Vincent Hayward*
Laboratoire d'Informatique et de Mécanique
pour les Sciences de l'Ingénieur
LIMSI-CNRS BP 30
91406 Orsay Cedex
France

Richard P. Paul
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, Pennsylvania 19104

Robot Manipulator Control under Unix RCCL: A Robot Control "C" Library

Abstract

In this paper, we present a general purpose manipulator control system. The system is run under the Unix operating system. Manipulator programs are written in the "C" language and make use of primitive functions included in a library. Manipulator control is thus integrated within the language in the same manner as is input-output. The system includes a world modeler and a trajectory generator that are accessed through two sets of primitive functions. The system's structured world modeler is designed for an easy integration of sensors. The first part of the paper reviews the functional organization of the system, going through world modeling, trajectory generation, force control, and synchronization. The second part describes actual robot programming examples.

1. Introduction

For more than a decade, robot manipulator control has been associated with the development of dedicated robot programming languages (Paul 1977; Mujtaba and Goldman 1981; Taylor, Summers, and Meyers 1982). The goal of these languages has been to provide a suitable framework for the expression of robot tasks, since it is believed that programmability of robots is their principal advantage over traditional automation (Nitzan and Rosen 1976; Lozano-Perez 1983). Robots can no longer be considered as separate devices acting on their own and must be integrated within manufacturing systems. As a result, robot languages have become increasingly more powerful in order to handle the interactions among robots, their working environment, and the remainder of the manufacturing control system. Robot programming systems have also evolved to implement advances made in several areas of industrial automation, such as sensing, machine perception, and control.

The majority of current robot programming systems are based on a robot controller designed around a special language (Shimano, Geshke, and Spalding 1983; Taylor 1983). The language on which robot control is based must cope with the complexities of real-time control of the arm and its end effector, world modeling, sensory feedback, and general input-output. The language must also implement, or be able to handle, man-machine and machine-to-machine communications. Based on these needs, robot programming languages have evolved to a point where they resemble extended, high-level computer languages. Research in task-level robot programming has demonstrated the need for intermediate robot control primitives as targets for task-planning systems or off-line programming systems. These primitives are often not provided in the most suitable and flexible manner by traditional robot programming languages.

This work has been partially supported by a grant from the ARA program (Automatique et Robotique Avancée) of CNRS, France. This material is also based on work supported by the National Science Foundation under Grant No. MEA-81119884. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

* Vincent Hayward is now with the Computer Vision and Robotics Laboratory, Department of Electrical Engineering, McGill University, Montréal, Québec, Canada H3A 2A7.

The International Journal of Robotics Research,
Vol. 5, No. 4, Winter 1986,

In this paper, we describe a different approach to robot programming. We wanted to avoid the creation of a specialized language, since advances in computer science have provided widely used and efficient general purpose languages. We chose the "C" language (Kernighan 1978) and integrated robot control in the language in the same manner as general input-output is integrated. The "C" language is well suited for this purpose because of its ability to handle low-level details. The services usually provided by operating systems (e.g., I/O, memory management) are revealed to the "C" programmer through subroutine calls instead of language primitives. In languages like Pascal, these services are reflected in the syntax of the language; therefore, extensions to the language often require revisions of its syntax. Other languages like ADA or Modula could have provided the required flexibility for the project at hand, but none of these languages was in sufficient widespread use at the time of the project inception.

RCCL is a self-standing, subroutine package offering a suitable environment for the programming of robots. Tools are provided for control over the scheduling of motions and the trajectory generation, thus leading to an easy integration of sensors. The robot control primitive functions are written in "C" and included in a library. Robot applications can directly be developed in "C," and no modification to the compiler is required. The RCCL subroutine package is built on top of a low-level robot control package called the *arm interface* (briefly described in Section 2.1). An Appendix describes the additions that have been made to the Unix kernel to provide for real-time control.

An initial implementation of the RCCL system suggests that many benefits can be gained from this approach as far as modularity, system and application development, and portability are concerned. The system has been designed with the following goals in mind:

Portability. The system is written in the portable language "C," available for a number of machines. RCCL was first implemented on a VAX minicomputer under Unix. However, the system has been ported and adapted to other machines than a Vax and can run under other operating systems. (See Kosman [1986], for example.)

Manipulator independence. The arm dependencies, such as the kinematics and the physical capabilities of the arm, have been isolated and can be modified easily. The present implementation of the library can be generated for two different arms by means of macro compilation. Robot programs are as independent of any particular manipulator as Cartesian programming can allow. Dependencies, such as the working envelope, sweep, reach, and kinematic configurations, cannot be avoided at manipulator-level programming.

World modeling. The system fully implements the structured position description introduced in the PAL language (Takase, Paul, and Berg 1979) with some extensions.

Cartesian programming. Locations are described in Cartesian space. An arbitrary coordinate frame can be programmed to move along straight-line trajectories or along arbitrary trajectories described with respect to the basic motion scheme.

Sensor integration. This point is one of the main issues of the system design. We make use of the idea that sensor integration is handled naturally if the world model can be synchronously or asynchronously modified. In any case, the user has full control over synchronization, whether the program flow needs to be synchronized with the arm motions, or the arm motions with the program flow.

Force control. In a matter of months, a simple technique of compliant motion control has been developed and integrated within the system.

2. Overview

The system is built around a trajectory generator and a world modeler. The trajectory generator is an interrupt-driven process that uses position specifications described in the world model to compute joint position or torque set points at a fixed sample rate. From the user's point of view, the trajectory generator acts like a background process. The robot program, or user's process, is similar to a Unix process executed under time-sharing that asynchronously issues motion requests to the trajectory generator. The motion requests

are entered into a queue and processed by the trajectory generator on a first-in-first-out basis.

The world model consists of a set of homogeneous transformations equations and is implemented in terms of dynamic data structures. These equations, or closed kinematics chains, structurally describe the relative locations of objects and features involved in the task description.

A motion request is a request to the system to modify the world model so that a position equation becomes verified. Several kinds of transformations have been defined in order to deal with sensor integration, time-dependent world modeling, and communications. Whenever necessary, the user's process is synchronized with the actual arm motions or other external events. Synchronous processing is performed by user written background functions specified as part of a motion request or attached to a functionally described transformation. Any trajectory segment can be interrupted at a given instant. This provides control over scheduling of motions according to arbitrary events or conditions. Synchronous input is provided by global variables that are updated at sample rate, with the state of the arm itself or with the state of the trajectory generator. Finally, the world model can be implicitly updated upon completion of a conditional motion. The information that is obtained on condition detection is thus directly included in the world model.

To summarize, we have sought to separate the world modeling problem, implemented in terms of dynamic data structures, from the trajectory generation and the real-time control of the arm. The control of the arm and the trajectory generation is viewed as a high priority background process. The next sections explain in more detail the functional organization of the system.

2.1. Low-Level Functions and Arm Interface

Two manipulators can be run under RCCL control: a Puma 600 and a Stanford Arm. In both cases, an industrial robot controller from Unimation is used to control the robot joints. The controller's LSI-11 processor, which usually serves to run the VAL code, runs a simple device driver in our system that establishes the communication between the servo code and a real-time arm interface. The arm interface and the rest of the software runs in a VAX minicomputer. The trajectory generator specifies either position or current/torque set points to the servo process.

