A DISTRIBUTED CONTROL AND COMMUNICATION STRUCTURE FOR MULTIPLE COOPERATING ROBOT AGENTS

Gen'ichi Yasuda

Department of Mechanical Engineering, Faculty of Engineering
Nagasaki Institute of Applied Science
536 Aba-machi, Nagasaki 851-0193, Japan

Abstract: This paper describes a distributed control and communication structure for industrial multirobot systems where several robots and devices have to be controlled simultaneously. A task-level programming and execution system, where task control is performed using Petri nets, is presented. A commercially available OS, Windows NT, is used to develop and debug application programs and to coordinate cooperating robot agents in real time. The control software is implemented on a widely used and powerful PC-based architecture using multithreaded programming in order to achieve efficient utilization of CPU time and real-time vision-based control. Workcell tasks demonstrating robotic handling operations, are successfully implemented using the control system scheme.

Keywords: Multirobot systems; robot agents; Petri nets; distributed cooperative processing; PC; multithreaded programming.

1. INTRODUCTION

The requirements for the industrial robot systems have been changed due to the need of increased functionalities such as real-time sensor based control, multirobot control, and other functions necessary for improving ease of use. Further, the increased need for adaptability and flexibility of manufacturing cells makes it increasingly necessary for robots of different architectures to be used together, although they have operating systems peculiar to themselves. Thus, the requirements for advanced robot control systems or cell controllers, are summarized as follows:

- to control one or more robots of one or more types,
- to interact with and control ancillary equipment,
- to interact with sensing equipment,
- to interact with other parts of the flexible manufacturing system, and
- to provide a user-friendly interface.

Because most of the commercially available robots are equipped with their own controllers, they have to be linked via a common communication system for multirobot applications. On the other hand, centralized control is needed to develop application programs, to download them into the local controllers, and to supervise the whole system. In order for such complex applications to be programmed and executed effectively, an easy-to-use and efficient program execution environment should be provided based on modularity.

The recent hardware and software technology improvements have changed the conventional way of

design for industrial robot controllers. In particular, the PC architecture is an attractive choice with its high-volume cost advantages and standardized interfaces. Because of the hardware improvement, controlling multiple robots and other motion devices, which typically reside in the same workcell, through a PC-based central control unit, becomes feasible, which results in a major impact to controller design. The PC-based architecture can be fully used in place of programmable logic controllers. One of the significant improvements in the PC architecture is the multitasking feature, which is essential for controlling multiple robots in an asynchronous manner. Besides it, the multitasking feature can be used for various purposes such as continuous process monitoring and human friendly host communication.

This paper presents the design and implementation of a runtime supervisory robot control system on a PC-based architecture. The work described in this paper is aimed at practical implementation in factory, and, therefore, commercially available robot controllers are used. For multirobot applications, these controllers communicate with each other on the architecture. Industrial intelligent devices or agents, such as robots, are defined as concurrent software modules, or threads. A multiagent control and communication structure for cooperating robot agents is described using Petri nets. The computing issues in coordinated multirobot control are discussed with respect to multithreaded programming under Windows NT. A case study of the software for an experimental robot system, to pick and place a workpiece using a single or multiple robots, is used to illustrate a central feature of multithreaded programming.

2. SPECIFICATIONS OF MULTIROBOT SYSTEMS

The experimental system for multirobot control is composed of a vision sensor, two or more industrial robots, input and output conveyors, and a simple process machine. The control system is implemented on a PC-based architecture (Pentium II CPU at 450 MHz) with an image capture board. Analog and digital I/O boards provide other low-level sensing and actuation functions. At present, two RS-232C ports are used to connect the control system to two robot controllers, which direct two six-degree-of-freedom industrial robots (MELFA RV-E2 and PUMA 560). These manipulators can be equipped with sensing devices such as six-axis force sensors and proximity sensors.

