code>

The given code (in SCODE or PCODE mode) is placed at the given address (in **SADDRESS** or **PADDRESS** mode). There is no space problem if both the address and the code are in P modes; other combinations cause difficulties. **SADDRESS** and PCODE is usually foolish; it replaces the entire code for the statement with a'single PCODE and a jump to the next SCODE entry. **PADDRESS** and SCODE is interpreted to mean that the SCODE at that **PADDRESS** is to be changed from the beginning, even thought the **PADDRESS** may 'be in the middle. **SADDRESS** and SCODE is hard because the new code might not fit in the old location. The newly compiled code is therefore placed in a fresh location, and appropriate jump instructions are inserted to patch it in.

Section 4
Control Flow

1. *TYPEOUT* MODES

Locations in the control' store can be referred to either by octal location in the PDP-11 (**PADDRESS** mode) or by labels and offsets in the source code (SADDRESS mode). The contents of the control store can be shown either as pseudo-instructions (PCODE mode) or by the source code that generated them (SCODE 'mode). (An extension to this facility would be tb' allow MC'ODE 'and **MADDRESS** to refer. to machine instructions.). If the location is a pseudo-instruction in the middle of several that all accomplish the same statement, then 'the **SADDRESS** and **SCODE** outputs will be annotated in *progress.*

2. EXAMINATION

SHOW EXECUTION

All interpreters in the given context are listed, with the name of the statement currently in execution. The statement is listed in pseudo-code and its address is given.

3. MODIFICATION

BREAK **<address>**

A breakpoint is inserted at the given address. When execution encounters this point, a message will be sent to the user and control will pause until the user allows the program to continue. The breakpoint influences only that process that hits it; all others that are active will continue. The message that the affected process transmits to the user includes a context specification that uniquely defines which process it is; in this way, the user can set

the context appropriately before issuing investigatory commands.

SINGLESTEP

Once a breakpoint has been encountered, the user often wishes to execute a small piece of code and observe its effect. The single step command allows a halted process to continue a short distance and then once again pause. The process that this command affects is the one pointed to by the current context; if that context includes several active processes, then the command applies to all of them. This exemplifies the convention that ALAID uses with regard to contexts: all commands given affect all processes in the current context. It is always possibk to restrict the context to contain only one process. If the. single step command is given to a process that is not in a halted state, then the process will be halted. The amount that an affected process will execute when singly stepping depends on the current code typeout mode: if the mode is SADDRESS or SCODE, then one statement of the source language is executed; if the mode is PADDRESS or PCODE, then one statement of the pseudo-code is executed. After the single step command has been executed, each affected process will send the user a message that identifies the process and where it is executing.

PROCEED

Any halted process in the current context is allowed to continue execution. Once the user is satisfied that the program is behaving properly in the region of a breakpoint, this command is useful for proceeding to the next breakpoint.

HALT

All processes within the current context are halted. As they stop, they send the user a message telling where they are. This command will not stop' a moving arm, since the process controlling the arm is in the middle of a single pseudo-operation.

JUMP <address> .

All processes in the current context stop executing at their current location and start executing at the given address. If a block must be exited or entered before this jump can be done, then the block exit/entry code is executed appropriately. Since this command is dangerous, it is not honored if the current context contains more than one active process.

EXECUTE **<instruction>**

All processes in the current context execute the given instruction. It is not necessary that the processes be halted first; as soon as they are finished with the current instruction, they perform the given instruction 'and then proceed with whatever they were doing. Of course, it is most usual to give this command to a stopped process. The instruction can be in SCODE or PCODE modes. This facility is quite powerful; any operation available to the AL language can be performed in this way. For example, to set the value of X to VECTOR(1,1,3), one would tell ALAID

EXECUTE SCODE "X ← VECTOR(1,1,3)".

SIGNAL <event **variable**>
, WAIT <event **variable**>

These commands are not strictly speaking either control or data modification commands, but have some of the flavor of both. Their intent is to allow processes to proceed from event waits by explicitly supplying the signal and to deactivate the **ALAID** portal **until the** program supplies a signal. **These** facilities allow feedback routines residing on the PDP-10 to communicate with the program on the PDP-I 1: When the AL program wishes feedback, it signals a particular "need feedback" event, and waits for **the** "feedback ready' event. The cooperating routine waits for the "need feedback" event, computes the desired quantities and feeds **them into** the program by means of **ALAID commands, and then'** signals the "feedback ready" event, thereby allowing the program to proceed.

Section 5
Advanced Coin **mands**

This section describes some miscellaneous powerful features of **ALAID** that do not easily fit into the pattern used to describe the other commands.

1. CONVERSION

CONVERT <new mode> <string>

Direct conversion of **typeout** modes is possible by means of this command. The **string** should **be** prefaced **with** the mode that it carries. Not only does this facility allow direct symbol-table **lookup,** but it also allows temporary modes to override the permanent modes. For **example, if** the **current mode is PCODE,** then

CONVERT **SCODE** 'SHOW PCODE **PADDRESS** 132024"

is equivalent to

SHOW **SCODE PADDRESS** 132024

ALAID will respond with the source language representation of the code at location 132024. This example also **demonstrates** the, **use** of ***embedding,*** which allows one **ALAID query** to be embedded in another one. First the innermost query **is** serviced, **and the result** of that query is treated as the argument **to** the next. The conversion facility is used by the PDP- I **1** to translate **modes** that it does not understand.

2. SUPERVISION

COMPILE **<logical name>** <source>
LOAD <logical name>
START <logical name>
DUMP **<logical name>**

GET <logical name>

INITIALIZE

These commands are intended to place ALAID as the sole supervisory program over the entire AL system. Each program module can be given a *logical name* by the user, for example "SAMPLE". (LNAME mode applies.) The source might be a file on the PDP-IO (FILE mode) or a literal statement (SCODE mode). The result of compilation is a file with name "SAMPLE.PSC"; this file can be loaded by the.command

LOAD LNAME SAMPLE

Before the loading can take place, the AL runtime environment must be available on the PDP-11. If it is not, then the INITIALIZE command will provide that environment. This command also can be used to flush any old A'L programs that might still be cluttering the execution system. The LOAD command will load the named module after whatever modules are already loaded, so that several modules can be linked together. Part of the LOAD command is to make the PDP-10 environment aware of the necessary symbol tabks.

The pair GET and DUMP are used to save an entire state of the world. DUMP creates the file "SAMPLE.ALD", which contains the entire *core* image of the PDP-11, the information necessary to continue it, 'and pointers to the necessary symbol tables in the PDP-10. GET reverses this operation. In this way, safe points can be created during the debugging of the program. After a GET command has restored the state, it is wise to issue the command .

EXECUTE SCODE "MOVE BARM TO BARH WITH DURATION . 5"

which will have the effect-of slowly moving the arm from whatever position it happens to' have back to the place that it occupied when the DUMP was performed.

3 . HISTORY

Messages pass through portals in both directions. Those portals that connect to humans contain the most important information from the user point of view; therefore it is natural to keep track of that information so that it is easy to recover. If ALAID has responded to a query, it is likely that the user will wish to 'use that response as part of his next query. Therefore two history lists are kept for each portal that leads to a user: queries he has issued and the responses that have been engendered by them. A legal field in any query is a reference to a 'previous query or response; these references are BACKQ and BACKR, which take numeric arguments. Therefore, if the user examines the value of a frame and then decides to add *a* small vector to it, the dialog may look like this:

```
user:   SHOW VALUE SNAME DEST
alaid:  FRAME(ROT(XHAT,180*DEG),VECTOR(1,2,3))
user:   SET VALUE SNAME DEST SCODE "BACKR(1) + VECTOR(0,0,1)"
```

For this reason, BACKQ and BACKR are not allowed as variables in the AL language.

4. THE AL PORTAL

AL programs can talk to ALAID in the same manner as the user. The AL statement

```
ALAID(<string>)
```

sends the string to ALAID and waits for a response; the string that contains the response is the value of the call to ALAID. In this way, a program can itself make use of the state-saving features of ALAID, and it can call in a new load module.

5. ABBREVIATIONS

Certain frequently used commands have standard abbreviations. For example, to look at a long set of consecutive pseudo-code instructions, it is awkward to repeat

```
SHOW CODE PADDRESS 132004
SHOW COOE PADDRESS 132006
SHOW COOE PADDRESS 132010
```

Instead, the simple command <linefeed> suffices after the first location has been opened. Also <control-P> can be used for the PROCEED command. Other abbreviations can be introduced as needed.

CHAPTER 6

CONCLUSIONS

In this report we have seen an approach to debugging that extends to control of the entire programming process. During debugging, the compiler is available, so that all code and data structures can be examined as they appear in the source language, and modifications can be made in the source language formalism. Modules of acceptable code are joined together into larger modules, and eventually a working program is prepared, all under the unified control of the debugger. In order to increase the investigatory power of the debugger, as many data structures as possible are available to the scrutiny of the source-language program, and the debugger has access to all the constructs of the source language.

Structures necessary to the implementation of such a debugger include special-purpose symbol tables to keep track of the correspondences between the source code and the object code, multi-machine links, and debugging processes that act in parallel to the processes they are manipulating.

The entire programming system that uses ALAID to cement it together and to act as a supervisory program can be extended to include computationally expensive sensory feedback by direct communication between the feedback processes and the running programs through the ALAID links. In addition, this unified structure allows simple programs to be written in 'AL that mimic several different modes of manipulator programming,,. from tape-recorder mode (in which positions are manually set and a program is written to repeat those positions) to completely textual modes (in which all positions and uses of feedback are specified in the source language). The unification of the various stages of AL compilation and execution also provides a groundwork on which to base saving and restoring contexts from one stage of debugging to another, and by the same token, allows backing up to a previous state in any production run. Real-world problems that mitigate against reversability are not solved by ALAID, but internal structures can be reset and then queried so that the user has some assistance in repairing the state of the world to what is expected.

BIBLIOGRAPHY

[Beeler 76] M. D. Beeler, *XNET, A Cross-net Debugger for TENEX: User's Manual,* Updated by R. S. Tomlinson, BBN report to be published, May 1976.

[Binford 75] T. 0. Binford, D. D. Grossman, E. Miyamoto, R. Finkel, B. E. Shimano, R. H. Taylor, R. C. Bolles, M. D. Roderick, M. S. Mujtaba, T. A. Cafford, *Exploratory Study of Computer Integrated Assembly Systems,* Prepared for the National Science Foundation. Stanford Artificial Intelligence Laboratory Progress Report covering September 1974 to November 1975.

[Mader 74] Eric Mader,' *Network Debugging Protocol,* Request For Comments #643, Bolt Beranek and Newman division 52, 50 Moulton St., Cambridge, Mass. 02138, July 1974.

[Reiser 75] John R. Reiser, *BAIL -- A Debugger for SAIL,* Stanford Artificial Intelligence Project Memo 270, Stanford Computer Science Report STAN-CS-75-523, October 1975.

[Satterthwaite 75] Edwin Hallowell Satterthwaite, Jr., *Source* Language *Debugging Tools,* PhD Thesis, Computer Science Department, Stanford University, May 1975.

[Swinehart 74] Daniel C. Swinehart, *COPILOT: A Multiple Process Approach to Interactive Programming Systems,* Stanford Artificial Intelligence Project Memo 230, Stanford Computer Science Report STAN-CS-74-412, PhD Thesis, Computer Science Department, Stanford University, August 1974.

[Teitelman 74] Warren Teitelman, *Interlisp Reference Manual,* Xerox Palo Alto Research Center, Palo Alto, California, 1974 (revised December, 1975).

[VanLehn 73] Kurt A. VanLehn, *ed, Sail User Manual,* Stanford Artificial Intelligence Project Memo 204, Stanford Computer Science Report STAN-G-73-373, July 1973, updated by James R. Low, Hanan J. Samet, Robert F. Sproull, Daniel C. Swinehart, Russet H. Taylor, Kurt A. VanLehn, March 1974.

III. IMPROVEMENTS IN THE AL RUN-TIME SYSTEM

Bruce E. Shimano

Artificial Intelligence Laboratory
Computer Science Department
Stanford University

The author is a graduate student in the Mechanical Engineering Department.

A. KINEMATIC SOLUTION PROGRAMS

To date, several methods have been described for computing the joint angles necessary to position the Stanford Scheinman Arm at a given point with a specified orientation. Pieper[3] presented a method of solution for the general case of any manipulator with three intersecting axes. Paul[1] presented a geometric solution which has been used at the Artificial Intelligence Project since 1972. More recently, Lewis[4] described a method using vector cross products to obtain expressions for the last three joint angles and Horn[2] presented a method of solution using Euler angles for the MIT-Scheinman Arm.

The following is yet another method of solution which has the advantage of extreme speed. The equations presented below were derived by Lou Paul 'and Bruce Shimano using two different methods, one using vectors and the other algebraic. Only the algebraic solution is presen ted here.

1. Transformation Equations

Given a 4x4 matrix (1) representing the transformation from a coordinate system fixed in the hand of the manipulator to the base coordinate system, we wish to find one set of link variables $\{\theta_1,\theta_2,S_3,\theta_4,\theta_5,\theta_6\}$ for the Stanford Arm which will produce an equivalent transformation.

$$\begin{vmatrix} T_{11} & T_{12} & T_{13} & T_{14} \\ T_{21} & T_{22} & T_{23} & T_{24} \\ T_{31} & T_{32} & T_{33} & T_{34} \\ 0 & 0 & 0 & 1 \end{vmatrix} \qquad (1)$$

The transformations for the individual links of the Stanford Arm were derived by Paul[1]. If we multiply these six matrices together symbolically and equate the components of the total manipulator transformation to the required matrix 'elements (1), we get the following 12 equations in the six joint variables, where "s" denotes the sine function and "c" denotes the cosine function.

$$T_{11} = s\theta_1(-s\theta_4 s\theta_6 + c\theta_4 c\theta_5 c\theta_6) + c\theta_1(-s\theta_2 s\theta_5 c\theta_6 + c\theta_2 s\theta_4 c\theta_5 c\theta_6 + c\theta_2 c\theta_4 s\theta_6) \qquad (2)$$

$$T_{12} = -s\theta_1(s\theta_4 c\theta_6 + c\theta_4 c\theta_5 s\theta_6) + c\theta_1(s\theta_2 s\theta_5 s\theta_6 - c\theta_2 s\theta_4 c\theta_5 s\theta_6 + c\theta_2 c\theta_4 c\theta_6) \qquad (3)$$

$$T_{13} = s\theta_1 c\theta_4 s\theta_5 + c\theta_1(s\theta_2 c\theta_5 + c\theta_2 s\theta_4 s\theta_5) \qquad (4)$$

$$T_{14} = s\theta_1(c\theta_4 s\theta_5 S_6 - S_2) + c\theta_1(s\theta_2 c\theta_5 S_6 + s\theta_2 S_3 + c\theta_2 s\theta_4 s\theta_5 S_6) \qquad (5)$$

$$T_{21} = s\theta_1(-s\theta_2 s\theta_5 c\theta_6 + c\theta_2 s\theta_4 c\theta_5 c\theta_6 + c\theta_2 c\theta_4 s\theta_6) + c\theta_1(s\theta_4 s\theta_6 - c\theta_4 c\theta_5 c\theta_6) \qquad (6)$$

$$T_{22} = s\theta_1(s\theta_2 s\theta_5 s\theta_6 - c\theta_2 s\theta_4 c\theta_5 s\theta_6 + c\theta_2 c\theta_4 c\theta_6) + c\theta_1(s\theta_4 c\theta_6 + c\theta_4 c\theta_5 s\theta_6) \qquad (7)$$

$$T_{23} = s\theta_1(s\theta_2 c\theta_5 + c\theta_2 s\theta_4 s\theta_5) - c\theta_1 c\theta_4 s\theta_5 \quad (8)$$

$$T_{24} = s\theta_1(s\theta_2 c\theta_5 S_6 + s\theta_2 S_3 + c\theta_2 s\theta_4 s\theta_5 S_6) + c\theta_1(-c\theta_4 s\theta_5 S_6 + S_2) \quad (9)$$

$$T_{31} = -(s\theta_2 s\theta_4 c\theta_5 c\theta_6 + s\theta_2 c\theta_4 s\theta_6 + c\theta_2 s\theta_5 c\theta_6) \quad (10)$$

$$T_{32} = s\theta_2 s\theta_4 c\theta_5 s\theta_6 - s\theta_2 c\theta_4 c\theta_6 + c\theta_2 s\theta_5 s\theta_6 \quad (11)$$

$$T_{33} = -s\theta_2 s\theta_4 s\theta_5 + c\theta_2 c\theta_5 \quad (12)$$

$$T_{34} = -s\theta_2 s\theta_4 s\theta_5 S_6 + c\theta_2 c\theta_5 S_6 + c\theta_2 S_3 + S_1 \quad (13)$$

Of these 12 equations only six are independent - the three equations representing position (5),(9),(13) 'and any three of the remaining nine equations which, specify orientation. Furthermore, no unique solution exists for the above set of equations. For the geometric configuration of the Stanford Arm, there' are always at least eight sets of joint variables which satisfy the equations, but due to physical stop limits only two of these eight can ever be attained.

2. *Solution for* θ_1, θ_2, S_3

To solve for the joint variables, we begin by taking advantage of the fact that the axes of the last three joints of the Stanford Arm intersect forming a spherical joint. This simplified geometry allows us to reduce the problem from one of simultaneously solving for all six degrees of freedom to two separate three degree of freedom problems. This is accomplished by subtracting the directed length of the last three joints from the position of the hand. This gives us three equations representing the position of the end of the prismatic joint. These equations are only functions of the first three joint variables. Denoting the end of the prismatic joint by $\{T_X, T_Y, T_Z\}$ and combining equations (4) & (5), (8) & (9), (12) & (13), we get:

$$T_X = T_{14} - S_6 T_{13} = -s\theta_1 S_2 + c\theta_1 s\theta_2 S_3 \quad (14)$$

$$T_Y = T_{24} - S_6 T_{23} = c\theta_1 S_2 + s\theta_1 s\theta_2 S_3 \quad (15)$$

$$T_Z = T_{34} - S_6 T_{33} = c\theta_2 S_3 + S_1 \quad (16)$$

We can now obtain a quation in, the single variable θ_1 by simultaneously eliminating $s\theta_2$ and S_3 from equations (14) and (15). After substituting tangent of the half angle equivalents for the sine and cosine of θ_1 the equation becomes a quadratic polynomial in $\tan\frac{\theta_1}{2}$ whose solution follows:

$$\tan\frac{\theta_1}{2} = \frac{-T_X \pm \sqrt{T_X^2 + T_Y^2 - S_2^2}}{S_2 + T_Y} \qquad (17)$$

This equation will yield two values for θ_1 corresponding to the two configurations obtainable by flipping the prismatic joint over, thus changing from a right to left shouldered arm or vise *versa*. The solutions computed using the "+" sign in front of the radical will produce positive rotation angles for joint 2, whereas the solutions using the negative sign will product negative values of θ_2. Since we always operate our arms with θ_2 in the range r-175, **-5], we will** use the negative form of the solution.

If S_2+T_Y is equal to zero, two special cases must be considered: either $T_X \geq 0$, which indicates that θ_1 is equal to 180 degrees, *or* $T_X<0$ in which case equation (17) is indeterminate and the following equation, can be derived after simplification of the polynomial used to develop eq uation (17):

$$\tan\frac{\theta_1}{2} = \frac{T_Y}{T_X} \qquad (18)$$

Once the tangent of the half angle is determined, the sine and cosine of θ_1 can be computed from **the** following trigonometric identities:

$$s\theta_1 = \frac{2\tan\frac{\theta_1}{2}}{1+\tan^2\frac{\theta_1}{2}} \qquad c\theta_1 = \frac{1-\tan^2\frac{\theta_1}{2}}{1+\tan^2\frac{\theta_1}{2}} \qquad (19)$$

Next, re-writing quations (15) and (16), we obtain the following expressions for the sine and cosine of θ_2.

$$s\theta_2 = \frac{T_Y - S_2 c\theta_1}{S_3 s\theta_1} \qquad c\theta_2 = \frac{T_Z - S_1}{S_3}$$

Substituting expressions (17),(19) for the sine and cosine of θ_1, we obtain:

$$s\theta_2 = -\frac{\sqrt{T_X^2 + T_Y^2 - S_2^2}}{S_3} \qquad c\theta_2 = \frac{T_Z - S_1}{S_3} \qquad (20)$$

Since we are limited to working with the extension of the prismatic joint, S_3, between 6 and 35 inches, the ratio of above equations will be determinate and independent of S_3. Hence, we can co-mpute θ_2 using an arc-tangent function and quations (20).

The extension of joint' three can now be found by evaluating the following equation which was derived by simultaneously solving equations (14) and (15) for S_3:

$$S_3 = \frac{T_X c\theta_1 + T_Y s\theta_1}{s\theta_2} \tag{21}$$

Since mechanical stop limits prevent θ_2 from being either 0 or -180 degrees, the equation for S_3 is never indeterminate.

Having found values for θ_1, θ_2, and S_3, we can now solve the remaining nine transform element equations for values of θ_4, θ_5, and θ_6.

3. *Solution for $\theta_4, \theta_5, \theta_6$*

There are primarily two forma of the following equations that can be used to solve for the last three joint 'angles. The two sets of quations differ in how the degenerate condition for the last joints must be treated. The arm configuration is called degenerate whenever θ_5 is equal to 0 or 180 degrees: At these times the axes of rotation for joints 4 and 6 are collinear and only the sum of θ_4 and θ_6 can be treated as an independent variable. One form of the solution equations for the last joints has the advantage of producing valid results whether or not the arm is in a degenerate position. However, these equations are slower to evaluate than the equations which require that degenerate configurations be treated as a special case. For this reason, only the latter form of the equations will be presented.

We will find it convenient to work with combinations of the equations for the third column of the transform matrix in deriving expressions for θ_4 'and θ_5. Since this column indicates the direction of the Z-axis of the hand, all of its terms are independent of θ_6. From equations (4), (8), (12) we obtain the following expressions:

$$T_{13}s\theta_1 - T_{23}c\theta_1 = c\theta_4 s\theta_5 \tag{22}$$

$$T_{23}s\theta_2 + T_{33}s\theta_1 c\theta_2 = s\theta_1 c\theta_5 - c\theta_1 s\theta_2 c\theta_4 s\theta_5 \tag{23}$$

$$T_{23}c\theta_2 - T_{33}s\theta_1 s\theta_2 = s\theta_1 s\theta_4 s\theta_5 - c\theta_1 c\theta_2 c\theta_4 s\theta_5 \tag{24}$$

$$T_{13}c\theta_2 - T_{33}c\theta_1 s\theta_2 = s\theta_1 c\theta_2 c\theta_4 s\theta_5 + c\theta_1 s\theta_4 s\theta_5 \tag{25}$$

In order to distinguish between degenerate and non-degenerate configurations, we will begin by deriving quations for the sine and cosine of θ_5 in terms of θ_1, θ_2, and S_3. Combining quations (22) & (23) and (22) & (24) & (25), we obtain:

$$s\theta_5 = \pm\sqrt{((T_{13}c\theta_1 + T_{23}s\theta_1)c\theta_2 - T_{33}s\theta_2)^2 + (T_{13}s\theta_1 - T_{23}c\theta_1)^2}$$

$$c\theta_5 = (T_{13}c\theta_1 + T_{23}s\theta_1)s\theta_2 + T_{33}c\theta_2 \tag{26}$$

The sine quation reflects the fact that θ_5 can be arbitrarily selected to be positive *or*

negative. Since the last three joints have intersecting axes any two sets of joint angles $\{\theta_4,\theta_5,\theta_6\}$ and $\{\theta_4+180,-\theta_5,\theta_6+180\}$ are equivalent and in fact occupy the same volume in space. If joints 4 or 6 have less than 360 degrees of rotational freedom, the duplicate solutions can be used to minimize the loss in orientation capability. Otherwise the choice among the solutions can be made on the basis of producing the minimum total change in joint angles.

If θ_5 is equal to 0 or 180 degrees, then either θ_4 or θ_6 can be selected arbitrarily and. the remaining angle of rotation must satisfy the transformation equations. As the full range of θ_5 for the Stanford Arm is [-110,110], we need only concern ourselves with the case of θ_5 equal to zero. Equations (10) and (11) yield the desired equations for this degenerate case.

$$\sin(\theta_6 + \theta_4) = -\frac{T_{31}}{s\theta_2} \qquad \cos(\theta_6 + \theta_4) = -\frac{T_{32}}{s\theta_2} \tag{27}$$

If the configuration is not degenerate than we can use the following expressions for θ_4 which can be derived from equations (22), (24), and (25).

$$s\theta_4 = \frac{(T_{13}c\theta_1+T_{23}s\theta_1)c\theta_2 - T_{33}s\theta_2}{s\theta_5} \qquad c\theta_4 = \frac{T_{13}s\theta_1 - T_{23}c\theta_1}{s\theta_5} \tag{28}$$

In order to form expressions for θ_6, we will now make use of the remaining two columns of the transform. These first two columns give the orientation of the hand about its Z axis. We will find it convenient to immediately combine equations (2) & (3), (6) & (7), and (10) & (11) to eliminate some of the variables.

$$T_{11}s\theta_6 + T_{12}c\theta_6 = -s\theta_4 s\theta_1 + c\theta_1 c\theta_2 c\theta_4 \tag{29}$$

$$T_{21}s\theta_6 + T_{22}c\theta_6 = s\theta_4 c\theta_1 + s\theta_1 c\theta_2 c\theta_4 \tag{30}$$

$$T_{31}s\theta_6 + T_{32}c\theta_6 = -s\theta_2 c\theta_4 \tag{31}$$

From these three equations, we can obtain explicit formulas for the sine and cosine of θ_6. However, rather than use these equations we can obtain much simpler expressions if we combine the three equations above with (22), (24), and (25) to obtain the following formulas.

$$s\theta_6 = \frac{(T_{12}c\theta_1+T_{22}s\theta_1)s\theta_2 + T_{32}c\theta_2}{s\theta_5} \qquad c\theta_6 = -\frac{(T_{11}c\theta_1+T_{21}s\theta_1)s\theta_2 + T_{31}c\theta_2}{s\theta_5} \tag{32}$$

This completes the solution for the joint angles of the Stanford Arm from a desired tranformation.

4. Solution Execution Time

A new arm solution program has been written which employs the equations presented in this paper. The execution time for this routine is approximately 2.2 milliseconds on a PDP 11/45 using hardware floating point arithmetic. This is roughly half of the time that was formerly required to compute the joint angles given a tranformation.

4. Reverse Solution Program

To compute the hand transformation from the joint angles, Horn [2] demonstrated that for the MIT-Scheinman Arm it was *very* efficient to symbolically expand the matrix products of the first three and last three joint transformations and hand code the computation of the resulting matrix elements. The total arm transformation could then be determined by multiplying the two matrices together in the standard fashion. A further improvement to this scheme has been suggested by Lou Paul. Instead of forming the full 4x4 matrix representing the transformation for the last three joints, only its last three columns are explicitly computed. The last three columns of the full arm transform can then be determined by multiplying the two partial transformations together, while the first column of the full transformation can be formed by taking the cross product of the second and third columns. We have written a program to perform these operations for the Stanford-Scheinman Arm and find that it executes in approximately a third of the time of our former method of multiplying the six link matrices together. The nominal execution time for this new program is approximately 2.0 msec. The two partial transforms used for this computation are presented below. All four columns of the transform for the last three joints are presented for the sake of completeness.

