Stanford Artificial Intelligence Laboratory
Memo AIM-282

July 1976

Computer Science Department
Report No. STAN-CS-76-560

A Synthesis of Manipulator Control Programs From Task-Level Specifications

by

Russell Highsmith Taylor

Research sponsored by

Advanced Research Projects Agency
ARPA Order No. 2494
and
National Science Foundation

COMPUTER SCIENCE DEPARTMENT
Stanford University

Stanford Artificial Intelligence Laboratory
Memo AIM-282

July 1976

Computer Science Department
Report No. STAN-CS-76-560

A Synthesis of Manipulator Control Programs From Task-Level Specifications

by

Russell Highsmith Taylor

ABSTRACT

This research is directed towards automatic generation of manipulator control programs from task-level specifications. The central assumption is that much manipulator-level coding is a process of adapting known program constructs to particular tasks, in which coding decisions are made by well-defined computations based on *planning information*. For manipulator programming, the principal elements of planning information are: (1) descriptive information about the objects being manipulated; (2) situational information describing the execution-time environment; and (3) action information defining the task and the semantics of the execution-time environment.

A standard subtask in mechanical assembly, insertion of a pin into a hole, is used to focus the technical issues of automating manipulator coding decisions. This task is first analyzed from the point of view of a human programmer writing in the target language, AL, to identify the specific coding decisions required and the planning information required to make them. Then, techniques for representing this information in a computationally useful form are developed. Objects are described by attribute graphs, in which the nodes contain shape information, the links contain structural information, and properties of the links contain location information. Techniques are developed for representing object locations by parameterized mathematical expressions in which free scalar variables correspond to degrees of freedom and for deriving such descriptions from symbolic relations between object features. Constraints linking the remaining degrees of freedom are derived and used to predict maximum variations. Differential approximations are used to predict errors in location values. Finally, procedures are developed which use this planning information to generate AL code automatically.

The AL system itself performs a number of coding functions not normally found in algebraic compilers. These functions and the planning information required to support them are also discussed.

This dissertation was submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-76-C-0206 and the National Science Foundation under Contract NSF APR 74-01390-A02 . The views and conclusions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, ARPA, NSF, or the U. S. Government.

Reproduced in the U.S.A. Available from the National Technical Information Service, Springfield, Virginia 22151.

Acknowledgements

I am indebted to a great many people for advice, support, and encouragement in this work. This short note attempts to express my gratitude to them, collectively and individually.

I must express special appreciation to Professor Jerome Feldman, my dissertation advisor, for help in understanding how the different pieces of my work fit together and for many suggestions improving the coherence of this document. I am also grateful to Vint Cerf and Tom Binford, the other members of my advisory committee, for their constant guidance and availability, which went above and beyond the call of duty.

I am especially indebted to Dave Grossman for his constant willingness to listen to my harangues, for his patience in reading successive versions of difficult chapters, and for innumerable helpful suggestions. Likewise, Cordell Green, with whom I had many useful discussions when this work was at an early stage, and Lou Paul, who has been a constant source of information, advice, and encouragement, deserve special mention.

My thanks go also to the many colleagues, both at Stanford and elsewhere, who read parts of this document or discussed with me various aspects of my work: Dave Barstow, Bob Bolles, Randy Davis, Ray Finkel, Ron Goldman, Pitts Jarvis, Doug Lenat, Hans Moravec, Brian McCune, Shahid Mujtaba, John Reiser, Ari Requicha, Mike Roderick, Bruce Shimano, Victor Scheinman, Herb Voelker, Richard Waldinger, Peter Will, and many others.

The work reported in this dissertation was performed at the Stanford Artificial Intelligence Laboratory. I am grateful to Professor John McCarthy, the laboratory director, and to all the members of the technical staff responsible for the magnificant research facilities there. A special vote of thanks must go to Lester Earnest, the laboratory executive officer, for constant help in making the documentation system work.

Financial support was provided, at various times, by the National Science Foundation, by the Advanced Research Projects Agency, and by the Alcoa Foundation. My thanks go to all these agencies.

Finally, this work is dedicated to my parents, Charles and Nancy Taylor, for their constant love, confidence, and support through what has been a long and sometimes discouraging period.

"Another letter. To the E.B. Huyson Agency, New York, U.S.A. 'We beg to acknowledge receipt of order for five thousand Robots. As you are sending your own vessel, please dispatch as cargo equal quantities of soft and hard coal for R.U.R., the same to be credited as part payment of the amount due to us.'"

Karel Čapek
R.U.R.
1921

Chapter 1.

Introduction

Over the past ten years, a new and potentially revolutionary class of machines has emerged: arm-like "manipulators" operating under control of a computer. These devices have two important characteristics:

1. They are *general purpose* machines which can be redirected to new tasks with little or no hardware modification.

2. Their behavior can be modified during the execution of a task, based on sensory data obtained from the environment. Further, the response to sensory data can be both complex and readily reprogrammed.

The flexibility inherent in this combination offers a number of significant advantages over current fixed automation. Errors in the positions of parts can be tolerated, thus reducing the need for expensive precise fixturing. Inspection can be integrated into automatic manufacturing processes. Capital costs can be spread over many products. Finally, increased standardization of equipment within a plant eases maintenance problems and allows more flexible production scheduling.

These advantages will soon make it possible to replace human workers in a number of situations in which automation has so far been economically feasible only for very large production runs, if at all. This work focuses on mechanical assembly as an example application area which seems particularly likely to make use of programmable manipulators. However, it should be remembered that assembly is not the *only* use for these devices. Many of the techniques developed here are directly applicable to other manipulatory domains.

1.1 Requirements of Programmable Automation

Before programmable automation can be applied to a particular class of tasks, the requirements of *function* and *programmability* must be met. Function includes basic hardware, motion control, and sensory capabilities necessary to perform the tasks in the domain. Programmability requires the development of a suitable formalism for specifying how these capabilities are to be applied to the tasks.

Most research on manipulators has been devoted to the first category. Functional capability for mechanical assembly was demonstrated by Paul and Bolles in 1973 at Stanford, using a Scheinman electric arm [80,21,81], and in subsequent experiments at Stanford [84], General Motors [110], IBM [112,113], MIT [56], and several other laboratories. These experiments demonstrate the importance of sensory feedback and programmed control structures for reliable and efficient assembly without extensive special tooling.

Perhaps as a result of the emphasis on function, relatively little attention has been paid to programming formalisms until recently. Languages for manipulator control were not so much designed as grown. As functional capabilities were developed to the point where sophisticated applications became feasible, however, it was recognized that better languages wouldhave to be developed, if the full potential of manipulators were to be realized. In the past two years, several advanced manipulator languages – notably, MAPLE[31] and AL[37, 18] – have been developed. These programming systems resemble PL/I or ALGOL and offer gains in programmability roughly comparable to those offered by algebraic compilers over assembly language.

Although these languages provide a fairly direct way for describing actions by the manipulator system, the production of assembly programs is far from trivial. The process of generating such programs may be broken down (crudely) into two components:

1. A *task-level* specification, in which the job is described as a sequence of assembly oriented operations, such as fitting a part into place or driving a screw into a hole.

2. A *manipulator-level* specification, in which the functional capabilities of the manipulator are applied to perform the individual assembly steps.

Although the task-level description is frequently straightforward, the coding effort required to produce the corresponding manipulator-level specification is still substantial. One must decide what motions to make, what forces to exert, what sensory data to check, error conditions to monitor, etc. To make these decisions intelligently, one must consider the design of the object being assembled, where the parts are at each point in the program execution, how accurately their positions can be known, how precisely the manipulator can be controlled, and many other similar factors. Once made, the decisions must then be reflected in code written to satisfy the requirements of the available formalism.

1.2 The Goal

The research reported in this dissertation is directed towards computer automation of manipulator-level coding.

The central assumption is that the process of generating manipulator-level specifications for common assembly operations is generally one of deciding how to adapt known program constructs to the particular task at hand. Furthermore, these decisions can be characterized in terms of well-defined computations, based on *planning information.* If the computer is to make these decisions, it must have access to much the same information that would be used by a human programmer. This information includes:

1. *Descriptive information* about the objects being manipulated.

2. *Situational information* about the execution-time environment.

3. *Action information* defining the task and the semantics of the manipulator language.

Eventually, we would like a system capable of generating complete manipulator programs from object models and task-level descriptions of assembly sequences. The construction of such a system is an immense undertaking. The requirements include:

1. Development of an adequate manipulator-level target language.

2. Development of a formalism for task-level specification of a large class of assembly operations.

3. Development of a suitable representation for object models, together with means for automatic construction of the representation from whatever is available from the Computer Aided Design (CAD) system, and for computing relevant values required by the planning system from the representation. This problem is especially severe, since computer representation of shape information is still in early stages of development.

4. Development of means for representing situational information, including where things are expected to be and how accurately their positions will be known when the programs are executed.

5. Development of a large "knowledge base" of manipulation techniques required to implement the assembly operations, together with computational descriptions of the individual decisions that must be made to tailor code sequences to particular situations.

6. Development of means for making coherent assembly strategies, so that interactions between assembly steps do not lead to inefficient programs.

Since any early system will necessarily have many deficiencies and limitations, another requirement is:

7. Development of good ways to ask for and accept help from the user. Since such help may require the user to write manipulator-level code for at least some operations, this means that the system must "understand" manipulator programs well enough to update its situational models appropriately.

Clearly, some of these requirements are more difficult than others. When this work was begun, it was believed that the key problem was (6), the production of a coherent and efficient program in the face of interactions between component subtasks. Early research was therefore directed towards ways to resolve partial orderings of subtasks (e.g., "Put in two aligning pins; then put on the engine head; then insert four head bolts; etc"), select workpiece positions that were convenient for several subtasks, and so forth. It proved fairly easy to produce a system which could make these decisions, *based on toy data* in the form of declarative assertions like "In stable position *upright*, the box can be grasped at position *gpos1*". The success of this program was very encouraging until the time came to make the transition to real data. The difficulty of representing planning knowledge about the manipulator environment in a computationally useful form and then applying it to actual assembly problems turned out to be much greater than anticipated.

Consequently, the focus of this work has shifted from "global" optimization to more "local" coding decisions. These decisions must still be based on planning information about the manipulator's environment. Since we wish to focus on the *use* of such information, rather than on provision of elegant descriptive formalisms, some simplifications have been possible.

We have concentrated on the coding decisions required for a single task-oriented operation: insertion of a pin into a hole. This has allowed us to demonstrate the feasibility of program automation while, at the same time, keeping the required systems development effort within some reasonable limits. The modelling requirements for this task are sufficiently broad to include essentially all of the elements – where things are, what can go wrong, what methods are available to correct errors – found in other tasks.

1.3 Relation to Automatic Programming Research

Earlier, we described manipulator programming as a process of figuring out a sequence of operations which will get the task done and then expressing those operations in terms of available functional primitives. This combination of problem solving and coding activities is not unique to manipulator programming. A recurrent theme in the development of programming systems has been the provision of more convenient levels of abstraction, with the computer taking over many of the "low-level" coding responsibilities.

Thus, symbolic assemblers keep track of address assignments, fill in numerical values for symbols, and perform other similar tasks. Algebraic compilers, in turn, take over many of the coding responsibilities of an assembly language programmer, such as allocation of variables, register management, loop control, translation of expressions into machine instructions, and so on. Recently, there has been a great deal of interest in "very high level" languages, in which the user describes his task in terms of abstract "information structures" and problem-oriented operations, and relies on the computer to implement his specification

in terms of available data structures and computational primitives.[1] Within this framework, automatic generation of manipulator programs from task descriptions may be viewed as provision of a very high level manipulator language. Although a full discussion of automatic coding in other domains is beyond the scope of this document, several points are worth making.

In order to construct an automatic coding system for any domain, we must do several things. First, we must analyze the coding task being automated to identify the specific decisions that must be made. As we will see in Chapter 3, the decisions for manipulator coding include where to grasp objects, selection of motion destinations and intermediate points, choice of sensor tests, anticipation of likely error conditions, and determination of recovery strategies. Second, we must identify the planning information needed to make the decisions. As we stated in the previous section, this information includes object descriptions, situational information about expected locations and accuracies, and action information. Third, once the relevant decisions and planning information have been identified, we must find ways to represent the information that are "understandable" by the computer, in the sense that we can perform well-defined computations to make the necessary coding decisions and produce the appropriate output programs. This problem is especially severe for manipulator coding. Much of the research effort reported here has been directed towards development of computationally useful representations of physical situations. The use of these representations to make coding decisions for a typical task (insertion of a pin into a hole) is the subject of Chapter 8.

1.4 The AL Manipulator Programming System

Any discussion of automatic coding must necessarily include some attention to the process being automated – here, the production of manipulator-level programs. This research has been done within the context of AL [37,18,19,39], which was designed at Stanford as a successor to Paul's WAVE system [80,82], and whose implementation has proceeded concurrently with the work reported here. AL is the "highest level" manipulator control language yet implemented, and, consequently, seemed like a natural target language for an automatic coding effort.[2]

Although this document does not attempt a complete description of AL, some attention is given to several important aspects of the system. Of particular importance is the fact that AL, itself, performs a number of coding functions not normally required of an algebraic compiler, including trajectory planning, rewriting motion statements, and resolving situation-

---

[1] For example, the work of Low [66], Rovner[67], Green [43], Barstow [7], Schwartz [98], Early, and many others [5]. It is interesting that the term "automatic programming" was first applied to compilers. Today, we tend to react with amusement to this usage. However, it is perhaps unfair to do so. The basic process – automation of coding decisions based on information maintained by the computer – is, after all, the same as attempted by most "advanced" systems. In twenty years, we will perhaps also consider automatic generation of ALGOL programs to be a "naive" form of automatic programming.

[2] The fact that I was actively engaged in the design and implementation of the language was, of course, also a relevant factor.

dependent conditional compilation requests. These functions require that the compiler keep track of much more situational information than might otherwise be the case. In particular, it must keep track of expected **frame** values, affixments between location variables, and user assertions about runtime states. Since this thesis is largely concerned with the use of planning information to make automatic coding decisions, we will discuss the techniques employed to keep track of this information, how the information is used, and the problems encountered. The basic paradigm used to keep track of information in AL is also used by the task-to-manipulator-level translation primitives, which are the heart of this work.

1.5 Overview of the Document

The scope of the material covered in this dissertation is quite broad. Considering the nature of the problem attacked, this is inevitable. I recognize that not everything will be of equal interest to all readers. This section is intended to provide an overview of how everything fits together, and to indicate which parts are most important to understanding the basic theme of this research: how coding decisions for manipulator programming can be automated.

Chapter 1. Introduction

Presumably, you've already read it. Congratulations on your good taste!

Chapter 2. A Discussion of Manipulator Programming.

This chapter is intended to establish a basis for discussion about manipulator programming. It is divided into three major components: (a) A discussion of the mechanical assembly domain. The key points are that programs must be flexible enough to handle variations in the execution environment and that sensory feedback is important in obtaining that flexibility. (b) Next, we discuss the advantages, disadvantages, and intellectual requirements of different programming paradigms for manipulators. The paradigms discussed are *tape recorder mode*, *augmented tape recorder mode*, and *"formal" languages*. The principal point is that many assembly tasks are complex enough to require the flexibility of formal language specifications, but that such systems require more from the user than do more "iconic" methods. (c) Finally, we provide an overview of existing formal language systems. If you are willing to believe these points, or don't care about them, then you can skip over this chapter without losing too much.

Chapter 3. AL, the Anatomy of a Manipulator Language.

This chapter provides an overview of the target system, AL, and describes the process which we are seeking to automate: generation of manipulator-level specifications of assembly operations. This is done by writing successively more complete versions of a program for accomplishing our prototype assembly task – inserting a pin into a hole. This approach allows us to show off several important features of AL, to identify the coding decisions that must be made, and to explain the factors that must be considered in making them. Finally, I cannot resist spending a few more pages discussing the two salient characteristics – as I see them – of AL: the use of coordinate frames to specify motion and the use of frame affixment to simplify book-keeping. *You should read this chapter*, even if you skip over

everything else in the thesis. Section 3.4 is especially important, since it analyzes the coding decisions whose automation is described in Chapter 8.

Chapter 4. Planning Models

A vital point about programming is that it is a form of planning – i.e., making *prior* decisions about actions to be performed at a later time. To make these decisions rationally, we must necessarily base them on our expectations about the circumstances in which they will be executed. If we wish to get the computer to take over some of the coding burden, we must find a way for it to represent and maintain the necessary planning information. This chapter discusses the relation between automatic coding and the maintenance of planning information, first within the "familiar" context of ALGOL programming, and then within the context of AL programming, using the pin-in-hole task of Section 3.4 as an example. The discussion explores how each category of information (object models, situational information, and action information) enters into the programming process. This is a short chapter and probably should be read by everyone, although those whose sole interest is in manipulation may want to skip over Section 4.2 rather quickly.

Chapter 5. 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. This chapter describes how the compiler maintains and uses this information and discusses many of the difficulties encountered. The approach is to associate a data base of assertional "forms" with each control point in the program graph, using simple simulation rules to propagate facts. Persons who are not particularly interested in the technical issues involved can get by without reading this chapter in detail. The important points are that the same mechanism – a multiple world assertional data base – is used by the automatic coding procedures discussed in Chapter 8 and that "understanding" manipulator level statements is necessary, though often very difficult, if the system is ever to incorporate user's "advice" with its own coding decisions.

Chapter 6. Object Models

As we mentioned earlier, techniques for computer representation of shape are not yet well developed. Unfortunately, manipulator programming necessarily involves some decisions based on such information. Since the number of problems that can be solved in one dissertation is, alas, finite, we have had to adopt a fairly *ad hoc* solution to this one. Objects are modelled by "attribute graphs", in which shape information is represented in the nodes, structural information by links, and location information by properties of the links. Within this framework, many important facts – such as the available "free area" around an object feature or the expected penetration of a pin into a hole – are represented explicitly, even though, in principle, they are computable from more primitive shape representations. This chapter describes some of the details of this representation scheme. You don't need to read it closely, unless you are particularly interested. I've just told you most of what you need to know. The most interesting point is that coding decisions can generally be based on "local" properties of objects.

Chapter 7. Representation of Location and Accuracy Information.

In manipulator programming, the most important forms of situational information are the *expected location* of the objects being manipulated and *how accurately* their locations will be known at execution time. A substantial part of the research effort reported here has been the development of techniques for representation of this information in forms that permit reasonable coding decisions to be made. The principal results are methods 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. Depending on the interpretation placed on the free variables, these techniques may be used to predict either locations or accuracies. In addition to describing these techniques, this chapter presents a number of applications, such as vision planning and parts tolerancing, which are not strictly part of our automatic coding effort. The discussion is rather mathematical, although nothing beyond freshman calculus and linear algebra is required. Try not to get bogged down in the details. By reading Section 7.1 through Section 7.5, Section 7.8, and the examples of Appendix E, and looking at those special applications that interest you, you can get enough of a "feel" to follow the uses made by our coding procedures of the methods discussed in this chapter.

Chapter 8. Automatic Coding of Program Elements

This chapter applies the modelling mechanisms developed above to automate the coding process described in Chapter 3. If you have read everything that comes before, you shouldn't have any trouble understanding how this feat is performed. If you have skipped material, you may have to take some things on faith, but you should be able to understand enough to see that decisions are being based on very definite computations on the planning model.

Chapter 9. Conclusions

The principal conclusion[3] is that I claim to have demonstrated a sufficient basis for the automation of at least some manipulator coding decisions. Additional points are that all the machinery built, or some better replacement, is also *necessary* to the process and that a great deal remains to be done before an integrated automatic programming system can be "put up" for users. This chapter also contains the traditional description of the direction further work should take.

---

[3] Of course

1 – A robot may not injure a human being, or, through inaction, allow a human being to come to harm.

2 – A robot must obey the orders given it by human beings except where such orders would conflict with the First Law.

3 – A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

Isaac Asimov
Handbook of Robotics, 56th Edition
2058 A.D.

Chapter 2.

A Discussion of Manipulator Programming

2.1 Introduction

The underlying goal of any manipulator programming system is to provide the user with an effective means of specifying what actions must be taken by the manipulator and its related hardware to achieve a desired result. The requirements placed on the manipulator programs and, consequently, on the programming system depend to a great extent on the characteristics of the task domain for which the programs are being produced. These characteristics include:

1. The number of times the program must be executed.
2. How much the environment may vary each time the program is executed.
3. The complexity of motions required to perform the task.
4. The complexity of the control sequences required.

For instance, one of the earliest uses of manipulator-like devices was for handling radioactive chemicals from a safe distance. More recent applications of the same idea – giving the user a way to manipulate objects when it would be dangerous, difficult, or unpleasant for him to do so with his own hands – include under-ocean or outer-space repairs, disarming bombs left by terrorists, or arming bombs for use against an enemy. In these teleoperator systems, the human takes a more or less active role in direct control of the device. Each situation is treated as a new problem, and the "programs" are executed as they are written and then thrown away. The motions required may be quite complicated, often requiring careful monitoring of and control over the forces involved, rates of motion, and

the like. The principal requirement is to provide the user with an efficient way of specifying *particular* motions and reacting to the sensory data from *current* environment. Typically, this is done by providing the user with a joystick, control yoke, or other contrivance for indicating the motions to be made and for accepting feedback from the hardware. Since the motions are specified in real time and on a once-only basis, questions of generality of specification, provision for alternative situations, etc., never arise.

In applications where the manipulator is expected to run on its own, without direct control from a human operator, it becomes necessary to construct a program. Perhaps a few words about just what is meant by a "manipulator program" are in order. Any definition is necessarily somewhat clouded by the close interrelation between the hardware and support software of manipulator systems. Some systems close the control loop for the manipulator through a computer. Should these servo routines be considered part of the manipulator program? Suppose the runtime system contains a facility for centering the manipulator's hand on an object. It may be possible to describe the necessary motions in terms of sensor tests and incremental arm motions. Should this code be included? What if the runtime system contains a genie capable of figuring out how to put any object together by looking at it?

If our hypothetical genie were available – i.e., if it were possible to defer *all* decisions about what to do until the time came to act – then "programming" would scarcely be necessary. All we would have to do would be to roll the robot up to a work station, turn on power, and say "go to it, fellah". The robot would then function very much like a teleoperator system, with a computer program playing the role of the human operator. Unfortunately, such a device seems to be beyond the current state of the art.[1] The key point about writing a program is that, by specifying how executable primitives in the system are to be applied to perform a particular task, we can reduce the remaining decisions to those that can be handled efficiently at run time.

Thus, the distinction between a programmable manipulator and a teleoperator system is that programming necessarily involves *prior* decisions about how to apply general capabilities to particular problems. In our discussion, we will mainly be concerned with the level at which pre-execution specification of the program begins to become unique to the problem – typically, at the level of motion or sensor control statements, conditional branches, or the like.

2.2 Characteristics of Automatic Assembly Domain

We have chosen to study manipulator programming for small to intermediate-scale mechanical assembly applications.

Why Study Assembly Programming?

There are several reasons why this domain is a "natural" one for the study of manipulator programming:

---

[1] Even if such a device were available, it probably would be too expensive to be practical for many applications.

1. It represents a "live" application for manipulators.

2. A universal runtime system that can figure out on its own what to do is beyond the current state of the art. Consequently, programming of specific tasks is, in fact, necessary.

3. Task environments are constrained enough so that efficient, non-trivial programs can indeed be written.

Of course, many characterists of assembly tasks may also be found in other manipulation domains, and much of what we have to say will be more generally applicable.

2.2.1 Example Task

The photographs in Figure 2.1 illustrate a simple assembly task well within the capability of current manipulator systems.[2] We are presented with parts of a metal box, and wish to put it together. We do this as follows:

1. The box is picked up off a "conveyor" and placed in a vise.

2. Aligning pins are inserted into two holes in the top surface.

3. A metal cover plate is fitted on over the pins.

4. The plate is secured with two screws.

5. The aligning pins are removed and replaced by screws.

6. The box is removed from the vise and returned to the conveyor.

2.2.2 Task Repetition

We are concerned principally with production runs ranging roughly from 1000 to 100,000 units. These numbers are somewhat arbitrary. The low end for any particular application is determined by the point at which the programming and setup costs for the manipulator outweigh the benefits of automation. The high end is given by the point at which it becomes economical to construct a special purpose device. To push the applicability region *downwards* clearly requires that programs be made easier to specify, with detailed specification being automated as much as possible. However, increased flexibility is just as important since elaborate special tooling is generally not economical where only a few items are being produced. The principal requirement for pushing the applicability reigon *upwards* is to increase the *efficiency* of programs. Typically, this involves ways to speed up motions, avoid wasteful motions, and to increase accuracies, so as to require less fumbling around.

---

[2] A similar assembly – putting the head and head gasket onto a Model-T Ford waterpump – was performed by Paul and Bolles in late 1973 [21].

Figure 2.1. Box Assembly Sequence

2.2.3 Variability

We are aiming at about the level of variability one might expect on an assembly line. These variations can occur at several times:

1. Between task repetitions within a single production run.
2. Between production runs of the same task.
3. Between tasks.

Within Production Runs

Within a production run, there is generally little need to vary the order in which the constituent parts are put together or the basic technique used in each individual assembly operation.[3] Most assembly operations require that the parts be in some relative orientation[4] and that the position errors between parts be kept within certain bounds for the operation to be performed successfully. Typically, this is accomplished by a combination of fixtures to hold the parts in place and sensory information to observe locations directly or to infer errors from indirect data like forces between parts. In our box assembly, aligning pins are used to make the holes in the cover plate line up with those in the box body, and the body itself is held in a vise while the assembly is done. Since the vise is not a precise fixture, the exact location of the box may vary slightly with each assembly. The use of sensory feedback to accommodate such variations is described in Chapter 3.

Location of Tools and Parts

Tools and fixtures usually can be kept at more or less the same place on each iteration of a production run. Parts either may come packed together in a "kit" – which may be more or less neatly arranged – or be introduced separately to more or less well defined reigons of the work station. Any of a number of techniques may be used, with widely varying degrees of control over part orientation. Some examples:

1. Parts unoriented and unseparated – tote bins
2. Parts unoriented but separated – egg cartons

---

[4] The principal exception to this is the case where several different models are being assembled concurrently. Depending on how different the assembly sequences are, this may be handled by using whole separate programs for each model, or by using conditional statements within a single program.(Of course, the former may be considered merely a degenerate case of the latter). It is interesting to note that a computer controlled device may exhibit more flexibility in this respect than a human, in the sense that it is less apt to become confused by a large number of different model requirements presented in random sequence. The principal requirements are that there be some means of specifying which model is to be built each time and that the work station setup be compatible with all versions of the task.

[4] In this discussion, "orientation", "position", and the like will be used rather loosely; usually, both translation and rotation are meant.

3. Parts sit in one of several "stable positions", but otherwise more or less unrestricted – table top

4. Part orientation fixed approximately – loose fixturing, "careful" placement on a surface, simple vise.

5. Part orientation fixed precisely – "better" fixtures.

The choice for a particular problem depends on the flexibility of the manipulator and on the time available. For instance, parts are frequently delivered to human assemblers in tote bins – each bin containing a great heap of some kind of part all jumbled together. The assembler picks parts individually out of the bin, orients them manually, and uses them for the assembly. Unfortunately, present-day manipulators lack the sensory capability and dexterity to accomplish this efficiently.

Fixtures

At the other extreme, we find cases where parts are kept carefully oriented by fixtures from the very beginning of the assembly process. Generally, this rigid control is most useful in high volume applications where automatic machines are being used; efficiency is gained by avoiding the need for any sensory adaptation, which is often non-existent on special purpose machines and (in any case) may take time, and by avoiding the need for any waste motion in reorienting parts or in getting around awkward workpiece positions. Where it is impractical to store or transport parts so as to maintain the required orientation, special purpose devices – vibratory feeders, shake boxes, and other ingenious gadgets – are frequently used, and the orienting operation may be done in several stages. For instance, a vibratory feeder might be used to align parts sufficiently well so that they may be grasped or placed unambiguously into a fixture which, in turn, will reduce alignment errors to the point where assembly can proceed.

Where a human is being used as the assembly device, precise fixturing for absolute positions is not generally useful, since people rely much more on sensory feedback than on absolute accuracy in fitting things together. However, where great accuracy is required or where sensory feedback doesn't work so well, fixtures may be used as alignment aids. The advantage of such aids, again, is that a simpler problem – loading a fixture – is substituted for the harder problem of accomplishing the assembly directly.

Between Production Runs

Frequently, small batches of a particular product may be produced at intervals. When the station is being set up for the new run, it is frequently desirable to alter its arrangement somewhat from that used before. For instance, some tools required by the task being set up may already be mounted at the work station, only not in the same place. Depending on how important placement is to efficiency, it may be desirable to leave them alone. Even if the location of everything is more or less the same, there usually will be small differences. If a human is doing the assembly, these differences will make no difference at all. Indeed, they may not even be noticable without careful measurements. On the other hand, many automatic assembly devices rely on knowing absolute locations to within very narrow tolerences. If the device is programmable, it may be possible to save the program in a form that allows it to be reused after a suitable recalibration phase. The ease to which such

recalibration can be done can have a substantial influence on whether it is economical to automate odd-lot production items.

Between Tasks

Clearly, much more variation occurs between tasks. One does not expect to assemble a stapler with exactly the same sequence of operations as used to assemble a pencil sharpener. On the other hand, standardization through the use of common fixtures, tools, fasteners, etc., and through similarities in the operations themselves is often possible. For example, many tasks require bolts to be driven into threaded holes; it is possible to devise general purpose bolt drivers, dispensers, and – to some extent – methods for accomplishing this. Aside from the obvious savings in setup and tooling cost that such standardization makes possible, there is also a substantial gain from increased ease in retraining or instructing workers in performance of a new job, since one doesn't have to teach basic skills each time. For programmable manipulators, we would like to have to a library of standard procedures for common operations, such as picking tools up from tool racks, dispensing and using standard fasteners, and actual assembly operations such as inserting pins into holes, fitting things together, etc.

In addition to standardization at the individual operation level, a group of tasks may exhibit a great deal of overall similarity. An extreme case of this is where the same object is to be assembled using different configurations of the work station, as discussed earlier. Model changes may require that only a few parts of the assembly procedure be modified. Workers familiar with the old task should find it easier to pick up the new task than comparably able workers new to the entire assembly. Where programmable devices are being used, such retraining corresponds to modifying an earlier program rather than generating a new one from scratch whenever the task is modified.

2.2.4 Complexity of Tasks and Programs

Generally speaking, assembly tasks may be described as linear or partially ordered sequences of operations. For instance, the alignment pins in Section 2.2.1 may be inserted in either order, though they must both be in place before the cover plate is put on.

The operations themselves may be rather more complicated, and (indeed) may be beyond the capability of current machines. For instance, reorienting a part taken from a bin is frequently performed by shifting it in the hand while it is being transported to the workpiece. Similarly, it is sometimes necessary to reach inside a workpiece, compress something, thread a limp object through a hole, or perform some other feat in order to get a part into place.

On the other hand, many assembly operations may be accomplished by extremely simple motions. The part is merely transported to a desired location and released; it literally falls into place. Typical intermediate cases might involve the activation of a tool, such as a power screwdriver; subsidiary locking motions, as with a bayonet thread; or some amount of careful fitting-together. Force sensing may be used to a greater or lesser extent for guiding the terminal phases of motion and to verify that the operation has been completed sucessfully.

The current state of programmable motion control for manipulators is still rather limited. The systems are still too clumsy to to perform many operations, although sometimes a special-purpose fixture or attachnent can be devised for a particular task. Much of this clumsiness comes from limitations in existing hardware and control software, especially for general-purpose end effectors.[5] However, limitations in programming formalisms and doctrine are at least as important. Even if the hardware is theoretically capable of performing some action, we still must know exactly what we want done and must have a way of expressing our desires in a way understandable to the machine. At present, our understanding of manipulation and of how to describe manipulatory actions is still quite limited. Extensions to basic hardware capabilities are thus, to some extent, dependent on progress in ways to enhance the programmability of the devices.[6]

Specific capabilities vary widely from system to system. We will mainly be concerned with manipulator programs in which motions are described by sequences of discrete manipulator positions, without explicit specification of what happens in between, and where limited forms of force and touch sensing are available. [7] These capabilities are sufficient for a number of common assembly operations, including:

1. Inserting pins into holes. This includes the important subcase of driving in bolts and screws.

2. Mating one part surface to another.

3. Fitting things over studs. This includes putting washers and nuts on over the end of a bolt.

4. Placing parts into fixtures.

Of course, it is possible to find instances of any of these operations that are too hard for our existing techniques. The interesting point is that a substantial subset of cases can be covered. Furthermore, the additional flexibility offered by programmability and even limited sensory feedback makes this subset somewhat larger than for comparable fixed automation machines.

---

[5] The current "canonical" device is a two or three fingered grasper.

[6] For instance, a five fingered hand is certainly constructable. Nevertheless, work proceeds with two fingered grippers, in part because no one has any good way to specify motions using anything more complicated. Where sensory data is concerned, the problem is even more pronounced. For instance, digitized TV data has been obtainable for some time now, but only recently has there been much progress in applying vision to industrial applications.

[7] This will exclude many continuous servoing operations such as fitting a disc over a spindle. However, many of the considerations relevant to the use of such techniques – such as where objects are and how much accuracy is required – are the same as for the constructs we will be using. As more advanced manipulation primitives are developed, therefore, we hope that the lessons learned here will still be applicable.

2.3 Programming Paradigms

This section will discuss several different approaches to manipulator programming, each of which offers some advantages and disadvantages. Crudely, these approaches may be divided into "textual" and "non-textual", and are superficially quite different. We will see, however, that they are not at all incompatible. Indeed, they may be combined so as to increase their power substantially.

2.3.1 Tape Recorder Mode

This method, also called "teach mode", is the only programming method that is in common industrial use today. Essentially, it is an attempt to adapt teleoperator control experience directly to the preparation of manipulator programs. A joystick, button box, or similar device is used to describe all motions of the manipulator. To make a new program, the user sets up the work station exactly as it will be at the start of task execution. He then uses the manipulator as a teleoperator to perform a sample execution of the task. Successive positions in this sample execution are then identified (usually, by pushing a button) and remembered in a control memory.[8] The position sequence is then "played back" to cause the manipulator to mimic the example, and, so, to accomplish the task.

Usually, though not always, there are simple "interrupt" provisions to allow an external signal, such as might be generated by a limit switch or completion of an NC machine program, to start or stop the motion sequence. These facilities can be used to provide a degree of synchronization between machines as well as some slight degree of sensory adaptability (e.g., one sometimes can close the manipulator's "hand" until a switch is triggered). Generally, however, programs written in this manner rely on the absolute repeatability of the manipulator, together with accurate fixtures, to accomplish the task.

Intellectual Requirements

This method requires very little "programming" ability, since it does not require the user to associate textual or otherwise abstract symbols with future manipulator motions. To "write" a program, one merely needs to be able to drive the manipulator around and to identify important points.[9] Programming is made very concrete, and the effects of each action are immediately reflected in the configuration of the work station.

A significant advantage of the formalism is that maximum use is made of the human

---

[8] There are several forms in which the necessary information can be saved. The most "direct" is simply to save the joint angles of the manipulator. More sophistocated systems generally save the position of the manipulator's end effector. This offers several advantages; one is that calibration changes do not cause the program to be invalidated.

[9] Actually, of course, the joystick or control buttons *are* formal specifications, and the remembered position sequence constitutes a formal program. However, the immediate feedback from the manipulator makes things somehow less intimidating to someone who is unused to computers. Less *explaining* is required to teach someone how to program the machine, although proficiency in controlling the device or in producing efficient programs may require considerable experience.

programmer's geometric intuition and problem solving abilities. No "computerable" model of the work station, parts, fixtures, etc. is needed, since the information is supplied by the real world. Similarly, the human's expertise on how the manipulator hardware can be used to accomplish his task is applied in a very direct manner.

Task Requirements

Teach mode makes the fundamental assumption that the task being programmed can be described adaquately specified by a sequence of *absolute* positions. This assumption has several consequences:

1. Object positions and other relevant aspects of the work station must remain unchanged from iteration to iteration of the task.

2. The manipulator and jigging must be sufficiently accurate so that the task can be performed without active accomodation by the control program.[10]

3. The task definition cannot include any specifications inherently requiring force monitoring, such as "compress to three pounds", or "tighten to three foot-pounds".[11]

Where these assumptions are not met, it is necessary to use force, tactile feedback, vision, or other techniques to produce the necessary corrections. Unfortunately, it is very difficult to describe sensory feedback methods within a pure teach mode paradigm.

Flexibility

One unfortunate consequence of specifying motions by means of absolute locations is that the resulting programs are fairly inflexible. For instance, suppose that the manipulator has been programmed to put the head and head bolts onto an engine block. If the fixture holding the block is moved, the programming must be done all over again. One solution is to define the remembered positions relative to calibration points. However, to do this purely within a teach-mode paradigm can be quite cumbersome, since several calibration frames will generally be needed. The user must continually specify which frame is to be used, as well as what points are to be remembered, and much of the conceptual simplicity of the method is lost.

---

[10] *Passive* accomdation refers to techniques used to build compliance into the hardware, such as mounting a fixture on springs or vibrating a part to break friction. *Active* accomodation refers to modifying what the manipulator is to do in order to overcome variations.

[11] One could conceivably produce a system that remembered forces as well as positions, and which sought to exert the same force each time the program was played back. However, in the absence of of task-related knowledge (e.g., "this motion is part of a sanding operation."), it is very difficult for the system to decide whether force or position should take precedence. Attempts to resolve such questions lead one naturally into the "augemented" or "formal" systems discussed later.

Contingencies

Another limitation of the method is that there is no good way to describe contingencies. This applies both to error recovery and to "normal" conditional actions. For instance, consider the use of a manipulator to sort boxes. The manipulator is to move to a certain position and close its hand until it grasps a box. It is then to place the box in an inspection device, which will signal either "good" or "bad". If the box is good, the manipulator is to regrasp and move the box to a conveyor belt. If bad, the box is to be dropped onto a trash heap. If the box slips out of the hand during any motion, the manipulator is to stop and a bell is to be rung. Here, all the positions may be defined in terms of absolute locations, and no active accomodation is required. The manipulator may be led through either motion sequence with no difficulty. The problem is that there is no way to refer to the sequences, so it is impossible to say "if so-and-so, then do thus-and-such." *We have thus made a trade-off: the user doesn't have to "understand" formal program structures; on the other hand, he cannot talk about them.*

Documentation and Editing

Another drawback arising from the purely iconic nature of a tape recorder mode program is that there is no self-evident way to describe what it does. Either *another* specification, written in English or some other language, must be used or else the program must be run and observed. Coupled with the inability to refer to particular motions or sequences explicitly, this means that teach-mode programs are very difficult to edit or modify. Some facility may be provided to allow a user to "retract" a position that was just remembered. One can generally "play back" a program until the place where the modification is to occur is reached, "teach" some more motions, and (if one is very lucky) continue with the rest of the program. However, the lack of any explicit means of representing the program to the user makes it easy to become hopelessly lost.

2.3.2 Augmented Tape Recorder Mode

This paradigm, which is being pursued vigorously at SRI [89,90], preserves many of the characteristics of simple tape recorder mode: the user sets up the work station just as it will be at the start of each program run and then guides the manipulator through a trial run, which is then played back each time the task is to be performed. However, the number of "built-in" functions is substantially increased, typically including

1. Differential motion commands, such as "move up one inch".

2. Simple sensory commands like "push down one pound" or "move left until you hit something".

3. A variety of special purpose "fine motion" primitives for common tasks, such as insertion of screws, fitting of nuts, operation of tools, etc. Typically, these primitives are capable of using sensory inputs to accommodate small differences in object position or to make simple local decisions. E.g., "detect color at a given spot relative to the hand; if red, then operate the welding gun; otherwise, do nothing."

Character of Programming

Selection from this repertoire can be made by any of a number of means: pushing an appropriate button, speaking into a microphone, typing at a terminal, etc. A typical programming session might look something like:

User moves manipulator over screwdriver handle

"grasp"

Manipulator closes until touch sensors on fingers indicate object is firmly held.

"up 3 inches"

Manipulator moves up 3 inches. Note that since the screwdriver holder is in a fixed place, the user could just as well have used a straightforward absolute positioning command to get the desired effect.

User positions manipulator so that screwdriver is just over a bolt head.

"Drive_down 3 foot-pounds"

Manipulator engages driver to head of bolt & tightens it to the indicated torque

"up 1"

et cetera

Software "tools"

In many ways, the "fine motion" primitives resemble an array of special-purpose tools that happen to have been implemented in software. Each "tool" has a particular function; the user has no responsibility for the inner workings of the primitive, nor can he modify it beyond (perhaps) specifying some parameters. Calling one of these primitives corresponds to sending an "operate" signal to a hardware device. Since the user has no control over the inner structure of the primitives, there is little *functional* difference, so far as he is concerned, between active and passive accomodation. For instance, a primitive that modified the manipulator's motions a small amount in responses to forces on the hand would look pretty much like one that relied on springs in the wrist to accomplish the same end. The difference lies in the speed with which the software "wrist" can be changed.

Advantages

Augmented tape recorder mode retains many of the advantages of pure teach mode. Most of the necessary modelling is done by the user interacting with physical parts, and programming is still very concrete. The principal new advantages offered by the technique are:

1. More tasks can be performed.

2. Absolute accuracies need to be less, thus reducing fixturing costs.

3. Programs may be shortened, since some of the special functions may include several motions.

Intellectual Requirements

However, the added facilities give programming a somewhat more "symbolic" character and require a better programming understanding on the part of the user. These requirements include:

1. The use of parameters. Since the parameters are still concrete, this doesn't differ much from setting up measurements on a machine tool. The hard step – which will come up later – is introducing the idea of variables.

2. Modelling of future situations that are not absolutely identical to the sample. The addition of "move until touch" and "move relative" commands allows the manipulator to handle somewhat greater variations than before. To use these commands effectively, the user must understand how positions can be defined relative to other positions and must consider whether the expected runtime variations are sufficiently limited so that the manipulator will always hit what he expects it to. Similarly, the special function operations will generally have an associated "capture radius" defining how close the actual position of the object being worked on must be to the nominal position built into the program. The user must understand these limits and consider whether they are sure to be met.

Evaluation

To the extent that programs written in this paradigm retain the use of absolute locations to specify motion sequences, they will share the inflexibility of simple teach mode. If an object is moved more than a small amount, then the program must be rewritten. Similarly, since program structures are not explicitly available, editing and contingency handling are still difficult.

In many ways, augmented tape recorder mode represents an awkward "stopping point" for manipulator programming. For instance, one would like to use the "move until touch" facility to define a location which can be remembered and used as a base for future motions. Without some way of naming things, there is no good way to do this. Similarly, the runtime system necessary to support many of the augmenting features – especially, the special functions – is a good deal more powerful than that required for a simple tape recorder mode, and the fuctions themselves are generally coded up using a formal programming system. These facilities would be very useful to the user, provided they were "packaged" in an appropriate manner. For instance, we would like to avoid forcing a user to write his programs directly in the machine language of the computer used to control the manipulator. Indeed, we would like to insulate the systems programmer who is producing the service functions as much as possible from this sort of bit fiddling. It is much more reasonable to design an augmented teach mode system *around* a formal programming system; with care, this can be done so as to allow simple guiding programs to be prepared in the way we have described, while at the same time avoiding the inherent restrictions of tape recorder mode.

2.3.3 Formal Languages

As we mentioned earlier, *any* of the programming methods described so far is "formal" in some sense. The semantics are well-defined, even if limited. A program is constructed and stored in the machine. Once the internal representation is built, *how* it was built becomes rather moot. We would like to distinguish "formal language systems" from the methods previously discussed by stressing the key notion that such systems have ways to talk *about* program constructs and that these ways are made accessible to the user. The principal consequences are:

1. Control structures are made explicit. The facilities provided vary from rudimentory to quite sophistocated. Generally, however, they are at least sufficiently powerful to allow for handling of contingencies, simple iterations, and other commonly needed features.

2. Variables are named entities which can be set and retrieved under control of the program. [12]

Text

Generally, we will be dealing with *textual* languages, although text isn't strictly necessary. One of the principal advantages of text is that it can be read by humans. It is much easier to edit text programs, to save them away in understandable forms, and do various other generally useful things. On the other hand, it is frequently much easier to describe positions by pointing than by measurement schemes (such as cartesian coordinates). This suggests that iconic programming techniques can be retained for such purposes. The program structures produced can be "decompiled" into a more readable form, edited, and merged into the overall program. Thus, joysticks, etc., can be viewed as useful "shorthand" tools for defining certain tedious (to program) steps.

The Role of Variables

The introduction of variables makes a tremendous difference in the flexibility and efficiency of programs, since the program can acquire and *remember* information learned after it is written and use that information to modify its behavior. For instance, suppose we are removing parts arranged neatly in a shipping container. The container may be displaced slightly, but the manipulator can locate it precisely by centering the hand on a locating tab affixed rigidly to it. An augmented tape recorder program for accomplishing the job might look something like this:

position hand around fiducial mark

"center"

"open"

"go left 2"

---

[12] Of course, the "name" doesn't have to be an identifier like BOX or BILLY; it may be a number, a reference generated by pushing a button, or even something implicitly defined, such as the top of a stack.

"go down 1"

"grasp"

"go up 5"

put object away

position hand around fiducial mark

"center"

"open"

"go left 4"

"go down 1"

"grasp"

put object away

et cetera

By using a variable to remember where the fiducial mark was, and then defining motion targets relative to that variable, we can eliminate much wasted motion.

position hand around fiducial mark

"center"

"remember this spot as *fiducial* "

"go to place 2 left of *fiducial*"

"go down 1"

"grasp"

put object away

"go to place 4 left of *fiducial*"

"go down 1"

"grasp"

put this object away

The advantage is magnified in cases where the variable values can be obtained without

groping around (e.g., from a vision system) or during an initialization phase. In these cases, even the first centering operation can be eliminated.

The syntax in this example has deliberately been left fuzzy. There are several different ways the variable *fiducial* could be used to produce the desired effect, depending on the system being used.

Control structures

The addition of even primitive control facilities allows much more general programs to be written, since the restriction that the same sequence of operations must be performed in all cases is now relaxed. For instance, suppose our part unpacker program doesn't know in advance how many parts will be in a particular crate, although it is known that the parts will be lined up nicely. When a crate is empty, it is to be discarded and the machine is to wait for a new one to be introduced to its work station. A program for accomplishing this task might look something like this:

1. Wait for a crate full of parts to be introduced to the work station.
2. center on fiducial tab, as before.
3. *spot* ← place 2 left of *fiducial*.
4. Open the hand, and move manipulator to *spot*.
5. Move manipulator down 1 and grasp. If the hand fails to grasp anything, open the hand and go on to step 7. Otherwise, put the part away and go on to step 6.
6. *spot* ← place 2 left of *spot*. Go back to step 4 to get the next part.
7. Pick up the crate and dispose of it. Go back to step 1.

Once again, we have left the syntax very informal. Actual languages range from primitive formalisms resembling computer assembly languages to sophisticated languages resembling ALGOL or PL/I. There are several generally applicable points illustrated here, however.

Contingencies

First, the action performed by the manipulator depends on whether or not the hand succeeds in grasping a part. In one case, it puts the part away; in the other, it discards the empty crate. Another common use of a conditional branching facility is checking for error conditions. For instance, the program could be modified so that if step 2 fails to locate the fiducial tab, it will stop, print a message to the operator, ring a bell, and generally make a nuisance of itself until the problem is remedied. Alternatively, there may be some error recovery actions that can be attempted before the program gives up.

Iterations

The second point is that the program contains a *loop*. It can perform the same series of actions repeatedly until some condition, such as the box being empty, is fulfilled. This is fundamentally impossible in a tape recorder mode; the program will execute exactly the steps you guide it through and will perform them exactly the same number of times you do. But the correct number of iterations *cannot* be predicted ahead of time. This indeterminacy comes up again and again in "fine" operations. Some task, such as insertion of a screw into a hole, is attempted, and a test is made for success. If the test succeeds, the program proceeds to the next operation. If it falls, a correction is computed and the task is retried. The process is repeated until either the operation is successfully completed or a decision is made to give up (for instance, because the proposed correction exceeds some threshold or because the program includes a limit on the number of iterations allowed for the loop).

Finally, even if the number of parts per crate were constant, the loop makes the program much shorter. In a tape recorder mode, each part has its own pickup point, and, hence, requires separate motions. This requirement is not altered by the use of a variable to correct for misalignments. The way we win is to alter *spot* to point to the *next* part. In other words, we are using *spot* as a *parameter* to the loop body.

