Design of a Supervisory Control System for Multiple Robotic Systems[1]

I.H.Suh[2], H.J.Yeo[2], J.H.Kim[2], J.S.Ryoo[2], S.R.Oh[3], C.W.Lee[3] and B.H.Lee[4]

[2] Intelligent Control and Robotics Lab., Dept. of Electronics Eng., Hanyang Univ.,
396 Daehak-dong Ansan-Si, Kyeongki-Do 425-791, Korea
e-mail : ihsuh@shira.hanyang.ac.kr
[3] Advanced Robotics Research Center, KIST, Seoul, Korea
[4] Dept. of Control and Instrumentation Eng., Seoul National Univ., Korea

Abstract

This paper presents a design experience of a supervisory control system for coordination of multiple robotic devices. To effectively program job commands, a Petrinet-type Graphical Robot Language(PGRL) is proposed, where some functions for coordination among tasks, such as concurrency and synchronization, can be easily programmed. And, each task of PGRL is described by employing formal model languages, which are composed of three modules, sensory, data handling, and action module. It is expected that by using our proposed PGRL and formal model languages, one can efficiently describe a job or task, and hence can easily operate a complex real-time concurrent system. The proposed control system has been implemented by using VME-based 32-bit microprocessor boards and a real-time multitasking operating system (VxWorks), and is shown to successfully work for robotic jobs.

1. Introduction

Supervisory controllers being capable of controlling multiple robotic devices within a work cell have been required in many an automated manufacturing area to skillfully handle complicated and dexterous tasks[13]. For this, an universal task or job programming language needs to be designed, and its corresponding interpreter or compiler should be developed together with a task coordinator. Up to now, however, most conventional robot control languages have been developed for the control of a single or dual robots, and are of the text-based type. And they usually provide primarily semantic commands implying specific motions to be difficult to modify[1-4].

Thus, such a type of programming language usually limits a wide applicability due to functional constraints. Especially, its inherent attributes may not fit for the expression of synchronization and parallelism among task-level or low-level actions. On the other hand, another approach has been proposed to model robot systems by a large number of cooperating agents solving a robot task[5].

Even though it is assumed that a robotic job requiring synchronization and parallelism can be programmed by a text-based programming language, there are still a lot of difficulties in understanding and debugging such a program. This implies that a job for a multiple robotic system made up of several manipulators, can not be effectively described by text-based type languages. Therefore, for an effective description of a job for a complex robotic system, it is required to develop a different set of control commands and a grammar for coordinating them. In this paper, we propose a Petrinet-type Graphical Robot Language to be called as PGRL with which one can easily represent parallelism and synchronization for multiple robot control. Petri net has been proven to be a very powerful modeling formalism for representing various aspects of manufacturing systems including plans, schedules, control policies, and performance models. However, in this paper, we employ Petri net as a means of the description of a job requiring multiple robotic devices. And also, to avoid logical errors such as liveness, deadlock, and resource misallocation problem, several error analysis algorithms are designed and implemented. Furthermore, a task scheduler is suggested by using task relation map(TRM) which indicates relations among tasks.

On the other hand, each place of PGRL implies a task level command which is expressed by using the formal model language[7]. For this, four production rules for formal model language are suggested to generate sensory,

[1]This work has been partially supported by KIST2000 Human Robot Program and a Korean National HAN-Project.

Proc. IROS 96
0-7803-3213-X/96/ $ 5.00 © 1996 IEEE

data handling, action and task command modules. If the function of a module can be completed by a single robotic device, the module is called to be a "simple" module. For the case of simple module, a formal model control scheme is designed in such a way that computation is performed in a distributed fashion by a number of concurrent computing agents in a single robotic device-controller under the support of a real time operating system. Otherwise, the module is called to be a "complex" module. For the case of the complex module, a supervisory algorithm to be called as "formal model coordinator" is proposed to allocate each nested module to its corresponding robotic device controller, while monitoring the degree of task completion. To verify the proposed design concept, the control system with the architecture shown in Fig. 1 has been implemented by using 32-bit microprocessor boards(Force CPU30) based on the VME-bus, and a commercialized real-time multitasking operating system, VxWorks. And it is shown that the control system successfully work for a sensor-based robotic job described by our proposed formal model language and PGRL, in which one SCARA robot and one 5-axis articulated robot with 6-axis force/torque sensor are employed.

