CHAPTER 12

Robot programming languages and systems

12.1 INTRODUCTION
12.2 THE THREE LEVELS OF ROBOT PROGRAMMING
12.3 A SAMPLE APPLICATION
12.4 REQUIREMENTS OF A ROBOT PROGRAMMING LANGUAGE
12.5 PROBLEMS PECULIAR TO ROBOT PROGRAMMING LANGUAGES

12.1 INTRODUCTION

In this chapter, we begin to consider the interface between the human user and an industrial robot. It is by means of this interface that a user takes advantage of all the underlying mechanics and control algorithms we have studied in previous chapters.

The sophistication of the user interface is becoming extremely important as manipulators and other programmable automation are applied to more and more demanding industrial applications. It turns out that the nature of the user interface is a very important concern. In fact, most of the challenge of the design and use of industrial robots focuses on this aspect of the problem.

Robot manipulators differentiate themselves from fixed automation by being "flexible," which means programmable. Not only are the movements of manipulators programmable, but, through the use of sensors and communications with other factory automation, manipulators can *adapt* to variations as the task proceeds.

In considering the programming of manipulators, it is important to remember that they are typically only a minor part of an automated process. The term **workcell** is used to describe a local collection of equipment, which may include one or more manipulators, conveyor systems, parts feeders, and fixtures. At the next higher level, workcells might be interconnected in factorywide networks so that a central control computer can control the overall factory flow. Hence, the programming of manipulators is often considered within the broader problem of programming a variety of interconnected machines in an automated factory workcell.

Unlike that in the previous 11 chapters, the material in this chapter (and the next chapter) is of a nature that constantly changes. It is therefore difficult to present this material in a detailed way. Rather, we attempt to point out the underlying fundamental concepts, and we leave it to the reader to seek out the latest examples, as industrial technology continues to move forward.

12.2 THE THREE LEVELS OF ROBOT PROGRAMMING

There have been many styles of user interface developed for programming robots. Before the rapid proliferation of microcomputers in industry, robot controllers resembled the simple sequencers often used to control fixed automation. Modern approaches focus on computer programming, and issues in programming robots include all the issues faced in general computer programming—and more.

Teach by showing

Early robots were all programmed by a method that we will call **teach by showing**, which involved moving the robot to a desired goal point and recording its position in a memory that the sequencer would read during playback. During the teach phase, the user would guide the robot either by hand or through interaction with a **teach pendant**. Teach pendants are handheld button boxes that allow control of each manipulator joint or of each Cartesian degree of freedom. Some such controllers allow testing and branching, so that simple programs involving logic can be entered. Some teach pendants have alphanumeric displays and are approaching hand-held terminals in complexity. Figure 12.1 shows an operator using a teach pendant to program a large industrial robot.

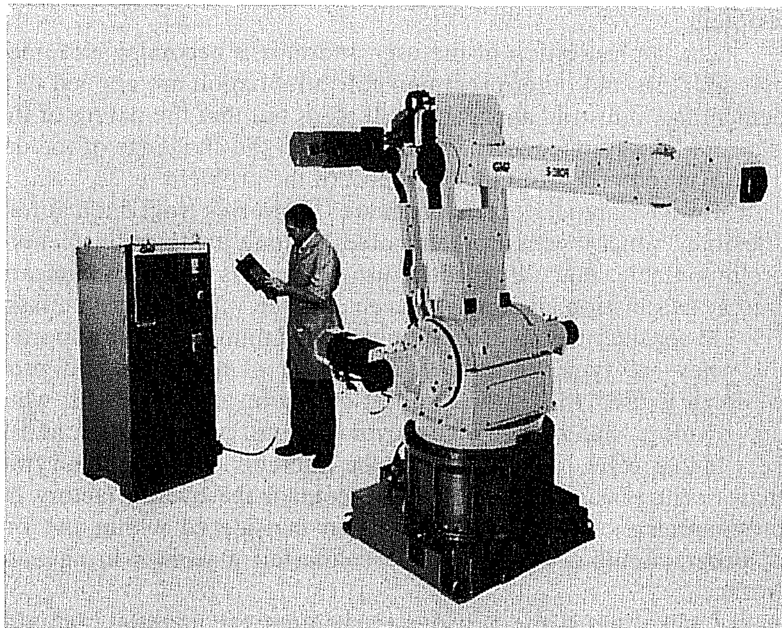

FIGURE 12.1: The GMF S380 is often used in automobile-body spot-welding applications. Here an operator uses a teach-pendant interface to program the manipulator. Photo courtesy of GMFanuc Corp.

