10

APPLICATION DESIGN ISSUES

In this chapter we discuss some important issues related to the design and the development of complex real-time applications requiring sensory acquisition, control, and actuation of mechanical components. The aim of this part is to give a precise characterization of control applications, so that theory developed for real-time computing and scheduling algorithms can be practically used in this field to make complex control systems more reliable. In fact, a precise observation of the timing constraints specified in the control loops and in the sensory acquisition processes is a necessary condition for guaranteeing a stable behavior of the controlled system, as well as a predictable performance.

As specific examples of control activities, we consider some typical robotic applications, in which a robot manipulator equipped with a set of sensors interacts with the environment to perform a control task according to stringent user requirements. In particular, we discuss when control applications really need real-time computing (and not just fast computing), and we show how time constraints, such as periods and deadlines, can be derived from the application requirements, even though they are not explicitly specified by the user.

Finally, the basic set of kernel primitives presented in Chapter 9 is used to illustrate some concrete programming examples of real-time tasks for sensory processing and control activities.

10.1 INTRODUCTION

All complex control applications that require the support of a computing system can be characterized by the following components:

1. The **system** to be controlled. It can be a plant, a car, a robot, or any physical device that has to exhibit a desired behavior.
2. The **controller**. For our purposes, it will be a computing system that has to provide proper inputs to the controlled system based on a desired control objective.
3. The **environment**. It is the external world in which the controlled system has to operate.

The interactions between the controlled system and the environment are, in general, bidirectional and occur by means of two peripheral subsystems (considered part of the controlled system): an *actuation* subsystem, which modifies the environment through a number of actuators (such as motors, pumps, engines, and so on), and a *sensory* subsystem, which acquires information from the environment through a number of sensing devices (such as microphones, cameras, transducers, and so on). A block diagram of the typical control system components is shown in Figure 10.1.

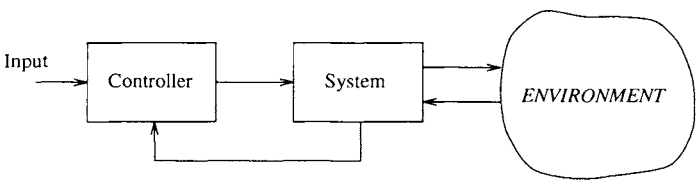

Figure 10.1 Block diagram of a generic control system.

Depending on the interactions between the controlled system and the environment, three classes of control systems can be distinguished:

1. Monitoring systems,
2. Open-loop control systems, and
3. Feedback control systems.

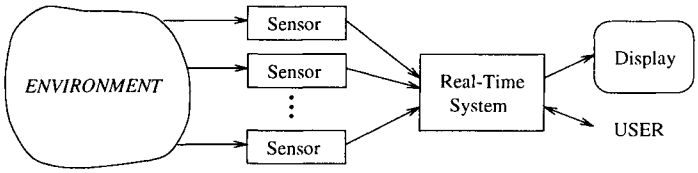

Figure 10.2 General structure of a monitoring system.

Monitoring systems do not modify the environment but only use sensors to perceive its state, process sensory data, and display the results to the user. A block diagram of this type of system is shown in Figure 10.2. Typical applications of these systems include radar tracking, air traffic control, environmental pollution monitoring, surveillance, and alarm systems. Many of these applications require periodic acquisitions of multiple sensors, and each sensor may need a different sampling rate. Moreover, if sensors are used to detect critical conditions, the sampling rate of each sensor has to be constant in order to perform a correct reconstruction of the external signals. In these cases, using a hard real-time kernel is a necessary condition for guaranteeing a predictable behavior of the system. If sensory acquisition is carried out by a set of concurrent periodic tasks (characterized by proper periods and deadlines), the task set can be analyzed off-line to verify the feasibility of the schedule within the imposed timing constraints.

Open-loop control systems are systems that interact with the environment. However, the actions performed by the actuators do not strictly depend on the current state of the environment. Sensors are used to plan actions, but there is no feedback between sensors and actuators. This means that, once an action is planned, it can be executed independently of new sensory data (see Figure 10.3).

As a typical example of an open-loop control system, consider a robot workstation equipped with a vision subsystem, whose task is to take a picture of an object, identify its location, and send the coordinates to the robot for triggering a pick and place operation. In this task, once the object location is identified and the arm trajectory is computed based on visual data, the robot motion does not need to be modified on-line; therefore, no real-time processing is required. Notice that real-time computing is not needed even though the pick and place operation has to be completed within a deadline. In fact, the correct fulfillment of the robot operation does not depend on the kernel but on other factors, such as the action planner, the processing speed of visual data, and the

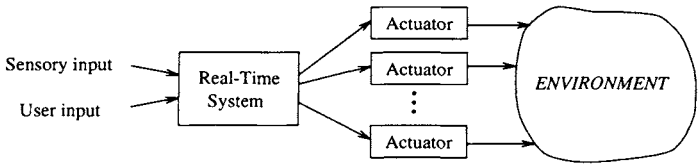

Figure 10.3 General structure of an open-loop control system.

