Proceedings of the 8th IEEE Euromicro Workshop on Real-Time Systems, L'Aquila, Italy, pp. 133-138, June 1996.

Real–Time Issues in Advanced Robotics Applications

Giorgio Buttazzo
Scuola Superiore S. Anna
Via Carducci, 40 – 56100 Pisa, Italy

Abstract

The aim of this paper is to discuss some important real-time issues involved in the design of complex robotic applications. In particular, the problem of evaluating why and when a robotics application needs real–time computing is discussed first. The second issue concerns the definition of time constraints for each task of the robot. We will show how time constraints, such as periods and deadlines, can be derived from the application, even though they are not explicitly specified in the requirements. As a third issue, we will describe a general hierarchical control architecture, which can be built on top of a real–time system, to greatly simplify the development of complex robotics applications having real–time requirements.

1 Introduction

The main motivation for this paper is to give a precise characterization of real–world robotics applications, so that the theory developed on real–time computing and scheduling algorithms can be practically used in this field to make complex robot systems more reliable. In fact, a precise observation of the timing constraints specified in the control loops and in the sensor acquisition processes is a necessary condition for guaranteeing a stable behavior of the robot, as well as a predictable performance. On the other hand, concrete implementations of real–time tasks may raise hidden problems, which can drive the real–time community to address new interesting issues and investigate productive research areas. For these reasons, we begin our discussion by introducing the essential features that characterize advanced robotics applications and thier related implications on the real-time control architecture.

A fundamental quality that a robot system should have to perform useful operations in unknown conditions is the ability to sense the environment with multiple sensors. The motivation for using multiple sensors is due to the non–ideality of the world and of the sensors. Measurements are noisy, partial, imperfect, and hence one sensor cannot provide the system with reliable data. On the contrary, by using multiple sensors, several different properties can be extracted from an explored object, and the probability of a correct recognition increases substantially. These properties may include geometric features (such as shape, contours, holes, edges, protruding regions), mechanical characteristics (such as hardness, flexibility, elasticity), or thermal properties (such as temperature, thermal conductivity).

However, reading signals coming from sensors and processing sensory data is not sufficient for exhibiting an intelligent behavior. To be adaptive and autonomous, a robot system should be able to discriminate stimuli, classify features, recognize objects, and eventually create a symbolic representation of the sensed environment. We call this complex activity as a "*perception process*". Depending on whether the perception process is, or is not, strictly related to a motor activity, we distinguish between *passive* and *active* perception. As we shall see later, the use of passive or active perception makes a lot of difference in terms of real–time processing requirements.

For passive perception we intend a perceptual activity in which sensors are fixed or are used in static mode, i.e., there are no feedback loops between sensors and actuators, whose movements follow predetermined trajectories. A typical example of passive perception is given by a vision system consisting of a fixed camera, which takes pictures of a scene, recognizes objects, identifies their location, and sends data to a robot arm for pick and place operations. In this task, once the object locations are 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.

On the other hand, active perception is a process that involves dynamic sensing, where movements are utilized as a mean for increasing and driving sensory information. For active perception we intend the ability to not only see and touch objects, but also to manipulate and probe them. Perceptual activity is exploratory, probing, searching, and involves complex tasks, such as recognition and manipulation, which are essential in unstructured environments. Unlike passive perception, where sensors and actuators can

be physically separated, in active perception sensing and control are tied together. Sensors are often mounted on actuators and are used by the robot system to probe the environment and continuously adjust fine movements based on actual data. Active perception is a problem of intelligent control strategies applied to data acquisition processes, which depend on the current state of the data interpretation [3].

For example, 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 sometimes we use our hands.

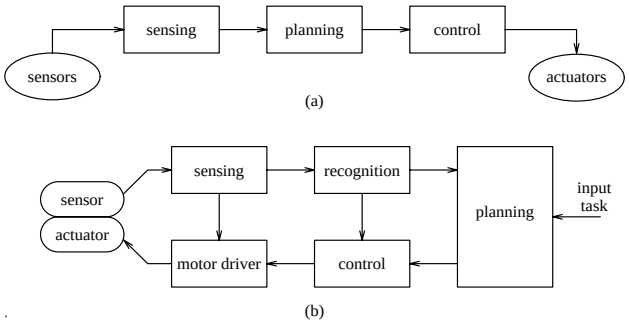

Figure 1. Difference between passive (a) and active (b) perception, in terms of control structures and processing requirements.