Disadvantages

The principal disadvantages with formal language systems are that the features they offer require somewhat more runtime support than simple tape recorder mode systems, although not significantly more than augmented tape recorder mode, and that they require a greater intellectual effort on the part of the user. If the facilities offered by tape recorder mode are fully adequate, there may be little reason to incur the overhead of any more powerful system. If this is not the case, however, we are led to the question of why programming seems so hard for so many people.

User Engineering is Important, but Neglected

One reason, unfortunately, is that the user engineering of most manipulator programming systems is rather poor. In part, this stems from the early concentration on functional capability at the expense of programming formalisms. The problem, of course, is that it isn't enough for a machine to be able to do something. One must also be able to say what one wants done.

A related problem is that the initial system design is often far too conservative, perhaps because of a desire to get something up quickly or "on the cheap", and allows insufficient room for subsequent growth. The drawbacks here generally apply about equally to language and execution-time system. This means that adding new features or fixing shortcomings in the original system can only be done with chewing gum and bailing wire. Eventually, the whole thing collapses of its own weight. Although an accomplished programmer can generally get around such defects, albeit (assuming he has been exposed to better things) with much grumbling, a novice – who also must learn some basic concepts about programming – is almost hopelessly handicapped.

Intellectual Requirements

Even with "the best of all possible" formal structures and ideal system support, writing programs still requires more from the user than does simple teach mode. In the first place, the user must understand the semantics of the formalism. Generally, formal languages offer a wider variety of manipulator functions than do more restrictive formalisms. These functions must be understood before they can be used effectively. Even if only basic motions are desired, it is still necessary to connect symbolic descriptions with future actions of the manipulator. Also, conditional execution of statements and iteration are concepts that require some understanding before they can be used. This understanding, in turn, must be based largely on the concept of a *program state.* The user must keep track of what is expected to be true in each set of circumstances that his program will face, and must understand, more or less, how the program will behave in those circumstances. Since greater variations in the task environment may be handled than for teach mode programs, the utility of carrying along a nominal case as a crutch may be somewhat weakened, although it is still quite useful. Modeling must be more abstract, and the user must be able to distinguish "generally" true aspects of his sample case from the merely "accidental", lest the program rely on false assumptions.[13]

A program's model of the world rests on the assumptions built into it – constants, program tests, etc. – and on the use of variables. To use a variable intelligently, user must employ several kinds of knowledge:

1. The semantic meaning of the variable: what quantity does a variable represent, how is the mapping of meaning to quantity established, how can the value be used, and so on.

2. The syntactic requirements for variables: declarations, scope rules, and the like.

3. What values variables may have at different points in the program.

4. How accurately variable values may be expected to reflect the quantities they model.

These points will be discussed at greater length in subsequent chapters.

2.4 Overview of Formal Language systems

This section provides an abbreviated overview of the various sorts of manipulator languages. It is intended to provide some indication of the scope of formal languages, and not as a complete survey of the field. If your favorite language is missing, please don't be offended. Instead, as an exercise, try to decide to which class it belongs.

---

[13] An analogy to this may be found in geometry, where a carelessly drawn figure may lead to conclusions that are only true in particular cases, if ever.

2.4.1 "Pseudo-machine" Languages

Almost all manipulator programming languages in existence today are "low level" formalisms that somewhat resemble machine-level assembly languages, in which program control is described by skips and jumps. The following short program (written in Paul's WAVE system [80,82]) illustrates the characteristics of this class of languages. The problem is to pick up a small metal box and move it to a new position. If the grasping operation fails, the manipulator is to move out of the way and ask the operator for assistance.

```
        open 2                      ;Open hand 2 inches
        move gbox1 z 4 nil 0        ;Move to a position 4 inches over box
l0:     go gbox1                    ;Move to grasping position
        center 0.3                  ;Grasp it
        skipe 2                     ;Test to see if there
        jump l1                     ;Yes, we have won
        open 2                      ;NO, move out of the way
        move gbox1 z 8 nil 0
        wait the box is missing     ;Complain
        jump l0                     ;Try again
l1:     move gbox2
```

Other languages which preserve this same general character include the target language for the AL compiler [18], ML and EMILY, which were developed at IBM [113].

2.4.2 "High Level" Languages

Recently, several "high level" languages, notably AL and MAPLE [31], have been developed. These languages offer ALGOL or PL/I control structures, variables, arithmetic capabilities, and many other nice features. AL will be discussed in Chapter 3. MAPLE is very similar in many respects, although the actual capabilities of the two systems differ in many others.[14] The MAPLE code for picking up our box would look something like this:[15]

```
flag = 0;
open to 2;
move to gbox1 translated by (0,0,4);
until flag = 1 do;
    move to gbox1;
    call centerandgrasp(.3);
```

---

[15] For instance, MAPLE lacks the AL affixment mechanism. On the other hand, it has closed subroutines, which AL lacks. (AL uses macro calls instead.)

[15] I am indebted to Dave Grossman for this coding example.

```
        if gap < 2 then do;
                move by (0,0,8);
                write("the box is missing");
                stop;
                end;
        else
                flag=1;
        end;
move to gbox2;
```

2.4.3 "Very High Level" Languages

The idea of a "very high level" language is that the user describes what he wants done in terms of task-oriented primitives, and the computer writes the corresponding manipulator program. So far, no such systems have been implemented. Indeed, the whole point of this dissertation is the establishment of a basis for automatic coding in this domain. Several attempts at providing a suitable descriptive formalism for the input language have been made, notably, in the AL report [37] and in AUTOPASS[64], a language designed at IBM.[16] In such languages, our box-moving example would be trivial:

```
move box to new_place;
```

---

[16] Work on implementing AUTOPASS is under way, but it is too early to say much about just what the final system will look like.

Chapter 3.

AL, The Anatomy of a Manipulator Language

3.1 Introductory Remarks

The work reported in this dissertation has been done primarily within the context of AL, which was developed at the Stanford Artificial Intelligence Laboratory as a successor to Paul's WAVE system. The design and implementation of this system has involved a substantial effort on the part of several people, including myself; a full exploration of all the issues raised by the system is clearly beyond the scope of our present discussion.

Although this chapter discusses AL, it is not intended to be a "complete" introduction, in the sense of covering the entire system, or even all constructs that are used in subsequent chapters.[1] Instead, we provide a brief overview which emphasizes the salient characteristics of the language and programming system. These characteristics are then illustrated by an extended programming example, which allows us to examine what goes on in writing an AL program, what the necessary tradeoffs are, etc. Finally, some problems and some advantages of AL's underlying formalism will be discussed in more detail.

3.2 Overview of the 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 **wait** primitives.[2]

Data Types

In Chapter 2, we stated that one of the key attributes of a formal language for manipulator control was the use of named variables to describe positions, forces, and other relevant data. In principle, this could be done using only the data types of ALGOL. However, such an approach can be rather tiresome, tends to make programs hard to read, and increases the chance that a program will contain bugs. AL seeks to avoid these difficulties by providing data types and "arithmetic" operations for the physical and geometric entities required for describing manipulation. The most important of these special types are **frames**, which are

---

[1] A "nutshell" description may be found in Appendix A. Further discussion of the AL system design may be found in the AL report [37], in the Stanford NSF reports [17,18,19], and in Finkel's dissertation [39]

[2] Various flavors of these primitives come under many names. See, for instance, [27] for further discussion.

used to represent coordinate systems, and transes, which tell how frames are related. [3] AL programs use frames to describe hand positions, object locations, and other similar information; 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 in the program. 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 by the frame *box*. This approach can get rather tedious where the same frames are being referenced repeatedly, and tends to hide the "intent" of a program behind a smokescreen of frame transformations, with a corresponding increase in the chance that a bug will creep in undetected. The alternative method is to use a separate frame variable for each frame of interest. This makes motion statements (described below) and other constructs easier to read and write, but means that all associated variables must be updated whenever something is changed. Again, this is very tedious, tends to obfuscate programs, and opens the chance for a terrible bug if something is forgotten. 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. [4] 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,

---

[3] Aside from this difference in usage, frames and transes are isomorphic. There is a distinguished frame, called station, which corresponds to the coordinate system of the work station. The value of any frame is given by the transformation needed to carry it from station to its current location. Thus, both frame and trans values consist of a position vector giving the location of the origin, together with a rotation specifying the orientation of the axes. A much fuller description of these, and other, AL data types, and the operations used to manipulate them may be found in Appendix A.

[4] Actually, this is an over-simplification. *box_grasp* would merely be marked as invalid and a new value recomputed when required. The problems associated with affixment will be discussed in more detail later in this chapter and in Appendix C.

affix *cover* **to** *box*;

and

affix *box* **to blue**;[5]

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

Motion Statements

Motion statements are the *raison d'etre* of AL. Ultimately, all manipulator motions must be described in terms of joint motions, since joints are what the runtime system can control. However, this representation is a very awkward one for user level specifications and introduces a needless degree of hardware dependency if it is used. 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.[6] Thus, AL programs describe motions by sequences 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 a particular place, this 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 system has been told (by the **affix** statement) that changes in the value of the blue hand are to cause corresponding changes in the value of *box*. This information is then used to produce hand positions that will cause *box* to pass through *midair_point* and wind up at *new_box_place*. This sequence of destination points is translated by AL into the corresponding joint behavior by a combination of compile-time planning and execution-time revision, which will be discussed further later.

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 modifying clauses. For example,

---

[5] The manipulator hardware at SAIL 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)

[6] Indeed, the ideal AL manipulator would be a disembodied hand, which wandered about without any visible means of support.

```
move carburetor to inspection_station
    via unloading_point
        where force(xhat)=0,[7]
              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. The statement specifies a three segment motion. During the first segment, when the carburetor is removed from the fixture, the arm is made compliant to forces in X and Y, and the motion is constrained 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 force in the Z direction exceeds a half pound. Finally, as soon as an electric eye detects something, an external control signal is generated.

3.3 Structure of the AL System

The overall structure of the AL system is illustrated in Figure 3.1 There are two major components: a compiler, which resides on a large, time-shared computer (PDP-10); and a runtime system, which resides on a moderately powerful minicomputer (PDP-11/45). [8] Superficially, this structure looks a great deal like that for any algebraic language: the compiler gobbles down source text, grovels over it, and spits out code which is executed by the runtime system. However, the requirement that the program produce more than purely computational results makes a substantial difference in the *internal* structure of both the compiler and runtime system.

3.3.1 Runtime System

The runtime system of AL consists of three principal components:

1. A kernel, which is a real-time operating system with some special features added to facilitate programming of the rest of the system.

2. An interpreter, which is responsible for all computation and control flow done by the AL program. The "pseudo-instructions" executed by this interpreter may be thought of as the instruction set for a machine explicitly

---

[7] xhat, yhat, and zhat are unit vectors in the x, y, and z directions.

[8] The large machine can examine and modify the memory of the small machine, and either one can interrupt the other.

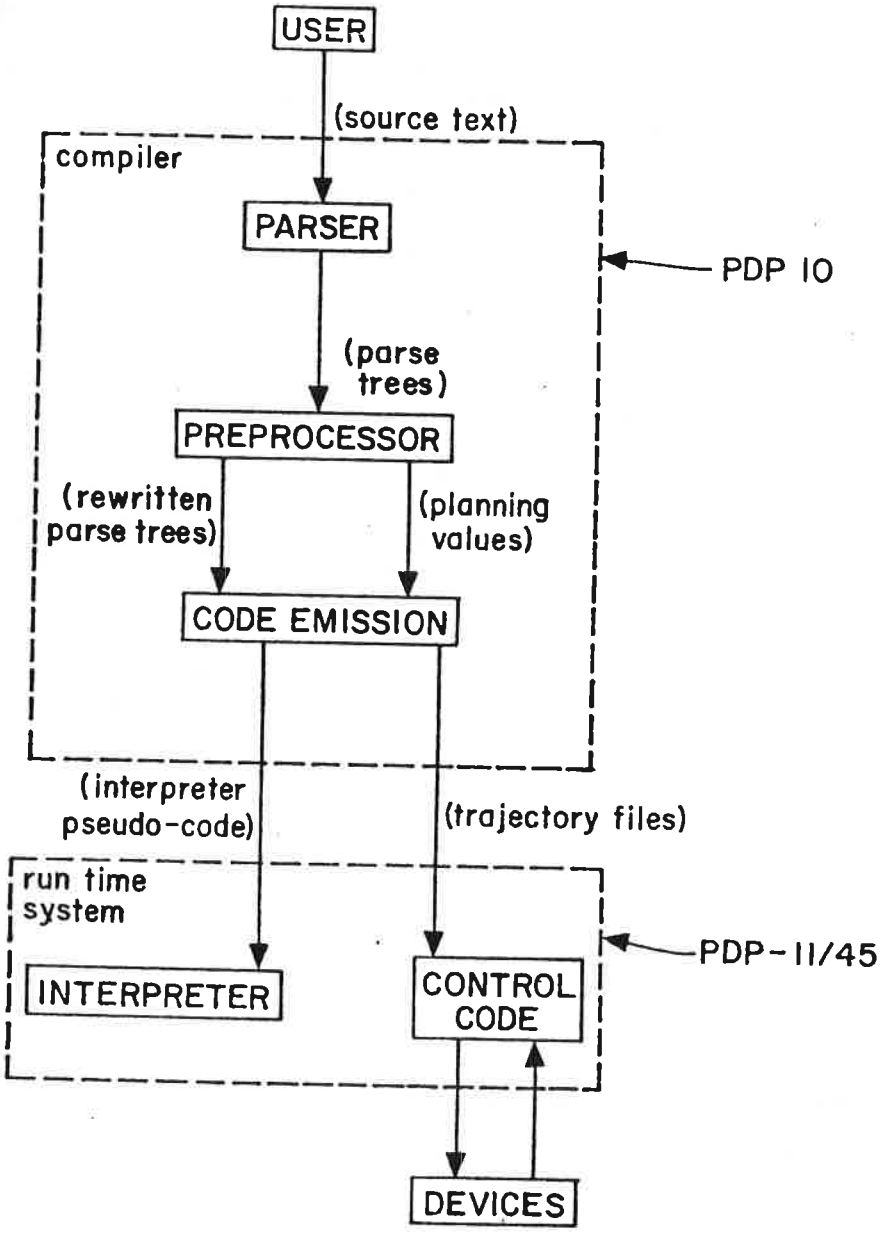


Figure 3.1. The AL System

designed for AL-like languages.[9]

3. **Control code**, which actually runs the manipulator and other devices.

The design and implementation of this runtime system raised many interesting "systems" issues outside the scope of this document. However, several points are important for understanding our subsequent discussion.

Trajectories

One of the central design decisions in AL was the use of polynomial joint "trajectories" as the basis for motion control. As we have seen, motions are specified in AL programs by giving a list of positions through which an object is to pass. The runtime system, however, must servo the individual joints. The required 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.[10]

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 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 does create 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. This will simplify the compilation of present-day AL programs, and greatly increase the flexibility of the system. However, it will not eliminate the need for a planning model, which is used for other purposes, as well.[11]

3.3.2 Compiler

The compiler is responsible for translation of AL source text into a form that can be executed by the runtime system. This process involves many activities, such as parsing, assignment of storage, and code emission, common to any compiler for an algebraic language with real-time extensions. In addition, however, the compiler must perform several functions that are closely tied to the special requirements of the manipulator system. These will be discussed below. Translation is accomplished in three phases:

---

[9] At present, this code is interpreted by means of jump table. However, it would be fairly well adapted to "direct" implementation in microcode, should anyone ever want to build a true "AL" machine.

[10] This method was developed by R. Paul, and is reported in [80]. More recent refinements may be found in [18] and [39]. In his recent work, Paul has abandoned polynomials in favor of an interpolation scheme [90].

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

1. *Parsing.* The source text is read in and translated into an internal structure (program graph) that is understandable by the rest of the compiler. This is a very straightforward process. The principal "unusual" activity at this stage is to check for consistency of physical units used in arithmetic expressions.

2. *Preprocessing.* The AL program is *simulated* to build up a detailed planning model of the expected state at each point in the program graph. This is an activity which is not commonly found in algebraic compilers,[12] but which plays an important role in AL. This planning model, which is discussed at length in Chapter 5, contains information about the affixment structure and the expected value of location variables. This information is needed by the code emission phase for calculation of joint trajectories. In addition, a number of important details are incorporated into the program graph, which may be rewritten somewhat. For instance, affixment statements are translated into sequences of lower-level "graph assignment" statements. Similarly, the modeled affixment structure is used to turn statements like "MOVE a TO b" into more explicit statements involving a specific arm, like "MOVE BARM TO bat".

3. *Code emission.* Joint trajectories are computed for all motion statements, based on the expected location values contained in the planning model. These trajectories are written into output files. Similarly, interpreter code is produced from the refined program graph and written into an output file.

3.4 Sample AL Program

This section illustrates the use of frames, affixment, and force feedback to accomplish a simple assembly operation – the insertion of an aligning pin into a hole – which is a typical subtask for many assembly programs. In addition to showing off some of the features of the language, the discussion should provide some insight into the process of writing an AL program.

This discussion will be rather extended. One of the points it makes is that manipulator programming is a non-trivial intellectual activity, even for simple tasks. We will proceed roughly as follows: First, an outline for the program will be developed. A simple, "first cut", program will be developed to implement the task outline. We will then examine the flexibility and "toughness" of this program. Method for error detection and recovery will be discussed, and a new, more elaborate, program will be produced. Finally, we will include further discussion of the role of error handling in AL programs.

---

[12] The closest analogy would be the flow analysis used by optimizing compilers

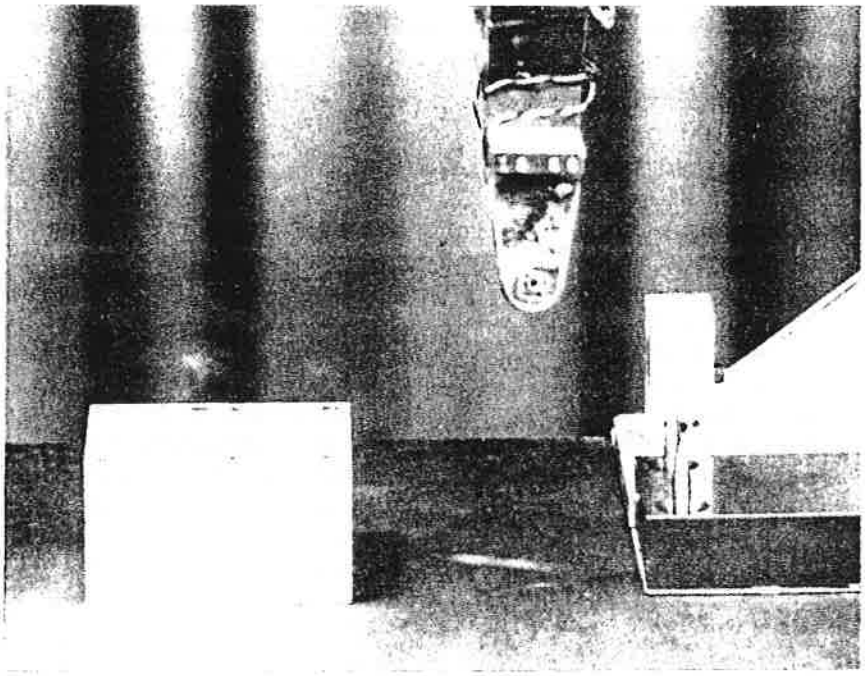


Figure 3.2. Before

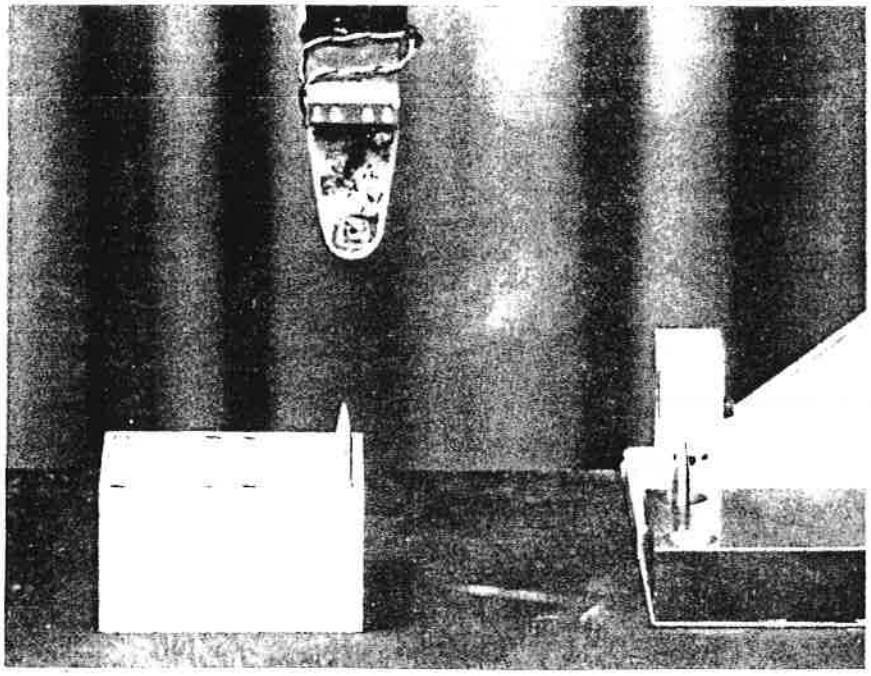


Figure 3.3. After

3.4.1 The Task

The bare-bones description of the task is quite simple. Initially, the pin sits in a tool rack, and a metal box with holes in it sits on the table in some known position, as shown in Figure 3.2 Our mission is to get the aligning pin into one of the holes, as shown in Figure 3.3. After a suitable period of omphaloskepsis, we conclude that 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, finally, 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
```

3.4.2 Declarations and Affixments

The declarations for this task 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. At first glance, this may seem like a purely descriptive task. Upon further reflection, we see that certain strategy decisions are actually embodied in this part of the program. 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? Some of the candidates are illustrated in Figure 3.4 If there is any chance that the pin can bind in the rack or box hole, then 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. This is reflected in the declaration and affixment portion of the program:[13]

```
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));
```

---

[13] In AL, rot(*axis*,*angle*) specifies a rotation of *angle* about the vector, *axis*. xhat, yhat and zhat are unit vectors in the x, y, and z directions, respectively. See Appendix A for more details.

```
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;
```

The declarations embody a number of other strategy assumptions;[14] these will be discussed as we come to them. 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.

Before we proceed, it is perhaps worth noting that 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 also 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.

3.4.3 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. }
```

There are several difficulties with this code. The most serious 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.[15] What points are required will depend 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 will assume that the manipulator is "clear" of any extraneous obstructions, and consider only the possibility that

---

[14] Note, for instance, that the intermediate position *pin_withdraw* is twisted and above the initial pin position, *pin_holder*.

[15] This raises an important issue concerning the adequacy of frames for specifying motion, which we will discuss further later on.

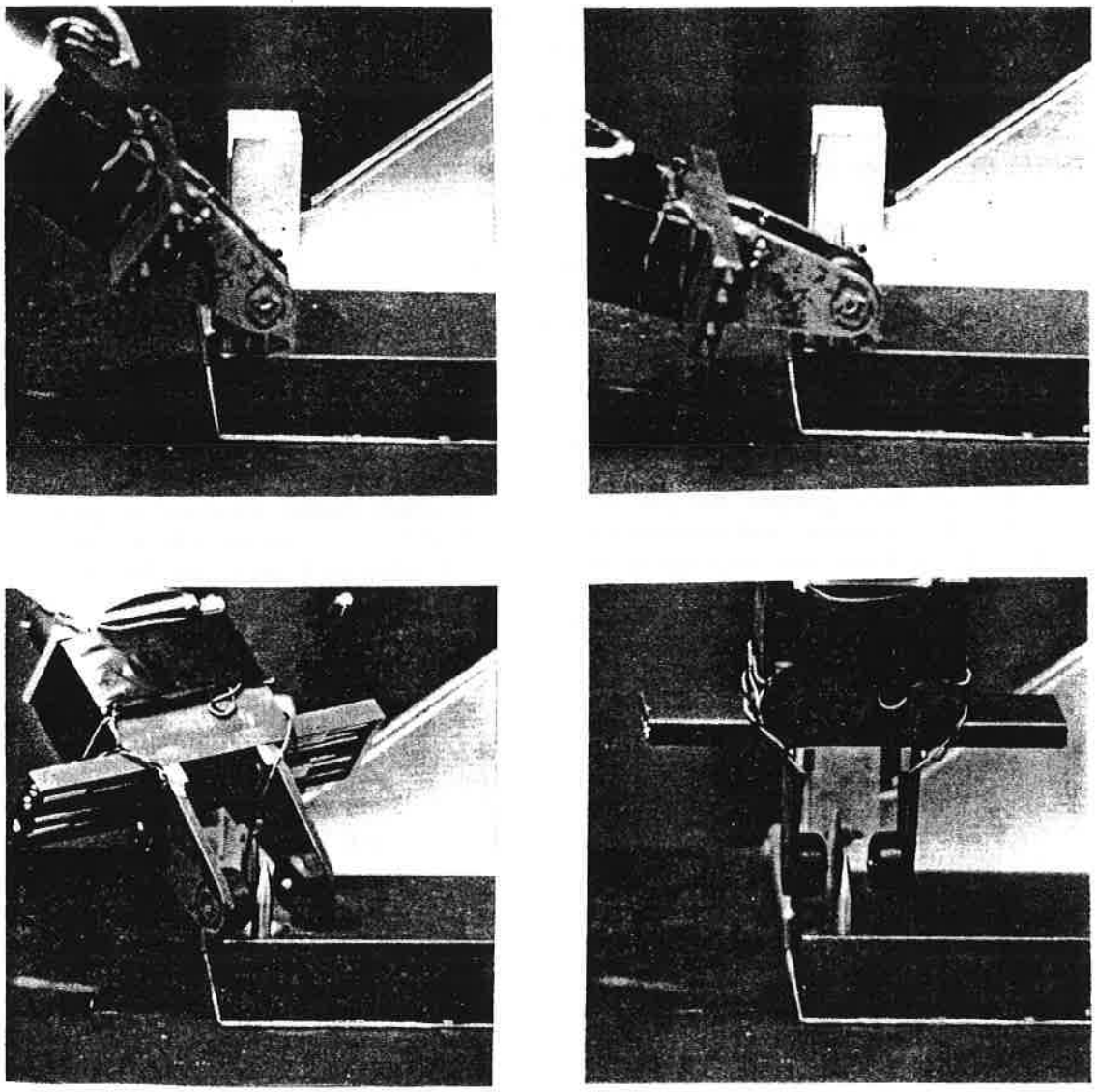


Figure 3.4. Possible Grasping Positions

the fingers might collide with the pin while moving to *pin_grasp*. Catastrophe 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.[16] 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.

3.4.4 Initial Program

Once we have grasped the pin, we can use a single motion statement to 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. Writing the statements for these operations and combining them with 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;
```

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

---

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

pin_holder, rather than to *pin*, this updating would not occur, and the motion specified would be rather wild. 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;
```

3.4.5 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.[17] 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.[18]

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.[19]

---

[17] Murphy [71] has investigated the reliability of systems in some detail. Experience has verified that his results apply with special force to manipulator programming.

[18] 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 [48,19]), and in Appendix F. 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. These considerations are discussed in more detail later.

[19] 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. Although it seems reasonable to demand at least a modicum of honesty from the variables, it is worthwhile to consider what can be done to reduce the accuracy required, especially since extreme precision may be rather expensive and difficult to attain.

3.4.6 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 somewhat 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. However, it is about the right magnitude for the present arm hardware at Stanford.

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. For a given range of location errors, this assumption may be valid. 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, thus 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(xhat,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←false;
            end;
```

Two objections (not necessarily fatal) 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 is used, and the hole is slightly higher than the runtime value says it is, then the forces produced as the arm tries to servo to the "nominal" position can become quite large. If the hole is slightly below nominal, then no real damage will be done for this particular task, since the pin will most likely drop into place when released. However, other tasks are not so forgiving. If we are inserting a screw, for instance, the initial insertion must bring the screw threads into contact with the threads in the hole.[20] In such cases, it is much better to get a positive contact than to rely on brute force accuracy.

---

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

"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. Here, 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, as in Figure 3.5. 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 us where the box surface is. This might be done as follows:

```
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 ← zhat · 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 updated appropriately. Thus, the assignment statement will translate the box location to

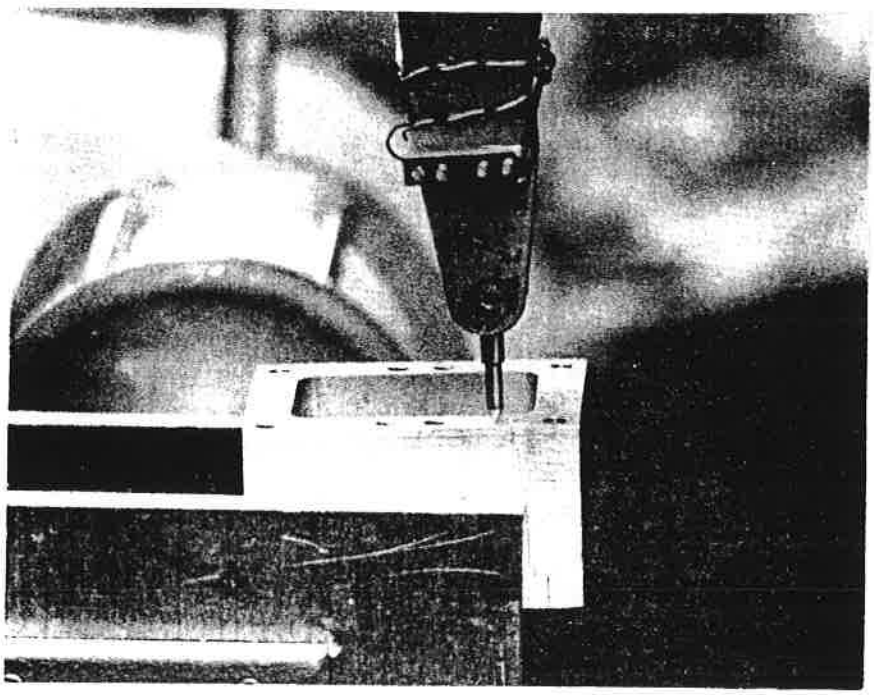


Figure 3.5. Box Held by Vise

account for whatever distance the pin actually travelled. This technique has some advantages:

1. It is easy to write, since you don't have to invent variables or figure out complicated arithmetic expressions.

2. It is easy to read, since the code is terser. Also, the assignment statement more nearly reflects the "intent" of the motion statement, which was to get the pin to *spot_on_surface*.

On the other hand, there are a number of difficulties associated with using affixment to infer object locations from local measurements. These will be discussed in Section 3.6.

3.4.7 Error Recovery

So far, we've been discussing ways for the program to discover that it has lost.[21] Once a failure has been detected, we must do something about it. The simplest course is to give up.

---

[21] An optimist would say "discover that it has won", but this is unjustified. The laws of the universe state that there will be at least one failure mode for which a program check has been left out. This is God's way of teaching humility to engineers (who rightly regard the Babel affair as a management, and not a technical, failure) and computer programmers (who seem to like a profusion of languages). Even if it were, in fact, possible to anticipate and test for *all* failures, it would not necessarily be economical to do so.

```
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 perhaps 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 (at least) 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:

```
if not in_hole_flag then
    begin
    vector dp;
    scalar n;
    dp ← vector(0.1*inches,0,0);
    for n ← 1 step 1 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;
```

Obviously, 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. The possibilities are endless.

3.4.8 Refined Program

Combining a search loop with the other refinements we have discussed, and adding a "free" 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_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 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 > -0.2*inches ) then
        begin
        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 ← zhat · 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 bpark via pin_grasp_approach;

end;
```

3.4.9 Further Discussion

The 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 relatively unimportant, except, possibly, for Procrustean considerations.[22] A more important cost is the extra time required in execution. Unless

---

[22] 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

something really hairy is contemplated, the extra computer time spent in "head scratching" isn't likely to be an issue.[23] The time spent in manipulator motion is another matter. For instance, each iteration through the loop may take nearly as long as 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 here 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 our 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, one hopes that the actual pin-fingers relation can be computed during the time that the pin is being transported to the hole.[24] This correction can then be used to get the insertion right the first time.

Multiple Error Sources

In our discussion of this task, we have mainly proceeded as if the only source of error was an inaccuracy in the location of the box. Actually, of course, we must consider errors from many sources; for instance:

1. There may be manufacturing errors in the objects being manipulated. For example, the hole may be drilled slightly off center, the height of the box may vary, or the pin may be slightly shorter than expected.

2. The hand will never be quite where the system thinks it is, due to the limited accuracy with which joint angles can be read. For the Scheinman arms at Stanford, the net error is usually on the order of around 0.05 to 0.1 inches in position and 0.25 to 0.5 degree in orientation, although these numbers depend somewhat on manipulator position.

3. The pin may be misaligned in the hand, as was mentioned earlier. This error, in turn, may depend on other errors introduced earlier in the program. For instance, when the pin is grasped, the hand's position won't be known precisely; this will be reflected in the pin-to-hand affixment.

---

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.

[23] Several systems to do "problem solving" at runtime to figure out how to correct errors as they arise. See for instance, [42]. Sproull [101] has investigated the question of when runtime planning is cost-effective.

[24] Bolles [23] 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.

Alternatively, the fingers might be slightly misshapen or the pin might slip while being extracted from the rack hole.

Critical Tolerances

In trying to anticipate problems and decide what checks are appropriate, it is necessary to consider how all these errors interact with the critical tolerances of the task. In this case, the important requirements are on how accurately the relation between the pin tip and the hole can be determined. The *horizontal* error (in the coordinate system of the hole) must be small enough so that the pin tip will make it into the hole, and the *orientation* error must be small enough so that the pin doesn't get stuck before making it in. The allowable thresholds here depend primarily on the relative geometry of the pin and hole (taking taper, chamfer, etc. into account) and, to a lesser extent, on the accomodation available during the insertion operation. For the pin and hole illustrated in our pictures, the relevant numbers are about 0.1 inches and about 5 degrees, respectively. On the other hand, small *vertical* errors will not necessarily prevent the pin from going into the hole; here, the critical tolerance is determined by the requirements of the verification method chosen. If distance travelled is used, as here, then the maximum difference between the actual and calculated pin-hole distance must be enough smaller than the difference between what would be observed for a hit or a miss so that an unambiguous test can be made. Since the pin goes in about a half inch, this means that if the combined error in *pin* and *hole* along zhat of *hole* is sure to be less than, say, 0.25 inches, we should be safe.

Influence of Different Errors

It is instructive to consider how various error sources influence our ability to meet these criteria. Since the box used in this example was turned out on a milling machine with a basic accuracy of about 0.001 inches, it is plausible to ignore any manufacturing errors. Unfortunately, arm errors are not so negligible. If we have positioning errors of 0.05 inch during both the pin grasp and insertion operations, the pin can miss the hole, even if everything else is perfectly accurate. Of course, the errors may may not always be this big or may well cancel, so that the first attempt will win, at least some of the time. On the other hand, it appears that our pessimism in including a search loop was justified. Also, it may be worth while considering ways to reduce these errors. One alternative, already mentioned, is vision, which may, or may not, be available. Another would be to use grooved fingers to "center" the pin, thus removing an important uncertainty in the pin-hand relation.[25]

If there are location errors in the box, they also must be taken into account. If the initial errors are large enough, then the hole occasionally may be displaced beyond the radius of the search loop. This can be fixed either by increasing the scope of the search, which seems unpalatable in terms of execution time, or by doing something to locate the box better. For instance:

---

[25] Errors along the pin axis would be unaffected by this fix. However, these are ignorable for this task. If they were not, a "tapping" scheme, such as discussed earlier, could be used. Here, it is interesting to note that a strategy of hitting the box surface with the pin tip will reduce the tip-hole uncertainty without necessarily affecting the pin-hand uncertainty, whereas tapping a known surface (e.g., the table top) will fix the pin-hand relation, but may require further checks to fix the plane of the hole.

```
{ Illustrated in Figure 3.6 }
frame box_grasp_1,box_grasp_2;

affix box_grasp_1 to box rigidly
      at trans(rot(xhat,90*deg),vector(0,0,1*inches));
affix box_grasp_2 to box rigidly
      at trans(rot(zhat,90*deg)*rot(xhat,90,deg),vector(0,0,1*inches));

open blue to 4.5*inches;
move blue to box_grasp_1;
center blue;
box_grasp_1 ← blue;

open blue to 3.5*inches;
move blue to box_grasp_2;
center blue;
box_grasp_2 ← blue;
```

This code should pin down the box position to within the accuracy of the hand, assuming that the initial errors are small enough to guarantee that the hand doesn't crash into the box. Consequently, displacement errors of about 0.5 inches can be reduced to errors on the order of 0.05 inches at the cost of two moves by the manipulator. Since search time goes up with the *area* of the error footprint, this can represent a significant improvment in efficiency.

3.5 Frames, Good and Bad

As we have seen, the dominant paradigm in AL is the motion of an object's coordinate frame through a sequence of destination values. One consequence of this is that the runtime "world model" of AL programs consists principally of frames and affixments, with little other geometric information except for what may be implicit in the program itself.[26] This section discusses some of the consequences, both good and bad, of the use of frames as a command and descriptive paradigm. To a great extent, of course, this use depends on affixment, which is discussed in the next section. On the other hand, a discussion of affixment depends on frames, so we might as well begin here.

The principal advantages derived from using frames to describe motions have already been mentioned: increased clarity and ease of programming and greater hardware independence of the resulting programs. These advantages are not unrelated. The tasks we wish to describe involve motions and interactions of objects; the manipulator is merely a means to an end. Thus, motion statements written in terms of object frames are a much more direct means of describing the programmer's intent than joint angles or even sequences of hand positions would be.

In addition to increased clarity, this directness leads to more durable programs. "Toughness", as applied to computer programs, is a somewhat fuzzy concept. Generally, we

---

[26] For instance, expected hand openings or "stop on force" thresholds make use of shape information without representing it explicitly.

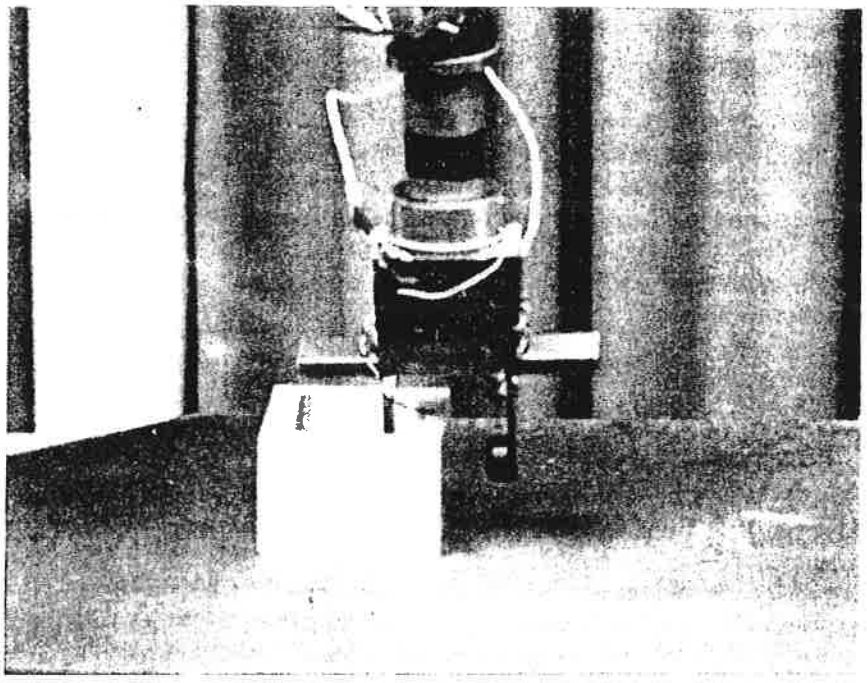


Figure 3.6. Finding the Box by Centering

are interested in a measure of how sensitive a program is to changes in the environment in which it is executed or in the task for which it was written; small changes in assumptions or specifications should not require us to rewrite major chunks of code.

From this point of view, **frame** specifications are distinctly better than joint specifications. Changes in the work station configuration are generally easier to describe in three dimensional Euclidean space than in joint angle space. More importantly, there are understandable language constructs (**affixment**, **frame** transformation, etc.) that allow us to describe how the behavior of the program should depend on the relevant input parameters. This means that variations in the initial position of the box in the previous section require that only a single assignment statement be changed for the program to work in the new situation. The advantage is magnified immensely when one considers the fact that the programs can be written to perform information-gathering operations themselves, thus allowing them to handle many situations without further human intervention. [27]

A further advantage of treating the manipulator as a disembodied hand is that the user doesn't have to rewrite the program if a different arm is used; the programming system assumes responsibility for translating his directions into the appropriate (hardware-dependent) joint trajectories. Of course, manipulators are not ideal objects, and the fact that different designs may have somewhat different characteristics necessarily introduces some degree of hardware dependence into programs. Since AL has, as yet, only been

---

[27] In principal, it is possible to devise systems that allow a similar degree of adaptability using purely joint-angle formalisms. However, it seems likely that the resulting programs would be extremely difficult to write and read.

implemented on one manipulator, it is hard to get a firm handle on exactly where the limits to its "universality" lie. Nevertheless, there is good reason to believe that the language allows at least approximately manipulator-independent programs. The following observations tend to support this view:

1. The characteristics of the Scheinman arm are not uniform through its entire useful "working radius", due (in part) to differences in joint sensitivities, gravity loading, and other factors that are not entirely compensated by the runtime system. Nevertheless, AL programs exhibit a reasonably high degree of stability over much of this space. Programs break down when hidden assumptions about the capabilities or behavior of the manipulator are violated. Typical causes of trouble include:

   A position is no longer reachable by the arm, due to joint limits being exceeded. [28]

   The motions of the manipulator in getting to the required destination frames are different in some deleterious way. Typically, this results from the fact that several arm solutions can exist for a particular frame specification, as will be discussed later.

   The arm's elbow (or wrist, or finger) hits something.

   The accuracy of the manipulator is degraded to the point where some critical tolerence is no longer met.

   The interesting thing about these causes for failure are that they are of the same character that one might expect from changing manipulators. Also, it is interesting to note that the fixes seldom require much recoding. Generally, changing a grasping position or adding a via point does the trick, although degraded accuracy is much harder to get around.

2. Recalibrations or modifications to the manipulator have been made periodically during the development of the system. These usually don't affect the validity of programs.

3. Two Scheinman arms, having somewhat different link sizes and joint limits, were interfaced to the WAVE system. Programs written for one arm could be used or adapted for the other with little difficulty.

Program toughness is further enhanced by the use of object frames, rather than the hand frame, to describe motions. Variations in object shape or manipulator characteristics, for instance, may require that an object be grasped with the hand in a different relative orientation. If the manipulator motions are all specified in terms of hand positions, then any statements using the revised grasping position may have to be rewritten. If the motions are specified in terms of object frames, then only the grasp sequence needs revision.

---

[28] Between manipulators of different designs, the failure might also be due to loss of a degree of freedom, or some other kinematic difficulty.

Actually it is possible to write all motion statements in terms of expressions computing hand positions from the destination desired and the current relation between the hand and the object being moved. Indeed, the AL compiler rewrites motion statements into exactly that form as one step in the compilation process. The point is that having the translation done automatically makes it easier to produce programs that take advantage of the capability.

We have already discussed some limitations in the "hardware independence" assumption. Several other points in this regard should, perhaps, be mentioned. Difficulties with the assumption fall, roughly, into two catagories:

1. The frame specification may assume capabilities not possessed by the actual hardware. The paradigm is based on an ideal six degree of freedom manipulator operated by levitation. Since real arms differ from this model, it is almost inevitable that certain other assumptions must be built into the programs. Limiting factors like link configuration, joint limits, accuracy, strength, and speed all fall within this catagory. Their effects are generally pretty obvious; this discussion won't dwell further on them.

2. The manipulator may possess important attributes that are impossible (or very difficult) to express in terms of the hand frame. One obvious example is special sensory hardware; to some extent this must be handled *as* special hardware (For instance, touch sensors are a good example. AL programs use "**on** *touch_sensor* **do** ... " and similar constructs to handle them.) However, the limitation is rather more severe in dealing with *kinematic* properties, since these are what frames were supposed to be used for. Examples include:

   There are frequently *multiple arm solutions* for a given hand position. There is no direct way for a user to say which solution he wants used, so the system makes its best guess. This can make a difference in the path followed by the arm in moving to its next position[29]; occasionally, the effects are rather dramatic (and confusing).

   If the manipulator has more than six degrees of freedom, then the number of solutions possible is much increased. In such cases, it seems especially unfortunate not to let the user say more about the desired joint configuration, since the extra freedom can be extremely useful in collision avoidance.

   In principle, these difficulties may be resolved by the use of enough via points and the addition of extra frames to pin down extra degrees of freedom. However, these solutions are not particularly palatable; eventually, some better way for the user to say what he means must be found.

---

[29] Here, it is important to recall that, although the manipulator may be able to reach any position, the joint limits may require that it follow a rather devious path in going from one point to the next, even when the successive positions are "near" to each other.

In addition, there are several difficulties with frames that are not tied quite so strongly to differences between manipulators. The first of these is that measurement of locations in not always trivial. A significant part of the effort in writing an AL program is spent in producing the initial frame values and affixments. Eventually, one hopes that the affixment trees for the objects being manipulated can be produced automatically as a by-product of computer aided design programs. However, the problem of "surveying in" the initial positions would still remain. One partial solution is to provide the user with more natural descriptive formalisms for saying where things are. For instance, let him say, "The box is held in the vise with surface A against jaw 1 and the bottom surface resting on the table top." Chapter 7 describes how such descriptions can be used to generate position estimates. Other options include use of vision or the manipulator hardware to construct the necessary structures. Of these, the latter is the more commonly used. We have developed a system which allows the user to define frames and affixments interactively, using the manipulator as a measuring device. Once the structures are built, the corresponding AL declarations and statements are then written into a file, which can then be used in an AL program. This system is discussed further in Appendix F. One interesting point about this approach is that it introduces a certain calibration dependency into programs. The values assigned to frame constants are chosen, not so much because they are the "most accurate" model of actual positions as because using those values gets the manipulator to the "right" place.

Another difficulty is that representing objects as affixment trees of frames fails to account for some very important aspects of the objects (like shape) and may introduce artificial restrictions into a motion specification. This problem can be especially severe where symmetries are involved.

For instance, consider the pin we've been playing with throughout this discussion. We've assigned its coordinate frame so that the z axis lies along the axis of symmetry of the pin. The definition of the other two axes is rather arbitrary. Unfortunately, the variable *grasp_pin* must be affixed to *pin* in a fixed place. This means that we cannot say something like "grasp the pin in the position most convenient for the arm, provided only that the approach vector of the hand intersects the z of the pin at 150 degrees and the finger pads grasp a point 1.3 inches along z."

The problem gets even worse when we consider the pin insertion step. *hole* is also symmetric; its affixment to *box*, consequently, is somewhat arbitrary. The program specifies that *pin* is to to be made to coincide with a frame, *in_hole_position*, affixed to *hole*. If we grasp the pin at an angle, then the program will not work unless the rotation of *in hole_position* with respect to the manipulator base is acceptable. Unfortunately, this means that our affixment decision cannot be made independent of the anticipated position of the box. Once again, we really want to be able to say something like "make the pin axis correspond with the hole axis, using whatever symmetry seems most appropriate."

It is possible to get around this difficulty by adding additional calculations to program to rotate *pin* and/or *in_hole_position* into good symmetries just before the motion statement is executed. However, such calculations are sometimes complicated, make programs harder to read, and are good breeding grounds for bugs. Eventually, a more direct formalism must be found.

Finally, we should note that an unaugmented frame paradigm is insufficient for collision

avoidance. It is most undesirable for the manipulator to go crashing into things.[30] The only way to handle this in AL is to supply enough via points so that the machine stays out of trouble. This can be a bothersome problem; we would like the programming system to take over and do it for us. To do so, shape information is clearly needed. As we have noted, this information is missing from a frame tree.

Collision avoidance is a very messy problem; it will not be addressed in this document. No one knows how to do a good job yet. However, it is fairly clear that any software likely to be developed soon will be too slow and large to be executed in real time. This suggests that the next step is likely to be a module that executes as part of the compiler (or automatic programming system) and generates a set of via points, based on an expected model of the runtime environment.

3.6 Affixment

Affixment performs two major functions in AL programs;

1. It allows description of motions in terms of object frames.

2. It provides for the automatic updating of variables.

The use of frames for motion specification was discussed in the previous section. This section won't go much further. It is perhaps worthwhile to note, here, that the *compiler* assumes responsibility for transforming object destination expressions into the corresponding hand expressions. This is largely a consequence of the decision to pre-plan trajectories: the expressions must be computed in order to get planning points for the trajectory calculator, and it seemed wasteful to put the corresponding symbolic reduction capabilities into the runtime code. The principal consequence (as seen by the user) is that certain options – such as deciding at runtime which of several arms to use – are ruled out.[31] There are also a number of consequences for the compiler's planning model, which is discussed in Chapter 5.

The importance of automatic updating in simplifying programs can hardly be overemphasized. For example, without this capability, each *pin* moving statement in our sample program would require a number of assignment statements:

```
move pin to pin_insert ... ;

pin_grasp ← pin*trans(rot(xhat,180*deg),vector(0,0,2*cm));
pin_grasp_approach ← pin_grasp*trans(nilrotn,vector(0,0,-3*cm));
```

For more complex structures the blowup is even worse. Although it is true that not all

---

[30] Except deliberately, as when a nail is being driven or a careless experimenter is within reach.

[31] Not strictly true, since code for each arm can be compiled, but the cost usually gets rather high.

frames have to be updated after every motion, if we make a practice of leaving things out, then the chance of creating a buggy program is much increased.

In addition to enhancing convenience and bug-resistance, affixment increases the toughness of programs in roughly the same ways as the use of frames does: if we introduce a new linkage or modify an old one, we only have to modify one spot in the program in order to account for the change. An interesting point here is that reliance on a *side effect* produces a marked improvment in program clarity. Side effects have acquired an evil reputation among computer scientists, since they frequently make programs very hard to analyze. Aside from any problems they may cause the programming system,[32] it is often very difficult to track down bugs which are produced by statements far from the place where the symptoms first appear, or to understand code which modifies variables whose names do not appear in the statements concerned. Indeed, an incorrect affixment in an AL program can produce spectacular and puzzling results. What saves us from total confusion is the strong connection between affixment and the semantics of moving an object: when the object moves, its subparts also move. Thus, affixment helps us to partition the problem of writing (or understanding) a manipulator program into "descriptive" and "active" components roughly corresponding to physical reality.

One way to implement affixment would be to emulate a programmer writing in a language lacking the construct. I.e., keep track of what is affixed to what and then insert the appropriate assignment statements wherever they are needed. This approach would be consistent with that taken in rewriting motion statements. However, several objections come up.

1. *Many* extra assignment statements would be produced. This would tend to make the object programs very expensive, both in size and execution time.

2. It is very difficult to keep track of all the affixments in a program. If, somehow, we should lose track of some of them, then the variables involved would not be updated correctly, and the manipulator could then run amok.[33]

A natural alternative to resolving affixments at compile time is to build a data structure that supports runtime updating. The solution we have adopted is to represent affixment in terms of "continuously evaluated" expressions. For instance,

---

[32] AL is not excepted. Affixments do indeed cause difficulties for the compiler's planning model, which is needed in the preparation of trajectories.

[33] It might be argued that this is a system bug, and we are no worse off, in any case, since the compiler needs to know about affixments in order to compile motion statements. However, such is not the case. If an affixment link between the hand and an object is lost, and we say "move *object*", then the compiler will discover that it has lost. If a subpart link is lost, then there is no such check. Trajectories planned with incorrect nominal targets may still be useful, provided that the runtime values are correct; if the compiled code fails to produce the correct runtime updating, then there isn't much hope of winning.

affix *hole* **to** *box* **rigidly at** *holexf*;

would be represented by two expressions:

hole <= *box***holexf*;
box <= *hole****inv**(*holexf*);

where "<=" is read as "is computed by". From the user's point of view, any change to the value of *box* should cause the assignment statement *hole* ← *box***holexf* to be executed. Similarly, changing *hole* should cause *box* ← *hole****inv**(*holexf*) to be executed. This information may be represented in the runtime system by keeping a list of compiled statements to be executed whenever a variable value is changed. **affix** and **unfix** are then implemented by compiling code to add and delete the appropriate links to the resulting graph structures. As stated, there are several problems with this approach, one of the most important being that changes to popular variables may cause much needless recomputation of values that may change many times before they are needed. Therefore, AL takes a slightly different tack: keep "validity" information with each variable. When a variable is changed, invalidate all values that depend on it; whenever a variable is needed, but its value is invalid, run through the set of expressions associated with it, looking for one which can be used to compute a valid value. Information associated with each runtime variable thus includes the following:

1. **value** – a frame, vector, scalar, or what-have-you.
2. **validity mark**
3. **dependents** – list of nodes in the "graph structure" to be invalidated whenever this one is.
4. **calculators** – list of expressions that may be used to recalculate this variable.
5. **side effects** – list of additional statements that are to be executed whenever this variable is changed.

The actual algorithms, which are given in Appendix C, include a number of refinements not apparent from this rough description.[34] This approach has proved to work quite well, and has been flexible enough to allow a number of useful extensions beyond our original reasons for adopting it. For instance, quantities parameterized by time or some other external signal – e.g., the position of a moving conveyor belt – are readily represented. An interesting potential limitation is that the affixments themselves are no longer *explicit* in the data structure. For present AL programs, this is no particular limitation. However, it does tend to rule out applications where the runtime program needs to make decisions based on

---

[34] For instance, it has proved convenient to assign a "node" to every expression, as well as every variable. Persons interested in languages for which continuously evaluated expressions are useful will, no doubt, find many of these problems familiar, and may be interested in the solutions we have adopted.

what is affixed to what. [35] If it should become desirable to keep track of affixments, *per se*, this could be done by means of "extra" data structures, which, presumably, could include pointers linking them with their corresponding expression graph elements. An additional benefit of such a structure would be increased support for debugging systems, since it would then be possible to keep track of what affixment statement was responsible for a particular side effect.

Two important limitations of **frame** affixment deserve mention. One is the fact that objects can be linked in many ways that aren't fully or easily describable by **affix** statements. For instance, joints may slide or rotate within certain limits. Often, the validity of an affixment assertion may depend on certain constraints that are not explicitly stated. Thus, it was reasonable to **affix** *pin* to *hole* so long as the box in which the hole is drilled isn't turned upside down. Similarly, if we place an object on a tray and move the tray, the object will move with it, so long as we keep the tray upright and don't move it too fast. Although some efforts have been made to deal with these difficulties,[36] no one has yet found a really adequate solution. Fortunately, a large class of useful programs can be written without needing a "full" solution to this very sticky problem.

The other difficulty with affixment is that, while it is an excellent means of propagating the effects of a motion or calculation, it is somewhat limited as a means of *inferring* correct object positions from multiple data points. The difficulty is illustrated by the following code.