Explicit robot programming languages

Ever since the arrival of inexpensive and powerful computers, the trend has been increasingly toward programming robots via programs written in computer programming languages. Usually, these computer programming languages have special features that apply to the problems of programming manipulators and so are called **robot programming languages** (RPLs). Most of the systems that come equipped with a robot programming language have nonetheless retained a teach-pendant-style interface also.

Robot programming languages have likewise taken on many forms. We will split them into three categories:

1. **Specialized manipulation languages**. These robot programming languages have been built by developing a completely new language that, although addressing robot-specific areas, might well be considered a general computer programming language. An example is the VAL language developed to control the industrial robots by Unimation, Inc [1]. VAL was developed especially as a manipulator control language; as a general computer language, it was quite weak. For example, it did not support floating-point numbers or character strings, and subroutines could not pass arguments. A more recent version, V-II, provided these features [2]. The current incarnation of this language, V+, includes many new features [13]. Another example of a specialized manipulation language is AL, developed at Stanford University [3]. Although the AL language is now a relic of the past, it nonetheless provides good examples of some features still not found in most modern languages (force control, parallelism). Also, because it was built in an academic environment, there are references available to describe it [3]. For these reasons, we continue to make reference to it.
2. **Robot library for an existing computer language**. These robot programming languages have been developed by starting with a popular computer language (e.g., Pascal) and adding a library of robot-specific subroutines. The user then writes a Pascal program making use of frequent calls to the predefined subroutine package for robot-specific needs. An examples is AR-BASIC from American Cimflex [4], essentially a subroutine library for a standard BASIC implementation. JARS, developed by NASA's Jet Propulsion Laboratory, is an example of such a robot programming language based on Pascal [5].
3. **Robot library for a new general-purpose language**. These robot programming languages have been developed by first creating a new general-purpose language as a programming base and then supplying a library of predefined robot-specific subroutines. Examples of such robot programming languages are RAPID developed by ABB Robotics [6], AML developed by IBM [7], and KAREL developed by GMF Robotics [8].

Studies of actual application programs for robotic workcells have shown that a large percentage of the language statements are not robot-specific [7]. Instead, a great deal of robot programming has to do with initialization, logic testing and branching, communication, and so on. For this reason, a trend might develop to

move away from developing special languages for robot programming and move toward developing extensions to general languages, as in categories 2 and 3 above.

Task-level programming languages

The third level of robot programming methodology is embodied in **task-level programming languages**. These languages allow the user to command desired subgoals of the task directly, rather than specify the details of every action the robot is to take. In such a system, the user is able to include instructions in the application program at a significantly higher level than in an explicit robot programming language. A task-level robot programming system must have the ability to perform many planning tasks automatically. For example, if an instruction to "grasp the bolt" is issued, the system must plan a path of the manipulator that avoids collision with any surrounding obstacles, must automatically choose a good grasp location on the bolt, and must grasp it. In contrast, in an explicit robot programming language, all these choices must be made by the programmer.

The border between explicit robot programming languages and task-level programming languages is quite distinct. Incremental advances are being made to explicit robot programming languages to help to ease programming, but these enhancements cannot be counted as components of a task-level programming system. True task-level programming of manipulators does not yet exist, but it has been an active topic of research [9, 10] and continues as a research topic today.