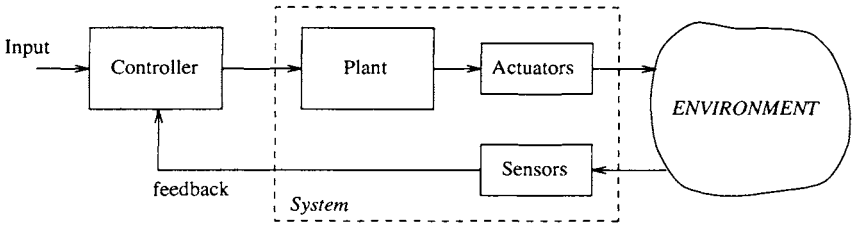

Figure 10.4 General structure of a feedback control system.

robot speed. For this control problem, fast computing and smart programming may suffice to meet the goal.

Feedback control systems (or closed-loop control systems) are systems that have frequent interactions with the environment in both directions; that is, the actions produced by the actuators strictly depend on the current sensory information. In these systems, sensing and control are tied together, and one or more feedback paths exist from the sensory subsystem to the controller. Sensors are often mounted on actuators and are used to probe the environment and continuously correct the actions based on actual data (see Figure 10.4).

Human beings are perhaps the most sophisticated examples of feedback control systems. When we explore an unknown object, we do not just see it, but we look at it actively, and, in the course of looking, our pupils adjust to the level of illumination, our eyes bring the world into sharp focus, our eyes converge or diverge, we move our head or change our position to get a better view of it, and we use our hands to perceive and enhance tactile information.

Modern "fly-by-wire" aircrafts are also good examples of feedback control systems. In these aircrafts, the basic maneuvering commands given by the pilot are converted into a series of inputs to a computer, which calculates how the

physical flight controls shall be displaced to achieve a maneuver, in the context of the current flight conditions.

The robot workstation described above as an example of open-loop control system can also be a feedback control system if we close a loop with the camera and use the current visual data to update the robot trajectory on-line. For instance, visual feedback becomes necessary if the robot has to grasp a moving object whose trajectory is not known a priori.

In feedback control systems, the use of real-time computing is essential for guaranteeing a predictable behavior; in fact, the stability of these systems depends not only on the correctness of the control algorithms but also on the timing constraints imposed on the feedback loops. In general, when the actions of a system strictly depend on actual sensory data, wrong or late sensor readings may cause wrong or late actions on the environment, which may have negative effects on the whole system. In some case, the consequences of a late action can even be catastrophic. For example, in certain environmental conditions, under autopilot control, reading the altimeter too late could cause the aircraft to stall in a critical flight configuration that could prevent recovery. In delicate robot assembling operations, missing deadlines on force readings could cause the manipulator to exert too much force on the environment, generating an unstable behavior.

These examples show that, when developing critical real-time applications, the following issues should be considered in detail, in addition to the classical design issues:

1. Structuring the application in a number of concurrent tasks, related to the activities to be performed;
2. Assigning the proper timing constraints to tasks; and
3. Using a predictable operating environment that allows to guarantee that those timing constraints can be satisfied.

These and other issues are discussed in the following sections.

10.2 TIME CONSTRAINTS DEFINITION

When we say that a system reacts in *real time* within a particular environment, we mean that its response to any event in that environment has to be effective, according to some control strategy, while the event is occurring. This means that, in order to be effective, a control task must produce its results within a specific deadline, which is defined based on the characteristics of the environment and the system itself.

If meeting a given deadline is critical for the system operation and may cause catastrophic consequences, the task must be treated as a *hard* task. If meeting time constraints is desirable, but missing a deadline does not cause any serious damage, the task can be treated as a *soft* task. In addition, activities that require regular activation should be handled as *periodic* tasks.

From the operating system point of view, a periodic task is a task whose activation is directly controlled by the kernel in a time-driven fashion, so that it is intrinsically guaranteed to be regular. Viceversa, an aperiodic task is a task that is activated by other application tasks or by external events. Hence, activation requests for an aperiodic task may come from the explicit execution of specific system calls or from the arrival of an interrupt associated with the task. Notice that, even though the external interrupts arrive at regular intervals, the associated task should still be handled as an aperiodic task by the kernel, unless precise upper bounds on the activation rate are guaranteed for that interrupt source.