The arm interface is a function package that allows a user to implement real-time programs in the VAX computer for the control of the manipulators. Programs written using this system execute as two tasks running in parallel: a *control task* or *background process,* which executes at sample rate, and a *planning task* or *user process,* which provides high-level directives to the control task. These tasks are arranged in a two-level hierarchy. The control level executes at high priority in a noninterruptable context, while the planning task executes in a conventional, time-sharing context. Both levels communicate with the robot through predefined data structures. A "C" structure, called **how,** contains information describing the state of the arm, while a structure called **chg** is used to control the arm. The global structure **how** is simply read by the application program to obtain the robot state information, while joint level commands can be specified by setting appropriate fields in the global structure **chg.** The levels communicate with each other using shared memory. In order to implement these features under Unix, it has been necessary to take liberties with the user-operating system interface. These are described in the Appendix.

The communications among levels have been found to fall into the following categories (Lloyd 1985):

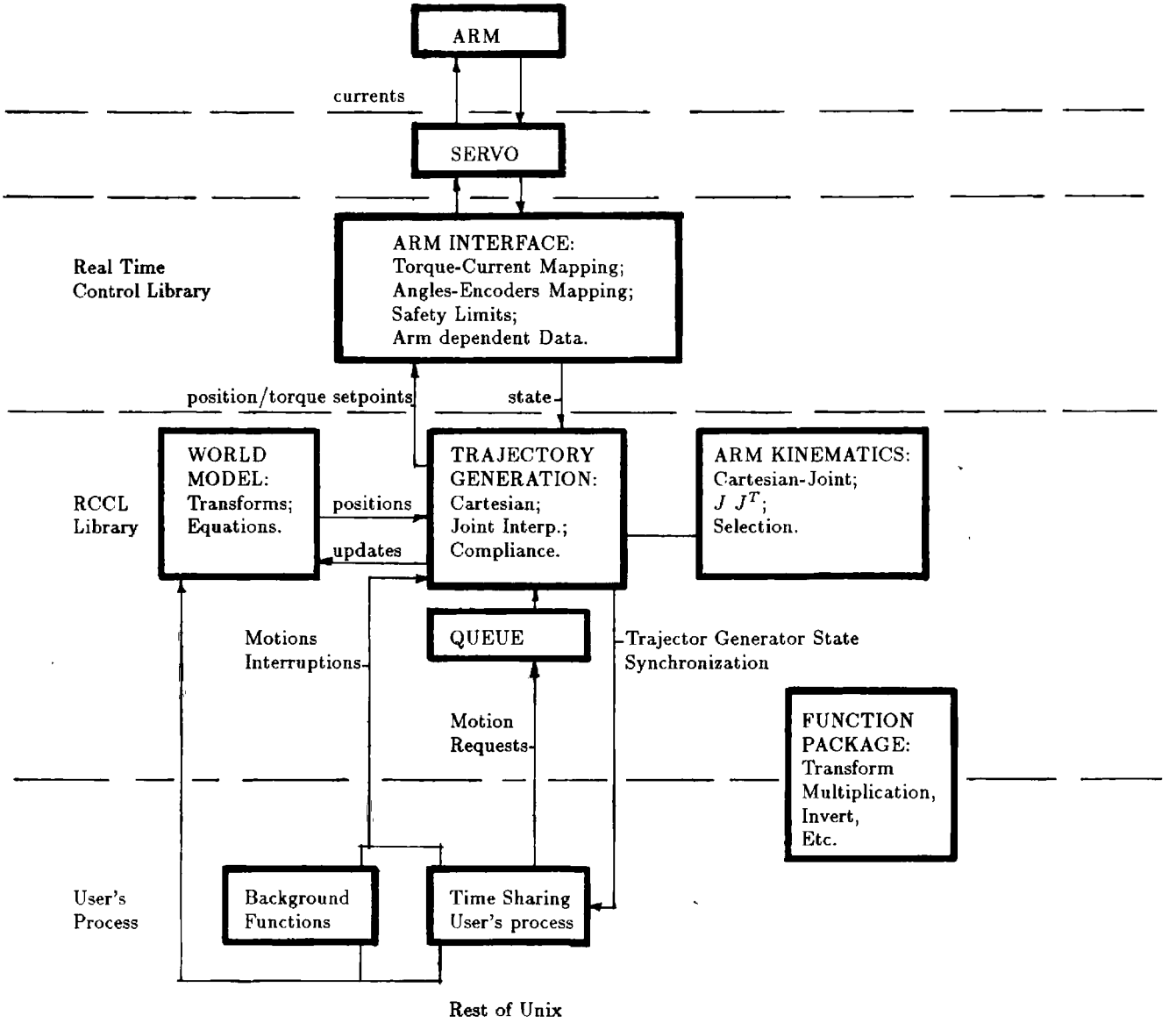
Directives from the planning level. Commands and associated parameters are sent to the control level from the planning level, either through global variables or through a motion queue.

Feedback from the control level. Information computed by the control level is placed in a global area where it may be sampled at the discretion of the planning level.

Synchronization. The tasks synchronize their activities through the setting of global flags.

These communication activities may be compared to the ones described by the designers of a hierarchical control system for legged vehicles (Schwan et al. 1985):

Asynchronous communication with data loss.
Synchronous communication without data loss.



Synchronous or asynchronous communication with possible loss of data.

Finally, a set of mapping functions allows the user to program in terms of physical units (mm, radians, newton-meters) instead of encoder counts and motor currents. When joint torques are measured or specified, Coulomb friction terms are automatically removed or added. The details of the arm interface are fully described in Hayward 1983*a*; Zhang 1983; and Lloyd 1985. The robot programs written using the actual RCCL functions have access to all the features of the arm interface.

2.2. World Model

The RCCL world model is a set of geometrical situations described by closed kinematic chains. Each situa-

tion corresponds to a homogeneous transformation equation (Paul 1981). These equations are directly implemented in terms of linked data structures. Equations are dynamically created and deallocated. Functionally, the world model describes the relative locations of chains of coordinate frames. The values of these transformations can be explicitly specified in the program text; read from files; and asynchronously or synchronously modified during the task execution from sensor readings, computations, or other sources of information (see Figure 1).

The world model enables the user to program robot tasks in Cartesian coordinates. It is derived from the PAL robot programming system (Takase, Paul, and Berg 1979). The basic component is a 4×4 frame transformation matrix that describes the position and orientation of one frame with respect to another. These matrices, called *homogeneous transformations,* possess a number of properties.

Let A be the transform that describes the position of frame F_2 with respect to F_1, and let B be the transform that describes frame F_3 with respect to F_2. The product AB is also a transform and describes the position of F_3 with respect to F_1. Frame transformations are thus easily composed. The inverse of a transform, obtained at low computational cost, is also a transform. For example, A^{-1} describes the position of F_1 with respect to F_2. A *transform* can be interpreted as the description of one frame with respect to another, or as a transformation performed on the first frame. Differential transforms are used to express generalized forces, differential motions, and velocities among various frames. Transforms not only lead to efficient computer implementations but are also powerful mathematical tools. For this reason, they have been used traditionally in manipulation, arm kinematics, dynamics, computer vision, and computer graphics.