Transform from A1 to A3:

$$\begin{vmatrix} s\theta_1 & c\theta_1 c\theta_2 & c\theta_1 s\theta_2 & -s\theta_1 S_2 + c\theta_1 s\theta_2 S_3 \\ -c\theta_1 & s\theta_1 c\theta_2 & s\theta_1 s\theta_2 & s\theta_1 s\theta_2 S_3 + c\theta_1 S_2 \\ 0 & -s\theta_2 & c\theta_2 & c\theta_2 S_3 + S_1 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Transform from A4 to A6:

$$\begin{vmatrix} -s\theta_4 s\theta_6 + c\theta_4 c\theta_5 c\theta_6 & -s\theta_4 c\theta_6 - c\theta_4 c\theta_5 s\theta_6 & c\theta_4 s\theta_5 & c\theta_4 s\theta_5 S_6 \\ s\theta_4 c\theta_5 c\theta_6 + c\theta_4 s\theta_6 & -s\theta_4 c\theta_5 s\theta_6 + c\theta_4 c\theta_6 & s\theta_4 s\theta_5 & s\theta_4 s\theta_5 S_6 \\ -s\theta_5 c\theta_6 & s\theta_5 s\theta_6 & c\theta_5 & c\theta_5 S_6 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

B. AUTOMATIC FORCE WRIST CALIBRATION

In an earlier report [6], we discussed the design criteria and operational specifications for a six degree of freedom force and moment sensing wrist that was designed and built for the

Stanford Arm. At that time, we noted the importance of calibrating the force wrist since we found that unpredictable mechanical coupling had caused our theoretical estimates of the response of the wrist to be in error by as much as 5 percent. Furthermore, while the thermal drift and hysteresis were fairly small, it nevertheless seemed a wise precaution to re-calibrate the wrist from time to time. For our initial tests, we calculated the calibration matrix using a method that required that we apply three orthogonally oriented forces and three orthogonally oriented moments to the geometric center of the wrist. To this end, we set up an elaborate system of pulleys and weights. While this type of procedure is acceptable for testing purposes, it will prove to be too time consuming to employ when the wrist is in daily use. Indeed, once the force wrist is mounted on the manipulator, applying pure forces and moments to the force sensing wrist may be impossible without either detaching the hand or attaching special collars. A more acceptable method than either of these two alternatives is to use a method of calibration that can deal with coupled combinations of forces and moments. Also, in order to minimize the set-up time, we wanted to employ a method of calibration that requires the minimum number of special purpose attachments to the arm. With this in mind, the following calibration procedure was devised.

1. Forward and Reverse Calibration Matrices

F'or our force sensing wrist, a total of eight pairs of strain gages must be sampled in order to resolve the three components of force and the three components of moment applied to the wrist. If the assumptions of superposition and perfect elasticity are made, any one of the components of force or moment would in theory be only a function of two or possibly four of the strain gage readings. In fact, we found that, it was necessary to consider each component of force to be a function of all eight strain gages in order to achieve better than 1% accuracy. If we let $\{F_X, F_Y F_Z, M_X, M_Y M_Z\}$ represent our force vector and ϵ_i ($i = 1$ to 8) the eight strain gage readings, calculating the force vector from the strain gage reading can be accomplished by the following matrix operation.

$$F = C \times r \tag{33}$$

where

$$F = | \; F_X \;\; F_Y \;\; F_Z \;\; M_X \;\; M_Y \;\; M_Z \; |^T$$

$$\epsilon = | \; \epsilon_1 \;\; \epsilon_2 \;\; \epsilon_3 \;\; \epsilon_4 \;\; \epsilon_5 \;\; \epsilon_6 \;\; \epsilon_7 \;\; \epsilon_8 \; |^T$$

$$C = \begin{vmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} & c_{16} & c_{17} & c_{18} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} & c_{26} & c_{27} & c_{28} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} & c_{36} & c_{37} & c_{38} \\ c_{41} & c_{42} & c_{43} & c_{44} & c_{45} & c_{46} & c_{47} & c_{48} \\ c_{51} & c_{52} & c_{53} & c_{54} & c_{55} & c_{56} & c_{57} & c_{58} \\ c_{61} & c_{62} & c_{63} & c_{64} & c_{65} & c_{66} & c_{67} & c_{68} \end{vmatrix}.$$

The objective of calibrating the force sensing wrist is to compute the c_{ij} matrix elements based upon experimental information. To do this, we will first compute the elements of the

pseudo inverse calibration matrix which was described by Watson [5]. The calibration matrix, C, and its pseudo inverse, Cn, are related by the following formula.

$$C = (Cn^T \times Cn)^{-1} \times Cn^T \qquad (34)$$

The pseudo inverse matrix has terms called cn_{ij} (i= 1 to 8, j= 1 to 6). The pseudo inverse is analogous to the **normal inverse** matrix but it is defined for non-square as well as square matrices. The Cn matrix can be used for computing the response of a single strain gage to the application of a specified force vector. This relationship can be written as follows:

$$\epsilon = Cn \times F \qquad (35)$$

Once we have computed Cn, we can use equation (34) to compute the elements of the forward calibration matrix, C.

2. Computing the Pseudo Inverse Calibration Matrix

To compute the elements of the pseudo inverse matrix, six independent, known force vectors must be applied to the wrist. These force vectors need not be orthogonal nor do they have to be pure forces or moments; however, we will require that their values be known at a single point whose position is known relative to the center of the force wrist. For each of these force systems, all eight strain gage readings are to be recorded. We will define the values of the force vector and the readings as follows:

ϵ_{ij} = the reading of the jth strain gage due to the application of the ith force vector (i= 1 to 6; j= 1 to 8).

f_{ik} = the kth component of the ith independent force vector (i=1 to 6; k=1 to 6).

For each of the six independent force vectors, equation (35) must apply, so we can write;

$$\epsilon_{ij} = f_{i1} * cn_{j1} + \ldots\ldots + f_{i6} * cn_{j6}$$
for i = 1,2,...,6; j = 1,2 ,...., 8

These 48 equations can be re-written as the following matrix equations:

$$\begin{vmatrix} \epsilon_{1j} \\ \epsilon_{2j} \\ \epsilon_{3j} \\ \epsilon_{4j} \\ \epsilon_{5j} \\ \epsilon_{6j} \end{vmatrix} = \begin{vmatrix} f_{11} & f_{12} & f_{13} & f_{14} & f_{15} & f_{16} \\ f_{21} & f_{22} & f_{23} & f_{24} & f_{25} & f_{26} \\ f_{31} & f_{32} & f_{33} & f_{34} & f_{35} & f_{36} \\ f_{41} & f_{42} & f_{43} & f_{44} & f_{45} & f_{46} \\ f_{51} & f_{52} & f_{53} & f_{54} & f_{55} & f_{56} \\ f_{61} & f_{62} & f_{63} & f_{64} & f_{65} & f_{66} \end{vmatrix} \times \begin{vmatrix} cn_{j1} \\ cn_{j2} \\ cn_{j3} \\ cn_{j4} \\ cn_{j5} \\ cn_{j6} \end{vmatrix} \qquad (36)$$

This formula represents a system of eight matrix equations, each of which relates the

response of an individual force sensing element to a series of force vectors. For each of the matrix equations, 'we have read the six values of the specified strain gage and so long as the six force vectors are independent, the 6x6 matrix in equation (36) will be non-singular. Therefore, by applying a standard routine that solves sets of linear equations, we can solve equation (36) for the values of the elements of one row of the inverse calibration matrix, cnjk (k = 1 to 6). By repeating this procedure for each of the eight matrix equations, all of the elements of the pseudo inverse calibration matrix can be determined . Equation (34) can then be used to compute the forward calibration matrix.

It should be noted that this basic method of calculating the pseudo inverse calibration matrix would still be applicable if one wanted to utilize the information from more than six force vectors. If there are n samples taken (n>6), the matrices in equation (36) can be replaced by (n x m) matrices and an approximate solution for the rows of the inverse calibration matrix can be found by the method of least squares.

3. Calibration Procedure for the Stanford Arm

As we are no longer restricted to applying pure forces and moments, we will be able to calibrate the force wrist while it is mounted on the Stanford Arm. In fact, we can utilize the positional and rotational degrees 'of freedom of the manipulator together with the weight of its hand to aid in the calibration procedure.

Since the force wrist is mounted between the last active rotary joint and the hand of the manipulator, the known weight of the hand will be used as a standard of reference against which all other weights will be compared. While the weight of the hand is not the best of all possible references, due to its light weight compared to the maximum load that the force wrist can measure, it does have the advantage of being constant and ever present. Also, since the weight of the hand must be subtracted from whatever readings we take with the force. wrist, using it as a reference will reduce the absolute error when small forces are to be measured. Furthermore, since calibration measurements only require the reading of static forces, many readings can be taken for each force vector and digital filtering can be applied to increase their precision.

We now present the outline of a simple program which can be used to calibrate the force sensing wrist automatically, i.e., without the intervention of manipulator programmers. For our measurements, we will define HW to be the weight of the hand, DCM to be the distance from the center of mass of the hand to the center of the force wrist, and DH to be the distance from the center of the hand fingers to the center of the force wrist. We will resolve all forces and moments at the geometric center of the force wrist.

1. The first force vector to be applied is {0,0,2HW,0,0,0}. To obtain the corresponding strain gages readings, the arm is first moved to a position with the hand pointing directly down and a series of readings are taken. Then the arm is re-positioned so that the hand is pointing directly up and a

second set of readings are taken. The difference between the two sets of readings are saved as our ϵ_{1j}

2. Next the wrist of the arm is rotated until the hand is horizontal and the X axis of the force balance is vertical. The readings taken in this position and with the hand rotated 180 degrees about its central axis correspond to the force vector {2H W ,0,0,0,2H W *DCM,0}.

3. The third set of readings are to be taken in exactly the same manner as the second set except that the hand is first rotated about its central axis 90 degrees to align the Y axis of the force sensor with the vertical. The force vector associated with this set of readings is {0,2HW,0,-2HW*DCM,0,0}.

4. In order to obtain two more independent readings that combine forces and moments along the X and Y axes, the manipulator is now directed to locate and pick up any convenient object in the work area. After the object has been grasped, all that we need to know is the position of center of mass of the object relative to the center of the force wrist. For this purpose, it is convenient to work with a fairly symmetric object that can be grasped such that its center of mass coincides with the geometric center of the finger tips. Ii this is true, then the weight of the object can be determined by repeating step 1 with the object in hand. The new readings can be scaled against the old and the weight of the object can be determined in terms of the known weight of the hand. We will call the weight of the object WT. We can now repeat step 2 with the object in hand. The force vector corresponding to these readings will be {2HW+2WT,0,0,0,2HW*DCM+2WT*DH,0}.

5. We now duplicate step 3 with the weight in hand to obtain a new set of readings. The force vector produced by the combined weight of the hand and object will be {0,2HW+2WT,0,-2HW*DCM-2WT*DH,0,0}.

6. For the final force vector, the manipulator must grasp an object fixed in place. This can be a vise, another manipulator, or even a willing and strong human volunteer. After ensuring that no net forces or moments exist along any of the axes, the motor of the last rotary joint of the arm is driven with a constant and known torque, T. Readings are taken for the torque directed in both directions and the corresponding force vector is given by {0,0,0,0,0,2T}.

Once the strain gage readings for the six independent force vectors have been taken, the procedure discussed in the previous section can be used to compute the calibration matrix for the-force sensing wrist.

4. Resolving Forces and Moments at an Arbitrary Point

It is often necessary to resolve the strain gage readings into forces and moments that act at a

point other than the position used for the calibration. For example, we might wish to monitor the forces at the finger tips to enable us to stop *on* contact or we might be interested in the interaction between a tool we are holding and a work piece. In either of these cases the problem is to determine the applied force vector at a point located at a distance $\{d_X,d_Y,d_Z\}$ from the calibration point. For a simple linear translation, the new force vector, $\{F_X',F_Y',F_Z',M_X',M_Y',M_Z'\}$ is related to the force vector at the point of calibration, $\{F_X,F_Y,F_Z,M_X,M_Y,M_Z\}$, by the following matrix equation.

$$\begin{vmatrix} F_X' \\ F_Y' \\ F_Z' \\ M_X' \\ M_Y' \\ M_Z' \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -d_Z & +d_Y & 1 & 0 & 0 \\ +d_Z & 0 & -d_X & 0 & 1 & 0 \\ -d_Y & d_X & 0 & 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} F_X \\ F_Y \\ F_Z \\ M_X \\ M_Y \\ M_Z \end{vmatrix} \qquad (37)$$

We now call the 6x6 matrix on the right D. Then in order to directly resolve the strain gage readings into an equivalent force and moment at a point located at $\{d_X,d_Y,d_Z\}$ in the calibration coordinate system, we combine equations (37) and (33) to obtain:

$$F' = D \times C \times \epsilon$$

Finally, if we desire to rotate the coordinate system along which the forces and moments are resolved, we can again pre-multiply the calibration matrix by an appropriate 6 x 6 matrix. Assuming that the rotation is represented by a 3 x 3 matrix, R, which defines the rotation from the calibration to the new coordinate system, the total transformation from strain gage readings to the desired force vector will be given by:

$$F' = R' \times D \times C \times \epsilon$$

where

$$R' = \begin{vmatrix} R & 0 \\ 0 & R \end{vmatrix}$$

5. *Current State of the Force Sensing System*

The calibration method that has been described in the preceding sections has been used to calibrate a force sensing wrist with an attached hand that was not as yet mounted on a manipulator. From these initial tests it appears that the calibration method works quite well. We were able to compute a calibration matrix that could accurately resolve subsequent forces and moments to within approximately 1%.

At present, we are awaiting the mounting of our force sensing wrist on one of our Stanford Arms. In anticipation of this event, software has been written which can be used to calibrate

the wrist automatically. In addition, the software now exists to compute forces from the strain gage readings given the calibration matrix and to resolve these forces at a point separated from the calibration center by a linear transformation.

Bibliography

[1] Richard Paul, *Modelling, Trajectory Calculation and Servoing of a Computer Controlled Arm,* Stanford Artificial Intelligence Laboratory Memo AIM- 177, Stanford Computer Science Report STAN-CS-72-311, November 1972.

[2] Berthold K. P. Horn, Hirochika Inoue, *Kinematics of the MIT-AI-VICARM Manipulator,* Massachusetts Institute of Technology Working Paper 69, May 1974.

[3] Donald L. Pieper, *The Kinematics of Manipulators Under Computer Control,* Stanford Computer Science Report, STAN-CS-68-116, October 1968.

143 Richard A. Lewis, *Autonomous Manipulation on a Robot: Summary of Manipulator Software Functions,* Jet Propulsion Laboratory Technical Memorandum 33-679, March 1974.

[5] P. C. Watson, S. H. Drake, *Pedestal and Wrist Force Sensors for Automatic Assembly,* Proceeding 'of the 5th International Symposium on Industrial Robots, September 1975, pp. 501-511.

[6] T. 0. Binford, R. Paul, J. A. Feldman, R. Finkel, R. C. Bolles, R. H. Taylor, B. E. Shimano, K. K. Pingle, T. A. Gafford, *Exploratory Study of Computer integrated Assembly Systems,* Prepared for the National Science Foundation. Stanford Artificial Intelligence Laboratory Progress Report covering March 1974 to September 1974.

IV. GENERATING AL PROGRAMS FROM HIGH LEVEL TASK DESCRIPTIONS

Russell H. Taylor

Artificial Intelligence Laboratory
Computer Science Department
Stanford University

The author is currently a Research Staff Member in the Computer Science Department, IBM T. J. Watson Research Center, P. 0. Box 218, Yorktown Heights, N. Y. 10598. At the time this research was performed, he was a graduate student in the Computer Science Department at Stanford University.

Chapter 1.

INTRODUCTION

This report summarizes recent work on the automatic generation of AL programs from high level task descriptions. It is divided into three major chapters. First, the AL language is reviewed briefly, and an extended programming example is used to illustrate the problems that arise when people write manipulator programs. Next, the modeling req uiremcnts for automatic coding are analyzed, since the automation of coding decisions requires that the necessary information be represented in a form usable by the computer. Finally, the extended programming example is revisited, this time with the computer using its planning model to generate the AL code automatically.

The material contained in this report is a condensed version of part of my dissertation [28]. The main omissions are a discussion of the AL planning model, object models, and the representation of location and accuracy information, although the appendix in this report summarizes some of the latter material.

Chapter 2.

THE AL PROGRAMMING PROCESS

2.1 Introductory Remarks

This chapter provides a brief overview of the AL language, illustrating its characteristics by an extended programming example, which allows us to examine in some detail the AL programming process.

2.2 Overview of the AL Language

Superficially, AL programs look very much like ALGOL programs. The language is block oriented, and variants of the usual ALGOL structures are used for program control. Since the programs must be executed in a real-time environment, where several things can be happening at once, additional control structures for concurrency and synchronization are required. The necessary capabilities are supplied by the well known cobegin ... coend and event signal and writ primitives.[1]

Data Types

One of the key attributes of a formal language for manipulator control is the use of named variables to describe positions, forces, and other relevant data. Using only the data types of ALGOL would make programs hard to read and would increase the chance of bugs. AL avoids these difficulties by providing data types and "arithmetic" operations for the physical and geometric entities rquired for describing manipulation. The most important of these special types are frames, which are used to represent coordinate systems, and transes, which tell how frames are related. AL programs use frames to describe hand positions and object locations; the set of frame variables and their associated values thus constitute a major part of a program's execution-time model of the world.

Affixment

In manipulation tasks, it is common to have several frames associated with the same object, with each frame playing an important role. When the object is moved, the frames all assume new values. AL provides two distinct ways for handling this. One way is to use a trans variable to recompute each frame value each time it is needed. Thus, a user might write an expression like

*box*grasp_xf*

to specify the proper hand position for grasping the object whose coordinate system is given

---

[1] Various flavors of these primitives come under many names. See, for instance, [8] for further discussion.

by the **frame** *box*. This approach can get tedious where the same frames **are being** referenced repeatedly, and tends to hide the *intent" of a program behind a smokescreen of frame transformations. The alternative method of using a separate **frame** variable for **each** frame of interest makes motion statements easier to read and write, but means that all associated variables **must** be updated whenever **something is changed.** The **affix construct** in AL allows the user to specify that a variable is to be **"continuously"** computed from other **variables.** For **instance,**

affix *box_grasp* **to** *box* at *grasp_xf*

would cause the assignment statement

box-grasp ← *box*•*grasp_xf*

to **be performed** automatically every time *box* is updated.* **When one** object **is assembled** to another, or when an object is grasped by the manipulator, it is customary to **affix their location** variables. For example,

affix *cover* **to** *box;*
affix *box* **to blue;**[3]

The data structures associated with affixments thus form another important part of **an** AL **program's** model of the world.

Motion Statements

In the tasks for which AL was designed, the hand is the only part of the manipulator that interacts directly with other objects. The position of the rest of the **arm is generally** irrelevant, so long as it doesn't collide with anything. Thus, AL programs describe motions **by specifying a sequence** of **frame** values through which the hand must pass. For instance,

move blue to *box_grasp* via *grasp_approach;*

Since the purpose of manipulation is to move objects, rather than to get the manipulator's **hand to particular places,** thts concept **has been** generalized to allow the user to describe **motions in terms** of frames other than **the** hand itself. Thus,

affix *box* to blue;
•
move *box* **to** *new_box_place* via *midair_point;*

Here, **the affix** statement tells the system that changes in the value of the **blue** hand are to

---

[2] **Actually,** this is an over-simplification. *box-grasp* would merely **be** marked as invalid and a **new** value recomputed when required.

[3] The manipulator hardware at Stanford consists of two Scheinman arms, one of which is **anodized blue,** and **the** other, gold. Thus, **blue** and yellow are **predefined** AL frames corresponding to the hands of the two arms. (At the time of this writing, only **blue has been** interfaced to the **runtime** system)

cause corresponding changes in the value of box. This information is used to produce hand positions **that** cause box to pass through *midair-point* and wind up at *new-box-place.* The sequence of destination points is translated by AL into the corresponding joint behavior by a combination of compile-time planning and execution-time revision.

Although a simple list of destination points is sufficient for some purposes, many tasks require a more detailed specification of how motions are to be performed. Items of interest include the time to be spent on each motion segment, forces to be exerted by the hardware, external forces to which the manipulator is to be compliant, and conditions to be monitored during the motion. This information is supplied in AL programs by the use of clauses **which** modify the basic motion statement. For example,

```
move carburetor to inspection-station
    via unloading-pint
        where force(xhat)=0,[4]
              force(yhat)=0,
              duration > 2*sec
    via approach-point
    on force(zhat) > 8*oz do
        stop
    on electric-eye-interrupt do
        signal passed-checkpoint;
```

might occur in a carburetor assembly program, where a carburetor has been assembled in a fixture and now must be moved to an inspection station. While the carburetor is removed from **the** fixture, the arm is made compliant to forces in x and y, and the motion is slowed down to take at least two seconds. The carburetor is then moved to the inspection-station via an intermediate approach point. To avoid the possibility that a small positioning error might cause the manipulator to shove the carburetor through the table, the motion is terminated as soon as the force **in** the **z** direction exceeds a half pound. Finally, as soon as **an** electric eye detects something, an external control signal is generated.

Tta jectories

Ultimately, ail **manipulator** motions must be described in terms of joint motions, since joints are what the runtimc system **can** control. However, this representation is awkward for specifying motions and introduces a needless degree of hardware dependency if it is used. Motions are specified in AL programs by giving a list of positions through which an object. is to pass. The requited coordination is achieved by solving the joint angle equations for each position. These data points are then used to produce polynomials (in time) which describe the behavior of each joint."

Unfortunately, the computation required for preparation of these polynomials is non-trivial. Consequently, the **compiler** must pre-compute trajectories, based on *a* ***planning*** *model* of

---

[4] xhat, yhat, and **that** are **unit** vectors in the x, y, and **z** directions.

[5] This method was developed by R. Paul, and, is reported in [19]. More recent refinements may **be** found in [5] and [10]. In his recent work, Paul has abandoned polynomials **in** favor of an intapolation scheme [21,20].

expected **affixment** and frame values. These **precomputed** polynomials are modified by the addition of higher order terms just before the motion **is executed,** so that the positions reached correspond to the actual **runtime** values. This approach produces well behaved motions, so long as the required modifications are not too great. However, it also creates a number of problems for the compiler, which must maintain the planning model. Eventually, it is hoped that trajectory planning can be done completely at run time. However, this will not eliminate the need for a planning model, which is also used for **affixments and** for other **purposes.**[6]

2.3 Sample AL Program

Manipulator programming is **a** non-trivial intellectual activity, even for simple tasks. This section illustrates the use of **AL** to accomplish a simple assembly operation – the insertion of an aligning pin into a hole – which is **a** typical **subtask** for many assembly programs. The discussion provides some insight into the process of writing AL programs. First, an outline for the program will be developed. A simple "first cut" program implements this task outline. **We** then examine the flexibility and "toughness" of this **program.** Methods for error detection and recovery are discussed, and **a** new, more elaborate, program is produced.

2.3.1 The Task

Initially, the pin sits in a tool rack, and a metal box with holes in it sits on the table in some known position. Our mission is to get the aligning pin into one of the holes. The way to do this is to **grasp** the pin between the manipulator's fingers, extract it from the rack **hole,** transport **it** to a point over the **hole,** and insert **it** into the hole. Thus, our program, in. **outline,** looks something like this:

```
begin "pin-in-hole"
{ Declarations and initial affixments }
{ Grasp the pin }
{ Extract & transport over hole }
{ Insert }
{ Let go of the pin }
end
```

2.3.2 Declarations and **Affixments**

The declarations include frame variables for the pin, hole, and other points of interest. In **addition,** we must write affixment statements describing how the various frames are linked. Strategy **decisions** are embodied in these declarations. For instance, we need to declare a **frame,** ***pin_grasp,*** for use in the grasping operation. It seems natural to **affix** this frame to *pin,* But where? If there is any chance that the pin can bind in the rack or box hole, then

---

[6] As present **capabilities** are extended, we will probably 'want to include other facilities (like **collision avoidance) which** are too expensive to be done at **runtime,** and, so, require pre-**planning.**

it will probably be a good idea to twist the pin during extraction and/or insertion operations. To do this effectively, we must grasp the pin so that its axis lines up with the wrist axis. Alternatively, grasping the pin at an angle may be better for reasons of collision avoidance or may allow us to produce a more efficient program by reducing arm motion times. In this case, we've decided to twist the pin, so that the "end on" grasping position must be used. Usually, one doesn't sit down and write all the declarations before writing any code. This has been done here largely for convenience of exposition:

```
frame pin, pin-grasp, pin_grasp_approach;
frame pin-holder, pin-withdraw;
frame hole, in-hole-position, hole-approach;
frame box;

affix pin-withdraw to pin-holder
      at trans(rot(zhat,30*deg),vector(0,0,4*cm));
pin-holder ← frame(nilrotn,vector(15*inches,10*inches,0));

affix pin-grasp to pin at trans(rot(xhat,180*deg),vector(0,0,2*cm));
affix pin-grasp-approach to pin-grasp at trans(nilrotn,vector(0,0,-3*cm));
pin ← pin-holder!

affix h-hole-position to hole rigidly
      at trans(nilrotn,vector(0,0,-1*cm));
affix hole-approach to hole at trans(nilrotn,vector(0,0,+1*cm));
affix hole to box at trans(nilrotn,vector(5*cm,4*cm,3*cm));
box ← initial-box-position;
```

There may be several choices of what affixments to make, as well as where to make them. For instance, *we* have *affixed pin-grasp* to *pin*. One consequence is that, if *pin* should be rotated, the position of the hand (with respect to the tool rack) when the pin is grasped will similarly be rotated. The rotation won't make much difference in this case, since *pin* is assigned an explicit value and since the arm configuration won't be much changed by rotations of *pin*, anyhow. In other circumstances, arm solution or collision avoidance considerations may make it desirable to affix *pin-grasp* to *pin-holder* instead.

2.3.3 Grasping the Pin

To grasp the pin, it is necessary to open the fingers an appropriate amount, move the hand to *pin-grasp*, and close the fingers. The corresponding AL code is

```
open bfingers to 1.0*inches;
      { The 1.0*INCHES is sort of arbitrary. }
move blue to pin-grasp;
close bfingers;
affix pin to blue;
      { The pin will move if the hand does. }
```

The most serious difficulty with this code is that the manipulator may collide with something on the way to *pin-grasp*. Since the AL compiler does not do collision avoidance,

we must tend to this detail for ourselves by specifying enough intermediate points so that we stay out of trouble. What points are required depends on where the manipulator is before starting the motion, which we haven't specified, and on what other objects are in the workspace. For the moment, we assume that the manipulator **is** "clear" of any extraneous obstructions, and consider only the **possibility** that the fingers might collide with the pin while moving to *pin-grasp.* This may be avoided by moving through an intermediate point, *pin-grasp-approach,* affixed to *pin-grasp* in such a way that the final part of the motion will take place along the wrist axis of the hand.! Note that this affixment structure guarantees that the fingers will stay out of the way of the pin even if we change the relation of *pin-grasp* to *pin.*

Another difficulty is that the execution-time value for *pin-holder* may be inaccurate. If the rack is bolted to the table, the CLOSE statement may overstrain the manipulator. This problem can be avoided by adding some compliance to the motion:

```
close blue
      with force(xhat)=0, force(yhat)=0;
```

An alternative is to use the **center** statement, which, makes the motion compliant to the touch sensors **on the** finger pads.

2.3.4 Initial Program

Once the pin is grasped, a single motion statement can perform the extraction, transport, and insertion operations. After the pin is in the hole, we can let go of it and move the arm back out of the way, again being sure not to hit the pin with the fingers while moving off. These operations and the (revised) grasp code gives us the following program:

```
begin "pin in hole"
{ Declarations and initial affixments }

{ Grasp the pin }
open bfingers to 1.0*inches;
move blue to pin_grasp via pin_grasp_approach;
center bfingers;
affix pin to blue;

{ Extract, transport, and insert }
move pin to in-hole-position via pin-withdraw, hole_approach;

{ Let go of the pin }
open bfingers to 1.0*inches;
unfix pin from blue;
move blue to bpark via pin-grasp-approach;

end;
```

---

[7] For this reason, Paul calls $\hat{z}$ the "approach" axis of the manipulator. **We will** adopt this **usage occasionally, also.**

The value of *pin-grasp-approach* in the final **move** statement will have been updated as a consequence of its (indirect) affixment to *pin*. If we had chosen to affix *pin-grasp* to *pin-holder*, rather than to ***pin***, this updating would not occur, and the motion specified would be rather wild. In **this case**, we could always invent a **new** variable and affix it to hole. Alternatively, we could compute the withdrawal point directly, as in:

```
move blue to bpark via blue*trans(nilrotn,vector(0,0,-3*cm));
```

This works because the values of all points in the destination list are computed **before** the motion is begun. If we have a number of such motions, it may be convenient to **invent** *a* **frame and affix it to the manipulator:**

```
frame withdraw,3;
affix withdraw_3 to blue at trans(nilrotn,vector(0,0,-3*cm));
      :
move blue to bpark via withdraw_3;
```

2.3.5 Critique of Initial **Program**

The program we have just written is complete in the sense that **it describes** a sequence of operations that should transfer the pin to the box hole. Whether it will work reliably enough is another **question.**[8] Certainly, any "easy" things that we can do to make the code more robust ought to be given careful consideration.