If the interrupt source is well known and interrupts are generated at a constant rate, or have a minimum interarrival time, then the aperiodic task associated with the corresponding event is said to be *sporadic* and its timing constraints can be guaranteed in worst-case assumptions – that is, assuming the maximum activation rate.

Once all application tasks have been identified and time constraints have been specified (including periodicity and criticalness), the real-time operating system supporting the application is responsible for guaranteeing that all hard tasks complete within their deadlines. Soft and non-real-time tasks should be handled by using a best-effort strategy (or optimal, whenever possible) to reduce (or minimize) their average response times.

In the rest of this section we illustrate a few examples of control systems to show how time constraints can be derived from the application requirements even in those cases in which they are not explicitly defined by the user.

10.2.1 Obstacle avoidance

Consider a wheel-vehicle equipped with range sensors that has to operate in a certain environment running within a maximum given speed. The vehicle could be a completely autonomous system, such as a robot mobile base, or a partially autonomous system driven by a human, such as a car or a train having an automatic braking system for stopping motion in emergency situations.

In order to simplify our discussion and reduce the number of controlled variables, we will consider a vehicle like a train, which moves along a straight line, and suppose that we have to design an automatic braking system able to detect obstacles in front of the vehicle and control the brakes to avoid collisions. A block diagram of the automatic braking system is illustrated in Figure 10.5.

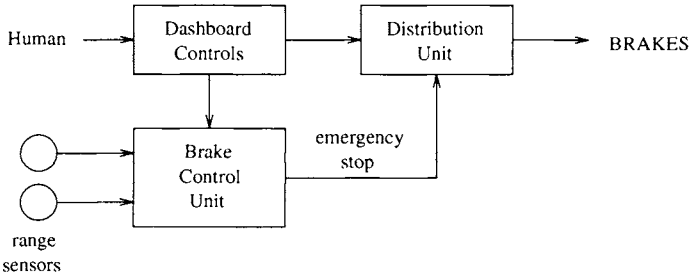

Figure 10.5 Scheme of the automatic braking system.

The Brake Control Unit (BCU) is responsible for acquiring a pair of range sensors, computing the distance of the obstacle (if any), reading the state variables of the vehicle from instruments on the dashboard, and deciding whether an emergency stop has to be superimposed. Given the criticalness of the braking action, this task has to be periodically executed on the BCU. Let T be its period.

In order to determine a safe value for T, several factors have to be considered. In particular, the system must ensure that the maximum latency from the time at which an obstacle appears and the time at which the vehicle reaches

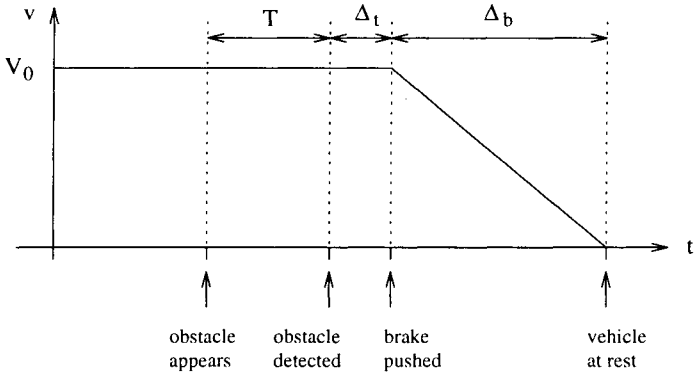

Figure 10.6 Velocity during brake.

a complete stop is less than the time to impact. Equivalently, the distance D of the obstacle from the vehicle must always be greater than the minimum space L needed for a complete stop. To compute the length L, consider the plot illustrated in Figure 10.6, which shows the velocity v of the vehicle as a function of time when an emergency stop is performed.

Notice that three time intervals have to be taken in to account to compute the worst-case latency:

- The detection delay, from the time at which an obstacle appears on the vehicle trajectory and the time at which the obstacle is detected by the BCU. This interval is at most equal to the period T of the sensor acquisition task.
- The transmission delay, Δ_t, from the time at which the stop command is activated by the BCU and the time at which the command starts to be actuated by the brakes.
- The braking duration, Δ_b, needed for a complete stop.