There are three levels of concurrency among robots in industrial applications: independent level, synchronous level, and coordinated level (Yasuda and Tachibana, 1994). At the independent level, multiple robots perform independent tasks in parallel, but accomplish a goal as a work cell. At the synchronous level, multiple robots perform cooperative or exclusive tasks by sending and receiving signals for synchronization. A typical example is a chain of synchronous tasks for exchange of a workpiece by two robots. At the coordinated level, they work together for the same workpiece at the same time, for example grasping and handling a long beam, by using force sensors and high-speed communication. Usually, the motion of the slave robot is relative to the coordinate frame described by the master robot, while the master robot is executed at programmed speed and motion trajectory. Simultaneous motion is also required for all robots to start and stop at the same time.

For industrial multirobot systems, a task-level programming and execution system has been designed with task-level commands at the independent and synchronous levels. In cases where, for example, the angle or position at which a workpiece must be presented varies, it is possible to have the workpieces pass between two robots. The conceptual diagram of a task-level program for exchange of a workpiece by two robots is shown in Table 1. The following task-level commands are used:

(1) *Pick*. This task moves the hand of a robot to the position and orientation of a specified workpiece, then holds it.
(2) *Put*. This task moves the hand of a robot to the specified position and orientation, holding a workpiece, to transfer it to another robot.
(3) *Get*. This task moves the hand of a robot approaching to the hand of another robot, then gets a workpiece.
(4) *Place*. This task moves the hand of a robot approaching to an object, holding a workpiece, then puts it on a surface of the object, releases it, and departs from that place.

Table 1 Task-level program for exchange of a workpiece by two robots

Robot 1		Robot 2	
Step No.	Task	Step No.	Task
1	*Pick* A	1	
2	*Put* 2	2	*Get* 1
3		3	*Place* B

A,B: workpieces 1,2: robots

In the task-level program, synchronization is introduced for coordinated operation of robots. As shown in Step No.2 of the program, if commands for robot 1 and robot 2 are described on the same line, they are executed simultaneously, and the commands on the next line are not executed until they terminate. Command parameters are the labels of robots, workpieces, or other objects in the environment. Further, for cooperative tasks, shared variables or semaphores are predefined. These variables are used as event objects for asynchronous communication. On the other hand, for exclusive tasks, critical regions where access is limited to one robot at a time are predefined.

Task-level commands are translated into motion-level commands with appropriate control parameters. For example, the "*Pick* the workpiece A" command described above is divided into the sequence of the following commands: *approach*, *grasp*, and *lift up*. Then synchronization and communication operations are inserted into the required situations of the single-robot program. Finally, motion-level robot programs with primitive commands are generated in terms of a general purpose, robot-independent representation. Motion-level commands are converted to the form for a particular robot type when needed and sent to the robot controller. The following five primitive commands are used: *move*, *open*, *close*, *signal*, and *wait*. The *open* and *close* commands control the opening and closing of the gripper. The *signal* and *wait* commands are communication commands for synchronization using event objects.

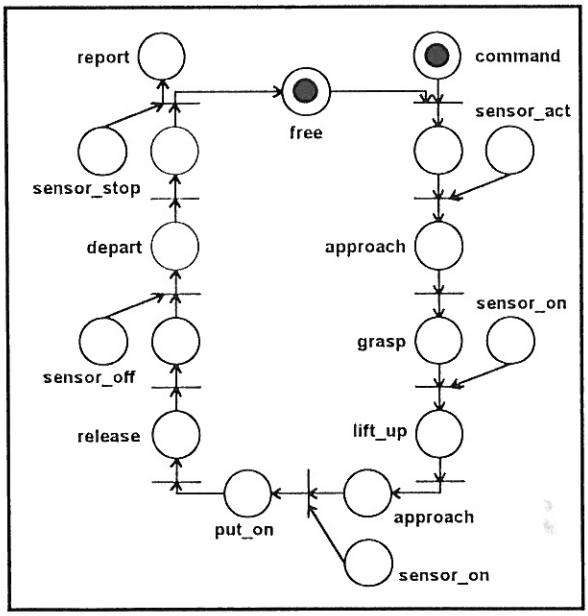

Fig.1. Petri net representation of pick and place operation.