12.3 A SAMPLE APPLICATION

Figure 12.2 shows an automated workcell that completes a small subassembly in a hypothetical manufacturing process. The workcell consists of a conveyor under computer control that delivers a workpiece; a camera connected to a vision system, used to locate the workpiece on the conveyor; an industrial robot (a PUMA 560 is pictured) equipped with a force-sensing wrist; a small feeder located on the work surface that supplies another part to the manipulator; a computer-controlled press that can be loaded and unloaded by the robot; and a pallet upon which the robot places finished assemblies.

The entire process is controlled by the manipulator's controller in a sequence, as follows:

1. The conveyor is signaled to start; it is stopped when the vision system reports that a bracket has been detected on the conveyor.
2. The vision system judges the bracket's position and orientation on the conveyor and inspects the bracket for defects, such as the wrong number of holes.
3. Using the output of the vision system, the manipulator grasps the bracket with a specified force. The distance between the fingertips is checked to ensure that the bracket has been properly grasped. If it has not, the robot moves out of the way and the vision task is repeated.
4. The bracket is placed in the fixture on the work surface. At this point, the conveyor can be signaled to start again for the next bracket—that is, steps 1 and 2 can begin in parallel with the following steps.

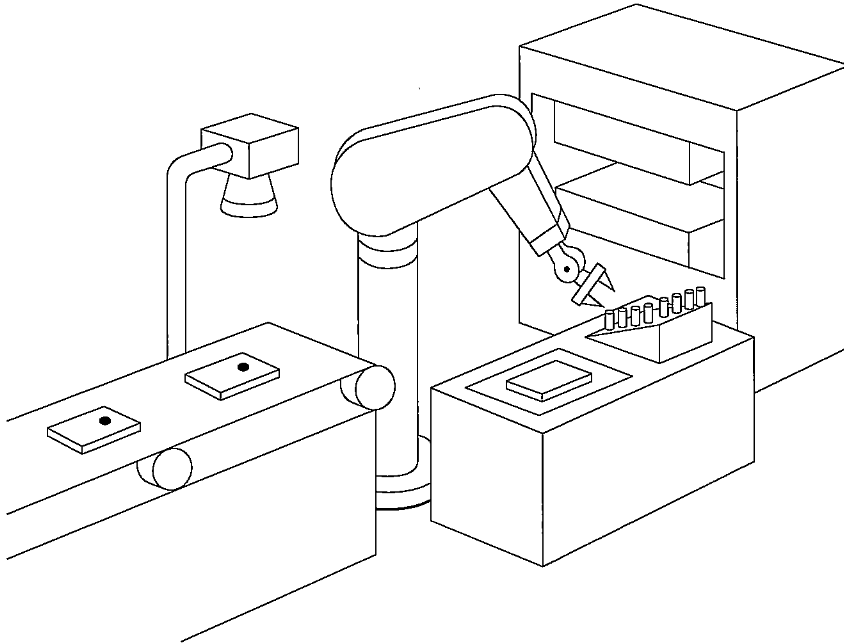

FIGURE 12.2: An automated workcell containing an industrial robot.

5. A pin is picked from the feeder and inserted partway into a tapered hole in the bracket. Force control is used to perform this insertion and to perform simple checks on its completion. (If the pin feeder is empty, an operator is notified and the manipulator waits until commanded to resume by the operator.)
6. The bracket–pin assembly is grasped by the robot and placed in the press.
7. The press is commanded to actuate, and it presses the pin the rest of the way into the bracket. The press signals that it has completed, and the bracket is placed back into the fixture for a final inspection.
8. By force sensing, the assembly is checked for proper insertion of the pin. The manipulator senses the reaction force when it presses sideways on the pin and can do several checks to discover how far the pin protrudes from the bracket.
9. If the assembly is judged to be good, the robot places the finished part into the next available pallet location. If the pallet is full, the operator is signaled. If the assembly is bad, it is dropped into the trash bin.
10. Once Step 2 (started earlier in parallel) is complete, go to Step 3.