If v is the actual velocity of the vehicle and μ_f is the wheel-road friction coefficient, the braking duration Δ_b is given by

$$\Delta_b = \frac{v}{\mu_f g},$$

where g is the acceleration of gravity ($g = 9.8m/s^2$). Thus, the resulting braking space x_b is

$$x_b = \frac{v^2}{2\mu_f g}.$$

Hence, the total length L needed for a complete stop is

$$L = v(T + \Delta_t) + x_b.$$

By imposing $D > L$, we obtain the relation that must be satisfied among the variables to avoid a collision:

$$D > \frac{v^2}{2\mu_f g} + v(T + \Delta_t). \tag{10.1}$$

If we assume that obstacles are fixed and are always detected at a distance D from the vehicle, equation (10.1) allows to determine the maximum value that can be assigned to period T:

$$T < \frac{D}{v} - \frac{v}{2\mu_f g} - \Delta_t. \tag{10.2}$$

For example, if $D = 100$ m, $\mu_f = 0.5$, $\Delta_t = 250$ ms, and $v_{max} = 30$ m/s (about 108 km/h), then the resulting sampling period T must be less than 22 ms.

It is worth observing that this result can also be used to evaluate how long we can look away from the road while driving at a certain speed and visibility. For example, if $D = 50$ m (visibility under fog conditions), $\mu_f = 0.5$, $\Delta_t = 300$ ms (our typical reaction time), and $v = 60$ km/h (about 16.67 m/s or 37 mi/h), we can look away from the road for no more than one second!

10.2.2 Robot deburring

Consider a robot arm that has to polish an object surface with a grinding tool mounted on its wrist, as shown in Figure 10.7. This task can be specified as follows:

> Slide the grinding tool on the object surface with a constant speed v, while exerting a constant normal force F that must not exceed a maximum value equal to F_{max}.

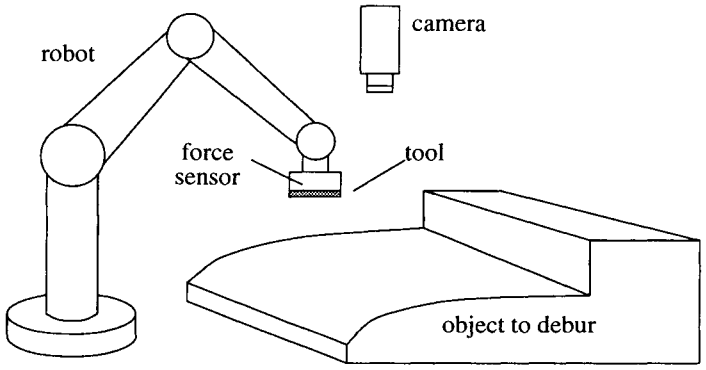

Figure 10.7 Example of a robot deburring workstation.

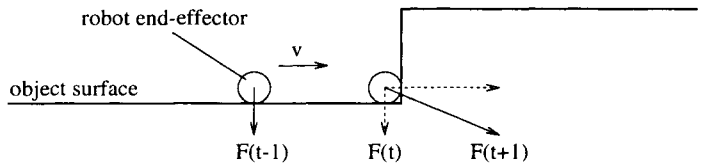

Figure 10.8 Force on the robot tool during deburring.

In order to maintain a constant contact force against the object surface, the robot must be equipped with a force sensor, mounted between the wrist flange and the grinding tool. Moreover, to keep the normal force within the specified maximum value, the force sensor must be acquired periodically at a constant rate, which has to be determined based on the characteristics of the environment and the task requirements. At each cycle, the robot trajectory is corrected based on the current force readings.

As illustrated in Figure 10.8, if T is the period of the control process and v is the robot horizontal speed, the space covered by the robot end-effector within each period is $L_T = vT$. If an impact due to a contour variation occurs just after the force sensor has been read, the contact will be detected at the next period; thus, the robot keeps moving for a distance L_T against the object, exerting an increasing force that depends on the elastic coefficient of the robot-object interaction.

As the contact is detected, we also have to consider the braking space L_B covered by the tool from the time at which the stop command is delivered to the time at which the robot is at complete rest. This delay depends on the robot dynamic response and can be computed as follows. If we approximate the robot dynamic behavior with a transfer function having a dominant pole f_d (as typically done in most cases), then the braking space can be computed as $L_B = v\tau_d$, being $\tau_d = \frac{1}{2\pi f_d}$. Hence, the longest distance that can be covered by the robot after a collision is given by

$$L = L_T + L_B = v(T + \tau_d).$$

If K is the rigidity coefficient of the contact between the robot end-effector and the object, then the worst-case value of the horizontal force exerted on the surface is $F_h = KL = Kv(T + \tau_d)$. Since F_h has to be maintained below a maximum value F_{max}, we must impose that

$$Kv(T + \tau_d) < F_{max},$$

which means

$$T < (\frac{F_{max}}{Kv} - \tau_d). \tag{10.3}$$

Notice that, in order to be feasible, the right side of condition (10.3) must not only be greater than zero but must also be greater than the system time resolution, fixed by the system tick Q; that is,