We have already built one important form of flexibility into the program by using variables, **rather than** constants, to describe locations. This has several advantages. **The** code is easier to understand, since an identifier like *"pin-holder"* is generally more informative than an expression like **"frame(nilrotn,vector(*15*inches,*10*inches,*0*))".** Modification of programs to accommodate changes in part locations is much easier, since the values only appear explicitly once?

These advantages could also have been derived from the use of ***compile-time*** variables or macros for symbolic definition of constants. An advantage unique **to** execution-time variables is the fact that values can be *recomputed* and saved when the program is run. Thus, our program will work correctly for many different initial box positions, so long as the built-in assumptions (that the box is upright on the table, in reach of the arm, etc.) are not **violated.**[10]

---

[8] Murphy **[17]** has investigated the reliability of systems in some detail. Experience **has** verified **that his results apply** with special force to manipulator programming.

[9] **Indeed,** one **can** write programs like the one developed in this section at one's desk. **The** required location values can then be measured during initial setup. (For instance, using a **system** like **POINTY,** which is discussed in **[14,6]). There** are number of tradeoffs involved in this mode of programming, the principal advantage being the reduction of manipulator downtime while a new application is programmed, and the principal **disadvantage** being the loss of immediate feedback while the program is being written.

[10] Actually, the fact that **AL** preplans arm **trajectories** means that the underlying **assumptions** are rather more restrictive, though still quite broad.

In addition to the relatively broad assumptions about what the various runtime values are apt to be, the program includes a number of much more restrictive assumptions about the *accuracy* of its runtime model. If the values stored in the variables differ by even a small amount from the actual locations they represent, then the program will not work correctly. It is worthwhile to consider what can be done to reduce the accuracy required, since extreme precision may be rather expensive and difficult to attain.

2.3.6 Error Detection

Missing the Hole

Earlier, we noted that *a* simple close statement could overstrain the arm if the pin rack were bolted down off center. A similar difficulty can arise if the box is displaced from where its location variable says it is. If the error is big enough, then the pin tip will hit the top surface of the box, rather than go into the hole. Here, we cannot just add a simple compliance clause to the motion statement and expect things to work. We can, however, detect failure by monitoring force and stopping if a collision is detected:

```
in_hole_flag ← true; { Assume it will work }
move pin to in_hole_position
        via pin_withdraw,hole_approach
        on force(pin*zhat) > 8*oz do
                begin
                stop; { Stop the motion }
                in_hole_flag←false; { We lost }
                end
```

The force threshold of eight ounces is rather arbitrary; a certain amount of "tuning" may be required to get the best' value.

, Post-insertion Checks

This code assumes that successful pin insertion occurs if and only if the pin doesn't hit the top of the box. However, if the box is displaced far enough, the pin may miss it entirely. Since the force threshold isn't exceeded in that case, the fingers will open, dropping the pin on the floor. One way to avoid this problem would be to attempt small hand motions after , insertion and check for resistance. For instance,

```
move pin to pin*rot(x hat,10*deg)
        on torque(xhat) > 10*oz*inches do
                begin
                stop;
                in_hole_check←true;
                end
        on arrival do
                begin
                { If the motion goes all the way, we lost }
                in_hole_check←f alse;
                end;
```

Two objections to this check are that the extra motion statements take time and that the box may be moved inadvertently.

Always Stop on Force

Another possibility is to alter the insertion statement so that the successful insertion, as well as a near miss, will trigger a force monitor that stops the motion. Success and failure can then be distinguished by looking at how far the motion actually went.

```
move pin to in_hole_position+vector(0,0,-.3*inches)
        via pin_withdraw,hole_approach
        on force(pin*zhat)> 8*oz do
                stop;

distance_off ← zhat · inv(in_hole_position)*pin*vector(0,0,0);

if distance_off < -.2*inches then
        missed_box_flag ← true
else if distance-off > .2*inches then
        hit-top-flag ← true
else
        in_hole_flag ← true;
```

An additional advantage of this "plan to hit something" strategy is that it is much less vulnerable to small errors in the vertical position of the hole. If a fixed destination point had been used, and the hole were slightly higher than the runtime value said it was, then the forces produced as the arm tried to *servo* to the "nominal" position could become quite large. If the hole were slightly below nominal, then no real damage would be done for this particular task, since the pin would most likely drop into place when released. However, other tasks are not so forgiving. If we were inserting a screw, for instance, the initial insertion must bring the screw threads into contact with the threads in the hole." In such cases, it is much better to get a positive contact than to rely on brute force accuracy.

"Tapping"

An important requirement for using distance travelled along the hole axis as a success criterion is that the plane of the hole and the expected penetration distance be known well enough so that the various cases can be distinguished. In this case, there is no problem, since the pin goes in a considerable distance and the box sits firmly on the table. However, we may not always be so lucky. For example, the box might have been placed in a vise. Instead of aligning pins, we could be inserting *screws* that go in only a short distance before the threads engage. In such cases, it is sometimes possible to win by *deliberately* missing the hole on the first attempt and then using the result to tell where the box surface is. This might be done as follows:

---

[11] Actually, this is a slight oversimplification, since we would probably push down while driving the screw.

```
move pin to spot_on_surface+vector(0,0,-1.0*inches)
     via pin_withdraw,spot_on_surface+vector(0,0,1.0*inches)
     on force(pin*zhat) > 8*oz do stop
     on arrival do
          begin
          { This should never happen }
          abort("Help! Help! The box has been stolen");
          end;

correction ← that · inv(spot_on_surface)*pin*vector(0,0,0);

move pin to in_hole_position via hole_approach
     on force(pin*zhat) > 8*oz do stop;

distance-off ← zhat . inv(in_hole_position)*pin*vector(0,0,0) . correction;

{ et cetera }
```

Alternatively, one could use *correction* to make an appropriate modification to the box or hole location. For instance,

```
box ← box + vector(0,0,correction);
```

It is possible to take advantage of affixment to do away with the need for any explicit mention of *correction. For* instance,

```
affix spot-on-surface to box rigidly . . . ;
     :
{ move down until hit the spot }
move pin to spot_on_surface+vector(0,0,-1.0*inches)
     on force(pin*zhat) > 8*oz do stop;

{ Say that's where we got to }
spot-on-surface ← pin;
```

The rigid affixment asserts that whenever either frame is updated, the other is to be update& appropriately. Thus, the assignment statement will translate the box location to account for whatever distance the pin actually travelled. This technique is easy to write, since you don't have to invent variables or figure out complicated arithmetic expressions. Also, it is easy to read, since the code is terser, and the assignment statement more nearly reflects the "intent" of the motion statement, which *was* to get the pin to *spot-on-surface.*

2.3.2 Error Recovery

So far, we've been discussing ways for the program to discover that it has lost.[12] Once a

---

[12] An optimist would say "discover that it has won", but this is unjustified. There is bound to be at least one failure mode for which a program check has been left out. Even if it

failure has been detected, we must do something about it. The simplest course is to give up.

```
if not in_hole_flag then
    abort("Pin is not in hole.");
```

A somewhat more graceful termination might include some cleaning up to get ready for the next iteration.

```
if not in_hole_flag then
    begin { Put your toys away }
    move pin to pin_holder via pin-widthdraw;
        { We really should do some checking here, too }
    open bfingers to 1.0*inches;
    unfix blue from pin;
    move blue to bpark via pin_grasp_approach;
    abort("Pin is not in hole.");
    end;
```

In many cases, this is all that can be done. On the other hand, it would be nice if some degree of error recovery could be built into the program.

Searches

Even if the first attempt to find the hole misses, it is plausible to assume that it is somewhere near where the runtime model says it is. This suggests that we try searching the vicinity of our first attempt. The original AL design included a very complicated search construct for doing this. This construct has since dropped from sight; the desired effect can still be had by means of a loop, however:

---

were possible to anticipate and test for all failures, it would not necessarily be economical to do so.

```
if not in_hole_flag then
        begin
        vector dp;
        scalar n;
        dp ← vector(0.1*inches,0,0);
        for n ← J step J until 6 do
                begin
                dp ← rot(zhat,60*deg)*dp;
                { Try to put pin in perturbed hole }

                move pin to in_hole_position*dp+vector(0,0,-1.*inches)
                        via hole_approach*dp+vector(0,0,1*inches)
                        on force(pin*zhat) > 8*oz do stop;
                { Check distance travelled, etc. }
                        .
                if in_hole_flag then
                        n←7; { This terminates the search }
                end;

        if not in-hole-flag then
                abort("The hole doesn't seem to be there");
        end;
```

There are many variations possible on this theme, depending on how large an area is to be searched, what pattern is to be used, etc. If vision is available, we may want to use it to compute a correction for the next trial.

2.3.8 Refined Program

Combining a search loop with the other refinements and adding a check to be sure that the pin is successfully grasped, we get the following program:

```
begin "pin-in-hole"

frame pin, pin_grasp, pin_grasp_approach;
frame pin-holder, pin-withdraw;
frame hole, in_hole_position, hole_ap proach;
frame box;

affix pin-withdraw to pin-holder
        at trans(rot(zhat,30*deg),vector(0,0,4*cm));
pin-holder ← frame(nilrotn,vector(15*inches,10*inches,0));


affix pin-grasp to pin at trans(rot(xhat,180*deg),vector(0,0,2*cm));
affix pin-grasp-approach to pin-grasp at trans(nilrotn,vector(0,0,-3*cm));
pin ← pin_holder;
```

```
affix in_hole_position to hole rigidly
        at trans(nilrotn,vector(0,0,-1*cm));
affix hole_approach to hole at trans(nilrotn,vector(0,0,+1*cm));
affix hole to box at trans(nilrotn,vector(5*cm,4*cm,3*cm));
box ← initial_box_position;

{ Grasp the pin }
open bfingers to 1.0*inches;
move blue to pin-grasp via pin-grasp-approach;
center bfingers
        on opening < 0.1*inches do
                begin
                stop;
                abort("Grasp failed to pick up pin");
                end;
affix pin to blue;

{ Extract, transport, and insert }
move pin to in_hole_position+vector(0,0,-.3*inches)
        via pin_withdraw,hole_approach
        on force(pin*zhat)> 8*oz do
                stop;

distance-off ← zhat . inv(in_hole_position)*pin*vector(0,0,0);

if not ( 0.2*inches > distance-off >  ●  .2sirlches ) then
        b e g i n
        . vector dp;
        scalar n; boolean in_hole_flag;
        dp ← vector(0.1*inches,0,0);
        in_hole_flag ← false; n←0;
        while (n←n+1) ≤ 6 and not in-hole-flag do;
                begin
                dp ← rot(zhat,60*deg)*dp;

                { Try to put pin in perturbed hole }
                move pin to in_hole_position+dp+vector(0,0,-1.*inches)
                        via hole_approach+dp+vector(0,0,1*inches)
                        on force(pin*zhat) > 8*oz do stop:

                { Check distance travelled, etc. }
                distance-off ← that · inv(in_hole_position)*pin*vector(0,0,0);
                if 0.2*inches > distance_off > -0.2*inches then
                        in_hole_flag←true;
                end;

        if not in_hole_flag then
                abort("The hole doesn't seem to be there");
        end;
```

```
{ Let go of the pin }
open bfingers to 1.0*inches;
unfix pin from blue;
in_hole_position←pin; { Update our model }
move blue to bpatk via pin_grasp_approach;

end;
```

2.3.9 Further Discussion

Cost of Error Recovery

An important consideration in writing error recovery code, such as the loop above, is that it is' not always cheap. The amount of programming involved can frequently rival that required for the 'main" part – as, indeed, is the case here. If a useful purpose is served, this cost is generally unimportant, aside from Procrustean considerations.[13] More important is the extra time rquired in execution. The extra computer time spent in "head scratching" isn't likely to be an issue.[14] The time spent in manipulator motion is another matter. For instance, each iteration through the loop may take nearly as long as does the initial attempt. In an assembly line, this kind of delay can get very expensive, although some provision for buffering between stations can help to smooth things somewhat.

Fortunately, some forms of error recovery impose almost no additional manipulation cost. The principal example is the use of previous measurements to correct future behavior. For instance, suppose we are putting screws into all the holes in the box. As each screw is inserted, its location can be noted and used to update the value of *box*. Since the remaining hole locations are updated implicitly, the likelihood of having to search decreases with each screw. Vision is especially important in this regard, since the computations can be done in the background, in parallel with necessary motions. For example, suppose there is some chance that the pin may be misaligned in the fingers. If a picture is taken when the pin is removed from the rack, the actual pin-fingers relation can be computed during the time that the pin is being transported to the hole.[15] This correction can then be used to get the insertion right the first time.

---

[13] If the program won't fit into the runtime space available to it, then it is necessary to decide what to cut out. In many cases, the answer may be to get a larger machine. Computers are already cheap, compared to other components in a manipulator system, and are getting cheaper by a factor of ten every five years. This suggests that manipulator systems should be designed for easy expansion, since the marginal cost of going to a whole new system is considerably greater than expanding *a* pre-existing one.

[14] An exception is if something really hairy is contemplated. For instance, several systems to do "problem solving" to figure out how to correct errors have been proposed. See [12]. Sproull [25] has investigated the question of when runtime planning is cost-effective.

[15] Bolles [7] is currently investigating techniques for accomplishing exactly this kind of task. Although, his system isn't quite up to the real time requirements described here, his results indicate that the task could be performed with essentially the present hardware, provided that someone wanted to do the necessary programming on the runtime machine.

"If we could first know *where* we are and *whither we* are tending, we could then better judge *what* to do, and *how* to do it."

Abraham Lincoln
Speech before the Illinois Republican Convention
June 16, 1858

Chapter 3.

PLANNING MODELS

3.1 Introductory Remarks

This chapter explores the relation of planning information to programming, in general, and to manipulator programming, in particular.

Programming is a form of planning; the essential quality of a computer program is that it is a *prior* specification of how the general capabilities of the machine are to be applied to a specific problem. Every program embodies some assumptions about the special circumstances in which it will be executed. Thus, an inherent part of the programming process is the maintenance of information about the predicted execution-time environment, and the use of such information as a basis for programming decisions. Indeed, the intellectual burden of maintaining such a *planning model* is one of the major factors in determining the effectiveness of a particular programming formalism, when applied to a task domain. This burden cannot be escaped; if we wish to help the programmer by taking over some of the coding effort, then the computer must keep track to the information relevant to the coding decisions it is asked to make.

5.2 Planning Information in Algorithmic Languages

3.2.1 An ALGOL Fragment

Consider the ALGOL fragment below, which is intended to select the largest element from an unsorted array, *a*.

```
integer array a[1:100];
integer i,n,maxel;
        :
maxel ← -2^35; { largest negative number in machine }
{ Assume we want the maximum of the first n elements of a. }
for i ← 1 step 1 until n do
        if maxel<a[i] then maxel←a[i];
```

When we write the statement in the loop body, we know that variable i will contain a value between 1 and n, that $maxel \leq a[j]$ for $1 \leq j < i$, that $maxel = a[j]$ for at least one j in that range, and that, by the time the loop has exited, we will have examined all values for i from 1 to n. Further we assume $n \leq 100$.[1] A process of great interest to researchers intent on proving the correctness of programs has been the formalization of these assertions and the use of well-formulated language semantics to prove the assumptions correct [I 1, 27, 1]. Similarly, one of the strongest claims of 'structured programming" advocates is that one should proceed from such assertions to a "correct" program [9]. There has been a great deal of interest in applying theorem proving methods to automate the generation of programs from assertions. [e.g., 161.

My own impression is that one does not, usually, write programs in such a step by step fashion. Rather than working out from first principles how to synthesize this loop to compute a maximum element, most programmers would reach into a grab-bag of tricks, pull out a skeleton program structure, and then fill in the appropriate slots.[2] To some extent, the program is thus composed of "higher-level" chunks, with the programmer acting in a dual role as a problem solver and coder (translating between the conceptual units in which the program was composed and those made available by the programming system).

Planning information is used at both levels. For instance, the fact a is an unsorted array or that the loop sets $maxel$ to the maximum element of $a[1:n]$ would be typical "high level" facts useful primarily in performing the problem-solving function. Coding information includes the fact that i is available for use as an index variable, that a is the name of the array to be searched, etc.

---

[1] Several people have commented that the loop should be written, $maxel \leftarrow a[1]$; for $i \leftarrow 2$ step 1 until n do .... It is interesting to note that this form is equivalent only if $n \geq 1$. In other words, we can make a marginal improvement in program performance if we have an additional piece of planning information.

[2] Program bugs happen when some precondition for using the trick is forgotten. (E.g., i might be in use for some other purpose). It is not necessary to accept the psychological validity of this paragraph in order to appreciate the main point: that much coding can be done by adaptation of standard "skeletons" to fit particular situations.

3.2.2 Getting the Computer Involved

A dominant theme in the history of programming system development is the progressive transfer to the computer of coding responsibilities. The nature of coding is largely clerical. One keeps track of particular facts and applies them in a stylized manner. As elements of programming practice become better understood or, at least, better formalized, this process has been extended into areas of increasing abstraction.

Thus, symbolic assemblers feature the ability to keep track of addresses, maintain a literal table, etc., providing a substantial improvement over 'octal" or "push the switches" programming. Similarly, algebraic compilers perform many functions of an assembly language coder. They keep track of information like assignment of variables to registers, where temporary results are stored, etc., and follow highly stylized (though sometimes extensive) rules to generate programs that are "equivalent"@ their input specifications.

There are several important points concerning such "automatic. coding" systems: First, they use their "understanding" of the formal semantics of-the source language and of their own decisions (e.g., to keep *maxel* in register 1) to keep track of those facts that are appropriate to its task as an assembly language coder. Second, optimizing compilers produce more efficient output code than non-optimizing compilers can, but they must keep track of more information to do it. Third, there are limits imposed by what can be slated explicitly in the source language. In general, it is much more difficult to "infer" the intent of a particular piece of code than to write code to achieve a particular purpose. The computer has no "understanding" that our loop is intended to compute the maximum element of *a*. It could not, for instance, decide (because of some earlier code) that *a* is sorted and compile the loop as though it were

```
maxel ← a[1];
```

On the other hand, if the user's program were expressed in terms of concepts like "sort array *a*", and "select the maximum element", then the computer might, in fact be able to write the appropriate code. Recently, there has been a great deal of interest in "very high' . level" languages, in which programs are expressed in exactly this fashion. [e.g., 2, 13, 15, 26, 3].

Occasionally a user map wish to share coding responsibility with the programming system. Fdr instance, he may wish to "hand-code" the inner loops of an ALGOL program, in the belief (however deluded) that he can do a better job. This creates certain difficulties for the compiler, which generally only really "understands" code that it has written itself, and there . has been a tendency among language designers (especially those wishing to enforce particular programming methodologies) to outlaw such tampering. An alternative would be to provide constructs that allow the user to tell the system about relevant assumptions or effects for a particular piece of code, such registers used to contain the results of machine language statements.[3]

---

[3] Another possibility, investigated by Samet [24] for LISP progams, is to write both "high level" and hand-coded versions of the same program. The system can then verify that both programs are, indeed, equivalent, even though it isn't necessarily clever enough to figure out the hand-coded version on its own.

This sharing of coding responsibility is especially important early in the evolution of an automatic coding system, when many things cannot yet be handled by the computer. In Chapter 4, we describe procedures for making the AL coding decisions of Section 2.3 automatically. Incorporation of this facility into a manipulator programming system requires either that enough primitives be available so that *all* manipulator-level coding decisions can be made by the system, or that coding be shared, perhaps by having the computer generate program text for subsequent modelling by the user. Again, some assertional mechanism is almost certainly necessary to help the system 'understand" code written for it by the user.

3.3 ● Planning Information in Manipulator Programming

Many of the book-keeping requirements of manipulator programming are essentially the same as those for "algebraic" programming. One must keep track of what variables mean, what things are initialized, what control structures do, etc.

In addition to these general requirements, the domain requires the maintenance of information particular to the problems of manipulation. This information may be divided, roughly, into the following categories:

1. Descriptive information about the objects being manipulated.
2. Situational information about the execution-time environment.
3. Action information defining the task and semantics of the manipulator language.

Subsequent sections discuss these issues in greater detail.

3.4 Object Models

Programs which specify explicitly what actions are to be performed by the manipulator generally need contain little explicit description of the objects being manipulated. In the AL program developed in Section 2.3, for instance, there is no information about the shape of the pin, hole, or anything else. The principal language construct for describing objects is the **affix** statement, which is used to specify how the location of an object is related to the location of its subparts or features. For instance,

```
affix hole to box at trans(rot(xhat,90*deg),vector(2.4,1.3,3.2));
```

On the other hand, a great many assumptions about the objects have been built into the program. For instance, the check used to verify that the pin has been grasped successfully relies on knowledge of the pin diameter; the extraction, grasping, and insertion positions implicitly assert that the hand or pin will not crash into anything; the insertion strategy assumes that the pin will accommodate to the hole somewhat, that misses will cause the pin to hit a surface coplanar with the hole or else miss the object altogether; and so on.

These assumptions do not get built into programs by accident. Information about objects is used extensively in both the 'problem solving" and coding functions involved in manipulator programming. In mechanical assembly programs, the task is largely *defined* by the design of the object being put together. In addition to specifying what is to be done, the design also dictates many aspects of how to do it, such as in what order the various parts must be assembled, how the parts can be grasped by the hand (or put in a fixture), what motions are required while mating parts, and so forth.

For manipulator programming, the most important aspects of object descriptions derive from the shape of the objects being manipulated. Unfortunately, good shape representations for computer use have yet to be developed. Many decisions that are intuitively obvious to a human programmer require a laborious computation by the computer. On the other hand, it is possible to identify many "local" properties that play an important role in coding decisions. For instance, in coding the pin-in-hole example of Section 2.3, we used object information in a number of ways:

1. Filling in parameters. The most obvious example is the location of the hole with respect to its parent object:

   affix *hole* to box at **trans(nilrotn,vector(*3.8*cm,3.2*cm,4.9*cm))**;

   Other uses include setting the minimum grasp threshold, the expected penetration of the pin into the hole, and selection of a grasp point that kept the fingers out of the way.

2. Estimating the accuracy required to guarantee that the pin will seat properly in the hole. The allowable error is determined by such factors as the point on the pin, chamfering around the hole, clearance between the pin shaft and hole bore, etc. It is Important in deciding whether the insertion method used here will work and in setting the "step size" for the search loop.

The object representations used in this work are described in my dissertation [28]. It is important to note, here, that these uses predominantly involve local properties of features (e.g., the chamfering around a hole, or the placement of holes in a surface) that are, in principle, *computable* from a uniform shape representation, but which may also be represented directly in several different forms to serve different purposes.

3.5 Situational Information

Manipulation programs transform their environment by moving objects around. This means that the principal fluent[4] properties that must be considered are:

1. Where objects are in the work station.

2. -What objects are attached to each other.

---

[4] By "fluent" properties, we mean any factors relevant to the task which may not remain constant during execution of the task.

3. How accurately relevant locations are known by the manipulation system.

The use of this information was illustrated in our discussion of the pin-in-hole example. Among the more important considerations were:

1. We made a number of unstated, though "obvious", assumptions about the location of the various entities. *For* example, the hole was assumed to be unobstructed (i.e., the box better be right side up).

2. In grasping the pin, we had to consider whether the hand could reach the required locations. If it is possible for the box to be in more than one position or orientation, then this must be taken into account.

3. We made use of the fact that the pin could be attached temporarily to the hand by grasping. Similarly, it is important to realize that a subsequent motion of the box will cause the pin to move, too.

4. The code contains many assumptions about the accuracy of our variables *pin* and hole. In deciding whether "tapping" or a search were necessary, for instance, it is necessary to consider whether the "along-axis" determination is good enough and whether in-plane errors are within the "capture" radius required by the pin to hole geometry.

Any reasonably sophisticated maniputator language allows much of this information to be represented explicitly in the program. In AL, for instance, object locations are represented by frame variables and attachments, by affixments. In writing programs, it is thus necessary. to keep track of programming information, such as what variables have been declared and what calculations have been performed, and to relate this information to the physical reality being modelled. It does little good to know that, once we have closed the fingers on the pin, it will move when the hand does, unless that information is reflected in the program by a corresponding affix statement:

3.6 Action Information

Clearly, it is necessary to understand the semantics of the manipulator language in order to write programs in it.[5] This is essential both for translating desired manipulator actions into the corresponding code and for keeping track of situational information.

Earlier, we described the coding component of programming as adapting previously defined code skeletons to fit particular facts. This occurs in manipulator programming to a surprising extent. For instance, the "grasping" sequence of our pin-in-hole example is readily adapted to pick up more or less arbitrary objects.

---

[5] Of course, this knowledge does not have to be perfect. There are those whose approach to programming is empirical, to say the least. Even where a certain amount of experimentation is attempted, however, one generally requires at least an approximate model of what a particular statement is supposed to mean.

```
open blue to initial-opening;
move blue to object_grasp
        via object_grasp*trans(nilrotn,vector(0,0,-4*inches);
center blue
        on opening < minimum-opening do
                begin
                stop;
                abort("It just isn't there!");
                end;
move object to object_pickup_point;
```

The slots to *fill* in are *initial_opening*, *minimum-opening*, *object_grasp*, and *object_pickup_point*. As we will see in Chapter 4, these may be computed from the situational and object modelling information.

3.7 Concluding Remarks

This chapter has discussed the role of planning information in programming and has described the particular kinds of information that are needed for manipulator programming. A key point, here, is that the burden of maintaining this information cannot be escaped. If manipulator programs are to be generated automatically, then the planning information must represented in a form that the computer can use to make reasonable decisions.

A full discussion of the representation methods used by the automatic coding procedures described in Chapter 4 may be found in my dissertation [28]. To make this report self-contained, a short summary of the most important technical results is given below.

The AL Planning Model

The AL compiler itself performs a number of coding functions, such as planning trajectories and rewriting motion statements, not ordinarily found in algorithmic languages. These functions require that the compiler keep a better model of situational information – especially, the expected value of frame variables and affixments – than might otherwise be the case. The compiler associates a data base of assertional "forms" with each control point in the program graph, and uses simple simulation rules to propagate facts. The same mechanism – a multiple world assertional data base – is used by the automatic coding procedures discussed in Chapter 4 to keep track of situational information.

Object Information

Object modeling is done by "attribute graphs", in which shape information is represented in the nodes, structural information by links, and location information by properties of the links. The most interesting point is that coding decisions can generally be based on "local" properties of the object.

Situational Information

In order for the computer to make reasonable coding decisions it must often have *numerical* estimates of object locations and of how accurately those locations will be known at execution time. Techniques were developed for expressing "semantic" relations between object features in terms of mathematical constraints on scalar "degrees of freedom" and for applying linear programming techniques to predict limits in inter-object relationships. Differential approximation methods were also developed and used to predict errors. The appendix to this paper gives examples of both techniques.

Action Informrtion

In. the system described in this report, action. information is not represented explicitly. Instead, it is embedded implicitly in the procedures that make the coding decisions and generate the output programs, as we will see in the next chapter.

*Watch me pull a rabbit out of my hat!"

Bullwinkle Moose

Chapter 4.

AUTOMATIC CODING OF PROGRAM ELEMENTS

In Section 2.3, we described the process of writing AL code for a common **subtask** in assembly operations -- insertion of a pin into a hole. We saw that writing the program **required** *a* number of decisions, based on our expectation of where the objects will be and how accurately their positions will be determined at **runtime.** The discussion of Chapter 2 focused on the modeling requirements for automatic coding. The key point was that automation of coding decisions requires that the necessary information be represented in a **form** usable by the computer. This chapter describes the use of the computer's planning model to make these decisions automatically.