```
frame box,pin1,pin2,hole1,hole2;

affix hole1 to box rigidly at hole1_xf;
affix hole2 to box rigidly at hole2_xf;

{ Code to insert pin1 in hole1. }
hole1 ← pin1; { This updates box. }

{ Code to insert pin2 in hole2.}
hole2 ← pin2; { This updates box, too.}
```

Suppose that the box's actual position is rotated slightly from the value given by the variable *box*, so that each hole is displaced slightly from its correct position. The assignment *hole1* ← *pin1* will cause the value for *box* to be translated so that *hole1* has a correct value. In the absence of any other information, this seems like a reasonable thing to do, although, unfortunately, it can displace *hole2* even further from its "true" position. Now the program goes on and makes the second pin insertion operation. The assignment *hole2* ← *pin2* causes *box* to be translated so that *hole2* is now correct. Unfortunately, this undoes the value we computed for *hole1*. The problem is that affixment *lacks memory* – it only can take the most recent change into account. This is clearly insufficient for computing rotations. To get the right effect, we can always do a calculation, such as:

---

[35] E.g., deciding what arm to use to move an object.

[36] E.g., Wesley & Lieberman at IBM [63] and the constraint work described in Chapter 7

```
hole1 ← pin1; { Get hole1 translation correct. }
h2v ← inv(hole1)*loc(hole2); { Center of hole2 with respect to hole1. }
p2v ← inv(hole1)*loc(pin2); { Center of pin2 with respect to hole1. }
hole1 ← hole1 * vvrot(h2v,p2v); { rotate everything into place. }
      { vvrot(v1,v2)*v1 always points in the same direction as v2 }
```

but this sort of thing can get old very fast, especially in cases where the geometry is more complicated. A related difficulty is that each measurement may be susceptable to error, so it may not be entirely clear what to believe. What we need is some way to represent how the different mearurements are related to the variables in the runtime model, and then to compute a "most consistent" interpretation to account for the data observed. The methods developed in Chapter 7 provide a basis for doing just that.

Chapter 4.

Planning Models

4.1 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. Since the "universal" program has yet to be written, any program necessarily 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 of the information relevant to the coding decisions it is asked to make.

4.2 Planning Information in Algorithmic Languages

As we indicated above, the use of planning information is not unique to manipulator programming. It is useful to consider, briefly, the information required to write programs in a more "traditional" domain.

4.2.1 An ALGOLish 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.[2] Similarly, one of the strongest claims of "structured programming" advocates is that one should proceed from such assertions to a "correct" program. Thus, one might work something like this:

> I need a way to hold the maximum element; I'll invent a variable to do this – call it *maxel*. Now, I need a way to get the maximum of a sequence $a[1],...,a[n]$. Some sort of iteration looks promising. If I had a way to guarantee that *maxel* was the largest element in the first $i-1$ elements, then if *maxel* is not less than $a[i]$, then it will also be the largest element in $a[1:i]$; otherwise $a[i]$ will be, so I'd better assign it to *maxel*. How about boundary conditions? After $i=n$, *maxel* is the maximum in $a[1:n]$, which is correct. What about $a[1]$? Oops! *maxel* better be initialized to something. If I pick the most negative integer, then either all values of a will be that small, in which case I've got the right answer, or else the test will succeed, and *maxel* will be set to a number in the sequence. Ok, that looks like a winner. I need an iteration from 1 to n; a for loop does that.[3]

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 a loop to compute a maximum element, most programmers would reach into a grab-bag of tricks, and pull out a skeleton program structure, and then fill in the appropriate slots.[4] To some extent, programs are 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 that *maxel* is set to the maximum element of $a[1:n]$ would be a typical "high level" fact useful primarily in performing the problem-solving function.[5] Coding information includes such matters as the

---

[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] See, for instance, [104, 4]

[3] There is also considerable interest in automating the process. See, for instance, [109, 68, 43].

[4] 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.

[5] Similarly, the fact that a is unsorted is important in deciding to use this particular trick, rather than, say, just setting $maxel \leftarrow a[1]$.

fact that i is available for use as an index variable, a is the name of the array to be searched, etc.

4.2.2 Getting the Computer Involved

A dominant theme in the history of programming system development is the progressive transfer to the computer of coding responsibilities. As we suggested in the previous section, 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., thus 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" to their input specifications.

For instance, a reasonably good compiler might translate our search loop into (PDP-10) machine code something like:

```
        hrlzi   1,400000        ; maxel← -2^35
        movei   2,1             ; i←1
l1:     camle   2,n             ; if i>n then
        jrst    l2              ; go to l2[6]
        cail    2,1             ; if i < 1 or
        caile   2,=100          ; if i>100 then
        err     ["Bounds error for array a"]
        camge   1,a(2)          ; if a[i]≥maxel
        move    1,a(2)          ; then maxel←a[i]
        aoja    2,l1            ; i←i+1 and go to l1
l2:     movem   1,maxel         ; save maxel in memory cell
```

Here, the machine "knows" that the function of the instruction at $l1$ is to test the termination condition of the loop, that i is being kept in register 2, *maxel* in register 1, and other similar information. A somewhat cleverer compiler might realize that n doesn't change in the loop, that i runs from 1 to n, and that i is only used as an index to a. This information can then be used to produce more efficient code:

---

[6] "Jack be nimble, Jack be quick; Jack jrst over the candle stick!" [R.E. Sweet]

```
        hrlzi   1,400000        ; maxel ← -2^35
        skipg   2,n             ; if n≤0 then
        jrst    l2              ; don't bother with the loop
        caile   2,=100          ; test array bound
        err     ["Value of n will cause index overflow of a"]
        hrloi   2,-1(2)         ; Well known PDP-10 trick
        eqvi    2,1
l1:     camge   1,a(2)          ; if maxel<a[i] then
        move    1,a(2)          ; maxel ← a[i]
        aobjn   2,l1            ; i←i+1;iterate
l2:     movem   1,maxel
```

This little exercise illustrates several points: The compiler uses its "understanding" of the formal semantics of the source language and of its 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. The more sophisticated the book-keeping, the better the job it can do. However, there are limits imposed by what can be stated 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 that (because of some earlier code) *a* is sorted and compile

```
        move    1,a             ; maxel←a[1]
        movem   1,maxel
```

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.[7]

An interesting point, here, is that the user may wish to share the coding responsibility with the computer. For 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 computer, 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[8]) to outlaw such tampering. Another possibility, however, is to provide constructs that allow the user to tell the system about relevant assumptions or effects for a particular piece of code. For instance,

---

[7] Recently, there has been a great deal of interest in "very high level" languages, in which programs are expressed in terms of operations on abstract information structures. See, for instance, the work of Low [66], Barstow[7], Schwartz [98], Early, and many others [5]. Sometimes, this work is carred out within the context of a larger automatic coding effort. For example, the "acquisition" and "synthesis" components of the PSI project at Stanford [45] correspond to the "problem solving" and "coding" functions discussed above.

[8] One is tempted to say, "ideologies".

```
register integer i,j; { Tell compiler that i and j are to be kept in accumulators }
real x,y;
      :
start_code { tell compiler you are going to take over }
register r;
movei   i,3(j)      ; i←3*j
skiple  r,a(i)      ; r←a[i]
skipa   r,x
halt
assert in_register(r,x); { Tell compiler that x is in r }
end

y←x*3;
```

Here, we have told the compiler that we want two variables to be kept in registers at all times. Also, the assertion tells the system something that it might not otherwise figure out: If the start_code block ever exits, then x will be in register r. This information can be used to good effect in compiling the assignment, $y \leftarrow x*3$.[9]

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. We will see instances of this in Chapter 5, where the system loses track of important facts, and must be helped out by assertions. In Chapter 8, we will describe procedures for making the coding decisions of Section 3.4 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 modification by the user. Again, some assertional mechanism is almost certainly necessary to help the system "understand" code written for it by the user.

4.3 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.

---

[9] Another possibility, investigated by Samet [96] 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.

3. Action information defining the task and semantics of the manipulator language.

Subsequent sections will discuss these issues in greater detail.

4.4 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 3.4, 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.[10] 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 3.4, 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:

---

[10] The language design of AL also included provisions for associating mass and radii of gyration with an object's **frame**, for use by the servoing routines. Although this feature was present in WAVE, it has yet to be implemented in AL, however.

affix *hole* to *box* at trans(nilrotn,vector(*3.85**cm,*3.20**cm,*4.90**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 Chapter 6; and further discussion will be postponed until there. 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, or in several different forms to serve different purposes.

4.5 Situational Information

Manipulation programs transform their environment by moving objects around. This means that the principal fluent properties[11] that must be considered are:

1. Where objects are in the work station.
2. What objects are attached to each other.
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.

---

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

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 manipulator 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. For instance, 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.

4.6 Action Information

Clearly, it is necessary to understand the semantics of the manipulator language in order to write programs in it.[12] 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 "tricks", or 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.

```
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 8, these may be computed from the situational and object modelling information.

---

[12] 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.

Chapter 5.

The AL Planning Model

In Section 4.2 we saw that any compiler is, to some extent, an automatic coding system. AL is no exception; it keeps track of variable bindings, control points, etc., and uses this information to generate object code for the runtime system to interpret. However, it also performs several coding functions which are not usually found in an algebraic compiler and which require AL to keep a somewhat more complete model of situational information than would otherwise be the case. Since a major theme of this document is the use of planning information to make coding decisions, this chapter will examine these functions and how AL keeps track of the necessary information.

5.1 Planning Model Requirements for Manipulator AL

To the extent that AL is "just another compiler", it is not very interesting. The internal structures used to keep track of ALGOL-like entities are very standard and will not be discussed further.[1] Where AL differs from other language systems is the extent to which it must maintain a model of situational information describing the expected value of variables and affixment structures through the program. Before we get into the mechanisms used to maintain this information, it is worthwhile to review, briefly, the coding functions that require them.

Planning Trajectories

An important design decision in AL was the use of pre-computed "joint trajectories" for motion control. Although the trajectories are modified at execution time, the precomputation requires that we know approximate values for starting, intermediate, and finishing points of each motion statement. Consequently, the compiler must maintain "planning values" for any variables that can enter into motion statement specifications.

Eventually, we hope that trajectory calculation can be moved into the runtime system. However, we also hope to add a collision avoidance facility. Since early implementations are apt to be too expensive to be used without preplanning, the need for planning values is likely to be with us for some time.[2]

Rewriting Motion Statements

The **affix** construct in AL allows one to specify linkages between frames and to describe motions in terms of object frames, rather than merely in terms of hand positions. Although the AL runtime data structures do reflect affixments, the interpreter requires hand positions to be specified explicitly in motion statements. Consequently, the compiler must *rewrite* all

---

[1] They are described briefly in [18].

[2] Of course, planning values are not all we will require. Some sort of shape description is also necessary.

the destination expressions so they are in the correct form before calculating trajectories. This process, which is discussed further in Section 5.4.2, requires that the compiler know the affixment chain between the frame whose motion is specified in the user's statement back to the arm which is to be moved.

Resolving Conditional Compilation

The AL language design includes facilities for conditional compilation of source programs. These facilities, which we will discuss briefly in Section 5.4.3, are intended to allow the user to construct a library of general purpose macro-operations. Such a macro library would constitute a sort of poor man's automatic coding system, with macro calls being used to reduce the programming effort required for a particular program.

There is nothing particularly new in this idea. Subroutine libraries are very nearly as old as computer programming. Similarly, many symbolic assemblers and compilers[3] offer extensive macro and conditional compilation facilities. An important difference between AL and these other systems is that the coding decisions made by the latter generally rely on *static* properties of the program (for instance, switch settings or the data type of variables), whereas the decisions for manipulation programs often depend on expected *runtime* states (such as which grasping point will be reachable by the hand).

5.2 Structure of the Model

The principal elements of situational information that AL *must* maintain are planning values and affixments. It would probably have been possible to design efficient special purpose structures for this information. However, we believed a more general representation, which would allow a wide variety of facts to be represented and manipulated uniformly, to be desirable. This generality is very important to users wishing to construct general-purpose macros and allows us to employ the same mechanism to model situational information for our own automatic coding primitives.

Consequently, we have chosen to represent all situation dependent facts by assertional "patterns" having the general form

(*element_1, element_2, ..., element_n*)

The two most important patterns are those used to specify planning values and affixments:

(**value**, *variable_name, value*)
(**affixed**, *frame_var_1, frame_var_2, by, trans_var*, [non]rigidly)[4]

which correspond to the results of statements like:

---

[3] For instance, SAIL [36,107,65] and PL/I.

[4] For simplicity of explanation, these patterns have been modified slightly from those used by the actual implementation. However, they are close enough to reality for the use we will make of them.

variable_name←value

and

affix *frame_var_1* to *frame_var_2* by *by_trans_var* [non]rigidly

With each statement in the program, we associate a pair of data bases containing all the facts known to be true before and after the execution of the statement. Simple simulation techniques, which are described in Section 5.4, are used to build these data bases. As we will see, there are limits to how well we can hope to do; occasionally, we may require help from the user in the form of assertions describing the effects of confusing code sequences.[5]

5.3 Data Base Primitives

As we have stated, information about program states is represented by assertional patterns stored in a data base. When a particular fact, such as the expected value of a variable, is required, it is found by matching a template pattern against the stored assertions. As there is an extensive literature describing such data bases,[6] there is no need to dwell on the particular implementation used here. However, a few general points will be helpful for understanding the subsequent discussion.

The basic elements in the data base are called "facts". Associated with each fact is the following information:

1. An assertional pattern.
2. A unique identifier[7] which is useful in talking about collections of facts or associations between facts.
3. A set of "worlds" in which the fact is asserted to be true. Each such world may be viewed as a sub-database containing the fact.
4. A list of actions to be taken whenever the fact is asserted or denied. This feature is mostly used to implement dependencies between facts.

The data base management routines fall generally into three categories:

1. Functions for manipulation of fact patterns.
2. Functions for manipulating worlds as if they were sets of facts.
3. Functions operating on facts by name.

---

[5] This is an instance of a more general phenomenon: it is almost always easier to say what a given piece of code does when you have written it than it is to read and understand an undocumented program. Thus, we can look at these assertions as machine-readable comments.

[6] See, for instance, [20], [70], [102]

[7] Internally, a LEAP item.

The principal routines in the first class are illustrated below:

assert(*w*,**pattern**(*fee*,*fie*,*fo*,*fum*)) – Adds the pattern "(*fee*, *fie*, *fo*, *fum*)" to the sub-data base for world *w*.

deny(*w*,**pattern**(*fee*,*fie*,*fo*,*fum*)) – Removes the pattern "(*fee*, *fie*, *fo*, *fum*)" from the sub-data base for world *w*.

match(*w*,**pattern**(*fee*,? *x*, ? *y*, *fum*)) – generates successively all four element patterns in *w* with *fee* as the first element and *fum* as the fourth. ? *x* and ? *y* are bound to the two middle elements. Similarly, **match**(? *w*,**pattern**(...)) generates all worlds *w* matching the specified pattern.

The second class includes:

copy(*w1*,*w2*) – copies world *w1* into world *w2*. It will be useful to adopt the notation ⟨*w*⟩ to refer to the set of facts true in ⟨*w*⟩. Thus, this has the effect of setting ⟨*w2*⟩←⟨*w1*⟩.

w←**new_world** – invents a new "world" item and returns it in *w*

clrwld(*w*) – removes all facts from *w*. (I.e., ⟨*w*⟩ ← null)

andwld(*w1*,*w2*,*w3*), **orwld**(*w1*,*w2*,*w3*), and **difwld**(*w1*,*w2*,*w3*) – perform the "obvious" set operations on worlds *w1* & *w2* and put the result in *w3*.

The third class is somewhat more varied. Typical routines are:

fassert(*w*,*f*), **fdeny**(*w*,*f*), and **true_in**(? *w*, ? *f*) – assert, deny, and generate facts by name, rather than by contents.

say_relies(*w*,*f1*,*f2*) – asserts that *f1* "relies on" *f2* in world *w*. If *f2* is subsequently denied in *w*, then *f1* will be also. This assertion is itself a fact in the data base. Consequently, the "reliance" will be inherited by any worlds copied from *w*.

5.4 Simulation of Language Constructs

5.4.1 The Basic Step

With every statement in the program graph, we associate an "input world" and an "output world", which contain assertions about the expected state before and after the statement is executed, respectively. For instance, for the sequence

s1: move blue to *pin_grasp*;
s2: close blue;
s3: affix *pin_grasp* to blue;
s4: move *pin* to *in_hole_position*;

would produce world assignments:

statement	input world	output world
s1	*w0*	*w1*
s2	*w1*	*w2*
s3	*w2*	*w3*
s4	*w3*	*w4*

The planning model simulation works under the assumption that only those attributes of a world state that are known to be changed by a particular statement will in fact be changed. The essential step in simulating each statement is to copy the input world into the output world and then modify the output world in accordance with the understood semantics of the statement. Simulation of the code sequence above would produce a data base containing elements somewhat like those shown below:

fact	assertion	worlds
f1	(affixed,*pin_grasp*,*pin*,*pgxf*,rigidly)	*w0*,...,*w4*
f2	(affixed,*pin*,*pin_grasp*,inv(*pgxf*),rigidly)	*w0*,...,*w4*
f3	(affixed,*pin_grasp*,blue,*g001*,nonrigidly)	*w3*,*w4*
f4	(value,blue,*frame0*)	*w0*
f5	(value,blue,*frame2*)	*w1*,*w2*,*w3*
f6	(value,blue,*frame4*)	*w4*
f7	(value,*g001*,niltrans)	*w3*,*w4*
f8	(value,*pgxf*,*trans0*)	*w0*,...,*w4*
f7	(value,*pin*,*frame1*)	*w0*,*w1*,*w2*,*w3*
f8	(value,*pin*,*frame3*)	*w0*,*w1*,*w2*,*w3*
f9	(value,*pin_grasp*,*frame2*)	*w0*,*w1*,*w2*,*w3*
f10	(value,*pin_grasp*,*frame4*)	*w4*
f11	(value,*in_hole_position*,*frame3*)	*w0*,...,*w4*

Initially, blue is at *frame0*, as asserted by fact *f4*, which is true in the initial world *w0*. To simulate the semantics of *s1*, we copy *w0* into *w1*, and then modify *w1* by deleting *f4* and asserting *f5*, which puts blue at the initial position of *pin_grasp*. In this case only one fact is changed. *s4* is somewhat more interesting. Here, the simulation must use its knowledge of the affixment structure to update *pin_grasp* and blue as well as *pin*.

5.4.2 Rewriting Motion Statements

An additional requirement for processing *s4* is to rewrite the statement in a way that the code emission and trajectory calculator parts of the compiler can understand. To do this, it is first necessary to determine the affixment chain leading back from *pin* to a "controllable" manipulator. The procedure for doing this is very straightforward:

```
expression procedure chain(world w;variable a;reference set already_seen);
    begin
    if a = "blue" or a = "yellow" then
        return("a");
    if a ∈ already_seen then return(null_expression);
    put a in already_seen;
    for each b,byexp such that match(w,pattern(affixed,a,b,byexp,any)) do
        begin
        c ← chain(w,b,already_seen);
        if c ≠ null_expression then
            return("c*byexp")
        end;
    return(null_expression);
    end;
```

The code above has been written in a "cleaned up" language based on SAIL [107,65] and is presented as an illustration of the sort of manipulations that go on using the planning model. The procedure is essentially a depth first search of a directed graph and works roughly as follows. When *chain* is entered, *a* is the name of a frame variable whose affixment to a manipulator must be determined. The value returned is to be an *expression* computing *a* in terms of one of the manipulator frames. First, a check is made to see if *a* is itself one of the manipulators, if so, the procedure just returns "*a*". If not, the procedure finds all variables, *b*, which are asserted to be affixed to *a* in planning world *w*. For each such *b*, it calls itself recursively to find a chain from *b* to a manipulator. The set, *already_seen*, is used to break cycles.

For example, suppose we have the data base shown in the previous section. Then the call

```
already_seen←empty;
c ← chain("w3","pin",already_seen)
```

would return "blue*g001*inv(*pgxf*)" as its value. Using this result, the compiler can rewrite *s4* as

```
move blue to in_hole_position*inv(g001*inv(pgxf))
```

i.e.,

```
move blue to in_hole_position*pgxf*inv(g001)
```

5.4.3 Conditional Compilation

One of the design criteria for AL was that it should be possible to write programs in some generality, save them in a library, and then use them with little or no modification in future tasks. The language therefore contains an extensive conditional compilation facility designed to allow programs to "tailor" themselves to a particular task environment without having to do extensive runtime checks or incur needless overhead from code that is never executed. Although we won't dwell on this aspect of AL, it is important to note that the language constructs involved rely quite strongly on the planning model.

The most direct example is the use of assertions or compile-time variables as "switches", either to select options or to stitch together different pieces of code. These uses are illustrated below. Suppose that we have written a "more general" version of the pin-in-hole primitive described in Chapter 8. This version allows the user to insert screws as well as aligning pins, and to specify whether or not the error recovery loop is to be included. A typical code sequence might look like this:

```
assert (kind,screw1,screw);
assert (kind,screw2,screw);
careful_flag ←← true;[8]

call pin_in_hole(screw1,hole1);
call pin_in_hole(screw2,hole2);
```

Here, *pin_in_hole(pin,hole)* would be defined as a macro, whose expansion would include the appropriate compile-time checks, as shown below:

```
plan if asserted (kind,pin,screw) then
   begin
   plan if not asserted (blue,holds,driver) then
      begin
      open bfingers to 3*inches;
      move blue to driver_grasp;
      affix driver to blue;
      assert (blue,holds,driver);
      end;
   { Code to use driver to pick up pin.}
   affix pin to driver;
   end
else
   begin
   plan if asserted (blue,holds,driver) then
      begin
      { Code to put driver down. }
      deny (blue,holds,driver);
      end;
   {  Code to pick up pin with blue hand }
   affix pin to blue;
   end
      :
move pin to hole;
plan if ¤(careful_flag) then
   begin
   {  Error checking & recovery code }
   end;
      :
```

[8] "←←" means that the planning value is to be updated, but no code is to be compiled.

Here, the first call to *pin_in_hole* will find the assertion (blue,*holds*,*driver*) false, and the code to grasp it will be included in the output program. On the next call, the assertion will be true, so the driver grasping code will be left out.

Planning values from motion statements are also an important decision criterion. For instance, we might have something like:

```
plan if zhat·orient(■(object_location))+zhat > 0 then
        move yellow to grasping_position_1
else
        move blue to grasping_position_2;
```

The details of these examples are not particularly important for our present purposes. What *is* important is the fact that the semantics of the program depend on the contents of the planning model. This means that planning and simulation cannot be separated.

5.4.4 Conditional Statements

AL treats conditionals in a very simple-minded way: the output world of an if statement is simply the intersection of the output worlds of the then and else parts. For instance, consider the code sequence,

```
    blue ←← place0 { Tells the compiler where blue is. }          [eg 1]
    pin ← blue; { Emits code as well as updates planning model. }
s1: if touch_sensor then
        s2: move blue to place1
    else
        s3: move blue to place2
s4: move blue to place3;
```

The world assignments for this fragment are:

statement	input world	output world
s1	*w0*	*w1*
s2	*w0*	*w2*
s3	*w0*	*w3*
s4	*w1*	...

Actually, these assignments are slightly oversimplified: two additional worlds are generally invented for use as the input worlds to the two branches, with assertions reflecting the semantics of the test. The resulting data base is as follows:

fact	assertion	worlds
f1	(value,blue,*place0*)	*w0*
f2	(value,blue,*place1*)	*w2*
f3	(value,blue,*place2*)	*w3*
f4	(value,*pin*,*place0*)	*w0*,*w1*,*w2*,*w3*

Here, the different effects of *s2* and *s3* on the planning value of blue are reflected by the differences in their output worlds. *w2* contains {*f2*,*f4*} and *w3* contains {*f3*,*f4*}. Thus the combined output world *w1* contains {*f2*,*f4*} ∩ {*f3*,*f4*} = {*f4*}.

This approach is "safe", in the sense that it doesn't make unwarranted assumptions about which branch of a conditional statement will be executed, but it is rather limited, in the sense that much useful information can be lost. Thus, the effect of the **if** statement, above, is to cause the compiler to forget where the blue hand is. When it gets to *s4*, it will then complain that it lacks sufficient information to plan a trajectory. This hasn't caused too much difficulty in the AL programs that have been written so far, largely because of the use to which conditional statements have been put: typically, an **if** statement has been used to verify that some error condition has not occurred, and to take appropriate remedial action if it has. The effect is to make the output of both branches of the conditional alike in most significant respects.

For those cases where a difference does cause important information to be dropped, AL requires the user to help out. There are several ways such help can be given. One way is to rewrite the program to leave both branches in a consistent state. For instance,

```
if touch_sensor then
        begin
        move blue to place1;
        move blue to place3;
        end
else
        begin
        move blue to place2;
        move blue to place3;
        end;
```

Another way is to use assertions to tell AL what to assume. Recall that the principal reason the *present* implementation needs planning values is to plan trajectories. Since these trajectories are modified at runtime, they do not require *exact* planning values.[9] Thus, the user may be able to get away with selecting some arbitrary position, *place4*, in between *place2* and *place3*, and telling AL to use that.

```
if touch_sensor then
        move blue to place2
else
        move blue to place3;
blue ←← place4; { Again, this is purely an assertion to the compiler. }
```

In some cases, it may be reasonable to ignore one branch or the other of a conditional, on the grounds that it can "never" happen, or that its effects will have no influence on the rest of the program, even though AL is too stupid to figure that out. For this purpose, AL

---

[9] Indeed recent improvements in the runtime system have made even grossly modified trajectories behave fairly well.

allows a user to specify that a particular planning world is to be used.[10] For example,

```
      move pin to place1;
s1:   if not touch_sensor then
          begin
          move blue to blue+vector(0,0,1);
          print ("Pin was dropped.  Put it back in hand and continue");
          wait go_signal;
          end;
      plan from s1;
          { Says to copy the input world of s1 into the current input world }
s2:   move pin to place2;
```

could be used to tell AL to ignore the conditional statement for purposes of planning.[11] It would probably be useful to add additional declarative assertions to the AL language, so that a user can say explicitly whether he expects a particular code branch to be executed. For instance,

```
move pin to place1;
if not touch_sensor then
        begin unexpected;
        move blue to blue+vector(0,0,1);
        print ("Pin was dropped.  Put it back in hand and continue");
        wait go_signal;
        end;
move pin to place2;
```

In addition to simplifying coding, this approach tends to produce clearer programs, since the user says explicitly what he intends. Here, we have an illustration of one of the main points of "structured programming" advocates: those things that are hard for a computer to follow are frequently difficult for a programmer as well.[12]

An interesting possibility, which hasn't been included in the present implementation, would be to *infer* unexpected declarations by evaluation of branch tests in the planning worlds. In general, this would probably require good object models and a much better understanding of the runtime semantics of manipulation than is now possible. However, it might be possible to win in certain cases. For instance,

---

[10] A useful default, not currently implemented, would be for the compiler to ignore the planning-model effects of any code sequences ending in an abort statement.

[11] The syntax for plan from used here is an invention. The construct was introduced into the language design after the design report (AIM-243) was published, and at a time when the "interim" (LISP like) syntax had already been adopted.[18]

[12] It may be more accurate to turn this around: if a construct is easily analyzed by a computer, then it won't be hard for a human, either.

```
move object to place;
if magnitude(loc(object) -loc(place))>0.5 then
     begin { AL assumes unexpected }
     :
     end;
```

Another alternative would be for AL to be more careful about its book-keeping. Instead of *forgetting* any facts not true in both parts of a conditional, it could remember what facts *might* be true. There are any number of schemes that could be adopted. One way for doing this would be to introduce a new "truth" value, "maybe", which could be assigned to any facts not known to be true in both branches.

A somewhat smaller modification would be to modify the data base entries to account for sets of planning values. This would mean that example [Eg 1] would produce a data base something like:

fact	assertion	worlds
f1	(value,blue,*place0*)	*w0*
f2	(value,blue,*place1*)	*w2*
f3	(value,blue,*place2*)	*w3*
f4	(value,blue,{ *place1*, *place2* })	*w1*
f5	(value,*pin*,*place0*)	*w0*,*w1*,*w2*,*w3*

Presumably, in compiling *s4*, AL could prepare two separate trajectories to get the hand to *place3* from *place1* and *place2* and include a runtime check to decide which trajectory to use.

Even if this can be done, there are several difficulties with this approach. Aside from the increased "hair" required in the compiler, the principal difficulty is the combinatorial explosion that results when value sets are combined. For instance, suppose that we have assertions

```
(value,pin,{ place1, place2 })
(value,hole,{ place3, place4 })
```

and then encounter code like

```
hole_to_pin ← inv(hole)*pin;
move pin2 to hole2*hole_to_pin;
```

We will wind up with four possible final values for *pin2*. Things get even more unpleasant when one recalls that *pin2* may have several possible input values, so that the number of paths which must be considered is much increased.[13] Eventually, it may be possible to

---

[13] It is interesting to compare this approach with the interval analysis done by Harrison [50], who analyzes PL/I programs to discover bounds on possible values for scalar variables (most importantly, indices) at control points. In addition to numerical limits, he is able to produce parameterized ranges of the form <relational operator><definition point><constant>. This analysis is shown to be useful for code optimization and diagnostic

characterize the "toughness" of a trajectory ahead of time, so that the compiler can, perhaps, pick a suitable point and plan only one trajectory. This ability to "collapse" planning value sets would be essential if AL were to start keeping multiple planning values.

So far, this discussion has focused primarily on uncertainties in planning values resulting from the compiler's inability to predict which branch of a conditional statement will be executed. The general difficulty is that AL allows the user to describe programs whose runtime semantics cannot be modelled fully by the compiler. It is possible to concoct far more pernicious examples of the limitation than mere loss of a few planning values. For instance, consider this program:

```
affix pin to blue;
if phase_of_the_moon > 0.5 then
        begin
        { Code to grasp pin with yellow }
        unfix pin from blue;
        affix pin to yellow;
        end;
move pin to dart_board;
```

Here, the compiler would not even be able to figure out which hand to move in order to move *pin*. Although this example may seem a bit farfetched, a somewhat similar problem could easily arise in programs where the exact task won't be determined until run time. Suppose we are writing a program to assemble a class of objects. The only difference between the various models is in which parts require heat treatment. The program would look like this:

```
if part_1_is_to_be_treated then
        begin
        { Code to move part_1 to pallet_place_1 }
        affix part_1 to pallet;
        end;
    :
if part_k_is_to_be_treated then
        begin
        { Code to move part_k to pallet_place_k }
        affix part_k to pallet;
        end;
{ Code to grasp pallet }
affix pallet to blue;
move pallet to oven;
unfix pallet from blue;
    :
{ Code to perform miscellaneous operations }
    :
move blue to part_1; { But where is it? }
```

---

purposes. For AL, the difficulty is in finding a useful way to represent "ranges" of frames or to predict how such ranges interact in arithmetic expressions. The techniques discussed in Chapter 7 would be applicable here.

Here, the compiler cannot tell which parts will be affixed to the pallet. If the pallet is moved any large distance, the trajectories planned for motions that are to pick up the various parts can be very bad, indeed. In principle, we could use one of the "maybe" schemes discussed before to handle such cases, but the analysis required becomes very complicated. Another choice would be to rewrite the program. In the absence of loops, it is fairly straightforward to produce a system that rewrites programs of the form:

```
if test then
        statement_1
else
        statement_2;
statement_3;
```

into

```
if test then
        begin
        statement_1;
        statement_3;
        end
else
        begin
        statement_2;
        statement_3;
        end;
```

but this gets extremely expensive. (Our heat treatment example, above, requires 2^k different paths.) Of course, it is possible to rewrite the affected statements far more locally. Indeed, this is just the sort of thing that keeping planning value sets would allow. However, as we have said, the analysis gets quite hard, and may (in some cases) be impossible. The current solution is simply to warn the user whenever an affixment fact is dropped, and then let him worry about it.

The problem would be much reduced by incorporation of trajectory planning capabilities into the runtime system. This would not entirely eliminate the need for a planning model; one would still like to preplan whenever the necessary information is available, thus avoiding runtime execution costs. Also, *other* uses of planning information may not be disposed of so easily. For instance, eventually, we hope to have collision avoidance performed by the system. The computation costs involved here are sufficiently large that planning is apt to be economically attractive for some time after the need for joint trajectories has been overcome.

5.4.5 Loops

Loops present many of the same difficulties that we encountered with conditionals. It is not always possible to predict how many times a loop body will be executed or just what the program state will be after the loop is finished. The problem is exacerbated by the fact that planning for the loop body may use facts that are invalidated later in the body. For instance, consider the program:

```
    move blue to place0;
s1: while not test do
        begin
    s2: move blue to place1;
        :
        { More code }
    s3: end;
```

The world assignment for this code is:

statement	input world	output world
s1	*w0*	*w4*
s2	*w1*	*w2*
s3	...	*w3*

The simulation proceeds by copying the input world for *s1* into *w1*, the input world for the loop body, and (possibly) adding assertions reflecting the result of *test*. At this point, the planning value of blue is *place0*; the trajectory for *s2* will be planned for a motion from *place0* to *place1*. Assume for the moment that the rest of the loop body doesn't affect blue. Then, after simulating down to *s3*, the data base would look something like this:

fact	assertion	worlds
f1	(value,blue,*place0*)	*w0,w1*
f2	(value,blue,*place1*)	*w2,w3*
f3	(value,*test*,false)	*w1,w2*

Unfortunately, there is no assurance that *test* will now be true. I.e., we may go around the loop again at execution time. On the second iteration, *s1* will ask for a motion from *place1* to *place1*; the trajectory is no longer correct.

Normal practice when modelling loop semantics would be to merge *w3* back into *w1*, and then continue the simulation of the loop; the process being terminated when some "fixed point" is reached. However, there are several difficulties in doing this for AL. Since the semantics of the loop may be sensitive to changes in the planning model, the process may be unstable. Here, the major problem is with affixments and symbolic assertions, which affect motion statements and conditional compilation. Incompatible planning values are, of course, lost on merging, as was discussed in the previous section. This means that waiting for a fixed point before planning cause the compiler to complain about lack of a start point for the motion statement at *s2*. If some scheme (such as value sets) were used to keep alternate planning values when worlds are merged, then the compiler could "do the right thing" in planning *s2*. However, keeping such sets would introduce additional problems, since there is a possibility that the set would grow each time around the loop. Thus, a method for coalescing "nearby" values would be necessary. Unfortunately, it might not be possible to tell whether a sequence of values was in fact stable.

Therefore, AL takes a cowardly approach: it warns the user whenever a fact which may be invalid is used to make a decision. Briefly, this is done as follows. When the loop body is entered by the simulation, all facts assumed to be true in the input world are noted.[14]

---

[14] This is done by copying them into a special "watch these guys" world associated with the loop.

Then, any time one of these facts is used to make a decision in the planning without having first been asserted explicitly inside the loop body, it is put into a set[15] of "guarded" facts. When the loop body processing is done, a warning is generated for any fact in the guard set that is not "true" in the output world for the loop body. In our example, *f1* is used by *s2* and is not true in *w3*, so we get the error.

When a user is warned, he has several choices. Usually, he will look at the code, mutter something unprintable about the stupidity of the system, and ignore the message.[16] This amounts to an assertion that nothing important really changed in the loop after all, so that the decisions made in planning its first pass are all ok. Where planning values are involved – by far, the most common case – this assumption is probably valid. Where affixments or other assertions differ, the user may be wrong, since AL may have become very confused.

If the user doesn't like seeing error messages, or if he wants to clarify the assumptions of his program, then he must add assertions. For instance, he might write:

```
while not test do
        begin
        blue ←← place0;
            { Says that the motion statement is to be planned from place0 }
        move blue to place1;
        :
        { More code }
        end;
```

or else

```
while not test do
        begin
    s2: move to place1;
        :
        { More code }
        plan from s2;
        end;
```

if he wants to ignore all inconsistencies. Once again, it would probably be desirable to introduce a construct like the **unexpected** declaration discussed earlier. This would enhance clarity and simplify programming, though it would not add significantly to what can be said in the language.[17]

In many cases, of course, mere assertions are not enough, and more substantial rewriting may be required to get a program to behave as intended. The obvious thing to try is to "unroll" the loop. For instance:

---

[15] Internally, another planning world

[16] In some cases, he will not even bother to look at the code, but, then, some of us like jumping out of airplanes, too.

[17] Here, we would probably want two flavors: one which said "ignore all inconsistencies", and one which said "ignore only planning value inconsistencies".

```
move blue to place0;
if not test then
      begin
      move blue to place1;
        :
      { More code }
      while not test do
            begin
            move blue to place1;
              :
            { More code }
            end;
      end;
```

In principle, the process could be continued several times.

This kind of thing is helpful in cases where the major changes (e.g., the starting position of blue) occur in the first iteration. Whether this is true depends, in part, on coding style. For instance, is a pin-in-hole primitive written as:

```
while <pin isn't in hole> do
      <put pin into hole>
```

or is the first try included separately, as was done in Section 3.4? [18] The latter style seemed likely enough so that we experimented with having the system perform one level of unrolling automatically, whenever facts were found to be in conflict. However, experience with this was not encouraging, since programs got much longer without any real gain, so the feature was dropped. The scheme might have worked better if there were a means of assessing the *significance* of particular differences in facts, and only trying to unroll when a significant difference – such as an affixment or a very large difference in planning points used for trajectories – was found. However, real success – especially with long loops – probably requires that code for differently planned iterations be "folded" together to take advantage of those parts that can be common.

5.4.6 Parallelism

Like conditionals and loops, parallelism is only handled approximately. The principal control structure for parallel execution in AL programs is the **cobegin ... coend** nest. Since AL runs on a single runtime processor, use of the construct for parallel computations is not particularly useful. The primary intent is to allow for concurrent execution of subtasks by several manipulators, with a secondary use being to allow a computation to be run "in the background" while a motion is taking place.[19] So long as the concurrent segments are

---

[18] Of course, in that case, it is possible to argue that the loop was *already* unrolled once.

[19] My personal view is that this second use is a bad idea, since it just complicates programs without any substantial speedup. In the current implementation, there isn't very much of the machine left over while servoing is going on. A more fundamental objection is that this sort of optimization should be the job of the compiler and runtime system, rather than of

independent of each other, there is no particular problem. The output world of the nest is formed by combining the changes introduced in each branch. Thus, a fact is considered to be true in the output world of the nest if and only if it is true in the output of at least one branch of the nest and is not denied by any of the other branches. For instance, suppose we have the following code:

```
    blue ←← place1; yellow ←← place2; object←←place3;
s0: cobegin { Park both arms.}
      s1: move blue to bpark;
      s2: move yellow to ypark;
    coend;
```

The world assignment is

statement	input world	output world
s0	*w0*	*w3*
s1	*w0*	*w1*
s2	*w0*	*w2*

and the data base looks something like this:

fact	assertion	worlds
f1	(value,blue,*place1*)	*w0,w2*
f2	(value,blue,bpark)	*w1,w3*
f3	(value,yellow,*place2*)	*w0,w1*
f4	(value,*object*,*place3*)	*w0,w1,w2,w3*

There are many problems with this approach. The data base does not *really* support the notion of "false". Assertions are either "true" – i.e., asserted – in a world or they are not present. On merging, denials have to be checked by looking for transitions between "true" in one world and "not true" in a successor world. An additional problem has to do with incompatible assertions. For instance,

```
cobegin
object ← place1;
object ← place2;
coend;
```

The method described above would leave both (value,*object*,*place1*) and (value,*object*,*place2*) in the data base. Since we don't know anything about the order in which the statements will be executed, there is no way of telling which planning value to believe. Since such contradictions can be very confusing, the planning model code makes a pass over the output

---

the programmer. For instance, the runtime interpreter could adopt the strategy of interlocking those parts of the data structure affected by the current motion statement, and then continuing on with the next statement, stopping when it comes to the next motion or a computation that depends on an unfinished motion. An obvious candidate for this activity would be a runtime trajectory calculator.

world after merging, to remove incompatible assertions. Of course, any attempt to identify *all* sets of incompatible assertions will be doomed. Instead, AL restricts itself to looking for only a few easily detected cases, such as incompatible values for single-valued fluent properties (e.g., planning values).

The principal mechanism in AL for synchronizing parallel control paths is the event. It is sometimes possible (for a human) to predict the order in which statements will be executed by inspection of the signals and waits in a nest. For instance, consider this program:

```
event inspection_request,inspection_done;
cobegin
   begin "inspection process"
      :
   wait inspection_request;
   { Perform inspection }
   signal inspection_done;
      :
   end;

   begin "manipulation process"
   { Manipulation A }
   signal inspection_request;
      :
   wait inspection_done;
   { Manipulation B }
   end;
coend;
```

Here, the inspection process will be performed between manipulations A and B. Eventually, we hope that AL will be able to follow simple event chains. However, the general problem is *very* difficult, and such analysis is beyond the capabilities of the current routines.

5.4.7 Complications with Motion Statements

The planning model simulation assumes that motion statements terminate "normally". The planning value of the frame being moved is set to the planning value of the destination expression. For many motion statements, this is adequate. However, as we saw in Section 3.4, it is not uncommon to write motion statements which are expected to stop short of their destinations. For instance, if we have picked up a box and want to set it on a table, the code might look something like this:

```
move box to table-vector(0,0,4)
      on force(zhat)>8*oz do
            stop;
```

The planning model will be updated to assert that *box* is at *table*-vector(*0,0,4*), even though it is expected to stop short of that position. For purposes of planning trajectories, the inaccuracies introduced in this way are generally unimportant. This is fortunate, since doing a better job automatically would require considerably better geometric models than are available with most manipulator-level AL programs.

If a user wants to keep his planning model more honest, he must use an assertion, such as *box* ←← *table*-vector(*0,0,4*). An alternative would be to introduce a new construct into AL for specifying this kind of motion. For instance:

```
move box through table to table-vector(0,0,4)
    on force(zhat)>8*oz do
        stop;
```

In addition to keeping the model more accurate, such a construct would make programs clearer to read and might make slightly more efficient motions possible.

"Oh, de hip bone connected to de thigh bone."

Traditional

Chapter 6.