$$\frac{F_{max}}{Kv} - \tau_d > Q. \tag{10.4}$$

Equation (10.4) imposes an additional restriction on the application. For example, we may derive the maximum speed of the robot during the deburring operation as

$$v < \frac{F_{max}}{K(Q + \tau_d)}, \tag{10.5}$$

or, if v cannot be arbitrarily reduced, we may fix the tick resolution such that

$$Q \leq (\frac{F_{max}}{Kv} - \tau_d).$$

Once the feasibility is achieved – that is, condition (10.4) is satisfied – the result expressed in equation (10.3) says that stiff environments and high robot velocities requires faster control loops to guarantee that force does not exceed the limit given by F_{max}.

10.2.3 Multilevel feedback control

In complex control applications characterized by nested servo loops, the frequencies of the control tasks are often chosen to separate the dynamics of the controllers. This greatly simplifies the analysis of the stability and the design of the control law.

Consider, for instance, the control architecture shown in Figure 10.9. Each layer of this control hierarchy effectively decomposes an input task into simpler subtasks executed at lower levels. The top-level input command is the goal, which is successively decomposed into subgoals, or subtasks, at each hierarchical level, until at the lowest level, output signals drive the actuators. Sensory data enter this hierarchy at the bottom and are filtered through a series of sensory-processing and pattern-recognition modules arranged in a hierarchical structure. Each module processes the incoming sensory information, applying filtering techniques, extracting features, computing parameters, and recognizing patterns.

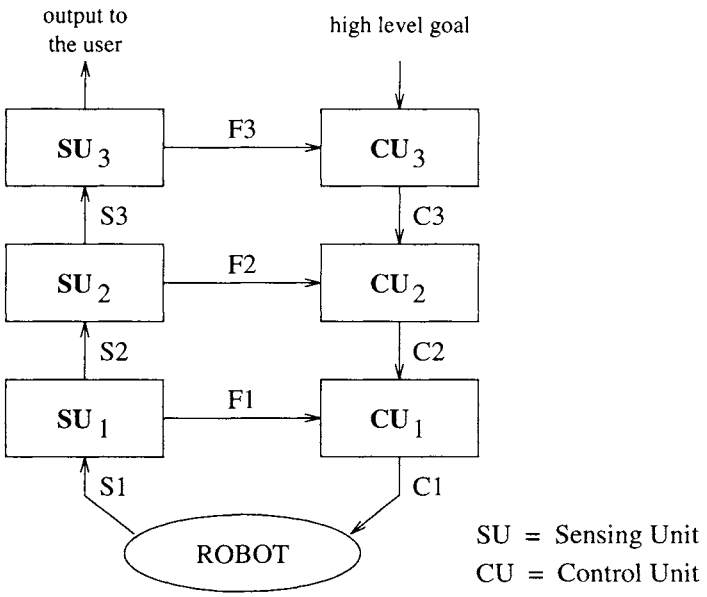

Figure 10.9 Example of a hierarchical control system.

Sensory information that is relevant to control is extracted and sent as feedback to the control unit at the same level; the remaining partially processed data is then passed to the next higher level for further processing. As a result, feedback enters this hierarchy at every level. At the lowest level, the feedback is almost unprocessed and hence is fast-acting with very short delays, while at higher levels feedback passes through more and more stages and hence is more sophisticated but slower. The implementation of such a hierarchical control structure has two main implications:

- Since the most recent data have to be used at each level of control, information can be sent through asynchronous communication primitives, using overwrite semantic and nonconsumable messages. The use of asynchronous message passing mechanisms avoids blocking situations and allows the interaction among periodic tasks running at different frequencies.
- When the frequencies of hierarchical nested servo loops differ for about an order of magnitude, the analysis of the stability and the design of the control laws are significantly simplified.

For instance, if at the lowest level a joint position servo is carried out with a period of 1 ms, a force control loop closed at the middle level can be performed with a period of 10 ms, while a vision process running at the higher control level can be executed with a period of 100 ms.

10.3 HIERARCHICAL DESIGN

In this section, we present a hierarchical design approach that can be used to develop sophisticated control applications requiring sensory integration and multiple feedback loops. Such a design approach has been actually adopted and experimented on several robot control applications built on top of a hard real-time kernel [But91, BAF94, But96].

The main advantage of a hierarchical design approach is to simplify the implementation of complex tasks and provide a flexible programming interface, in which most of the low- and middle-level real-time control strategies are built in the system as part of the controller and hence can be viewed as basic capabilities of the system.