Petri nets can be utilized for the modelling and analysis of asynchronous, discrete event systems with concurrency (Freedman, 1991). A Petri net based implementation in a parallel programming language for exchange of a workpiece by two robots was demonstrated (Yasuda and Tachibana, 1994). The Petri net can be transformed into control computer code and executed on a parallel distributed control architecture, for example, transputer network. Fig.1 shows a Petri net representation of pick and place operation by a single robot. In the Petri net, a place represents a state of executing a motion-level command, waiting for some message, or detecting a specified sensor signal. A transition represents the beginning or end of a command. A token represents the status of a robot. For multirobot applications with two or more robots, new tokens are simply added and the overall Petri net is constructed similarly.

The discrete event behaviors of robot agents are managed based on state transition diagrams like Petri nets with their asynchronous and synchronous interactions. The Petri-net based control system is composed of the task control module and robot control modules, and system coordination is performed through communication between the task control module and robot control modules (Yasuda and Tachibana, 1996). A robotic task is specified as a Petri net in the task control module. For cooperative or exclusive tasks at the synchronous level, the Petri net is used as a means to communicate the status of robots. The task control module executes the Petri net and arbitrates conflicts among robot agents. If a transition fires, it sends the next motion-level command in terms of the names of the current transition and the next place to the associated robot control module. According to the name of the place, the robot control module monitors the specified sensor signal and, if the condition is satisfied, sends the status to the task control module. The conditions of monitoring are predefined in the knowledge base of the robot control module.

The robot control modules are designed with a subsumption architecture approach to realize autonomous reactive control tasks of robots using multiple sensors in dynamic environments (Yasuda, 1998). The control procedure is also represented as a Petri net and implemented on a multiprocessing architecture.

A 3D graphics system under Windows NT is used as an interactive programming and debugging device. All robots motions are taught interactively one by one. The menu and question and answer methods of programming are used, because a major aim of the system is to make it easier for a user with little

experience of computing to be able to program a robot. In order to enable the user to control more than one robot via a friendly interface the control system must have knowledge of how the robots are to be programmed. For example, it is assumed that the workpiece will either be picked up from above or from the side. The user can set the status of the robots, confirm the synchronization, detect collisions among robots, and estimate a cycle time using the simulator. The functions of interactive debugging aid provided by Windows NT are used for efficient programming of the cooperative or exclusive control. Concurrent processes are simulated step by step. As a result, a program written with the robot-type-independent motion-level commands is translated into a robot language.

The simulator can undertake error checking and error recovery procedures. Possible causes of error include such things as workpieces which are misplaced, slip or are dropped when being carried, collisions during movements, and errors in the original program. A collision may be the result of an unknown obstruction or a programming error. Once the cause of the error has been determined by searching a decision tree, the appropriate remedial action is taken. For unmanned operation, the simulator can be incorporated with sensing devices monitoring the force on the robot joints to detect any faults during runtime operation.

3. FUNCTIONS OF MULTITHREADED SOFTWARE MODULES

For industrial multirobot control systems, especially at the coordinated level, where the various code subsystems need the capability to easily communicate with each other in order to perform one primary activity, a single-process, multithreaded application is considered the best approach. The threads are designed to perform well-designed jobs with simple, clear interfaces. Overall software performs distributed cooperative processing with independent robot control threads.

3.1 Overall Structure of Control Software

In order to split the multirobot application into threads, activities that would benefit from being performed in parallel are identified. For each kind of I/O in the system, a separate thread is designed, so that each I/O thread can block whenever necessary to wait for I/O to complete without having any effect on the responsiveness of the rest of the system. Consequently, for real-time vision-based control, software components such as image processing, task control, robot control, and bidirectional serial communication between the PC and real robots are defined as threads. From the specifications, the overall control software is implemented on a PC-based control architecture using multithreaded programming, written in Visual C++ language with Win32 API functions (Pham and Garg, 1996). The essential structure of the control software is shown in Fig.2. All the code is put in one process, and all the threads share global data and other system resources, making communication between different parts of the system straightforward. Most of the code for each thread is unique to that thread. The functions of the major threads are described as follows.