Figure 1 schematically illustrates the difference between passive and active perception in terms of control structures and data processing requirements. As we can see, the difference between the two processes causes a radical change in the system architecture. In active perception, the influence that the actuator movements have on the sensor responses forces the system to react in real–time, causing the control architecture to be hierarchically organized in a multilevel structure of feedback loops.

In general, to support a wide range of sensory–motor capabilities, ranging from low level reactions to complex exploratory procedures, the system architecture must be able to handle hierarchical control loops operating at different frequencies. Efficient and time bounded communication mechanisms are also required to close real–time control loops at each level of the hierarchy, including short range reflex arcs, for effective support of guarded movements.

An anthropomorphic approach to this issue has been discussed by Albus [1], who described the organization of a behavior–oriented architecture based on multilevel control loops organized in a hierarchical fashion. More specific control architectures for robotics have been proposed in the literature by [9, 11]. Although some of these solutions are based on highly parallel hardware and can be organized in a modular structure, they are difficult to test and to program, since require the knowledge of many low level details. The result is that these architectures are not practical, and can only be used by experts. For this reason, recently, a lot of effort has been done to design flexible and open programming environments to deal with multisensory perception and control [2, 10, 12]. However, in most of the cases, they are developed above commercial priority–based kernels, that do not support explicit time constraints, and do not have any form of guarantee in task scheduling. As a consequence, the robot may have an unpredictable behavior in heavy load conditions, and cannot be used in critical applications.

The approach presented in this paper provides a solution to this problem by adopting a hierarchical control architecture built on top of a hard real-time kernel, called HARTIK [5], which supports explicit deadline specification and on–line guaranteed scheduling. This allows to achieve predictability in robotics applications with strict real–time requirements. A flexible programming environment also facilitates the development of tasks at different level of abstraction.

The rest of the paper is organized as follows. In section 2, we address the problem of evaluating why and when robotics applications need real–time computing. In section 3, we will consider a specific application and show how time constraints, such as periods and deadlines, can be derived, even though their are not explicitly specified in the requirements. In section 4, we present a general hierarchical design methodology which greatly simplifies the development of complex robotics applications having real–time requirements, and provides a modular framework for programming robot tasks at different level of abstraction. Finally, summary and conclusions are reported in section 5.

2 When is real–time needed?

When implementing a control application, it is not always clear whether real–time computing is a necessary condition for achieving a correct timing behavior. Therefore, a crucial question that one should keep in mind when developing a control task is whether the application requires time constraints. Unfortunately, answering this question is not always so obvious. In fact, there are control applications in which the goal is specified in terms of explicit time requirements, but tasks execution does not need a real–time support.

Consider, for example, a sorting operation, in which a set of N objects, arrived at time t_0, must be classified and sorted by a robot system in M different classes, based on their local features. Suppose that each object has a firm deadline within which it has to be sorted.

This sorting operation can be decomposed into three main

tasks, which involve object recognition, action planning, and robot control. Once the objects are recognized (say by vision), and their deadlines are derived, the purpose of the planning task is to construct a sequence of actions so that each object is sorted by its deadline. Clearly, the action plan has to take into account the time needed by the robot to pick an object and place it in the proper location. However, once the plan is completed, the robot trajectory is determined, and the arm can start its blind motion in a table–driven fashion. Observe that the deadlines associated to the objects do not impose any time constraint in the execution of the robot tasks. The meet of the deadlines only depends on the action plan and on the robot speed, that has to be known in advance. It is also worth to notice that the processing structure of the sorting application is similar to the typical scheme of passive perception, where sensing, planning, and control are quite separated, and must be executed sequentially. In other words, a lack of feedback implies a lack of real–time requirements.

In contrast, there are robotics applications which do not have explicit time requirements, but need real–time support. Consider, for example, a deburring operation, in which a robot arm has to polish the surface of an object with a grinding tool mounted on its wrist. 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}*".