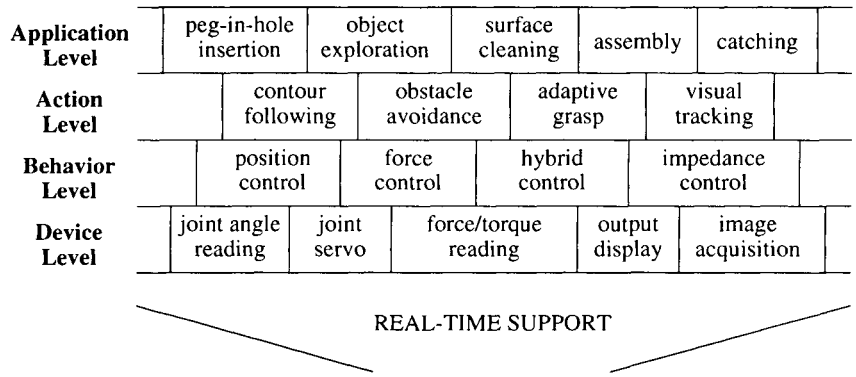

Figure 10.10 Hierarchical software environment for programming complex robotic applications.

Figure 10.10 shows an example of a hierarchical programming environment for complex robot applications. Each layer provides the robot system with new functions and more sophisticated capabilities. The importance of this approach is not simply that one can divide the program into parts; rather, it is crucial that each procedure accomplishes an identifiable task that can be used as a building block in defining other procedures.

The *Device Level* includes a set of modules specifically developed to manage all peripheral devices used for low-level I/O operations, such as sensor acquisition, joint servo, and output display. Each module provides a set of library functions, whose purpose is to facilitate device handling and to encapsulate hardware details, so that higher-level software can be developed independently from the specific knowledge of the peripheral devices.

The *Behavior Level* is the level in which several sensor-based control strategies can be implemented to give the robot different kinds of behavior. The functions available at this level of the hierarchy allow the user to close real-time control loops, by which the robot can modify its trajectories based on sensory information, apply desired forces and torques on the environment, operate according to hybrid control schemes, or behave as a mechanical impedance. These basic control strategies are essential for executing autonomous tasks in unknown conditions, and, in fact, they are used in the next level to implement more skilled actions.

Based on the control strategies developed in the Behavior Level, the *Action Level* enhances the robot capability by adding more sophisticated sensory-motor activities, which can be used at the higher level for carrying out complex tasks in unstructured environments. Some representative actions developed at this level include (1) the ability of the robot to follow an unknown object contour, maintaining the end-effector in contact with the explored surface; (2) the reflex to avoid obstacles, making use of visual sensors; (3) the ability to adapt the end-effector to the orientation of the object to be grasped, based on the reaction forces sensed on the wrist; (4) visual tracking, to follow a moving object and keep it at the center of the visual field. Many other different actions can be easily implemented at this level by using the modules available at the Behavior Level or directly taking the suited sensory information from the functions at the Device Level.

Finally, the *Application Level* is the level at which the user defines the sequence of robot actions for accomplishing application tasks, such as assembling mechanical parts, exploring unknown objects, manipulating delicate materials, or catching moving targets. Notice that these tasks, although sophisticated in terms of control, can be readily implemented thanks to the action primitives included in the lower levels of the hierarchical control architecture.

10.3.1 Examples of real-time robotics applications

In this section we describe a number of robot applications that have been implemented by using the control architecture presented above. In all the examples, the arm trajectory cannot be precomputed off-line to accomplish the goal, but it must be continuously replanned based on the current sensory information. As a consequence, these applications require a predictable real-time support to guarantee a stable behavior of the robot and meet the specification requirements.

Assembly: peg-in-hole insertion

Robot assembly is an active area of research since several years. Assembly tasks include inserting electronic components on circuit boards, placing armatures, bushings, and end housings on motors, pressing bearings on shafts, and inserting valves in cylinders.

Theoretical investigations of assembly have focused on the typical problem of inserting a peg into a hole, whose direction is known with some degree of uncertainty. This task is common to many assembly operations and requires the robot to be actively compliant during the insertion, as well as to be highly responsive to force changes, in order to continuously correct its motion and adapt to the hole constraints.

The peg-in-hole insertion task has typically been performed by using a hybrid position/force control scheme [Cut85, Whi85, AS88]. According to this method, the robot is controlled in position along the direction of the hole, whereas it is controlled in force along the other directions to reduce the reaction forces caused by the contact. Both position and force servo loops must be executed periodically at a proper frequency to ensure stability. If the force loop is closed around the position loop, as it usually happens, then the position loop frequency must be about an order of magnitude higher to avoid dynamics interference between the two controllers.