This is an example of a task that is possible for today's industrial robots. It should be clear that the definition of such a process through "teach by showing" techniques is probably not feasible. For example, in dealing with pallets, it is laborious to have to teach all the pallet compartment locations; it is much preferable to teach only the corner location and then compute the others from knowledge of the dimensions of the pallet. Further, specifying interprocess signaling and setting up parallelism by using a typical teach pendant or a menu-style interface is usually

not possible at all. This kind of application necessitates a robot programming-language approach to process description. (See Exercise 12.5.) On the other hand, this application is too complex for any existing task-level languages to deal with directly. It is typical of the great many applications that must be addressed with an explicit robot programming approach. We will keep this sample application in mind as we discuss features of robot programming languages.

12.4 REQUIREMENTS OF A ROBOT PROGRAMMING LANGUAGE

World modeling

Manipulation programs must, by definition, involve moving objects in three-dimensional space, so it is clear that any robot programming language needs a means of describing such actions. The most common element of robot programming languages is the existence of special **geometric types**. For example, *types* are introduced to represent joint-angle sets, Cartesian positions, orientations, and frames. Predefined operators that can manipulate these types often are available. The "standard frames" introduced in Chapter 3 might serve as a possible model of the world: All motions are described as tool frame relative to station frame, with goal frames being constructed from arbitrary expressions involving geometric types.

Given a robot programming environment that supports geometric types, the robot and other machines, parts, and fixtures can be modeled by defining named variables associated with each object of interest. Figure 12.3 shows part of our example workcell with frames attached in task-relevant locations. Each of these frames would be represented with a variable of type "frame" in the robot program.

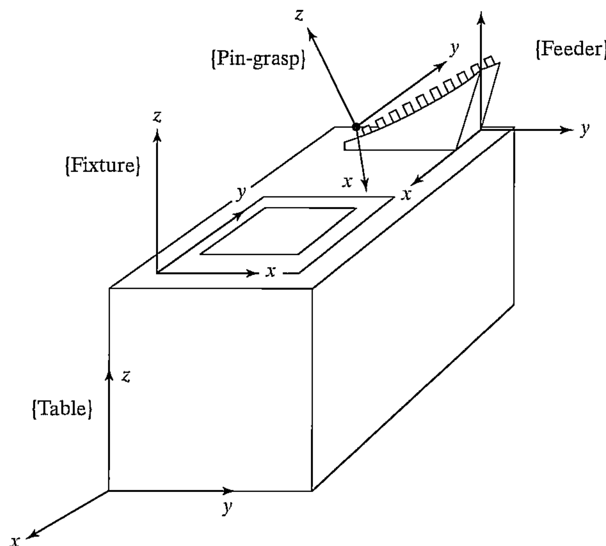

FIGURE 12.3: Often, a workcell is modeled simply, as a set of frames attached to relevant objects.

In many robot programming languages, this ability to define named variables of various geometric types and refer to them in the program forms the basis of the world model. Note that the physical shapes of the objects are not part of such a world model, and neither are surfaces, volumes, masses, or other properties. The extent to which objects in the world are modeled is one of the basic design decisions made when designing a robot programming system. Most present-day systems support only the style just described.

Some world-modeling systems allow the notion of **affixments** between named objects [3]—that is, the system can be notified that two or more named objects have become "affixed"; from then on, if one object is explicitly moved with a language statement, any objects affixed to it are moved with it. Thus, in our application, once the pin has been inserted into the hole in the bracket, the system would be notified (via a language statement) that these two objects have become affixed. Subsequent motions of the bracket (that is, changes to the value of the frame variable "bracket") would cause the value stored for variable "pin" to be updated along with it.

Ideally, a world-modeling system would include much more information about the objects with which the manipulator has to deal and about the manipulator itself. For example, consider a system in which objects are described by CAD-style models that represent the spatial shape of an object by giving definitions of its edges, surfaces, or volume. With such data available to the system, it begins to become possible to implement many of the features of a task-level programming system. These possibilities are discussed further in Chapter 13.

Motion specification

A very basic function of a robot programming language is to allow the description of desired motions of the robot. Through the use of motion statements in the language, the user interfaces to path planners and generators of the style described in Chapter 7. Motion statements allow the user to specify via points, the goal point, and whether to use joint-interpolated motion or Cartesian straight-line motion. Additionally, the user might have control over the speed or duration of a motion.