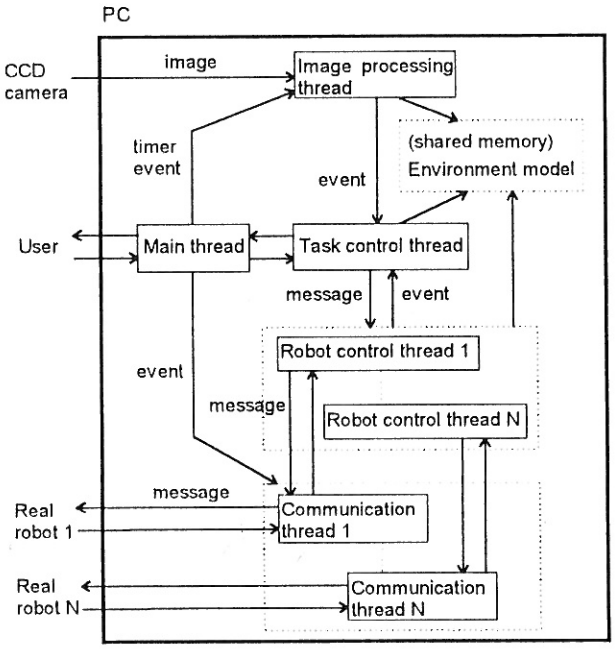

Fig.2. Overall structure of control software using multithreaded programming.

3.2 Image Processing Thread

To operate autonomously, a robot should know the environment in which it is placed, plus the condition of the workpieces and other devices. The structured descriptions of a workpiece in the working area of the robots include the following geometric and physical properties: absolute/relative position and orientation, shape, dimensions, center of gravity, and grasp position. Abstracted parameters are pre-coded and stored in the environment model by off-line programming using a CAD database. When actually activated, sensory data is processed by pattern recognition and feature extraction modules, and

these properties are confirmed, corrected, and modified to accord with the actual environment. The environment model is stored in the shared memory and is accessible from every thread. The image processing thread is activated by the main thread at constant intervals of 400 ms. When it has updated the environment model, the event object is set to the signaled state to wake up the task control thread.

3.3 Task Control Thread

This thread coordinates the runtime system and communicates with the user which inputs task-level programs. As described in Chapter 2, based on a Petri net model of robotic task, the thread generates motion-level commands and sends them to robot control threads by message passing. The environment model includes the positional information of robots, workpieces, and other ancillary equipment, and their labels are used to specify a robotic task with task-level commands.

3.4 Robot Control Thread

Each robot control thread receives motion-level commands from the task control thread and generates primitive robot control commands in the robot language which are sent to the robot controller through the RS-232C interface. By following the motion-level commands, a robot control thread computes the target values of the robot's movements using position data in the environment model. Concurrent execution of program step generation by the task control thread and trajectory generation by a robot control thread enables continuous path generation. Each robot control thread performs its own tasks autonomously as a remote brain without the support of the task control thread.

Modularized sensory information management threads are activated and used by robot control threads as functional units to obtain, abstract, and manage sensory data, since the input and preprocessing of sensory data is performed concurrently with control of robots. Communications between robot control threads and sensory information management threads are required for synchronization in cooperative or exclusive tasks.

3.5 Simulation Thread

In place of the image processing thread, the simulation thread can be activated by the user. To simulate concurrent motions including synchronization and communication, the task control thread and the robot control threads conduct the simulation according to a task specification. Each robot control thread computes its configuration at constant intervals, and stores them in the environment model. The task control thread synchronizes multiple robot control threads on the time axis. The simulator displays the robots whenever a robot completes a motion.

4. IMPLEMENTATION AND PERFORMANCE EVALUATION

To raise system performance with respect to throughput and response time, the following communication methods were evaluated: access to shared memory, use of event objects, and message passing. For coordinating mutually exclusive access to shared memory or other resources in a single-process, multithreaded application, critical section objects are preferably used. A thread waits on a critical section object at the beginning of a critical section of code. When it has finished accessing the shared resource at the end of the critical section, it releases the object. Critical sections of code are kept as short as possible to avoid making other threads wait for a long time.