Surface cleaning

Cleaning a flat and delicate surface, such as a window glass, implies large arm movements that must be controlled to keep the robot end-effector (such as a brush) within a plane parallel to the surface to be cleaned. In particular, to efficiently perform this task, the robot end-effector must be pressed against the glass with a desired constant force. Because of the high rigidity of the glass, a small misalignment of the robot with respect to the surface orientation could cause the arm to exert large forces in some points of the glass surface or loose the contact in some other parts.

Since small misalignments are always possible in real working conditions, the robot is usually equipped with a force sensing device and is controlled in real time to exert a constant force on the glass surface. Moreover, the end-effector orientation must be continuously adjusted to be parallel to the glass plane.

The tasks for controlling the end-effector orientation, exerting a constant force on the surface, and controlling the position of the arm on the glass must proceed in parallel and must be coordinated by a global planner, according to the specified goal.

Object tactile exploration

When working in unknown environments, object exploration and recognition are essential capabilities for carrying out autonomous operations. If vision does not provide enough information or cannot be used because of insufficient light conditions, tactile and force sensors can be effectively employed to extract local geometric features from the explored objects, such as shape, contour, holes, edges, or protruding regions.

Like the other tasks described above, tactile exploration requires the robot to conform to a give geometry. More explicitly, the robot should be compliant in the direction normal to the object surface, so that unexpected variations in the contour do not produce large changes in the force that the robot applies against the object. In the directions parallel to the surface, however, the robot needs to maintain a desired trajectory and should therefore be position-controlled.

Strict time constraints for this task are necessary to guarantee robot stability during exploration. For example, periods of servo loops can be derived as a function of the robot speed, maximum applied forces, and rigidity coefficients, as we have shown in the example described in Section 10.2.2. Other issues involved in robot tactile exploration are discussed in [DB87, Baj88].

Catching moving objects

Catching a moving object with one hand is one of the most difficult tasks for humans, as well as for robot systems. In order to perform this task, several capabilities are required, such as smart sensing, visual tracking, motion prediction, trajectory planning, and fine sensory-motor coordination. If the moving target is an intelligent being, like a fast insect or a little mouse, the problem becomes more difficult to solve, since the *prey* may unexpectedly modify its trajectory, velocity, and acceleration. In this situation, sensing, planning, and control must be performed in real time – that is, while the target is moving – so that the trajectory of the arm can be modified in time to catch the prey.

Strict time constraints for the tasks described above derive from the maximum velocity and acceleration assumed for the moving object. An implementation of this task, using a six degrees of freedom robot manipulator and a vision system, is described in [BAF94].

10.4 A ROBOT CONTROL EXAMPLE

In order to illustrate a concrete real-time application, we show an implementation of a robot control system capable of exploring unknown objects by integrating visual and tactile information. To perform this task the robot has to exert desired forces on the object surface and follow its contour by means of visual feedback. Such a robot system has been realized using a Puma 560 robot arm equipped with a wrist force/torque sensor and a CCD camera. The software control architecture is organized as two servo loops, as shown in Figure 10.11, where processes are indicated by circles and CABs by rectangles. The inner loop is dedicated to image acquisition, force reading, and robot control, whereas the outer loop performs scene analysis and surface reconstruction. The application software consists of four processes:

- A sensory acquisition process periodically reads the force/torque sensor and puts data in a CAB named *force*. This process must have guaranteed execution time, since a missed deadline could cause an unstable behavior of the robot system. Hence, it is created as a hard task with a period of 20 ms.
- A visual process periodically reads the image memory filled by the camera frame grabber and computes the next exploring direction based on a user defined strategy. Data are put in a CAB named *path*. This is a hard task with a period of 80 ms. A missed deadline for this task could cause the robot to follow a wrong direction on the object surface.
- Based on the contact condition given by the force/torque data and on the exploring direction suggested by the vision system, a robot control process computes the cartesian set points for the Puma controller. A hybrid position/force control scheme [Whi85, KB86] is used to move the robot end-effector along a direction tangential to the object surface and to apply forces normal to the surface. The control process is a periodic hard task with a period of 28 ms (this rate is imposed by the communication protocol used by the robot controller). Missing a deadline for this task could cause the robot to react too late and exert too large forces on the explored surface, that could break the object or the robot itself.
- A representation task reconstructs the object surface based on the current force/torque data and on the exploring direction. Since this is a graphics activity that does not affect robot motion, the representation process is created as a soft task with a period of 60 ms.