To illustrate various syntaxes for motion primitives, we will consider the following example manipulator motions: (1) move to position "goal1," then (2) move in a straight line to position "goal2," then (3) move without stopping through "via1" and come to rest at "goal3." Assuming all of these path points had already been taught or described textually, this program segment would be written as follows:

In VAL II,

```
move goal1
moves goal2
move via1
move goal3
```

In AL (here controlling the manipulator "garm"),

```
move garm to goal1;
move garm to goal2 linearly;
move garm to goal3 via via1;
```

Most languages have similar syntax for simple motion statements like these. Differences in the basic motion primitives from one robot programming language to another become more apparent if we consider features such as the following:

1. the ability to do math on such structured types as frames, vectors, and rotation matrices;
2. the ability to describe geometric entities like frames in several different convenient representations—along with the ability to convert between representations;
3. the ability to give constraints on the duration or velocity of a particular move—for example, many systems allow the user to set the speed to a fraction of maximum, but fewer allow the user to specify a desired duration or a desired maximum joint velocity directly;
4. the ability to specify goals relative to various frames, including frames defined by the user and frames in motion (on a conveyor, for example).

Flow of execution

As in more conventional computer programming languages, a robot programming system allows the user to specify the flow of execution—that is, concepts such as testing and branching, looping, calls to subroutines, and even interrupts are generally found in robot programming languages.

More so than in many computer applications, parallel processing is generally important in automated workcell applications. First of all, very often two or more robots are used in a single workcell and work simultaneously to reduce the cycle time of the process. Even in single-robot applications, such as the one shown in Fig. 12.2, other workcell equipment must be controlled by the robot controller in a parallel fashion. Hence, *signal* and *wait* primitives are often found in robot programming languages, and occasionally more sophisticated parallel-execution constructs are provided [3].

Another frequent occurrence is the need to monitor various processes with some kind of sensor. Then, either by interrupt or through polling, the robot system must be able to respond to certain events detected by the sensors. The ability to specify such **event monitors** easily is afforded by some robot programming languages [2, 3].

Programming environment

As with any computer languages, a good programming environment fosters programmer productivity. Manipulator programming is difficult and tends to be very interactive, with a lot of trial and error. If the user were forced to continually repeat the "edit-compile-run" cycle of compiled languages, productivity would be low. Therefore, most robot programming languages are now *interpreted*, so that individual language statements can be run one at a time during program development and debugging. Many of the language statements cause motion of a physical device, so the tiny amount of time required to interpret the language statements is insignificant. Typical programming support, such as text editors, debuggers, and a file system, are also required.

Sensor integration

An extremely important part of robot programming has to do with interaction with sensors. The system should have, at a minimum, the capability to query touch and force sensors and to use the response in if-then-else constructs. The ability to specify event monitors to watch for transitions on such sensors in a *background* mode is also very useful.

Integration with a vision system allows the vision system to send the manipulator system the coordinates of an object of interest. For example, in our sample application, a vision system locates the brackets on the conveyor belt and returns to the manipulator controller their position and orientation relative to the camera. The camera's frame is known relative to the station frame, so a desired goal frame for the manipulator can be computed from this information.

Some sensors could be part of other equipment in the workcell—for example, some robot controllers can use input from a sensor attached to a conveyor belt so that the manipulator can track the belt's motion and acquire objects from the belt as it moves [2].

The interface to force-control capabilities, as discussed in Chapter 9, comes through special language statements that allow the user to specify force strategies [3]. Such force-control strategies are by necessity an integrated part of the manipulator control system—the robot programming language simply serves as an interface to those capabilities. Programming robots that make use of active force control might require other special features, such as the ability to display force data collected during a constrained motion [3].