In order to illustrate our discussion, we shall borrow a simple task example from the AL system user's manual (Mujtaba and Goldman 1981). The task involves a table top, a robot, and two blocks. The task is to grasp and move the first block, then to grasp and stack the second block on top of the first one. Figures 2A and 2B show the six intermediate states necessary to describe this task. The states are expressed in terms of frame transformations and coincidence. For the sake of simplicity, no mention is made of trajectory generation and collision avoidance.

2.2.1. *Description Using Affixment*

The AL language and other languages (e.g., Latombe and Mayer 1981) model the geometric relationship of the frame with the **AFFIX** statement:

```
AFFIX f1 TO f2 AT trans
```

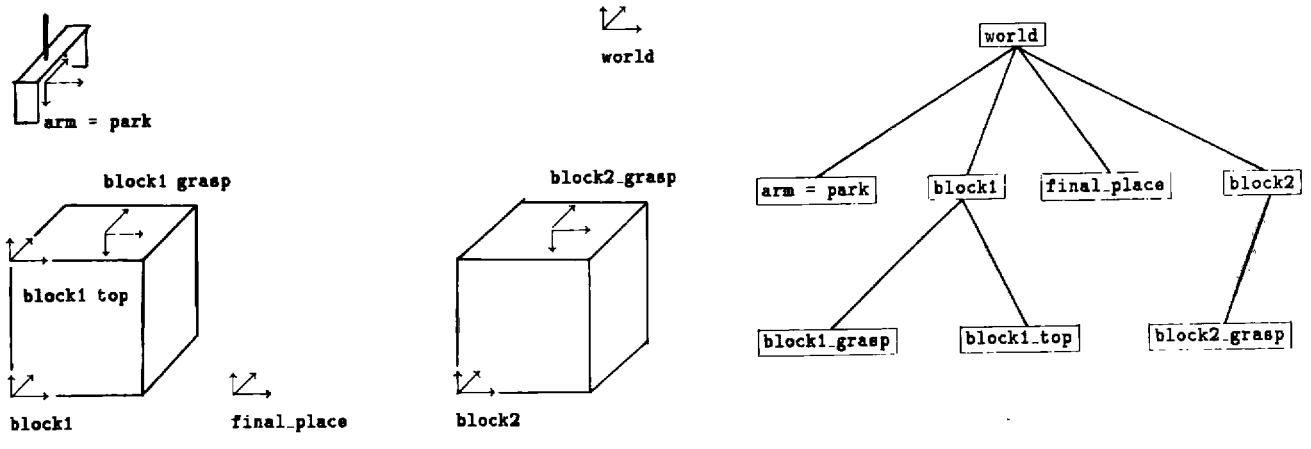
where the transform **trans** defines the position of **f2** with respect to **f1**. The relationship is symmetric: whenever **f1** or **f2** moves, the other frame also moves. With *affixment,* a situation is modeled as a tree in which frames correspond to nodes or leaves, and transformations correspond to links, as in Figs. 2A and 2B. State changes are requested through the **MOVE** statement:

```
MOVE f2 TO f3
```

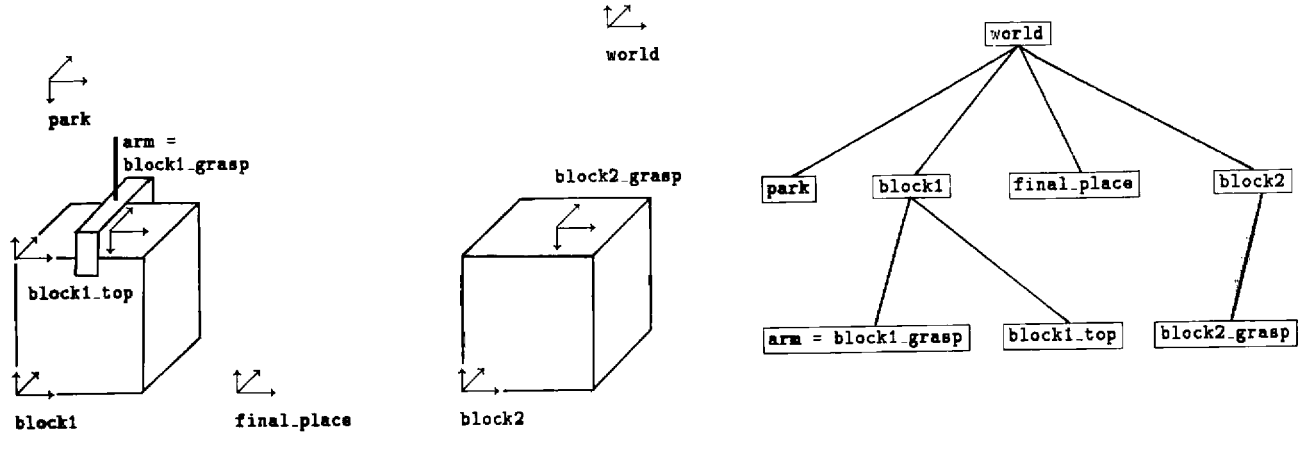
The execution of this **MOVE** statement involves a walk through the affixment tree to determine the movable frame **arm** and the trajectory that lead to the coincidence of **f2** with **f3**. When the motion is performed, the position of all the frames affixed to the movable frame are updated.

The task is described by the following sketch of statements:

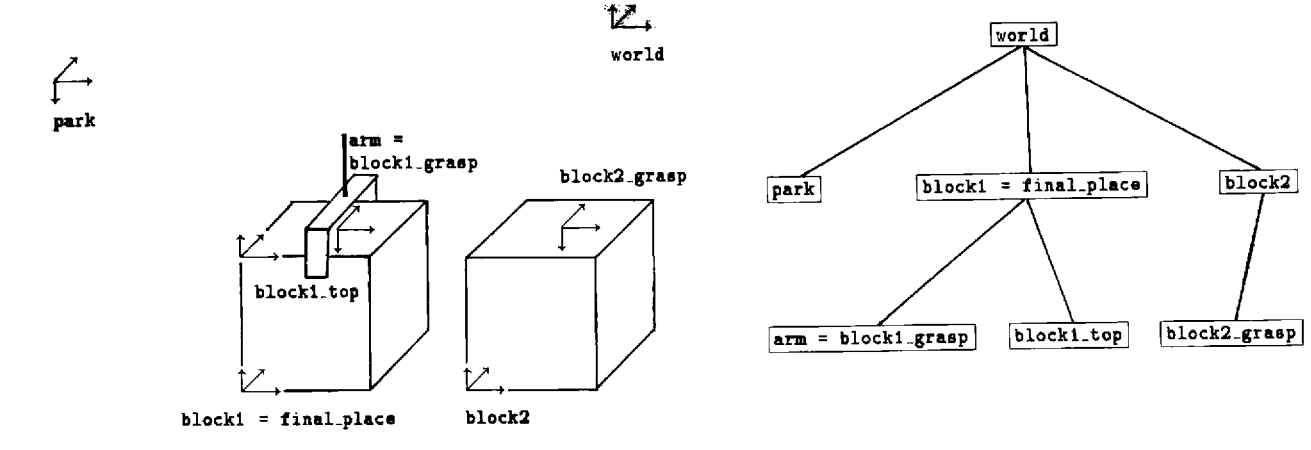
```
AFFIX block1_grasp TO block1 AT ...
AFFIX block2_top TO block2 AT ...
AFFIX block2_grasp TO block2 AT ...
MOVE arm TO block1_grasp
GRASP
AFFIX block1 TO arm
MOVE block1 TO final_place
RELEASE
UNFIX block1 FROM arm
MOVE arm TO block2_grasp
GRASP
AFFIX block2 TO arm
MOVE block2 TO block1_top
RELEASE
UNFIX block2 FROM arm
MOVE arm TO park
```