In order to maintain a constant contact force against the object surface, the robot must be equipped with a force/torque 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 frequency, which depends on the environment characteristics and on the task requirements. Note that, in this case, the robot trajectory is not known in advance and hence cannot be precomputed off–line. The robot end–effector has to be moved on the object surface according to current force readings. This means that at each step, the robot has to correct its trajectory in order to maintain the contact force within the specified range.

In this example, time constraints are not explicitly given, however they must be imposed on the tasks execution to guarantee the meet of the application requirements. The specific time constraints for this example will be derived in the next section.

3 Time constraints definition

When we say that a robot 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 according to the characteristics of the robot-environment system.

If meeting a given deadline is critical for the system operation and may cause catastrophic consequences, the task and its associated deadline are said to be *hard*. If meeting time constraints is desirable, but missing a deadline does not cause any serious damage, the task and its deadline are said to be *soft*. In addition to their criticalness, tasks that require regular activations are called *periodic*, whereas tasks which have irregular arrival times are called *aperiodic*.

Once all time critical tasks are identified and time constraints are specified (including criticalness and periodicity) the real-time operating system supporting the application should guarantee that all hard tasks complete within their deadlines, while using a best-effort strategy for soft and non real-time tasks.

To show how to derive time constraints from the application requirements, let us consider the robot deburring example described in the previous section. During the execution of this task, the robot slides the grinding tool on the object surface with constant speed, while exerting a constant normal force against it.

As illustrated in figure 2, 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 $\Delta x_1 = vT$. In case of impact, we also have to consider the space Δx_2 covered from the time at which the stop command is delivered to the time in 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 frequency response with a transfer function having a dominant pole f_0 (as typically done in most cases), then the breaking space can be computed as $\Delta x_2 = v\tau_0$, being $\tau_0 = \frac{1}{2\pi f_0}$. Hence the total distance covered by the robot is given by:

$$\Delta x \;=\; \Delta x_1 + \Delta x_2 \;=\; v(T + \tau_0)$$

If K is the rigidity coefficient of the contact between the robot end–effector and the object, then, in the worst case, the value of the horizontal force exerted on the surface is $F^* = K\Delta x = Kv(T + \tau_0)$. Now, if we want to maintain F^* below a maximum force F_{max}, we must impose that:

$$Kv(T + \tau_0) \;<\; F_{max} \tag{1}$$

which means:

$$T \;<\; (\frac{F_{max}}{Kv} - \tau_0) \tag{2}$$

Notice that, in order to be feasible, the right hand side of condition 2 must not only be greater than zero, but it must be greater than the system time resolution, fixed by the system tick Q. This may impose additional restrictions on the application. For example, we may derive the maximum speed of the robot during the deburring operation as:

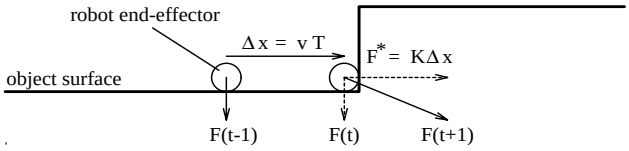

Figure 2. Derivation of the period of the force control task.

$$v < \frac{F_{max}}{K(Q + \tau_0)} \quad (3)$$

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

$$Q \leq (\frac{F_{max}}{Kv} - \tau_0) \quad (4)$$

Once the feasibility is achieved, i.e. condition 4 is satisfied, the result expressed in equation (2) says that stiff environments and high robot velocities requires faster control loops to respect the force limit given by F_{max}.

In more complex 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 3. Each stage 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 stage of the control hierarchy, 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, extracting features, computing parameters, recognizing patterns and applying various types of filters to the sensory data.

Information relevant to the control is extracted and sent to the control module 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, messages can be sent through asynchronous communication primitives, using overwrite semantic and non consumable buffers. The use of asynchronous message passing mechanisms avoid blocking situations and allows the interaction among periodic tasks running at different frequencies.