In systems that support active force control, the description of the desired force application could become part of the motion specification. The AL language describes active force control in the motion primitives by specifying six components of stiffness (three translational and three rotational) and a bias force. In this way, the manipulator's apparent stiffness is programmable. To apply a force, usually the stiffness is set to zero in that direction and a bias force is specified—for example,

```
move garm to goal
with stiffness=(80, 80, 0, 100, 100, 100)
with force=20*ounces along zhat;
```

12.5 PROBLEMS PECULIAR TO ROBOT PROGRAMMING LANGUAGES

Advances in recent years have helped, but programming robots is still difficult. Robot programming shares all the problems of conventional computer programming, plus some additional difficulties caused by effects of the physical world [12].

Internal world model versus external reality

A central feature of a robot programming system is the world model that is maintained internally in the computer. Even when this model is quite simple, there are ample difficulties in assuring that it matches the physical reality that it attempts to model. Discrepancies between internal model and external reality result in poor or failed grasping of objects, collisions, and a host of more subtle problems.

This correspondence between internal model and the external world must be established for the program's initial state and must be maintained throughout its execution. During initial programming or debugging, it is generally up to the user to suffer the burden of ensuring that the state represented in the program corresponds to the physical state of the workcell. Unlike more conventional programming, where only internal variables need to be saved and restored to reestablish a former situation, in robot programming, physical objects must usually be repositioned.

Besides the uncertainty inherent in each object's position, the manipulator itself is limited to a certain degree of accuracy. Very often, steps in an assembly will require the manipulator to make motions requiring greater precision than it is capable of. A common example of this is inserting a pin into a hole where the clearance is an order of magnitude less than the positional accuracy of the manipulator. To further complicate matters, the manipulator's accuracy usually varies over its workspace.

In dealing with those objects whose locations are not known exactly, it is essential to somehow refine the positional information. This can sometimes be done with sensors (e.g., vision, touch) or by using appropriate force strategies for constrained motions.

During debugging of manipulator programs, it is very useful to be able to modify the program and then back up and try a procedure again. Backing up entails restoring the manipulator and objects being manipulated to a former state. However, in working with physical objects, it is not always easy, or even possible, to undo an action. Some examples are the operations of painting, riveting, drilling, or welding, which cause a physical modification of the objects being manipulated. It might therefore be necessary for the user to get a new copy of the object to replace the old, modified one. Further, it is likely that some of the operations just prior to the one being retried will also need to be repeated to establish the proper state required before the desired operation can be successfully retried.

Context sensitivity

Bottom-up programming is a standard approach to writing a large computer program in which one develops small, low-level pieces of a program and then puts them together into larger pieces, eventually attaining a completed program. For this method to work, it is essential that the small pieces be relatively insensitive to the language statements that precede them and that there be no assumptions concerning the context in which these program pieces execute. For manipulator programming, this is often not the case; code that worked reliably when tested in isolation frequently fails when placed in the context of the larger program. These problems generally arise from dependencies on manipulator configuration and speed of motions.

Manipulator programs can be highly sensitive to initial conditions—for example, the initial manipulator position. In motion trajectories, the starting position will influence the trajectory that will be used for the motion. The initial manipulator position might also influence the velocity with which the arm will be moving during some critical part of the motion. For example, these statements are true for manipulators that follow the cubic-spline joint-space paths studied in Chapter 7. These effects can sometimes be dealt with by proper programming care, but often such

problems do not arise until after the initial language statements have been debugged in isolation and are then joined with statements preceding them.

Because of insufficient manipulator accuracy, a program segment written to perform an operation at one location is likely to need to be tuned (i.e., positions retaught and the like) to make it work at a different location. Changes in location within the workcell result in changes in the manipulator's configuration in reaching goal locations. Such attempts at relocating manipulator motions within the workcell test the accuracy of the manipulator kinematics and servo system, and problems frequently arise. Such relocation could cause a change in the manipulator's kinematic configuration—for example, from left shoulder to right shoulder, or from elbow up to elbow down. Moreover, these changes in configuration could cause large arm motions during what had previously been a short, simple motion.

The nature of the spatial shape of trajectories is likely to change as paths are located in different portions of the manipulator's workspace. This is particularly true of joint-space trajectory methods, but use of Cartesian-path schemes can also lead to problems when singularities are nearby.