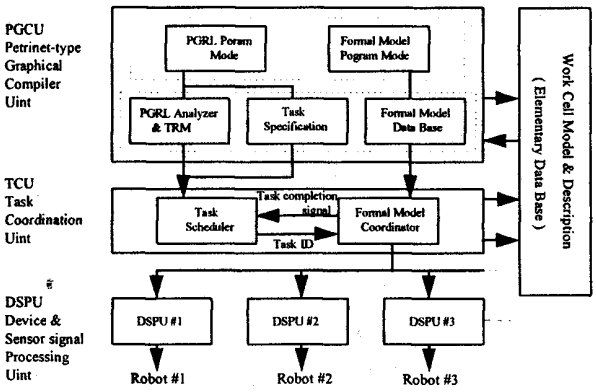


Fig. 1 Overall System Architecture for a Multi-Robot System

2. A Formal Model Language for Task Description

Since a robot system should be operated in a real-time and intelligent behavior is a major goal of robotics[9], it is essential to have a robot task representation which facilitates sensor-based motion programming. For this, a Formal Model concept in [7] is here used for the description of task level robot command to be used in PGRL as a place. In the Formal Model Language, a robot task is represented by using three modules each of which is working as an independent computing agent under a real-time multitasking O.S. And, a module is designed to have the form of port automata with 20 input ports and 20 output ports of integer and double data types such as in Fig. 2 to receive data of the different type and to send the processing results from and to other modules,

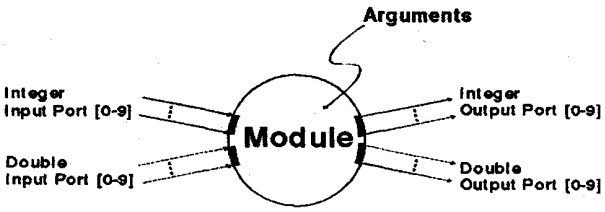


Fig. 2 Basic Module Structure

respectively, at every sampling time. And also, each module has internal variables, called arguments, which are to be specified at the task programming stage.

We denote a module MN as MN(*i1, i2, i3, ...*)(*o1, o2, ...*)(*a1, a2, ...*), where MN is a module name, the *i*'s and *o*'s are input and output port names, and the *a*'s are argument names to be initialized. A type of data is associated with each port. Only ports of the same data type can be connected together. An exemplary module description is given as follows :

```
Module Name : Adder (i_input,d_input) (i_output,d_output)
                    (offset,end_con)
Input Ports : int i_input[10]; double d_input[10];
Output Ports : int i_output[10]; double d_output[10];
   Arguments    : int offset,  end_con;
   Preprocessing : int i=0;
   Behavior     :
   {
        for(i=0;i<10;i++)
             i_output[i] = i_input[i] + offset;
        if(i_output[i]>end_con) End_Condition = TRUE;
   }
```

As shown in the exemplary module, function of a module is determined by preprocessing and behavior sections. A preprocessing section defines internal variable and initializes the function of the module. A behavior section is a specification of a computing behavior to be described in a commercial C language and it is executed until an end condition is satisfied.

Modules are primarily classified into sensory, data handling, and action module according to their type of behaviors. A sensory module takes charge of sensing external environmental states or current states of the system, and it has only output ports to transmit sensory information to a data handling module. An action module has only input ports to receive the data from a data handling module, and takes charge of the actuation of robotic devices. And a data processing module includes

computing algorithms and has both input and output ports.

These modules are put together with each other to build up a task level robot command. We define a task level command formally by specifying connections and communications between sensory and data handling modules, or data handling and action modules using input and output ports. These definitions not only ensure consistency and well-definedness, but also permit an easy sensory-based robot programming.

A new module implying a task command can be made by specifying connections among modules by using square brackets as follows :

New_Task_Command_Module ( )( )(*as*, *ad*, *aa*) =
[Sensory_Module()(*os*)(*as*)||Data_Handling_Module(*id*)(*od*)(*ad*)||Action_Module(*ia*)()(*aa*)],

where partitions of sensory, data handling, and action modules can be made by the symbol "||". In the connections, the output port of a sensory module, *os*, should be connected to the input port of a data handling module, *id*, and the output port of a data handling module, *od*, should be connected to the input port of an action module, *ia*. If a partition includes more than one module, there are two types of connection between two modules. One is for sequential relation and the other for parallel relation. They are denoted as ";" and ":", respectively.

Now, four production rules to generate four type of new modules are proposed as follows :

Rule 1. (Sensory Module Production)

A new sensory module can be produced by specifying connections between a sensory module and a data handling module. For example, a new sensory module, NSM, can be made by connecting SM and DM1 in Fig. 3(a) as

NSM( )(*c1*)(*sn*, *jn*) = [SM( )(*as*, *bs*)(*sn*) ||
DM1(*a1*, *b1*)(*c1*)(*jn*) ||].

It is noted that NSM has no input port and has only output ports which are the same as output ports of DM1.

Rule 2. (Data Handling Module Production)

A new data handling module can be produced by specifying connections among data handling modules. For example, a new data handling module, NDM, can be made by connecting DM1 and DM2 in Fig. 3(b) as

NDM(*a1*, *b1*)(*b2*)(*jn*) = [|| DM1(*a1*,*b1*)(*c1*)(*jn*);
DM2(*a2*)(*b2*)( ) ||].

It is noted that NDM has input ports and output ports, which are the same as input ports of DM1 and output ports of DM2, respectively.