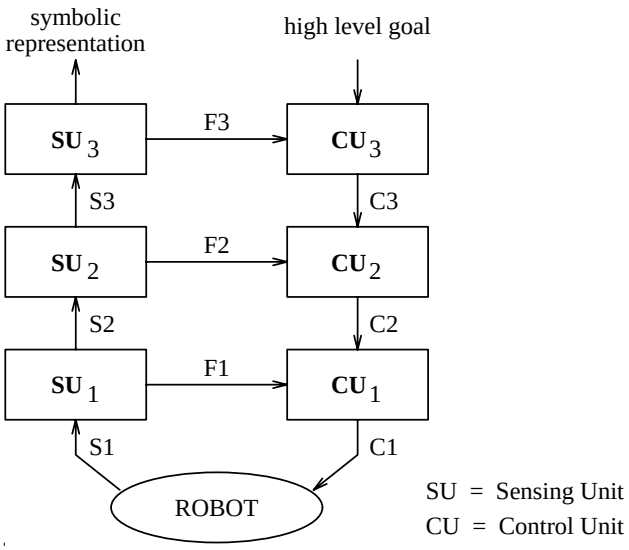

Figure 3. Hierarchical control of a robot system.

- 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 a joint position servo is carried out at the lowest level 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.

4 A hierarchical programming environment

In this section, we present a hierarchical programming environment that can be built on top of a real–time kernel to develop robot control applications. The main purpose of this software structure is to simplify the implementation of complex tasks and to provide a flexible 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 robot system [4].

As shown in figure 4, the control architecture is organized in a hierarchical structure of layers, each of them 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.

This programming environment has been built on top of a hard real–time kernel, called HARTIK [5], specifically designed to develop predictable robotics applications. Briefly, the main features of this kernel include explicit specification of time constraints (such as periods and deadlines) preemptive dynamic scheduling, coexistence of hard, soft, and non real–time tasks, separation between time constraints and importance, deadline tolerances, dynamic guarantee of critical tasks, and asynchronous communication channels particularly suited for exchanging information among periodic control tasks having different rates. A novel scheduling mechanism, described in [7], allows to achieve graceful degradation in overload conditions, integrating the execution of hard periodic and soft aperiodic tasks [13]. Hard aperiodic tasks can also be handled by the system by a slightly different mechanism presented in [14].

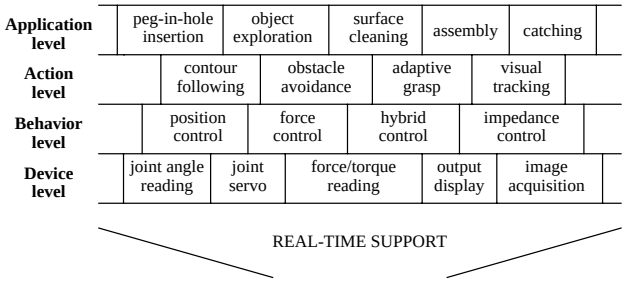

Figure 4. Hierarchical software environment for programming complex robotic applications.

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 are implemented, in order to give the robot different kinds of behavior. The functions available at this level of the hierarchy allows the user to close real–time control loops, by which the robot can modify on line planned 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: a) the ability of the robot to follow an unknown object contour, maintaining the end–effector in contact with the explored surface; b) the reflex to avoid obstacles, making use of visual sensors; c) 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; d) 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 in order to accomplish applicative tasks, such as the assembly of mechanical parts, the exploration of unknown objects, the manipulation of 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.

The hard real-time kernel which supports the application guarantees the timely execution of all critical activities and insures a regular activation of all periodic tasks, even in presence of aperiodic load that may derive from asynchronous events in the environment. In normal load conditions, the scheduling algorithm adopted in the kernel is able to achieve full processor utilization and guarantee a feasible schedule for all hard periodic and aperiodic tasks, without jeopardizing the response time of the soft aperiodic activites. If an overload occurs (for instance, caused by a multiple activations of aperiodic tasks), hard periodic tasks are still guaranteed, while aperiodic tasks are handled by considering their importance value specified by the programmer. This scheduling methodology adds robustness to the system and insures a minimum performance even in critical load conditions.