Object models

6.1 Introduction

The AL compiler can get by without making use of shape information. The only "object models" it really understands are affixment declarations between object frames; and, here, it doesn't really care what the frames correspond to. On the other hand, to generate a manipulator program, we must make explicit use of much more information about the objects being manipulated. In writing the pin-in-hole example of Section 3.4, for instance, we used shape information in answering the following questions:

1. Where can the pin be grasped?
2. How far must the hand be opened before moving to the grasping position?
3. When we close the hand, what is the minimum acceptable hand opening before we assume that we have missed the pin?
4. For a given misalignment between the pin and hole axes, how far will the pin make it into the hole before sticking on something?
5. How far is the pin *supposed* to go into the hole?
6. If "tapping" is necessary, is there a good spot on the object to use?

If manipulator coding is to be automated, then ways must be found for the computer to answer these questions, based on its own model of the objects involved. Furthermore, if our automatic coding system is ever to be useful, construction of the necessary object models must be made easier (for the user) than writing the manipulation program. Ideally, we could use the output of a Computer-Aided-Design (CAD) program. In this case, the user would merely have to specify a task-level description specifying the assembly sequence, and the computer would do the rest.[1] Unfortunately, good mechanisms either for description of

---

[1] Indeed, the object design could be used to provide important contextual information clarifying the task specification. For instance, "Put the cover on the box" could be used, rather than "Put the cover on the box so that the bottom surface of the cover is against the top surface of the box, and the axes of cover holes 1 and 2 line up with the axes of box holes 1 and 2."

object shapes or for doing many of the computations we need haven't been developed yet.[2]

In this work, we wish to concentrate on the *use* of object information, rather than on provision of elegant descriptive formalisms. Consequently we have adopted a somewhat *ad hoc* representation, based on "attribute graphs", which will allow us to represent directly any properties that are awkward to compute from existing "pure" shape representations.

6.2 Basic Constructs

As was stated in the previous section, we represent objects by means of a graph structure, which is represented internally by a combination of LEAP [35,36,107] associations and record structures. The nodes in this graph correspond to parts and features of objects, and the links describe relations between the nodes. Additional nodes and links are used to represent important attributes of the objects, features, and links.

Within this general framework, there is a great deal of flexibility in how particular kinds of information are to be represented. We have adopted the following conventions:

Shape Information is represented by the use of different node types and by the value of parameters in each node's associated record.

Structural Information is represented by link types like **subpart, feature, inside,** etc.

Location Information is represented as a property of the corresponding structural links.

For instance, the representation of the box whose assembly was described in Section 2.2.1 would include the following elements:

subpart⊗*box_assembly*▪*box_body*[3]
subpart⊗*box_assembly*▪*cover_plate*

subpart⊗*box_body*▪*bore_1*
subpart⊗*box_body*▪*bore_2*

---

[2] Braid [26] has written an excellent survey of the current "state of the art" for shape representations. This paper describes six recent systems for computer representation of shape information.. Other relevant work, not mentioned in the paper, includes the "generalized cone" representation developed by Binford, and subsequently elaborated by Agin[1], Nevatia [73] and Miyamoto [18]; the "procedural" representations explored by Grossman[46]; and the parts graphs developed by Lieberman and Wesley [63]. Each of these systems has some advantages and disadvantages; none are yet sufficiently well developed to serve as our hypothetical CAD system.

[3] In English, this says "A subpart of *box_assembly* is *box_body*." Note that these "associations" may themselves be treated as elements in other associations. This facility allows us to represent properties of relations as well as properties of objects.

feature⊗*box_body*▪*hole_1*
feature⊗*box_body*▪*hole_2*
feature⊗*box_body*▪*top_surface*
feature⊗*box_body*▪*side_1*

lies_in⊗*top_surface*▪*hole_1*
lies_in⊗*top_surface*▪*hole_2*
inside⊗*hole_1*▪*bore_1*

nomxf⊗[subpart⊗*box_assembly*▪*box_body*]▪niltrans
nomxf⊗[subpart⊗*box_assembly*▪*cover_plate*]▪trans(nilrotn,vector(*0,0,4.9*))
current_xf⊗[subpart⊗*box_assembly*▪*cover_plate*]▪<fluent *232605*>

This structure is illustrated in Figure 6.1. Appendix D contains a printout of the complete model for the box assembly. Subsequent sections will describe the more common node and link types in more detail.

6.3 Object Nodes

The most important nodes in this graph structure are those used to represent objects or parts of objects. Contained within each object node is the following information:

kind – Currently, there are three "kinds" of object:

assembly – a collection of separable subparts. E.g., the box assembly consists of a bottom, a cover, and several screws.

part – a single lump of metal. E.g., the box bottom. A part may be primitive, in which case its description will give its shape, or it may itself be composed of subparts.

bore – a negative part. For instance, the hollow space inside the box.

description – a pointer to an appropriate shape description for the part. Typical shape descriptions are rpp, sphere, cylinder, and profile. rpp(*x,y,z*) is just a rectangular parallelopiped with sides (*x,y,z*). sphere(*r*) is likewise self-explanatory. cylinders and profiles are explained below.

Cylinders

A cylinder is the solid you get by sweeping a cross section along an axis.[4] By convention, we use the z-axis as the "long axis" of the cylinder. Relevant parameters are:

cross_section – The plane figure that is to be swept along the axis. Typically, either a circle or a polygon.

---

[4] C.f., the "generalized cones" of Agin and Binford.

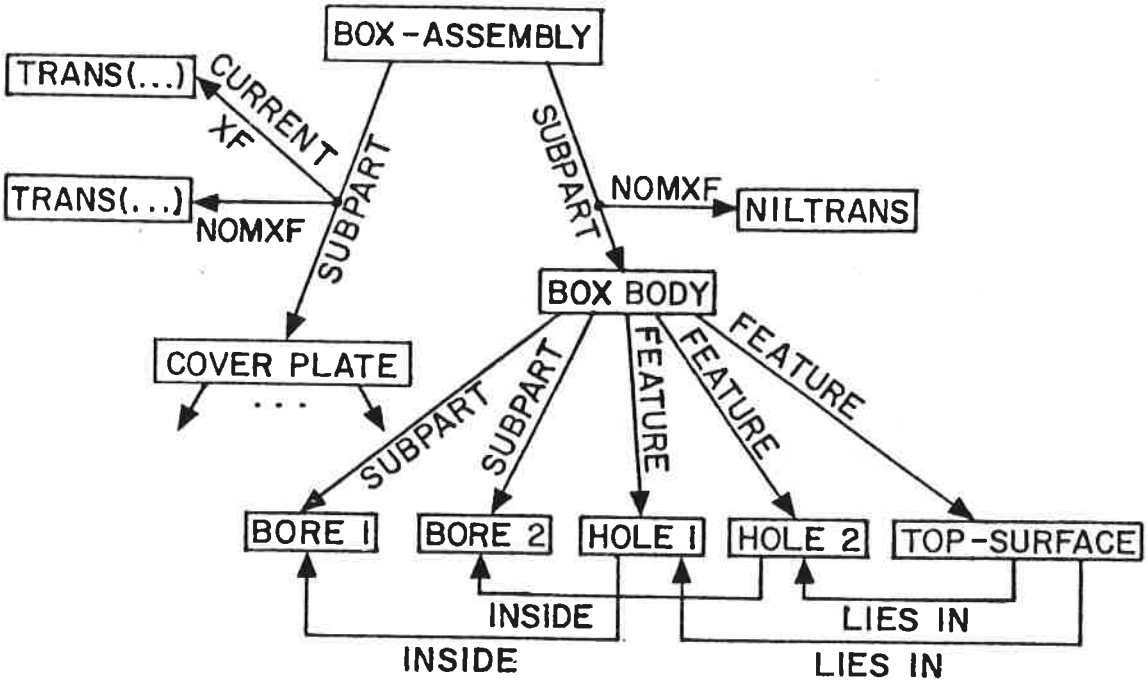


Figure 6.1. Box Assembly Relations

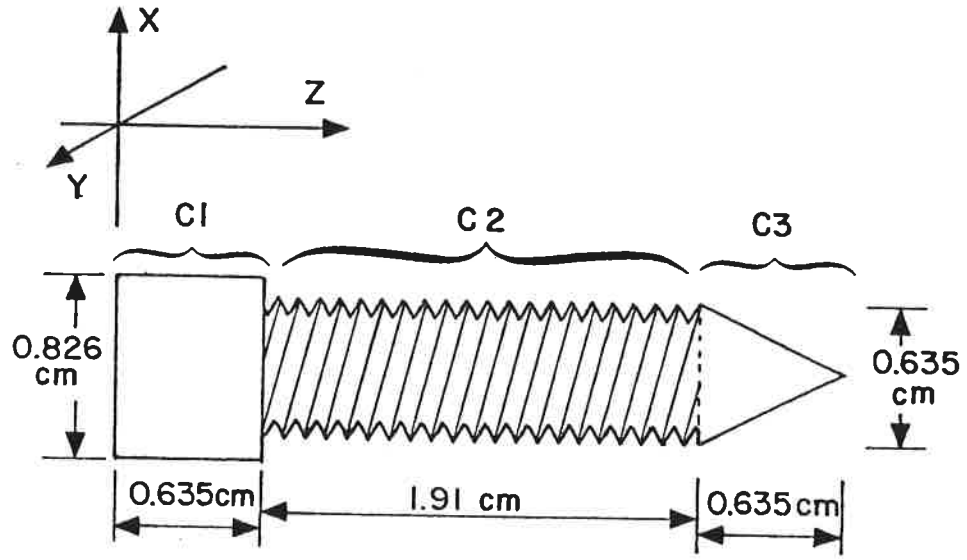


Figure 6.2. Representation of a Screw

length – The significance of this parameter will be left as an exercise for the reader.

top_diameter, bottom_diameter – Tell the size of the cross section at each end.[5]

finish – Many objects have surface properties – such as threads – that are awkward to represent directly as part of the shape representation. Instead, this information is represented symbolically.

Profiles

Many objects, such as screws, are most conveniently modelled as a stack of cylinders. Rather than use separate subpart nodes for each cylinder, we combine them into a single array. Important parameters are:

length – overall length of the stack.

n_sects – the number of sections in the stack.

section[*1*:n_sects] – array of cylinders. **section[*i*]** is the *i*'th cylinder in the stack.

sect_dir[*1*:n_sects] – If the z-axis of **section[*i*]** is parallel to the z-axis of the stack then **sect_dir[*i*]** = 1. Otherwise, **sect_dir[*i*]** = -1.

The representation of a screw by a profile is illustrated in Figure 6.2. The screw is composed of three cylinders, all oriented parallel to the screw axis. Hence, **sect_dir[*i*]** = *1* for all $1 \le i \le 3$. The data for each section cylinder is shown below:

section	cross-section	top diam.	bottom diam.	length	finish
1	circle	*.826* cm	*.826* cm	*.635* cm	smooth
2	circle	*.635* cm	*.635* cm	*1.91* cm	*tap 28*
3	circle	*.635* cm	*.000* cm	*.635* cm	smooth

6.4 Object Links and Link Attributes

The principal link between objects is the subpart link:

subpart⊛*box_assembly*▫*cover_plate*

which asserts that *cover_plate* is a subpart of *box_assembly*. This link type suffices to describe the topological structure of objects. However, in addition to knowing structure, we

---

[5] Actually, these numbers are *scale factors*. However, the only *circular* cross section used has a diameter of 1 cm, so the "usual" interpretation is also correct.

must also know the relative location of subparts. To maintain this information, we make use of LEAP's "bracketed triple" facility, which allows us to associate additional data with any link in the graph structure.

nomxf⊗[subpart⊗*box_assembly*▪*cover_plate*]▪trans(nilrotn,vector(*0,0,4.9*))

This states that, when the cover is in its "nominal" position on the box,

<location of *cover_plate*>▪<location of *box_assembly*>⊗**trans(nilrotn,vector(*0,0,4.9*))**

Of course, the cover might not always be in its nominal position. Its actual relation to the box bottom is a *fluent* property, dependent on situational information. This is reflected by link properties like:

current_xf⊗[subpart⊗*box_assembly*▪*cover_plate*]▪<fluent *232605*>

Here, **<fluent *232605*>** is a reference into the data base used to hold situational information. Essentially, it consists of a pattern template like:

(value_of_fluent_*232605*, ? *val*)

which can be used to retrieve values from the desired "worlds". Typically, the value stored will either be a **trans** or else a parameterized estimate of the kind discussed in Chapter 7.

In manipulating objects, we must often consider manufacturing tolerances and similar causes of variation in part-subpart relations. These are reflected by the use of link attributes:

determ_template⊗[subpart⊗*box_body*▪*bore_1*]▪<paramterized estimate>
determ_estimate⊗[subpart⊗*box_body*▪*bore_1*]▪<fluent *123456*>

The **determ_template** attribute of a link gives *a priori* limits on the accuracy of the location attributes associated with the link. Typically, this means the accuracy of the manufacturing processes involved. The properties of the <parameterized estimate>s used to represent this information will be discussed extensively in Chapter 7. Here, we might add that, if the errors are known to be negligible, we use a special value, **fully_determined**, for this attribute. Since it is possible to make measurements at execution time, the actual "degree of determination" of the link will vary; **determ_estimate** gives the corresponding situational fluent.

In addition to subpart links, we must frequently describe other relations between objects. For instance, although the cover plate is a subpart of the box assembly, it may initially be placed in a parts dispenser. The representation of such situations is greatly simplified by the use of another link type, **related**.[6]

---

[6] In the actual implementation, we use related links as the basis for *all* location attributes. Thus, in addition to **subpart⊗*box_assembly*▪*cover***, we will have a link **related⊗*box_assembly*▪*cover***, and link attributes like **current_xf⊗[related⊗*box_assembly*▪*cover*]▪<fluent *23605*>**. This hack simplifies programming but complicates explanation. Hence, the small deception practiced earlier.

related⊗*cover_dispenser*▪*cover*
current_xf⊗[related⊗*cover_dispenser*▪*cover*]▪<fluent 76672>
determ_estimate⊗[related⊗*cover_dispenser*▪*cover*]▪<fluent 76464>

Other link attributes are used to associate programming information with the object models. The most important of these attributes is **xfvar**, which connects an AL **trans** variable with a **subpart** or **related** link. For instance,

xfvar⊗[subpart⊗*box_body*▪*bore_1*]▪*bore_1_xf_var*
xfvar⊗[related⊗workstation▪*cover*]▪*cover_frame_var*

assert that the locations of *bore_1* and *cover* are given by

bore_1 = *box_body***bore_1_xf_var*
cover = workstation**cover_frame_var* = *cover_frame_var*

For simplicity of discussion, we will generally ignore this distinction, and use the object names to refer both to the objects and to the **frame** variables relating them to the work station.

6.5 Feature Nodes

One property shared by all the object nodes discussed in Section 6.3 is that they all have volume.[7] There are many other attributes, such as holes, surfaces, edges, and the like, which lack this property. We call these entities "features", and relate them to their parent objects by **feature** links.

feature⊗*box_body*▪*hole_1*
feature⊗*box_body*▪*top_surface*
feature⊗*top_surface*▪*hole_1*[8]

Locations and other link attributes are the same as for object nodes. For instance:

nomxf⊗[feature⊗*box_body*▪*top_surface*]▪trans(niltrans,vector(*0,0,4.9*))
determ_template⊗[feature⊗*top_surface*▪*hole_1*]▪<parameterized estimate>

An important point about features is that they provide a symbolic or quasi-symbolic means for referring to places on objects. This is very important, since it allows us to describe relations between objects in other than purely numerical terms. For instance, we can say "the bottom surface of the box is resting on the top surface of the work table" or "Align *vertex_1* of *cover* with *vertex_1* of *top_surface*." Chapter 7 will develop techniques for translating such descriptive statements into more explicit mathematical forms which can be used by the system to make coding decisions.

The characteristics of the most important classes of features are discussed below.

---

[7] Although, in some cases, negative volume.

[8] Note that features can have sub-features and that the same feature can "belong" to several parents. The only restriction is that a feature only belong to one object node.

Surfaces

Currently, only planar surfaces are represented. There are two flavors:

planar_circle(*r*) – a circle of radius r in the xy plane, centered at (0,0).

planar_polygon(*n*, *x*[*1:n*], y[*1:n*]) – a polygon in the xy plane, with vertices at (*x*[*i*],*y*[*i*]).

These structures are also used as **cross_section** descriptions for **cylinders**.

Edges and Vertices

Edges and vertices are primarily important in describing object locations from feature-to-feature contacts. To simplify certain coding problems associated with the methods discussed in Chapter 7, we represent them as zero diameter cylinders and spheres, respectively. An additional parameter, θ is used to describe the angle between the planes adjacent to the edge, as is shown in Figure 6.3.

Holes

We make a distinction between *bores*, which are the negative volumes left by metal removal operations, and *holes* which are the features formed by intersection of bores with surfaces. The reason for this semantic quibble is that it allows us to establish some useful conventions. Holes are represented as zero-length cylinders. The origin of their associated coordinate system is assumed to be "centered" in the hole, with the z-axis parallel to the outward facing normal of the surface in which the hole lies, as shown in Figure 6.4. Holes are related to their corresponding bores by use of the **inside** relation:

inside⊗*hole_1*▪*bore_1*

Generally, we will not try to preserve the distinction between holes and bores in our discussion, except in those few cases where it is important.

6.6 Other Properties

The principal advantage of the modelling scheme described here is its flexibility. It is very easy to add additional data to the model or to change the way a particular fact is represented. This section presents some additional examples.

Redundant Shape Descriptions

crude_shape⊗*obj_1*▪*obj_2* – asserts that *obj_2* has approximately the same shape as *obj_1*. For instance, **crude_shape**⊗*box_assembly*▪**rpp**(*7.9,6.4,5.0*) provides an approximate shape for *box_assembly*. A typical use for this kind of information would be in a collision avoider, which doesn't need (or want) detailed shape information.

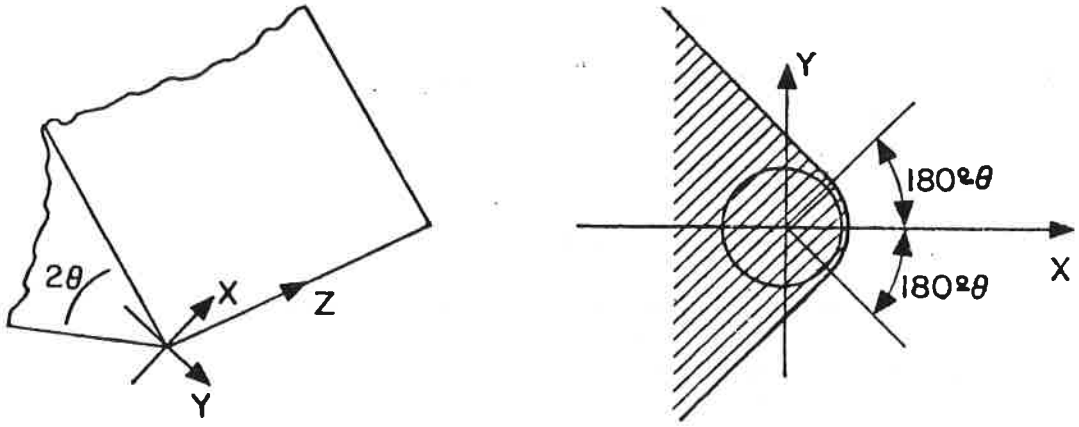


Figure 6.3. Coordinate System of an Edge

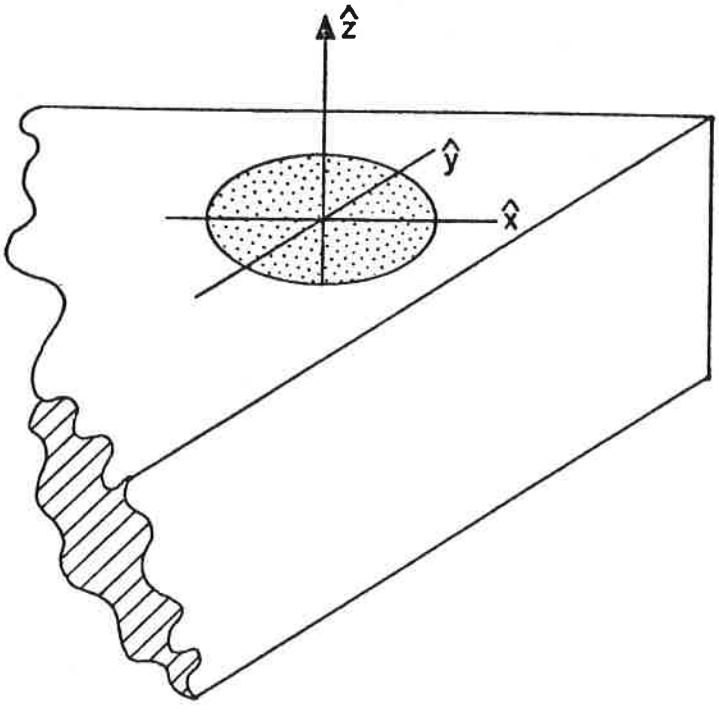


Figure 6.4. Coordinate System of a Hole

geomed⊗*box_body*■"*box.b3d*" – specifies the name of a file giving a GEOMED representation of *box_body*. GEOMED [9] is a system developed by Baumgart which uses polyhedral models to generate efficient graphic representations of predicted scenes. The representation is optimized for such applications, and is not particularly well suited to many of the things we need to do. The point, here, is that we can incorporate several redundant representations without much trouble.

Symmetries

symmetry_rot⊗*obj*■rot(a,γ) – states that if **rot(a,γ)** is applied to *obj*, the result will be indistinguishable from *obj*.

symmetry_axis⊗*obj*■*v* – states that rot(v,ω) is a symmetry rotation of *obj* for any value of ω. For instance, if we have a circular cylinder, the z-axis will be a symmetry axis.

Grasping Positions

grasp_method⊗*obj*■*meth* – asserts that *meth* is a prototype "grasping strategy" for *obj*. The contents of such grasping strategies are discussed in Chapter 8. There, we will compute the necessary information (finger spread, grasping orientation, etc.) directly from the geometric information describing the pin. In other cases, where the shape information is more complicated, or where the user has some reason to demand a particular grasping position, we might have to resort to prespecified methods.

Chapter 7.

Representation of Location and Accuracy Information

7.1 Introductory Remarks

Since programs are *prior* specifications of actions to be taken at a later time, coding decisions must rely on our *expectations* about relevant attributes of the execution-time environment. Some of these attributes, such as object design, remain essentially static during execution of the program. Others, however, are subject to variation. As we saw in Section 4.5, the principal situation-dependent properties for manipulator programming are:

1. Where objects are in the work station.
2. How objects are attached to each other.
3. How accurately the locations and attachments will be known at execution time.

Since AL programs use **frame** variables to represent object locations, the planning values and affixment assertions maintained by the AL compiler are obvious candidates for use in our automatic coding procedures. However, there are several problems with doing so directly.

First, planning values do not tell us anything about the accuracy of our runtime knowledge. As we have seen, such information is extremely important in making coding decisions, and cannot be ignored, even where the possible perturbations in object locations are small enough to ignored in deciding the gross motions required to perform the task.

Second, in writing programs, a *single* planning value is often insufficient.[1] In picking a grasping position for our pin-in-hole task, for instance, we must be sure that the hand will not be required to move to an impossible position. If the hole's orientation can vary considerably, then certain grasping orientations may be ruled out. A similar situation is illustrated in Figure 7.1, where we are considering how to pick up a small carburetor from a pallet, which, in turn, sits on a table top. Assume, for the moment, that we have narrowed the choice to two candidates, "from the top" and "from the side", as shown in the figure. Since the pallet's rotation about the table z axis is unrestricted, the "from the side" grasping position could require impossible arm positions and probably should be excluded.

---

[1] As we saw in Chapter 5, single values cause problems for AL, too. The assumption made by the trajectory calculator is that destination points will not diverge too grossly from the planned values. It is up to the user to make sure that this assumption is not violated. In cases where variations will be greater, then he must insert explicit checks to select alternative trajectories. Similarly, if the compiler gets confused, then it asks the user for clarifying assertions. The point, here, is that if the computer is take over more of the coding, then it must be able to do more of the modelling work for itself.

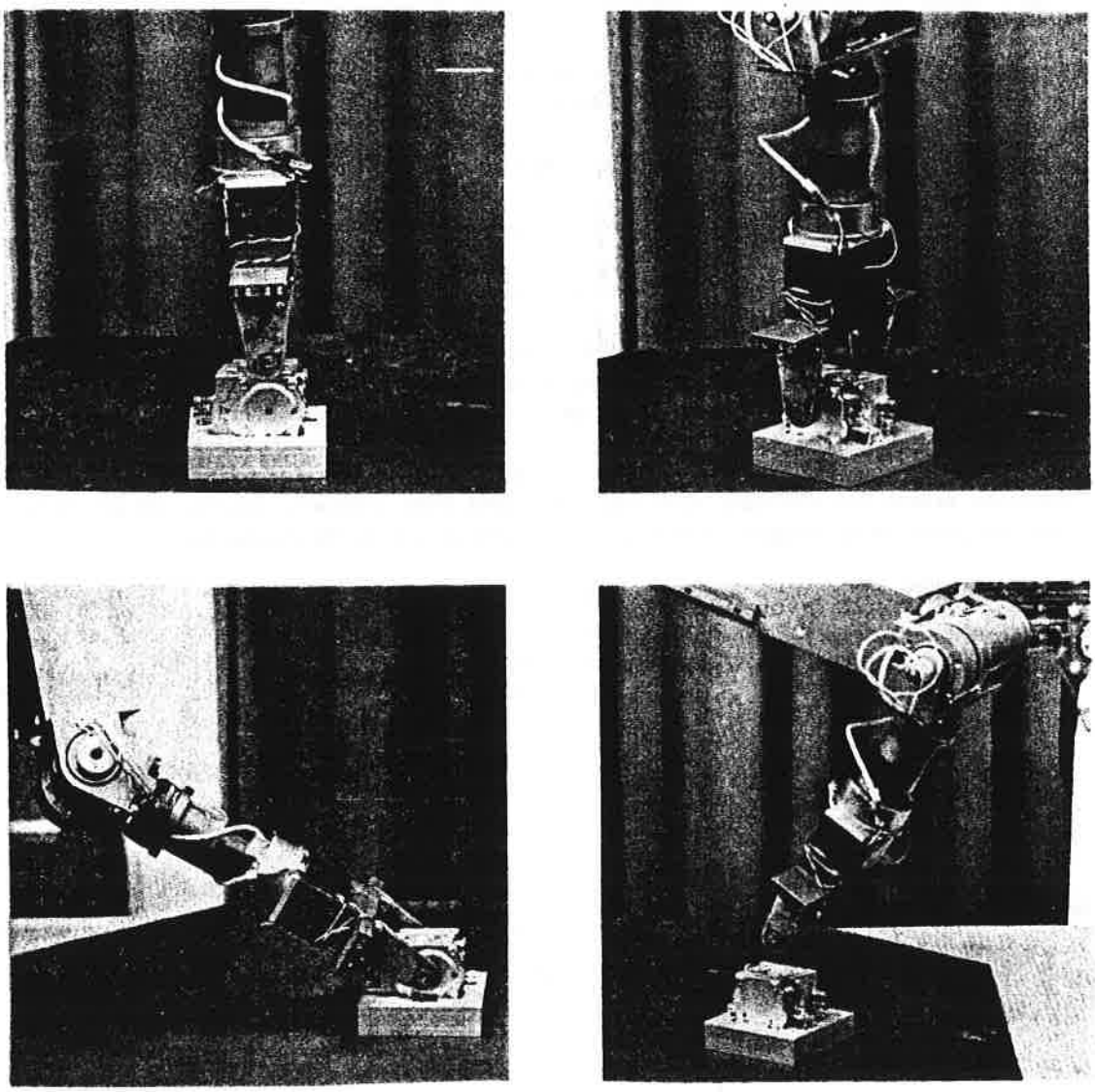


Figure 7.1. Picking Up a Carburetor

Finally, numerical planning values are often very difficult for users to specify, since humans don't come equipped with accurate surveying instruments and usually don't have particularly good intuition in mapping object locations into frame values. It is often much more natural, in task-level specifications, to describe object locations by assertions about the relationship between features. For instance,

> The bottom of the pallet is flush against the top of the work table.
>
> Pin 1 is inserted 0.45 inches into hole 1.[2]
>
> The valve body is sitting on its left side in the box.

Even in cases where nominal values are obtained by direct measurement, assertions are particularly important in distinguishing what aspects of the object locations are mere accidents of the particular test example, and what always will be true at execution time.

This chapter develops mathematical techniques for representing object positions and position errors, for deriving such representations from assertions relating object features, and for performing computations that can be used as a basis for coding decisions.

The approach taken is to translate symbolic assertions about inter-object relationships into constraint equations involving the location variables of the objects and free scalar variables representing degrees of freedom in the relationship. These equations are then further reduced to eliminate redundant degrees of freedom, so that we are left with a parameterized trans expression,

$$T(\lambda_1,\ldots,\lambda_n)$$

in which the remaining free variables, $(\lambda_1,\ldots,\lambda_n)$, are (1) readily related to translations and rotations in some understood frame of reference, and (2) linked by constraints whose form is sufficiently simple to be of some use by the system. Depending on the exact interpretation placed on the λ_k and the constraint equations used, $T_{AB}(\bar{\lambda})$ may be used to make predictions about the relative position of two objects, A and B, or the accuracy with which the relation can be determined at execution time.

The principal numerical technique used is linear programming, which is employed to compute ranges on parameters, maximum displacement along given directions, and other similar data. As we shall see, this works best in those cases where rotations are small enough so that a single linear approximation may be used. Consequently, the methods work better for estimating errors than for predicting large ranges of positions. Fortunately, there is a large class of assembly tasks in which the approximate position of objects is constant, or, at least, limited to one rotational degree of freedom.[3]

---

[2] Note the similarity to the task-level command, "Insert pin 1 0.45 inches into hole 1."

[3] For instance, a part which sits on a table in a known stable orientation is free to rotate only about an axis perpendicular to the tabletop.

7.2 Contact constraints

One common way to restrict an object's freedom of motion is to place some part of it in contact with another object. For instance, suppose we know that a round peg P fits snugly into a hole H. This gives us the relation:

$$P = H*FRAME(ROT(\hat{z},\eta),VECTOR(0,0,\zeta))$$

where

ζ = a free scalar variable representing the distance that P penetrates into H.

η = a free scalar variable representing the rotation of the peg about its axis.

Similarly, suppose we know that plane surface S_1 of object A rests against plane surface S_2 of object B. These planes are defined with respect to their corresponding body coordinate systems by the relations:

S_1 = all x such that $a_1 \cdot x = d_1$

S_2 = all x such that $a_2 \cdot x = d_2$

where a_1 and a_2 are the outward facing normals of S_1 and S_2, respectively. Then A and B will be related by the constraints:

$$A = B*transl(VECTOR(x,y,z))*ROT(a_2,\omega)*R_{12}$$

where

transl(VECTOR(x,y,z)) = TRANS(VECTOR(x,y,z),NILROTN)

R_{12} = a constant rotation such that $R_{12}*a_1 = -a_2$

$a_2 \cdot (x,y,z) = d_1+d_2$

$0 \le \omega \le 2\pi$

In general, there may be a number of such relational assertions with the form "Feature X of object A is in contact with feature Y of object B". The three most common forms of contact relation are "fits in", which usually applies to shafts fitting into holes, "fits over", which is just the other way around, and "up against", which applies to features like points, lines, plane surfaces, (circular) cylindrical surfaces, and spherical surfaces. Thus, some typical contact relations that the system may have to "understand" include:

1. the side of a cylinder resting against a plane surface.
2. a cylinder inserted snugly into a hole.
3. a spherical knob resting against a surface.
4. a point on one object touching a plane surface.

We wish to model how such contact relations affect the linkage between the location variables of the objects. In general, this relation is given by an expression of the form

$$B = A * T_{AB}(\lambda_1, \ldots, \lambda_m)$$

where T_{AB} is a trans linking A and B, and $(\lambda_1,\ldots,\lambda_m)$ are free scalar variables reflecting the degrees of freedom between A and B.

The method we use to derive T_{AB} from feature contact assertions makes use of the fact that if a feature has location f_B (in the coordinate system of object B), then the feature's location in the coordinate system of A will be given by $T_{AB} * f_B$. It is convenient to break T_{AB} down into its rotational and translational components,

$$T_{AB} = \text{transl}(p_{AB}) * R_{AB}$$

Now, given some feature of object A in contact with a feature of object B, we represent the set of points for the feature of B in terms of a parameterized "generator" expression

$$F_B = \{ v \mid v = f_B(\mu_1,\ldots,\mu_n) \}$$

where the μ_i are free scalars (see the table, below). Suppose, now, that we have a constraint that every point on the feature of A (expressed in the coordinate system of A) must obey.

$$G_A(v) = 0 \qquad \text{for all } v \text{ on } F_A$$

Then, if every point of F_B is to lie on F_A we get

$$G_A(T_{AB} * f_B(\mu_1,\ldots,\mu_n)) = 0$$

for all μ_i. We can then substitute values of μ_i in order to obtain constraints on T_{AB}.

The expressions for points, lines, and planes are:

Feature type	Generator expression	Constraint Expression
point	f_0	$v - f_0 = 0$
line	$\lambda \hat{f}_1 + f_0$	$(v-f_0)\cdot\hat{f}_2 = 0$
		$(v-f_0)\cdot\hat{f}_3 = 0$
plane	$\mu \hat{f}_2 + \lambda \hat{f}_1 + f_0$	$(v-f_0)\cdot\hat{f}_3 = 0$

Here, $\hat{f}_1$, $\hat{f}_2$, and $\hat{f}_3$ are orthonormal vectors. Together with a displacement, f_0, they define a coordinate system for the features. Conventions used in defining these vectors are shown in Figure 7.2 (assuming the mapping $\hat{x} = \hat{f}_1$, $\hat{y} = \hat{f}_2$, $\hat{z} = \hat{f}_3$).

For instance, if we know that plane $[\mu\hat{f}_2+\lambda\hat{f}_1+f_0]$ of object B lies against a plane of object A constrained by $\hat{g}\cdot(v-g_0)=0$, we get

$$\hat{g}\cdot(p+R(\mu\hat{f}_2+\lambda\hat{f}_1+f_0)-g_0) = 0$$

where

p = displacement between A & B
R = Rotation between A & B

which reduces to

$$\hat{g}\cdot R(\mu\hat{f}_2+\lambda\hat{f}_1+f_0) = \hat{g}\cdot g_0 - \hat{g}\cdot p$$
$$\hat{g}\cdot R*(\mu f_2+\lambda\hat{f}_1+f_0) = d - \hat{g}\cdot p$$

where

$$d = \hat{g}\cdot g_0$$

This gives

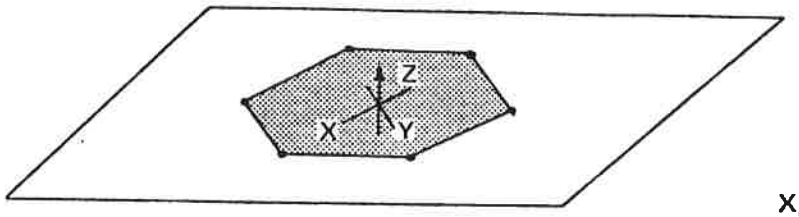
$$\hat{g}\cdot R f_0 = d - g\cdot p$$
$$\hat{g}\cdot R\hat{f}_1 = 0$$
$$\hat{g}\cdot R\hat{f}_2 = 0$$

The two "=0" constraints tell us that $R^{-1}\cdot\hat{g}$ is perpendicular to $\hat{f}_1$ & $\hat{f}_2$. (I.e. $R^{-1}\cdot\hat{g} = \pm\hat{f}_3$, where $\hat{f}_3=\hat{f}_1\times\hat{f}_2$). Thus, $R = R_\gamma*ROT(\hat{f}_3,\phi)$, where R_γ is a constant rotation such that $R_\gamma*\hat{f}_3 = \pm\hat{g}$. The ambiguity here comes from the fact that there are two possible relative orientations for A and B that will cause one plane to coincide exactly with the other. This ambiguity may be resolved by insisting that no part of A be inside of B. If we assume that $\hat{f}_3$ and $\hat{g}$ are the outward facing normals to their respective planes, this gives us a constraint of the form

$$\hat{g}\cdot R\hat{f}_3 < 0$$

so $R_\gamma\hat{f}_3=-\hat{g}$. Also, f_0 can expressed uniquely as a linear combination of $\hat{f}_1$, $\hat{f}_2$, and $\hat{f}_3$:

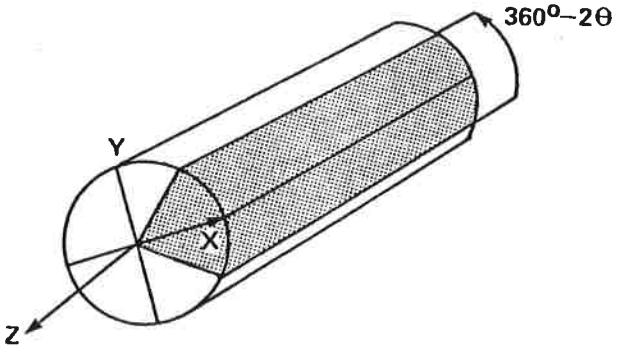
COORDINATES OF A SURFACE REGION



Z = OUTWARD FACING NORMAL

X,Y = ARBITRARY

COORDINATES OF A CYLINDER SECTOR

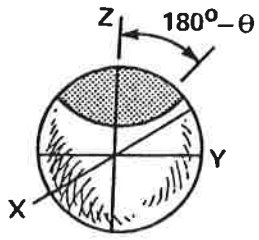


Z = AXIS

X = CENTER OF SECTOR

Y = Z × X

COORDINATES OF SPHERE SEGMENT



Z = CENTER POINT OF SECTOR

Figure 7.2. Feature Coordinate System Conventions

$$f_0 = \nu\hat{f}_3 + \mu f_2 + \lambda f_1$$

for some ν, μ, λ. Substituting this decomposition for f_0 gives us

$$\begin{aligned}\hat{g}\cdot Rf_0 &= \hat{g}\cdot R(\nu\hat{f}_3 + \mu\hat{f}2 + \lambda\hat{f}1)\\ &= \nu\hat{g}\cdot R\hat{f}_3 + \mu\hat{g}\cdot Rf_2 + \lambda\hat{g}\cdot R\hat{f}_1\\ &= \nu\hat{g}\cdot R_\gamma ROT(\hat{f}_3,\phi)\hat{f}_3 + 0 + 0\\ &= \nu\hat{g}\cdot R_\gamma\hat{f}_3 = -\nu\,\hat{g}\cdot\hat{g} = -\nu\end{aligned}$$

Rearranging our earlier equation for $\hat{g}\cdot Rf_0$ and then substituting this new value, we get.

$$\begin{aligned}\hat{g}\cdot Rf_0 &= d - g\cdot p\\ \hat{g}\cdot p &= d - \hat{g}\cdot Rf_0\\ &= d + \nu\end{aligned}$$

Thus

$$T_{AB} = transl(p)*R_\gamma*ROT(\hat{f}_3,\phi) \qquad \text{[Eq 1]}$$

where

R_γ = constant rotation such that $R_\gamma\hat{f}_3 = -\hat{g}$

$g\cdot p = d + \nu$

which is equivalent to what we got in Section 7.2.

For another example, suppose that we have a circular cylinder B of diameter r and axis $\lambda\hat{f}_1 + f_0$ resting against a planar surface $\hat{g}\cdot v = d$ of A. Then,

$$\begin{aligned}\hat{g}\cdot Rf_0 &= d + r - \hat{g}\cdot p\\ \hat{g}\cdot R\hat{f}_1 &= 0\end{aligned}$$

This reduces to

$$R = ROT(\hat{g},\phi)*R_\alpha*ROT(\hat{f}_1,\omega) \qquad \text{[Eq 2]}$$

where

R_α = a constant rotation such that $R_\alpha\hat{f}_1$ is perpendicular to $\hat{g}$.

Now, suppose that we also know that some point f_0' on B is resting against the same surface. This gives

$$\hat{g}\cdot Rf_0' = d - \hat{g}\cdot p$$

Combining this with the earlier equations gives

$$\hat{g}\cdot R\Delta f = r$$

where $\Delta f = f_0 - f_0'$. Substituting the value for R given by [Eq 2], we get

$$\hat{g}\cdot ROT(\hat{g},\phi)R_\alpha ROT(f_1,\omega)*\Delta f = r$$

$$(ROT(\hat{g},-\phi)*\hat{g})\cdot ROT(R_\alpha f_1,\omega)*(R_\alpha \Delta f) = r$$

$$\hat{g}\cdot ROT(R_\alpha f_1,\omega)*(R_\alpha \Delta f) = r$$

which may be solved to give us zero, one, or two values for ω. Once again, the ambiguity may be resolved (sometimes) by checking to see which value for ω requires some part of object A to penetrate object B. Once a value for ω has been found, we can use it to produce a "one degree of freedom" value for R.

$$R = ROT(\hat{g},\phi)*R_\gamma$$

where

$$R_\gamma = R_\alpha * ROT(f_1,\omega)$$

This value can then be used to produce a fixed value for $\hat{g}\cdot f_0$ and, hence, for $\hat{g}\cdot p$, giving us the same form of constraints as we obtained for plane to plane mating. (This is not surprising, since a line and a point not on the line determine a plane.)[4]

7.3 Inequality Constraints

Not all constraints on objects are characterized by a snug fit or by hard contact between features. One simple constraint of this type, which was used in the previous section, is that solid objects may not interpenetrate. Other typical "loose" relations include:

1. A peg of diameter d fitting into a hole of diameter $d+\epsilon$.
2. An object sitting in a box.
3. A feature on object A in contact with a region of a feature of object B.

---

[4] Ambler and Popplestone [2] have constructed a system using many of the same principles described here: They translate contact relations between object features into constraint equations on the objects' location variables. These equations are then reduced symbolically, with remaining freedoms being expressed by trigonometric expressions. They do not, however, handle non-contact relations or compute limits on free variables.

Such relations are reflected by inequality constraints on the degrees of freedom. For instance, suppose we insert a peg of diameter r_p into a hole of diameter r_h, where $r_h > r_p$. Then,

$$P = H*transl(VECTOR(x,y,z))*ROT(\hat{z},\theta)$$

where

$x^2+y^2 \leq (r_h-r_p)^2$

$0 \leq z$ – since we know that the peg is in the hole.

$0 \leq \theta \leq 2\pi$ – i.e. no constraints on θ.

In other words, the peg can rattle around within the constraints imposed by the sides of the hole. Actually, the above equation for P is not quite right, since it assumes that P is still held coaxially to H, whereas it actually can wobble around a bit, producing a much messier expression:

$$P = H*transl(VECTOR(x,y,z))*ROT(\hat{z},\omega)*ROT(\hat{x},\phi)*ROT(\hat{y},\theta)$$

with very complicated constraints on the free variables.

The formalism for handling inequality constraints is essentially the same as that developed in the previous section for handling the contact constraints. (Indeed, the inequalities frequently arise as a refinement to a contact constraint. For instance, a planar face of one object may rest against a region of the planar face of another object.) Again, suppose we have a feature F_A of object A given by

$$F_A = \{ v \mid v = F_A(\mu_1,...,\mu_n) \}$$

subject to

$$c_1(\mu_1,...,\mu_n) \geq 0$$
$$c_2(\mu_1,...,\mu_n) \geq 0$$
$$\vdots$$
$$c_m(\mu_1,...,\mu_n) \geq 0$$

We wish to constrain this F_A to lie in some region of another object B, restricted by

$$F_B = \{ u \mid G_e(u) = 0 \text{ and } G_n(u) \geq 0 \}$$

Substituting, we get

$$G_e(T_{AB}*F_A(\mu_1,...,\mu_n)) = 0$$
$$G_n(T_{AB}*F_A(\mu_1,...,\mu_n)) \geq 0$$
$$C(\mu_1,...,\mu_n) \geq 0$$

For example suppose that we have a face of some object B given by all points v such that

$\hat{g}\cdot v = d$ (saying it is on the plane)
$Gv \geq b$ (defining the extent of the face)

Then, if we know that some point f_0 of object A lies on the face, we get

$$\hat{g}\cdot Rf_0 = d - \hat{g}\cdot p$$
$$G*Rf_0 + Gp \geq b$$

where R and p give the relative rotation and translation, respectively, between the two objects. Similarly, suppose that we have a region on a planar face of A asserted to be up against the face of B. Then, we get

$$\hat{g}\cdot R(\mu\hat{f}_2+\lambda\hat{f}_1+f_0) = d - \hat{g}\cdot p$$
$$G*R(\mu\hat{f}_2+\lambda\hat{f}_1+f_0) + Gp \geq b$$
$$C(\mu,\lambda) \geq d$$

where the last constraint limits the region of the face of A that must match with the face of B. Note that the possible values for μ and λ are determined only by the constraint $C(\mu,\lambda) \geq d$. If the set of feasible solutions, (μ,λ) to this constraint is convex, then we can use the extreme points (μ_i,λ_i) to produce a new set of constraints on R and p.

$$\hat{g}\cdot p = d + |f_0|$$
$$R = ROT(\hat{g},\phi)*R_\gamma$$
$$G*R(\mu_1\hat{f}_2+\lambda_1\hat{f}_1) \geq b'_1$$
$$\vdots$$
$$G*R(\mu_k\hat{f}_2+\lambda_k\hat{f}_1) \geq b'_k$$

where

$$b' = b - G*Rf_0$$

In Section 7.5, we will see how these equations can be turned into a set of linear constraints.

7.4 Objective Functions

In the previous sections, we saw how relations between object features can be expressed in terms of constraint equations involving their relative locations.

$$T_{AB} = transl(p_{AB})*R_{AB}$$

subject to

$$c_1(p_{AB},R_{AB}) \gtrless 0$$
$$\vdots$$
$$c_m(p_{AB},R_{AB}) \gtrless 0$$

In some cases, these equations can be solved completely to give a firm value for T_{AB}. Generally, however, this will not be the case, and T_{AB} will still be able to vary somewhat. As we have seen, this can be expressed in terms of scalar variables representing the remaining degrees of freedom.

$$T_{AB} = T_{AB}(\mu_1,\ldots,\mu_n)$$

so that our constraint equations may be rewritten

$$c_1'(\mu_1,\ldots,\mu_n) \gtrless 0$$
$$\vdots$$
$$c_m'(\mu_1,\ldots,\mu_n) \gtrless 0$$

Although the assignment of these scalar variables may be largely arbitrary, it turned out that we could frequently assign them in semantically meaningful ways. Thus, if we have two plane surfaces in contact, we get

$$R_{AB} = R_\gamma ROT(\hat{a},\omega)$$
$$p_{AB} = [x,y,z]$$

$$\hat{g}\cdot R_{AB}f = d - \hat{g}\cdot p$$

Here, parameter ω represents a rotation about a known axis of A, and x, y, and z are translations.

In such cases, it may be reasonable to ask about the allowable range of a parameter. For instance,

$$\min \omega = \omega_k^{min} \le \omega \le \omega_k^{max} = \max \omega$$

In other cases there may be no obvious parameter giving the relation desired. For instance, suppose we wish to know the maximum displacement, δ_{max}, of a point f on object A along an axis $\hat{g}$ of object B. (f might be the tip of a pin, and $\hat{g}$ might be the z axis of a hole). Then,

$$\begin{aligned}\delta_{max} &= \max \hat{g}\cdot T_{AB}f \\ &= \max \hat{g}\cdot R_{AB}f + \hat{g}\cdot p_{AB} \\ &= \max \hat{g}\cdot R_{AB}(\mu_1,\ldots,\mu_n)f + \hat{g}\cdot p_{AB}(\mu_1,\ldots,\mu_n) \\ &= \max c_0'(\mu_1,\ldots,\mu_n)\end{aligned}$$

In other words, we have expressed the problem of finding the maximum perturbation of some relation between two objects as a mathematical programming problem:

$$\max c_0'(\mu_1,\ldots,\mu_n)$$

subject to

$$c_1'(\mu_1,\ldots,\mu_n) \geq 0$$
$$\vdots$$
$$c_m'(\mu_1,\ldots,\mu_n) \geq 0$$

In the next section, we will explore ways to approximate the c_k' with forms that are simple enough for known efficient numerical techniques to be applied.

7.5 Linear Constraints

We are especially interested in obtaining systems of *linear* constraints on the free variables, since there are powerful numerical techniques that can be applied to such systems. In particular, it becomes very simple to find maximum and minimum values for linear (or many quadratic) functions of the free variables, thus giving a quick check on the likely range of the object's location. Similarly, it is possible (though sometimes rather expensive) to compute a convex polyhedron which encloses all feasible solutions to the location constraints.