state 1

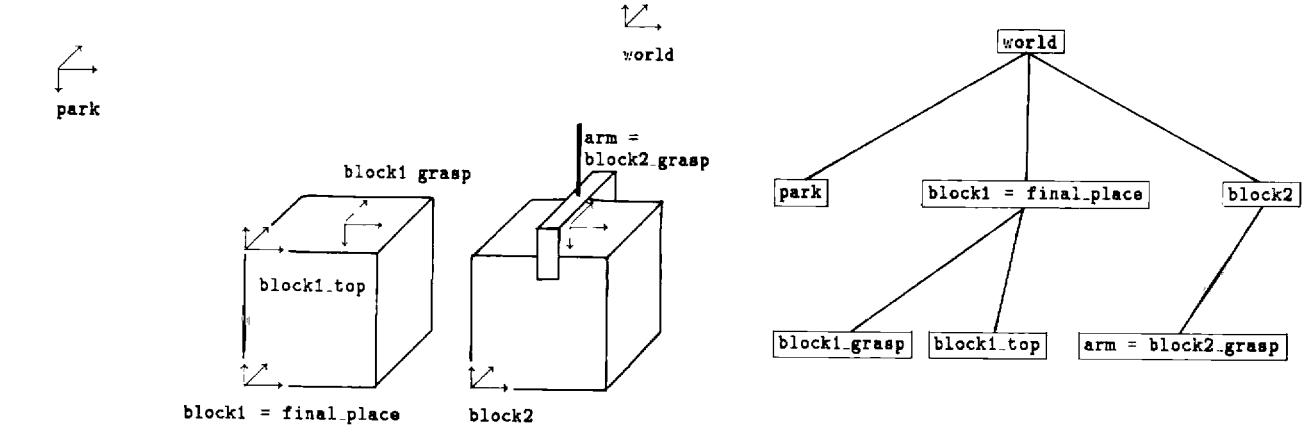


state 2

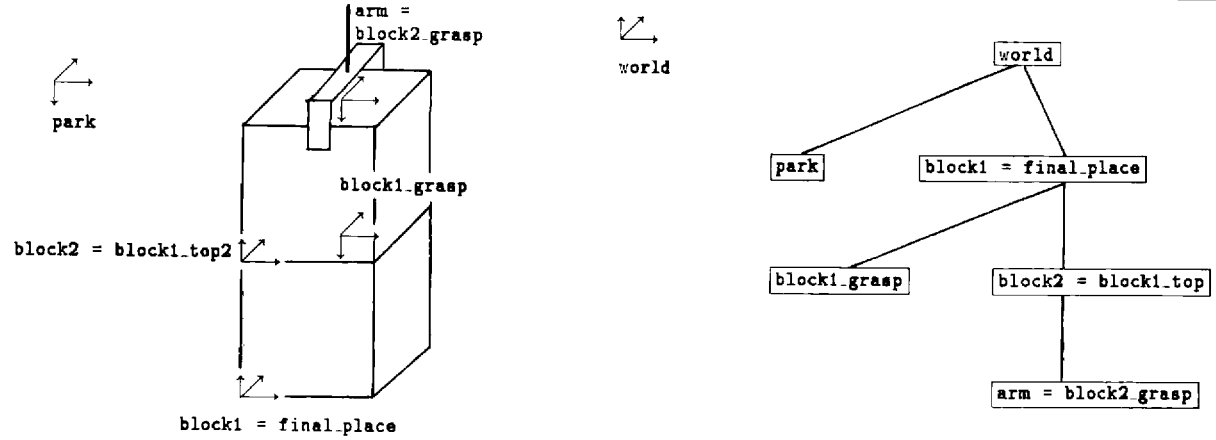


state 3

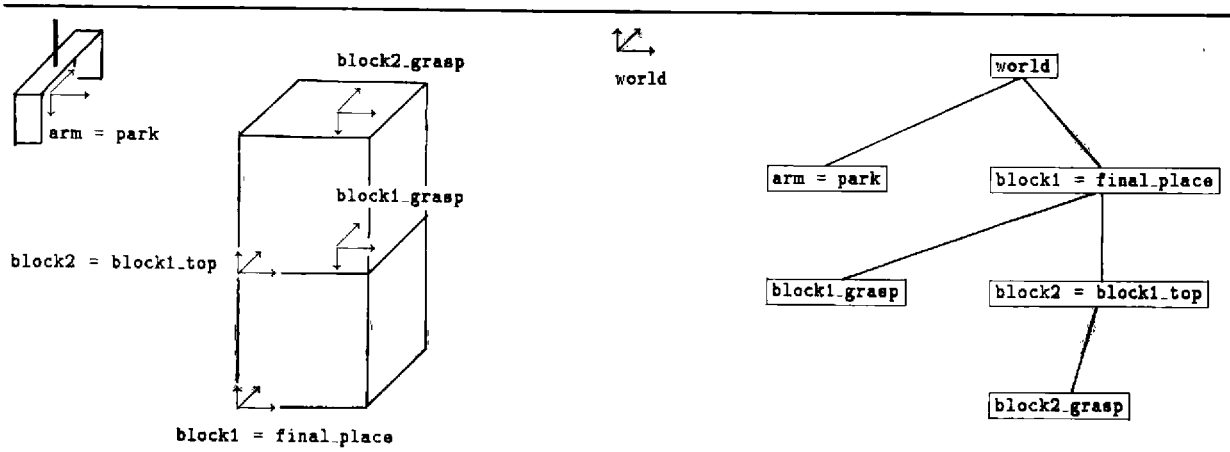
Figure 2a



state 4



state 5

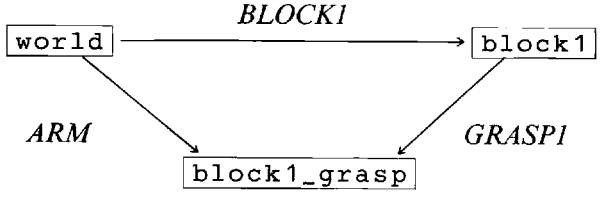


state 6

Figure 2b

2.2.2. Description with Position Equations

In the RCCL system, the notion of frame does not exist explicitly. Instead, all the transformations involved in position description are explicit. Frames are uniquely defined by their mutual relationships, and only the frames involved in a position description are considered at one time. For example, consider state 2 of the task, as in figure 2A. The following graph is obtained by establishing a link between the frame **block1_grasp** and the frame **world:**