Event objects are used for synchronizing thread processing. A thread can be in one of three states: running, ready, or blocked. When a thread waits on a nonsignaled event object, the thread blocks until another thread sets the object. A common use of an event object is to notify a thread when a pending I/O operation has been completed. Besides the image processing thread, for example, a sensor thread that processes data acquired by an A/D board might wait on an event object that is set when an A/D driver thread fills a buffer with data. A sensor thread that needs to execute intermittently can request to be put to sleep, placed in a suspended state, for a fixed period of time.

Although threads can communicate through shared global data, robot control threads have a need for interthread communication by message passing because of modularity. It is used for transfer of task-level, motion-level, or robot-dependent commands to directly control the receiving thread by specifying one among several alternatives. Whereas a pipe allows the transmission of streamed data, a message queue allows the transmission of arbitrary data structures from thread to thread. When a thread attempts to write to a full queue or read from an empty queue, it is blocked.

Context switching between threads in the same process involves simply saving the CPU state for the current thread and loading the CPU state for the new thread. Memory management data structures do not need to be changed, because the threads share the same memory areas. Consequently, less work is required for context switching between threads than between processes. However, if the time slice is too small, an unacceptably large percentage of overall processor time will be spent in context switching. If the time slice is too large, responsiveness is deteriorated. By measuring context switching time and application run time using some sort of debugging mechanism, the best time slice value has been determined for the application.

Each communication thread handles the low-level serial I/O (the transmission and reception of individual bytes). The thread priority is set to a higher value than the other threads to ensure that the subsystem is responsive to serial interrupts. The serial interrupt handler signals the event object to wake up the communication thread to process the interrupt. Only one pair of threads uses one serial port, because, if multiple threads read a single serial port, there would be no easy way to guarantee that one thread wouldn't read data intended for another. For the send and receive pipes, critical section objects are used to keep a count of the number of bytes of data in each pipe as a shared global variable.

The advantages of the multithreaded control system are summarized as follows:

(1) Communication using shared memory is the fastest of all interthread communication mechanisms, because transfers occur at memory access speeds. The multirobot application needs the throughput offered by shared memory, in order to utilize the environment model efficiently.
(2) The software can be partially substituted by advanced hardware modules, such as an image processing module and a fast communication module, in order to improve the system performance.
(3) Since each thread is modularized, the control program may be debugged unit by unit. Consequently, software development is simplified.

It is theoretically shown that the required interval between motion commands sent to the robot controller must be longer than the sum of the data transfer time and the processing time on the control computer. So, the less is the processing time of one motion command on the control computer, at the shorter interval robot motion can be controlled (Yasuda and Tachibana, 1994). Synchronization through signal handling between robot agents for simultaneous actions and pick and place operations in response to user command was confirmed using real experiments.

5. CONCLUSIONS

The paper describes the design and implementation of a distributed control and communication structure on a PC-based architecture for industrial robotic applications. The control system has been developed to provide instruction for different robot types using a task-orientated instruction sequence. The proposed control structure is composed of the upper-level task control for cooperative or exclusive control and lower-level robot motion control with a subsumption architecture, both based on the unified Petri net representation. By utilizing the multithreading capabilities provided by Windows NT, supervisory coordination, through event sequencing, is achieved between cooperating robot agents. This relates directly into reduced cycle time for the industrial application. Although, at present, the control system handles tasks which are sequences for simple assembly operations usually involving a workpiece being picked and placed at a target destination, sequences for other tasks can easily be added because of the internal Petri net based control structure.

REFERENCES

Freedman, P. (1991). Time, Petri nets, and robotics. *IEEE Trans. on Robotics and Automation*, 7, 417-433.

Pham, T.Q. and P.K. Garg (1996). *Multithreaded Programming with Windows NT*. Prentice Hall PTR, N.J.

Yasuda, G. and K. Tachibana (1994). A parallel distributed control architecture for multiple robot systems using a network of microcomputers. *Computers & Industrial Engineering*, **27**, 63-66.

Yasuda, G. and K. Tachibana (1996). A computer network based control architecture for autonomous distributed multirobot systems. *Computers & Industrial Engineering*, **31**, 697-702.

Yasuda, G. (1998). A parallel distributed control architecture using communicating process networks for intelligent autonomous robot systems. *Proc. of the 3rd Annual International Conference on Industrial Engineering Theories, Applications and Practice*, pn026.PDF.