For instance, if we have an object sitting on a table in some known orientation, then we get

$$OBJ = \text{transl}(x,y,h)*ROT(\hat{z},\alpha)*ROT(\hat{y},\beta)*ROT(\hat{z},\gamma)$$

where

α,β, & γ are known angles
h = some known height
$x_{min} \leq x \leq x_{max}$ (defining the size of the table)
$y_{min} \leq y \leq y_{max}$

This rather simple linear system can be solved (trivially) to give us the rectangle in which the object lies.

Even in cases where the constraints on degrees of freedom are non-linear, it is frequently convenient to approximate them with linear constraints. For instance, suppose that we have an object sitting in a rectangular box and we wish to know what possible positions and orientations it may be in. (We might, for example, be considering how to pick it up.) Assuming we know in which stable position the object is sitting, the problem reduces to the two-dimensional problem shown in Figure 7.3. If we approximate the outline of the object by a list of points $p_i=(x_i,y_i)$ in the object coordinate system, then our problem reduces to finding x,y, and ω such that

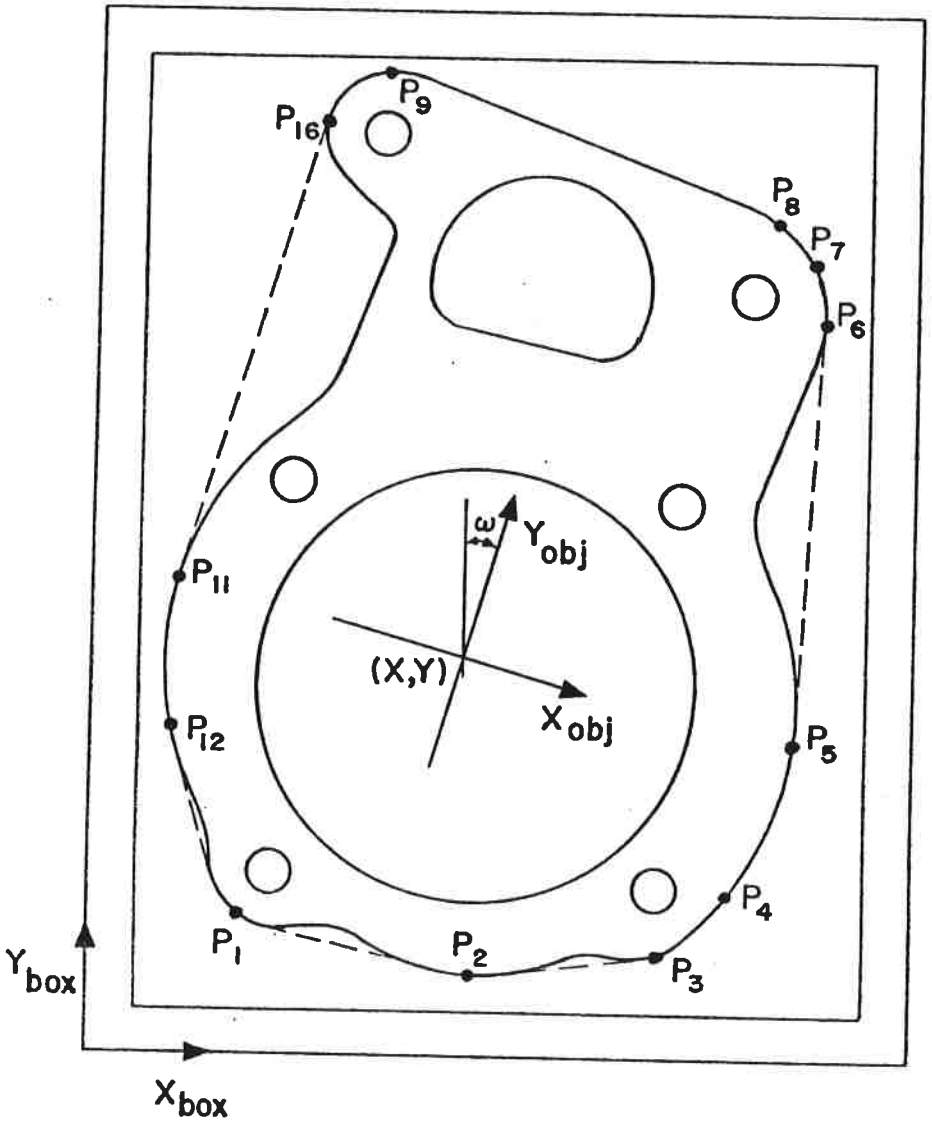


Figure 7.3. Object in a Box

$$\text{left} \quad \leq x + x_i \cos\omega - y_i \sin\omega \leq \text{right}$$
$$\text{bottom} \quad \leq y + y_i \cos\omega + x_i \sin\omega \leq \text{top}$$

for all the p_i. In addition, we may have some *a priori* knowledge about the object orientation:

$$\omega_{min} \leq \omega \leq \omega_{max}$$

Here, we have assumed that the sides of the box are aligned with the x and y-axes. If this is not the case, then the constraints are modified in the obvious manner.

We really would like to have linear constraints; unfortunately, these are not. However, we can rewrite the system to contain only one non-linear constraint by introducing two new variables, s & c, corresponding to $\sin\omega$ and $\cos\omega$.

$$\text{left} \quad \leq x + x_i c - y_i s \leq \text{right}$$
$$\text{bottom} \quad \leq y + y_i c + x_i s \leq \text{top}$$

for all p_i, and

$$c_{min} \leq c \leq c_{max}$$
$$s_{min} \leq s \leq s_{max}$$
$$s^2 + c^2 = 1$$

Now we can temporarily ignore the last constraint, and apply our LP methods. Then, the allowable range for ω (and, hence, for s & c) can be approximated by a number of small ranges centered about "typical" values $\omega_0,\ldots,\omega_k$ and approximate constraints of the form

$$c \cos\omega_j + s \sin\omega_j = 1$$

can be applied to obtain as nearly accurate a locus estimate as desired. In fact, it is frequently the case that the allowable range of ω is sufficiently tight so that a single such value is sufficient, and the corresponding constraint can just be "thrown into the pot" along with everything else.[5]

---

[5] Actually, it is sometimes possible to treat such constraints directly, since they have a nice, convex form. However, the current system prefers to work parametrically, as indicated above.

7.6 Algorithms

This section will present algorithms for applying the general methods discussed so far to produce a parameterized estimate of the relative location between two objects, given a set of relational constraints between features of the "affixment trees" rooted at each object. In particular, we will produce an estimate of the general form:

$$T_{AB} = \text{transl}(p_{AB}) * R_{\gamma} * \text{ROT}(\hat{a}_{AB}, \omega_{AB})$$

subject to linear constraints on p_{AB}, $\sin \omega_{AB}$, and $\cos \omega_{AB}$. These constraints will then be solved to produce estimates of legal ranges for ω_{AB}, and of the limits on p_{AB}. The translation proceeds roughly as follows:

1. Enumerate all relations between features "affixed to" object A and object B.

2. For each such relation, produce a set of corresponding constraint equations involving R_{AB} and p_{AB}. As each such relation is generated, it is merged with previous relations, and symbolic reductions are performed, if possible, to fix values for p_{AB}, $\hat{a}_{AB}$, and ω_{AB}. If any of these values is fixed, it is used to simplify the equations in the constraint set still further. Occasionally, these reductions will produce more than one candidate value for the axis $\hat{a}_{AB}$. This necessitates separate cases being kept for each possibility. If an inconsistency is discovered between the constraint set for some subcase and a newly added equation, then that subcase is dropped; exhaustion of all subcases indicates an "impossible" situation.

3. Once the equations for all relations have been added and simplified, they are "rewritten" in terms of constraints on scalar variables corresponding to the remaining degrees of freedom in R_{AB} and p_{AB}. These constraints are linearized, with new variables being invented to correspond to $\sin \omega_{AB}$ and $\cos \omega_{AB}$. This constraint set may then be used to compute maximum values for linear forms involving the free variables, with iterative techniques being used to enforce the constraint that

$$(\sin \omega_{AB})^2 + (\cos \omega_{AB})^2 = 1$$

and to find extreme values involving ω_{AB}.

We will deal with each of these phases in somewhat more detail below. If you are reading this document for the first time, you may want to skip over this material, and pick up reading again at Section 7.8

7.6.1 Finding relevant relations

We are interested in finding all relations between pairs of features, F_A and F_B, that constrain the position of object A with respect to object B. Also, we need to find the position of F_A and F_B with respect to A and B. This is essentially a clerical task of using information stored in the object models and the assertions in the planning world's data base:

1. Compute $Q_A \leftarrow$ *affixment set* for A. I.e., find the set of all objects X having a known fixed relation to A in the current planning world. Details of this computation are given later. For each such X, compute T_{AX} such that.

$$X = A * T_{AX}$$

Here, recall that when we use an object name in an arithmetic context, we mean the location frame associated with the object.

2. In the same manner, compute $Q_B \leftarrow$ *affixment set* for B. If there is an object X in $Q_A \cup Q_B$, then the relation of A to B will be given by

$$T_{AB} = T_{XA}^{-1} * T_{XB}$$

and we have no need to proceed further.

3. Now, for each object X in Q_A and each object Y in Q_B, search the data base for any assertions relating a feature F_X of X to a feature F_Y of Y. As stated, this process could be extremely tedious, since each possible relation pattern would have to be checked for all features of X and Y. [6] An easy way around this difficulty is to note what objects are related by an assertion at the time the assertion is made. We have done this by means of a LEAP bracketed triple:

symbolic⊗[related⊗x⊗y]⊗<*symbolic assertion id*>

This means that the relevant assertions may be found fairly quickly by

```
relations ← empty;
for each x,y such that x ∈ Q_A and y ∈ Q_B do
  for each rln such that symbolic⊗[related⊗x⊗y]⊗rln do
    if true_in(current_world,rln) then
      put rln in relations;
```

In other words, enumerate by "name" all symbolic assertions linking any element x of Q_A with some element y of Q_B. Put any such relations which are "true" into the set *relations*.

---

[6] Alternatively, all instances of each pattern could be generated, and a check made to see if the features involved belong to objects in Q_X and Q_Y.

The relevant assertion patterns are given below.

assertion patterns

The assertion patterns currently "understood" by the system are discussed below.

pattern: (contacts, F_A, F_B, <*kind of contact*>)

Asserts that F_A and F_B are in contact. F_A and F_B may be surfaces, regions of surfaces, sectors of cylinders, or sectors of spheres. [7] Here, we insist that the F_B be "larger" than F_A. Thus, if F_A is a surface, then F_B may not be an edge, etc. This rather unfortunate limitation comes from difficulties in linearizing the resulting mathematical forms. We will discuss the difficulty more in the next subsection.

<*kind of contact*> supplies additional information about how the relation is to be interpreted (i.e., what inequality constraints are to be generated). For feature to surface contact, the relevant values are:

inside of – If F_A and F_B are surface regions, then F_A lies entirely inside of F_B. If F_A is a cylinder, then both ends are assumed to contact F_B. If F_A is a sphere, then the point of contact will be inside of F_B.

overlaps – Asserts that some point of F_A contacts some point of F_B. The "normal" contact conditions remain in force. Thus, if two surfaces are asserted to overlap, we still assume that the outward facing normals to those two surfaces must point in opposite directions.

extent_irrelevant – Asserts that only the fact of contact is to be considered. This is principally useful as a means of eliminating many completely useless inequalities. For instance, if we assert that an object lies on the top of the work table, it is probably not too useful to go ahead and add the assertion that its possible locations are bounded by the sides of the table. We will most likely have other relations that restrict the position much more than this.

---

[7] Recall that edges (between two surfaces) and vertices are treated as sectors of zero radius cylinders and spheres, respectively. (See Figure 6.3).

Pattern: (inserted,<*shaft*>,<*hole*>,<*direction*>,<*min dist*>,<*max dist*>)

Asserts that <shaft> is inserted into <hole> the distance indicated. <direction> is ±1, and gives the direction of the shaft axis with respect to the outward facing normal of the hole. Presently, only round shafts in round holes are considered.

Pattern: (above,F_A,<*surface region*>,<*min dist*>,<*max dist*>,<*kind of contact*>)

This relation is sometimes useful when one wishes to describe a region in which a feature must be contained. It asserts that feature F_A is inside a cylinder over the indicated surface region, extending up within the indicated distance parameters.

Pattern: (points_at,F_A,F_B,<*angle*>)

This relation is most useful as a means of specifying the approximate orientation of an object. F_A and F_B must have some natural direction associated with them. (For a cylinder, the axis; for a surface or hole, the outward facing normal; etc.). If <angle> is positive then the direction vectors are assumed to be parallel to within <angle>; if <angle> is negative, they are anti-parallel.

7.6.2 Generating the constraint equations

Here, we assume that we have features F_A and F_B related to objects A and B by

$$F_A = A * T_{AF}$$
$$F_B = B * T_{BF}$$

together with some "symbolic" relation between them. Typically, this relation involves contact between the two features, together with additional information specifying a region on one feature that is constrained to be within a region of the other. The mathematical constraints we will be generating follow the general forms:

$$\hat{g} \cdot R_{AB} f \; ? \; d - \hat{g} \cdot p_{AB}$$
$$\hat{g} \cdot R_{AB} f \; ? \; d$$
$$0 \; ? \; d - \hat{g} \cdot p_{AB}$$

where $\hat{g}$ and f are constant vectors and d is a constant scalar. The "?" will generally be "=" for constraints arising out of contacts, and "≤" for remaining attributes of the relation. Since we are dealing with standard relations, it is not necessary to go through a full derivation of the constraint forms each time a relation is to be added. Instead, we use

special procedures for each relation type. The first step of all these procedures is the same, however: produce values of f_i and g_i corresponding to the coordinates of F_A and F_B. Thus,

$f_0 \leftarrow p_{AF}$ (i.e., translation part of T_{AF})
$\hat{f}_1 \leftarrow R_{AF}\hat{x}$
$\hat{f}_2 \leftarrow R_{AF}\hat{y}$
$\hat{f}_3 \leftarrow R_{AF}\hat{z}$

$g_0 \leftarrow p_{BF}$
$\hat{g}_1 \leftarrow R_{BF}\hat{x}$
$\hat{g}_2 \leftarrow R_{BF}\hat{y}$
$\hat{g}_3 \leftarrow R_{BF}\hat{z}$

The conventions for feature coordinate systems may be found in Figure 7.2. Once making these transformations, each procedure then adds the appropriate constraint equations, as shown below.

Constraints for Contact Relations

F_A	F_B	Constraint forms to add
Surface	Surface	$\hat{g}_3 \cdot R_{AB}\hat{f}_3 = -1$ [8]
		$\hat{g}_3 \cdot R_{AB} f_0 = (\hat{g}_3 \cdot g_0) - \hat{g}_3 \cdot p_{AB}$
Cylinder	Surface	$\hat{g}_3 \cdot R_{AB}\hat{f}_3 = 0$
		$\hat{g}_3 \cdot R_{AB} f_0 = (\hat{g}_3 \cdot g_0) + r_{cyl} - \hat{g}_3 \cdot p_{AB}$
		$\hat{g}_3 \cdot R_{AB}\hat{f}_1 \leq \cos\theta_{cyl}$ (ignore if full cylinder)
Sphere	Surface	$\hat{g}_3 \cdot R_{AB}\hat{f}_0 = (\hat{g}_3 \cdot g_0) + r_{sph} - \hat{g}_3 \cdot p_{AB}$
		$\hat{g}_3 \cdot R_{AB}\hat{f}_3 \leq -\cos\theta_{sph}$ (ignore if full sphere)

---

[8] Note that this is slightly different than our earlier derivation.

point	line	$\hat{g}_2 \cdot R_{AB} f_0 = \hat{g}_2 \cdot g_0 - \hat{g}_2 \cdot p_{AB}$
		$\hat{g}_1 \cdot R_{AB} f_0 = \hat{g}_1 \cdot g_0 - \hat{g}_1 \cdot p_{AB}$
point	point	$\hat{g}_1 \cdot R_{AB} f_0 = \hat{g}_1 \cdot g_0 - \hat{g}_1 \cdot p_{AB}$
		$\hat{g}_2 \cdot R_{AB} f_0 = \hat{g}_2 \cdot g_0 - \hat{g}_2 \cdot p_{AB}$
		$\hat{g}_3 \cdot R_{AB} f_0 = \hat{g}_3 \cdot g_0 - \hat{g}_3 \cdot p_{AB}$

Additional constraints arising from the <kind of contact> attribute are generated in just the manner described in Section 7.2. For instance, if we know that a vertex f_0 of object A is resting on a rectangular face of object B, we will get constraints of the form:

$$\hat{g}_1 \cdot R_{AB} f_0 \leq \delta_x - \hat{g}_1 \cdot p_{AB}$$
$$\hat{g}_1 \cdot R_{AB} f_0 \geq -\delta_x - \hat{g}_1 \cdot p_{AB}$$
$$\hat{g}_2 \cdot R_{AB} f_0 \leq \delta_y - \hat{g}_2 \cdot p_{AB}$$
$$\hat{g}_2 \cdot R_{AB} f_0 \geq -\delta_y - \hat{g}_2 \cdot p_{AB}$$

Similarly, if a region of surface F_A, with extreme points (x_i, y_i), is inside F_B, constraints are generated to put each point

$$f_0 + x_i \hat{f}_1 + y_i \hat{f}_2$$

inside F_B. Circular patches on F_A may be handled slightly more efficiently by constraining the center of F_A to lie within an appropriately "shrunken" patch on F_B.

The remaining relations are handled similarly. Insertion of a shaft into a hole gives the relations:

$$\hat{g}_3 \cdot R_{AB} \hat{f}_3 = \pm 1$$
$$\hat{g}_1 \cdot R_{AB} \hat{f}_0 = \hat{g}_1 \cdot g_0 - \hat{g}_1 \cdot p_{AB}$$
$$\hat{g}_2 \cdot R_{AB} \hat{f}_0 = \hat{g}_2 \cdot g_0 - \hat{g}_2 \cdot p_{AB}$$

$$\hat{g}_3 \cdot R_{AB} \hat{f}_0 \leq -\langle \text{min dist} \rangle - \hat{g}_3 \cdot R_{AB} \hat{p}_{AB}$$
$$\hat{g}_3 \cdot R_{AB} \hat{f}_0 \geq -\langle \text{max dist} \rangle - \hat{g}_3 \cdot R_{AB} \hat{p}_{AB}$$

for a snug fit. The "points at" relation produces a constraint like

$$\hat{g}_3 \cdot R_{AB} \hat{f}_3 \geq \cos \langle \text{angle} \rangle$$

and the "above" relation is handled in a manner analogous to the contact constraints, with inequalities replacing the equalities resulting from contact.

7.6.3 Merging the constraint equations

Once the mathematical constraints are generated, they must be simplified before useful information can be extracted from them. In part, this is because of limitations in the ability of our numerical solution technique (linear programming) to handle rotations. [9] It is convenient to think of a body rotation R as consisting of a rotation of the body by some parameter ω about an axis $\hat{a}$ of the body, followed by a rotation R_γ that reorients $\hat{a}$.

$$R = R_\gamma * \mathrm{ROT}(\hat{a},\omega)$$

R_γ, in turn may be parameterized in terms of a pair of rotations such as

$$R_\gamma = \mathrm{ROT}(\hat{x},\alpha) * \mathrm{ROT}(\hat{y},\beta)$$

However, to produce even approximately linear equations (i.e., with no terms involving products of parameters), we must have constant values for R_γ and $\hat{a}$. To get such values, we must perform symbolic reductions similar to those discussed earlier in Section 7.2. The simplifications are performed incrementally, as each constraint is added. The procedure followed looks something like this:

0. Initially, assume that we have a set U consisting of constraint equations u_i that are to be added, and a set V consisting of constraints v_i which have already been merged. In addition, our knowledge of R_{AB} will be one of:

   unspecified – Nothing is known about R_γ, $\hat{a}$, or ω.

   axis specified – R_γ and $\hat{a}$ fixed, but ω free.

   fully specified – $R_{AB} = R_\gamma * \mathrm{ROT}(\hat{a},\omega_0)$ = a constant.

---

[9] Of course, an important additional consideration is reducing the amount of computation required by simplifying the equations and reducing the number of degrees of freedom that must be considered.

and p_{AB} may be either *free* or *fixed*.

1. If U is empty, then we are done. Otherwise, take the next constraint u_i from U. Perform any simplifications possible on u_i, based on what is already known about R_{AB} and p_{AB}. [10]

   a. If u_i includes $\hat{g} \cdot R_{AB} f$ and R_{AB} is fully specified, then perform the indicated multiplications.

   b. Similarly, if u_i includes $\hat{g} \cdot p_{AB}$, and p is fixed, then perform the indicated multiplication.

   c. If u_i includes $\hat{g} \cdot R_{AB} f$, and R_{AB} is axis-specified, then replace $\hat{g} \cdot R_{AB} f$ with $\hat{g} \cdot R_\gamma f$ if $R_{\hat{\gamma}} \perp \hat{g}$ or f is parallel to $\hat{a}$.

   Once simplification has taken place, we may wind up with a "degenerate" constraint, such as

   $$0.5 \geq 0$$

   or

   $$0 = .35$$

   In this case, check to see whether the constraint is feasible. If it is, then just drop it, and go back to the start of step 1 to get the next one. If not, then we have specified an "impossible" situation. [11] If an earlier reduction was "ambiguous" (see below), then this subcase is just dropped. If no subcases remain, then the user has specified something impossible, so we just give up and complain.

2. Now compare u_i with the elements v_j of V, looking for additional simplifications to make. This is done by making a great number of special case checks. The most important reductions are described below. Such simplifications will typically produce one or more new constraint equations, which are to be added to U, together with (possibly) additional restrictions on R_{AB} and p_{AB}. In this latter case, all previous constraint equations will be considered for simplification in the manner described in step 1. Any constraints so modified will also be added to U.

---

[10] Actually, these simplifications are best performed when the constraint equations are added to U, rather than when they are removed, as here.

[11] In practice, it has proved desirable to allow a small amount of tolerence before assuming that the situation is impossible, since situations may be overconstrained in some fairly innocent way. Thus, "0.001=0", would not be considered cause for rejection.

Constraints	Actions
$\hat{g}\cdot R_{AB}\hat{f} = \pm 1$	fix rotation axis
$\hat{g}_1\cdot R_{AB} f_1 = 0$ $\hat{g}_1\cdot R_{AB} f_2 = 0$	if $\hat{g}_1$ parallel to $\hat{g}_2$ or f_1 parallel to f_2 then fix rotation axis
$\hat{g}\cdot R f_1$? <rhs> $\hat{g}\cdot R f_2 = 0$	Form new constraint $\hat{g}\cdot R(f_1-\alpha f_2)$? <rhs> where $\alpha = (f_2\cdot f_1)/(f_2\cdot f_2)$
$g_1\cdot Rf$? <rhs> $g_2 R_f = 0$	Form new constraint $(g_1-\alpha g_2)Rf$? <rhs> where $\alpha = (g_2\cdot g_1)/(g_2\cdot g_2)$
$\hat{g}\cdot R_{AB} f_1 = d_1 - \hat{g}\cdot p$ $\hat{g}\cdot R_{AB} f_2$? $d_2 - \hat{g}\cdot p$	form new constraint $\hat{g}\cdot R_{AB}(f_1-f_2)$? d_2-d_1
$0 = d_1 - \hat{g}_1\cdot p$ $0 = d_2 - \hat{g}_2\cdot p$ $0 = d_3 - \hat{g}_3\cdot p$	if $\hat{g}_1, \hat{g}_2, \hat{g}_3$ linearly independent, then solve for p
$\hat{g}\cdot R_{AB} f = d_1$	if axis already fixed, and f or $R^{-1}\hat{g}$ non parallel to a, attempt to fix ω.

Ambiguities in Fixing the Rotation Axis

As we saw in Section 7.2, ambiguities can sometimes arise when we attempt to fix the axis of rotation. For instance, if we have constraints

$$\hat{x}\cdot R_{AB}\hat{z} = 0$$
$$\hat{y}\cdot R_{AB}\hat{z} = 0$$

Then we get

$$R_{AB} = R_\gamma * ROT(\hat{z},\omega)$$

where

$$R_\gamma \hat{z} = \pm \hat{z}$$

I.e., we will have *two* distinctly different rotations. In some other cases, we may have four or more possibilities. Currently, this is handled by *splitting* the problem into subcases. Each subcase is then reduced independently. If a contradiction is found, the subcase may be dropped, as we indicated earlier. If not, then either it will be "solved" completely, or the numerical estimation techniques of the next two sections are applied to produce a parameterized range estimate.

7.6.4 Converting the Constraint Equations

Once the constraint equations have been simplified as much as possible, we wish to use numerical techniques to investigate the properties of the remaining degrees of freedom. The technique chosen for this purpose was linear programming, which has the advantage that efficient algorithms are available, but also has a drawback in that it requires that constraints and the objective function be expressed in terms of linear forms, whereas rotational degrees of freedom may introduce non-linearities. At first glance, this may seem to be a fatal objection. However, wherever the rotation has been successfully constrained to a single degree of freedom – as is the case when two object faces are in contact or where a pin on one object has been inserted into a hole in the other object – then it is possible to approximate the constraint equations in terms of linear forms.

In this case, we have

$$R_{AB} = R_\gamma ROT(\hat{a},\omega)$$

which we rewrite as

$$R_{AB} = R_\gamma R_\alpha^{-1} ROT(\hat{z},\omega) R_\alpha$$

where

$$R_\alpha \hat{a} = \hat{z}$$

Thus, whenever

$$\hat{g} R_{AB} f$$

appears in a constraint equation, it will reduce to

$$\begin{aligned}\hat{g} R_{AB} f &= \hat{g} R_\gamma R_\alpha^{-1} ROT(\hat{z},\omega) R_\alpha f \\ &= \hat{g}' ROT(\hat{z},\omega) f' \\ &= (\hat{g}'_x f'_x + \hat{g}'_y f'_y)\cos\omega + (\hat{g}'_y f'_x + \hat{g}'_x f'_y)\sin\omega + \hat{g}'_z f'_z \\ &= \beta_1 \sin\omega + \beta_2 \cos\omega + \beta_3\end{aligned}$$

where

$$\hat{g}' = R_\alpha R_{\bar{\gamma}} \hat{g}$$
$$f' = R_\alpha f$$

Now, we introduce two new variables

$$c_\omega = \cos \omega$$
$$s_\omega = \sin \omega$$

and rewrite the equations in the obvious manner. For instance, if R_{AB} is axis-specified, and $p = [x,y,z]$, then

$$\hat{g} R_{AB} f = d - \hat{g} \cdot p$$

would be rewritten

$$p_x x + p_y y + p_z z + \beta_1 c_\omega + \beta_2 s_\omega = d - \beta_3$$

This leaves us with only one non-linear constraint equation.

$$c_\omega^2 + s_\omega^2 = 1$$

This constraint may be handled parametrically, as described in the next section.

7.6.5 Computing Rotation Ranges

This section describes an iterative technique for finding feasible ranges for a rotation angle ω, given linear constraint equations in $\sin\omega$, $\cos\omega$, and other free variables.

The method starts by dividing the unit circle into arcs:

$$\omega_0 - \delta_0 \le \omega \le \omega_0 + \delta_0$$

An iterative technique is then used to refine each arc into one or more tighter subranges, with the process being continued until the endpoints of each subrange have been determined to within a desired precision. Finally, adjacent subranges are merged. [12]

The details of the iterative refinement process are shown below, and are illustrated in Figure 7.4 and Figure 7.5.

0. Initially, we have a system of constraints

---

[12] This last step is necessitated by the fact that more than one initial subrange was used.

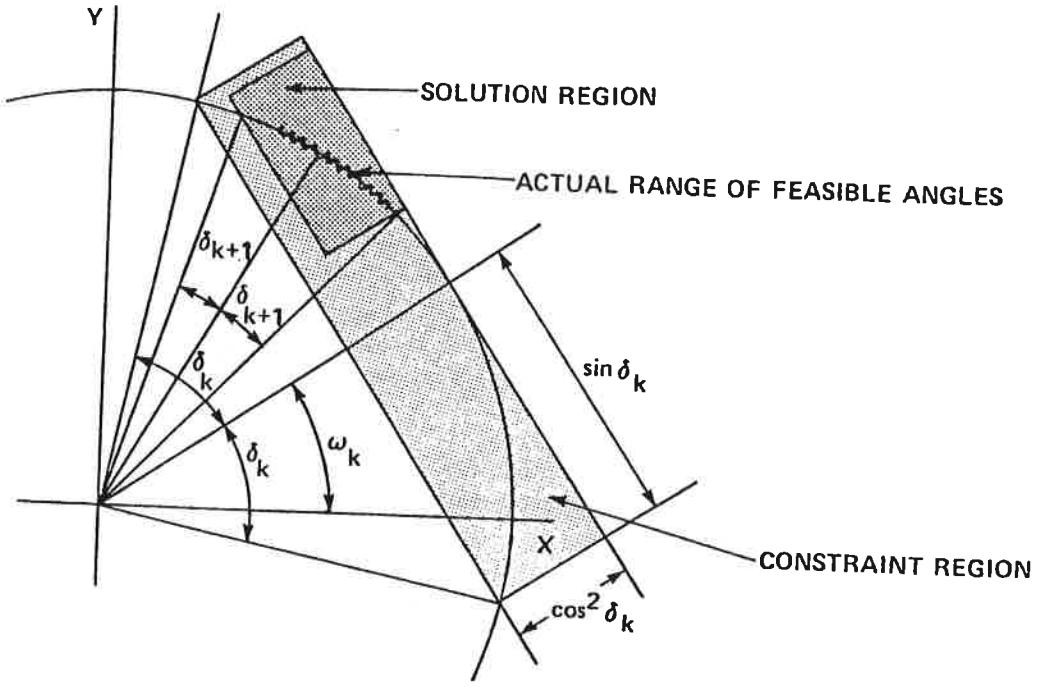


Figure 7.4. Computing Rotation Range: Iteration k

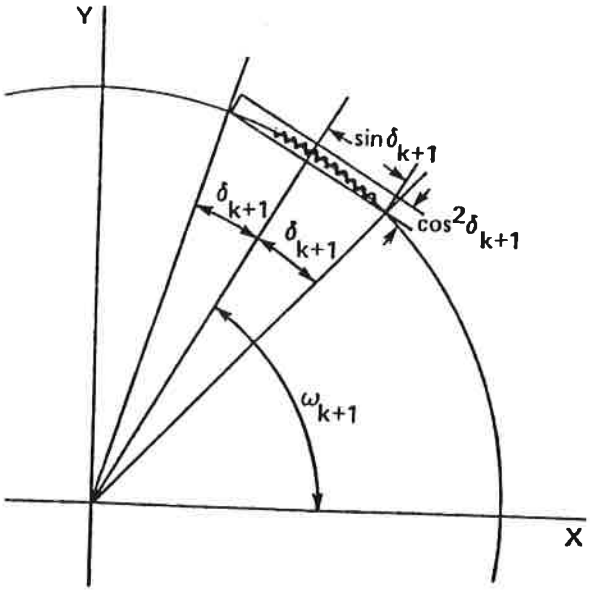


Figure 7.5. Computing Rotation Range: Iteration k+1

$$C(\lambda_1,\ldots,\lambda_n,c_\omega,s_\omega) \ge 0 \qquad \text{[Eq 3]}$$

where

$$c_\omega = \cos\omega$$
$$s_\omega = \sin\omega$$
$$\omega_0-\delta_0 \le \omega \le \omega_0+\delta_0$$

We wish to refine this interval into finer sub-intervals.

1. At the k'th iteration, we assume that we still have system [Eq 3], but that the interval has been narrowed down to

$$\omega_k-\delta_k \le \omega \le \omega_k+\delta_k$$

Let

$$p_k = \cos\omega_k$$
$$q_k = \sin\omega_k$$

Augment the system [Eq 3] with constraints to restrict c_ω and s_ω to lie in a rectangle bounded by

$$\cos^2\delta_k \le p_k c_\omega + q_k s_\omega \le 1$$
$$-\sin\delta_k \le -q_k c_\omega + p_k s_\omega \le \sin\delta_k$$

2. Use linear programming methods[13] to solve the augmented system to produce a new rectangle bounded by

$$\min(p_k c_\omega+q_l s_\omega) \le p_k c_\omega + q_k s_\omega \le \max(p_k c_\omega+q_k s_\omega)$$
$$\min(-q_k c_\omega+p_l s_\omega) \le -q_k c_\omega + p_k s_\omega \le \max(-q_k c_\omega+p_k s_\omega)$$

Zero, one, or two intervals of the unit circle will be inside this rectangle. Treat each of these subcases as follows:

---

[13] At present, we use the revised simplex method [30] on the *dual* of the system derived in step 1. The reason for solving the dual problem is that step 1 typically produces many more constraints than free variables. In the dual, constraints and variables are interchanged. Since computation times in the simplex method are more strongly dependent on the number of constraints, making this swap speeded up the solution process. There are several minor complications arising from this approach. For instance, if the dual has no feasible problem, then the primal is either unbounded or infeasible. However, the resolution of such "sticky" points is straightforward, and is not particularly relevant to our present discussion.

a. *zero* – Terminate. There are no feasible solutions in the current subinterval.

b. *one* – Compute values ω_{k+1} and δ_{k+1} so that the arc cut by the rectangle is described by

$$\omega_{k+1}-\delta_{k+1} \le \omega \le \omega_{k+1}+\delta_{k+1}$$

If $(\delta_k-\delta_{k+1})$ is less than the desired stopping threshold, then stop; otherwise, set k←k+1, and go back to step 1 for another iteration.

c. *two* – In this case, compute the desired boundary angles for each sub-interval. Save one of the sub-intervals away for later processing and proceed with the refinement of the other.

A Bug

It is possible for the procedure described above to terminate prematurely, with the assertion that all rotations are feasible within a range when, in fact, there are infeasible sub-intervals. This is illustrated in Figure 7.6 In practice, this hasn't turned out to be much of a problem. If desired, the likelihood of getting "bitten" can be reduced by breaking any subinterval that satisfies the termination condition of step 2.b but has δ_{k+1} larger than some threshold into two subintervals and then proceeding as in step 2.c.

7.7 Experience

The algorithms described in Section 7.6 worked fairly well – though slowly – so long as the rotation axis between the two objects could be fixed.[14] The principal limitations were (1) the necessity for fixing the rotation axis before the numerical solution procedures could be applied, and (2) difficulties in handling chains of non-rigid linkages. For instance, suppose our box is placed in a vise. Then it will be free to rotate about an axis perpendicular to the vise jaws. Now, suppose we place the cover plate on the top surface of the box. The plate can rotate freely about an axis perpendicular to the top surface of the box. If we are interested in the location of the cover with respect to the work station, we are in trouble, since there are *two* rotational degrees of freedom, but our numerical technique relies on there being only one. Of course, we could always split one of the rotational freedoms into many subcases, and apply the algorithms already developed to each subcase, although such an approach might be rather expensive. Alternatively, we could expand our symbolic solution procedures to handle more complex expressions.

In practice, this sort of problem hasn't been too severe in cases where we are interested in

---

[14] Appendix E.1 gives a typical example problem (a familiar-looking box sitting inside a box-like fixture).

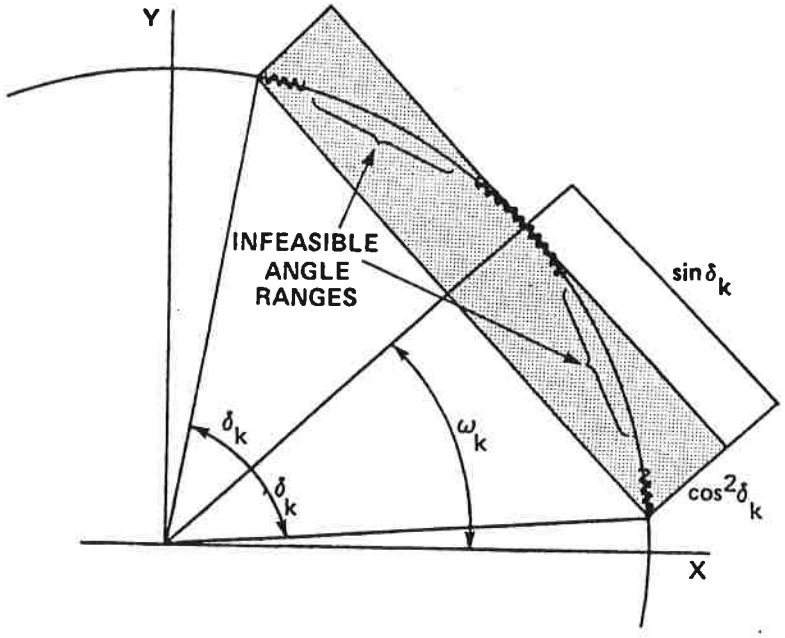


Figure 7.6. Premature Termination Bug

expected *locations* of objects, since there is frequently only one *significant* rotational freedom. Generally, one can ignore small perturbations while planning the overall motions to be used to accomplish a task. On the other hand, such perturbations cannot be ignored when one considers the accuracy issues involved. Fortunately, if we are willing to assume only small perturbations, it is possible to make many simplifying assumptions that allow us to apply efficient numerical techniques to cases involving many interacting degrees of freedom. These techniques will be discussed in the next section.

7.8 Differential Approximation

In general, the relation between two objects may be represented by an expression of the form

$$T = \text{transl}(p(\lambda_1,\ldots,\lambda_m))*R(\mu_1,\ldots,\mu_n)$$

It is convenient to interpret the λ_i and μ_i as degrees of freedom perturbing p and R from some nominal values

$$p(0,\ldots,0) = p_0$$
$$R(0,\ldots,0) = R_0$$

For suitably small λ_i and μ_i, we can make linear approximations for p and R:

$$p \cong p(0,\ldots,0) + \Sigma\, \lambda_i(\partial p/\partial\lambda_i)(0,\ldots,0)$$
$$= p_0 + \Sigma\, \lambda_i p_i$$

$$R \cong R(0,\ldots,0) + \Sigma\, \mu_i(\partial R/\partial\mu_i)(0,\ldots,0)$$
$$= R_0 + \Sigma\, \mu_i M_i$$

where

$$p_i = (\partial p/\partial\lambda_i)(0,\ldots,0)$$
$$M_i = (\partial R/\partial m_i)(0,\ldots,0)$$

Example

Consider the situation shown in Figure 7.7, which could arise during execution of an assembly program for a small engine. Here, the crank assembly would be in approximately the same position each time the task is executed, although it will be subject to minor variations.

$$\text{Crank} = \text{Vise} * T_{VC}$$
$$= \text{Vise} * \text{transl}(p_{VC}) * R_{VC}$$

where

$$p_{VC} = p_{\sqrt{C}} + \lambda_1\hat{x} + \lambda_2\hat{y} + \lambda_3\hat{z}$$
$$R_{VC} = R_{\sqrt{C}} * \text{Rot}(\hat{z},\omega) * \text{Rot}(\hat{y},\phi) * \text{Rot}(\hat{x},\theta)$$

where

$$-\epsilon_\omega \le \omega \le \epsilon_\omega, \quad -\epsilon_\phi \le \phi \le \epsilon_\phi, \quad -\epsilon_\theta \le \theta \le \epsilon_\theta$$

Suppose, now, that we are interested in finding out whether placing the hand at a fixed position with respect to the vise

$$\text{Hand} = \text{Vise} * T_{grab}$$

will guarantee that the shaft end will always be between the finger pads. The algorithms discussed in the previous section would require that we fix the rotation axis of the crank before we can get very far, but we cannot do this. Fortunately, since the orientation variations are small, we can approximate the rotation component by

$$R_{VC} \cong R_{\sqrt{C}} * (I + \omega M_z + \phi M_y + \theta M_x)$$

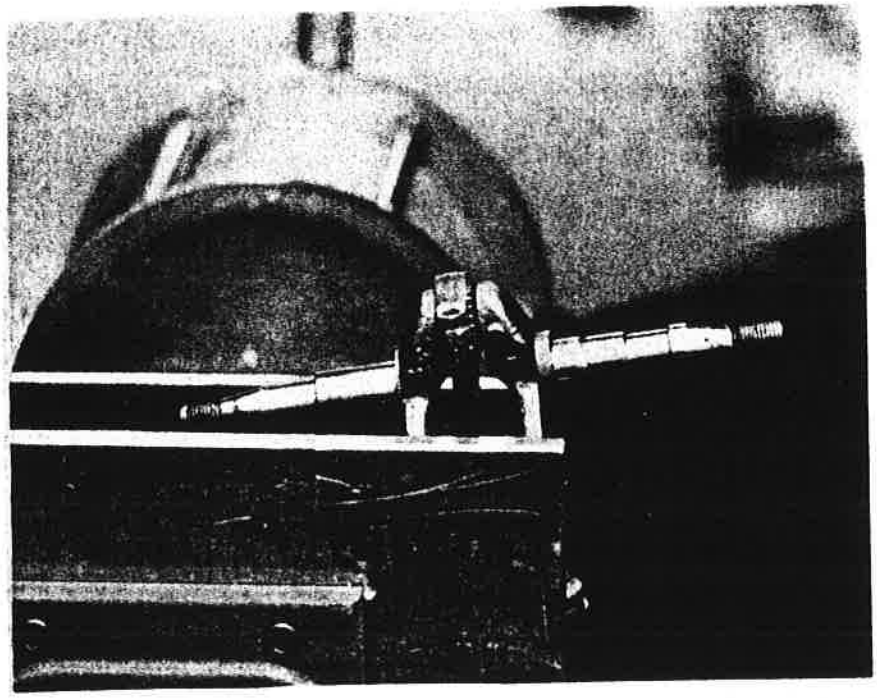


Figure 7.7. Crankshaft in Vise

where

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

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

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

It is then a straightforward matter to compute the maximum displacement of the shaft tip with respect to the expected hand position.

Chains of Linkages

One difficulty with our earlier techniques was that they did not work well where two objects were related through a chain of semi-fixed links. So long as the perturbations possible on the links are small, we can use differential approximations to handle such cases.

Suppose that the vise is free to rotate a bit around a fixed axis relative to the table.

$$\begin{aligned} \text{Vise} &= \text{Table} * T_{TV} \\ &= \text{Table} * \text{transl}(p_{TV}) * R_{TV} \\ &= \text{Table} * \text{transl}(p_{TV}^0) * R_{TV}^0 * \text{Rot}(\hat{z},\nu) \end{aligned}$$

Then the location of the crank with respect to the table will be given by

$$\begin{aligned} T_{TC} &= T_{TV} * T_{VC} \\ &= \text{transl}(p_{TV}) * R_{TV} * \text{transl}(p_{VC}) * R_{VC} \\ &= \text{transl}(p_{TV}^0) * R_{TV}^0 * \text{ROT}(\hat{z},\nu) * \text{transl}(p_{VC}) * R_{VC} \end{aligned}$$

If we take advantage of the fact that ν is small, this decomposes to

$$\begin{aligned} p_{TC} &= p_{TV}^0 + R_{TV}(\nu) p_{VC}(\lambda_1,\lambda_2,\lambda_3) \\ &\simeq p_{VC}^0 + R_{TV}^0 p_{VC}^0 + \lambda_1 R_{TV}^0 \hat{x} + \lambda_2 R_{TV}^0 \hat{y} + \lambda_3 R_{TV}^0 \hat{z} + \nu R_{TV}^0 M_z p_{VC}^0 \end{aligned}$$

$$p_{TC} \simeq p_{TC}^0 + \Sigma \delta_k p_{TC}^k$$

and

$$\begin{aligned} R_{TC} &= R_{TV} R_{VC} \\ &\simeq R_{TV}^0 R_{VC}^0 (I + \omega M_z + \phi M_y + \theta M_x + \nu M_z) \end{aligned}$$

which is again linear in the degrees of freedom. In general, if we have

$$\begin{aligned} T_{AC} &= T_{AB} * T_{BC} \\ &= \text{transl}(\, p_{AB}(\bar{\lambda})\,) R_{AB}(\bar{\omega}) \text{transl}(\, p_{BC}(\bar{\mu})\,) R_{BC}(\bar{\theta}) \end{aligned}$$

we will get expressions of the form

$$\begin{aligned} p_{AC} &\simeq p_{AB}^0 + \Sigma \lambda_k p_{AB}^k + \Sigma \mu_k R_{AB}^0 p_{BC}^k \\ R_{AC} &\simeq R_{AB}^0 R_{BC}^0 (I + \Sigma \omega_k (R_{BC}^0)^{-1} M_{AB}^k R_{BC}^0 + \Sigma \theta_k M_{BC}^k) \end{aligned}$$

Similarly, for T_{AB}^{-1} we have

$$T_{AB}^{-1} = T_{BA}$$

$$\begin{aligned} p_{BA} &= -R_{AB}^{-1} p_{AB} \\ &\simeq -(R_{AB}^0)^{-1} p_{AB}^0 + \Sigma -\lambda_k (R_{AB}^0)^{-1} p_{AB}^k + \Sigma -\omega_k M_{AB}^k (R_{AB}^0)^{-1} p_{AB}^0 \end{aligned}$$

$$\begin{aligned} R_{BA} &= R_{AB}^{-1} \\ &\simeq (R_{AB}^0)^{-1} (I + \Sigma -\omega_k R_{AB}^0 M_{AB}^k (R_{AB}^0)^{-1}) \end{aligned}$$

Error Estimates

The ability to handle perturbations in several rotation axes and involving multiple linkages is especially important when error estimates must be considered. In general, the "actual" value of a quantity can differ somewhat from that believed by the program.

$$T_{AB}^{actual} = T_{AB}^{calculated} * \Delta T_{AB}$$

ΔT_{AB}, in turn, can be parameterized by free variables corresponding to various measurement or modelling errors. For instance, suppose that an assembly program used constant affixment values to describe the crank-vise-table linkage discussed above.

AFFIX Crank TO Vise AT T_{VC}^{0};
AFFIX Vise TO Table AT T_{TV}^{0};

Then,

$$\Delta T_{TC} = (T_{TC}^{actual})^{-1} * T_{TC}^{0}$$

would be approximated by the linear expressions in $\bar{\lambda}$, ω, ϕ, θ, and ν which we derived earlier.

Multiple Path Linkages

It is not uncommon for two objects to be related via several paths. For instance, suppose that the manipulator grasps the crankshaft at one end. This will give us an equation relating the hand to the table.

$$\begin{aligned} \text{Hand} &= \text{Crank} * T_{CH} \\ &= \text{Table} * T_{TV}(\nu) * T_{VC}(\lambda_1,\lambda_2,\lambda_3,\omega,\phi,\theta) * T_{CH} \end{aligned}$$

Also, the hand will be related to the table via a series of joint links

$$\text{Hand} = \text{Table} * J_0 * J_1(\eta_1) * \ldots * J_6(\eta_6)$$

where the η_i correspond to deviations in each of the joint angles.

$$-\epsilon_{\eta_i} \le \eta_i \le \epsilon_{\eta_i}$$

In principle, either one of these chains could be used to calculate the hand position.[15] We can compute the relative accuracy of either chain straightforwardly. For instance, suppose we want to estimate the maximum variation in the hand's orientation about its "approach" vector.