The program outline followed is essentially that derived in Section **2.3:**

1. Grasp the pin.

2. Extract it from the pin rack and transport it to the hole via *a* point **just "above"** the hole.

3. Attempt insertion by moving the pin along the axis of the hole until *a* resisting force is encountered. Use the distance **travelled** to determine whether or not the pin insertion is successful.

4. If the insertion is unsuccessful, then use a local search to attempt to correct the error.

The decisions that must be made include:

1. Where to grasp the pin.

2. How to approach the hole. Although we have decided on a co-axial approach, we still must decide the relative rotation of the pin and hole frames.

3. What threshold values to use on our success test. Also, whether or not it is necessary to "tap" the object surface to get a better determination of the pin-hole relation before trying the insertion.

4. What search pattern,' if any, to use in error recovery. ,

The overall approach is fairly direct: First a number of preliminary calculations are performed, based on the task specification and initial planning model, to obtain initial position and accuracy estimates and to determine basic tolerances. Then, the system generates possible ways to grasp the pin, subject to geometric feasibility constraints. For each distinct grasping strategy, a "best" approach symmetry for the pin relative to the hole is computed, using expected motion time as an objective function.' The grasp-approach. pairs are sorted by "goodness" and. then are reconsidered in "best first" order, to see what additional refinements are required, based on the estimated pin-to-hole determination. If the error along the hole axis is too large, then a "tapping place" is found as near the hole as is safely possible. Similarly, if the errors in the plane of the hole are too great, then a decision to search is made. The expected time required for tapping and search are calculated and added to the cost. The process is continued until an optimal strategy can be chosen. Once the decisions have been made, it is fairly straightforward to generate the corresponding AL code sequences, which are quite stylized.

Subsequent sections will describe each of these phases in greater detail.

4.1 Data Structures

Internally, strategies are tepresented by SAIL record structures summarizing the decisions that have been made. This section describes the more important parameters kept for pin-in-hole and pickup strategies.[2]

Pin-in-Hole Strategy

prelim in aries – A list of "preliminary" actions that must be performed before the code for the actual pin-in-hole code is begun. Typically, this involves cleanup actions left *over* from the previous task, and is set up by the initial processing.

pickup - A "pickup strategy" to get the pin affixed to the hand and free of any obstructions. For picking up an object by grasping it in the fingers, this field would point to a grasp strategy", defined below.

dtty – The distance into the hole that we will try to poke the pin.

standoff – The distance above the hole that we will place an approach point.

ϕ – The relative rotation of the pin to the hole upon insertion.

---

[1] The combination of grasping method, approach symmetry, and expected penetration distance into the hole constitutes sufficient information to write a "first order" program that ignores errors, such as was produced in Section 2.3.4. However, as we saw earlier, the job isn't yet half done.

[2] The structures shown here are slightly different from those actually kept. The changes have been made for ease of explanation; the information content is the same. Section 4.8 includes a computer generated summary of the actual internal structures. You have been warned, so don't get confused.

tapping place – Point on the object to be 'tapped" to reduce the error along the hole axis, if necessary.

Δx, Δy, Δz, ξ - Parameters summarizing the "error" footprint of the pin, tip with respect to the hole. Define a rectangular **parallelepiped** with sides **(Δx, Δy, Δz),** rotated by **ξ** about the hole axis. See Figure 4.1.

A6 - Maximum expected tilt *error* for the pin axis with respect to the hole axis.

ttime – Expected time spent in grasping the pin and transporting it to the hole.

finetime – Time expected to be spent in "fine adjustment" motions. Currently, time for tapping motion + search time.

goodness – Estimated cost of this strategy. Here, **ttime + finetime + goodness(pickup).**

Grasp Strategy

object – The object to be grasped.

preliminaries - As before, a list of preliminary actions that must be performed before the object (here, a pin) can be grasped. A typical element would be code to put down a tool.

grasp point – Point where object is to be grasped. The structure used to specify such "destination points" is discussed below.

approach point - Via point on the way to the grasp point.

approach opening – Required opening for the fingers by the time the hand gets to the approach point.

grasp opening – Minimum expected opening for the hand to hold the object.

grasp determ - An estimate of the accuracy with which the object will be held by the hand, once the grasping operation is successfully completed.

departure point – Via point through which pickup-and-move operation must pass.

goodness – Measure of the cost of this pickup strategy. Typically, an estimate of the amount of time required.

Destination Points

Destination points in AL motion statements really involve two.components:

1. A frame-valued expression specifying some location in the work station.

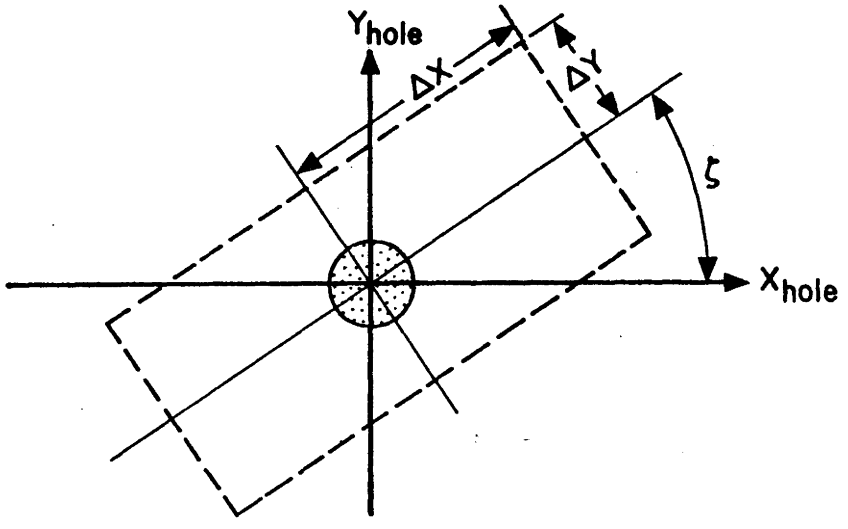


Figure 4.1. Error Footprint

2. A 'controllable" frame variable whose value is to be made to coincide with the target value.

Thus,

move *a* **to** *b*;

is, in some sense, a manipulatory equivalent to the assignment statement

a ← *b*;

For our present purposes, it will be sufficient to restrict the "right hand side" component of all destination points to the form

<object or feature name>+<constant trans expression>

Thus, the data associated with each destination point consists of:

whrt – The object or feature which is to furnish the controllable frame.

base – Object or feature for target expression.

xf – Constant trans for target expression.

Section 4.7 describes how sequences of such "destination points" may be turned into motion statements.

4.2 Initial Coinputations

The most important initial computations are those responsible for calculating the expected initial positions and error determinations of the pin and hole. Following the methods developed in my dissertation [28], we get

H = estimated position of hole (with respect to work station)
$= H(\lambda_1,\ldots,\lambda_k) = H(\bar{\lambda})$

P_{init} = estimated initial position of the pin.
$= P_{init}(\bar{\mu})$

ΔH = estimated accuracy of H at runtime.[3]
$= \Delta H(\bar{\delta})$

ΔP_{init} = estimated runtime accuracy of P.
$= \Delta P_{init}(\bar{\epsilon})$

subject to constraints on $\bar{\lambda}$, $\bar{\mu}$, $\bar{\delta}$, and $\bar{\epsilon}$. For planning purposes, we will mainly deal with the expected locations

$H^0 = H(0)$
$P^0_{init} = P_{init}(0)$

Also, we need several important paramters describing how the pin fits into the hole:

direction – The end of the pin which is to be inserted into the hole.

d_f – The distance the pin is to go into the hole.

d_s – The maximum "sticking distance" into the hole that the pin can "jam" without making it all the way to where it is supposed to go. Thus, d_f-d_s represents a minimum threshold for telling whether the pin insertion is successful.

---

[3] Here, we are being a bit sloppy in our use of "H". The difficulty is that we must deal with three separate entities representing the hole: (1) the object model representation (a LEAP item); (2) our location estimation; and. (3) a variable in the output program. Generally, this discussion will center on (I) and (2); (3) isn't needed until time comes to generate the actual program text.

$\Delta\theta_{ok}$ – Maximum possible axis misalignment for the pin insertion to succeed.

Δr_{ok} – Maximum possible radius misalignment between the pin and hole for the insertion to succeed. Thus, $\Delta\theta_{ok}$ and Δr_{ok} constitute a measure of the effectiveness of accommodation during insertion.

The direction must be supplied by the user as part of the task description.[4] In principle, df and d_s may be computed by looking at the profiles of the pin and hole. At one time, this was done. However, the computation turned out to be extremely tedious, and ignored some important factors, such as friction.[5] Therefore, these numbers were determined by experimenting with the actual objects and included in the object models, as were Δr_{ok} and $\Delta\theta_{ok}$. This approach does not seem unreasonable, since pins and holes may be standardized. Presumably, a data base could be built containing the relevant parameters for each tip-hole combination encountered in a class-of assemblies.

4.3 Grasping the Pin

Once the initial computations have been done, we proceed to generate alternative strategies for picking up the pin. For each such strategy, we create a "grasp strategy" record, as described in Section 4.1. Although we confine ourselves to grasping the pin directly between the 'fingers, it is interesting to note that alternative methods, such as loading a screw onto the end of a screwdriver, could be handled similarly. The rest of the pin-in-hole code (except for the part about "letting go") makes no assumptions about what the hand actually holds onto. The important data used by the rest of the planning arc:

1. The relative position of the pin to the hand.
2. The accuracy with'which the pin is held with respect to the hand.

So tong as this information is available, the remaining decisions can proceed more or less in ignorance of the actual technique used.

---

[4] This may not be strictly necessary. If the pin is to be part of a finished assembly, then the direction and df may be obtained from the description of the object being assembled. Alternatively, it might be possible to tell which end to use by looking at the pin and hole diameters or to keep a data base telling what the "standard" direction for each pin type.

[5] Whitney [18] has done an exhaustive analysis of some of the factors required to compute tolerance requirements for insertion of a peg into a hole.

4.3.1 Assumptions

The grasping method described in this section assumes that the pin initially sits in a hole and that the hand is empty. The basic strategy is to open the fingers, move the hand to the grasping position, and center the hand on the pin. Thus, we require that there be at least one grasping position reachable by the manipulator and that the pin's position be known with sufficient accuracy for the centering operation to succeed. Once the pin is grasped, it is extracted from the hole. We assume that the pin will be free of obstructions once its tip has cleared the plane of the hole by some fixed amount (currently, 1 inch).

4.3.2 Grasping Position

The key element in our grasping strategy is where to grasp the pin. The present hand consists of a pair of opposed "fingers", which open and close through a range of about 4.5 inches. On each finger is a circular rubber pad, and in the middle of each pad is a microswitch "touch sensor". The AL center command assumes that the object being grasped will trigger the touch sensors whenever it is in contact with one of the fingers. Since we intend to use center, the finger pads must be centered on the pin shaft? The important parameters remaining are thus:

γ – The "grasp angle" between the pin axis and the approach vector $(\hat{z})$ of the hand.

d – The "grasp distance" along the pin axis.

ω – The orientation of the approach vector of the hand about the pin axis.

Following the convention that the "long" axis of pins is the z-axis, this means that the grasping position will be given by

blue = *pin***grasp_xf* = *pin**trans(rot(zhat,ω)*rot(xhat,γ),vector(0,0,*d*));

Geometric Considerations

In selecting values for these parameters, it is important to guarantee that the hand not get in the way of accomplishing the task. In general, this might require much better geometric modelling capabilities than the system described here currently possesses. Therefore, we must assume a relatively "uncluttered" environment. The following considerations are, however, enforced by the present implementation:

1. The hand cannot intersect the body in which the hole is drilled. As an approximation, we enforce this constraint with two sub-constraints for both the initial and target holes:

---

[6] When sensitive force sensors are added to the fingers, center will presumably be modified to respond to forces on the fingers, rather than triggering of a microswitch. This would allow greater freedom in picking finger positions and would relax the accuracy requiremen ts.

a. The fingers may not pass below the plane of the hole.

b. The hand's approach direction must be *at least* 90 degrees to the outward facing normal to the hole.

2. The "palm" of the hand cannot intersect the shaft of the pin.

Method

It is necessary to verify that arm solutions exist for the approach, grasp, and liftoff points. However, the computation required by the arm solution procedure is non-trivial. Thus, we proceed by pretending that the hand is moved by levitation. Arm solutions will only be attempted for those grasping positions that do not try to do'something bad with the hand. If we assume that conditions (a) and (b) above are sufficient to guarantee that the hand stays clear of any objects, then we can ignore ω in selecting grasping positions to consider. Our overall selection method looks like this:

1. Use the position of the pin in the initial and target holes to determine legal limits on γ and d.

2. Use the limits established in step 1 to generate "significantly" distinct values for γ and d. For each such (γ,d) pair, determine values of ω for which there is an arm solution.[7] Each (γ,d,ω) will then specify a possible grasping position for the pin.

3. Once a grasping position has been generated, the remaining parameters to the grasping strategy may be filled in, and the cost of the strategy assessed. This process may result in some of the proposed grasping positions being rejected, due to inability to find a suitable approach or departure position or because of accuracy considerations.

These steps are discussed in somewhat greater detail below.

Determining values for γ and d

To simplify the discussion, let us assume that the pin initially has its z-axis parallel with its starting hole, and that the origin of the pin's coordinate system is at the pin tip inserted into the hole.[8] The first step in determining γ and d is to determine the distances, di and d_f, that the pin goes into the initial and final holes. There are two subcases:

1. The same end of the pin is inserted in both holes.

---

[7] If additional feasibility tests are to be made, this would be a good place to include them. For instance, if good enough shape models (e.g., those produced by CEOMED [4]) are available, then a check can be made to see if the hand or arm do, in fact, interfere with objects in the environment. Two problems with this check are (1) the difficulty, of distinguishing intersections caused by approximations and those caused by actual collisions and (2) the difficulty of modelling *sets* of possible positions.

[8] If the initial hole and pin axes *are* anti-parallel, the modifications required are obvious.

2. Opposite ends of the pin are inserted in the initial and final holes. I.e., we must *turn over" the pin while transporting it from hole to hole.

In the first case, the lower bound on d will be given by

$$d \geq d_{min} = \max(d_i, d_f) + r_{fp} + \kappa$$

where

r_{fp} = radius of finger tips.
κ = a small extra clearance factor (currently 0.1 inch)

To compute the upper bound, d_{max}, we must consider the pin geometry; if the pin has a painted tip, then we must grasp further down the shaft:

$$d \leq d_{max} = l_{pin} - (l_{taper} + \kappa)$$

where

l_{pin} = length of pin
l_{taper} = length of point on pin tip

If the interval d_{max}-d_{min} is relatively short (currently, less than 2.5 inches), then we just pick the midpoint

$$d \leftarrow (d_{min} + d_{max})/2$$

Otherwise, a succession of values must be considered. Currently, *three values are considered: one near the top of the pin, one near the bottom, and one in the middle.

d_1 = dmax - 0.6 inches
d2 = d_{min} + 0.6 inches
$d_3 = (d_{min} + d_{max})/2$

. For each value of d, the system must generate values for γ. Similarly, three approach directions are considered:

γ = 180 degrees (i.e. anti-parallel to the pin axis)
γ = 135 degrees
γ = 100 degrees (i.e., approximately perpendicular to the axis)[9]

, With γ = 180 degrees, it is necessary to check that the pin doesn't poke up through the palm of the hand. This is easily handled by checking to be sure that l_{pin}-d is less than the length of the fingers.

---

[9] 90 degrees could be used here; however, the extra 10 degrees lessens the chance that the hand or wrist will interfere with something.

In the second case, where both ends of the pin will go into holes, we have

$$l_{pin}-(d_f+r_{fp}+\kappa) \geq d \geq di + r_{fp} + \kappa$$

Again, if the interval is short, its midpoint will be picked. If the interval is longer, then three values will. be used. Since the pin must be turned around, the only value for 7 is 90 degrees.

Picking Values for ω

Once we have picked values for γ and d, we still must determine the rotation value ω. Here it is necessary to consider actual arm solutions. Unfortunately, the only way presently available for doing this is to invent values and try them out.[10] Values of ω are considered in increments of 45 degrees. For each value, the grasping position is calculated, and the arm solution procedure is called to see if the position is feasible. In some cases, we may produce a great number of candidate grasping positions. Therefore, the solutions for all feasible positions are graded for "toughness" and non-degeneracy, 'and only the best few values are retained for further investigation. The current rule for evaluating arm solutions is very crude: the angle of the "elbow" (joint 5 of the Scheinman arm) is examined; Angles near 45 degrees are considered best.[11] Our selection procedure looks something like this:

```
for ω ← 0 step 45*deg until 315*deg do
    begin trans hand_place,grasp_xf;
    grasp_xf ← trans(rot(zhat,ω)*rot(xhat,γ),vector(0,0,d));
    hand-place ← initial_pin_location*grasp_xf;
    if solve_arm(hand_place) then
        begin
        cost ← abs(45*deg-joint_angle[5 ]);
        << insert ω into list of candidates, ranked by cost >>
        :
        end;
end;
```

For the example situation described in Section 4.8, and grasping parameters:

---

[10] Shimano is currently investigating the possibility of a "closed form" solution that will give the range of possible approach orientations for a given hand position. Such a solution would be extremely useful, both as a guide for selecting grasping positions and as a means for evaluatingthe robustness of a particular position under variations in object position.

[11] Alternatives include examining the error hypercube at the fingers or just using the expected time to reach the grasping position. The latter objective function will eventually be applied to any points that get through this filter (see Section 4.4).

γ = 135 degrees
d = 3.54 cm

we get:

ω	cost
0°	42.1°
45°	9.98°
90°	31.4°
135°	56.0°
180°	56.0'
225°	56.0°
270°	45.5°
315°	56.0°

At present, only the best three values are retained, so we will select ω = 45°, 90°, and 0°. This pruning introduces some risk that the program will fail to find an acceptable strategy in some cases where it might otherwise have won. If this problem should become significant; it would be fairly easy to provide a "try harder" mode *where* all possibilities are retained.

4.3.3 Approach and Departure Positions

The purpose of an approach point for the grasping operation is to prevent the arm from trying to run its fingers through the pin. Currently, the only approach direction considered is one along the approach vector of the hand, as shown in Figure 4.2. One plausible alternative would be to move to a point above the pin and then move down along the pin axis to the grasping position. If it should prove desirable to consider such alternatives, we could do so by planning each route and then selecting the via point which gives the shortest time.

Similarly, a departure point is needed to get the pin clear of its initial hole before trying to move it away, We presently only use a standard takeoff point two inches above the hole.

move *pin* to *pin**trans(nilrotn,vector(0,0,2*inches*d_i));

where d_i is the distance the pin is inserted into its starting hole. If this fixed choice should *ever* become troublesome, it would be fairly easy to generate a set of alternative departure points, and then pick the one giving the shortest motion time.

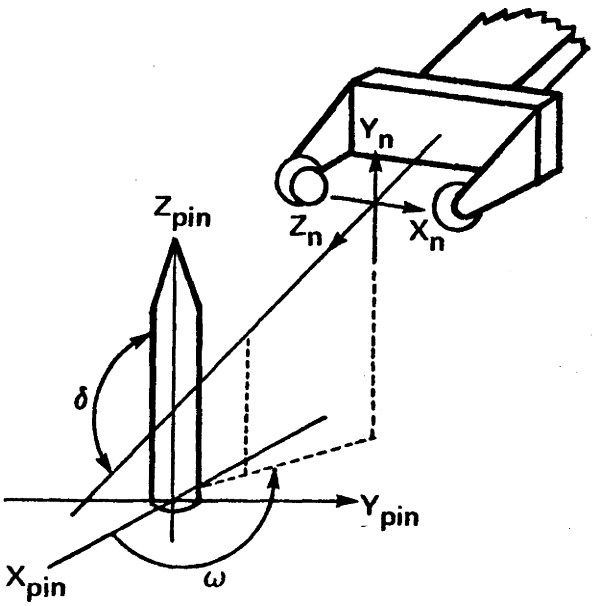


Figure 4.2. Approaching the Pin

4.3.4 Hand Openings

The present decision for hand opening is similarly arbitrary. On approach the hand is opened by 1 inch plus the diameter of the pin at the grasp point. The closure threshold is set to the pin diameter minus 0.1 inch.[12]

4.4 Moving to the Hole

Once the pin has been grasped and lifted clear of its initial hole, the next step is to try inserting it into the target hole. For the sake of simplicity, we assume that the origin of the pin coordinate system is at the tip being inserted into the hole. Thus, our motion statement will look something like:

```
move pin to hole*trans(rot(zhat,φ),vector(0,0,-dtry))
    via hole*trans(rot(zhat,φ),vector(0,0,standoff))
    on force(pin*zhat)>δ*oz do . . .
```

where

---

[12] This latter figure comes from the observed behavior of the center primitive; relevant factors include flexion of the fingers and compression of the Anger pads.

ϕ = rotation angle of pin with respect to hole.

dtry = distance try to push pin into hole.

standoff = distance of approach point from the plane of the hole.

Of these parameters, the most important is ϕ. The considerations in choosing a good value are essentially the same as for selection of the grasping orientation, ω. The method followed is also the same, except that a single value of ϕ is picked to minimize the expected motion time and the destination location is used instead of the initial pin location. Thus, the expected final position of the hand will be given by:

hand_destination = *pin_destination***grasp_xf*
. *hole**trans(rot(zhat,ϕ,vector(0,0,-d_f))**grasp_xf*;

For our example situation (Section 4.8) and grasping-parameters:

γ = 135 degrees
ω = 90 degrees
d = 3.54 cm

we get

ϕ	time
0°	.960 sec
45°	.491 sec
90°	.468 sec
135°	.582 sec
225°	1.21 sec
270°	1.54 sec
315°	1.67 sec

ϕ = 90 degrees will therefore be chosen;

The exact values of *dtry* and *standoff* are less important. The principal constraint is that they be large enough to guarantee that location errors in the hole (or pin) will not cause the motion to stop prematurely or to knock the pin into the object while approaching the approach point. Currently, arbitrary values,

dtry = df + 1 inch
standoff = 1 inch

are used. Thus, for this case, our destination approach and target locations will be, respectively:

```
pin = hole*trans(rot(zhat,90*deg),vector(0,0,2.54));
pin = hole*trans(rot(zhat,90*deg),vector(0,0,-4.25));
```

When values for ϕ, *dtry*, and *standoff* have been picked, they are combined with the grasping strategy to form an embryo "pin-in-hole" strategy. The expected time to execute it is just the time expected for the pickup operation plus the time for moving to the hole.

4.5 Accuracy Refinements

In the absence of errors, the 'strategies derived in the previous section. would suffice to accomplish the task. Unfortunately, the world is not so kind, and we must consider the effects of errors. For each strategy, we apply the machinery given in my dissertation [28] and illustrated in the appendix to estimate the error between the pin and hole at the approach point as a function of free variables:

$$\Delta p_{hp} = \Sigma \delta_k p^k_{hp} \qquad \text{[Eqn 3.5.1]}$$
$$\Delta R_{hp} = I + \Sigma \epsilon_k M^k_{hp}$$

subject to constraints

$$c_j(\bar{\delta},\bar{\epsilon}) ? b_j$$

on the free variables. We are principally interested in three things:

1. Axis misalignment ($\Delta\theta$) between the pin and hole.
2. Displacement error (At) along the axis of the hole.
3. Displacement *errors* ($\Delta x, \Delta y$) in the plane of the hole.

Each of these entities is discussed below.

4.5.1 Axis Misalignment

For suitably small values, $\Delta\theta$ may be approximated by

$$\Delta\theta \simeq \hat{y}\cdot\Delta R_{hp}\hat{z} + \hat{x}\cdot\Delta R_{hp}\hat{z}$$

Thus, we can use [Eqn 3.5.1] to compute the maximum expected misalignment.

$$\Delta\theta_{max} = \max |\Delta\theta_i|$$

where

$$\Delta\theta_i = \max | \text{vector}(\cos\gamma_i, \sin\gamma_i, 0)\cdot\Delta R_{hp}\hat{z} |$$

At present, we consider six values of γ_i, ranging from 0 to 315 degrees.

For example, suppose that we are considering the in-hole position

```
pin = hole*trans(nilrotn,vector(0,0,-1.71));
hand = pin*trans(rot(zhat,315*deg)*rot(xhat,180*deg),vector(0,0,3.54));
```

corresponding to grasping parameters ω = 315 degrees, γ = 180 degrees, and d = 3.45 cm; and pin-hole rotation angle ϕ = 90 degrees.[13] We assume that the hand holds the pin with essentially no error, but the hand may be subject to orientation errors of up to ±0.25 degrees about the hand x, y, and z axes, and the hole orientation may be subject to rotation errors of ±5 degrees about the z axis. These values give us an estimate of the pin-hole rotation error:

$$\Delta R_{hp} \approx I + ROT(\hat{z},225^0)*(\eta_x M_x + \eta_y M_y + \eta_z M_z)*ROT(\hat{z},-225^0) + \nu M_z$$

where M_x, M_y, and Mz are related to infinitesimal rotations about the x, y, and z axes and are shown below:

$$M_x = \begin{vmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{vmatrix}$$

$$M_y = \begin{vmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{vmatrix}$$

$$M_z = \begin{vmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{vmatrix}$$

The constraints on the free variables are:

- 5 deg $\leq \nu \leq$ 5 deg
-0.25 degrees $\leq \eta_x \leq$ 0.25 degrees
-0.25 degrees $\leq \eta_y \leq$ 0.25 degrees
-0.25 degrees $\leq \eta_z \leq$ 0.25 degrees

where η_x, η_y and η_z represent the hand rotation errors, and ν represents the rotation error of the hole. Solving, we get

---

[13] These parameters correspond to the best overall strategy found in Section 4.8.

γ_i	$\Delta\theta_i$
0°	.354°
30°	.306°
60°	.306°
90°	.354°
120°	.306°
150°	.306°

Consquently, $\Delta\theta_{max}$=.354°.

Once this value is computed, we compare it to the allowabk limit, $\Delta\theta_{ok}$. If the value is out of bounds, then the pin-to-hole alignment may not be good enough to guarantee success. Presently, this is grounds for rejection of the strategy. Other options would be to add another parameter to the search loop, so that different pin orientations, as well as different "xy" positions are tried; to include "smarter" accommodation techniques; or to attempt in some way to ascertain the pin-hole orientation more accurately.

4.k.2 Error Along the Hole Axis

Δz is easily computed from

$$\Delta z = \hat{z}\cdot\Delta p_{hp}$$

Recall that our "in hole" test examines how far the pin gets along the hole axis before being stopped. If it doesn't, get far enough, then we assume that we hit the object, and must try again. For this test to work, we must be sure that Az cannot be big enough to cause confusion. I.e.,

$$|\Delta z| \leq \tau*(d_f-d_s)$$

where τ is a suitable' "fudge factor" (currently' 0.75) designed to keep us well within the "safe" region. If the maximum value of At falls within this limit, then no further refinement is needed. If not, then 'tapping" is considered as a means of getting the necessary accuracy, To use this strategy, the system must select a place to tap. The principal considerations in making this choice are:

1. The point should be as close to the hole as practical, to minimize the effects of rotation errors in the hole surface[14] and to minimize the time wasted in moving to a tapping place.

---

[14] -Actually, this consideration is too strong. The "right" thing to do is to compute the expected misorientation and then use that result to compute the allowable distance from the hole.

2. The point should be far enough from any confusing features (like holes) so that we are sure to hit the surface we expect to hit.