ARM, BLOCK1, and *GRASP1* are transformations. Eliminating the frames, an equivalent representation of this graph is a transformation equation:

$$ARM = BLOCK1\ GRASP1.$$

Following this principle, the six intermediate states of the task can be described in terms of six labeled equations:

```
(park)          ARM = PARK
(get_block1)    ARM = BLOCK1 GRASP1
(place_block1)  ARM = FINAL_PLACE GRASP1
(get_block2)    ARM = BLOCK2 GRASP2
(stack_block2)  ARM = FINAL_PLACE TOP1 GRASP2
(park)          ARM = PARK
```

The computer representation of these equations is straightforward, and their solution for the transform *ARM* is simple regardless of their complexity. Once these equations have been specified to the system, the program is sketched as follows:

```
MOVE(get_block1)
CLOSE
MOVE(place_block1)
OPEN
MOVE(get_block2)
CLOSE
MOVE(stack_block2)
OPEN
MOVE(park)
```

2.2.3. RCCL Position Equations

As shown above, the RCCL system only uses transformations to describe the robot workspace. Position equations reflect the spatial structure of the task independently from the flow of control. There is no need to keep track of the affixment relationships in the flow of the program, because this information is contained in the layout of the equations. By the addition of terms to the equations, the user can refine the descriptions without changing the structure of the programs.

We shall now consider the structure of commonly encountered equations. A frame is first assigned to the last link of the manipulator. We call *T6* the transform that describes the position of the last link of the manipulator with respect to its base. When the manipulator is required to move to some known location *POS,* also described with respect to the manipulator's base, the position equation is

$$T6 = POS. \tag{1}$$

We may wish, however, to specify that a certain frame described with respect to the last link of the manipulator (at the tip of a gripper, for example) must reach the given position. The equation can then be changed to

$$T6\ E = POS. \tag{2}$$

We now want to describe the position for grasping a pin lying on a table. We write:

$$B\ T6\ E = T\ P\ PG. \tag{3}$$

In this equation, the transforms are:

- *B* is the position of the base of the robot with respect to an arbitrary reference frame.
- *T6* is the position of the manipulator's last link with respect to its base.
- *E* is the position of the controlled frame with respect to the manipulator's last link.
- *T* is the position of the table with respect to the arbitrary reference frame.
- *P* is the position of the pin with respect to the table.
- *PG* is a grasping position with respect to the pin.

If the pin is to be inserted into a hole of an assembly,

we define two more transforms:

AS is the position of the assembly with respect to the table.
H is the position of the hole with respect to the assembly.

The corresponding equation is

$$B\ T6\ E = T\ AS\ H\ PG. \tag{4}$$

A task description includes a number of these equations. The system computes trajectories such that these equations are successively satisfied. A motion from one position to the next is generated by the system by implicitly inserting an additional transformation in the equation. This transformation *DRIVE(s)* represents a rotation about an axis and a straight-line translation that will drive the robot from one state to the next. The rotation and the translation are linearly dependent on a scalar variable *s*. The *DRIVE(s)* transformation is inserted into the position equation to the right of the controlled frame selected by the user. The position equations can always be rewritten as follows:

$$T6_1 = R_1\ P_1\ E_1, \tag{5}$$
$$T6_2 = R_2\ P_2\ E_2, \tag{6}$$
$$\cdots = \cdots,$$
$$T6_i = R_i\ P_i\ E_i, \tag{7}$$
$$\cdots = \cdots$$

where the P_i's describe the controlled frames, the R_i's are the transform expressions to the left of the P_i's, and the E_i's are the transform expressions to the right of the P_i's. In order to express the motions from one position to the next, we write the first transform expression in terms of the destination position

$$T6_1 = R_2\ {}^2P_1\ E_2, \tag{8}$$

with

$${}^2P_1 = R_2^{-1}\ R_1\ P_1\ E_1\ E_2^{-1}. \tag{9}$$

During motion, the position equation is evaluated as

$$T6 = R_2\ {}^2P_1\ DRIVE(s)\ E_2 \quad for \quad 0 \leq s < 1, \tag{10}$$

with

$$DRIVE(0) = Identity, \tag{11}$$
$$DRIVE(1) = {}^2P_1^{-1}\ P_2. \tag{12}$$

The goal position is obtained when $s = 1$, such that

$$T6 = R_2\ P_2\ E_2. \tag{13}$$

In RCCL, the transformations that make up the terms of the position equations actually fall into one of the following categories:

Constant transformations are used to represent the geometrical relationships of features and objects that remain unchanged during the execution of trajectories. This is the basic type found in most other robot programming systems. Constant transformations are internally premultiplied by the system before the actual computation of the trajectories in order to reduce the computational load.

Variable transformations can be read and written throughout the execution of the user process. The resulting trajectories will immediately reflect the modifications made to those transforms. The programmer is then responsible for providing small smooth changes in order to obtain meaningful results. The changes can occur asynchronously, and this type of transformation is usually useful for tracking applications when the sensory information cannot be made available at sample rate. This is an example of communication with data loss.

Hold transformations can be modified at arbitrary instants by the user process. When a motion request involving such a transformation is issued, the system makes a copy of it. The copy then becomes part of the motion request. These transforms find their application when positional information cannot be obtained within a predictable period of time. They allow the user process to perform input-output operations with a transformation database, for example, without having to stop the arm. This is an example of communication with no loss of data.

Functionally defined transformations are attached to

a "C" function. During the computation of the corresponding trajectory, the function is evaluated at sample rate and is expected to compute values for the transformations. If the values are functions of some parameter (time, for example), one obtains parametrized trajectories. If the values are functions of sensor readings, one obtains synchronous sensory feedback for control of the arm. The applications are limited by the execution time of the function, which must fit in the allocated CPU time slot. Tracking and other active path correction methods are easily implemented, however.

Position equations may include any combination of transformation types. Section 3.4, Sensor Integration, will show examples of the use of these transformations.

2.3. Trajectory Generation

The RCCL system handles two types of trajectories. In both cases, the manipulator is controlled to move toward a target position described by a transformation equation, but the two modes differ by the way intermediate points are generated. In *joint mode,* for each motion request, the final manipulator position is computed by Eq. (7). The corresponding joint set point is then obtained with the inverse kinematic solution of the arm. Intermediate set points are linearly interpolated in joint space. This type of motion leads to efficient trajectories, but the path of the controlled frame is not always easily predictable. The other type of motion, or *Cartesian mode,* uses Eq. (10) to compute the manipulator position at sample rate. Joint set points are once again obtained by the inverse kinematics. The path of the controlled frame is then determined by the parametric transform *DRIVE*(*s*). These path generation techniques are described in more details in Paul (1981).

Smooth transitions between each path segment are provided by an interpolating, quartic polynomial. Since the position equations may contain arbitrary parametric transformations, unpredictable velocity changes may occur at the beginning of the transition. Discontinuities of that nature also occur when the control is switched from *joint interpolated* to *Cartesian* path generation, or vice versa. These cases are handled by adding a third-order polynomial to the quartic transition. The slope of the polynomial is made equal to the measured extra velocity at the beginning of the transition. The slope is set to zero at the end of the transition. One important feature of the trajectory generator is its capacity to initiate a transition at arbitrary instants.