---

[15] Because of the special nature of the hand frame in the AL system, the joint linkage chain is always used.

$$R_{TH} \simeq R^{0}_{TH} * ROT(\hat{z},\rho)$$

The orientation deviation angle will be approximated by

$$\rho \simeq \hat{y} R_{TH} \hat{x}$$

Thus, each chain can be evaluated by computing the limits on

$$\rho = \hat{y}(R_{TV}(\nu)R_{VC}(\omega,\phi,\theta)R_{CH})\hat{x}$$

and

$$\rho = \hat{y}(R_{J_0} * R_{J_1}(\nu_1) * \ldots * R_{J_6}(\nu_6))\hat{x}$$

respectively, subject to whatever additional constraints may be appropriate. We can then select whichever computation promises to give the best result. However, neither of these two affixment chains, by itself, will necessarily fully reflect what is known about the hand position. To do this, we must consider both constraint equations together. We require some way to express equality of two FRAME expressions in terms of linear constraints. This may be done by rewriting equations of the form

$$T_1 = T_2$$

into

$$T_{12} = T_1^{-1} * T_2 = I$$

which we can re-express in terms of six scalar equations

$$\hat{x} \cdot p_{12} = 0 \quad \hat{z} R_{12} \hat{y} = 0$$
$$\hat{y} \cdot p_{12} = 0 \quad \hat{x} R_{12} \hat{z} = 0$$
$$\hat{z} \cdot p_{12} = 0 \quad \hat{y} R_{12} \hat{x} = 0$$

For instance, suppose we have represented the location of an object B with respect to object A in terms of two different sets of parameters, with independent constraints on each set.

$$p_{AB} \simeq p^{0}_{AB} + \Sigma \sigma_k p^{k}_{1}$$
$$R_{AB} \simeq R^{0}_{AB}(I + \Sigma \rho_k M^{k}_{1})$$

and

$$p_{AB} \approx p^0_{AB} + \Sigma \lambda_k p^k_2$$
$$R_{AB} \approx R^0_{AB}(I + \Sigma \mu_k M^k_2)$$

We will get additional constraints:

$$\Sigma \lambda_k [(R^0_{AB})^{-1} p^k_2]_x - \Sigma \rho_k [(R^0_{AB})^{-1} p^k_1]_x = 0$$
$$\Sigma \lambda_k [(R^0_{AB})^{-1} p^k_2]_y - \Sigma \rho_k [(R^0_{AB})^{-1} p^k_1]_y = 0$$
$$\Sigma \lambda_k [(R^0_{AB})^{-1} p^k_2]_z - \Sigma \rho_k [(R^0_{AB})^{-1} p^k_1]_z = 0$$
$$\Sigma \mu_k (M^k_2)_{3,2} - \Sigma \sigma_k (M^k_1)_{3,2} = 0$$
$$\Sigma \mu_k (M^k_2)_{1,3} - \Sigma \sigma_k (M^k_1)_{1,3} = 0$$
$$\Sigma \mu_k (M^k_2)_{2,1} - \Sigma \sigma_k (M^k_1)_{2,1} = 0$$

7.9 Algorithms

This section summarizes algorithms for using the methods discussed in the previous section. We wish to calculate either the location T_{AB} or the determination ΔT_{AB} of an object B with respect to an object A. The algorithm below uses available constraint information about individual object-object relations to produce a set of scalar variables δ_i corresponding to all degrees of freedom between A and B. Using the assumption that the δ_i are suitably small, it produces a set of linear constraints

$$A\bar{\delta} \; ? \; b$$

linking the variables. Linear programming is then used to compute the limits of relevant quantities. Briefly, the steps followed are as follows:

1. Find all acyclic chains of relations joining A and B. For each path

   $$(A, N_1, \ldots, N_k, B)$$

   we get an equation of the form

   $$T_{AB} = T_{AN_1} * \cdots * T_{N_k B}$$

   In general, each T_{PQ} will have an associated set of free variables

$$p_{PQ} \tilde{=} p^0_{PQ} + \Sigma \lambda_i p^i_{PQ}$$
$$R_{PQ} \tilde{=} R^0_{PQ}(I + \Sigma \mu_i M^i_{PQ})$$

with limits on each λ_i and μ_i

$$\lambda_i^{min} \le \lambda_i \le \lambda_i^{max}, \quad \mu_i^{min} \le \mu_i \le \mu_i^{max}$$

together with additional constraint relations of the general form

$$\hat{g} R_{PQ} f \; ? \; d - \hat{g} p_{PQ}$$
$$\hat{g} R_{PQ} f \; ? \; 0$$
$$0 \; ? \; d - \hat{g} p_{PQ}$$

If we are interested in a location estimate, this suffices. If determination is desired, then we get

$$\Delta T_{AB} = T_{N_k B} * \ldots * T_{A N_1} * \Delta T_{N_1} * \ldots * T_{N_k B} \Delta T_{N_k B}$$

with corresponding additional free variables and constraints for the ΔT_{PQ}. Relations with negligible error will have $\Delta T_{PQ} = I$.[16]

Each equality relation will be simplified algebraically as far as possible. The rules currently used for this include:

a. Constant elimination.

b. $T_{PQ} * T_{PQ}^{-1} = T_{PQ}^{-1} * T_{PQ} = I$

c. $T_{PQ} * I = I * T_{PQ} = T_{PQ}$

2. Once all the path equations have been found and simplified, produce the corresponding set of linear constraints involving the free variables of the various relations. For limiting values on individual degrees of freedom,

$$\epsilon_i^{min} \le \delta_i \le \epsilon_i^{max}$$

this is trivial. Constraint forms expressed in terms of R_{PQ} and p_{PQ} are

---

[16] Here, it is worth pointing out that taking account of *all* paths between A and B tells you the *potential* accuracy with which T_{AB} can be determined. To get this accuracy at runtime, you have to synthesize a calculation to use the corresponding runtime values. Considering only one path gives you the accuracy of that particular calculation chain. One strategy would be to evaluate each possible chain and then specify which one is to be used to compute the value you are interested in. Section 7.11.3 will discuss an alternative approach.

translated by multiplying and simplifying. Thus, wherever $\hat{g} \cdot R_{PQ} f$ appears, we substitute

$$\begin{aligned} \hat{g} \cdot R_{PQ} f &\approx \hat{g}(R^0_{PQ}(I + \Sigma \mu_i M^i_{PQ}))f \\ &= \hat{g} \cdot R^0_{PQ} f + \Sigma \mu_i \hat{g} \cdot R^0_{PQ} M^i_{PQ} f \\ &= \alpha_0 + \Sigma \alpha_i \mu_i \end{aligned}$$

where

$$\begin{aligned} \alpha_0 &= \hat{g} R^0_{PQ} f \\ \alpha_i &= \hat{g} R^0_{PQ} M^i_{PQ} f \quad (i > 0) \end{aligned}$$

Similarly, $\hat{g} \cdot p_{PQ}$ is translated

$$\begin{aligned} \hat{g} \cdot p_{PQ} &\approx \hat{g} \cdot (p^0_{PQ} + \Sigma \lambda_i p^i_{PQ}) \\ &= \hat{g} \cdot p^0_{PQ} + \Sigma \lambda_i \hat{g} \cdot p^i_{PQ} \\ &= \beta_0 + \Sigma \beta_i \lambda_i \end{aligned}$$

Equality of two frame expressions is handled by forming the appropriate set of six constraint forms, as discussed in the previous section, and then applying the substitutions just mentioned to the resulting expressions.

3. Once the constraint equations have been translated into linear form, then linear programming techniques are applied to find limits of appropriate objective functions.

Appendix E.2 gives an example of the workings of this method.

7.10 Experience

The methods described in Section 7.9 worked very well, even for problems involving many free variables. Since all rotation angles are assumed to be small, extreme values of perturbations can be found without solving many sub-problems, as was required by the iterative method discussed in Section 7.6.5.

On the other hand, this assumption limits the applicability of the method to those cases – principally error analysis or small variations in fixturing – for which the rotational uncertainty is, in fact, small. In order to handle larger perturbations, we would need to resort to iteration, or else to more powerful numerical methods better adapted to non-linear problems. Fortunately, we can handle a very large subset of interesting problems with the machinery developed here. Chapter 8 will illustrate the use of these techniques as a basis for reasonable coding decisions, and Section 7.11 will describe some additional application possibilities.

7.11 Other Uses of Differential Approximation

7.11.1 Sensitivity Analysis

The differential approximation gives a measure of how sensitive a relation is to each individual parameter. Furthermore, a useful by-product of our numerical solution method is information on which constraints are binding at limit points and on how sensitive the limit points are to perturbations in the individual constraints. This information can be useful in a number of ways.

Design and Use of Fixtures

One purpose of a fixture is to facilitate task performance by introducing an intermediate assembly step that requires lower initial precision than does the final operation. The fixture must be designed to hold the parts with sufficient accuracy for the task to be completed. On the other hand, if it is made too "tight", it may be difficult to use or to fabricate. The methods discussed above provide a useful way to evaluate how much particular design parameters of the fixture affect the performance of the fixture. Alternatively, they can suggest limits on parameters imposed by design requirements.

For instance, suppose we are using aligning pins to hold a cover plate in place while it is secured with screws. In order for the screws to be inserted successfully, we require that the holes in the cover plate and box line up to within some small error. This requires that the aligning pins be big enough to hold the plate to a desired accuracy. On the other hand, if the pins are made too big, they will tend to bind on the sides of the holes. We are interested in picking a pin size that is both easy to insert and that does the job.

Let

H_C = transl(h_C) = location of cover hole wrt cover.
H_B = transl(h_B) = location of box hole wrt box.

T_{match} = location of cover hole wrt box hole when parts are in position
= $(Box*H_B)^{-1}*(Cover*H_C)$
= $H_B^{-1}*Box^{-1}*Cover*H_C$
= $transl(-h_B)*T_{BC}*transl(h_C)$

$p_{match} = -h_B+p_{BC}+R_{BC}h_C$
$R_{match} = R_{BC}$

If the holes have diameter d_h and the aligning pins have radius d_p, then, the xy error of the match will be less than (d_h-dp).

$$-(d_h-d_p) \le p_{match}\hat{a} \le (d_h-d_p)$$

for all $\hat{a}$ in the xy plane. Depending on the size of (d_h-d_p) and the precision desired, a suitable number of vectors $\hat{a}$ can be chosen to approximate this constraint. Generally, four constraints

$$-(d_h-d_p) \le p_{match}\cdot\hat{x} \le (d_h-d_p)$$
$$-(d_h-d_p) \le p_{match}\cdot\hat{y} \le (d_h-d_p)$$

will be enough. In our example, T_{BC} is given by

$$p_{BC} \approx z_0\hat{z}+\lambda\hat{x}+\mu\hat{y}$$
$$R_{BC} \approx I + \omega M_z$$

Our two aligning pins will give us the constraints

$$-(d_h-d_p) \le \lambda - \omega(h_1)_y \le (d_h-d_p)$$
$$-(d_h-d_p) \le \mu + \omega(h_1)_x \le (d_h-d_p)$$
$$-(d_h-d_p) \le \lambda + \omega(h_2)_y \le (d_h-d_p)$$
$$-(d_h-d_p) \le \mu + \omega(h_2)_x \le (d_h-d_p)$$

Given these constraints, we can evaluate the potential misalignment at the target hole h_t.

$$\Delta h = -h_t + p_{BC} + R_{BC}h_t$$
$$\approx \lambda\hat{x}+\mu\hat{y} + \omega M_z h_t$$

to determine if the maximum deviations of Δh fall comfortably within acceptable limits. Alternatively, we can express the requirements on Δh as constraints

$$-\epsilon_h \le \Delta h\cdot\hat{x} \le \epsilon_h$$
$$-\epsilon_h \le \Delta h\cdot\hat{y} \le \epsilon_h$$

where

$$\epsilon_h = 0.707*\text{<maximum acceptable misalignment>}$$ [17]

We can then compute the smallest pin diameter d_p that meets all the constraints. When this value is determined, we can look at which constraints were binding at that time. By mapping the constraints back into their corresponding semantics, we determine which aspects of the task are responsible for requiring that a particular size pin be used. This may suggest a suitable modification of the task.

---

[17] The 0.707 is necessary to correct for our approximating a circle with a square. If more constraints are employed, then a fudge factor closer to 1 may be used.

Specification of Object Dimensions and Tolerences

There is nothing about the preceding analysis that restricts it to use with design of fixtures. When one designs an object, it is not uncommon to worry about clearances, etc., which may be affected by small perturbations in design parameters. This is especially true where manufacturing tolerences are being specified. For instance, in our previous example, errors in the location or diameter of the holes can make the box impossible to assemble. Similarly, consider the example illustrated in Figure 7.8.[18] Here, the important design consideration is that the gears mesh properly. Whether this actually will happen is dependent on a number of individual tolerences that can combine in many ways. A great temptation, when one is faced with situations of this nature, is to "brute force" the problem. I.e., to specify all, or most, of the tolerences involved somewhat more tightly than actually required. Since overdesign may increase the cost of the product substantially, an interactive system that identifies critical tolerences and evaluates proposed specifications could be a very useful and cost-effective tool.

An important additional point here is that the constraint equations tell us how a set of individual errors work *together* to produce a particular effect. In general, these errors will not all be independent; their relation depends on the particular processes used to manufacture the object. For example, all four holes in our box example may be gang drilled in a single operation. Similarly, both bearings for each shaft in Figure 7.8 could well be drilled in the same operation. In this case, the alignment errors in the shafts would be considerably less than if the position errors of each bearing hole were independent. This suggests that the *functional* design include only those tolerences that are critical to the actual function of the part, with remaining parameters being fixed when decisions about how the part is to be manufactured are made.

Planning for Measurements.

We have already discussed the importance of error estimates in deciding whether a particular technique will work. Once it has been established that more accuracy is required, the question arises of how to obtain the necessary data. Generally, there will be a number of object features whose positions can be measured and a number of different techniques available to measure each location. These measurements vary in difficulty, in accuracy, and in the contribution made to reducing critical tolerances. The goal is to select a minimum cost set of measurements that produces the desired accuracy. The techniques developed in this chapter are useful, because they allow us to assess the effectiveness of a given set of measurements and provide a measure of the sensitivity of critical quantities to particular measurements.

7.11.2 Vision

Many of these techniques are directly applicable to "verification vision" systems, such as are being investigated by Bolles. In verification vision,

> the system knows the identity of all objects in the scene and approximately where they are; the goal is to determine the precise location of one or more

---

[18] A number of similar examples may be found in Fortini's book on dimensioning [40].

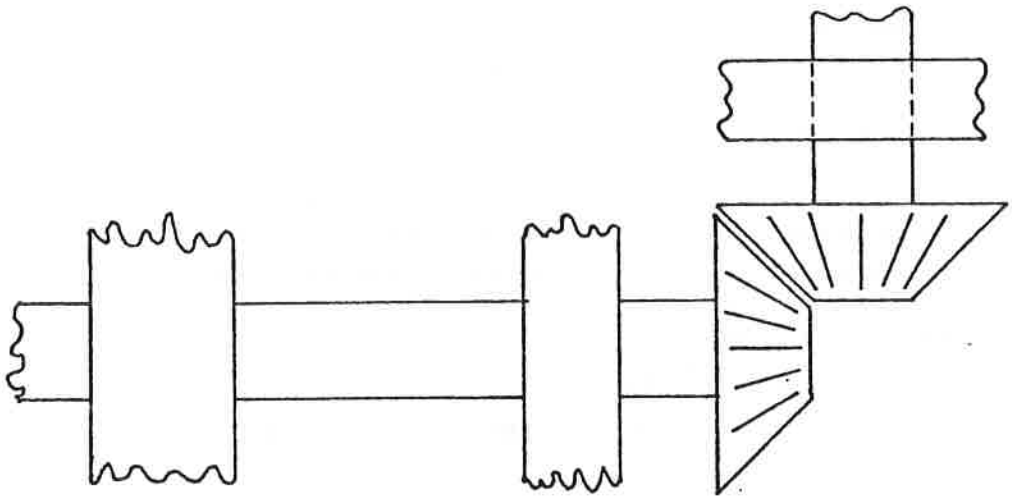


Figure 7.8. Gears Must Mesh

of the objects.[19]

This is exactly the situation that one encounters in a very large class of assembly and other manipulation tasks. Typical visual applications in such tasks include:

1. Determine the position of an object with sufficient accuracy so that it may be grasped and inserted into a jig.

2. Find the end position and direction of a shaft over which a nut is to be placed.

3. Verify that a screw is indeed held at the end of a screwdriver.

4. Find the displacement between a screw tip and the center of the hole into which the screw is to be inserted.

An essential requirement for a visual system that seeks to perform this kind of thing is the ability to set bounds on possible positions of features in the picture[20] and to relate small changes in object locations to corresponding changes in picture locations. The formalisms discussed in this chapter provide this capability.

---

[19] This definition was given by Bolles [22].

[20] The ability to predict the appearance of objects is also quite useful, but its lack may be compensated to a great extent by the use of training pictures.

Suppose we have a feature point q (for instance, the center of a hole, a corner of the object, or what-have-you) defined with respect to an object B,

$$q^{world} = B*q$$

Suppose, further, that we have pointed a camera so that q appears on the image at point (u,v). This constrains the possible values of the relation T_{CB} relating the camera location C to B. The location of q with respect to the camera will be given by

$$\begin{aligned} q' &= T_{CB}q \\ &= p_{CB} + R_{CB}q \\ &\simeq p^0_{CB} + \Sigma \lambda_k p^k_{CB} + R^0_{CB}q + \Sigma \mu_k R^0_{CB} M^k_{CB} q \\ &\simeq q_0' + \Sigma \eta_k q_k' \end{aligned}$$

where the q_k' and the η_k are computed from q and T_{CB} in the usual way. (If desired, manufacturing errors in the position q can, of course, be added in) If the focal length of the camera is given by f, then

$$u = \alpha q' \cdot \hat{x}$$
$$v = \alpha q' \cdot \hat{y}$$

where

$$\alpha = f / (\hat{z} \cdot q')$$

For small values of η_k, we can approximate $\alpha q'$ by

$$\alpha q' \simeq (f/d)(q_0' + \Sigma \eta_k (q_k' - (\hat{z} \cdot q_k'/d) q_0'))$$

where

$$d = \hat{z} \cdot q_0'$$

The previous sections have shown how we can produce a set of (linear) constraints limiting the values of the η_k. It is now a straightforward process to compute the extreme values of u and v, thus giving a rectangle in which the feature q may appear. If desired, extreme values for linear combinations of u and v

$$\zeta = \sin\tau\, u + \cos\tau\, v$$

can be computed for various values of τ to produce a more closely cropped "window" for (u,v). [21]

---

[21] Here, we are ignoring the important problem of whether or not the feature can become obscured as object locations are perturbed. This is still largely an unsolved problem for

The system works by applying various operators to locate features in the image. The result of each such operator is to constrain the feature found to lie within a cone or pyramid whose apex is at the lens center and whose width is determined by the accuracy of the operator. The mathematical constraints associated with this pyramid are easily derived. For instance, suppose that a hole drilled into a box, B, is found to lie at $(u_0 \pm \Delta u, v_0 \pm \Delta v)$. For simplicity, let's assume that the camera's position C with respect to the table is known exactly, there is no manufacturing error in the position, h, of the hole, and that the box location with respect to the table is given by

$$B^{world} = Table \cdot B^0 \cdot \Delta B$$

where

$$\Delta p_B = \lambda \hat{x} + \mu \hat{y}$$
$$\Delta R_B = Rot(\hat{z},\omega) \cong I + \omega M_z$$

so

$$T_{CB} = C^{-1} \cdot B^0 \cdot \Delta B$$

$$p_{CB} = C^{-1} p_B^0 + C^{-1} R_B^0 \Delta p_B$$
$$= C^{-1} p_B^0 + \lambda C^{-1} R_B^0 \hat{x} + \mu C^{-1} R_B^0 \hat{y}$$

$$R_{CB} = R_C^{-1} R_B^0 \Delta R_B$$
$$\cong R_C^{-1} R_B^0 (I + \omega M_z)$$

then the location h' of the hole with respect to the camera will be given by

$$h' = T_{CB} h = p_{CB} + R_{CB} h$$
$$\cong C^{-1} p_B^0 + R_C^{-1} R_B^0 h + \lambda C^{-1} R_B^0 \hat{x} + \mu C^{-1} R_B^1 \hat{y} + \omega R_C^{-1} R_B^0 M_z h$$
$$= h_0' + \lambda h_\lambda' + \mu h_\mu' + \omega h_\omega'$$

The (linearized) constraints are then

$$u_0 - \Delta u \le \hat{x} \cdot (f/d)(h_0' + \lambda h_\lambda' + \mu h_\mu' + \omega h_\omega') \le u_0 - \Delta u$$
$$v_0 - \Delta v \le \hat{y} \cdot (f/d)(h_0' + \lambda h_\lambda' + \mu h_\mu' + \omega h_\omega') \le v_0 - \Delta v$$

---

cases where objects are free to move about. One solution might be to use computer graphics techniques together with systematic or random variations in the parameter values. Alternatively, it may sometimes be possible to express constraints on object location which constitute necessary and/or sufficient conditions for visibility of the feature. These constraints could then be verified directly by the mathematical programming machinery.

These constraints can be combined with other sources of information and then used at planning time, in a manner analogous to that discussed earlier, to evaluate proposed operations or sequences for their effectiveness in pinning down the object's location. Similarly, the system of constraints can be carried over to runtime and solved, using the techniques discussed in Section 7.8, to produce actual location values.

7.11.3 Runtime Updating

One of the major deficiencies of affixment is that, while it provides an excellent means of reducing the bookkeeping required when an object location is modified, it is not particularly good as a means for establishing object locations, based on measurements of feature locations. The problem, which we discussed in Chapter 3, is that affixment only operates on the basis of the most recent change, while we need to combine information from several measurements in order to compute an object location. The constraint formulations discussed in this chapter provide a mechanism for doing just this.

As we have seen, an object A will in general be related to another object B by a series of equations of the form

$$T_{AB} = T_{AN_1} * \ldots * T_{N_k B} \qquad \text{[Eq 4]}$$

where the N_i are various intermediate objects and features. Further, the current value of any relation variable T_{AB} will be inaccurate by some amount ΔT_{AB}.

$$T_{AB}^{actual} = T_{AB} * \Delta T_{AB}(\nu_1, \ldots, \nu_m)$$

where the ν_i correspond to various rotational and translational errors, about which something may be known *a priori*., and which are further linked by the [Eq 4] equations. The act of taking a "measurement" merely adds another equation or set of constraints linking the object or feature measured to the reference frame of the measuring device, perhaps with the addition of additional error variables. For instance, in the previous section, the hole-finder produced a set of constraints relating the hole to the camera's coordinate system. Thus, the problem of updating location variables, given the results of a number of measurements, reduces to one of finding some "best" set of values ν_i^* for the ν_i that satisfy all the constraints. Once such a set is found, it may be used to produce values for the ΔT_{AB}, which may then be used to update the corresponding variables T_{AB}.

If the various errors can be assumed to be suitably small, we can produce a set of *linear* constraints

$$\zeta_j(\bar{\nu}) = \hat{a}_j \cdot \bar{\nu} - d_j \leq 0$$

which can be used to compute several sorts of "best" values for $\bar{\nu}$. For instance,

1. *A Chebychev Point*.. I.e., a $\bar{\nu}^*$ with the property that it gives

$$\xi(\bar{\nu}^*) = \max_j \zeta_j(\bar{\nu}^*) = \min_{\nu} \max_j \zeta_j(\bar{\nu})$$

which may be found easily by inventing a new variable γ and solving the linear programming problem [22]

$$\xi(\bar{\nu}^*) = \min \gamma$$

such that

$$\zeta_j(\bar{\nu}) = \hat{a}_j \bar{\nu} - d_j \leq \gamma$$

2. *A least squares point.* I.e., a point $\bar{\nu}^*$ that minimizes a quadratic form

$$\bar{\nu} Q \bar{\nu}$$

Here, any of several quadratic programming algorithms would be applicable.

The principal problem in using this method is that one needs to keep the differential terms around. If the approximate values for the location variables are known at planning time, these may be precomputed. This method is acceptable for minor adjustments, but clearly breaks down if big rotations can happen. The alternative is to do differentiation of the [Eq 4] equations at runtime, at least in cases where large or additive corrections will happen. In this case, once an initial solution for $\bar{\nu}^*$ has been calculated and the correction applied, a new linear approximation for $\Delta T_{AB}(\bar{\nu})$ would be computed. The process could then be iterated several times until a sufficiently good set of ν_i are found.

Another problem is that the mathematical programming problems may become needlessly encumbered with many superfluous constraints, that will never be binding. Possible solutions to this include (1) using algorithms that are not slowed down by many extra constraints, or (2) doing analysis ahead of time to determine which constraints to use. This latter approach, of course, is the same thing that was discussed in Section 7.11.1 on planning for measurements.

---

[22] See, e.g, [116] for details.

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

Bullwinkle Moose

Chapter 8.

Automatic Coding of Program Elements

In Section 3.4, we described the process of writing AL code for a common subtask in assembly operations – insertion of a pin into a hole. In the discussion, 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. Subsequent chapters have been concerned largely with techniques for representing the necessary information in a form "understandable" by a 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 3.4

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 then computed, using expected motion time as an objective function.[1] 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 sufficiently 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 a fairly straightforward matter to generate the corresponding AL code sequences, which are quite stylized.

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

8.1 Data Structures

Internally, strategies are represented 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

preliminaries – 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.

dtry – 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 and approach symmetry, together with information about the expected penetration distance into the hole constitute sufficient information to write a "first order" program that ignores errors, such as was produced in Section 3.4.4. 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 8.8 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, ζ – 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 ζ about the hole axis. See Figure 8.1

$\Delta\theta$ – 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.

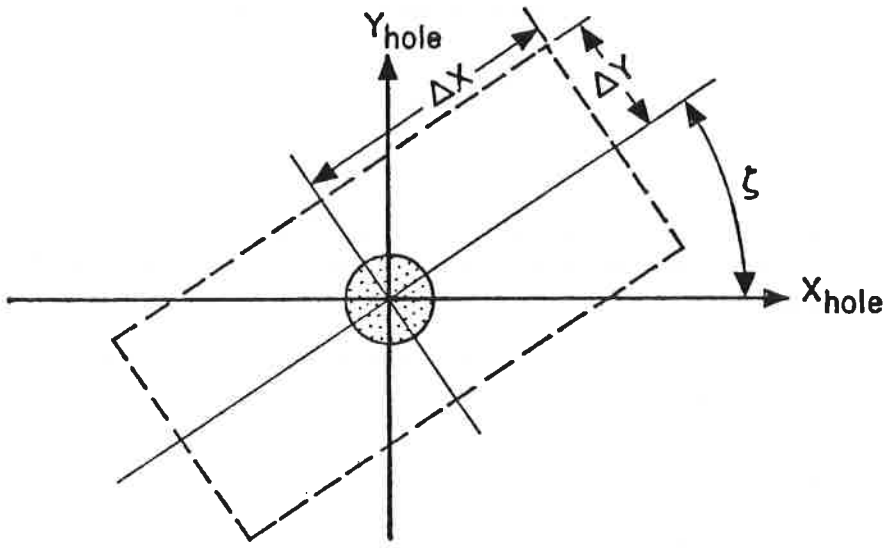


Figure 8.1. Error Footprint

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

$a \leftarrow b$;

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:

what – 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 8.7 will describe how sequences of such "destination points" may be turned into motion statements.

8.2 Initial Computations

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 of Chapter 7 we get

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

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 ("sticking distance") – the maximum 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 (1) 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, d_f 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.

8.3 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 will create a "grasp strategy" record, as described in Section 8.1. Although we shall 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 are:

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 long 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 d_f 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 the "standard" direction for each pin type.

[5] Whitney [75] has done an exhaustive analysis of some of the factors required to compute tolerance requirements for insertion of a peg into a hole.

8.3.1 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. Here, 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).

8.3.2 Grasping Position

The key element in our grasping strategy is, of course, where to grasp the pin. Here, the structure of the hand must be considered. 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.[6] The important parameters remaining are thus:

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

d ("grasp distance") – the 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] Sensitive force sensors for the fingers are currently being built. When this new hardware is completed, 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 relax the accuracy requirements required to ensure that the microswitches contact the object being picked up.

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

Clearly, it is a bad idea to specify a grasping position that cannot be reached by the arm; it will be 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 will 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 something 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, d_i and d_f, that the pin

---

[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 GEOMED [9]) 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.

goes into the initial and final holes. There are two subcases:

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

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 pointed tip, then we must grasp further down the shaft:

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

where

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

l_{pin} = length of pin.

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.

$d_1 = d_{max}$ - 0.6 inches

$d_2 = d_{min}$ + 0.6 inches

$d_3 = (d_{min}+d_{max})/2$

For each value of d, the system must generate values for γ. Currently, 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

---

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

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.

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

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

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 γ 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 8.8, 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 evaluating the 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 8.4).

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 should become a significant problem, it would be fairly easy to provide a "try harder" mode where all possibilities are retained.

8.3.3 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 (see Figure 8.2) 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 a fairly easy matter to generate a set of alternative departure points, and then pick the one giving the shortest motion time.

8.3.4 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]

---

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

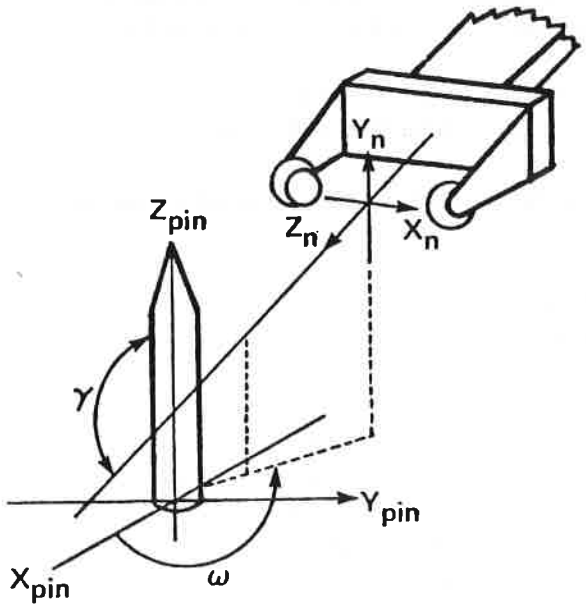


Figure 8.2. Approaching the Pin

8.4 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 will assume throughout this discussion that the origin of the pin coordinate system is at the tip being inserted into the hole. The modifications if it is at the other end are obvious, but would only confuse the discussion. 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)>8*oz do ...
```

where

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:

$$hand\_destination = pin\_destination * grasp\_xf$$
$$= hole * \text{trans}(\text{rot}(\text{zhat}, \phi, \text{vector}(0,0,-d_f)) * grasp\_xf;$$

For our example situation (Section 8.8) 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,

$dtry = d_f$ + 1 inch
standoff = 1 inch

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

$$pin = hole * \text{trans}(\text{rot}(\text{zhat}, 90 * \text{deg}), \text{vector}(0,0,2.54));$$

and

$$pin = hole * \text{trans}(\text{rot}(\text{zhat}, 90 * \text{deg}), \text{vector}(0,0,-4.25));$$

respectively.

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.

8.5 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 of Chapter 7 to estimate the error between the pin and hole at the approach point as a function of free variables:

$$\Delta p_{hp} = \Sigma\, \delta_k p^k_{hp} \qquad \text{[Eq 1]}$$
$$\Delta R_{hp} = I + \Sigma\, \epsilon_k M^k_{hp}$$

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 (Δz) 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.

8.5.1 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 the system [Eq 1] to compute the maximum expected misalignment.

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

where

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

At present, we consider six values of ζ_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:

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

where M_x, M_y, and M_z are the matrices defined in Section 7.8. The constraints on the free variables are:

$$-5 \text{ deg} \le \nu \le 5 \text{ deg}$$
$$-0.25 \text{ deg} \le \eta_x \le 0.25 \text{ deg}$$
$$-0.25 \text{ deg} \le \eta_y \le 0.25 \text{ deg}$$
$$-0.25 \text{ deg} \le \eta_z \le 0.25 \text{ deg}$$

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

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

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

Once this value is computed, we compare it to the allowable 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 to greater accuracy.

---

[13] These parameters correspond to the best overall strategy found in Section 8.8.

8.5.2 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 Δz 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 Δz 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.

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:

$s \leftarrow$ surface into which the hole is drilled;
$(x_h, y_h) \leftarrow$ location of hole in coordinate system of surface;
$r_{td} \leftarrow$ radius of hole + radius of pin tip;
$\Delta r_{td} \leftarrow \max(0.3 \text{ inches}, \Delta x_{hp}, \Delta y_{hp})$
$maxr \leftarrow$ maximum distance of any point on s from the hole;
$dbest \leftarrow 0$;

---

[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.

```
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
            begin 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

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

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 to 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,

Δr_{td} = max (*0.3 inches*, Δx, Δy)
= max (*.762 cm*, *.243 cm*, *.226 cm*)
= *.762 cm*

On the first iteration through our outer (*r*) 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

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

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, so, can stop looking. The corresponding tapping point is:

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

Once such a point has been found, then Δz 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.

8.5.3 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

$$\zeta_k = 30k \text{ degrees}$$
$$0 \le k \le 5$$

Then, we take

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

This produces an "error footprint" rectangle with sides $2\Delta x$ and $2\Delta y$, rotated by ξ with respect to the hole, as shown in Figure 8.1 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 errors are the same as those given in Section 8.5.1. In addition, the rotation errors are as previously stated and 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.

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

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 , .000 , .000 ] . V1 ≤ .127
[ 1.00 , .000 , .000 , .000 , .000 , .000 ] . V1 ≥-.127
[ .000 , 1.00 , .000 , .000 , .000 , .000 ] . V1 ≤ .127
[ .000 , 1.00 , .000 , .000 , .000 , .000 ] . V1 ≥-.127
[ .000 , .000 , 1.00 , .000 , .000 , .000 ] . V1 ≤ .127
[ .000 , .000 , 1.00 , .000 , .000 , .000 ] . V1 ≥-.127
[ .000 , .000 , .000 , 1.00 , .000 , .000 ] . V1 ≤ .436e-2
[ .000 , .000 , .000 , 1.00 , .000 , .000 ] . V1 ≥-.436e-2
[ .000 , .000 , .000 , .000 , 1.00 , .000 ] . V1 ≤ .436e-2
[ .000 , .000 , .000 , .000 , 1.00 , .000 ] . V1 ≥-.436e-2
[ .000 , .000 , .000 , .000 , .000 , 1.00 ] . V1 ≤ .436e-2
[ .000 , .000 , .000 , .000 , .000 , 1.00 ] . V1 ≥-.436e-2
[ 1.00 , .000 , .000 ] . V2 ≤ .762
[ 1.00 , .000 , .000 ] . V2 ≥-.762
[ .000 , 1.00 , .000 ] . V2 ≤ .508
[ .000 , 1.00 , .000 ] . V2 ≥-.508
[ .000 , .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, Δx = 1.50 cm, Δy = 1.15 cm, and ζ = 60 degrees.

If Δr 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 8.8.

If a search is required, 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 obtained 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.

8.6 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.
        end;

shortest_time ← 10^39 seconds; { a long time }
best_strategy ← incantation;
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.

8.7 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 looks 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);
    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;[17]
```

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 8.8). Similarly, in

---

[16] "Carriage Return, Line Feed"

[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. The reasons for this particular renaming have been lost in antiquity, and are of only historical interest, in any event. The only reason for mentioning the matter is to prevent careful readers from being confused by the computer generated program given in Section 8.8.

deciding whether to emit code for a search, it looks to see if Δx 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 8.8.

Motion Sequences

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

```
procedure write_motion_sequence(list 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;
      comment 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);
      print(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. This approach has several possible pitfalls, since the semantics of two successive motion statements are not identical to a single statement, 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 "hair" and makes 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

---

[18] Recall that in Section 8.5.3, we selected ζ so that $\Delta x \geq \Delta y$.

AL compiler to worry about the relation to manipulator frames.[19]

8.8 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 T_{bh} with respect to the box, the pin (*pin1*) is held in a tool rack at T_{wp}, and the manipulator (bmanip) is parked at bpark, where

T_{wb} = trans(nilrotn, vector(*45.2, 102., 0*))
T_{bh} = 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
$\Delta\theta_{ok}$ = *10* 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,

d_{grasp} = 3.54 cm

and a range of grasp angles

100 degrees ≤ γ ≤ 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

---

[19] Such an approach is a natural extension to the present translation performed by the AL compiler, which was discussed in Section 5.4.2.

is shown below:[20]

```
PHL SPEC 132757
  PRELIMS: NULL_RECORD
  PICKUP: PICKUP SPEC 162775
   PRELIMINARIES: NULL_RECORD
   APPROACH OPENING:  2.98
   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))
   GRASP DETERM:  NILTRANS
   DEPARTURE POINT: PIN1=PIN1*TRANS(NILROTN,VECTOR(0,0,6.79))
   GOODNESS:  4.13
   REMARKS:  W =  90.0 deg  Grasp Angle =  135. deg  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:  .000
  PH FP DX:  .000
  PH FP DY:  .000
  PH FP ROT:  .000
```

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

ω = 90 degrees
γ = 135 degrees
grasp distance = 3.54 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 strategy just shown, we get

Δz = .180 cm
Δx = 1.50 cm
Δy = 1.15 cm
ζ = 60 degrees

The value of Δz 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:

---

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

```
PHL SPEC 134023
  PRELIMS: NULL_RECORD
  PICKUP: PICKUP SPEC 163075
    PRELIMINARIES: NULL_RECORD
    APPROACH OPENING:  2.98
    APPROACH: BMANIP=PIN1*TRANS(ROTN(VECTOR(.608,.608,.510),126.*DEG),VECTOR(-5.,0,4.42))
    GRASP OPENING:  .185
    GRASP: BMANIP=PIN1*TRANS(ROTN(VECTOR(.608,.608,.510),126,*DEG),VECTOR(0,0,3.54))
    GRASP DETERM:  NILTRANS
    DEPARTURE POINT: PIN1=PIN1*TRANS(NILROTN,VECTOR(0,0,6.79))
    GOODNESS:  4.08
    REMARKS:  W = 90.0 deg  Grasp Angle = 180. deg  Grasp Distance = 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
  GOODNESS:  6.64
  TAP: NULL_RECORD
  FINE TIME:  .000
  PH DZ:  .000
  PH FP DX:  .000
  PH FP DY:  .000
  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 130523
  PRELIMS: NULL_RECORD
  PICKUP: PICKUP SPEC 72027
    PRELIMINARIES: NULL_RECORD
    APPROACH OPENING:  2.98
    APPROACH: 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))
    GRASP DETERM:  NILTRANS
    DEPARTURE POINT: PIN1=PIN1*TRANS(NILROTN,VECTOR(0,0,6.79))
    GOODNESS:  4.16
    REMARKS:  W =  315.     deg  Grasp Angle =  180.     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 DX:  1.50
  PH FP DY:  1.15
  PH FP ROT:  1.05
```

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]

---

[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 "0.00106", instead of "0.001".)

```
! PIN-IN-HOLE STRATEGY 130523:
  DROK = .762      FPX = 1.50      FPY = 1.15      FPW = 1.05
  DZ = .127       ESTIMATED TIME = 6.14 !

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

OPEN BHAND TO 2.98;
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) > .169 THEN
    BEGIN !PIN1 MISSED BH1 !
    BOOLEAN FLAG;
    ! SEARCH LOOP: !
    REAL R,DW,W,X,Y;FLAG←FALSE;
    R ← .572; ! 0.75*DROK !
    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))< 1.50 AND ABS(Y←R*SIN(W))< 1.15 THEN
          BEGIN FRAME SETPNT;
          SETPNT←BH1*TRANS(NILROTN,ROT(ZHAT,1.05)*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,-5.08));
```

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,θ)*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 penetratiion of the pin into the hole. If we 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 134356;
  DROK = .762      FPX = .243      FPY = .226      FPW = .524
  DZ = .188       ESTIMATED TIME = 6.67    !

! PICKUP 147157;
  W = 90.0 deg  Grasp Angle = 135. deg  Grasp Distance = 3.54 !

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(0,0,6.79));

! MUST TAP !

MOVE PIN1 TO BH1*TRANS(NILROTN,VECTOR(-1.21,-.002,-5.00))
    VIA BH1*TRANS(NILROTN,VECTOR(-1.21,-.002,5.00))
      ON FORCE(ORIENT(PIN1)*ZHAT) > 8*OZ DO STOP
      ON ARRIVAL DO ABORT("EXPECTED A FORCE HERE");
CORR ← ZHAT . INV(BH1* TRANS( NILROTN, VECTOR(-1.21,-.002, .000)))*DISPL(PIN1);
```

---

[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 required to compute a target point.

```
! 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)-CORR;
IF ABS(DISTANCE_OFF) > .239 THEN
    ABORT("PIN1 MISSED BH1 UNEXPECTEDLY.");

! 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));
```

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.

"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 9.

Conclusions and Future Work

The goal of this research was the establishment of a basis for computer generation of manipulator control programs from task-level specifications. As we have seen, this is an extremely broad topic. Many narrowing assumptions have been necessary in order for us to demonstrate basic feasibility while keeping the scope of effort within reasonable bounds for a single dissertation.

The task of inserting a pin into a hole was used to focus the technical issues of automating manipulator coding decisions. The analysis of Section 3.4 discussed this task from the point of view of a human programmer. Skeleton techniques for accomplishing the task and for overcoming likely errors were developed, and the planning information needed to adapt these skeletons to actual situations was identified. Subsequent chapters were largely concerned with developing techniques for representing this information is a form usable by the computer. Finally, Chapter 8 showed that we could, in fact, automate these coding decisions, given the modelling mechanisms developed in Chapters 4-7.

Section 1.2 listed seven requirements for development of a "full" automatic coding system for mechanical assembly applications. In conclusion, I'd like to go back through this list and make some final comments summarizing what has been done with respect to each point, what remains to be done, and where future efforts should be concentrated.

Development of an Adequate Manipulator-Level Target Language.

Although a complete design review of the target language, AL, used in this work would be inappropriate here, several points are worth noting.

First, the language seems to be adequate for at least some assemblies. The salient features – **frame** motion and **affixment** – significantly reduce the difficulty of programming and enhance the robustness of the resulting code. The force control and sensory primitives are still rather crude but do allow the simple threshold tests needed for the class of tasks addressed here. Likewise, the control structures allow for considerable runtime flexibility and error recovery.

An important area for future development is the incorporation of more sophisticated forms

of force accommodation and the provision of better links with other forms of sensory data, such as vision.[1] Such development will require improvements both in the runtime system and in ways to describe how the runtime capabilities are to be applied.

A second point is that AL performs a number of coding functions that require the maintenance of a much better planning model than is usual for "algebraic" compilers. The uses to which this model are put and the problems encountered in maintaining it were discussed in Chapter 5. Problems encountered and some possible solutions were also discussed. The basic point is that "understanding" a computer program can be a *very* hard problem, which isn't apt to be solved fully for some time yet. It would probably be desirable to defer some of the coding decisions now made by AL – especially trajectory planning – until execution time. Such a deferral would substantially simplify translation of *present* AL programs, since planning values for trajectory start and destination points would no longer be needed. However, we hope to add *other* facilities – such as collision avoidance – which will, in fact, require *more* planning information.

In this regard, the automatic coding primitives discussed in Chapter 8 are somewhat better off: since they "know" the purpose behind the program structures they write, they will have less difficulty in updating the planning model appropriately. Indeed, it may be possible to use a task-level specification as a guide to keeping planning information even in cases where the computer isn't smart enough to write the code on its own.

Finally, there is the question of whether the use of AL as a target language isn't perhaps overkill. If an automatic system takes over the production of manipulator-level code, are fancy features like affixment or implicit specification of manipulators still useful? Would it not be better to eliminate them and save the extra overhead? My belief is that any advantages gained from going to a "simpler" target language would be far outweighed by the disadvantages. The facility (affixment) most often proposed for excision does more than merely make programs easier to write. Programs using affixment can be made smaller and more efficient than those which rely on explicit updating or recalculation, and the mechanism provides a convenient "hook" for runtime functions such as continuous tracking of conveyors.

Even if such language features were purely programming conveniences, there would still be good reasons to retain them. First off, the automatic coding procedures can also benefit from the facilities. For instance, the pin-in-hole writer of Section 8.7 described the insertion task and search loop in terms of the pin's frame. Very little would be gained by transferring the effort of relating these motions to the manipulator frame from the AL compiler to *write_pin_in_hole*, which already has enough to worry about. Second, compatibility with a user-oriented programming formalism is especially important for a developing system.[2] Many assembly operations will be too hard for the system, and will require "hand coding" by the user. Similarly, it may be necessary to edit programs

---

[1] The existing knothole allows the vision system to read and write variables and to cause and wait for events. This is adequate for some purposes, but limits the possibilities for close interaction.

[2] An interesting parallel would be a primitive compiler which could translate assignment statements but not loop control or subroutine linkages. For such a system, compatibility with a good symbolic assembler would be indispensable.

produced automatically to improve their efficiency or to make other minor refinements. Even if the computer could easily produce very low level output programs, the user would probably find higher level constructs to be preferable.

Development of a Formalism for Task-Level Specification.

The research reported in this document concentrated on a single task-oriented operation. Consequently, the problem of providing a good formal language for describing assembly tasks was not really addressed. The automatic coding examples discussed in Chapter 8 were generated by the use of SAIL procedure calls in lieu of an input language:

```
COMMENT Describe the initial situation;
STD_AFX(W1,BOX,BXBTM);
        ;
BXPLACE←NEW_TRANS(NILROTN,NEW_V3ECT(18*INCHES,40*INCHES,0));
EST_SET(W1,LOCUS_ESTIMATE,WORKSTATION,BOX,BXPLACE);
EST_SET(W1,DETERM_ESTIMATE,WORKSTATION,BOX,
        SIX_PE(0.3*INCHES,0.2*INCHES,0,0,0,5*DEGREES));
EST_SET(W1,LOCUS_ESTIMATE,WORKSTATION,PRACK,
        NEW_TRANS(AXW_ROTN(ZHAT,90*DEGREES),NEW_V3ECT(8*INCHES,47*INCHES,0)));
EST_SET(W1,DETERM_ESTIMATE,WORKSTATION,PRACK,NILTRANS);
EST_SET(W1,LOCUS_ESTIMATE,WORKSTATION,BMANIP,FRAME:VAL[BPARK]);
EST_SET(W1,DETERM_ESTIMATE,WORKSTATION,BMANIP,NILTRANS);
        ;
COMMENT Put pins into initial rack holes;
PINHPUT(W1,W2,PIN1,PRKH1,1,1000.,TRUE);
PINHPUT(W2,W3,PIN2,PRKH2,1,1000.,TRUE);
        ;
COMMENT Put pin into box hole;
PINHPUT(W3,W4,PIN1,BH1,1,100.,FALSE);
        ;
```

In principle, we could perhaps provide a user-oriented input language by cleaning up the calling conventions and supplying a large set of supporting SAIL macros. Although it is unclear whether such a "language" would be very satisfactory to users, it was acceptable for our purposes.

Object Models

Any object modelling scheme must fill the requirements of computational utility and user convenience. Objects were represented in this work by the use of attribute graphs, in which shape properties are represented in the nodes, structural properties by links, and location properties by link attributes. On the whole, this scheme proved quite satisfactory from a computational standpoint. The directness with which manufacturing tolerances and similar data could be incorporated was quite useful. Another significant advantage was that we could represent directly many useful properties, such as pin-to-hole tolerance requirements, which would be awkward to compute from a "purer" shape description formalism. This flexibility proved a great help to experimentation.[3]

It should be emphasized that *much* work remains to be done before we have a shape representation formalism capable of supporting a full scale automatic coding system for

---

[3] For instance, the distance of each hole to the nearest edge or other hole was originally represented explicitly. Later, code was added to compute this quantity from the shape information.

manipulators. A system that requires the user to spend more effort building models than he would spend writing manipulator code is not particularly useful. On the other hand, if the parts to be assembled are *already* represented in a computer-understandable form -- perhaps as output of a Computer Aided Design system -- then the marginal effort needed to use the automatic coding system may be acceptable. Consequently, provision of an economically attractive task-level manipulator programming system depends, in part, on further improvements in CAD systems and on development of ways to compute from the design descriptions most of the information required for manipulator coding.

Representation of Situational Information

In manipulator programming, we must deal with (1) object locations, (2) attachments between objects, and (3) errors in location values. The principal mechanisms we used to represent this information were symbolic assertions and parameterized mathematical expressions. Chapter 7 developed techniques for relating the two representations. In particular, "semantic" assertions relating object features were translated into mathematical constraints on scalar variables representing degrees of freedom. Linear programming methods were then applied to predict limits. This ability to "understand" symbolic descriptions of inter-object relations is quite useful for a task-level programming system. Similarly, the ability to deal with ranges of values is necessary for many tasks.

Although the basic approach taken by Chapter 7 seems to be sound, there were several difficulties with the specific techniques used. Linear programming is not particularly well suited to dealing with rotational degrees of freedom. Numerical techniques capable of handling rotation constraints directly might be better. A more serious problem, however, is the difficulty of characterizing the range of positions for which a particular strategy is valid. For instance, in deciding how to pick up an object, we must know whether the hand will always be able to reach a particular grasping position and how the motion time will vary with different object positions. Usually, the only way to get this information is to generate a number of trial points within the range of possible object positions. Analytic characterizations that could be combined with the object constraints would be very desirable.[4] For instance, if we have a function

$$\tau(A,B) = \text{expected time to move from A to B.}^{5}$$

and can describe the positions A and B in terms of some free variables

$$A = A(\lambda_1,\ldots,\lambda_m)$$
$$B = B(\mu_1,\ldots,\mu_n)$$

subject to constraints

---

[4] Shimano's present research on manipulator kinematics may lead to such characterizations. This work will be reported in his thesis [99].

[5] In Chapter 8, motion times were estimated by computing arm solutions for successive positions and then examining how much each joint had to move on each segment of motion. The difficulty with this procedure is that it doesn't give us a convenient analytic expression for time as a function of locations.

$$C_j(\bar{\lambda},\bar{\mu}) \leq 0$$

We could then compute the maximum time required to make the motion by

$$\tau_{max} = \max_{\lambda_i,\mu_i} \tau(A(\lambda_1,\ldots,\lambda_m),B(\mu_1,\ldots,\mu_n))$$

and use the result in evaluating alternative strategies.

The differential approximation techniques used for small perturbations worked much better, since the resulting problems were better matched to the numerical methods available. Since small differences are not generally important in determining the feasibility or cost of large motions, the principal use of these techniques was for estimation of errors, for which they were quite appropriate. The estimates produced were for *worst case* errors. It would probably be useful to consider the distributions of errors, as well. Thus, instead of computing

$$\xi^* = \max \xi(\delta_1,\ldots,\delta_n) = \max \Sigma c_i\delta_i$$

subject to constraints on the δ_i, we might wish to investigate the distribution of $\xi(\bar{\delta})$, given some assumptions about the distributions of the δ_i, in order to estimate average search times, success rates, or similar statistics.

Knowledge Base

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 computational techniques described in Chapters 4-7 were developed without any particular task in mind. When time came to write the automatic coding procedures described in Chapter 8, 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.[6] The principal

---

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

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[7] 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.

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

As I mentioned in the introduction, 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; 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 8, 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

---

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

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.[8] Before these advantages can be obtained for real manipulator programs, we need a better understanding of how individual coding decisions affect each other. Although 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.

Accepting Advice from the User

The development of a good paradigm for sharing coding responsibilities is an important goal. Even when automatic manipulator programming systems become sophisticated enough to write most programs on their own, some facility for helping out the computer in exceptional cases will still be desirable. Until that time is reached, it will be indispensible.

There are (at least) two problems that must be solved before advice-giving can be made effective. First, the computer must be able to understand the advice it is being given. Second, it must be able to make its own decisions in a manner consistent with what it has been told. For instance, if help is given in the form of manipulator code, the system must understand what the code does, where it is to be included in the manipulator program, and what initialization actions, if any, must be performed. To obtain this information, it may have to "read" the user's code; as we have seen, this can be a *very* hard problem. Additional assertions ("comments") by the user may be required. The difficulties are somewhat lessened if contextual information from a task-level description is available or if the computer has *asked* for help rather than been presented with unsolicited assistance.

How to Get There from Here

As we have seen, many of the relevant factors for manipulator programming – accuracies, expected motion times, joint limits, etc. – inherently involve numerical estimates which are difficult for human programmers to obtain or work with. As better computational representations for such information are developed, it is reasonable to expect that automatic systems will eventually become *better* manipulator programmers than their human counterparts. As we have also seen, however, much research must be done before this can come about. Thus, we are left with the question of what to do in the meantime.

It seems to me that the most the most fruitful near-term effort would be the development of a highly interactive system in which programming is a joint endeavor by the computer and the user. At the center of such a system would be data structures representing the program

---

[8] Sacerdoti [94, 95] successfully applied similar ideas to a different domain.

being developed, together with editing, display, and interpretation routines.[9] The underlying philosophy would be progressive refinement. Initially, the user would make a task-level specification of the program to be written. This specification would then be elaborated into manipulator-level code by means of:

1. Statements supplied by the user.
2. Automatic coding primitives.
3. "Smart" guiding, in which the task context is used to interpret manipulation actions performed under manual control by the user.
4. Combinations of the above.

Although there are a number of technical problems,[10] there are also several advantages to this approach. First, it is "evolutionary". As new facilities are developed, they can be added without radical changes to the underlying formalism. If a needed task-level primitive is not available, the user has other ways to get the desired effects. Second, debugging facilities can be incorporated in a natural way. Third, the construction of special-purpose subsystems or provision of a tape recorder mode package would be relatively easy within such a system. And, finally, the system would be fun to build.

---

[9] Compare this with the POINTY system, which is described in Appendix F. In POINTY, the central data structure is an affixment tree of frame variables. The system contains facilities for editing this tree, for using the manipulator to measure location values, and for translating the tree into the corresponding AL declarations and affixments.

[10] One problem is keeping track of of the relation between a task-level statement and the manipulator-level constructs used to implement it. Another is ensuring that the user's code remains compatible with that generated by the computer.

Chapter 10.

Bibliography

[1] Gerald J Agin, *Representation and Description of Curved Objects*, Ph.D Dissertation, Stanford Artificial Intelligence Laboratory Memo AIM-243, October 1972.

[2] A P. Ambler and R. J. Popplestone, *Inferring the Positions of Bodies from Specified Spatial Relationships*, manuscript, Department of Machine Intelligence, University of Edinburgh, Published in AISB Summer Conference, University of Sussex, July 1974.

[3] Robert H. Anderson and Nake M. Kamrany, *Advanced Computer-Based Manufacturing Systems for Defense Needs*, USC Information Sciences Institute Report RR-73-10, May 1973.

[4] *Proceedings of an ACM Conference on Proving Assertions About Programs*, SIGPLAN Notices, January 1972.

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

[6] Robert M. Balzer, *Automatic Programming*, Information Sciences Institute Technical Report, September 1972.

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

[8] Bruce G. Baumgart, *MICRO-PLANNER Alternate Reference Manual*, Stanford Artificial Intelligence Laboratory Operating Note 67, April 1972.

[9] Bruce G. Baumgart, *GEOMED — A Geometric Editor*, Stanford Artificial Intelligence Laboratory Memo AIM-232, Stanford Computer Science Report STAN-CS-74-414, May 1974.

[10] Bruce G. Baumgart, *Geometric Modelling for Computer Vision*, Ph. D. Dissertation, Stanford Artificial Intelligence Laboratory Memo AIM-249, Stanford Computer Science Report STAN-CS-74-463, October 1974.

[11] Jack R. Buchanan, *A Study in Automatic Programming*, Ph.D Dissertation, Stanford Artificial Intelligence Laboratory Memo AIM-245, Stanford Computer Science Report STAN-CS-74-458, May 1974.

[12] Jack R. Buchanan and David C. Luckham, *On Automating the Construction of Programs*, Stanford Artificial Intelligence Laboratory Memo AIM-236, Stanford Computer Science Report STAN-CS-74-433, May 1974.

[13] Antal K. Bejczy, "Machine Intelligence for Autonomous Manipulation", *Proceedings of the First National Conference for Remotely Manned Systems for Exploration and Operation in Space*, California Institute of Technology, 1973.

[14] Antal K. Bejczy, *New Techniques for Terminal Phase Control of Manipulator Motion*, JPL Technical Memorandum 760-98, February 1974.

[15] Antal K. Bejczy, *Robot Arm Dynamics and Control*, JPL Technical Memorandum 33-669, February 1974.

[16] Antal K. Bejczy, "Environment-Sensitive Manipulator Control", *1974 IEEE Conference on Decision and Control*, November 1974.

[17] T. O. 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.

[18] T. O. 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. Gafford, *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.

[19] T. O. 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. Gafford, *Exploratory Study of Computer Integrated Assembly Systems*, Prepared for the National Science Foundation. Stanford Artificial Intelligence Laboratory Progress Report covering November 1975 to June 1976.

[20] Daniel Bobrow and Bertram Raphael, "New Programming Languages for Artificial Intelligence", *Computing Surveys*, Vol. 6, No. 3, September 1974.

[21] Robert C. Bolles, Richard Paul, *The Use of Sensory Feedback in a Programmable Assembly System*, Stanford Artificial Intelligence Laboratory Memo AIM-220, Stanford Computer Science Report STAN-CS-73-396, October 1973.

[22] Robert C. Bolles, *Verification Vision Within a Programmable Assembly System: an Introductory Discussion*, Stanford Artificial Intelligence Laboratory Memo AIM-275, Stanford Computer Science Report STAN-CS-75-537, December 1975.

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

[24] I.C. Braid, *Designing With Volumes*, Cantab Press, Cambridge, England, 1974.

[25] I.C. Braid, "The Synthesis of Solids Bounded by Many Faces", *Communications of the ACM*, Volume 18, Number 4, April 1975.

[26] I.C. Braid, *Six Systems for Shape Design and Representation — a Review*, Computer Aided Design Group Document No. 87, University of Cambridge, May 1975.

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

[28] R. T. Chien, V. C. Jones, "Acquisition of Moving Objects and Hand-Eye Coordination", *Proceedings of the Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 737-741, September 1975.

[29] L. Stephen Coles, *Categorical Bibliography of Literature in the Field of Robotics*, Stanford Research Institute Artificial Intelligence Center Technical Note 88-3, March 1975.

[30] George B. Dantzig, *Linear Programming and Extensions*, Princton University Press, Princton, New Jersey, 1963.

[31] John A. Darringer and Michael Blasgen, *MAPLE: A High Level Language for Research in Mechanical Assembly*, IBM Research Report RC-5606, September 1975.

[32] H. A. Ernst, *MH-1, A Computer-Operated Mechanical Hand*, Sc. D. Thesis, Massachusetts Institute of Technology, December 1961.

[33] Scott E. Fahlman, *A Planning System for Robot Construction Tasks*, MIT Artificial Intelligence Laboratory Report AI TR-283, May 1973.

[34] Gary Feldman and Donald Peiper, *Avoid*, Film (16mm, color, silent 1972), Stanford Artificial Intelligence Laboratory, March 1969.

[35] J. A. Feldman, P. D. Rovner, "An Algol-Based Associative Language," *Communications of the ACM* 12, 8, pp. 439-449, August 1969.

[36] J. Feldman, J. Low, R. Taylor, D. Swinehart, "Recent Developments in SAIL, an Algol Based Language for Artificial Intelligence," *Proceedings of the FJCC*, pp 1193-1202, Stanford Artificial Intelligence Laboratory Memo AIM-176, Stanford Computer Science Report STAN-CS-72-308, November 1972.

[37] Raphael Finkel, Russell Taylor, Robert Bolles, Richard Paul, Jerome Feldman, *AL, A Programming System for Automation*, Stanford Artificial Intelligence Laboratory Memo AIM-177, Stanford Computer Science Report STAN-CS-74-456, November 1974.

[38] Raphael Finkel, Russell Taylor, Robert Bolles, Richard Paul, Jerome Feldman, "An Overview of AL, A Programming System for Automation", *Proceedings of the Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 758-765, September 1975.

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

[40] Earlwood T. Fortini, *Dimensioning for Interchangeable Manufacture*, Industrial Press, Inc., New York, 1967.

[41] Aharon Gill, *Visual Feedback and Related Problems in Computer Controlled Hand Eye Coordination*, Ph.D. Dissertation, Stanford Artificial Intelligence Laboratory Memo AIM-178, Stanford Computer Science Report STAN-CS-72-312, October 1972.

[28] R. T. Chien, V. C. Jones, "Acquisition of Moving Objects and Hand-Eye Coordination", *Proceedings of the Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 737-741, September 1975.

[29] L. Stephen Coles, *Categorical Bibliography of Literature in the Field of Robotics*, Stanford Research Institute Artificial Intelligence Center Technical Note 88-3, March 1975.

[30] George B. Dantzig, *Linear Programming and Extensions*, Princton University Press, Princton, New Jersey, 1963.

[31] John A. Darringer and Michael Blasgen, *MAPLE: A High Level Language for Research in Mechanical Assembly*, IBM Research Report RC-5606, September 1975.

[32] H. A. Ernst, *MH-1, A Computer-Operated Mechanical Hand*, Sc. D. Thesis, Massachusetts Institute of Technology, December 1961.

[33] Scott E. Fahlman, *A Planning System for Robot Construction Tasks*, MIT Artificial Intelligence Laboratory Report AI TR-283, May 1973.

[34] Gary Feldman and Donald Peiper, *Avoid*, Film (16mm, color, silent 1972), Stanford Artificial Intelligence Laboratory, March 1969.

[35] J. A. Feldman, P. D. Rovner, "An Algol-Based Associative Language," *Communications of the ACM* 12, 8, pp. 439-449, August 1969.

[36] J. Feldman, J. Low, R. Taylor, D. Swinehart, "Recent Developments in SAIL, an Algol Based Language for Artificial Intelligence," *Proceedings of the FJCC*, pp 1193-1202, Stanford Artificial Intelligence Laboratory Memo AIM-176, Stanford Computer Science Report STAN-CS-72-308, November 1972.

[37] Raphael Finkel, Russell Taylor, Robert Bolles, Richard Paul, Jerome Feldman, *AL, A Programming System for Automation*, Stanford Artificial Intelligence Laboratory Memo AIM-177, Stanford Computer Science Report STAN-CS-74-456, November 1974.

[38] Raphael Finkel, Russell Taylor, Robert Bolles, Richard Paul, Jerome Feldman, "An Overview of AL, A Programming System for Automation", *Proceedings of the Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 758-765, September 1975.

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

[40] Earlwood T. Fortini, *Dimensioning for Interchangeable Manufacture*, Industrial Press, Inc., New York, 1967.

[41] Aharon Gill, *Visual Feedback and Related Problems in Computer Controlled Hand Eye Coordination*, Ph.D. Dissertation, Stanford Artificial Intelligence Laboratory Memo AIM-178, Stanford Computer Science Report STAN-CS-72-312, October 1972.

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

[43] 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.

[44] C. Green and D. Barstow, "Some Rules for the Automatic Synthesis of Knowledge", *Advanced Papers of the Fourth International Joint Conference on Artificial Intelligence*, Tblisi, Georgia, USSR, September 1975.

[45] C. Green, *The PSI Program Synthesis System: an Overview*, manuscript, submitted to the Second International Conference on Software Engineering, October 1976.

[46] David D. Grossman, *Procedural Representation of Three Dimensional Objects*, IBM Research Report RC-5314, March 1975.

[47] D. D. Grossman and M. W. Blasgen, *Orienting Mechanical Parts With a Computer Controlled Manipulator*, IEEE Transactions on Systems, Man, and Cybernetics, September 1975.

[48] 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.

[49] David D. Grossman, *Monte Carlo Simulation of Tolerancing in Discrete Parts Manufacturing and Assembly*, Stanford Artificial Intelligence Laboratory Memo AIM-280, Stanford Computer Science Report STAN-CS-76-555, June 1976.

[50] William Harrison, *Compiler Analysis of the Value Ranges for Variables*, IBM Research Report RC-5544, July 1975.

[51] Brian Harvey, *Monitor Command Manual*, Stanford Artificial Intelligence Laboratory Operating Note 54.5, Revised by Martin Frost, January 1976.

[52] Carl Hewitt, *PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot*, MIT Project MAC, Artificial Intelligence Memo NO. 168, August 1970.

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

[54] *Industrial Robots – A Survey*, International Fluidics Services Ltd., Bedford, England, 1972.

[55] *Fifth International Symposium on Industrial Robots*, Proceedings, IIT Research Institute, Chicago, September 1975.

[56] Hirochika Inoue, *Force Feedback in Precise Assembly Tasks*, MIT Artificial Intelligence Laboratory Technical Report 308, August 1974.

[57] Suzanne Kandra, *Motion and Vision*, Film (16mm, color, sound, 22 minutes), Stanford Artificial Intelligence Laboratory, November 1972.

[58] Mark A. Lavin, *MODFEAT: A System for Naming Polyhedral Features of Three Dimensional Objects*, IBM Research Report RC-5764, December 1975.

[59] Mark A. Lavin and Lawrence I. Lieberman, *A System for Modelling Three-Dimensional Objects*, IBM Research Report RC-5765, December 1975.

[60] W. H. P. Leslie, ed., *Numerical Control Programming Languages*, Proceedings of the First International IFIP/IFAC PROLOMAT Conference, 1969, North Holland Publishing Company, London, 1970.

[61] Richard A. Lewis, Antal K. Bejczy, "Planning Considerations for a Roving Robot with Arm", *Proceedings of the Third International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 308-316, August 1973.

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

[63] L. I. Lieberman and M. A. Wesley, *The Design of a Geometric Data Base for Mechanical Assembly*, IBM Research Report RC-5489, August 1975.

[64] L. I. Lieberman and M. A. Wesley, *AUTOPASS, A Very High Level Programming Language for Mechanical Assembler Systems*, IBM Research Report RC-5599, August 1975.

[65] James R. Low, John F. Reiser, Hanan J. Samet, Robert F. Sproull, Robert Smith, Daniel C. Swinehart, Russell H. Taylor, Kurt A. VanLehn, *SAIL User Manual Update*, March 1976.

[66] 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.

[67] James Low and Paul Rovner, *Techniques for the Automatic Selection of Data Structures*, Computer Science Department Report TR4, University of Rochester, 1976.

[68] Zohar Manna and Richard Waldinger, *Knowledge and Reasoning in Program Synthesis*, Stanford Research Institute Artificial Intelligence Center Technical Note 98, November 1974.

[69] John McCarthy, *Plans for the Stanford Artificial Intelligence Project*, Stanford Artificial Intelligence Laboratory Memo AIM-31, April 1965.

[70] D. V. McDermott and G. J. Sussman, *CONNIVER Reference Manual*, MIT Artificial Intelligence Laboratory AI Memo 259, May 1972.

[71] C. Murphy, *The Reliability of Systems*, unpublished manuscript, date unknown.

[72] Eiji Nakano, Shotaro Ozaki, Tatsuzo Ishida, Ichiro Kato, *Cooperational Control of the Anthropomorphous Manipulator "MELARM"*, Fourth International Symposium on Industrial Robots, November 1974.

[73] Ramakant Nevatia, *Structured Descriptions of Complex Curved Objects for Recognition and Visual Memory*, Ph.D. Dissertation, Stanford Artificial Intelligence Laboratory Memo AIM-250, Stanford Computer Science Report STAN-CS-74-464, October 1974.

[74] J. L. Nevins, D. E. Whitney, S. N. Simunovic, *Report on Advanced Automation: System Architecture for Assembly Machines*, The Charles Stark Draper Laboratory, Inc., Prepared for the National Science Foundation, Memo No. R-764, covering June 1972 to November 1973, November 1973.

[75] 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.

[76] Nils J. Nilsson, *Artificial Intelliegnce — Research and Applications*, Progress Report for Defense Advanced Research Progress Agency covering April 1973 through April 1974.

[77] Nils J. Nilsson, *Artificial Intelliegnce — Research and Applications*, Progress Report for Defense Advanced Research Progress Agency covering March 1974 through March 1975.

[78] *An Introduction to PADL*, Production Automation Technical Memorandum 22, University of Rochester, December 1975.

[79] R. Paul and K. Pingle, *Instant Insanity*, Film (16 mm, color, silent, 5 minutes), Stanford Artificial Intelligence Laboratory, August 1971.

[80] 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.

[81] R. Paul, K. Pingle, and R Bolles, *Automated Pump Assembly*, Film (16 mm, color, silent, 7 minutes), Stanford Artificial Intelligence Laboratory, April 1973.

[82] R. Paul, *WAVE: A Model-Based Language for Manipulator Control*, Manuscript, Submitted to Sixth IITRI Conference, Octobet 1976.

[83] D. L. Peiper, *The Kinematics of Manipulators Under Computer Control*, Stanford Artificial Intelligence Laboratory Memo AIM-72, Stanford Computer Science Report STAN-CS-68-116, October 1968.

[84] K. Pingle, R. Paul, R. Bolles, *Programmable Assembly, Three Short Examples*, Film (16 mm, color, sound, 8 minutes), Stanford Artificial Intelligence Laboratory, October 1974.

[85] Rene Reboh and Earl Sacerdoti, *A preliminary QLISP Manual*, Stanford Research Institute Artificial Intelligence Center Technical Note 81, August 1973.

[86] John F. Reiser, *BAIL — A Debugger for SAIL*, Stanford Artificial Intelligence Laboratory Memo AIM-270, Stanford Computer Science Report STAN-CS-75-523, October 1975.

[87] A. A. G. Requicha, N. M. Samuel, H. B. Voelker, *Part and Assembly Description Languages — II*, Production Automation Project Technical Memo TM-20a, November 1974.

[88] L. G. Roberts, *Homogeneous Matrix Representation and Manipulation of N-Dimensional Constructs*, Document MS1045, Lincoln Laboratory, Massachusetts Institute of Technology, May 1965.

[89] C. Rosen, D. Nitzan, G. Agin, R. Blean, R. Duda, G. Gleason, J. Kremers, W. Park, R. Paul, A. Sword, *Exploratory Research in Advanced Automation*, Prepared for the National Science Foundation, Stanford Research Institute Project 2591 Fourth Report, June 1975.

[90] C. Rosen, D. Nitzan, R. Duda, G. Gleason, J. Kremers, W. Park, R. Paul, *Exploratory Research in Advanced Automation*, Prepared for the National Science Foundation, Stanford Research Institute Project 4391 Fifth Report, January 1976.

[91] Bernard Roth, Jahangir Rastegar, Victor Scheinman, "On the Design of Computer Controlled Manipulators", Lecture given at the First CISM - IFToMM Symposium, 5-8 September 1973, from *On Theory and Practice of Robots and Manipulators*, Vol. I, Springer-Verlag, Vienna-New York, 1974.

[92] Bernard Roth, *Performance Evaluation of Manipulators from a Kinematic Viewpoint*, manuscript, Department of Mechanical Engineering, Stanford University, 1975.

[93] J. Rulifson, *et al.*, *QA4: A Procedural Calculus for Intuitive Reasoning*, Stanford Research Institute Artificial Intelligence Center, Technical Note 73, November 1973.

[94] Earl D. Sacerdoti, *The Nonlinear Nature of Plans*, Stanford Research Institute Artificial Intelligence Center Technical Note 101, January 1975.

[95] Earl D. Sacerdoti, *A Structure for Plans and Behavior*, Stanford Research Institute Artificial Intelligence Center Technical Note 109, August 1975.

[96] 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.

[97] Victor D. Scheinman, *Design of a Computer Controlled Manipulator*, Stanford Artificial Intelligence Laboratory Memo AIM-92, June 1969.

[98] J. T. Schwartz, *Automatic Data Structure Choice in a Language of Very High Level*, Courant Institute, NYU, 1974.

[99] Bruce Shimano, **** Title Unknown ****, Ph.D. Dissertation, Stanford University, Mid 1977.

[100] Frank Skinner, *Design of a Multiple Prehension Manipulator System*, The American Society of Mechanical Engineers, United Engineering Center, New York, May 1974.

[101] Robert F. Sproull, **** Title Unknown ****, Ph.D. Dissertation, Stanford Computer Science Department, Summer 1976.

[102] G. J. Sussman, T. Winograd, and E. Charniak, *MICROPLANNER Reference Manual*, MIT Artificial Intelligence Laboratory Memo 203, July 1970.

[103] G. J. Sussman, *A Computational Model of Skill Acquisition*, Ph.D. Dissertation, MIT Artificial Intelligence Laboratory TR-297, August 1973.

[104] 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.

[105] Daniel C. Swinehart, *Copilot, a Multiple Process Approach to Interactive Programming Systems*, Stanford Artificial Intelligence Laboratory Memo AIM-230, Stanford Computer Science Report STAN-CS-74-412, July, 1974.

[106] A. J. Sword and W. T. Park, *Location and Acquisition of Objects in Unpredictable Locations*, Stanford Research Institute Artificial Intelligence Center Technical Note 102, June 1975.

[107] Kurt A. VanLehn, ed, *Sail User Manual*, Stanford Artificial Intelligence Laboratory Memo AIM-204, Stanford Computer Science Report STAN-CS-73-373, July 1973.

[108] Richard Waldinger, *An Interactive System for the Verification of Computer Programs*, Stanford Research Institute Artificial Intelligence Center September 1973.

[109] Richard Waldinger, *Achieving Several Goals Simultaneously*, Stanford Research Institute Artificial Intelligence Center Technical Note 107, July 1975.

[110] Mitchel R. Ward, *Specifications for a Computer Controlled Manipulator*, Computer Science Department Research Laboratories, General Motors Corp., Warren, Michigan, Research Publication GMR-2066, February, 1976.

[111] William M. Wichman, *Use of Optical Feedback in the Computer Control of an Arm*, Stanford Artificial Intelligence Laboratory Memo AIM-55, August, 1967.

[112] P. M. Will, *Computer Controlled Mechanical Assembly*, IBM Research Report RC-5428, May 1975.

[113] P. M. Will, D. D. Grossman, "An Experimental System for Computer Controlled Mechanical Assembly", *IEEE Transactions on Computers*, Vol. C-24, No. 9, September 1975.

[114] P. H. Winston, Ed., *Progress in Vision and Robotics*, MIT Artificial Intelligence Laboratory TR-281, May 1973.

[115] P. H. Winston, Ed., *New Progress in Artificial Intelligence*, MIT Artificial Intelligence Laboratory TR-310, September 1974, revised 1975.

[116] S. I. Zukovitsky and L. I. Avdeyeva, *Linear and Convex Programming*, Translated by Scripta Technica, Inc., W. B. Saunders Company, Philadelphia Pa., 1966.

Appendix A.

A Summary Description of AL

The AL language has been described elsewhere, particularly in the AL report [37]. A short description will be repeated here in order to make the text self-contained.[1] This description will be divided into several parts, discussing in turn data structures, control structures, and motion specifications.

A.1 Data structures

The concept of *variable* has become standard in programming languages. In AL, variables are intended to represent such diverse constructs as locations, orientations, repeat counts, and forces. Variables are declared as in ALGOL, except there is no restriction that all declarations in a block precede all executable statements; variables must only be declared before they are used. Declarations assign to variables two independent qualities: *type* and *dimension*. These are described separately.

Data Types

Type **scalar**

Scalars are floating-point numbers that can be added, subtracted, multiplied, and divided. Scalar constants are written with or without a decimal point; they are represented exactly at least to 1000. (The first unrepresentable scalar is dependent on the implementation.)

Type **vector**

Vectors are ordered triples of scalars. Vectors are understood to apply to a fixed coordinate system called the station. They can be generated from three scalars by the construct vector(*s1*, *s2*, *s3*), and they may be added, subtracted, and multiplied by scalars. There are three predeclared vector constants: xhat, yhat, and zhat, corresponding to vector(*1,0,0*), vector(*0,1,0*) and vector(*0,0,1*) respectively. The dot product is also available; by dotting a vector with xhat, yhat, and zhat one can extract its three components.

Type **rotation**

Rotations represent either orientations or rotation operators. The orientation is derived as the result of applying the rotation to the station coordinate system. Rotations can be generated from an axis vector and a magnitude scalar by the construct rot(*v*, *s*), where *s* is understood to be in degrees (see the discussion of dimension). Rotations can be multiplied together. There is one predeclared rotation constant, nilrotn, which represents the identity rotation.

---

[1] I am indebted to Ray Finkel, my co-worker on AL, for providing the bulk of this short description of the language.

Type frame

A frame represents a coordinate system, that is, the location and orientation of its origin with respect to the station. Frames can be generated from a rotation and a vector by the construct **frame**(*r*, *v*). Frames can be rotated by application of rotations and displaced by addition to vectors. There are several predeclared frame constants: **station**, **bpark**, which is the rest position of the blue arm, and **ypark**, the rest position of the yellow arm. To say that an arm is at a frame means that the local coordinate system of its hand coincides in location and orientation with the frame.

Type transform

A transform represents an operator that can move vectors, rotations, and frames from one coordinate system to another. Transforms can be generated from a rotation and a vector by the construct **trans**(*r*, *v*). The application of a transform to another object is indicated by multiplication with the transform, as described in Appendix B. Frames and transforms have the same runtime representation, and one is readily coerced into the other. The inverse of a transform is computed by the function **inv**(*t*), as described in Appendix B.

Dimensions

Dimensions are used to tell what units entities possess and to ensure that they are not misused. Dimensions have not been implemented in the current version of AL, but they serve to clarify the use of scalars especially, and allow the programmer to use units of his own choosing. The available base dimensions are time, distance, angle, and mass. It is also possible to leave an entity undimensioned, in which case it can be coerced to any necessary dimension. When two scalars are added or subtracted, they must have the same dimension, which will become the dimension of the result. When scalars are multiplied or divided, the dimension of the result is derived from the dimensions of the arguments; in this way, new dimensions can be achieved. Vectors must also agree in dimension to be added or subtracted. Frames may only be formed from distance vectors. Rotations must have angle scalars to indicate the magnitude. A transform carries the dimension of its translational part, which must agree with any argument to which it is applied. The basic units are centimeters[2] seconds, grams, and degrees. If one wants to work in other units, this can be done by multiplication by suitable constants; for example, one can say that the constant **inches** is defined to have the value 2.54. Then 4*inches really four inches.

Affixment

Assembly involves affixing one object to another. The affixment mechanism of AL provides a mechanism for automatically keeping track of the values of variables that have been affixed to others. Since this construct is discussed extensively in Section 3.6 and Appendix C, we won't say anything more about it here.

---

[2] For reasons that are not entirely clear, the initial AL runtime system was implemented with inches as the basic unit. The planning system, however, was based on centimeters, as was the original language design. For consistency, we will assume the default is centimeters throughout this document.

A.2 Control structures

AL has many of the traditional ALGOL structures, including statement as the primary locus of control, block structure, declared variables, for and while loops, if-then[-else] conditionals, assignment statements. Procedures have not been implemented, although they are specified in the original AL report. There are some control structures not at all standard in ALGOL, which are necessary for parallel execution of several tasks and for asynchronous use of feedback.

Simultaneous Processes

The cobegin-coend construct is used to split control from one locus into several simultaneously executing loci. The two words cobegin and coend bracket a set of statements, each of which is to execute at the same time as the others. These statements can, of course, be entire blocks performing large tasks. The principal use for the cobegin construct is the independent control of several manipulators, although it is quite feasible to let some complex computation execute as a background to arm motions through this mechanism. cobegins can be nested, to split the locus of control arbitrarily.

There is no way to specify the priority of any thread of the simultaneously executing processes, but it is possible to have one thread wait for another to reach important places. This is done by the event mechanism. Events are declared as if they were variables. The two operations available on events are signal and wait. Waiting on an event decrements its count, which is initially zero. If the count should be negative after the decrement, the thread of execution is temporarily suspended. Signalling an event adds one to the count. If the result should be zero or negative, one of the waiting threads is reawakened and continues its execution.

Asynchronous Feedback

Use of asynchronous feedback is accomplished by means of the *condition monitor*. A condition monitor is written in this way:

on ⟨*condition*⟩ **do** ⟨*body*⟩

The condition can be any scalar or boolean expression, just as is used in while loops and if-then[-else] conditionals. (The convention used in SAIL [107,65] applies: any non-zero scalar result is considered true, and zero is considered false.) The condition may also be an event.

After the execution of the on statement, the condition monitor periodically checks to see if the condition has the value true. If it ever should happen (or if the event given as the condition should ever be signalled), then the monitor is said to *trigger*, and the body of the monitor is executed. A side effect of triggering is that the monitor is disabled. It can explicitly re-enable itself in the body by executing the statement enable. If the block in which a condition monitor was declared is exited, the monitor ceases to function.

Condition monitors are essential for manipulator control, since they allow programmed response to expected termination or error conditions detected by testing the force on the moving arm.

Force is detected by means of scalar functions that have these forms:

force(<*vector expression*>)
torque(<*vector expression*>)

The arguments refer to direction (for force) and axis (for torque) along which the force is to be sensed. The vectors are understood as pertaining to the hand coordinate system; as the arm moves, the vectors move with the hand.

A.3 Motion Specifications

Simple Motions

Motions are accomplished by executing *motion statements*. The motion statement has the following form:

move <*controllable variable*>
<*clauses*>

where the <*clauses*> include information about the destination, intermediate points, timing, and conditions. The *controllable variable* should either be the name of a predeclared manipulator variable, such as blue, or should be a variable affixed to a controllable variable. (Long chains of affixment ultimately involving a manipulator variable are legal.) The various clauses are these:

Clause to < frame *expression*>

This clause is required to indicate where the terminus of the motion is to be. The remaining clauses are all optional.

Clause with duration { ≤ | = | ≥ } <*scalar expression*>

The duration clause tells the time for the whole motion. If this optional clause is omitted, the time required for the motion, based on the distance each joint is expected to travel and its maximum speed, will be used.

Clause via <*list of decorated frame expressions*>

The controllable frame is to pass through each of the frames listed on its way to the destination. The optional decorations can be of these forms:

with <*duration clause*>
with velocity = <*velocity vector expression*>
then <*statement*>

The first of these specifies how long it should take to reach this intermediate point from the previous intermediate point (or the start of the motion); the second tells what velocity the manipulator should have. (This was included for the sake of accurate tossing of objects).

If some particular action should begin at the time that the intermediate point is reached, it can be specified in the THEN decoration.

Clause on ⟨*condition*⟩ do ⟨*statement*⟩

The condition monitor is active only during the motion. One legal condition is "ARRIVAL", which will trigger only if the arm arrives at its destination without having been halted, either through execution of the STOP command or some error condition detected by the arm control code. The principal use of the condition monitor is to stop the arm on acquistion of some force limit.

Clause with force(⟨*vector expression*⟩) = ⟨*scalar expression*⟩
Clause with torque(⟨*vector expression*⟩) = ⟨*scalar expression*⟩

These unimplemented options allow the arm to apply force or torque during the course of the motion. The vectors, for direction (force) and axis (torque), are understood to be in the hand's coordinate frame.

Center Motions

The purpose of the center statement is to grasp an object in the hand of a manipulator by closing the hand until one of the fingers touches it, then continuing to close, moving the whole arm to accomodate to the location of the object, until the other finger has touched it. This process leaves the object unmoved. The syntax is quite simple:

center ⟨*hand frame*⟩
⟨*clauses*⟩

The only legal clause is the condition monitor.

Stopping

Any device may be stopped by the command

stop ⟨*manipulator variable*⟩.

Although this statement is likely to be used most often in the conclusion of a condition monitor, it is legal anywhere.

Device Control

AL is intended to be usable even when devices like electric screwdrivers and pneumatic vises are interfaced into the runtime machine. These devices are what may be called "scalar devices", that is, their operation can be described as a scalar function of time. There is an operate statement to run such machines:

operate ⟨*device variable*⟩
⟨*clauses*⟩

The clauses will depend on the particular device, but duration and velocity will likely be

common. Note that even though the gripping end of the arm is a scalar device, it is treated by means of the move statement.

Appendix B.

Notational and Arithmetic Conventions

In AL, frames are used to represent coordinate systems, and transes are used to represent mappings between coordinate systems. Aside from this difference in usage, the two data types are isomorphic, since the value of any frame is given by a transformation from a distinguished coordinate system, station.[3] AL allows free coercion between the two types, both of which have the same internal representation:[4]

$$\text{trans}(R,v) = \text{frame}(R,v) = \begin{vmatrix} R_{11} & R_{12} & R_{13} & v_1 \\ R_{21} & R_{22} & R_{23} & v_2 \\ R_{31} & R_{32} & R_{33} & v_3 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

where the R_{ij} form a rotation matrix. Similarly, the internal representation for a vector is:

$$\text{vector}(x,y,z) = \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix}$$

The rules for multiplication of frames and transes with each other and with vectors are then easily derived from the corresponding matrix products.

$$\text{trans}(R,v)*u = Ru + v$$
$$\text{frame}(R_a,v_a)*\text{trans}(R_b,v_b) = \text{trans}(R_aR_b, v_a+R_av_b)$$
$$\text{inv}(\text{trans}(R,v)) = \text{trans}(R^{-1}, -R^{-1}v)$$

AL rotations are represented internally as transes

$$\text{rot}(axis,magn) = \text{trans}(\text{rot}(axis,magn),\text{nilvect})$$

and are multiplied in the obvious manner.

In Chapter 7, we frequently wanted to split a trans into its constituent rotation and translation. This was facilitated by use of the construction function, transl:

---

[3] In retrospect, the inclusion of two data types for essentially the same thing was probably a bad idea: the principal consequence has been the confusion of persons trying to understand the language.

[4] Since the bottom row is constant, it does not actually have to be stored. This representation is just a special case of the "homogeneous coordinates" first popularized by Roberts [88].

transl(v) = trans(nilrotn,v)

Thus

trans(R,v) = transl(v)*R

Transformation between coordinate systems (frames) in AL is generally accomplished by multiplying *on the right* by a trans.

$$B = A * T_{AB}$$

In this case, T_{AB} gives the location and orientation of B's coordinate system with respect to the coordinate system of A.

For instance, suppose that we have a box with a hole drilled in it. Suppose, further, that the hole's center is at h with respect to the box, and that the box is at position frame(rot($\hat{z}$,β),b) with respect to the work station. Then the location of the hole center will be given by

$$\begin{aligned} h^{station} &= \text{frame}(\text{rot}(\hat{z},\beta),b) * h \\ &= \text{rot}(\hat{z},\beta) * h + b \end{aligned}$$

Similarly, if R_{BH} gives the hole's rotation with respect to the box, then the rotation of the box in the work station will be given by:

$$R_h^{station} = \text{rot}(\hat{z},\beta) * R_{BH}$$

Thus, the coordinate system of the hole (with respect to the work station) will be given by

$$\begin{aligned} \text{Hole} &= \text{frame}(R_h^{station}, h^{station}) \\ &= \text{frame}(\text{rot}(\hat{z},\beta) * R_{BH}, \text{rot}(\hat{z},\beta) * h + b) \\ &= \text{frame}(\text{rot}(\hat{z},\beta),b) * \text{trans}(R_{BH},h) \\ &= \text{Box} * T_{BH} \end{aligned}$$

Text Conventions

A number of textual conventions have been used in this document. Although we haven't adhered rigidly to any single convention, it seems only fair to tell you about the most common ones. Generally, **boldface** symbols in the coding examples are reserved words, while *italic* symbols are user identifiers. In the mathematical sections, vectors are commonly given as bold lower case letters, such as "**v**", and unit vectors by the use of a hat, as in "$\hat{x}$". T_{AB} refers to the trans that takes A into B:

$$B = A * T_{AB}$$

Similarly, R_{AB} and p_{AB} refer to the orientation and displacement parts of T_{AB}.

$$T_{AB} = \text{trans}(R_{AB}, p_{AB}) = \text{transl}(p_{AB}) * R_{AB}$$

In Chapter 7, especially, we use "I" to refer to the identity matrix, transformation, or rotation, as appropriate for the context.

Appendix C.

Automatic Updating Implementation

C.1 Overview

As we stated in Chapter 3, affixments are translated by AL into commands to build a "graph structure" expressing the relation between the variables. Whenever a variable is changed, all values that depend on it are marked as "invalid". Whenever an "invalid" value is needed, it may be computed from the dependency relations.

This appendix defines the various flavors of affixment in terms of the corresponding graph structure modifications. It then presents a "pseudo-ALGOL" description of the data structures and algorithms used to implement the graph structure primitives. Minor details of implementation, such as ring structure pointers, will be left out of this description.

C.2 Semantics of Affixment

The most general form of the affix statement is

affix *frame1* **to** *frame2* **[by** *trans_var* **] [at** *trans_exp* **] [rigidly]**

where the clauses in the brackets are optional and may occur in any order. If the **by** clause is omitted, then the AL compiler will invent a variable to use in relating *frame1* to *frame2*. If the **at** clause is missing, then **inv**(*frame2*)*****frame1* is used as a default. The semantics of **rigid** affixment are:

```
trans_var ← trans_exp;
frame1 <= frame2*trans_var; { Read "frame1 is computed by frame2*trans_var" }
frame2 <= frame1*inv(trans_var);
```

For example,

```
affix handle to driver by hxf rigidly;
```

would be translated into

```
hxf ← inv(driver)*handle;
handle <= driver*hxf;
driver <= handle*inv(hxf);
```

Non-rigid affixment is translated as follows:

```
trans_var ← trans_exp;
frame1 <= frame2*trans_var;
when_changing frame1 also do trans_var ← inv(frame2)*frame1;
```

C.3 Data Structures

Every "active" instance of a variable in an AL program is represented by a *graph node* containing the following information:

value – a pointer to a block of storage containing the current value of the variable. The size of the block depends on the data type of the variable.

invmark – an integer. If **invmark**(*var*)=*0*, then **value**(*var*) points to a "valid" value. Otherwise, **value**(*var*) contains an old value which may no longer be correct.

calculators – a list of expressions which may be used to calculate new values of the variable. Each such expression is represented by a graph node, whose contents are described below.

dependents – a list of (expression) nodes in the graph structure whose value directly depends on the variable. If a node *e* uses *var*, then *e* will be in **dependents**(*var*).

side_effects – a list of additional statements that are to be executed whenever the variable is changed by the AL program.

Nodes associated with "continuously" evaluated expressions are very similar to those associated with variables. They have the following fields:

value – pointer to the value.

invmark – integer. Same use as for variables.

needed – list of (variable) nodes whose values must be valid in order for this expression to produce a valid value.

dependents – list of (variable) nodes currently calculated by this expression.

code – pointer to the code needed to evaluate the expression. Executing **code**(*e*) returns a pointer to a block of storage containing the value of the expression.[5]

---

[5] In the actual version used by the runtime system, there is also a pointer to an "environment" of address bindings to be used in interpreting the code. This will be left out of our discussion.

C.4 Algorithms

Explicit updating of values in AL occurs when the interpreter executes an assignment statement. This is accomplished by a call to change(*var*,*val*), which is given below. Briefly, this procedure works by invalidating all nodes whose value depends on the node being changed, putting the new value away, and executing all side-effect procedures.

```
procedure change(pointer(variable node) var;pointer(value) val);
    begin
    pointer(value) oldval;[6]
    pointer(statement) s;
    invalidate(var,time←time+1);
        { Causes all nodes dependent on var to be marked invalid. (see below)}
    value(var) ← val;
    invmark(var) ← 0; { We have put away a good value }
    for each s such that s ∈ side effects(var) do
        execute(s);
    end;
```

invalidate(*n*,*t*) is called as a subroutine by change. *n* is marked "invalid", and the procedure is called recursively to invalidate all dependents of *n*. One difficulty with the algorithm is that there may be dependency cycles, which could cause infinite recursion. Therefore, *t* is used as a cycle breaker. Any time a node and its dependents are to be invalidated, a unique integer is generated for use as the "invalid" mark. Any time invalidate encounters this value, it just returns right away.

```
procedure invalidate(pointer(node) n;integer t);
    if invmark(n)≠ t then
        begin
        { Note that this procedure works on both flavors of graph node. }
        pointer(node) d;
        invmark(n) ← t;
        for each d such that d ∈ dependents(n) do
            invalidate(d,t);
        end;
```

When the interpreter requires the value of a graph node, it calls getvalue(*n*). If *n* is invalid, then evalnode is called to try to get a good value. After the call to evalnode the value is returned, even if evalnode failed to produce a "valid" answer. This may seem a bit risky – an alternative would be to generate a runtime error message – but doesn't seem to cause much trouble in practice. The assumption embodied in this decision is that even an old value may still be approximately correct.[7]

---

[6] Actually, *oldval* is a global variable. It is used to allow side effect procedures to have access to both the old and new values of the variable being changed.

[7] An "invalid" result most often comes from an explicit change to a "by" trans linking two rigidly affixed frames. The problem is that it is impossible to tell which frame has moved.

```
pointer(value) procedure getvalue(pointer(node) n);
    begin
    if invmark(n) ≠ 0 then
        evalnode(n,time←time+1);
        { time generates a unique invocation count (see below) }
    return(value(n));
    end;
```

evalnode(n,t) is the workhorse procedure for the automatic updating. If n contains a valid value, then the procedure just returns. Otherwise, if n is a variable, then the procedure tries to obtain a valid value from one of the calculators of n. To avoid needless recomputation, an initial check is made to see if one of the calculator nodes is already valid before trying to re-evaluate any expressions. If n is an expression node, then the procedure attempts to get valid values for all the needed parameters to n. If it succeeds, the expression is evaluated and the value tucked away. Once again, t is used as a cycle breaker. Each time evalnode is called from a "top" level – i.e., from getvalue – a unique value is generated for t. Any time evalnode is called for an invalid node n, t is put into invmark(n); if the procedure encounters a node with an invmark = t, then it just gives up right away.

```
procedure evalnode(pointer(node) n;integer t);
    begin
    if invmark(n)=0 or invmark(n) = t then
        return; { value(n) is either valid or cannot be validated }
    invmark(n)←t; { This is used to break cycles }
    if kind(n) = "variable" then
        begin
        pointer(node) e;
        { First check to see if there is some calculator
          with a currently valid value. }
        for each e such that e ∈ calculators(n) do
            if invmark(e) = 0 then
                begin
                { This one will do.}
                value(n) ← value(e);
                invmark(n) ← 0;
                return;
                end;
        { If we get here, no calculator had a value "ready to go".
          We will now try to get new values the hard way.}
        for each e such that e ∈ calculators(n) do
                begin
                evalnode(e,t); { Try to find a new value.}
                if invmark(e) = 0 then
                        begin
                        { Eureka! }
                        value(n) ← value(e);
                        invmark(n) ← 0;
                        return;
                        end;
                end;
        end
```

```
    else
        begin { Must be an expression node}
        pointer(node) x;
        { Try to evaluate all arguments}
        for each x such that x ∈ needed(n) do
            begin
            evalnode(x,t);
            { Give up if could not get valid value }
            if invmark(x)≠0 then
                return;
            end;
        value(n) ← evaluate(code(n));
        invmark(n)←0;
        end;
end;
```

C.5 Fine Points

Generating Unique Invalidity Marks

One minor difficulty with using the cycle-breaking mechanism discussed here on a small word-size machine is that the invocation counter may overflow: If *t* gets to 0, then invalidate will mark nodes "valid", and evalnode will give up on encountering a valid node. This is readily remedied by the simple expedient of checking for overflow after *time* is incremented:

```
procedure bumptime;
    begin
    pointer(node) n;
    time ← time+1;
    if overflow then
        begin
        time ← 1;
        for each n such that n ∈ set_of_all_graph_nodes do
            if invmark(n)≠0 then invmark(n)← -1;
        end;
    end;
```

Changes in Dependency Relations

Another important point with this method is that before a calculator is deleted, any nodes that depend on it should be validated. The easiest way to do this is to run down the dependents list and call getvalue for each element.

An interesting bug

The original implementation of invalidate always used -1 as the invalidity mark, and gave up whenever it encountered an invalid node:

```
procedure invalidate(pointer(node) n);
    if invmark(n)≠ 0 then
        begin
        { Note that this procedure works on both flavors of graph node. }
        pointer(node) d;
        invmark(n) ← -1;
        for each d such that d ∈ dependents(n) do
            invalidate(d);
        end;
```

At first glance, this seems perfectly safe. Why go to the trouble of invalidating a node whose value is already known to be bad? Unfortunately, the following code sequence then produces the wrong result.

```
affix a to b rigidly;
affix b to c rigidly;
   :
{ Assume that, here, a,b, and c all have valid values. }
a ← value_1; { Invalidates b and c }
c ← value_2; { Implicitly changes b and c }
move blue to a; { goes to the wrong place }
```

Here the change to *c* results in a call to **invalidate**(*b*). Since *b* is already invalid, **invalidate**(*a*) is never called, thus leaving *value_1* marked as an incorrect "valid" value to *a*.

Appendix D.

Object Model for Box Assembly

This section gives a computer generated printout of the object model for the box assembly, together with several other objects.