The method used is roughly as follows:

```
s ← surface into which the hole is drilled;
(x_h,y_h) ← location of hole in coordinate system of surface;
r_td ← radius of hole + radius of pin tip;
Δr_td ← max( 0.3 inches, Δx_hp, Δy_hp)
maxr ← maximum distance of any point on s from the hole;
dbest ← 0;
for r ← r_td + Δr_td step Δr_td until rmax do
    begin real ξ;
    for ξ ← 0 step Δr_td/r until 2π do
        begin real x,y,d;
        x ← x_h + r*cosξ; y ← y_h + r*sinξ;
        d ← distance of nearest hole or edge in s from (x,y);
        comment d<0 if (x,y) is outside of s ;
        if d>dbest then
            b e g i n dbest←d;xbest←x;ybest←y;end;
        end;
    if dbest>Δx_hp then done;
    end;
```

The tapping place is then computed from *xbest* and *ybest* as

$$pin = hole * trans(nilrotn, R_{sh} * vector(xbest, ybest, 0) - p_{sh});$$

where

$$T_{sh} = trans(R_{sh}, p_{sh})$$

= position of hole with respect to *s*

The results of a typical application of this method is shown below. Here, we are looking for a tapping place near one of the corner holes of our box, located at (3.85,3.20) with respect Co the top surface of the box. In this case, we assume that the box location is known precisely, so that, the only xy error comes from the hand. Thus,

$$\Delta r_{td} = \max(0.3 \text{ inches}, \Delta x, \Delta y)$$

$$= \max(.762 \text{ cm}, .243 \text{ cm}, .226 \text{ cm})$$

$$= .762 \text{ cm}$$

On the first iteration through our outer (Y) loop,

r = .450 cm + .762 cm = 1.21 cm

Going through our inner loop produces:

x	y	d
5.06	3.20	-.612[15]
4.83	3.91	-.397
4.22	4.35	-.553
3.47	4.35	-.552
2.87	3.91	-.111
2.64	3.20	.577
2.87	2.49	-.115
3.48	2.05	.229

Thus, *xbest* ← 2.64, *ybest* ← 3.20, and *dbest* ← .577 on this iteration. Since this value of *dbest* is considerably larger than our possible confusion radius (.243 cm), we have found an acceptable tapping place, and can stop looking. The corresponding tapping point is:

```
trans(nilrotn,vector(-1.21,.002,0));
```

Once such a point has been found, then At is re-evaluated, taking account of the additional measurement. If the potential error has now been sufficiently limited, then the tapping place is entered into' the strategy record, and the estimated cost is updated to include the time of the extra motion. In this case, the reduced error is Δz = .180 cm, which is much smaller than the required accuracy of 1.71 cm, and the estimated extra time is 1.2 seconds.

If no tapping place.can be found, then the system currently must give up on the strategy, and hope that one of the other grasping positions will produce more accuracy along the hole axis. Unfortunately, this hope is frequently a forlorn one. Eventually, we would like to consider *other* measurement tricks to try if tapping doesn't work. These alternative tricks presumably could be weighted according to their expected cost, and a "best" combination picked.

4.5.3 Errors in the Plane of the Hole

These errors cause the pin to miss the hole, and are overcome by searching. To estimate in-plane errors, we compute

$$\xi_k = \max | (\cos\zeta_k, \sin\zeta_k, 0) \cdot \Delta p_{hp} |$$

for

ζ_k = 30k degrees
-0sks5

Then, we take

---

[15] Negative values mean outside surface or on top of a hole.

$$\Delta x = \max \xi_k$$
$$\Delta y = \xi_{(k+3) \bmod 6}$$
$$\zeta = \zeta_k$$

This produces an "error footprint" rectangle with sides $2\Delta x$ and $2\Delta y$, rotated by ξ with respect to the hole. We set

$$\Delta r = \max(\Delta x, \Delta y)$$

A, typical instance of this calculation is illustrated below. Here, the nominal pin and hole positions are the same as those given in Section 4.5.1. In addition, we assume that the rotation errors are as previously stated and that the object in which the hole is drilled is subject to small displacement errors in x and y. This gives us the following expression for pin-hole displacement errors.

$$\begin{aligned}\Delta p_{hp} = {} & \nu * \text{vector}(-3.20,-3.85,0) \\ & + \eta_x * \text{vector}(2.5,-2.5,0) + \eta_y * \text{vector}(-2.5,2.5,0) + \eta_z * \text{vector}(0,0,0) \\ & + \delta_x * \text{vector}(.707,-.707,0) + \delta_y * \text{vector}(-.707,-.707,0) - \delta_z * \text{zhat} \\ & - \epsilon_x * \text{xhat} \cdot \epsilon_y * \text{yhat}\end{aligned}$$

where η_x, η_y, and η_z represent rotation errors in the hand; δ_x, δ_y, and δ_z represent displacement errors in the hand; ν represents rotation error in the object containing the hole (our familiar box); and ϵ_x and ϵ_y represent object displacement errors.

The corresponding constraint equations are:

```
[ 1.00 , .000 , .000 , .000 , .888 , .888 ] . V1 ≤.127.127
[ 1.00 , .000 , .000 , .000 , .888 , .000 ] . V1
[ .000 , 1.00 , .000 , .000 , .000 , .000 ] . V1 ≤.127.127
[ .000 , 1.88 , .00 , .000 , .000 , .000 ] . V1
[ .000 , .888 , .000 , .000 , .888 , .000 ] . V1 ≤.127.127
[ .888 , .000 , 1.00 , .000 , .868 , .000 ] . V1
[ .000 , .000 , .000 , 1.00 , .888 , .000 ] . V1 ≤ .436e-2
[ .000 , .888 , .000 , 1.0 , .000 , .000 ] . V1 ≥-.436e-2
[ .000 , .888 , .000 , .000 , 1.00 , .000 ] . V1 ≤ .436e-2
[ .000 , .888 , .000 , .000 , 1.00 , .000 ] . V1 ≥-.436e-2
[ .888 , .000 , .000 , .000 , .000 , 1.00 ] . V1 ≤ .436e-2
[ .000 , .888 , .000 , .000 , .000 , 1.88 ] . V1 ≥-.436e-2
[ 1.88 , .000 , .000 ] . V2 ≤ .762
[ 1.88 , .000 , .000 ] . V2 ≥-.762
[ .000 , 1.00 , .000 ] . V2 ≤ .688
[ .000 , 1.88 , .000 ] . V2 ≥-.508
[ .688 , .000 , 1.00 ] . V2 ≤ .873e-1
[ .000 , .000 , 1.00 ] . V2 ≥-.873e-1
```

where

$v1 = [\ \delta_x, \delta_y, \delta_z, \eta_x, \eta_y, \eta_z\ ]$
$v2 = [\ \epsilon_x, \epsilon_y, \nu\ ]$

Computing ξ_k for six values of ζ_k gives us:

ζ_i	ξ_i
0°	1.05 cm
30°	1.43 cm
60°	1.50 cm
90°	1.24 cm
120°	1.16 cm
150°	1.15 cm

Consequently, Ax = 1.50 cm, Ay = 1.15 cm, and ζ = 60 degrees.

If Ar is less than Δr_{ok}, then we won't have to worry about searching, since the pin will always be within the allowable error radius of the hole, If not, then a search will have to be planned. The search loop used is shown in Section 4.8.

If a search is requited, the cost of the strategy must be adjusted to account for the time spent doing it. This is difficult, since we don't know anything about the *distributions* of the errors. A worst-case estimate can, of course, be obtain& by multiplying the time to make one try by the total number of points in the search pattern. However, this seems too pessimistic. Therefore, we only count those points within $\Delta x/2$ and $\Delta y/2$ of the hole.

4.6 Selecting a Strategy

We wish to select the strategy with the smallest execution time. The most direct way to do this is to plan all strategies out fully, evaluate them, and then take the cheapest. This approach has the drawback that we may spend considerable time refining strategies whose basic motions are so inefficient as to rule them out. Therefore, we first decide on the basic motions for each distinct grasp point. All candidate strategies are sorted according to gross motion time, and then considered in "best first" order. If we reach a point where the next best unrefined strategy is more expensive than a fully planned strategy, then we can stop searching.

```
strategies ← null;
'for each g such that g is a grasping strategy do
        begin
        Decide best way to get pin to hole, using g.
        if there is a way then
                Create a pin in hole strategy & insert it in strategies,
                ranked by expected time.
        e   n   &
```

```
best-strategy ← Incantation;
shortest_time ← 10^39 seconds; { a very long time }
minimum_refinement_cost ← lower bound on "fine motion" time;

for each s such that s ∈ strategies do
    begin
    if cost(s)+minimum_refinement_cost≥shortest_time then
        done; { best_strategy is the best strategy we've found)
    Refine s to account for accuracy considerations.
    Revise the cost estimate for s.
    if cost(s)<shortest_time then
        begin
        shortest_time ← cost(s);
        best-strategy ← s; .
        end;
    end;
```

Here, we have used *minimum_refinement_cost* to tighten our cutoff somewhat. It may be computed by assuming that there is no error in the arm *or* grasp, so that all error between pin and hole comes from errors in the hole location, and then considering what refinements would be necessary.

4.7 Code Generation

Once we have selected a strategy, the actual synthesis of program text is accomplished by calling procedures that extract the appropriate values from the strategy record, substitute them into the appropriate slots in code skeletons, and print the results.

Pickup Strategies

The procedure for writing pickup strategies ldoks something like this:

```
procedure write_pickup(pointer(pickup_strategy) pkp);
    begin
    print("{ PICKUP ",pkp,":",remarks[pkp],"}",crlf [16]);
    print("OPEN BHAND TO ",approach_opening[pkp],";");
    write_motion_sequence({{approach_point[pkp],grasp_point[pkp]}},null);
    print("CENTER    BMANIP",crlf);
```

---

[16] "Carriage Return, Line Feed"

```
print(    ON OPENING < ",grasp_opening[pkp]," DO ",crlf);
print(        ABORT('GRASP FAILED');",crlf);
print("AFFIX ",location_variable(object[pkp])," TO BMANIP;",crlf);
write_motion_sequence({{departure_point[pkp]}},null);
end;"
```

Pin-in-Hole Strategies

The *write_pin_in_hole* procedure is slightly more elaborate than *write-pickup,* which it uses as a subroutine. In addition to generating more output, *write_pin_in_hole* must make several *decisions* about what code to emit:

1. Is "tapping" to be performed?

2. Is *a* search to be made?

Actually, these decisions have *already* been made and are reflected in the data structures. Thus, our code writer looks at the tapping place field of the strategy record to decide question 1. If the record is null, it does nothing; if a point is specified, it emits the appropriate code. (An example may be found at the end of Section 4.8). Similarly, in deciding whether to emit code for a search, it lookt to see if Ax is greater than Δr_{ok}.[18] If so, the search is produced; otherwise, a perfunctory check:

```
IF ABS(DISTANCE_OFF) > T*(d_f-d_s) THEN
        ABORT("pin MISSED hole" UNEXPECTEDLY)
```

is written instead. The program text produced for a typical strategy, together with further discussion of the particular constructs used to implement search loops, may be found in Section 4.8.

Motion Sequences

Both *write_pickup* and *write_pin_in_hole* use ,a procedure, *writ&-motion-sequence, to* generate motion statements. This procedure works roughly as follows:

---

[17] "bmanip" is an alternative name (used in the current AL implementation) for the blue 'arm, and "bhand" is the name for the blue hand.

[18] Recall that in Section 4.5.3, we selected ζ so that $\Delta x \geq \Delta y$.

```
procedure write-motion-sequence&t destinations;string qualifiers);
   begin integer i,j,k;
   j←0;
  . while j< length(destinations) do
      begin
      i←j←j+1;
      controllable ← what[destinations[i ]];
      while j < length(destinations) and what[destinations[j+1]]=controllable do
         j←j+1;
      corn men t Now, {{destinations[i],...,destinations[j]}} is a
         subsequence with, the same controllable frame.;
      print("MOVE ",location_variable(controllable)," TO ",
           location_variable(base[destinations[i]]),"*",xf[destinations[i]],crlf);
      for k ← i+1 step 1 until j do
         print(if k=i+1 then "VIA " else ", ",
              location_variable(base[destinations[k]]),
              "*",xf[destinations[k]],crlf);
      prin t(qualifiers,crlf );
      end;
   end:
```

Here, we first break . the destination sequence up into subsequences with common "controllable" frames, and then generate a motion statement for each subsequence. There are several possible' pitfalls, since the semantics of two successive motion statements are not identical to *a* single statemerit, especially where the *qualifiers* include stop-on-force tests. At present, this difficulty is solved by being careful that the procedure will not be called with arguments that "split" the motion at a bad point. This solution was satisfactory for our *present (small) set of code emitters, but something better will have to be done in the long run. An alternative approach would be to compute the relation between each controllable frame and the manipulator, and then to 'write the motion purely in terms of the manipulator frame. This solves the abovementioned difficulty, but introduces additional complexity, making the output programs harder to read. A better fix would probably be to extend the syntax of AL to allow hybrid destination lists, and then allow the AL compiler to worry about. the relation to manipulator frames.[19]

4.8 Example

. The task, strangely -enough, is insertion of an aligning pin into a hole drilled in the top surface of a small metal box. Initially, the box body sits on the work table at T_{wb}, and is subject to displacement errors of up to ±0.3 inches along the x-axis of the table and up to 0.2 inches along the y-axis and to rotation errors of up to 5 degrees about the table z-axis. The hole (*bh1*) is located at Tbh with respect to the box, the pin (*pin1*) is held in a tool rack at T_{wp}, and the manipulator (bmanip) is parked at bprrk, where

[19] Such an approach is a natural extension to the present translation performed by the AL compiler.

T_{wb} = trans(nilrotn, vector(45.2, 102., 0))
Tbh = trans(nilrotn, vector(3.85, 3.20, 4.90))
T_{wp} = trans(rot(zhat,90*deg), vector(24.1, 117., .537));
bpark = trans(rot(yhat,180*deg), vector(43.5,56.9,10.7));

From the initial computation, we determine that

direction = axes parallel
d_f = 1.71 cm
d_s = 0
Δr_{ok} = 0.762 cm
$\Delta\theta_{ok}$ = JO degrees

In other words, the pin is expected to go 1.71 cm into the hole. When we make the attempt, if the pin tip is within 0.762 cm of hole center and the axes are within 10 degrees of parallel, then the insertion operation will succeed. If we miss, then we won't go any distance into the hole at all. (I.e., we won't get stuck halfway in).

The pickup strategy generator now goes to work and decides on a single grasping distance, and a range of grasp angles:

d_{grasp} = 3.54 cm
100 degrees ≤ 7 ≤ 180 degrees

It then produces nine feasible pickup strategies, ranging in cost from 4.08 seconds to 8.58 seconds. These are then elaborated into unrefined motion strategies, with time estimates of 5.47 seconds to 12.7 seconds. A computer generated summary of the best of these strategies is shown below?

```
PHL SPEC 132757
  PREL IMS: NULL-RECORD
PICKUP:  PICKUP  SPEC  162775
   PRELIMINARIES:  NULL-RECORD
   APPROACH OPENING: 2.88
   APPROACH: BMANIP=PIN1*TRANS(ROTN(VECTOR(.679,.679,.281),149*DEG),VECTOR(-3.59,0,7.13))
   GRASP  OPENING: .185
   GRASP: BMANIP=PIN1*TRANS(ROTN(VECTOR(.679,.679,.281),149*DEG),VECTOR(0,0,3.54))
   GRRSP  DETERM: NILTRANS
   DEPARTURE POINT:  PIN1=PIN1*TRANS(NILROTN,VECTOR(0,0,6.79))
   GOODNESS:  4.13
   REMARKS: W = 90.0 dog Grasp Angle = 135. dog Grasp Distance = 3.54
  APPROACH: PIN1=BH1*TRANS(ROTN(ZHAT,90.*DEG),VECTOR(0,0,2.54))
  DESTINATION: PIN1=BH1*TRANS(ROTN(ZHAT,90.*DEG),VECTOR(0,0,-1.71))
  TARGET: PIN1=BH1*TRANS(ROTN(ZHAT,90.*DEG),VECTOR(0,0,-4.25))
  XPORT  TIME: 1.34
  GOODNESS:  5.47
  TAP: NULL_RECORD   (The fields below aren't filled in yet)
  FINE  TIME: .000
  PH DZ: .800
  PH FP DX: .000
```

[20] The output has been edited slightly to improve readability.

```
PH FP DY: , 888
PH FP ROT: ,888
```

In terms of the parameters described in earlier sections, this strategy corresponds to:

ω = 90 degrees
γ = 135 degrees
grasp distance = 5.54 cm
dtry = 4.25 cm
standoff = 2.54 cm
ϕ = 90 degrees

Once all our candidate motion strategies have been generated, we set about refining them, in best-first order. To do this, we generate the error terms and compare them against the requirements established at the very beginning. For the'rtrategy just shown, we get

Az = ,180 cm
Ax = 1.50 cm
Ay = 1.15 cm
ζ = 60 degrees

The value of Az is thus small enough so that we are sure not to be confused about whether the pin will make it into the hole. Thus, we don't have to "tap". On the other hand, the "error footprint" is bigger than Δr_{ok}, so we will have to search. The estimated extra time for this is 1.8 seconds, giving us a total estimated cost of 7.27 seconds.

The refinement of strategies continues until we reach:

```
PHL SPEC  134823
  PRELIMS: NULL-RECORD
  PICKUP:  PICKUP SPEC 163875
    PRELIMINARIES:  NULL-RECORD
    APPROACH OPENING: 2.98
    APPROACH: BMANIP=PIN1*TRANS(ROTN(VECTOR(.608,.608,.518),126.*DEG),VECTOR(-5.,0,4.42))
    GRASP OPENING: .185
    GRASP: BMANIP=PIN1*TRANS(ROTN(VECTOR(.608,.608,.518),126,*DEG),VECTOR(0,0,3.54))
    GRASP DETERM: NILTRANS
    DEPARTURE POINT: PIN1=PIN1*TRANS(NILROTN,VECTOR(0,0,6.79))
    GOODNESS:  4.88
    REMARKS: W = 88.8 dog Grasp Rng lo = 100. drg Grasp Dttancr = 3.54
  APPROACH: PIN1=BH1*TRANS(ROTN(ZHAT,90.*DEG),VECTOR(0,0,2.54))
  DESTINATION: PIN1=BH1*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,-1.71))
  TARGET: PIN1=BH1*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,-4.25))
  XPORT TIME: 2.56
  GOOONESS: 6.64
  TAP: NULLJECORO
  FINE TIME: .000
  PH DZ: .000
  PH FP DX: .000
  PH FP DY: ,888
  PH FP ROT: .000
```

This strategy will take *at least* 6.64 seconds to execute, and all the rest will take even longer. However, at this point, the best completely refined strategy is:

```
PHL SPEC 138523
  PREL IMS: NULL-RECORD
  PICKUP: PICKUP SPEC 72827'
    PRELIMINARIES: NULL_RECORD
    RPPRORCH OPENING: 2.98
    RPPRORCH: BMANIP=PIN1*TRANS(ROTN(VECTOR(.924,.383,.001),180.*DEG),VECTOR(0,0,8.62))
    GRASP OPENING: .185
    GRASP: BMANIP=PIN1*TRANS(ROTN(VECTOR(.924,.383,.001),.180.*DEG),VECTOR(0,0,3.54))
    GRRSP DETERM: NILTRANS
    DEPARTURE POINT: PIN1=PIN1*TRANS(NILROTN,VECTOR(0,0,6.79))
    GOODNESS: 4.16
    REMARKS: W = 315. dog Grasp Anglo = 188. deg Grasp Distance = 3.54
  APPROACH: PIN1=BH1*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,2.54))
  DESTINATION: PIN1=BH1* TRANS( ROTN( ZHAT, 90.000*DEG), VECTOR( .000, .000,-1.71))
  TARGET:PIN1=BH1*TRANS (ROTN(ZHAT,90.0*DEG),VECTOR(0,0,-4.25))
  XPORT TIME: 1.38
  GOODNESS: 6.14
  TAP: NULL-RECORD
  FINE TIME: .600
  PH DZ: .127
  PH FP OX: 1.58
  PH FP OY: 1.15
  PH FP ROT: 1.85
```

Since we already have a refined strategy better than any of the **remaining** unrefined strategies, we can stop looking, and write the AL code for **our current best** strategy. In this case, **the** computer generated the following program **text:**[21]

```
! PIN-IN-HOLE STRATEGY 138523:
  DROK = .762      FPX = 1.58      FPY = 1.15      FPU = 1.05
  02 = .127        ESTIMATED TIME = 6.14 !

! PICKUP 72827:
  W = 315. dog Grasp Anglo = 180. deg Grasp Distance = 3.54 !

OPEN BHRNO TO 2.88;
MOVE BMANIP TO PIN1*TRANS(ROTN(VECTOR(.924,.383,.001),180.*DEG),VECTOR(0,0,3.54))
    VIA PIN1*TRANS(ROTN(VECTOR(.924,.383,.001),180.*DEG),VECTOR(0,0,8.62));
CENTER BMANIP
    ON OPENING < .185 DO
        BEGIN ABORT("GRASP FAILED");END;
AFFIX PIN1 TO BMANIP;
MOVE PIN1 TO PIN1*TRANS(NILROTN,VECTOR(0,0,6.79));

! FIRST ATTEMPT !

MOVE PIN1 TO BH1*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,-4.25))
    VIA BH1*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,2.54))
      ON FORCE(ORIENT(PIN1) *ZHAT) > 8*OZ DO STOP
      ON ARRIVAL DO ABORT("EXPECTED A FORCE HERE");
DISTANCE_OFF←ZHAT . INV(BH1*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,-1.71)))*DISPL(PIN1);
IF ABS(DISTANCE_OFF) > .168 THEN
    BEGIN !PIN1 MISSED BH1 !
    BOOLEAN FLAG;
    ! SEARCH LOOP: !
    RERL R,DW,W,X,Y;FLAG←FALSE;
    R ← .572; ! 0.75*DROK !
```

---

[21] The program has been edited very slightly to improve readability by removing excess **blanks and** by rounding all numbers to three significant digits. **(For instance,** the **computer output had** ● 0.00 **106",** instead- of **"0.00 1".)**

```
    WHILE NOT FLAG AND R ≤ 1.72 DO
      BEGIN
      W ← 0; DW ← ( .572/R)*RAD;
      WHILE NOT FLAG AND W<259*DEG DO
        BEGIN
        IF ABS (X←R*COS(W))< 1.58 AND ABS(Y←R*SIN(W))< 1.15 THEN
          BEGIN FRAME SETPNT;
          SETPNT←BH1*TRANS(NILROTN,ROT(ZHAT,1.85)*VECTOR(X,Y,0));
          MOVE PIN1 TO SETPNT* TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,-4.25))
               VIA SETPNT*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,2.54))
                 ON FORCE(ORIENT(PIN1)*ZHAT) > 8*OZ DO STOP
                 ON ARRIVAL DO ABORT("EXPECTED a FORCE HERE");
          DISTANCE_OFF←ZHAT . INV(SETPNT*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,-1.71)))
                                                                    *DISPL(PIN1);
          IF ABS(DISTANCE_OFF) < .169 THEN
              FLAG←TRUE;
          END;
        W←W+DW ;
        END;
      R←R + .572;
      END;
    IF NOT FLAG THEN ABORT("PIN1 MISSED BH1");
    END;

  ! LET GO !
  OPEN BHAND TO 2.98;
  UNFIX PIN1 FROM BMANIP;

  ! NOW GET HAND CLEAR !
  MOVE BMANIP TO BMANIP*TRANS(NILROTN,VECTOR(0,0,-5.88));
```

The search loop used here works by generating **(x,y)** offsets **in ever-widening circles** about **the origin. Each point generated is** tested to see if it is within **the footprint limits:**

$$-\Delta x \le x \le \Delta x$$
$$-\Delta y \le y \le \Delta y$$

If so, then a displacement vector (in the coordinate system of **the hole) is computed by:**

rot(zhat,ζ)*vector(x,y,0)

and used to produce an offest candidate location **(*setpnt*) for** the **hole location. If the insertion attempt for this, point** succeeds, **then** *flag* is set to indicate success and **the** loop **is terminated. If the attempt fails,** or if **(x,y)** was outside **the** error footprint, then the next **point is tried. . The loop continues** to be executed **until either the entire** expected **error range has been exhausted or** the **insertion succeeds.**[22]

Variation

The example above **required a** search, but **no** "tapping", since the error along the **z-axis** of the hole **was much smaller than** the expected **penetration** of the pin into the hole. if we

---

[22] **Some people have** commented on the computational inefficiency of generating **(possibly) many values** of **(x,y) which** will **be** thrown away. For any reasonable error limits, however, **this** cost **can be** ignored, since the **time** required for moving the manipulator far exceeds **that rquired to compute** a **target** point.

increase the uncertainty along this axis, then a tap (or some other measurement) must be used before the insertion can be tried. This is illustrated by the code below, which was written for the same assumptions as those used earlier, except that the box position is assumed to be subject to no rotation or "in plane" displacement errors, but may have an error of up to 0.75 inches up or down.

```
! PIN-IN-HOLE STRATEGY 1343561
  DROK = .762       FPX = .243       FPY = .226       FPU = .524
  DZ = .188       ESTIMATED TIRE = 6.67    !

! PICKUP 147157:
  W = 90.0_deg Grasp Angle = 135. deg Grasp Distance = 3.64 !

OPEN BHAND TO 2.98;
MOVE BMANIP TO PIN1*TRANS(ROTN(VECTOR(.679,.679,.281),149*DEG),VECTOR(0,0,3.54))
    VIA PIN1*TRANS(ROTN(VECTOR(.679, .679, .281),149*DEG),VECTOR(-3.59,0,7.13));
CENTER BMANIP
    ON OPENING < .185 DO
        BEGIN ABORT("GRASP FAILED")) END;

AFFIX PIN1 TO BMANIP;
MOVE PIN1 TO PIN1*TRANS(NILROTN,VECTOR(0,0,6.78));

! MUST TAP !

MOVE PIN1 TO BH1*TRANS(NILROTN,VECTOR(-1.21,-.002,-5.08))
    VIA BH1*TRANS(NILROTN,VECTOR(-1.21,-.002,5.08))
      ON FORCE(ORIENT(PIN1)*ZHAT) > 8*OZ DO STOP
      ON ARRIVRL DO ABORT("EXPECTED A FORCE HERE");
CORR = ZHAT . INV(BH1* TRANS( NILROTN, VECTOR(-1.21,-.002, .000)))*DISPL(PIN1);

! FIRST ATTEMPT !

MOVE PIN1 TO BH1*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,-4.25))
    VIR BH1*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,2.54))
      DN FORCE (ORIENT(PIN1)*ZHAT) > 8*OZ 00 STOP
      ON RRRIVRL DO ABORT("EXPECTED A FORCE HERE");
DISTANCE_OFF=ZHAT.INV(BH1*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,-1.71)))*DISPL(PIN1)-CORR;
IF ABS(DISTANCE_OFF) > .239 THEN
    ABORT("PIN1 HISSED BH1 UNEXPECTEDLY.");

! LET GO !
OPEN BHAND TO 2.98;
UNFIX PIN1 FROM BMANIP;

! NOW GET HAND CLEAR !
HOVE BMANIP TO BMANIP*TRANS(NILROTN,VECTOR(0,0,-5.08));
```

In this case, the error footprint in the plane of the hole is small enough so that no search is needed. On the other hand, the uncertainty along the hole axis is quite large. Consequently, the system has chosen a tapping place at ,

trans(nilrotn,vector(*-1.21,-.002,0*))

with respect to the hole, which is then used to locate the top surface of the box more precisely. This is accomplished by moving the pin along a path starting two inches above the nominal height of the surface and ending two inches below it. When the pin hits the surface, the motion is stopped and used to compute a correction (CORR) for use in the success test.

Chapter 5.

CONCLUSIONS AND FUTURE WORK

The goal of this research was the generation of AL manipulator control programs from high level task descriptions. The full topic of automatic generation of AL code is extremely broad, and many narrowing assumptions have been necessary in order for us to demonstrate basic feasibility while keeping the scope of effort within reasonable bounds. This report has explained how AL programs have been generated automatically for a particular programming example, the insertion of a pin into a hole, which is a typical subtask of many assembly operations.

The example was first discussed from the point of view of a programmer coding directly in AL, to show that the task is non-trivial if attention is given to making the code rugged with respect to positioning errors. Next, the modeling requirements for automatic coding were analyzed, since the automation of coding decisions requires that the necessary information be represented in a form usable by the computer. Finally, the programming example was revisited, this time with the computer using its planning model to generate the AL code automatically.

Extensions

Although the pin-in-hole task was used as an example throughout this work, a conscious effort was made to avoid undue specialization. The modelling requirements for this task – expected locations, accuracies, etc. – are applicable to 'other assembly operations, and the techniques used to represent planning information were developed without any particular task in mind. When time came to write the automatic coding procedures described in Chapter 4, no substantial changes to the modelling mechanisms were required, although a certain amount of bug killing was necessary.