When testing a manipulator motion for the first time, it often is wise to have the manipulator move slowly. This allows the user a chance to stop the motion if it appears to be about to cause a collision. It also allows the user to inspect the motion closely. After the motion has undergone some initial debugging at a slower speed it is then desirable to increase motion speeds. Doing so might cause some aspects of the motion to change. Limitations in most manipulator control systems cause greater servo errors, which are to be expected if the quicker trajectory is followed. Also, in force-control situations involving contact with the environment, speed changes can completely change the force strategies required for success.

The manipulator's configuration also affects the delicacy and accuracy of the forces that can be applied with it. This is a function of how well conditioned the Jacobian of the manipulator is at a certain configuration, something generally difficult to consider when developing robot programs.

Error recovery

Another direct consequence of working with the physical world is that objects might not be exactly where they should be and, hence, motions that deal with them could fail. Part of manipulator programming involves attempting to take this into account and making assembly operations as robust as possible, but, even so, errors are likely, and an important aspect of manipulator programming is how to recover from these errors.

Almost any motion statement in the user's program can fail, sometimes for a variety of reasons. Some of the more common causes are objects shifting or dropping out of the hand, an object missing from where it should be, jamming during an insertion, and not being able to locate a hole.

The first problem that arises for error recovery is identifying that an error has indeed occurred. Because robots generally have quite limited sensing and reasoning capabilities, *error detection* is often difficult. In order to detect an error, a robot program must contain some type of explicit test. This test might involve checking the manipulator's position to see that it lies in the proper range; for example, when doing an insertion, lack of change in position might indicate jamming, or too much

change might indicate that the hole was missed entirely or the object has slipped out of the hand. If the manipulator system has some type of visual capabilities, then it might take a picture and check for the presence or absence of an object and, if the object is present, report its location. Other checks might involve force, such as weighing the load being carried to check that the object is still there and has not been dropped, or checking that a contact force remains within certain bounds during a motion.

Every motion statement in the program might fail, so these explicit checks can be quite cumbersome and can take up more space than the rest of the program. Attempting to deal with all possible errors is extremely difficult; usually, just the few statements that seem most likely to fail are checked. The process of predicting which portions of a robot application program are likely to fail is one that requires a certain amount of interaction and partial testing with the robot during the program-development stage.

Once an error has been detected, an attempt can be made to recover from it. This can be done totally by the manipulator under program control, or it might involve manual intervention by the user, or some combination of the two. In any event, the recovery attempt could in turn result in new errors. It is easy to see how code to recover from errors can become the major part of the manipulator program.

The use of parallelism in manipulator programs can further complicate recovery from errors. When several processes are running concurrently and one causes an error to occur, it could affect other processes. In many cases, it will be possible to back up the offending process, while allowing the others to continue. At other times, it will be necessary to reset several or all of the running processes.

BIBLIOGRAPHY

[1] B. Shimano, "VAL: A Versatile Robot Programming and Control System," Proceedings of COMPSAC 1979, Chicago, November 1979.

[2] B. Shimano, C. Geschke, and C. Spalding, "VAL II: A Robot Programming Language and Control System," SME Robots VIII Conference, Detroit, June 1984.

[3] S. Mujtaba and R. Goldman, "AL Users' Manual," 3rd edition, Stanford Department of Computer Science, Report No. STAN-CS-81-889, December 1981.

[4] A. Gilbert et al., *AR-BASIC: An Advanced and User Friendly Programming System for Robots*, American Robot Corporation, June 1984.

[5] J. Craig, "JARS—JPL Autonomous Robot System: Documentation and Users Guide," JPL Interoffice memo, September 1980.

[6] ABB Robotics, "The RAPID Language," in the *SC4Plus Controller Manual*, ABB Robotics, 2002.

[7] R. Taylor, P. Summers, and J. Meyer, "AML: A Manufacturing Language," *International Journal of Robotics Research*, Vol. 1, No. 3, Fall 1982.

[8] FANUC Robotics, Inc., "KAREL Language Reference," FANUC Robotics North America, Inc, 2002.