2.4. Force Control

The RCCL system allows the user to program the manipulator so that it exerts forces and torques along or around selected directions. The manipulator is then said to perform compliant motions. This capability raises both the problem of the motion specification and of the control of the arm. Force controllers have been implemented by Raibert and Craig (1981) and Salisbury (1980) using feedback signals from wrist force sensors. For practical reasons, we have implemented a version of Paul and Shimano's (1976) compliant motion scheme. Force specifications are expressed in the controlled frame. Compliant motion is obtained by matching the manipulator joints with each programmed compliant direction. For each direction, the joints most suitable for providing the desired forces or torques are selected and are force or torque controlled, instead of being position servoed. The amount of joint force or torque to exert the desired forces or torque is obtained by the transposed Jacobian matrix and offset by gravity compensation terms (Paul 1981):

$$\tau = \mathbf{S}\,({}^{\mathbf{R}}\mathbf{J})^{T}\,\mathbf{T}_{\mathbf{C}\rightarrow\mathbf{R}}({}^{\mathbf{C}}\mathbf{f}) + \mathbf{G}, \qquad (14)$$

where

τ is the vector of forces or torques applied to the joints.
$\mathbf{S}$ is a selection vector composed of 0's and 1's (the joints corresponding to the 0 elements are position servoed).
${}^{\mathbf{R}}\mathbf{J}$ is the Jacobian matrix, computed in frame R.
$\mathbf{T}_{\mathbf{C}\rightarrow\mathbf{R}}$ is the function mapping forces expressed in frame C into frame R.
${}^{\mathbf{C}}\mathbf{f}$ is the desired forces and torques expressed in the controlled frame C.

R is the frame in which the Jacobian matrix is computed.
G is the gravity compensation term obtained from the dynamics.

We used the method of Renaud (1981) to compute the Jacobian matrix. The matrix is computed in link four coordinates because it leads to the greatest simplicity for manipulators like the Puma or the Stanford arm.

For reasons of simplicity, we used this force control scheme, although it is only an approximation since the contributions of the nonselected joints are ignored. In our installation, the Stanford arm controller was modified to incorporate joint torque control (Fisher 1981; Luh, Fisher, and Paul 1983) and good quality compliant motions could be demonstrated. The Puma robot controller can drive the joint motors with current specifications. A method for relating joint torques to currents, which takes into account the Coulomb friction effects, was implemented by Zhang (1983). Although the method lacks accuracy in the case of the Puma robot, compliant behavior could also be demonstrated for experiments such as the insertion of a peg into a hole with loose tolerance.

As the selected manipulator joints cause motions that never exactly match the compliant directions, any motion along or around these directions causes unwanted motions along or around orthogonal directions. These effects are eliminated by computing compensating motions. The basic position equation is once again modified during compliant motions to

$$T6 = R\ P\ DRIVE(s)\ COMPLY\ E. \tag{15}$$

The terms of the *COMPLY* transform are computed by transforming the differential motions of the compliant joints back into Cartesian space using the Jacobian matrix. The *COMPLY* transform is reset to identity before being updated. The unwanted motions are canceled each time a new solution is obtained:

$$\mathbf{j_e} = j_o - j_d, \tag{16}$$
$$^{R}x_e = {}^{R}J\, j_e, \tag{17}$$
$$^{C}x_e = S\, M_{R\to C}(^{R}x_e), \tag{18}$$
$$COMPLY = COMPLY\ \Delta(^{C}x_e), \tag{19}$$

where

$\mathbf{j_e}$ is the joint error vector computed from the desired joint position j_d and the observed joint position j_o.
$^{R}x_e$ is the Cartesian position error computed in the frame R in which the Jacobian matrix is expressed.
$^{C}x_e$ is the Cartesian error in the controlled frame.
$M_{R\to C}$ is a function mapping forces and torques expressed in frame R into frame C.
Δ is a function that builds a differential transform from a differential motion vector.

2.6. Synchronization

In ordinary robot programming languages, the **MOVE** statements are implicitly synchronized with the arm motions. In the best cases, however, the choice is left to the user to decide if the flow of the program, after a **MOVE** statement, must proceed when the corresponding motion is initiated or when it is about to terminate.

For RCCL we wanted to provide a larger amount of generality because we were stressing system integration. As the motion requests and the associated position equations fully describe the desired motions, a general queueing mechanism and a set of synchronization primitives have been implemented, at the expense of greater programming complexity. Several motion requests can be programmed ahead, making the user process available to perform simultaneous computations. Synchronization becomes necessary when the program flow depends on some external sources of information, such as sensors and motion termination conditions, or when *variable transformations* are used to cause incremental modifications to the manipulator position.

There is no provision for a dynamic management of the motion queue, as we felt that such a facility would become useful only when on-line, collision-free, path-finding algorithms become practical.

3. Introduction to Robot Programming with RCCL

In this section, we would like to show some aspects of robot programming with RCCL and give the flavor of the primitive functions.

3.1. Location Description

A set of RCCL functions allows the user to dynamically create transformations. The basic call is

```
t = newtrans(name, type);
```

where **t** is a pointer to the created transform, **name** is a character string used by the system to keep traces of the program execution, and **type** specifies the desired kind of transformation. Transforms are created as identity transforms. A family of calls, however, both creates and initializes transforms. For example,

```
t = gentr_rot(name, px, py, pz, v, a);
```

creates a transform of name **name,** which is made up of a translation part **px, py, pz** and a rotation of angle **a** around vector **v**. Similar functions are provided for dealing with Euler angles, "roll pitch and yaw" angles, and the like. Of course, users may write their own functions. The following statement creates a position equation:

```
p = makeposition
        (name, lhs, EQ, rhs, TL, t);
```

where **p** is a pointer to the newly created position equations. The argument **name** is also used to generate traces of the program execution. The arguments **lhs** and **rhs** are lists of pointers to previously created transforms. The list **lhs** must contain the system defined pointer **t6** to the *T6* transform. The argument **t** must belong to the **lhs** or **rhs** list and specifies the controlled frame. For example, the equation $R\ T6\ E = C\ O$ will be created by

```
p = makeposition
   ("P", r, t6, e, EQ, c, o, TL, e);
```

assuming that the controlled frame is described by the transform pointed by **e** with respect to the last link of the manipulator.

To a user, a position appears as a "C" structure that can be viewed as a position descriptor:

```
struct position {
        char *name;
        int code;
        float scal;
        event end;
};
```

in which the entry **code** is set upon termination of a motion to this position to reflect the reason of the termination. The value **scal** varies from 0 to 1, while the motion is performed. This is useful for generating parametrized motions or to synchronize the user process at some intermediate point of the trajectory path segment. Finally, the entry **end** is an event count that is signaled upon termination of the corresponding motion.

3.2. Motion Specification

The call

```
move(p)
```

causes a motion request to be transmitted to the trajectory generator. When all previous requests are processed, the system actuates the arm so that the position equation pointed by **p** becomes and remains true, if no other request is pending. This effect is obtained by periodically reissuing the last motion request whenever the queue becomes empty. If the last position equation contained functionally defined transforms, they will continue to be evaluated. This is desirable if the robot is tracking a moving coordinate frame. If an absolute stop is required, the system provides a built-in position equation that always reflects the current position of the arm.