However, it is worthwhile to consider how hard it would be to add automatic coding procedures for other tasks.

As one might expect, the easiest additions would be for variations of pin-in-hole, such as screw-in-hole, for which most of the analysis has already 'been done.' The principal additional difficulty that a screw-in-hole writer must handle would be figuring out how to pick up a screwdriver and how to load 'a screw onto it. Since these are fairly specialized operations, it seems reasonable to construct a small library containing the appropriate code for different drivers and screw dispensers. We would also want to consider alternative methods, such as using the hand to start the screw into the hole[2] and then driving it down.

Almost as easy would be the task of fitting a nut or washer over a stud, although keeping the fingers out of the way would probably be more of a problem. Only slightly harder would be mating operations, such as fitting a cover plate or gasket over aligning pins, and operations such as putting a part into a vise or simple fixture.

---

[1] Appendix A.2 illustrates a typical error calculation for a screw on the end of a driver.

[2] 'This is just pin-in-hole with a twist at the end.

The important characteristics of these tasks are that they can be performed with relatively simple motion sequences and straightforward verification tests, that the accuracy requirements are fairly easy to state, and that the coding decisions all rely on fairly *local* properties. Where these characteristics are not present, automatic coding will be much harder. Assembly tasks requiring clever uses of force, working in cluttered environments, and handling limp objects are typical difficult tasks. It should be pointed out that humans don't know much about programming such operations, either. Since it is very difficult to automate coding decisions which cannot be clearly identified, these tasks must be much better understood before much success can be expected.

Planning Coherent Strategies

My early research on automatic manipulator programming was primarily concerned with the problem of how to write coherent programs which took account of interactions between individual coding decisions. This work was done at a somewhat "symbolic" level;[3] typical decisions were selection of the order in which operations were to be performed, selection of "good" workpiece positions, etc. It proved fairly easy to get a system to make these decisions *in a toy* world of symbolic assertions.. The rude awakening came with the transition to real data. The work reported in this dissertation has been largely concerned with representation of planning knowledge about real-world situations and then using it to make rather more "local" coding decisions.

Although it is certainly possible to "put up" a system which plans each task-oriented operation independently of the others, interactions must be considered if really efficient programs are to be produced.

In Chapter 4, we saw that when selecting a grasping point to pick up the pin, we had to consider both the initial and final positions of the pin. The estimated motion time included both the time for the hand to reach the pin and the time for the pin to move to the hole. Also, we discovered that some grasping strategies gave larger search patterns than others. All these factors affected our final choice.

This sort of interaction is not confined to choices made within individual assembly operations. For instance, suppose we must place pins into two holes in our favorite box. Then, in selecting a grasping method for the first pin, we should remember that our choice will also affect how much time will be required to pick up the second pin. Other interactions may be more subtle. Inserting the first pin gives us information about the box location. Since this information can be used to reduce the search required for the second insertion, we perhaps ought to consider the accuracy associated with different grasping orientations as well.

One of the key ideas of the earlier work was planning by progressive refinement. Within this paradigm, a program outline is prepared, then elaborated into a more detailed one, and the process is iterated until a finished product is produced. The advantages of this approach are that planning for individual operations can proceed within the context of other parts of the program and that effort is not wasted on contradictory or irrelevant strategies." Before these advantages can be obtained for real manipulator programs, we need a better understanding of how individual coding decisions affect each other. Although

---

[3] Sacerdoti [22, 23] successfully applied similar ideas to a different domain.

the modelling techniques developed in this dissertation – particularly, those for representing object relations and for relating planned actions to accuracy information – can, perhaps, provide a basis for such understanding, much very hard work needs to be done. The development of a good constraint formalism for position **requirements,** discussed earlier, would be especially helpful.

5.1 Acknowledgements

I 'wish to thank my thesis advisory committee, Jerome Feldman, Vinton Cerf, and Thomas **Binford,** for their continued interest and encouragement and for many suggestions which have helped give this work some measure of coherence. Also, I must thank all the many people with whom I have discussed various aspects of this work. I owe a special debt to Dave Grossman, who had. the patience to read these chapters several times, and who made many **valuable** editorial improvements.

Chapter 6.

BIBLIOGRAPHY

[1] *Proceedings of an ACM Conference on Proving Assertions About Programs,* SIGPLA N Notices, January 1972.

[2] Association for Computing Machinery, *Proceedings of a Symposium on Very High Level Languages,* SICLAN Notices, April 1974.

[3] David Barstow, *The PSI Coding Expert: A Knowledge-Based Approach to Automatic Coding,* Manuscript, Submitted to Second International Conference on Automatic Coding, October 1976.

[4] Bruce G. Baumgart, *GEOMED – A Geometric Editor,* Stanford Artificial Intelligence Laboratory Memo AIM-232, Stanford Computer Science Report STAN-CS-74-4 14, May 1974.

[5] T. 0. Binford, D. D. Grossman, E. Miyamoto, R. Finkel, B. E. Shimano, R. H. Taylor, R. C. Bolles, M.' D. Roderick, M. S. Mujtaba, T. A. Cafford, *Exploratory Study of 'Computer Integrated Assembly Systems,* Prepared for the National Science Foundation. Stanford Artificial Intelligence Laboratory Progress Report covering September 1974 to November 1975.

[6] T. 0. Binford, D. D. Grossman, C. R. Liu, R. C. Bolles, R. Finkel, M. S. Mujtaba, M. D. Roderick, B. E. Shimano, R. H. Taylor, R. H. Goldman, J. P. Jarvis, V. Schdnman, T. A. Gafford, *Exploratory Study of Computer Integrated Assembly Systems,* Prepared for the National Science Foundation. Stanford Artificial Intelligence Laboratory Progress Report covering November 1975 to July 1976.

[7] Robert C. Bolles, *Verification Vision Within a Programmable Assembly System,* Ph.D. Dissertation, Summer 1976.

[8] Per' Brinch-Hansen, *Operating System Principles,* Prentice-Hall Series in Automatic Computation, Englewood Cliffs, New Jersey, 1973.

[9] 0. J. Dahl, E. W. Dijstra, C. A. R. Hoare, *Structured Programming,* Academic Press, New York, 1972.

[10] Raphael Finkel, *Constructing and Debugging Manipulator Programs,* Ph.D. Dissertation, Stanford Computer Science Department, 1976.

[11] Robert Floyd, *Towards the Interactive Design of Correct Programs,* Stanford Computer Science Report STAN-U-71-235, September 1971.

[12] Guiseppi Gini, Maria Gini, and Marco Somalvico, *Emergency Recovery in Intelligent Robotr,* Proceedings of the Fifth International Symposium on Industrial Robots, September 1975.

[13] C. Cordell Green, et al., *Progress Report on Program-Understanding Systems,* Stanford Artificial Intelligence Laboratory Memo AIM-240, Stanford Computer Science Report STAN-CS-72-444, August 1974.

[14] David D. Grossman and Russell H. Taylor, *Interactive Generation of Object Models With a Manipulator,* Stanford Artificial Intelligence Laboratory Memo AIM-274, Stanford Computer Science Report STAN-CS-75-536, December 1975.

[15] James R. Low, *Automatic Coding: Choice of Data Structures,* Ph.D. Dissertation, Stanford Artificial Intelligence Laboratory Memo AIM-242, Stanford Computer Science Report STAN-CS-74-452, August 1974.

[16] Zohar Manna and Richard Waldinger, *Knowledge and Reasoning in Program Synthesis,* Stanford Research Institute Artificial Intelligence Center Technical Note 98, November 1974.

[17] 'C. Murphy, *The Reliability of Systems,* unpublished manuscript, date unknown.

[18] J. L. Nevins, D. E. Whitney, H. H. Doherty, D. Killoran, P. M. Lynch, D. S. Seltzer, S. N. Simunovic, R. Sturges, P. C. Watson, E. A. Woodin, *Exploratory Research in Industrial Modular Assembly,* The Charles Stark Draper Laboratory, Inc., Prepared for the National Science Foundation, Memo No. R-800, covering June 1973 to January 1974, March 1974; Memo No. R-850, covering February 1974 to November 1974, December 1974.

[19] Richard Paul, *Modelling, Trajectory Calculation and Servoing of a Computer Controlled Arm,* Stanford Artificial Intelligence Laboratory Memo AIM-177, Stanford Computer Science Report STAN-CS-72-311, November 1972.

[20] Richard Paul, *Manipulator Path Control;* Proceedings of the 1975 International Conference on Cybernetics and Society, 1975, pp. 147-152.

[21] C. Rosen, D. Nitzan, R. Duda, G. Gleason, J. Kremers, W. Park, R. Paul, *Exploratory Research in Advanced Automatton,* Prepared for the National Science Foundation, Stanford Research Institute Project 4391 Fifth Report, January 1976.

[22] Earl D. Sacerdoti, *The Nonlinear Nature of Plans,* Stanford Research Institute Artificial Intelligence Center Technical Note 101, January 1975.

[23] Earl D. Sacerdoti, *A Structure for Plans and Behavior,* Stanford Research Institute Artificial Intelligence Center Technical Note 109, August 1975.

[24] Hanan Samet, *Automatically Proving the Correctness of Translations Involving Optimized Code,* Ph.D Dissertation, Stanford Artificial Intelligence Laboratory Memo AIM-259, Stanford Computer Science Report STAN-CS-75-498, May 1975.

[25] Robert F. Sproull, *(Title Unknown),* Ph.D. Dissertation,, Stanford Computer Science Department, Summer ● 1976.

[26] J. T. Schwartz, *Automatic Data Structure Choice in a Language of Very High Level,* Courant Institute, NYU, 1974.

[27] Norihisa Suzuki, *Automatic Verification of Programs with Complex Data Structures,* Ph.D. Dissertation, Stanford Artificial Intelligence Laboratory Memo AIM-279, Stanford Computer Science Report STAN-CS-76-552, February 1976.

[28] Russell H. Taylor, *The Synthesis of Manipulator Control Programs from Task-Level Specifications,* Ph.D. Dissertation, July 1976.

[29] Richard Waldinger, *Achieving Several Coals Simultaneously,* Stanford Research Institute Artificial Intelligence Center Technical Note 107, July 1975.

Appendix A.

EXAMPLES OF LOCATION AND ACCURACY CALCULATIONS

Box in a Fixture

This sequence of problems illustrates the translation of symbolic relations into constraints, and shows the output estimates that result from application of the iterative method described in my dissertation [28]. Here, we have placed our box into an open-topped fixture, as illustrated in Figure A.1. In the first problem, the box is allowed to rattle around loosely inside the confines of the fixture. In subsequent subproblems, we push the cornei edges up against sides of the fixture, thus further restricting the box.

First Problem

The box has been placed in the fixture, with the bottom surface of the box in contact with the bottom inside surface of the box. This is reflected *in* our data base by the assertion:

(contacts, *bxbtm, bfl.sb,* inside-of)

where *bxbtm* is the bottom of the box, and *bfl.sb* is the bottom of the fixture. This produces the constraint set:

```
 YHAT*R* 5.85* VECTOR(-.760,-.649, .886) ≤  5.8 IO - YHAT . PV
-XHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤  4.0 IO - -XHAT . PV
-YHAT*R* 5.85* VECTOR(-.760,-.649, .000) 4  5.0 IO - -YHAT . PV
 XHAT*R* 5.85* VECTOR(-.760,-.649, .888) ≤  4.8 IO - XHAT . PV
 YHAT*R* 5.85* VECTOR(.760,-.649, .808) 4   5.0 IO - YHAT . PV
-XHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  4.8 IO - -XHAT . PV
-YHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  5.0 IO - -YHAT . PV
 XHAT*R* 5.85* VECTOR(.760,-.649, .888) ≤   4.0 IO - XHAT . PV
 YHAT*R* 5.85* VECTOR( .760,.649, .868) ≤   5.8 IO - YHAT . PV
-XHAT*R* 5.85* VECTOR( .760,.649, .000) ≤   4.0 IO - -XHAT . PV
-YHAT*R* 5.85* VECTOR( .768, .649, .888) ≤  5.0 IO - -YHAT . PV
 XHAT*R* 5.85* VECTOR( .760, .649, .000) ≤  4.0 IO - XHAT . PV
 YHAT*R* 5.85* VECTOR(-.760, .649, .888) ≤  5.0 IO - YHRT . PV
-XHAT*R* 5.85s VECTOR(-.760, .649, .888) ≤  4.0 IO - -XHAT . PV
-YHAT*R* 5.85; VECTOR(-.760, .649, .000) ≤  5.0 IO - -YHAT . PV
 XHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤  4.8 IO - XHAT . PV
0 = .000 - ZHAT . PV
        WHERE R = N LROTN*ROTN(-ZHAT,W)
              PV = [X,Y,Z]
```

Applying the algorithm gives two possible orientations:

```
ESTIMATE LIST:
ITEM416:
X:      -.284 TO .284
Y:      -.555 T    O .555
Z:      -.881 T O .881
W:       87.368*DEG T O 92.632*DEG
COS(W0) = .000 SIN(W0) = 1.888
COS (DW) = .999 R . .046
```

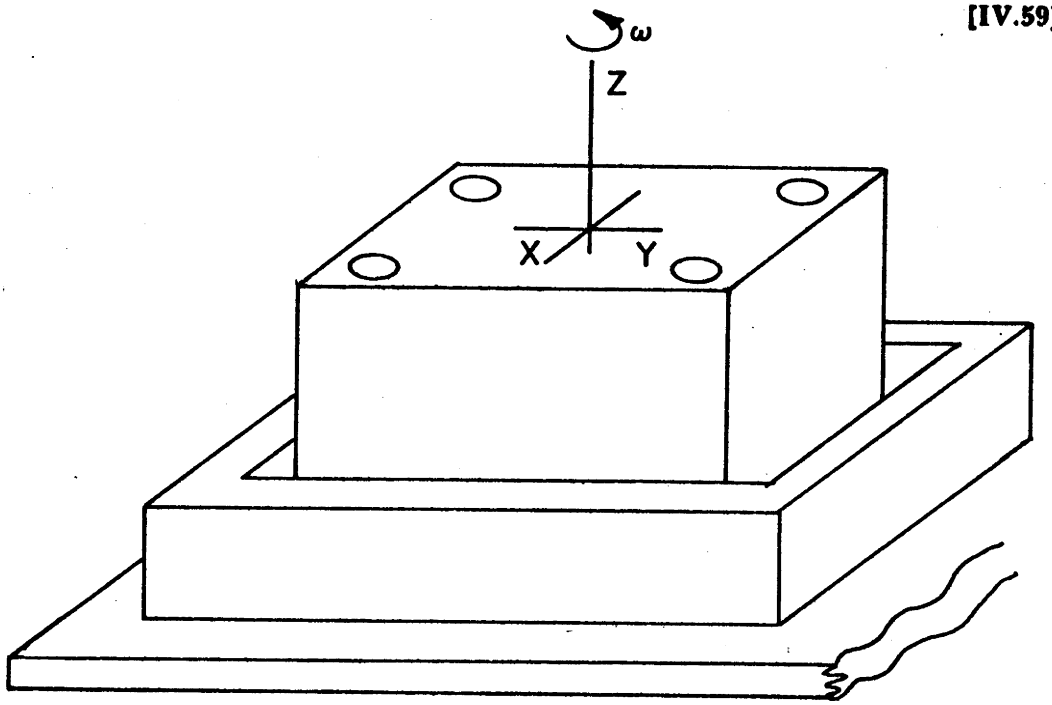


Figure A.1. Box in Fixture

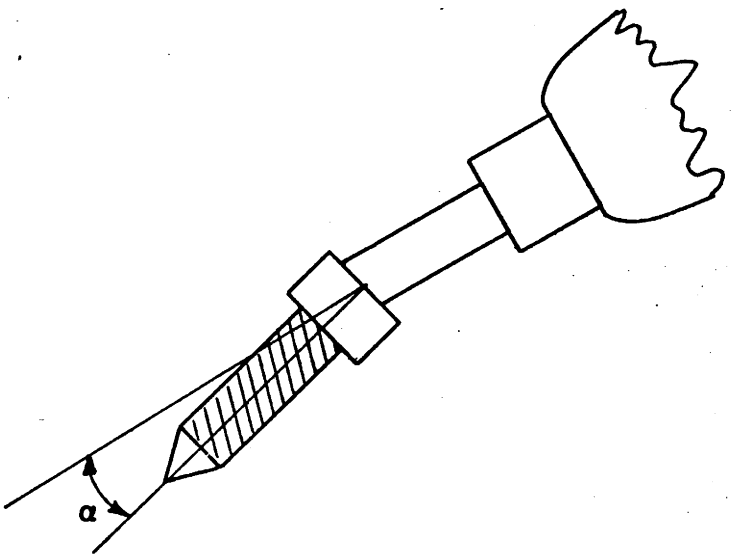


Figure A.2. Screw on Driver

```
ITEM417:
X:      -.284 TO  .284
Y:      -.555 TO  .555
Z:      -.001 TO  .001
W:      -92.632*DEG TO -87.368*DEG
COS(W0) =  .000 SIN(W0) = -1.000
COS(DW) =  .999  R =  .046
```

These results also iilustrate the replacement of equality . **constraints** with a pair of inequalities: **here, Z** goes from -0.001 to 0.001. **This** approximation is **not** strictly necessary. However, it proved useful **in some** *(other)* cases where overdetermination was *a* problem.

Second Problem

We now assert that one of the corner edges of the **box** is in **contact with** *a* side of the fixture.

(contacts, ***bxbtm, bj1.sb,*** inside-of)
(con tacts, ***be9, bj1.s2,*** **ex** ten **t_irrelevan t)**

This gives: .

```
-XHAT*R* 1.00* VECTOR(-.707,-.707, .000) ≤ -.707
 YHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤ 5.888 - YHRT . PV
-XHAT*R* 5.85* VECTORb.788, ;, 649,  , 888) ≤ 4.888 - -XHAT . PV
-YHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤ 5.868 - -YHAT . PV
 XHAT*R* 5.85* VECTOR(-.760,-.649,  , 888) ≤ 4.888 - XHAT . PV
 YHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤ 5, 888 - YHRT . PV
-XHAT*R* 5.85* VECTOR( .760,-.649,  , 888) ≤ 4.888 - -XHAT . PV
-YHAT*R* 5.85* VECTOR( .760,-.649,  , 888) ≤ 5.808 - -YHAT . PV
 XHAT*R* 5.85* VECTOR( .760,-.649,  , 888) ≤ 4, 888 - XHAT . PV
 YHAT*R* 5.85* VECTOR( .760,  , 649,  , 888) ≤ 5.888 - YHAT . PV
-XHAT*R* 5.85* VECTOR( .760, .649,  , 888) ≤ 4.888 - -XHAT . PV
-YHAT*R* S.85* VECTOR( .760,  , 649,  .000) ≤ 5.989 - -YHAT . PV
 XHAT*R* 5.85* VECTOR( .760, .649, .000) ≤ 4.888 - XHAT . PV
 YHAT*R* 5.85* VECTORb.768,  , 649,  .000) ≤ 5.888 - YHAT . PV
-XHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤ 4.888 - -XHAT . PV
-YHAT*R* 5.85* VECTOR(-.760, .648, .000) ≤ 5.888 - -YHAT . PV
 XHAT*R* 5.85* VECTOR(-.760,  , 649,  , 898) ≤ 4.000 - XHRT . PV
-XHAT*R* S.85* VECTOR(-.760,-.648, .000) = -4.000 - -XHAT . PV
0 =  .000 -  ZHAT . PV
       WHERE R =  NILROTN*ROTN(-ZHAT,?)

ESTIMATE LIST:
ITEM426:
X:      -.000 TO  .288
Y:
Z:      -.001 -.550 TO TO  .550 .001
W:      -92.032*DEG TO -90.000*DEG
COS(W0) = -.023  SIN(W0) = -1.000
COS(DW) =  1.880  ☼  •  .023
```

Notice that we have **now** rid ourselves of the ambiguity in **the** gross orientation of **the box.**

Final **Problem**

We now proceed to add two more edge-to-surface **contacts:**

(contacts, *bxbtm, bjl.sb,* inside-of)
(contacts, *be9, bjl.s2,* extent_irrelevant)
(contacts, *be10, bjl.s3,* extent_irrelevant)
(contacts, *be11, bjl.s4,* extent_irrelevant)

and wind up with the final estimate:

```
-YHAT*R* 1.881, VECTOR( .707,-.707, .000) ≤ -.707
 XHAT*R* 1.00* VECTOR( ,787, ,787, .000) ≤ -.707
-XHAT*R* 1.00* VECTOR(-.707,-.707, .000) ≤ -.707
 YHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤ 5.000 - YHAT . PV
-XHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤ 4.888 - -XHAT . PV
-YHAT*R* 5.850 VECTOR(-.760,-.649, .000) ≤ 5.988 - -YHAT . PV
 XHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤ 4.899 - XHAT . PV
 YHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤ 5.888 - YHAT . PV
-XHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤ 4.888 - -XHAT . PV
-YHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤ 5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤ 4.888 - XHAT . PV
 YHAT*R* 5.85* VECTOR( .760, .649, .000) ≤ 5.000 - YHAT . PV
-XHAT*R* 5.85* VECTOR( ,768, .649, .000) ≤ 4.888 - -XHAT . PV
-YHAT*R* 5.85* VECTOR( .760, .649, .000) ≤ 6.888 - -YHAT . PV
 XHAT*R* 5.85* VECTOR( .760, ,649, .000) ≤ 4.888 - XHAT . PV
 YHAT*R* 5.85* VECTORL.768, .649, .000) ≤ 5.888 - YHAT . PV
-XHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤ 4.889 - -XHAT . PV
-YHAT*R* 5.855 VECTOR(-.760, .649, .000) ≤ 5.888 - -YHAT . PV
 XHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤ 4,888 - XHAT . PV
-YHAT*R* 5.85* VECTOR( .760,-.649, .000) . -5.888 - -YHAT . PV
 XHAT*R* 5.85* VECTOR( ,768, ,649, .000) = -4.880 - XHAT . PV
-XHAT*R* 5.85* VECTOR(-.760,-.649, .000) . -4.688 - -XHAT . PV
0 = ,888 - ZHAT . PV
       WHERE R = NILROTN*ROTN(-ZHAT,?)

ESTIMATE LIST:
ITEM442:
X:       .000 TO .000
Y:       .379 TO .381
Z:      -.001 TO ,881
W:      -92.632*DEG TO -92.603*DEG
cos (W0) . -.046 SIN(W0) = -.999
COS(DW) = 1.888 R = .000
```

A.2 Screw on Driver

This example illustrates use of, the differential approximation methods to estimate runtime errors. The task is insertion of a screw into a hole of our favorite box. The box is assumed to sit on the table, with possible displacement errors in the xy plane and rotation error about the z axis:

$$\Delta box = transl(\lambda\hat{x} + \mu\hat{y})*rot(\hat{z},\gamma)$$

w here

-0.3 inches ≤ λ ≤ 0.3 inches
-0.2 inches ≤ μ ≤ 0.2 inches
-5 degrees ≤ γ s 5 degrees

The screw is held on the end of a driver, as shown in Figure A.2, and the driver is held in the hand. We assume that errors in the driver's position with respect to the hand are negligible. However, the hand's position will only be assumed accurate to within 0.05 inch in displacement and 0.25 degree in orientation.

$$\Delta hand = transl(vector(\delta_x,\delta_y,\delta_z))*rot(\hat{x},\phi_x)*rot(\hat{y},\phi_y)*rot(\hat{z},\phi_z)$$

where

$$-0.05 \text{ inches} \le \delta_x, \delta_y, 8, \le 0.05 \text{ inches}$$
$$-0.25 \text{ degrees} \le \phi_x, \phi_y, \phi_z \le 0.25 \text{ degrees}$$

Likewise, the screw can wobble about the tip of the driver.

$$\Delta T_{ds} = rot(\hat{x},\alpha)*rot(\hat{y},\beta) \approx I + \alpha M_x + \beta M_y$$

where

$$-5 \text{ degrees} \le a \le 5 \text{ degrees}$$
$$-5 \text{ degrees} \le \beta \le 5 \text{ degrees}$$

We are interested in producing a parameterized estimate for ΔT_{ht}, the relation between the center of the hole and the tip of the screw. In this case, the system finds only one acyclic path of relations linking the hole and tip.

$$T_{ht} = hole^{-1}*tip$$
$$= (box*T_{bh})^{-1}*(hand*T_{hd}*T_{ds}*T_{st})$$
$$= T_{bh}^{-1}*box^{-1}*hand*T_{hd}*T_{ds}*T_{st}$$

where

T_{bh} = Location of hole with respect to box.
T_{hd} = Location of driver with respect to hand.
T_{ds} = Location of screw with respect to driver.
T_{st} = Location of tip with respect to screw.
box = Location of box in work station
hand = Location of hand in *work* station

In this case, the nominal values for these quantities are given by:

T_{bh} ≅ trans(nilrotn,vector(3.85,3.20,4.90)) (Distances in cm)

T_{hd} ≅ niltrans

T_{ds} ≅ trans(nilrotn,vector(0,0,25.4))

T_{st} ≅ trans(nilrotn,vector(0,0,3.18))

box ≅ trans(nilrotn,vector(45.7,101.6,0))

hand ≅ trans(rot($\hat{y}$,180*deg),vector(49.6,104.8,30.3))

All errors other than. those described above are assumed to be negligible. Using this information, application of the algorithm gives us a **parameterized** form for ΔT_{ht}:

$$\Delta T_{ht} = \text{transl}(\Delta p_{ht}) * \Delta R_{ht}$$

where

$$\Delta p_{ht} \cong \gamma * \text{vector}(3.20,3.85,0) + \phi_x * \text{vector}(0,28.6,0) + \phi_y * \text{vector}(-28.6,0,0) + \phi_z * \text{vector}(0,0,0) + \alpha * \text{vector}(0,3.18,0) + \beta * \text{vector}(-3.18,0,0) + \lambda\hat{x} - \mu\hat{y} + \delta_x\hat{x} + \delta_y\hat{y} + \delta_z\hat{z}$$

$$\Delta R_{ht} \cong I + \gamma M_z + \phi_x M_x + \phi_y M_y + \phi_z M_z + \alpha M_x + \beta M_y$$

Subject to constraints:

```
[ 1.00  ,  .000  ,  .000  ] . V1 ≤  .762
[ 1.00  ,  .000  ,  .000  ] . V1 ≥-.762
[ .000  ,  1.88  ,  .000  ] . V1 ≤  .508
[ .000  ,  1.88  ,  .000  ] . V1 ≥-.508
[ .000  ,  .000  ,  1.00  ] . V1 ≤  .873e-1
[ .000  ,  .000  ,  1.00  ] . V1 ≥-. 873e-1
[ 1.00  ,  .000  ,  .000  ,  .888  ,  .888  ,  .000  ] . V2 ≤  .127
[ .000  ,  .000  ,  .000  ,  .888  ,  .000  ,  .000  ] . V2 ≥-.127
[ 1.00  ,  1.00  ,  .000  ,  .000  ,  .000  ,  .000  ] . V2 ≤  .127
[ .000  ,  1.88  ,  .000  ,  .000  ,  .888  ,  .000  ] . V2 ≥-.127
[ .000  ,  .000  ,  1.00  ,  .000  ,  .000  ,  .000  ] . V2 ≤  .127
[ .000  ,  .000  ,  1.00  ,  .000  ,  .000  ,  .000  ] . V2 ≥-.127
[ .000  ,  .000  ,  .000  ,  1.88  ,  .000  ,  .888  ] . V2 ≤  .436e-2
[ .000  ,  .000  ,  .000  ,  .000  ,  .000  ,  .000  ] . V2 ≥-.436e-2
[ .000  ,  .000  ,  .000  ,  1888  ,  1.00  ,  .000  ] . V2 ≤  .436e-2
[ .000  ,  .688  ,  .000  ,  .000  ,  1.88  ,  .000  ] . V2 ≥-.436e-2
[ .000  ,  .888  ,  .000  ,  .000  ,  .000  ,  1.00  ] . V2 ≤  .436e-2
[ .000  ,  .000  ,  .000  ,  .000  ,  .000  ,  1.80  ] . V2 ≥-.436e-2
[ 1.00  ,  .888  ] . V3 ≤  .873e-1
[ 1.00  ,  .000  ] . V3 ≥-.873e-1
[ .000  ,  1.00  ] . V3 ≤  .873e-1
[ .000  ,  1.88  ] . V3 ≥-.873e-1
```

where

$v1 = [\ \lambda, \mu, \gamma\ ]$
$v2 = [\ \delta_x, \delta_y, \delta_z, \phi_x, \phi_y, \phi_z\ ]$
$v3 = [\ \alpha, \beta\ ]$

We are interested in finding the maximum displacement errors in the plane of the hole (Ax and Δy) and along the axis of the hole (Az). These quantities are given by the objective functions:

$Ax = [\ 3.28,\ .000,\ -28.6,\ .000,\ .000,\ -3.18,\ 1.00,00,\ .000,\ .000\ ]\ .\ V$
$\Delta y = [\ 3.86,\ 28.8,\ .000,\ .000,\ 3.18,\ .000,\ .000,\ -1.88,\ .000,\ 1.88,\ .000\ ]\ .\ V$
$\Delta z = [\ .000,\ .000,\ .000,\ .000,\ .000,\ .000,\ .000,\ .000,\ .000,\ .000,\ 1.00\ ]\ .\ V$

where

$V = [\ 7,\ \phi_x, \phi_y, \phi_z, \alpha, \beta, \lambda, \mu, \delta_x, \delta_y, \delta_z\ ]$

Solving these linear programming problems, the system gets

$-1.57 \le Ax \le 1.57$	(1.57 cm ≃ 0.62 inches)
$-1.37 \le Ay \le 1.37$	(1.37 cm ≃ 0.54 inches)
$-.127 \le At \le .127$	(.127 cm = 0.05 inches)

Also, we need to know the maximum direction error between the screw and hole axes. This quantity will be given by:

$$\Delta\theta \simeq \max\ (|\Delta\theta_x|, |\Delta\theta_y|)$$

where

$\Delta\theta_x = [\ .000,\ -1.88,\ .000,\ .000,\ -1.88;\ .000\ ]\ .\ V$
$\Delta\theta_y = [\ .000,\ .000,\ -1.00,\ .000,\ .000,\ -1.00\ ]\ .\ V$

and

$V = [\gamma, \phi_x, \phi_y, \phi_z, \alpha, \beta]$

Solving gives us:

$-.0916 \le \Delta\theta_x \le .0916$	(0.0916 radians ≃ 0.525 degrees)
$4916 \le \Delta\theta_y \le .0916$	

V. CASE STUDY OF ASSEMBLY OF A PENCIL SHARPENER

M. Shahid Mujtaba

Artificial Intelligence Laboratory
Computer Science Department
Stanford University

The author is a graduate student in the Industrial Engineering Department.