The control architecture presented above has been used to develop a number of real-time robotic applications, such as assembling mechanical parts, cleaning flat surfaces with unknown orientation, exploring unknown contours with tactile sensors /citeDar87, following moving objects by a mobile camera, and catching moving targets by a robot arm using visual based control /citeBut94. In all these applications, 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, the guaranteed schedule provided by the HARTIK kernel was essential for achieving a stable and predictable behavior of the robot.

5 Conclusions

In this paper we have addressed the problem of using real–time computing in complex robotics applications, in which multiple sensors have to be integrated to cope with unknown environments. We have shown that the real–time features of a control application strongly depend on active perception, which causes the control architecture to be organized in a hierarchical structure of controllers, each characterized by a proper frequency of execution. We also described how time constraints, such periods and deadlines, can be derived from the application, even though their are not explicitly specified in the requirements.

To provide a general design methodology, we have presented a hierarchical control architecture that can be built on top of a real–time system to facilitate the development of complex robotics applications having real–time requirements. The architecture also provides a modular framework for programming robot tasks at different level of abstraction. A number of real–world robotics applications implemented according to the presented hierarchical design methodology, demonstrated the effectiveness of the proposed approach.

References

[1] J. S. Albus. Brains, Behavior, and Robotics, McGraw–Hill, 1981.

[2] R. J. Anderson. "SMART: A Modular Architecture for Robotics and Teleoperation", *Proceedings of the 1993 IEEE International Conference on Robotics and Automation*, Atlanta, Georgia, pp. 416–421, May 1993.

[3] R. Bajcsy. "Active Perception", *Proceedings of the IEEE*, Vol. 76, No. 8, pp. 996–1005, August 1988.

[4] G. C. Buttazzo. "HAREMS: Hierarchical Architecture for Robotics Experiments with Multiple Sensors", *IEEE Proceedings of the Fifth International Conference on Advanced Robotics* ('91 ICAR), Pisa, Italy, June 19–22, 1991.

[5] G. C. Buttazzo. "HARTIK: A Real–Time Kernel for Robotics Applications", *Proceedings of the Real-Time System Symposium*, Raleigh–Durham, December 1993.

[6] G. C. Buttazzo, B. Allotta, F. Fanizza. "Mousebuster: a Robot for Catching Fast Objects", *IEEE Control Systems Magazine*, Vol. 14, No. 1, pp. 49–56, February 1994.

[7] G. C. Buttazzo and J. Stankovic. "Adding Robustness in Dynamic Preemptive Scheduling", in Responsive Computer Systems: Step Toward Fault-Tolerant Real-Time Systems, Edited by D. S. Fussel and M. Malek, Kluwer Academic Publishers, Boston, 1995.

[8] P. Dario and G. C. Buttazzo. "An Anthropomorphic Robot Finger for Investigating Artificial Tactile Perception", *International Journal of Robotics Research*, 6(3), pp. 25–48, Fall 1987.

[9] Y. Fujioka and M. Kameyama. "2400–MFLOP Reconfigurable Parallel VLSI Processor for Robot Control", *Proceedings of the 1993 IEEE International Conference on Robotics and Automation*, Atlanta, Georgia, pp. 149–154, May 1993.

[10] V. Hayward et al.. "The Kali Multi–Arm Robot Programming and Control Environment", *Proceedings of the NASA Conference on Space Telerobotics*, JPL Publication 89-7, January 1989.

[11] D. R. Lefebvre and G. N. Saridis. "A Computer Architecture for Intelligent Machines", *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, Nice, France, pp. 2745–2750, May 1992.

[12] E. C. Maniere, B. Espiau, D. Simon. "Reactive Objects in a Task Level Open Controller", *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, Nice, France, pp. 2732–2737, May 1992.

[13] M. Spuri, G. C. Buttazzo, F. Sensini. "Robust Aperiodic Scheduling Under Dynamic Priority Systems", *Proceedings of the 16th Real-Time Systems Symposium*, Pisa, Italy, December 1995.

[14] M. Spuri and G. C. Buttazzo. "Scheduling Aperiodic Tasks in Dynamic Priority Systems", *Journal of Real-Time Systems*, Vol. 10, No. 2, pp. 179-210, March 1996.