Some motion parameters, when set, affect all subsequent motions until they are reset to another value:

`setval(tv, rv)`: specify translational and rotational velocity.
`setmode(m)`: motion mode, Cartesian or joint.
`setconf(c)`: request an arm configuration change.
`sample(s)`: change the sampling period.

Another set of parameters affects one subsequent motion:

`setime(ta, ts)`: specify travel duration **ts**, transition time **ta**.
`evalfn(fn)`: specify the function **fn** to be run in the background during the motion.
`distance(spec, values)`: specify any combination of small translations or rotations, expressed in the controlled frame, as modifiers for the goal position.
`limit(spec, values)`: specify forces and torques limit values along or around selected directions. The same function serves to specify maximum differential motions.
`comply(dirs, values)`: causes the arm to enter active comply mode when the next motion begins and to exert forces or torques until reset in position servo mode.
`lock(dirs)`: reset the arm in position servo mode along or around the selected directions.
`update(trans, equa)`: calculates the value of the transform **trans** in the equation when the motion ends.

This set of functions serves to build a motion request packet that contains all the information necessary to generate a motion. The functions listed above only fill entries in a "C" structure. The choice of these functions is somewhat arbitrary, and any program able to supply such a motion request packet could control the arm equally well.

3.3. Synchronization

Synchronization is achieved through two basic mechanisms: (1) user process suspension and (2) motion interruption. Two primitives can be used to synchronize the program flow:

```
waitfor(event)
waitas(predicate)
```

The macro **waitfor** suspends the program execution until the specified event occurs. Event counts are signaled by the trajectory generator or by the background functions set up by the user. They are waited for by the user's foreground process. The macro **waitas** repeatedly evaluates its argument until it yields a nonzero value, and then allows the program execution to proceed.

The use of the **scal** and **end** fields in a position descriptor leads to various coding combinations. For example, let **p0, p1, p2,** and **p3** describe the four corners of a square. Let the manipulator move around the square, causing synchronous processing to begin each time the manipulator moves through **p0**:

```
for (i = 0; i < 4; ++i) {
        move(p0);
        move(p1);
        move(p2);
        move(p3);
}
waitfor(p0->end);
printf("starting first square\n");
waitfor(p0->end)
printf("starting second square\n");
etc...,
```

Consider the implementation of a grasping sequence in three steps. First, the tool is moved to an approach position defined by translating the final position along the negative z-axis of the tool frame. Then, the tool is moved to the final position. In the last step, the arm is commanded to stop at the final position for 400 milliseconds and the gripper to close at the middle of this period of time.

```
p->end = 0;              /* reset count     */
distance("dz", -10.);/* backup 10 mm    */
move(p);                 /* move approach   */

move(p);                 /* move final      */
setime(0, 400);          /* during 400 ms   */
move(p);                 /* stay there      */
```

```
waitfor(p->end)        /* wait for motion*/
waitfor(p->end)        /* completion     */
waitas(p->scal > .5) /* wait half time */
CLOSE                  /* close hand     */
```

The following example takes advantage of the program asynchrony with respect to the motions in order to obtain locations stored in a database. The access time is not predictable. The function **gettr** obtains values until **NO** is returned. Although the response time of **gettr** is random, there is no need to stop the arm:

```
e = gentr...
ref = gentr...
loc = newtrans("LOC", hold);

p = makeposition
  ("P", t6, e, EQ, ref, loc, TL, e);

while(gettr(loc) != NO) {
        move(p);
}
```

The robot will wait for new data if it can execute the motions faster than **gettr** can provide values. If this is not the case, the overflow of the motion queue needs to be prevented. The program is slightly modified to make use of the global variable **nbrequest** maintained by the system to reflect the number of nonserved requests.

```
while (gettr(loc) != NO) {
        waitas(nbrequest < MAX)
        move(p);
}
```

Next, we would like to illustrate how motions can be interrupted on external events. The following program causes the arm to move to some location and to stop at the position it occupies when the user hits **<return>** at the terminal. Within the same position equation, the **update** primitive records the location at the time the motion is interrupted. We shall introduce two more system variables: **nextmove** and **completed.** The variable **nextmove** interrupts a motion whenever it is set to a nonzero value, and its value is stored in the **code** field of the corresponding position descriptor. The variable **completed** is an event signaled whenever the motion queue becomes empty.

```
update(loc, p);
move(p);
printf
("hit <return> to stop the arm\n");
getchar();
nextmove = YES;
waitfor(completed)
if (p->code != YES)
        printf("The arm was in 'P'
        when <return> was hit\n");
```

3.4. Sensor Integration

Sensors are used to modify the behavior of a robot at run time and allow it to deal with uncertainties in time and space. There is a wide variety of sensors and information likely to be collected. The RCCL primitives are neither concerned with particular sensors, nor with the nature of the acquired information, nor by the way it is processed. The primitives provide the means for an efficient utilization of sensory information by the robot.

3.4.1. Active Path Correction

Active path correction is obtained by synchronously updating functionally defined transformations from sensor readings. The next example demonstrates this technique with a proximity sensor. The sensor, fixed with respect to the manipulator's last link, measures the distance along the z-axis of the controlled frame. The corrections are made along the same direction. The manipulator is programmed to move close to a surface. This type of motion is conceptually similar to a compliant motion, because the motion along z is determined by the geometry of the surface. A functional transform is used to cause the arm velocity along the z-axis to be proportional to the position

error. We assume that the global variable **sensor** is updated at sample rate to reflect the sensor readings.

```
/* User's process: */

z = gentr...
table = gentr...
disp = gentr...
e = gentr...
sens = newtrans("SENS", sensfn);

p1 = makeposition
     ("P1", z, t6, e, EQ, table, TL, e);
p2 = makeposition
     ("P2", z, t6, e, EQ, table, disp, TL, e);

...
distance("dz", 10.);/* causes overshoot */
evalfn(mon);
move(p1);
move(p2);

...
/* Background functions: */

mon()
{
if (sensor > OFFSET)
   nextmove = YES;
}

sensfn(t)
TRSF_PTR t;
{
t->p.z += (sensor - OFFSET) * gain;
}
```

3.4.2. Force Control Example

We shall now show how the Puma manipulator has been programmed to turn a crank in the active compliance mode. The controlled frame is set to keep a fixed relationship with the handle. This is obtained with two functionally defined transforms, both rotations about the shaft axis of equal magnitudes but of opposite directions. Because of the transform **HANDLE**, the rotations axes are offset by the length of the handle. Two compliant directions, expressed in the controlled frame, are required for this task (Paul 1981).