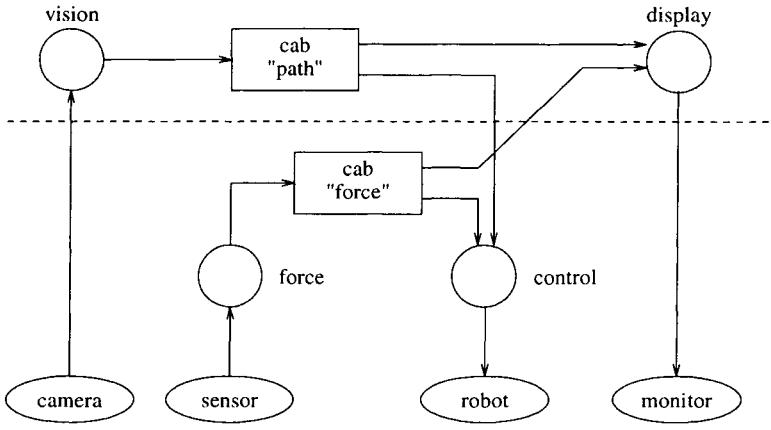

Figure 10.11 Process structure for the surface exploration example.

To better illustrate the application, we show the source code of the tasks. It is written in C language and includes the DICK kernel primitives described in the previous chapter.

```
/*------------------------------------------------------------*/
/* Global constants                                           */
/*------------------------------------------------------------*/
#include "dick.h"              /* DICK header file            */
#define TICK       1.0         /* system tick (1 ms)          */
#define T1         20.0        /* period for force   (20 ms) */
#define T2         80.0        /* period for vision  (80 ms) */
#define T3         28.0        /* period for control (28 ms) */
#define T4         60.0        /* period for display (60 ms) */
#define WCET1      0.300       /* exec-time for force   (ms) */
#define WCET2      4.780       /* exec-time for vision  (ms) */
#define WCET3      1.183       /* exec-time for control (ms) */
#define WCET4      2.230       /* exec-time for display (ms) */
```

```
/*-------------------------------------------------------------*/
/* Global variables                                            */
/*-------------------------------------------------------------*/
cab     fdata;                  /* CAB for force data          */
cab     angle;                  /* CAB for path angles         */
proc    force;                  /* force sensor acquisition    */
proc    vision;                 /* camera acq. and processing */
proc    control;                /* robot control process       */
proc    display;                /* robot trajectory display    */
```

```
/*-------------------------------------------------------------*/
/* main -- initializes the system and creates all tasks        */
/*-------------------------------------------------------------*/
proc    main()
{
        ini_system(TICK);
        fdata = open_cab("force", 3*sizeof(float), 3);
        angle = open_cab("path", sizeof(float), 3);
        create(force,   HARD, PERIODIC, T1, WCET1);
        create(vision,  HARD, PERIODIC, T2, WCET2);
        create(control, HARD, PERIODIC, T3, WCET3);
        create(display, SOFT, PERIODIC, T4, WCET4);
        activate_all();
        while (sys_clock() < LIFETIME) /* do nothing */;
        end_system();
}
```

```
/*-------------------------------------------------------------*/
/* force -- reads the force sensor and puts data in a cab     */
/*-------------------------------------------------------------*/
proc    force()
{
float   *fvect;                         /* pointer to cab data */
        while (1) {
                fvect = reserve(fdata);
                read_force_sensor(fvect);
                putmes(fvect, fdata);
                end_cycle();
        }
}
```

```
/*-------------------------------------------------------------*/
/* control -- gets data from cabs and sends robot set points */
/*-------------------------------------------------------------*/
proc    control()
{
float   *fvect, *alfa;                  /* pointers to cab data */
float   x[6];                           /* robot set-points     */
        while (1) {
                fvect = getmes(fdata);
                alfa = getmes(angle);
                control_law(fvect, alfa, x);
                send_robot(x);
                unget(fvect, fdata);
                unget(alfa, angle);
                end_cycle();
        }
}
```

```
/*-------------------------------------------------------------*/
/* vision -- gets the image and computes the path angle        */
/*-------------------------------------------------------------*/
proc    vision()
{
char    image[256][256];
float   *alfa;                          /* pointer to cab data */
        while (1) {
                get_frame(image);
                alfa = reserve(angle);
                *alfa = compute_angle(image);
                putmes(alfa, angle);
                end_cycle();
        }
}
```

```
/*-------------------------------------------------------------*/
/* display -- represents the robot trajectory on the screen    */
/*-------------------------------------------------------------*/
proc    display()
{
float   *fvect, *alfa;              /* pointers to cab data    */
float   point[3];                   /* 3D point on the surface */
        while (1) {
                fvect = getmes(fdata);
                alfa = getmes(angle);
                surface(fvect, *alfa, point);
                draw_pixel(point);
                unget(fvect, fdata);
                unget(alfa, angle);
                end_cycle();
        }
}
```