INTRODUCTION

A mechanical pencil sharpener was assembled using the Stanford Arm to gain insight into analyzing the mechanical assembly process. The process can be considered as a sequence of motions of the component parts; these motions in turn dictate the need for a sequence of motions of the manipulator hand.

The software system used for programming hand motions is of considerable importance in determining the ease with which a manipulator can be used, and the path along which it moves. For analyzing the hand motion times, however, software is much less important than hardware, since the hardware characteristics of the manipulator largely determine the speed and types of motions which may be made. It is believed, therefore, that whether the pencil sharpener is assembled using WAVE or AL will not significantly affect the types of motions used or the speed of execution. We plan to check this conjecture by assembling 'the pencil sharpener both in WAVE and in AL. The results reported here use WAVE.

The main results which emerged from this study are these:

a) Special purpose fixtures were desirable for holding the parts in place so that the manipulator could work on them. Plaster of Paris fixtures were easy to design and produce, relatively cheap, and adequate for the purpose.

b) Positioning of parts could often be accomplished more readily by dropping the parts and tapping them into place than by trying to position them accurately.

c) Analysis of the movements made by the manipulator showed them to be similar to human movements as defined by Methods Time Measurement (MTM),[2,3] but since the mechanical arm was larger, clumsier and less versatile,' and had to avoid objects in the path of movement, had to check the precise location of the spindle in its hand, and had to grope for the hole to insert the spindle shaft in, total assembly by the manipulator took longer than by a human by a factor of 8 times when the assembly was done, neglecting overhead. The same factor was'expected from theoretical 'analysis.

DESCRIPTION OF THE ASSEMBLY TASK

The parts of the assembly are shown in Figures 1, 2, and 3. Four parts were assembled together - the handle (crank), body of the sharpener (base), spindle (assembled with the cutters in place), and the shell.

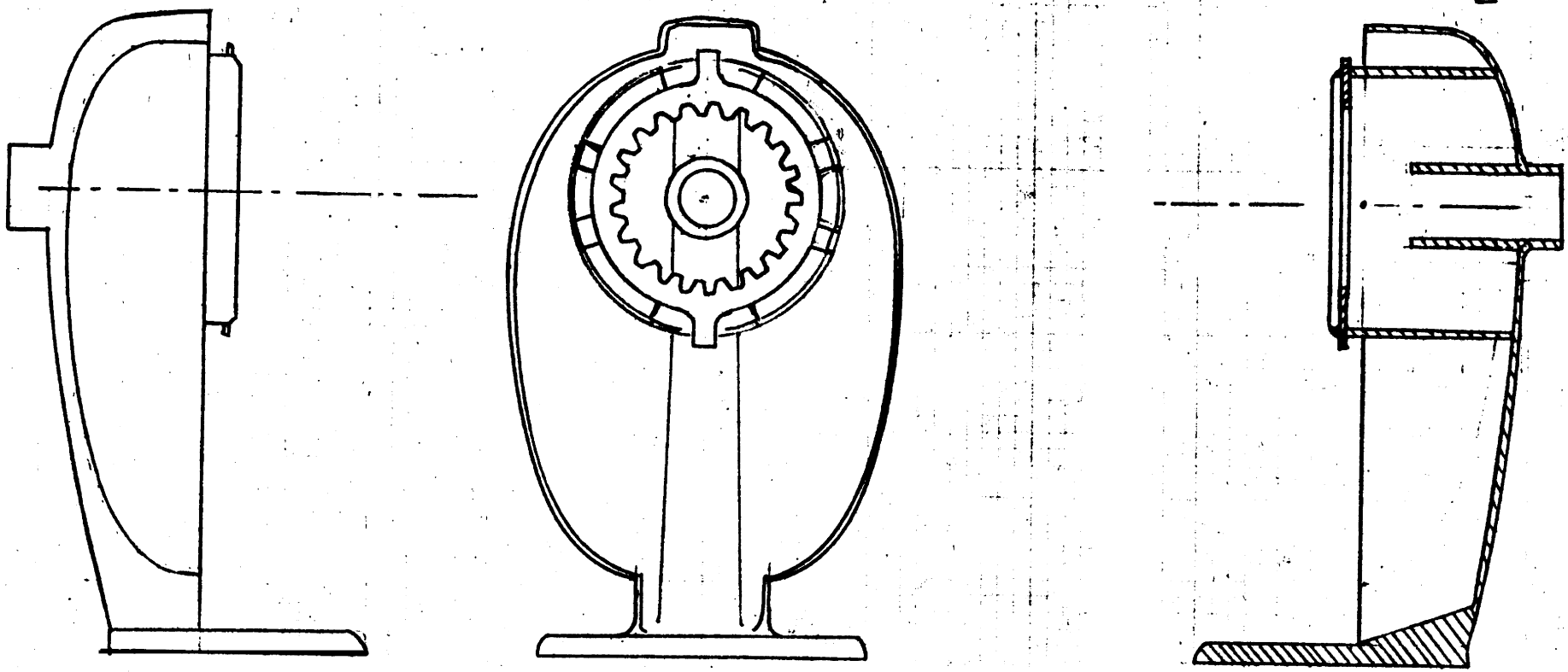


Figure 1: Base of Pencil Sharpener

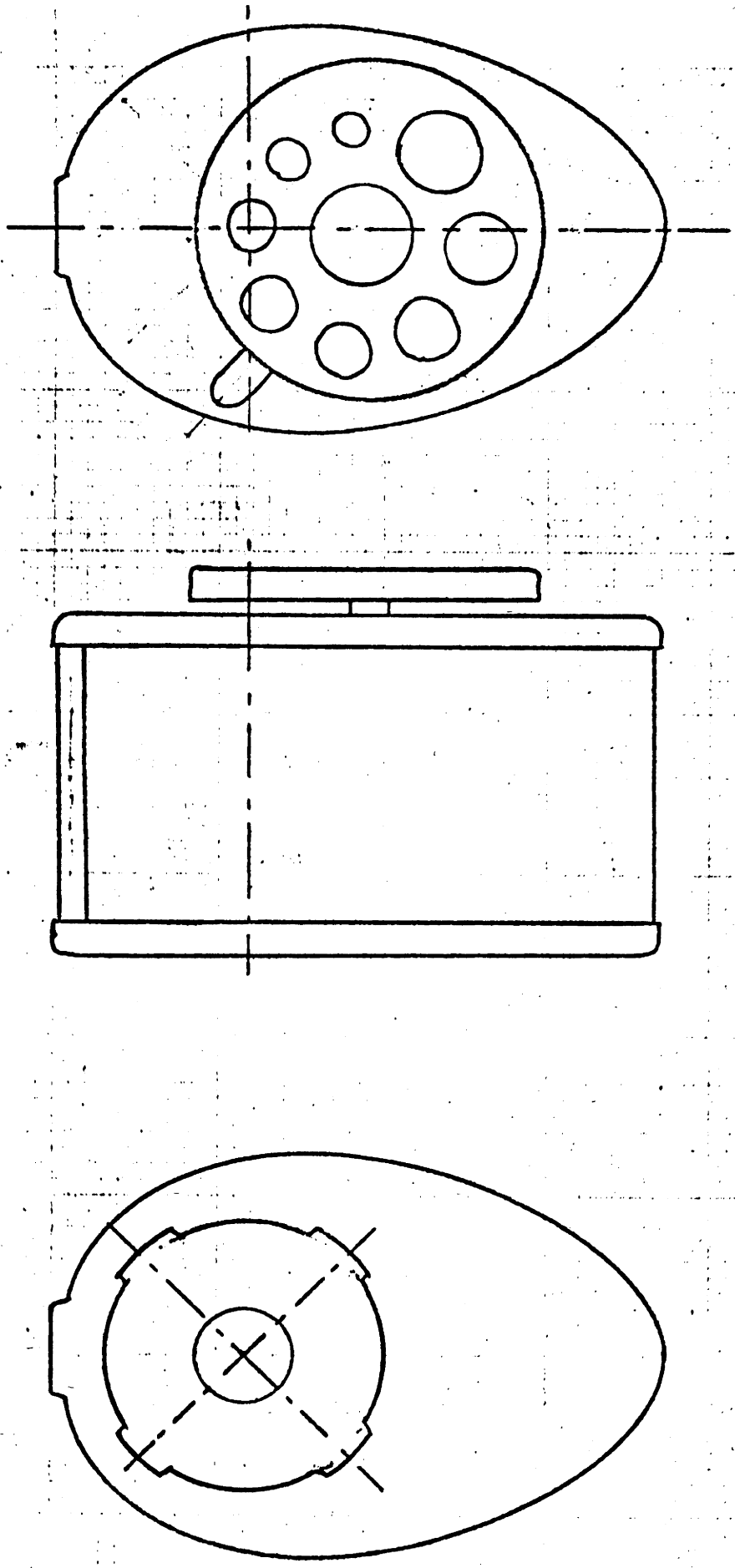


Figure 2: Shell of Pencil Sharpener

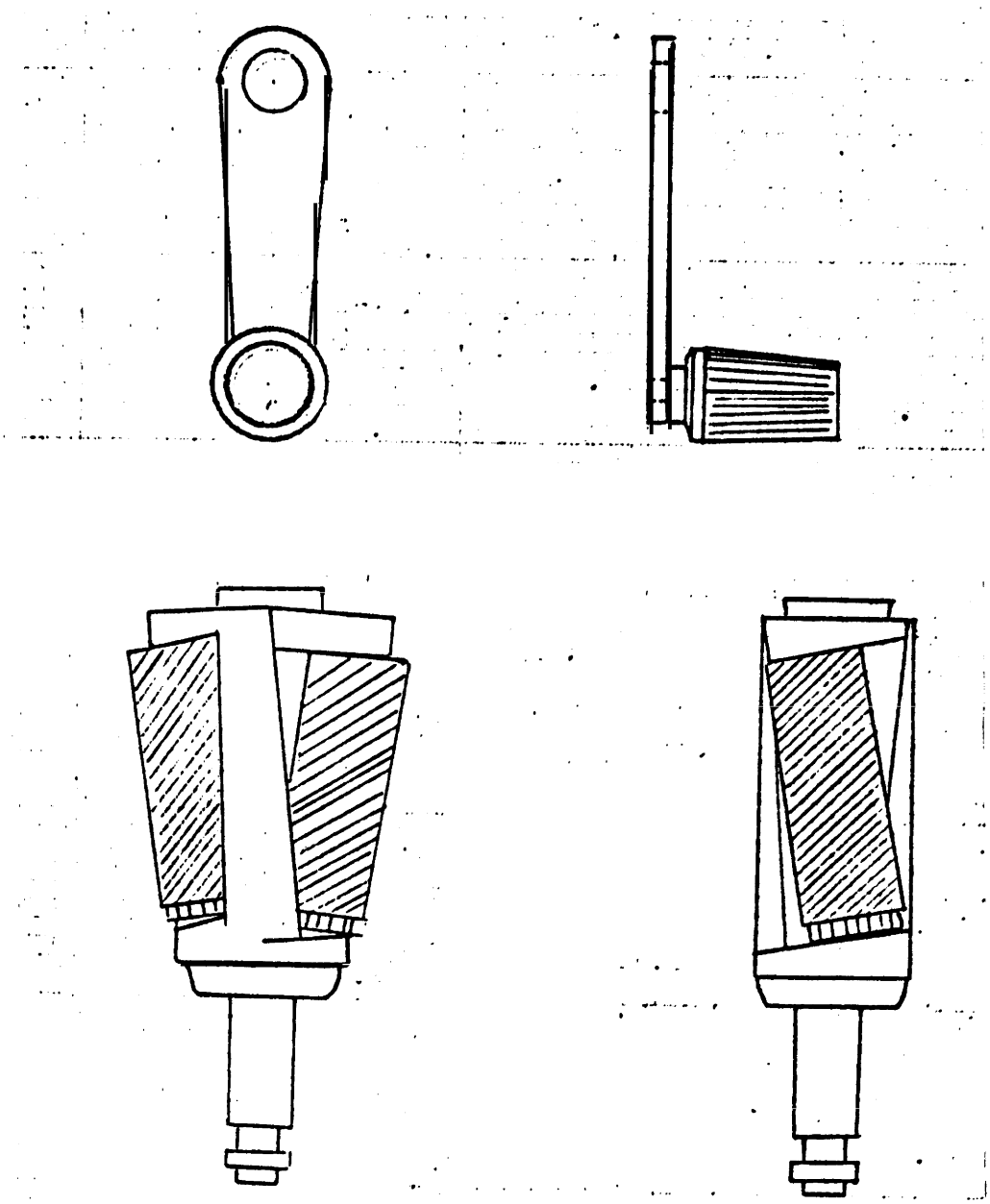


Figure 3: Handle and Spindle of Pencil Sharpener

The whole assembly task was broken up in broad terms as follows:

a) get and position handle (crank)
b) get and' position base (body)
c) get and insert spindle shaft through hole in base
d) screw spindle into hole in crank
e) get and position shell of sharpener against base, placing it over spindle
f) seat shell and turn it 45 degrees into place
g) bring arm back to initial position

FIXTURES

A fixture"' is a holding device which supports the workpiece in a fixed orientation with. respect to the tool (in this case the manipulator hand). Each fixture has locators to position the workpiece and clamps to hold it rigidly.

A free rigid body has three degrees of freedom of rotation and three degress of freedom of translation. Locators restrict these six degrees of freedom in order to give points of reference. As shown in Figure 4, the workpiece would lose three degrees of freedom when placed and maintained, on the locators lettered (A); locators lettered (B) restrict another two degrees of freedom, and locator (C) restricts the last degree of freedom. The form of the locator selected depends on the condition of the reference surface of the part; finished surfaces can be supported on a surface rather than suspended on points, while rough surfaces are given as few points of contact as deemed necessary for stability of the part. Clamps hold the workpiece firmly against the locators provided and resist all forces introduced by the operation.

In the assembly of the pencil sharpener, fixtures were used to locate the parts precisely; since the forces encountered in the assembly process were small (much smaller than in the case of machining), clamping was not done by external clamps. Instead the manipulator was used to provide the necessary reaction against the locators.

Fixtures were made by casting plaster of Paris in a box, and dipping the parts, suitably covered with modelling clay and masking tape, and coated with a thin layer of petroleum jelly into the plaster to make a mold for the part. The plaster was then machined to provide space for the manipulator fingers to be inserted around the part to be gripped. Since the surfaces of the sharpener were smooth, surface contact was used instead of point contact. Drawings of some of the fixtures used are shown in Figures 5, 6, and 7.

The advantage of using plaster of Paris was that making the initial mold and machining was a very simple and economical process which did not require specialized tools.

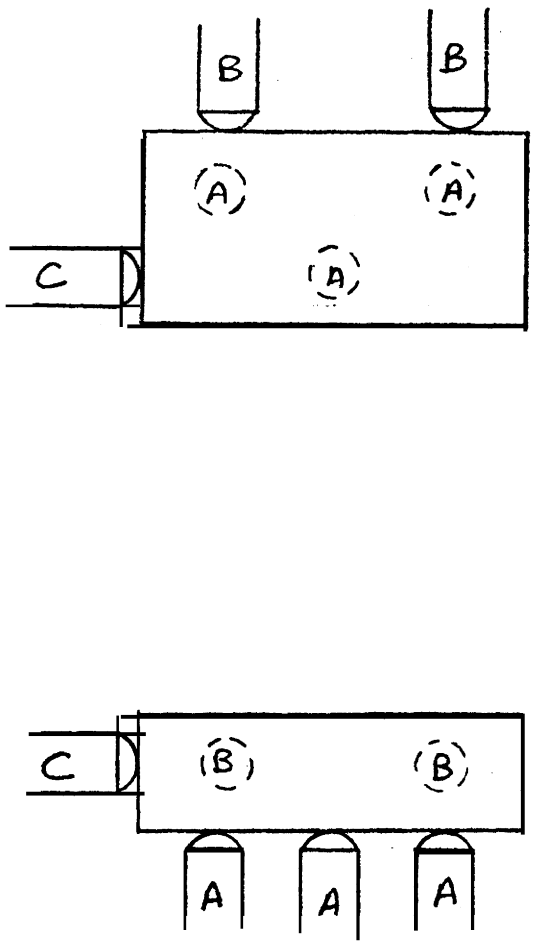


Figure 4: Placement of Locators

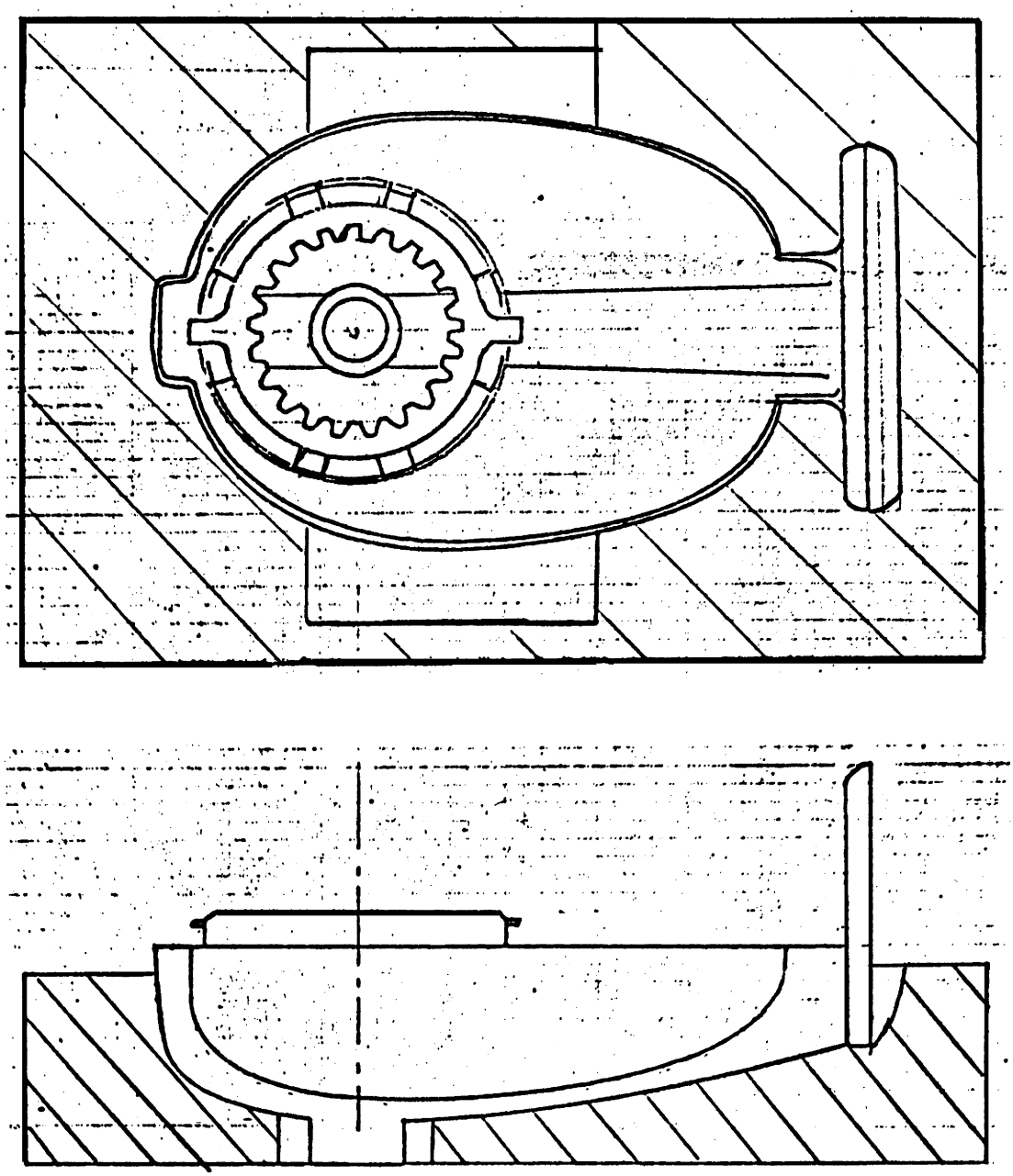


Figure 5: Fixture for Pencil Sharpener Base

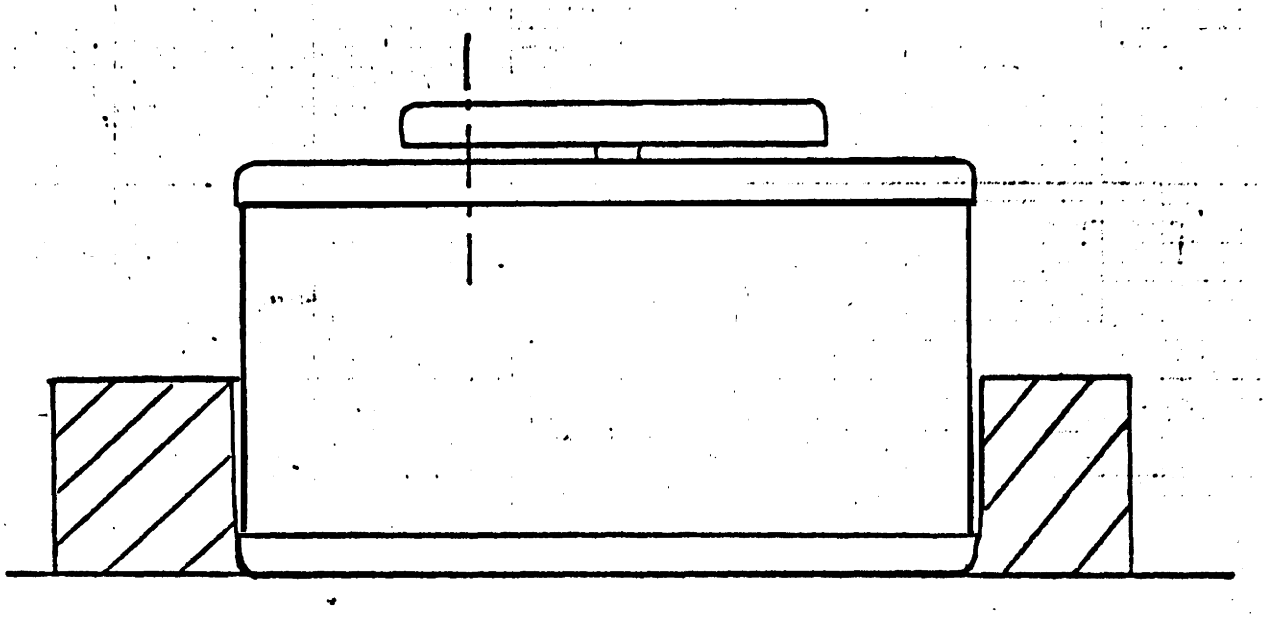


Figure 6: Fixture for Pencil Sharpener Shell

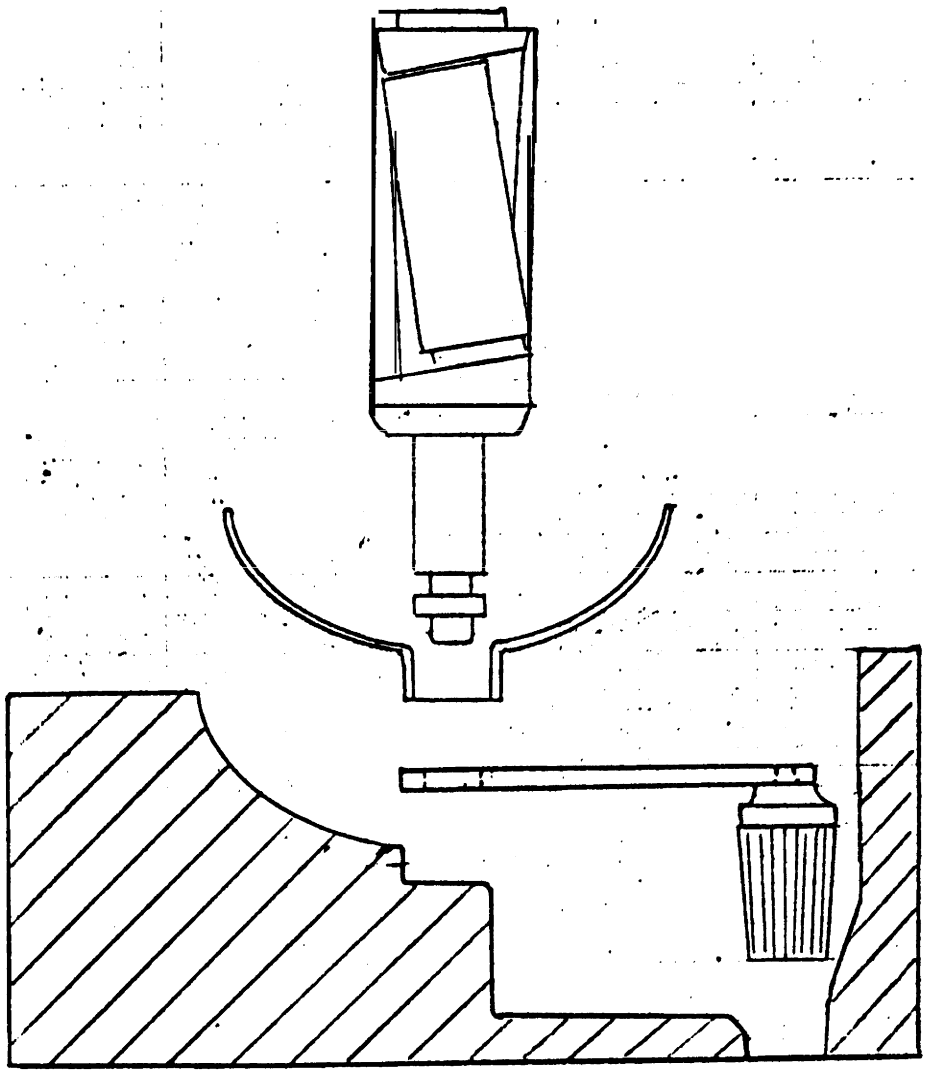


Figure 7: Section Through Main Fixture

Overmachining of the plaster resulting in too much iossage could be easily corrected by the application of more wet plaster which was later allowed to dry. The biggest disadvantage of using plaster as the potting material was that it chipped very easily, and each time the fixture was used, a little bit of it wore out or broke off, resulting in more uncertainty of the locations. This disadvantage could be overcome by using other potting compounds instead.