In the following example, the dimension of the handle is supposed to be a known, but the position of the shaft will be taught via the **teach** function, which is also included in the library. This **teach** function, implemented in terms of RCCL primitives, has the same call convention and conceptually serves the same purpose as the **update** primitive:

```
/* User's process: */

rotpx = newtrans("ROTPX", pxfn);
rotnx = newtrans("ROTNX", nxfn);
handle = gentr_trsl("HANDLE", ...
e = gentr...
z = gentr...

get = makeposition("GET",
      z, t6, e, EQ, shaft, handle, TL, e);
turn = makeposition("TURN",
       z, t6, e, EQ, shaft, rotpx,
       handle, rotnx, TL, e);

move(get);
teach(shaft, get);
CLOSE
comply("fx fz", 0., 0.);
update(handle, get);
turns = 4;                    /* 4 turns */
setime(200, 4000 * turns);    /* 4 sec per turn */
move(turn);
waitfor(turn->end);
OPEN
lock("fx fz");
distance("dx", -50.);
move(get);                    /* depart */
...
/* Background function attached to the functional transforms: */

pxfn(t)
TRSF_PTR t;
{
rot(t, xunit, turn->scal*360*turns);
}

nxffn(t)
TRSF_PTR t;
{
rot(t, yunit, - turn->scal*360*turns);
}
```

4. Conclusion

The integration of robot manipulator control within the "C" programming language has been performed as a library of functions, in the same manner as is input-output. Manipulator task description has been separated into two parts: a world model and motion scheduling. Positions descriptions are implemented in terms

of graphs representing closed kinematic chains. The elements of these chains are homogeneous transformations that may be parameterized in terms of time, and sensor readings. Motion requests allow the user to specify a desired position together with force and distance modifiers. The position information gained as motions are made may be reintegrated in the world model by means of an **update** primitive.

RCCL is a powerful programming system and may be extended as desired, since programming is in the "C" under the Unix system. For example, one version of the system generates off-line trajectory files suitable for graphics display, instead of running the manipulator.

We are considering using interpreted languages, such as Lisp, as a base to develop robot programs in an interactive fashion and using RCCL as the underlying, real-time control system. Furthermore, if specialized languages are needed, the software packages LEX and YACC (Johnson 1980), available under Unix, provide a suitable environment for rapidly creating these languages.

The programming of robot manipulators requires a level of language suitable to express motion algorithms. We believe that languages such as Pascal and "C" are of the appropriate level. The fact that robot motion algorithms expressed in these languages are not simple reflects the complexity of the algorithms rather than the complexity of the language. Simple application robot packages could, of course, be written in RCCL and would have all the attributes of user friendliness as well as the limitations. Finally, we have found the system useful as a research tool, since new algorithms can be tested at any level of the control hierarchy. The system is currently being transferred to a number of small machines and currently supports a number of research projects.

5. Acknowledgments

We wish to thank the following persons who contributed to this project: Bill Fisher, Hong Zhang, Juan Juan, and George Goble. We also wish to thank John Lloyd, who helped us with the section on the arm interface, and the reviewers, who commented on an earlier draft of this paper.

Appendix

The material in this section is based on Lloyd (1985). The VAX/Unix architecture required some modifications to be made to the standard software. These are:

- —The VAX is a virtual memory machine. Segments of the program can be present either in fast memory or paged out on secondary storage. Since the control-level software is executed in kernel mode at elevated priority, all the codes and data that it references must be resident in fast memory. A special locking mechanism and a linking procedure have been developed for this purpose.
- —The address translation is temporarily altered at interrupt time to allow the control process and the planning process to access a shared area of memory.
- —A particular problem associated with running functions in kernel mode on VAX systems is that run-time, hardware-detected errors will result in a system crash. Because it is assumed that the operating system is largely bug free, any errors that do occur should result in system crash. A modification was made so that while the user functions are executing, a flag is raised indicating their presence to the kernel. Hardware errors occuring at this time are hence recognized as belonging to the control program, and instead of causing a crash, they result in a Unix error signal being sent to the control program.

References

Fisher, D. W. 1981. The modification of a robotic manipulator and digital controller to incorporate both force and position control. Master's thesis, Purdue University, Department of Electrical Engineering.

Hayward, V. 1983*a*. Robot real time user's manual. TR-EE 83-42. West Lafayette, Ind.: Purdue University, Department of Electrical Engineering.

Hayward, V. 1983*b*. RCCL Version 1.0 User's Manual. TR-EE 83-46. West Lafayette, Ind.: Purdue University, Department of Electrical Engineering.

Johnson, C. S. 1980 (Aug.). Language development tools with the Unix operating system. *Computer*, pp. 16–21.

Kernighan, B. W., and Ritchie, D. M., "The C Programming Language," Prentice-Hall, 1980.

Kosman, D. 1986. Adapting a high-level robot control environment for an industrial robot. M.Eng. thesis, McGill University, Department of Electrical Engineering.

Latombe, J. C., and Mayer, E. 1981 (Tokyo). LM: A high-level language for controlling assembly robots. *Proc 11th Int. Symp. Indust. Robots.*

Lloyd, J. 1985. Implementation of a robot control development environment. M.Eng. thesis, McGill University, Department of Electrical Engineering.

Lozano-Perez, T. 1983. Robot programming. *Proc. IEEE* 71(7), pp. 821–841.

Luh, J. Y. S., Fisher, W. D., and Paul, R. P. 1983. Joint torque control by direct feedback for industrial robots. *IEEE Trans. Autom. Contr.* AC-28 (2).

Mujtaba, S., and Goldman, R. 1981. AL user's manual. AIM-344, Stanford, Calif.: Stanford University Artificial Intelligence Laboratory.

Nitzan, D., and Rosen, C. A. 1976. Programmable industrial automation. *IEEE Trans. Comp.* C-25 (12) pp. 1259–70.

Paul, R. P. 1977 (Mar.). WAVE: A model based language for manipulator control. *Indust. Robot* 4(1):10–17.

Paul, R. P. 1979. Manipulator path control. *IEEE Trans. Sys., Man, Cyber.* SMC-9 (11):702–711.

Paul, R. P., and Shimano, B. 1976 (San Francisco). Compliance and control. *Proc. Joint Conf. Autom. Contr.*, pp. 679–699.

Paul, R. *Robot Manipulators: Mathematics, Programming, and Control.* MIT Press, 1981.

Raibert, M. H., and Craig, J. J. 1981. Hybrid position/force control of manipulators. *Trans. ASME J. Dyn. Sys., Meas., Contr.* 103:126–133.

Renaud, M. 1981 (Oct., Tokyo). Geometric and kinematic models of a robot manipulator: calculation of the Jacobian matrix and its inverse. *Proc. 11th Int. Symp. Indust. Robots,* pp. 757–763.

Salisbury, J. K. 1980 (Albuquerque, N. Mex.). Active stiffness control of a manipulator in Cartesian coordinates. *Proc. 19th Conf. Decision and Contr.* 1:95–100.

Schwan K., et al. 1985 (St. Louis, Mo.). GEM: operating system primitives for robots and real-time control systems. *IEEE Int. Conf. Robotics and Automation,* pp. 807–813.

Shimano, B. E., Geshke, C. C., and Spalding, C. H. 1983. VAL-II: a robot programming language and control system. *1983 ISRR,* Bretton Woods, N.H.

Takase, K., Paul, R. P., and Berg, E. J. 1979 (Chicago). A structured approach to robot programming and teaching. *IEEE COMPSAC.* pp. 273–278.

Taylor, R. H. 1983. An integrated robot system architecture. Report RC-9824. Yorktown Heights, N.Y.: IBM Research Center. pp. 57–63.

Taylor, R. H., Summers, P., and Meyers, J. 1982. AML: a manufacturing language. *Int. J. Robotics Res.* 1(3).

Zhang, H. 1983. Determination of the simplified dynamics of the Puma manipulator. Tech. Rep. West Lafayette, Ind.: Purdue University, Department of Electrical Engineering.