```
OBJECT:WORKSTATION  KIND=ASSEMBLY
    SUBPART BMANIP
        LOCUS_ESTIMATE IS <FLUENT 131223>
        DETERM_ESTIMATE IS <FLUENT 131230>
    SUBPART TABLE AT NILTRANS
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    OBJECT:BMANIP  KIND=ASSEMBLY
        PARENT = WORKSTATION LOCUS ITEM IS ITEM_0367
    OBJECT:TABLE  KIND=PART
        DESCR =( .100e4, .100e4, .000)
        PARENT = WORKSTATION  XF ITEM IS NILTRANSITM
        FEATURE TABLETOP AT NILTRANS
            DETERM_ESTIMATE IS FULLY DETERMINED
            DETERM_TEMPLATE IS FULLY DETERMINED
        TABLETOP: SURFACE(SMOOTH  PLANAR POLYGON
            (.100e4,.100e4) (-.100e4,.100e4) (-.100e4,-.100e4) (.100e4,-.100e4))

OBJECT:BOX  KIND=ASSEMBLY
    SUBPART BXBTM AT NILTRANS
        LOCUS_ESTIMATE IS <FLUENT 131014>
        DETERM_ESTIMATE IS <FLUENT 131021>
        DETERM_TEMPLATE IS FULLY DETERMINED
    SUBPART COVER AT TRANS(NILROTN, VECTOR(.000, .000, 4.90))
        LOCUS_ESTIMATE IS <FLUENT 131026>
        DETERM_ESTIMATE IS <FLUENT 131033>
        DETERM_TEMPLATE IS FULLY DETERMINED
    SUBPART SC1 AT TRANS(ROTN(YHAT, 180.000*DEG), VECTOR(3.85, 3.20, 5.84))
        LOCUS_ESTIMATE IS <FLUENT 131045>
        DETERM_ESTIMATE IS <FLUENT 131052>
        DETERM_TEMPLATE IS FULLY DETERMINED
    SUBPART SC2 AT TRANS(ROTN(YHAT, 180.000*DEG), VECTOR(-3.85, 3.20, 5.84))
        LOCUS_ESTIMATE IS <FLUENT 131057>
        DETERM_ESTIMATE IS <FLUENT 131064>
        DETERM_TEMPLATE IS FULLY DETERMINED
    SUBPART SC3 AT TRANS(ROTN(YHAT, 180.000*DEG), VECTOR(-3.85,-3.20, 5.84))
        LOCUS_ESTIMATE IS <FLUENT 131071>
        DETERM_ESTIMATE IS <FLUENT 131076>
        DETERM_TEMPLATE IS FULLY DETERMINED
    SUBPART SC4 AT TRANS(ROTN(YHAT, 180.000*DEG), VECTOR(3.85,-3.20, 5.84))
        LOCUS_ESTIMATE IS <FLUENT 131103>
        DETERM_ESTIMATE IS <FLUENT 131110>
        DETERM_TEMPLATE IS FULLY DETERMINED
    OBJECT:BXBTM  KIND=PART
        PARENT = BOX  XF ITEM IS ITEM_0293 LOCUS ITEM IS ITEM_0340
        SUBPART BB1 AT TRANS(ROTN(YHAT, 180.000*DEG), VECTOR(3.85, 3.20, 4.90))
            DETERM_ESTIMATE IS FULLY DETERMINED
            DETERM_TEMPLATE IS FULLY DETERMINED
        SUBPART BB2 AT TRANS(ROTN(YHAT, 180.000*DEG), VECTOR(-3.85, 3.20, 4.90))
            DETERM_ESTIMATE IS FULLY DETERMINED
            DETERM_TEMPLATE IS FULLY DETERMINED
        SUBPART BB3 AT TRANS(ROTN(YHAT, 180.000*DEG), VECTOR(-3.85,-3.20, 4.90))
            DETERM_ESTIMATE IS FULLY DETERMINED
            DETERM_TEMPLATE IS FULLY DETERMINED
        SUBPART BB4 AT TRANS(ROTN(YHAT, 180.000*DEG), VECTOR(3.85,-3.20, 4.90))
            DETERM_ESTIMATE IS FULLY DETERMINED
            DETERM_TEMPLATE IS FULLY DETERMINED
        SUBPART BIB AT TRANS(NILROTN, VECTOR(.000, .000, 1.00))
```

```
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE B1S AT TRANS(ROTN(XHAT, 90.000*DEG), VECTOR(.000,-3.80, 2.45))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE B2S AT TRANS(ROTN(YHAT, 90.000*DEG), VECTOR(4.45, .000, 2.45))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE B3S AT TRANS(ROTN(-XHAT, 90.000*DEG), VECTOR(.000, 3.80, 2.45))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE B4S AT TRANS(ROTN(-YHAT, 90.000*DEG), VECTOR(.000,-4.45, 2.45))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE B5S AT TRANS(ROTN(YHAT, 180.000*DEG), NILVECT)
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BTS AT TRANS(NILROTN, VECTOR(.000, .000, 4.90))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BV1 AT TRANS(NILROTN, VECTOR(-4.45,-3.80, .000))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BV2 AT TRANS(NILROTN, VECTOR(4.45,-3.80, .000))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BV3 AT TRANS(NILROTN, VECTOR(4.45, 3.80, .000))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BV4 AT TRANS(NILROTN, VECTOR(4.45, 3.80, .000))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BV5 AT TRANS(NILROTN, VECTOR(-4.45,-3.80, 4.90))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BV6 AT TRANS(NILROTN, VECTOR(4.45,-3.80, 4.90))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BV7 AT TRANS(NILROTN, VECTOR(4.45, 3.80, 4.90))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BV8 AT TRANS(NILROTN, VECTOR(4.45, 3.80, 4.90))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BE1 AT TRANS(ROTN(VECTOR(.357, .863, .357), 98.421*DEG), VECTOR(-4.45,-3.80, .000))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BE2 AT TRANS(ROTN(VECTOR(.863, .357, .357),-98.421*DEG), VECTOR(4.45,-3.80, .000))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BE3 AT TRANS(ROTN(VECTOR(.679, .281, .679), 148.600*DEG), VECTOR(4.45, 3.80, .000))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BE4 AT TRANS(ROTN(VECTOR(.281, .679, .679),-148.600*DEG), VECTOR(4.45, 3.80, .000))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BE5 AT TRANS(ROTN(VECTOR(.679, .281, .679),-148.600*DEG), VECTOR(-4.45,-3.80, 4.90))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BE6 AT TRANS(ROTN(VECTOR(.863, .357, .357),-98.421*DEG), VECTOR(4.45,-3.80, 4.90))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BE7 AT TRANS(ROTN(VECTOR(.357, .863, .357),-98.421*DEG), VECTOR(4.45, 3.80, 4.90))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BE8 AT TRANS(ROTN(VECTOR(.281, .679, .679), 148.600*DEG), VECTOR(4.45, 3.80, 4.90))
    DETERM_ESTIMATE IS FULLY DETERMINED
    DETERM_TEMPLATE IS FULLY DETERMINED
FEATURE BE9 AT TRANS(ROTN(ZHAT,-135.000*DEG), VECTOR(-4.45,-3.80, .000))
```

```
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    FEATURE BE10 AT TRANS(ROTN(ZHAT,-45.000*DEG), VECTOR(4.45,-3.80, .000))
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    FEATURE BE11 AT TRANS(ROTN(ZHAT, 45.000*DEG), VECTOR(4.45, 3.80, .000))
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    FEATURE BE12 AT TRANS(ROTN(ZHAT, 135.000*DEG), VECTOR(4.45, 3.80, .000))
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    OBJECT:BB1  KIND=BORE
        DESCR =PROFILE: NSECTS =2 LEN = .000     CROSS SECTION =CIRCLE
            DIR =  1.00 :   (-1,CIRCLE, .525, .525, 2.54, .000,ITEM_0011)
            DIR =  1.00 :   (-1,CIRCLE, .525, .000, .000, .000,SMOOTH)
        PARENT = BXBTM  XF ITEM IS ITEM_0177
    OBJECT:BB2  KIND=BORE
        DESCR =PROFILE: NSECTS =2 LEN = .000     CROSS SECTION =CIRCLE
            DIR =  1.00 :   (-1,CIRCLE, .525, .525, 2.54, .000,ITEM_0011)
            DIR =  1.00 :   (-1,CIRCLE, .525, .000, .000, .000,SMOOTH)
        PARENT = BXBTM  XF ITEM IS ITEM_0182
    OBJECT:BB3  KIND=BORE
        DESCR =PROFILE: NSECTS =2 LEN = .000     CROSS SECTION =CIRCLE
            DIR =  1.00 :   (-1,CIRCLE, .525, .525, 2.54, .000,ITEM_0011)
            DIR =  1.00 :   (-1,CIRCLE, .525, .000, .000, .000,SMOOTH)
        PARENT = BXBTM  XF ITEM IS ITEM_0187
    OBJECT:BB4  KIND=BORE
        DESCR =PROFILE: NSECTS =2 LEN = .000     CROSS SECTION =CIRCLE
            DIR =  1.00 :   (-1,CIRCLE, .525, .525, 2.54, .000,ITEM_0011)
            DIR =  1.00 :   (-1,CIRCLE, .525, .000, .000, .000,SMOOTH)
        PARENT = BXBTM  XF ITEM IS ITEM_0192
    OBJECT:BIB  KIND=BORE
        DESCR =PROFILE: NSECTS =2 LEN = 3.90     CROSS SECTION =ITEM_0194
            DIR =  1.00 :   (-1,ITEM_0194, 1.00, 1.00, 3.90, .000,SMOOTH)
            DIR =  1.00 :   (-1,ITEM_0194, 1.00, .000, .000, .000,SMOOTH)
        PARENT = BXBTM  XF ITEM IS ITEM_0197
    B1S: SURFACE(SMOOTH  PLANAR POLYGON
        (4.45,2.45) (-4.45,2.45) (-4.45,-2.45) (4.45,-2.45))
    B2S: SURFACE(SMOOTH  PLANAR POLYGON
        (3.80,2.45) (-3.80,2.45) (-3.80,-2.45) (3.80,-2.45))
    B3S: SURFACE(SMOOTH  PLANAR POLYGON
        (4.45,2.45) (-4.45,2.45) (-4.45,-2.45) (4.45,-2.45))
    B4S: SURFACE(SMOOTH  PLANAR POLYGON
        (3.80,2.45) (-3.80,2.45) (-3.80,-2.45) (3.80,-2.45))
    BBS: SURFACE(SMOOTH  PLANAR POLYGON
        (4.45,3.80) (-4.45,3.80) (-4.45,-3.80) (4.45,-3.80))
    BTS: SURFACE(SMOOTH  PLANAR POLYGON
        (4.45,3.80) (-4.45,3.80) (-4.45,-3.80) (4.45,-3.80))
    BV1: (SMOOTH, .000, .000)
    BV2: (SMOOTH, .000, .000)
    BV3: (SMOOTH, .000, .000)
    BV4: (SMOOTH, .000, .000)
    BV5: (SMOOTH, .000, .000)
    BV6: (SMOOTH, .000, .000)
    BV7: (SMOOTH, .000, .000)
    BV8: (SMOOTH, .000, .000)
    BE1: (1,CIRCLE, .000, .000, 8.90, .785,SMOOTH)
    BE2: (1,CIRCLE, .000, .000, 7.60, .785,SMOOTH)
    BE3: (1,CIRCLE, .000, .000, .000, .785,SMOOTH)
    BE4: (1,CIRCLE, .000, .000, 7.60, .785,SMOOTH)
    BE5: (1,CIRCLE, .000, .000, 8.90, .785,SMOOTH)
    BE6: (1,CIRCLE, .000, .000, 7.60, .785,SMOOTH)
    BE7: (1,CIRCLE, .000, .000, .000, .785,SMOOTH)
    BE8: (1,CIRCLE, .000, .000, 7.60, .785,SMOOTH)
    BE9: (1,CIRCLE, .000, .000, 4.90, .785,SMOOTH)
    BE10: (1,CIRCLE, .000, .000, 4.90, .785,SMOOTH)
    BE11: (1,CIRCLE, .000, .000, 4.90, .785,SMOOTH)
    BE12: (1,CIRCLE, .000, .000, 4.90, .785,SMOOTH)
OBJECT:COVER  KIND=PART
```

```
    PARENT = BOX  XF ITEM IS ITEM_0295 LOCUS ITEM IS ITEM_0342
    SUBPART CB1 AT TRANS(ROTN(YHAT, 180.000*DEG), VECTOR(3.85, 3.20, .300))
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    SUBPART CB2 AT TRANS(ROTN(YHAT, 180.000*DEG), VECTOR(-3.85, 3.20, .300))
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    FEATURE C1S AT TRANS(ROTN(XHAT, 90.000*DEG), VECTOR(.000,-3.80, .150))
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    FEATURE C2S AT TRANS(ROTN(YHAT, 90.000*DEG), VECTOR(4.45, .000, .150))
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    FEATURE C3S AT TRANS(ROTN(-XHAT, 90.000*DEG), VECTOR(.000, 3.80, .150))
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    FEATURE C4S AT TRANS(ROTN(XHAT, 90.000*DEG), VECTOR(.000,-4.45, .150))
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    FEATURE CBS AT TRANS(ROTN(YHAT, 180.000*DEG), NILVECT)
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    FEATURE CTS AT TRANS(NILROTN, VECTOR(.000, .000, .300))
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    OBJECT:CB1  KIND=BORE
        DESCR =PROFILE: NSECTS =2 LEN = .000     CROSS SECTION =CIRCLE
            DIR =  1.00 :   [-1,CIRCLE, .900, .700, .100, .000,SMOOTH]
            DIR =  1.00 :   [-1,CIRCLE, .700, .700, .200, .000,SMOOTH]
        PARENT = COVER  XF ITEM IS ITEM_0122
    OBJECT:CB2  KIND=BORE
        DESCR =PROFILE: NSECTS =2 LEN = .000     CROSS SECTION =CIRCLE
            DIR =  1.00 :   [-1,CIRCLE, .900, .700, .100, .000,SMOOTH]
            DIR =  1.00 :   [-1,CIRCLE, .700, .700, .200, .000,SMOOTH]
        PARENT = COVER  XF ITEM IS ITEM_0130
    C1S: SURFACE(SMOOTH  PLANAR POLYGON
        (4.45,.150) (-4.45,.150) (-4.45,-.150) (4.45,-.150))
    C2S: SURFACE(SMOOTH  PLANAR POLYGON
        (3.80,.150) (-3.80,.150) (-3.80,-.150) (3.80,-.150))
    C3S: SURFACE(SMOOTH  PLANAR POLYGON
        (4.45,.150) (-4.45,.150) (-4.45,-.150) (4.45,-.150))
    C4S: SURFACE(SMOOTH  PLANAR POLYGON
        (3.80,.150) (-3.80,.150) (-3.80,-.150) (3.80,-.150))
    CBS: SURFACE(SMOOTH  PLANAR POLYGON
        (4.45,3.80) (-4.45,3.80) (-4.45,-3.80) (4.45,-3.80))
    CTS: SURFACE(SMOOTH  PLANAR POLYGON
        (4.45,3.80) (-4.45,3.80) (-4.45,-3.80) (4.45,-3.80))
OBJECT:SC1  KIND=PART
    PARENT = BOX  XF ITEM IS ITEM_0297 LOCUS ITEM IS ITEM_0344
    SUBPART SC1.SCRWSH AT NILTRANS
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    SUBPART SC1.SCRWSLT AT NILTRANS
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    OBJECT:SC1.SCRWSH  KIND=SHAFT
        DESCR =PROFILE: NSECTS =3 LEN = 3.18     CROSS SECTION =CIRCLE
            DIR =  1.00 :   [1,CIRCLE, .826, .826, .635, .000,SMOOTH]
            DIR =  1.00 :   [1,CIRCLE, .635, .635, 1.91, .000,ITEM_0011]
            DIR =  1.00 :   [1,CIRCLE, .635, .000, .635, .000,SMOOTH]
        PARENT = SC1  XF ITEM IS ITEM_0270
    OBJECT:SC1.SCRWSLT  KIND=BORE
        DESCR =PROFILE: NSECTS =2 LEN = .508     CROSS SECTION =ITEM_0263
            DIR =  1.00 :   [-1,ITEM_0263, .635, .635, .508, .000,SMOOTH]
            DIR =  1.00 :   [-1,ITEM_0263, .635, .000, .000, .000,SMOOTH]
        PARENT = SC1  XF ITEM IS ITEM_0268
OBJECT:SC2  KIND=PART
    PARENT = BOX  XF ITEM IS ITEM_0298 LOCUS ITEM IS ITEM_0346
    SUBPART SC2.SCRWSH AT NILTRANS
```

```
            DETERM_ESTIMATE IS FULLY DETERMINED
            DETERM_TEMPLATE IS FULLY DETERMINED
        SUBPART SC2.SCRWSLT AT NILTRANS
            DETERM_ESTIMATE IS FULLY DETERMINED
            DETERM_TEMPLATE IS FULLY DETERMINED
        OBJECT:SC2.SCRWSH  KIND=SHAFT
            DESCR =PROFILE: NSECTS =3 LEN = 3.18     CROSS SECTION =CIRCLE
                DIR =  1.00 :    (1,CIRCLE, .826, .826, .635, .000,SMOOTH)
                DIR =  1.00 :    (1,CIRCLE, .635, .635, 1.91, .000,ITEM_0011)
                DIR =  1.00 :    (1,CIRCLE, .635, .000, .635, .000,SMOOTH)
            PARENT = SC2  XF ITEM IS ITEM_0270
        OBJECT:SC2.SCRWSLT  KIND=BORE
            DESCR =PROFILE: NSECTS =2 LEN = .508     CROSS SECTION =ITEM_0263
                DIR =  1.00 :    (-1,ITEM_0263, .635, .635, .508, .000,SMOOTH)
                DIR =  1.00 :    (-1,ITEM_0263, .635, .000, .000, .000,SMOOTH)
            PARENT = SC2  XF ITEM IS ITEM_0268
    OBJECT:SC3  KIND=PART
        PARENT = BOX  XF ITEM IS ITEM_0301 LOCUS ITEM IS ITEM_0348
        SUBPART SC3.SCRWSH AT NILTRANS
            DETERM_ESTIMATE IS FULLY DETERMINED
            DETERM_TEMPLATE IS FULLY DETERMINED
        SUBPART SC3.SCRWSLT AT NILTRANS
            DETERM_ESTIMATE IS FULLY DETERMINED
            DETERM_TEMPLATE IS FULLY DETERMINED
        OBJECT:SC3.SCRWSH  KIND=SHAFT
            DESCR =PROFILE: NSECTS =3 LEN = 3.18     CROSS SECTION =CIRCLE
                DIR =  1.00 :    (1,CIRCLE, .826, .826, .635, .000,SMOOTH)
                DIR =  1.00 :    (1,CIRCLE, .635, .635, 1.91, .000,ITEM_0011)
                DIR =  1.00 :    (1,CIRCLE, .635, .000, .635, .000,SMOOTH)
            PARENT = SC3  XF ITEM IS ITEM_0270
        OBJECT:SC3.SCRWSLT  KIND=BORE
            DESCR =PROFILE: NSECTS =2 LEN = .508     CROSS SECTION =ITEM_0263
                DIR =  1.00 :    (-1,ITEM_0263, .635, .635, .508, .000,SMOOTH)
                DIR =  1.00 :    (-1,ITEM_0263, .635, .000, .000, .000,SMOOTH)
            PARENT = SC3  XF ITEM IS ITEM_0268
    OBJECT:SC4  KIND=PART
        PARENT = BOX  XF ITEM IS ITEM_0303 LOCUS ITEM IS ITEM_0350
        SUBPART SC4.SCRWSH AT NILTRANS
            DETERM_ESTIMATE IS FULLY DETERMINED
            DETERM_TEMPLATE IS FULLY DETERMINED
        SUBPART SC4.SCRWSLT AT NILTRANS
            DETERM_ESTIMATE IS FULLY DETERMINED
            DETERM_TEMPLATE IS FULLY DETERMINED
        OBJECT:SC4.SCRWSH  KIND=SHAFT
            DESCR =PROFILE: NSECTS =3 LEN = 3.18     CROSS SECTION =CIRCLE
                DIR =  1.00 :    (1,CIRCLE, .826, .826, .635, .000,SMOOTH)
                DIR =  1.00 :    (1,CIRCLE, .635, .635, 1.91, .000,ITEM_0011)
                DIR =  1.00 :    (1,CIRCLE, .635, .000, .635, .000,SMOOTH)
            PARENT = SC4  XF ITEM IS ITEM_0270
        OBJECT:SC4.SCRWSLT  KIND=BORE
            DESCR =PROFILE: NSECTS =2 LEN = .508     CROSS SECTION =ITEM_0263
                DIR =  1.00 :    (-1,ITEM_0263, .635, .635, .508, .000,SMOOTH)
                DIR =  1.00 :    (-1,ITEM_0263, .635, .000, .000, .000,SMOOTH)
            PARENT = SC4  XF ITEM IS ITEM_0268

OBJECT:PRACK  KIND=PART
    SUBPART PRACK.B1 AT TRANS(ROTN(YHAT, 180.000*DEG), VECTOR(-2.54,-3.81, 2.54))
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    SUBPART PRACK.B2 AT TRANS(ROTN(YHAT, 180.000*DEG), VECTOR(2.54,-3.81, 2.54))
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    FEATURE PRKTS AT TRANS(NILROTN, VECTOR(.000, .000, 2.54))
        DETERM_ESTIMATE IS FULLY DETERMINED
        DETERM_TEMPLATE IS FULLY DETERMINED
    OBJECT:PRACK.B1  KIND=BORE
        DESCR =PROFILE: NSECTS =3 LEN = 2.41     CROSS SECTION =CIRCLE
            DIR =  1.00 :        (-1,CIRCLE, 1.91, .508, .635, .000,SMOOTH)
            DIR =  1.00 :        (-1,CIRCLE, .508, .508, 1.78, .000,SMOOTH)
```

```
            DIR =  1.00 :       [-1,CIRCLE, .500, .000, .000, .000,SMOOTH]
        PARENT = PRACK  XF ITEM IS ITEM_0315
    OBJECT:PRACK.B2  KIND=BORE
        DESCR =PROFILE: NSECTS =3 LEN = 2.41     CROSS SECTION =CIRCLE
            DIR =  1.00 :       [-1,CIRCLE, 1.91, .500, .635, .000,SMOOTH]
            DIR =  1.00 :       [-1,CIRCLE, .500, .500, 1.78, .000,SMOOTH]
            DIR =  1.00 :       [-1,CIRCLE, .500, .000, .000, .000,SMOOTH]
        PARENT = PRACK  XF ITEM IS ITEM_0320
    PRKTS: SURFACE(SMOOTH  PLANAR POLYGON
        (3.81,5.08) (-3.81,5.08) (-3.81,-5.08) (3.81,-5.08))

OBJECT:ITEM_0321  KIND=SHAFT
    DESCR =PROFILE: NSECTS =4 LEN = 4.51     CROSS SECTION =CIRCLE
        DIR =  1.00 :    [1,CIRCLE, .000, .445, .445, .000,SMOOTH]
        DIR =  1.00 :    [1,CIRCLE, .445, .445, 1.27, .000,SMOOTH]
        DIR =  1.00 :    [1,CIRCLE, .635, .635, 1.40, .000,SMOOTH]
        DIR =  1.00 :    [1,CIRCLE, .635, .000, 1.40, .000,SMOOTH]

OBJECT:PIN1  KIND=SHAFT
    DESCR =PROFILE: NSECTS =4 LEN = 4.51     CROSS SECTION =CIRCLE
        DIR =  1.00 :    [1,CIRCLE, .000, .445, .445, .000,SMOOTH]
        DIR =  1.00 :    [1,CIRCLE, .445, .445, 1.27, .000,SMOOTH]
        DIR =  1.00 :    [1,CIRCLE, .635, .635, 1.40, .000,SMOOTH]
        DIR =  1.00 :    [1,CIRCLE, .635, .000, 1.40, .000,SMOOTH]
```

Appendix E.

Examples of Location and Accuracy Calculations

E.1 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 of Section 7.6.5 to the resulting sets of equations. Here, we have placed our box (whose model is given in Appendix D) into an open-topped fixture, as illustrated in Figure E.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 corner 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*, *bjl.sb*, inside_of)

where *bxbtm* is the bottom of the box, and *bjl.sb* is the bottom of the fixture. This produces the constraint set:

```
 YHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤  5.000 -  YHAT . PV
-XHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤  4.000 - -XHAT . PV
-YHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤  5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤  4.000 -  XHAT . PV
 YHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  5.000 -  YHAT . PV
-XHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  4.000 - -XHAT . PV
-YHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  4.000 -  XHAT . PV
 YHAT*R* 5.85* VECTOR( .760, .649, .000) ≤  5.000 -  YHAT . PV
-XHAT*R* 5.85* VECTOR( .760, .649, .000) ≤  4.000 - -XHAT . PV
-YHAT*R* 5.85* VECTOR( .760, .649, .000) ≤  5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR( .760, .649, .000) ≤  4.000 -  XHAT . PV
 YHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤  5.000 -  YHAT . PV
-XHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤  4.000 - -XHAT . PV
-YHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤  5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤  4.000 -  XHAT . PV
0 =  .000 -  ZHAT . PV
        WHERE R =  NILROTN*ROTN(-ZHAT,W)
              PV = [X,Y,Z]
```

Applying the algorithm given in Section 7.6.5 gives two possible orientations:

```
ESTIMATE LIST:
ITEM416:
X:      -.204 TO  .204
Y:      -.555 TO  .555
Z:      -.001 TO  .001
W:       87.368*DEG TO  92.632*DEG
COS(W0) =  .000  SIN(W0) =  1.000
COS(DW) =  .999  R =  .046
```

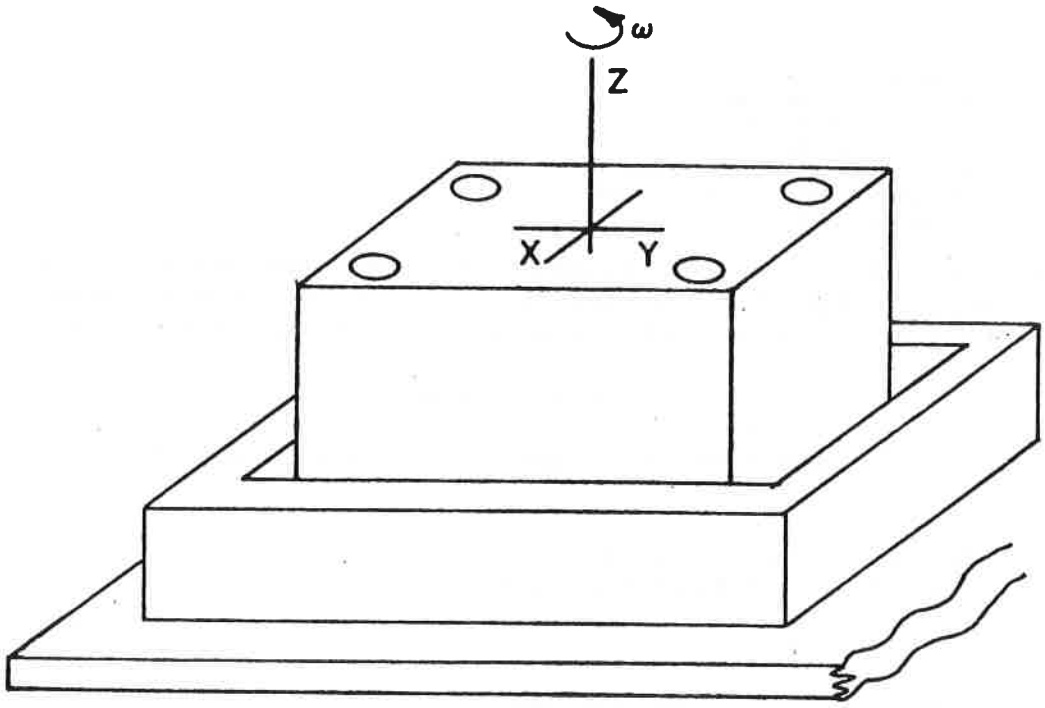


Figure E.1. Box in Fixture

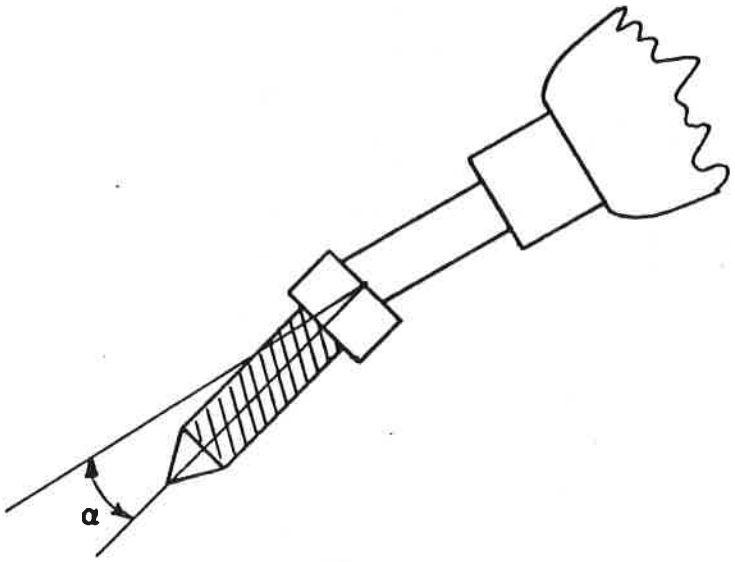


Figure E.2. Screw on Driver

```
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 illustrate 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)
(contacts, *be9*, *bj1.s2*, extent_irrelevant)

This gives:

```
-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) ≤  4.000 - -XHAT . PV
-YHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤  5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤  4.000 -  XHAT . PV
 YHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  5.000 -  YHAT . PV
-XHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  4.000 - -XHAT . PV
-YHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  4.000 -  XHAT . PV
 YHAT*R* 5.85* VECTOR( .760, .649, .000) ≤  5.000 -  YHAT . PV
-XHAT*R* 5.85* VECTOR( .760, .649, .000) ≤  4.000 - -XHAT . PV
-YHAT*R* 5.85* VECTOR( .760, .649, .000) ≤  5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR( .760, .649, .000) ≤  4.000 -  XHAT . PV
 YHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤  5.000 -  YHAT . PV
-XHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤  4.000 - -XHAT . PV
-YHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤  5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤  4.000 -  XHAT . PV
-XHAT*R* 5.85* VECTOR(-.760,-.649, .000) = -4.000 - -XHAT . PV
0 =  .000 -  ZHAT . PV
        WHERE R =  NILROTN*ROTN(-ZHAT,?)

ESTIMATE LIST:
ITEM426:
X:      -.000 TO  .200
Y:      -.550 TO  .550
Z:      -.001 TO  .001
W:      -92.632*DEG TO -90.000*DEG
COS(W0) = -.023  SIN(W0) = -1.000
COS(DW) =  1.000  R =  .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*, *bj1.sb*, inside_of)
(contacts, *be9*, *bj1.s2*, extent_irrelevant)
(contacts, *be10*, *bj1.s3*, extent_irrelevant)
(contacts, *be11*, *bj1.s4*, extent_irrelevant)

and wind up with the final estimate:

```
-YHAT*R* 1.00* VECTOR( .707,-.707, .000) ≤ -.707
 XHAT*R* 1.00* VECTOR( .707, .707, .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) ≤  4.000 - -XHAT . PV
-YHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤  5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤  4.000 -  XHAT . PV
 YHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  5.000 -  YHAT . PV
-XHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  4.000 - -XHAT . PV
-YHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  4.000 -  XHAT . PV
 YHAT*R* 5.85* VECTOR( .760, .649, .000) ≤  5.000 -  YHAT . PV
-XHAT*R* 5.85* VECTOR( .760, .649, .000) ≤  4.000 - -XHAT . PV
-YHAT*R* 5.85* VECTOR( .760, .649, .000) ≤  5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR( .760, .649, .000) ≤  4.000 -  XHAT . PV
 YHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤  5.000 -  YHAT . PV
-XHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤  4.000 - -XHAT . PV
-YHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤  5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤  4.000 -  XHAT . PV
-YHAT*R* 5.85* VECTOR( .760,-.649, .000) = -5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR( .760, .649, .000) = -4.000 -  XHAT . PV
-XHAT*R* 5.85* VECTOR(-.760,-.649, .000) = -4.000 - -XHAT . PV
0 =  .000 -  ZHAT . PV
       WHERE R =  NILROTN*ROTN(-ZHAT,?)

ESTIMATE LIST:
ITEM442:
X:       .000 TO  .000
Y:       .379 TO  .381
Z:      -.001 TO  .001
W:      -92.632*DEG TO -92.603*DEG
COS(W0) = -.046  SIN(W0) = -.999
COS(DW) =  1.000  R =  .000
```

E.2 Screw on Driver

This example illustrates use of the differential approximation methods of Section 7.9 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:

$$\Delta box = \text{transl}(\lambda\hat{x} + \mu\hat{y}) * \text{rot}(\hat{z},\gamma)$$

where

-0.9 inches $\leq \lambda \leq$ 0.9 inches
-0.2 inches $\leq \mu \leq$ 0.2 inches
-5 degrees $\leq \gamma \leq$ 5 degrees

The screw is held on the end of a driver, as shown in Figure E.2, and the driver is held in

the hand. We will 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.

$$\Delta hand = \text{transl}(\text{vector}(\delta_x,\delta_y,\delta_z))*\text{rot}(\hat{x},\phi_x)*\text{rot}(\hat{y},\phi_y)*\text{rot}(\hat{z},\phi_z)$$

where

$-0.05 \text{ inches} \le \delta_x, \delta_y, \delta_z \le 0.05 \text{ inches}$

$-0.25 \text{ degrees} \le \phi_x, \phi_y, \phi_z \le 0.25 \text{ degrees}$

Likewise, the screw can wobble about the tip of the driver.

$$\begin{aligned}\Delta T_{ds} &= \text{rot}(\hat{x},\alpha)*\text{rot}(\hat{y},\beta)\\ &\simeq I + \alpha M_x + \beta M_y\end{aligned}$$

where

$-5 \text{ degrees} \le \alpha \le 5 \text{ degrees}$

$-5 \text{ degrees} \le \beta \le 5 \text{ degrees}$

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.

$$\begin{aligned}T_{ht} &= hole^{-1}*tip\\ &= (box*T_{bh})^{-1}*(hand*T_{hd}*T_{ds}*T_{st})\\ &= T_{bh}^{-1}*box^{-1}*hand*T_{hd}*T_{ds}*T_{st}\end{aligned}$$

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}	$\simeq$ trans(nilrotn,vector(*3.85,3.20,4.90*)) (Distances in cm)
T_{hd}	$\simeq$ niltrans
T_{ds}	$\simeq$ trans(nilrotn,vector(*0,0,25.4*))
T_{st}	$\simeq$ trans(nilrotn,vector(*0,0,3.18*))
box	$\simeq$ trans(nilrotn,vector(*45.7,101.6,0*))

$$hand \quad \tilde{=} \text{trans}(\text{rot}(\hat{y}, 180*\text{deg}), \text{vector}(49.6, 104.8, 30.3))$$

All errors other than those described above are assumed to be negligible. Using this information, application of the algorithm given in Section 7.9 gives us a parameterized form for ΔT_{ht}:

$$\Delta T_{ht} = \text{transl}(\Delta p_{ht}) * \Delta R_{ht}$$

where

$$\Delta p_{ht} \tilde{=} \gamma * \text{vector}(3.20, 3.85, 0) + \phi_x * \text{vector}(0, 28.6, 0) + \phi_y * \text{vector}(-28.6, 0, 0) + \phi_z * \text{vector}(0, 0, 0) + \alpha * \text{vector}(0, 3.18, 0) + \beta * \text{vector}(-3.18, 0, 0) + \lambda\hat{x} - \mu\hat{y} + \delta_x\hat{x} + \delta_y\hat{y} + \delta_z\hat{z}$$

$$\Delta R_{ht} \tilde{=} I + \gamma M_z + \phi_x M_x + \phi_y M_y + \phi_z M_z + \alpha M_x + \beta M_y$$

Subject to constraints:

```
[ 1.00 , .000 , .000 ] . V1 ≤ .762
[ 1.00 , .000 , .000 ] . V1 ≥-.762
[ .000 , 1.00 , .000 ] . V1 ≤ .508
[ .000 , 1.00 , .000 ] . V1 ≥-.508
[ .000 , .000 , 1.00 ] . V1 ≤ .873e-1
[ .000 , .000 , 1.00 ] . V1 ≥-.873e-1
[ 1.00 , .000 , .000 , .000 , .000 , .000 ] . V2 ≤ .127
[ 1.00 , .000 , .000 , .000 , .000 , .000 ] . V2 ≥-.127
[ .000 , 1.00 , .000 , .000 , .000 , .000 ] . V2 ≤ .127
[ .000 , 1.00 , .000 , .000 , .000 , .000 ] . V2 ≥-.127
[ .000 , .000 , 1.00 , .000 , .000 , .000 ] . V2 ≤ .127
[ .000 , .000 , 1.00 , .000 , .000 , .000 ] . V2 ≥-.127
[ .000 , .000 , .000 , 1.00 , .000 , .000 ] . V2 ≤ .436e-2
[ .000 , .000 , .000 , 1.00 , .000 , .000 ] . V2 ≥-.436e-2
[ .000 , .000 , .000 , .000 , 1.00 , .000 ] . V2 ≤ .436e-2
[ .000 , .000 , .000 , .000 , 1.00 , .000 ] . V2 ≥-.436e-2
[ .000 , .000 , .000 , .000 , .000 , 1.00 ] . V2 ≤ .436e-2
[ .000 , .000 , .000 , .000 , .000 , 1.00 ] . V2 ≥-.436e-2
[ 1.00 , .000 ] . V3 ≤ .873e-1
[ 1.00 , .000 ] . V3 ≥-.873e-1
[ .000 , 1.00 ] . V3 ≤ .873e-1
[ .000 , 1.00 ] . 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 (Δx and Δy) and along the axis of the hole (Δz). These quantities are given by the objective functions:

$$\Delta x = [\ 3.20,\ .000,\ -28.6,\ .000,\ .000,\ -3.18,\ 1.00,\ .000,\ 1.00,\ .000,\ .000\ ]\ .\ V$$
$$\Delta y = [\ 3.85,\ 28.6,\ .000,\ .000,\ 3.18,\ .000,\ .000,\ -1.00,\ .000,\ 1.00,\ .000\ ]\ .\ V$$
$$\Delta z = [\ .000,\ .000,\ .000,\ .000,\ .000,\ .000,\ .000,\ .000,\ .000,\ .000,\ 1.00\ ]\ .\ V$$

where

$$V = [\ \gamma,\ \phi_x,\ \phi_y,\ \phi_z,\ \alpha,\ \beta,\ \lambda,\ \mu,\ \delta_x,\ \delta_y,\ \delta_z\ ]$$

Solving these linear programming problems, the system gets

$-1.57 \le \Delta x \le 1.57$	(1.57 cm $\tilde{=}$ 0.62 inches)
$-1.37 \le \Delta y \le 1.37$	(1.37 cm $\tilde{=}$ 0.54 inches)
$-.127 \le \Delta z \le .127$	(.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:

$$\Delta\theta \tilde{=} \max\ (\ |\Delta\theta_x|,\ |\Delta\theta_y|)$$

where

$$\Delta\theta_x = [\ .000,\ -1.00,\ .000,\ .000,\ -1.00,\ .000\ ]\ .\ V$$
$$\Delta\theta_y = [\ .000,\ .000,\ -1.00,\ .000,\ .000,\ -1.00\ ]\ .\ V$$

where

$$V = [\gamma,\ \phi_x,\ \phi_y,\ \phi_z,\ \alpha,\ \beta]$$

Solving gives us:

$-.0916 \le \Delta\theta_x \le .0916$	(0.0916 radians $\tilde{=}$ 0.525 degrees)
$-.0916 \le \Delta\theta_y \le .0916$	

Appendix F.

The POINTY System

One of the more bothersome aspects of AL programming is the difficulty of writing statements like

```
affix in_hole_position to hole rigidly
      at trans(nilrotn,vector(0,0,-4.25*cm));
affix hole_approach to hole at trans(nilrotn,vector(0,0,2.54*cm));
affix hole to box at trans(nilrotn,vector(3.85*cm,3.2*cm,4.9*cm));
box ← frame(nilrotn,vector(45.2*cm,102*cm,0));
```

that describe accurately the relevant "constant" values in the program. We have implemented an interactive system, POINTY, which simplifies this process by allowing the user to use the manipulator as a measuring tool to define positions. This system is described fully elsewhere [48,19]. The summary description given in this appendix has been included because of the references made to POINTY in Chapter 3, and because the system exhibits, at very simple level, the use of an interactive system to write parts of a manipulator program, as is discussed in Chapter 9.

System Architecture

POINTY contains three principal "working" modules:

1. The *affixment editor* contains facilities for creating and modifying the frame affixment hierarchy. The principal data structures associated with these routines are the tree used to represent the hierarchy and a set of stack-structured context pointers (called "cursors") are used to control editing functions.

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

3. The *manipulator interface* routines contain facilities for moving the manipulator under either computer or user control and for retrieving the current location of the manipulator for use by the rest of the system. For the latter purpose, the node arm in the affixment hierarchy always contains the current location of the manipulator.

In addition, there are three "service" modules:

1. The *command interpreter* accepts commands typed in from the terminal and executes them. In the current implementation, this facility is provided by BAIL[86], which is a source-level debugger for SAIL. In addition, there is

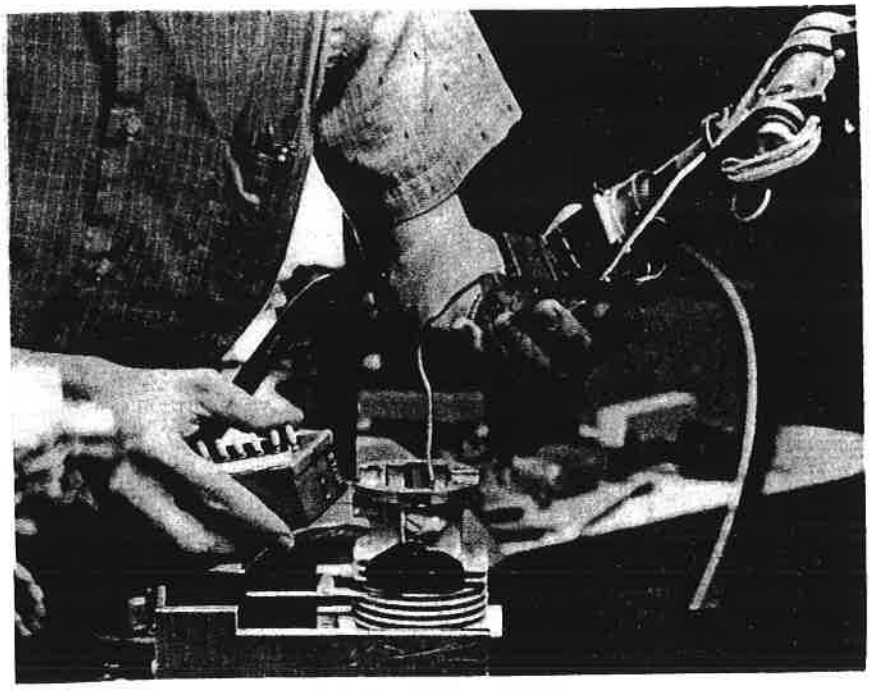


Figure F.1. Pointing at a Feature

a simple macro facility, which allows users to string together commonly occuring command sequences.

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

3. The *input-output facility* includes routines for saving and restoring affixment heirarchies in files, for saving and restoring macro definitions, and for translating affixment trees into a text file of AL declarations, affixments, and assignment statements.

Display Scene

The display facilities of POINTY must provide the user with an up-to-date picture of the data structures. To do this, the display screen is broken up into regions, as shown in Figure F.2.

Here, the current state of the affixment editor is displayed in the large box in the upper left. Parent-child relations are indicated by paragraphing, the kind of affixment by the use of a single character flag ("*" means rigid, "+" means nonrigid, and "-" means unaffixed), and the current values of the editing cursors are indicated by placing the cursor name ("N:", "D:", etc.) on the appropriate line. The relative location of each node in the affixment tree to its parent node is displayed using Euler's angles for rotations. Thus, the location of the arm in Figure F.2 is given by

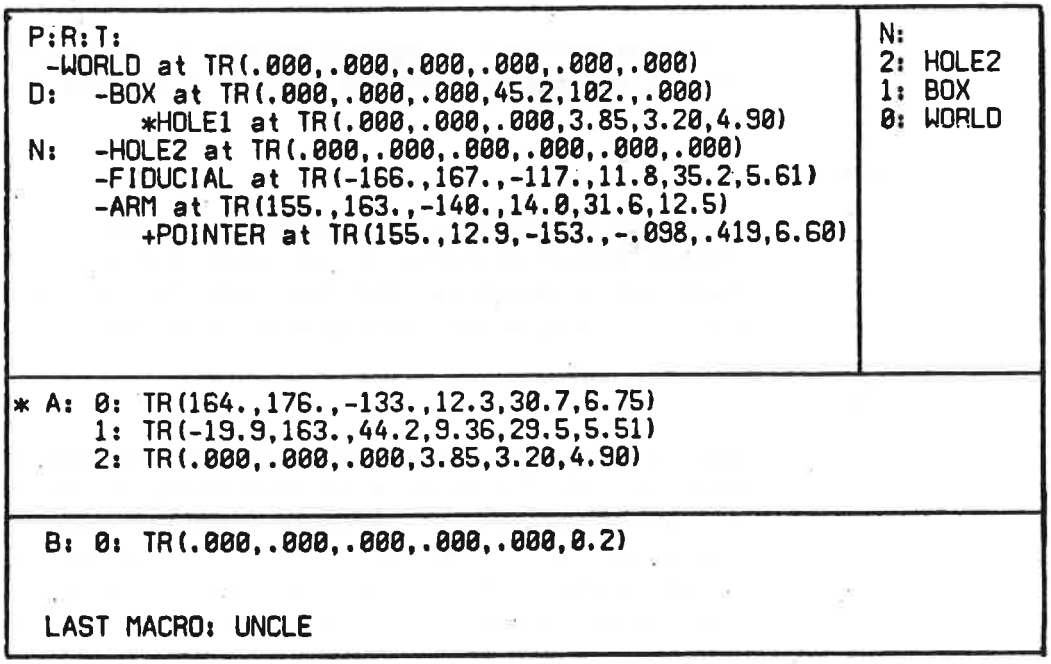


Figure F.2. POINTY Display Scene

trans(rot(zhat,-*166*deg)*rot(yhat,*167*deg)*rot(zhat,-*117*deg),vector(*14.0*,*31.6*,*12.5*))

The column to the right of the affixment tree gives the most recent values of the most recently referenced editing cursor. These cursors are "stack structured"; whenever a new value is assigned to a cursor, the old value is saved away (to a depth of four) where it can be recalled if need be. This facility provides a nice way for a user to recover from errors, as well as to suspend temporarily one editing sequence and go do something else.

The boxes below the affixment tree show the top few elements of the arithmetic stacks ("A:" and "B:"). Both stacks are "general purpose", and either may be used to perform any arithmetic or manipulator control operation. In the absence of an explicit request by the user, operations requiring a stack use whichever one was used last. The current default is indicated by an asterisk.

Example

The following sequence illustrates a typical use of the system to define a feature location. In this dialog, the system typeout is distinguished from user input by underlining.

1: MK_NODE("HOLE2");

This command creates a new node, HOLE2 and sets the "current node" cursor, N:, to point to it. At this point, the situation is as shown in Figure F.2.

Now, we wish to use the manipulator to determine the

position of the hole with respect to the box. To do this, we position the tip of the pointer at the center of the hole and say

```
1: POINTIT;
```

This causes the system to read the arm position, compute the absolute location of the tip of the pointer, and push the result onto A:, the current arithmetic stack. Now we store this location away as the absolute position of the hole.

```
1: ABSSET;
1: RIGID;
```

The first command causes the absolute position of the current node (here, HOLE2) to be set to the value in the top of the current arithmetic stack.[8] The second command causes the tree structure to be modified so that the current node is affixed "rigidly" to the node pointed to by D:, the current "dad" cursor. When this affixment is made, the location attribute of HOLE2 is modified to show its relative location to BOX. A typical value would be something like:

```
TR(160.,15.8,-155.,-3.85,3.20,4.90);
```

Here, the rotation angles are incorrect because the pointer's rotation was not well defined when we performed the pointing operation. To correct this deficiency, the following code sequence may be used:

```
1: APUSH(RELLOC);
1: TEDIT;
1: APUSH(TR(160.,15.8,-155.,-3.85,3.20,4.90));
```

The first command causes the relative location of the current node to be pushed onto the A: stack. The second command removes the top element of the A: stack and assembles *command text* to push it back. This text is then loaded into the terminal line editor, where it can be modified using standard facilities provided by the timesharing system [51]. By changing the first three numbers to 0, and activating with a carriage return, we get:

```
1: APUSH(TR(0,0,0,-3.85,3.20,4.90));
```

which puts the corrected value back on the stack. We then put it back away into the current node.

```
1: RELSET;
```

---

[8] In practice, macro operations are used to reduce the amount of typing required. For instance, The first three commands in this example, can be replaced by a macro, PNTNO, which expands into MK_NODE(PROMPT("NAME="));POINTIT;ABSSET;.

Now, to generate the AL declarations corresponding to the box model, we execute the command sequence:

```
1: CPOP("N:");
1: AL_WRITE;
       OUTPUT FILE = BOX.AL
```

The first command pops the N: stack, so the current node is now BOX. The second command writes out the AL declarations for the current node. If an output file were already open, then the system would use it without prompting for a new one. Thus, several structures can be written out on one file. The output file is closed when the program is exited or else by typing the command:

```
1: AL_CLOSE;
```

The declarations generated by the program are shown below:

```
FRAME BOX;
BOX ← FRAME(NILROTN,VECTOR( 45.2, 102., .000));

FRAME HOLE2;
AFFIX HOLE2 TO BOX AT TRANS(NILROTN,VECTOR(-3.85, 3.20, 4.90)) RIGIDLY;

FRAME HOLE1;
AFFIX HOLE1 TO BOX AT TRANS(NILROTN,VECTOR( 3.85, 3.20, 4.90)) RIGIDLY;
```