Another possible improvement in fixtures would be to design them to allow functional inspection. By putting sensors in the fixture at the appropriate places, it would be possible to tell if the object has fallen in at the right position. An interesting but unanswered question is whether or'not a "universal" fixture can be designed.

ASSEMBLY OF THE SHARPENER

The handle was located with respect to the edge at the hole end. It was placed in the fixture, and pushed until it touched the side of the fixture. The base fixture had parts removed to ensure that the bottom or the back end did not bind against the base when the base was lifted. In fact, without those parts removed, the fixture came up with the base when. the latter was lifted! The shell fixture did not need to have any parts removed, since it held the shell securely at the bottom.

Problems were encountered with the original main fixture in which the assembly was done. When the handle was being positioned in the fixture, either the hole end or the roller end , touched the fixture and then the orientation of the handle was lost. As a result, the handle ended up at unpredictable places, making correct positioning impossible. To correct this problem, the well for the roller handle was tapered,. as shown in Figure 7, so that when the handle was dropped from about half an inch above the well, it landed in. roughly the right place, and just needed to be tapped into place by having the manipulator hand rest on it and drag it towards the center by friction until the hole end of the crank was flush with the fixture. The base was put into place by wobbling it a little while moving it down, and stopping motion of the arm when a force was encountered; after releasing the base, the hand was lifted, closed, and tapped down on the base.'

The spindle presented special problems since the clearance between the shaft and the hdle into which it was inserted was 0.004 inches while the arm reading was given in terms of 0.01 inches; although noise in the A/D 'channels and devices resulted in an uncertainty of 0.04 inches. (This meant that two successive readings without movement would indicate the arm to have moved by as much as 0.04 inches.) The spindle shaft was touched against the sides of the main fixture in order to locate more precisely the position of the hole, and then a spiral search in steps of '0.04 inches was done to. actually insert the spindle shaft into the hole. While this small step may appear to be very close to the uncertainty of the A/D channels, it had to be used since larger steps would have. resulted in the arm making serious overcorrections. Once the first part of the insertion had been accomplished (as evidenced by

not encountering reaction at the end of the spindle), the spindle was pushed down and twisted at the same time in order to seat it without binding. It was then twisted into the handle hole by applying a downward force and turning the hand through 360 degree revolutions.

The shell was lifted vertically out of its fixture and positioned over the assembled spindle, then lowered in place and released at a height of 0.2 inches from the mating surface, regrasped and wobbled gently while being pressed down in order to ensure seating. It was then turned 45 degrees to finish the final assembly, stopping when a resisting moment was encountered.

COMPARISON OF ASSEMBLY BY MANIPULATOR VERSUS ASSEMBLY BY HUMAN

TIME DATA

The compiled program of about 20k bytes was able to assemble the pencil sharpener in 2.4 minutes. Of this time, about 1.8 minutes or 108 seconds was actual CPU time, mainly for servoing. The rest was overhead due to interprocessor communication and loss of the processor in time-sharing mode. A theoretical estimate of the time taken for the arm to perfotm the assembly also gave 108 seconds, assuming continuous motion and disregarding lossage of the processor and overhead. Detailed analysis of the assembly procedure by the manipulator is given in Appendix 2, and it should be noted that much time is spent in opening and closing the hand, centering over the object, and trying to verify the position of *the hole.

Estimates of a human operator using one hand on the basis of MTM data to do the same assembly showed that it took 14 seconds (verified by the author taking 15 seconds to do the job), a factor of 8 times faster than the manipulator. It must be remembered that the manipulator did not utilize vision, for help as a human operater does, and was thus akin to a blindfolded, one armed, two fingered human operator doing the job. A table showing the analysis of the MTM study for the human operator is given in Appendix 1.

It should be noted that the assembly procedure does not represent the optimum sequence of movements, or placement of the component parts initially. The determination and elimination of inefficiencies would mean running the system at the limit of its capability, which would result in reduced assembly time.

ASSEMBLY AND MOTION PRIMITIVES

MTM and Draper use assembly primitives in studying the sequence of tasks involved in putting an assembly together, while WAVE uses motion primitives to specify arm motion. The former primitives are descriptive in nature since they describe the actions performed,

but not the motion for the manipulator to achieve the action. WAVE primitives are strategic, since they specify where and how the arm has to move rather than what the assembly task is.

MTM ASSEMBLY PRIMITIVES[2,3]

MTM primitives generally consist of several parts: the assembly primitive, the distance involved, and. specific cases involved in the the assembly primitive. For example, R 24 D means to REACH 24 inches to an object in a fixed location, or to an object in the other hand, or to an object on which, the other hand rests., The following is a partial list of primitives, their abbreviations and a description. A fuller description of the specific cases is given in Appendix 1.

REACH(R)	Move hand to a destination or general location.
MOVE(M)	Transport an object to a destination.
TURN(T)	Turn the hand by a movement that rotates the hand,' wrist, and forearm about the long axis of the forearm.
GRASP(G)	Secure sufficient control of one or more objects with the hand.
POSITION(P)	Align, orient and engage one object with another when the motions are minor.
RELEASE LOAD(RL)	Relinquish control of an object by the fingers or hand.
APPLY PRESSURE(AP)	Apply force along the axis of the forearm.
TURN & APPLY PRESSURE(T & AP)	TURN and APPLY PRESSURE are tabulated together in MTM tables.

Draper Lab ASSEMBLY PRIMITIVES[4]

While Draper Lab has defined 9 main primitives and 9 subprimitives for "ACCOMMODATE", only the ones used in this paper are described below:

GRASP	Device uses tool to grasp part(s) to be assembled or to grasp another tool.
POSITION	Device executes gross motion trajectory carrying tools and/or parts.
INTERFACE	Device goes from state of no contact between tool or carried parts and other parts to a state of contact: i.e. device touches something,"makes contact".
RELEASE	Device causes tool to release its grasp on part or other tool.
RETURN	Device returns tool to storage area,
ACCOMMODATION	Device allows the forces between parts to modify the motion of parts according to one of the following subprimitives:

COMPLEX ACCOMMODATE	Accommodation executed during **a** complex motion having no convenient name to describe motion.
INSERT	Push shaft into hole.
DEPRESS	Deflect a part in its compliant direction.

WAVE MOTION ***PRIMITIVES***[7]

The following **is a** list of WAVE motion primitives used in programming the arm.

PARK	Generates a trajectory to the PARK (at rest) position.
GOTO	Generates a three part trajectory consisting of the departure, center and approach segments.
GO	One part direct move without liftoff or **setdown.**
MOVE	Same as **GOTO** except that **a** smooth trajectory is fitted through the three segments.
CHANGE	Generates **a** trajectory for differential motion.
PLACE	This causes the hand to move down until it meets some resistance.
OPEN	Opens the hand.
CLOSE	Closes the hand.
CENTER	Closes the hand centrally over the object to be grasped.

WAVE ASSEMBLY PRIMITIVES

The following primitives, similar to 'motion primitives, were used to describe the assembly using the **mechanical arm:**

CENTER* and *CLOSE	Results in the hand grasping the object.
GOTO(MOVE)	Three part move that uses GOT0 primitive to make a gross motion.
POSITION	Motion which allows some form of mating of one object with another, or adjusts the position of the hand so that the next motion can be easily executed.
OPEN	Has the same effect as releasing the object.
WAIT	A short pause between movements.
TURN & APPLY PRESSURE	Turning about an axis while applying force along the **axis**

COMMENTS

It should be noted that the comparisons are made between *assembly* primitives rather than *motion,* primitives, except where it is of interest to show correspondence between them. As there was no one to one correspondence with MTM and WAVE primitives because similar motions could be specified in several ways in WAVE, it was decided to use primitives similar to those in MTM for the mechanical arm. Different fonts are used when referring to different primitives to enable easier recognition of what the primitives are. The fonts are summarized below:

MTM ASSEMBLY
DRAPER ASSEMBLY
WAVE MOTION
WAVE ASSEMBLY

No distinction is made between **MOVE** and **REACH** in the case of the mechanical arm, since the parts are so small and light compared with the arm that it does not make a difference in the movement whether the arm is carrying anything or not. ***WAIT***s are tabulated, since these were explicitly inserted for the purpose of preventing the overlapping of consecutive movements which tended to cause unpredictable results. For instance the ***WAIT*** after dropping the handle ensured that the drop was not affected by the hand closing before the handle had a chance to drop in place. The ***CENTER*** and ***CLOSE*** operations are equivalent to the **GRASP** of the human but take much longer to do. ***OPEN*** for the manipulator is equivalent to **RELEASE** performed by the human operator except that it is not quite as gentle, and the hand usually opens quite suddenly.

The ***POSITION*** (obtained generally by GO) in the case of the mechanical arm is almost the same as ***GOTO(MOVE)*** and could have been considered the same and tabulated together. The ***GOTO(MOVE)*** was movement to an approach point, while the ***POSITION*** (mainly GO) was a one part directed move to the location of the grasping position - a smooth move tended to cause a collision with the object being grasped even when the direction of approach was well defined, since the arm did not successfully null out errors in all the six joints at the end of the allotted motion time - a one part downward directed move required only the movement of three joints; joint 2 to lower the arm, joint 3 to extend the boom so that the hand would move down vertically, and joint 5 to keep the hand approach vector vertical.

ASSEMBLYDATA

NUMERICAL BREAKDOWN OF ASSEMBLY ELEMENTS FOR ASSEMBLY BY HUMAN

	GRASP	REACH	MOVE	POSI-TION	RELE-ASE	TURN + APPLY PRESSURE
PUT HANDLE	1	1	1	1	1	
PUT BASE	1	1	1	1	1	
PUT SPINDLE	7	1	2	1	6	12
PUT SHELL	1	1	1	1	1	1
MOVE BACK		1				
TOTAL	10	5	5	44	9	13

TIMES REQUIRED FOR THE ELEMENTS (JIFFIES)

(60 jiffies = 1 second)

	GRASP	REACH	MOVE	POSI-TION	RELE-ASE	TURN + APPLY PRESSURE
PUT HANDLE	4	32	29	22	4	
PUT BASE	4	27	40	45	4	
PUT SPINDLE	30	32	62	45	26	244
PUT SHELL	4	30	51	45	4	8
MOVE BACK		38				
TOTAL	42	159	182	157	38	252
AVERAGE	4	32	36	39	4	19

ESTIMATED TOTAL ASSEMBLY TIME = 830 JIFFIES = 13.9 SECONDS

BREAKDOWN OF ASSEMBLY ELEMENTS FOR YELLOW ARM USING WAVE

	CENTER + CLOSE	GOTO (HOVE)	POSI-TION	OPEN	TURN + APPLY PRESSURE	WAIT
PUT HANDLE	2	3	3	2		1
PUT BASE	2	4	3	2		
PUT SPINDLE						
GET SPINDLE	1	1	1	1		
PLACE SPINDLE	1	6	3	1	1	
TURN SPINDLE	3			3	18	
ASSEMBLE SHELL	2	2	3	2	1	2
PARK ARM		2		1		
TOTAL	ii	18	13	12	20	3

TIMES FOR ASSEMBLY BY YELLOW ARM USING WAVE (JIFFIES)

	CENTER + CLOSE	GOTO (MOVE)	POSI-TION	OPEN	TURN + APPLY PRESSURE	WAIT
PUT HANDLE	70	370	240	80		50
PUT BASE	110	450	330	80		
PUT SPINDLE						
GET SPINDLE	140	140	45	0		
PLACE SPINDLE	65	1125	330	45	80	
TURN SPINDLE	90			90	1170	
ASSEMBLE SHELL	230	580	150	20	80	100
PARK ARM		170		40		
TOTAL	705	2835	1095	355	1330	150
AVERAGE TIME	64	158	84	30	67	50

ESTIMATED ASSEMBLY TIME.- 108 SECONDS

BREAKDOWN OF ASSEMBLY TASKS BY YELLOW ARM USING DRAPER ANALYSIS

	GRASP	POSITION	PRE-CISE POS	REL-EASE	ROT-ATE	INT-ERF-ACE	WAIT	RET-URN	ACCO-MMOD-ATE	DEP RES S	INS ERT	CPX ACC *
PUT HANDLE	2	3	2	2			1		1	1		
PUT BASE	2	4	2	2					1	1		
PUT SPINDLE												
GET SPINDLE	1	1	1	1								
PLACE SPINDLE	1	6		1		2			2		1	1
TURN SPINDLE	3			3	18							
ASSEMBLE SHELL	2	2	1	2			2		3			3
PARK ARM				1				1				
TOTAL	11	16	6	12	18	2	3	1	7	2	1	4

ASSEMBLY TIMES FOR YELLOW ARM BY DRAPER ANALYSIS (HFFIES)

	GRASP	POSITION	PRE-CISE POS	REL-EASE	ROT-ATE	INT-ERF-ACE	WAIT	RET-URN	ACCO-MMOD-ATE	DEP RES S	INS ERT	CPX ACC *
PUT HANDLE	70	370	190	80			50		50	50		
PUT BASE	110	450	190	80					140	140		
PUT SPINDLE												
GET SPINDLE	140	140	45	0								
PLACE SPINDLE	65	1125		45		250			160		80	80
TURN SPINDLE	30			30	1170							
ASSEMBLE SHELL	230	580	35	20			100		195			195
PARK ARM				40				170				
TOTAL	705	2665	460	355	1170	250	150	170	545	190	80	275
AVERAGE TIME	64	167	77	30	65	125	50	170	78	35	80	63

ESTIMATED ASSEMBLY TIME = 108 SECONDS

* Complex accommodate

DISCUSSION

COMPARISON OF MOTION PRIMITIVES

A comparison of different motion primitives reveals several interesting features. More motions are required by the mechanical arm than the human arm (64 vs 36, or 78% more), since the mechanical arm cannot perform complex motions as easily as the human arm; motions have to be broken up in order to prevent the hand from hitting something when it tries to null out errors and mechanical arm motions take longer to complete especially when nulling out errors.

Consider ***GRASP vs CENTER*** and ***CLOSE.*** 42 jiffies (0.7 second) are required by the human operator for IO **GRASPs** during the whole assembly while the mechanical arm requires 805 jiffies (13.4 seconds) for 1 I ***CENTER*** and ***CLOSEs***. Average motion times were 4 vs 64 jiffies (0.07 vs 1.07 seconds), or a factor of 16 times. Grasping is thus performed a lot more quickly by the human arm than by the mechanical arm. The human operator can move his fingers according to what he sees, while the mechanical arm in the ***CENTER*** operation closes the hand until one touch sensor is triggered, and then moves the arm until both sensors are triggered. By moving only small inertias, the human operator is able to accomplish the GRASP much more quickly than the mechanical arm. While the difference in total number of **GRASPs** and ***CENTER*** and ***CLOSEs*** may be small, their distribution between the different tasks of the assembly is different. The human hand cannot rotate through 360 degrees, and 2 **GRASPs** and 2 **RELEASEs** need to be done for one done by the mechanical arm rotating through 360 degrees. However, the mechanical hand has to release the object and close the fingers before it can tap the part in place - unlike the human who can position the object precisely while holding it all the time.

The human operator performs IO **REACHes** or **MOVEs** in 341 jiffies (5.68 seconds) compared to the 18 ***GOTO(MOVE)*** by the mechanical arm in 2835 jiffies (47.25 seconds) which amounts to 34 vs 160 jiffies (0.57 vs 2.67 seconds) per movement. The human arm performs faster than the mechanical arm by a factor of 5, white the mechanical arm does 2 times as many movements as the human arm when it is trying to position the spindle in place. The reason is that the human operator does not need to worry about nulling out errors, and utilizing visual feedback, does not have to spend time trying to locate the relative positions between the spindle and the hole precisely, something which the mechanical arm requires 4 ***MOVEs*** and 2 ***POSITIONs*** to accomplish, and in addition the use of movements that are too rapid result in overloading the motors with a demand torque that is too high.

The human operator performs 4 **POSITIONs** in 157 jiffies (2.62 seconds) compared to 13 ***POSITIONs*** in 1095 jiffies (18.25 seconds) performed by the mechanical arm, i.e. 39 vs 85 jiffies (.65 vs 1.42 seconds) per movement which means a factor of 2 in speed and a factor of 3 in number of movements. In each of the assembly subtasks the mechanical arm does three

times as many ***POSITION***s as the human arm, since it has to get itself vertically over the part before grasping and vertically above the main fixture before releasing the part, and then actually position the part in place.

The human arm does 9 **RELEASE**s in 38 jiffies (.63 second) compared to 12 ***OPEN***s in 355 jiffies (5.92 seconds) by the mechanical arm, i.e. 4 vs 30 jiffies (.07 vs .5 second) per motion which means a factor of 7 in speed and a factor of 1.3 in number of movements.) The human does twice as many **RELEASE**s in the turning of the spindle as the mechanical arm, just as it did twice as many **GRASP**s, but in the placement of the parts, the mechanical arm does twice as many ***OPEN***s, since the mechanical arm must first open the right amount to grasp the part, then do a second ***OPEN*** to release the part.

The mechanical arm requires 3 ***WAIT***s after opening the hand to ensure that the subsequent motion does not overlap with the opening of the hand. Waits are not tabulated for' the human arm since the human operator does not consciously have to take any discernable pauses between overlapping motions.

All the motions seem to be fundamental and necessary in the mechanical assembly, except for the 3 ***WAIT***s which took 150 jiffies (2.5 seconds), and trying to locate the position of the hole which took 4 ***MOVE***s and 2 ***POSITION***s and 1140 jiffies (19 seconds) of assembly time. Eliminating these items would have resulted in a time saving of 21.5 seconds or roughly 20% Spiral searching for the hole was not considered since the arm performed this operation only some of the time.

VALIDITY OF MOTION PRIMITIVES FOR THE MECHANICAL ARM

The analysis done has tried to model mechanical arm motion primitives in the light of motion primitives known for the human arm enabling a direct comparison of the two. It is apparent that the process of programming the manipulator to do the assembly the way a human being does required special techniques in positioning and force and touch sensing which the human operator takes for granted. The human operator makes use of vision, which enables him not only to precisely locate the parts, but also to avoid obstacles, and perform smooth and precise motions. Having more fingers and additional degrees of freedom over the mechanical manipulator enables the human operator to perform motions without having to go through the contortions the manipulator does. For instance, the mechanical arm has to turn through 90 degrees when the hand touches the top and sides of the main fixture to maintain the fingers parallel to the surface being touched, since otberwise the spindle would tilt in the hand in the plane of the fingers and lose its orientation.

COMPARISON OF MOTION PRIMITIVES WITH THOSE OF DRAPER ANALYSIS

The similarities between the assembly primitives used by Draper and MTM and relationships to WAVE are shown in the following table:

Draper	MTM(HUMAN)	MTM(YELLOW)	WAVE
RELEASE	RELEASE	*OPEN*	OPEN
GRASP	GRASP	*CENTER* *CLOSE*	CENTER CLOSE
POSITION	MOVE, REACH	*GOTO(MOVE)*	**GOTO,** MOVE
ROTATE	**T &** AP	***T & AP***	CHANGE
INTERFACE, ***ACCOMMODATE,*** ***CPX ACCOM,*** ***INSERT,*** ***DEPRESS,*** ***[PRECISE*** ***POSITION]***	POSITION	*POSITION*	PLACE, CHANGE, GOT0
RETURN	REACH	***GOTO(MOVE)***	PARK
[WAIT]	[WAIT]	*WAIT*	WAIT

Elements is square brackets **[]** indicate elements that were necessary as assembly primitives but were unavailable.

PRECISE POSITIONING OF THE MANIPULATOR

The time taken for the manipulator to null out position errors, while not obvious in the analysis, slowed down the assembly process. To speed up the assembly, the motions which did not require precise positioning were performed without nulling out the final position errors. Of special importance was screwing in the spindle since software limitations required that the rotation be made in steps of 120 degrees. To null out the error at the end of each 120 degree twist meant that the motor ground away to achieve the last bit of unnecessary precision. Not nulling the movement resulted in the handle turning a few degrees more or less than the desired amount, but this was not at all critical.

It was found that asking the arm to move directly to the part to be picked up inevitably resulted in the collision of the fingers with the part even though a vertical approach vector had been specified, since although feedback was used to correct errors, at the end of the allotted motion time there were errors in some of the joints which had to be corrected. While the fingers did eventually get to the position desired, they did so with a lot of pressure and hard pushing against the part, since the approach vector may tend to be tilted slightly from the direction of force application. To overcome this problem the arm was asked to go to a point vertically above the part to be picked up and then told to swoop down upon the part in a vertical motion so that there was no danger of lateral movements of the fingers hitting the part, since joint I, the motor at the shoulder, did not move.

The arm performed differential motion precisely when the movement involved only one joint and the change was of the nature of the joint movement, e.g., angular motion could be performed precisely by the rotary joints as long as the joint axis was parallel to the axis of the desired rotation. This was illustrated particularly when the hand performed rotation precisely around the z-axis when the wrist was vertical, but tended to change the wrist orientation when told to make a differential vertical motion.

FORCE CONTROL OF THE MANIPULATOR

Paul[5] has shown that the arm can exert forces with a typical tolerance of 10 oz. Depending on the motor used, the tolerance could be worse. This imprecision of the force application and measurement caused problems where these should not have occurred. Firstly, low contact forces of the order of 2 or 3 oz were dominated by the noise force. Secondly, the manipulator tended to apply more force than necessary or specified, especially in the sideways direction when trying to locate the hole position by touching the sides of the fixture, and at times caused a slight movement or tilt of the spindle in the hand that resulted in difficulty later when insertion of the spindle into the hole was attempted. The magnitude of force applied in the downward direction did not matter so long as buckling did not occur, or the spindle did not tilt, since the reaction of the table prevented any movement in the vertical direction. Draper[4] has shown that such forces are important to the extent that jamming occurs, and this is discussed further below.

FURTHER ANALYSIS OF THE SPINDLE IN HOLE INSERTION PROCESS

Insertion of the spindle into the hole was an example of the pin in hole problem studied intensively at Draper[4] with -the parameters being as follows:

d = shaft diameter = 0.40 inches
D-d = clearance = 0.004 inches
l = insertion depth = 1.06 inches at full insertion

C = clearance ratio = 1 -(d/D) = 0.01

$2\theta_l$ = minimum wobble at full insertion = 2C (D/l) = 0.430 deg

θ_l = wobble from center line = 0.2 15 deg

To allow initial entry into the hole the limiting tilt angle is arc cos(1-C) = 8 degrees.

Note that while the Stanford Arm gripper was designed not to be compliant, compliance was assumed at the gripping point for purposes of this discussion. The spindle had a step and the hole had a chamfer, so the Draper parameters are:

L_g = distance from spindle end to grasping position = 2 inches

δ = chamfer = 0.025 inch

Step = 0.05 inch

Dealing with the step as though it were a chamfer, insertion all the way was possible without two point contact if the offset from the center, $\epsilon < 0.05 + 0.025 = 0.075$ in. in addition, the entrance tilt ($\theta < \theta_l - \epsilon/L_g$) will be less than 0.215 deg, depending on the offset ϵ.

With the hardware available and the friction characteristics of the joint motors, it was calculated that a minimum penetration of 0.8 inch using a nominal downward force of 10 oz was necessary to prevent jamming.

SENSING REQUIREMENTS

Position Sensing

Position sensing would be all that is necessary if the arm could be positioned with a tolerance of within 0.001 inch and tilt of 0.1 degree, and if parts could be positioned to these tolerances within the fixtures. However, given the Stanford Yellow arm with a repeatability of 0.04 inch and possibility of specifying distances to within 0.01 inch, it is essential that force and touch feedback be used.

Vision Sensing

Verification vision[6] would be useful in determining the initial process of inserting the spindle into the hole. Before insertion takes place, it is assumed that the spindle and the hole are "near" each other, and verification vision could tell how close they are and actually monitor the positions of the spindle and the hole as the spindle approaches the hole. For the task given, being able to sense a tolerance of 0.002 inch and an angle of 0.1 deg would enable decisions to be made as to which direction to move or tilt the spindle. Resolution at a finer level would enable "how much" to be computed as well. With verification vision, a

spiral search would not need to be done to locate the hole, as was necessary in the assembly,

Force and Touch Sensing

Touch sensing is necessary at the fingers, together with ability to measure hand openings as a means of telling whether the object has been grasped at the right place, if at all. Since the parts are fairly rigid, the touch resolution is not critical, as the gripping force is about 5 lb. Force sensing to a resolution of 0.5 oz would allow the arm to know if the part has slipped out of its grasp, by checking the weight at the end of the hand. Force sensors behind the hole (at the fixture) and behind the spindle would be helpful in telling the forces and moments at the hole and the spindle and together help to prevent jamming.

ARM DESIGN

Some of the problems in the movement of the arm stem from the fact that six degrees of freedom determine an essentially unique solution for motion from any frame to any other frame, so that even small motions may require that large inertias have to be moved. This fact suggests alternative arm designs with redundant degrees of freedom allowing small motions to be made with low inertia. Some of the possibilities are described below:

a) Extendible wrist which can elongate about 2-3 inches, so that hand can move along the direction of approach without moving joints 1, 2, or 3.

b) Extendible boom, so that joint 4 can move out of the boom a distance of 1-2 inches without moving joints 1, 2, or 3.

c) Independent finger movement, so that to grasp something without moving it, it would be sufficient to move only the fingers without having to use the CENTER command in which the whole arm has to move. When necessary, it would be possible to move both fingers together, e.g., when the position of the hand is known precisely, and it is desired to move the grasped object to the position defined by the location of the hand.

If redundant fine motions were provided in this fashion, then one might consider providing detents for joints 1 through 3 so that these joints can stop only in a finite number of known positions (say every 5 degrees or 1 degree apart) which can be determined to a high degree of precision. If there were no backlash and no static deflection of the arm components due to loading, the use of stepper motors in joints I, 2, and 3 would accomplish the same purpose.

The advantages of these changes would be faster nulling out of small errors and higher spatial resolution for given A/D resolution. Disadvantages would be that the programming language might become more highly hardware dependent, and there would be times when additional gross motions would be needed to bring the fine motors back nearer the centerpoints of their ranges.

It is not known whether or not the advantages outweigh the disadvantages. In any event, we do not plan to modify our arm hardware in the foreseeable future.

CONCLUSION

This paper has discussed some of the problems encountered assembling a pencil sharpener with the six degree of freedom Stanford Yellow Arm and comparison of the assembly motions required with those of the human operator using MTM and Draper assembly primitives. While arm resolution was lower than the clearances involved in the assembly, the use of suitably designed fixtures enabled parts to be located to a high degree of precision by just dropping the part and nudging it into place rather than actually trying to position it precisely. Analysis showed that the human operator is faster and requires fewer operations for the assembly process than the mechanical arm since the human operator makes more effective use of far more sensory feedback information, and the human arm is lighter and more flexible, and the hand is more dexterous and has more fingers than the mechanical counterpart. With these handicaps it was found that the manipulator took eight times longer to do the assembly job than the human operator did. It should be emphasized that this study has indicated the presence of inefficiencies in the present setup. The quantitative determination and elimination of these inefficiencies, the optimization of movements (in itself another important research area), and increased use of sensory feedback, would bring about a reduction in the assembly time.

ACKNOWLEDGEMENTS

The author would like to express his thanks to Richard Liu who suggested this particular assembly and for his valuable comments, to Tom Binford whose valuable suggestions and advice provided new insights, and to Dave Grossman for his editorial help in putting this paper together.