[9] R. Taylor, "A Synthesis of Manipulator Control Programs from Task-Level Specifications," Stanford University AI Memo 282, July 1976.

[10] J.C. LaTombe, "Motion Planning with Uncertainty: On the Preimage Backchaining Approach," in *The Robotics Review*, O. Khatib, J. Craig, and T. Lozano-Perez, Editors, MIT Press, Cambridge, MA, 1989.

[11] W. Gruver and B. Soroka, "Programming, High Level Languages," in *The International Encyclopedia of Robotics*, R. Dorf and S. Nof, Editors, Wiley Interscience, New York, 1988.

[12] R. Goldman, *Design of an Interactive Manipulator Programming Environment*, UMI Research Press, Ann Arbor, MI, 1985.

[13] Adept Technology, *V+ Language Reference*, Adept Technology, Livermore, CA, 2002.

EXERCISES

12.1 [15] Write a robot program (in a language of your choice) to pick a block up from location A and place it in location B.

12.2 [20] Describe tying your shoelace in simple English commands that might form the basis of a robot program.

12.3 [32] Design the syntax of a new robot programming language. Include ways to give duration or speeds to motion trajectories, make I/O statements to peripherals, give commands to control the gripper, and produce force-sensing (i.e., guarded move) commands. You can skip force control and parallelism (to be covered in Exercise 12.4).

12.4 [28] Extend the specification of the new robot programming language that you started in Exercise 12.3 by adding force-control syntax and syntax for parallelism.

12.5 [38] Write a program in a commercially available robot programming language to perform the application outlined in Section 12.3. Make any reasonable assumptions concerning I/O connections and other details.

12.6 [28] Using any robot language, write a general routine for unloading an arbitrarily sized pallet. The routine should keep track of indexing through the pallet and signal a human operator when the pallet is empty. Assume the parts are unloaded onto a conveyor belt.

12.7 [35] Using any robot language, write a general routine for unloading an arbitrarily sized source pallet and loading an arbitrarily sized destination pallet. The routine should keep track of indexing through the pallets and signal a human operator when the source pallet is empty and when the destination pallet is full.

12.8 [35] Using any capable robot programming language, write a program that employs force control to fill a cigarette box with 20 cigarettes. Assume that the manipulator has an accuracy of about 0.25 inch, so force control should be used for many operations. The cigarettes are presented on a conveyor belt, and a vision system returns their coordinates.

12.9 [35] Using any capable robot programming language, write a program to assemble the hand-held portion of a standard telephone. The six components (handle, microphone, speaker, two caps, and cord) arrive in a *kit*, that is, a special pallet holding one of each kind of part. Assume there is a fixture into which the handle can be placed that holds it. Make any other reasonable assumptions needed.

12.10 [33] Write a robot program that uses two manipulators. One, called GARM, has a special end-effector designed to hold a wine bottle. The other arm, BARM, will hold a wineglass and is equipped with a force-sensing wrist that can be used to signal GARM to stop pouring when it senses the glass is full.

PROGRAMMING EXERCISE (PART 12)

Create a user interface to the other programs you have developed by writing a few subroutines in Pascal. Once these routines are defined, a "user" could write a Pascal program that contains calls to these routines to perform a 2-D robot application in simulation.

Define primitives that allow the user to set station and tool frames—namely,

```
setstation(SrelB:vec3);

settool(TrelW:vec3);
```

where "SrelB" gives the station frame relative to the base frame of the robot and "TrelW" defines the tool frame relative to the wrist frame of the manipulator. Define the motion primitives

```
moveto(goal:vec3);

moveby(increment:vec3);
```

where "goal" is a specification of the goal frame relative to the station frame and "increment" is a specification of a goal frame relative to the current tool frame. Allow multisegment paths to be described when the user first calls the "pathmode" function, then specifies motions to via points, and finally says "runpath"—for example,

```
pathmode; (* enter path mode *)

moveto(goal1);

moveto(goal2);

runpath; (* execute the path without stopping at goal1 *)
```

Write a simple "application" program, and have your system print the location of the arm every n seconds.