Rule 3. (Action Module Production)

A new action module can be produced by specifying connections between a data handling module and an action module. For example, a new action module, NAM, can be made by connecting DM2 and AM in Fig. 3(c) as

NAM(a2)()(sp) =[|| DM2(*a2*)(*b2*)( )
|| AM(*ca*)( )(*sp*)].

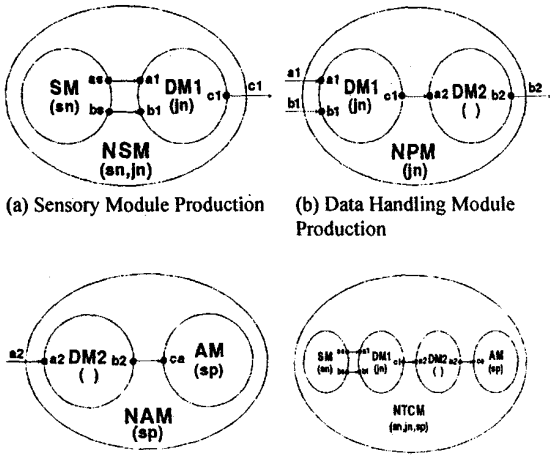
It is noted that NAM has no output port and has only input ports which are the same as input ports of DM2.

Rule 4. (Task Command Module Production)

A new task command module can be produced by specifying connections among sensory, data handling and action modules. For example, a new task command, NTCM, can be made by connecting SM, DM1, DM2 and AM in Fig. 3(d) as

NTCM( )( )(sn, *jn*, *sp*) = [SM( )(*as*, *bs*)(*sn*) ||
DM1(*a1*, *b1*)(*c1*)(*jn*) ; DM2(*a2*)(*b2*)( ) ||
AM(*ca*)( )(*sp*)].

It is noted that NTCM has no input/output port and has only arguments which are used in each module.



(a) Sensory Module Production (b) Data Handling Module Production

(c) Action Module Production (d) Task command module production.

Fig. 3 Production of Sensory, Data Handling, and Action Modules

3. PGRL, PGCU, and TCU

3.1 Petrinet-type Graphical Robot Language (PGRL)

PGRL can be considered as a Petri net. In a Petri net, events are referred to as transitions. In order for a transition to occur, several conditions may have to be satisfied. Information related to these conditions is included in places. Some places are viewed as inputs to a

transition; they are associated with the conditions required for the transition to occur[12]. Other places are viewed as output of a transition ; they are associated with conditions that are affected by the occurrence of this transition. Transition, places, and arcs, which represent certain relationships between them, define the basic components of a Petri net. In PGRL, a place implies a task command for controlling manipulator. One can represent concurrent motions, and preconditions and synchronized motions by using the fork and the join of a Petri net, respectively[9-12]. Therefore, one can easily describe a job of multi-robot by using PGRL, where completion of the job usually needs parallelism, preconditions and synchronization between employed multi-manipulator motions[8][10].

A graphic editor has been designed and implemented in Windows environment to generate places, transitions, and arcs for the PGRL programming by simply clicking graphical icons.

3.2 PGRL Compiler Unit(PGCU)

PGCU receives a graphical task-level program from PGRL editor. PGCU checks whether there are logical errors such as deadlock and resource misallocations, If there are no errors, the graphical symbolic high-level task program is compiled to a map indicating relations among tasks, which is to be used by TCU. If not, errors are displayed on the monitor for the operator to correct them.

For this, PGRL program analyzer is designed to investigate that the given PGRL program include undesirable properties such as deadlock, liveness, and resource misallocation. Specifically, to ensure that a single task is assigned to a robotic device at a time, the number of tokens in a place should be unity. Thus, the analyzer should check if the number of tokens in a place is confined to one. And, liveness should be checked to confirm that all transitions are possibly fired. Deadlock state should be checked to avoid that one task wastefully awaits the resources being used by the other task, and vice versa.

To comply with the requirements of those analyses, two principal Petri net analysis techniques have been utilized[14], where one is the use of reachability tree, and the other is to involve matrix equations. Since the use of matrix equations cannot generate a task sequence to complete a given robotic job, reachability tree is here employed.

The reachability tree can be made by sketching all possible marking states from initial marking state. Since all possible firing sequences can be generated in the tree, use of reachability tree seems to be appropriate for robot programming. Now, to explain how a reachability tree can be constructed, consider the marked Petri net in Fig. 4. The initial marking, M(0), is given as M(0)= (1,0,0,1,1,1,0,0), where M(node number) is defined as M(Node Number) = (number of tokens in place 1, number of tokens in place 2, ... ,number of tokens in place n). In the initial marking, 4 transitions such as T0, T1, T3, and T4 are established on places 1, 4, 5 and 6. To generate entire reachability tree, new marking states are made by firing T0, T1, T3, and T4, when the firing conditions on each transition are met.

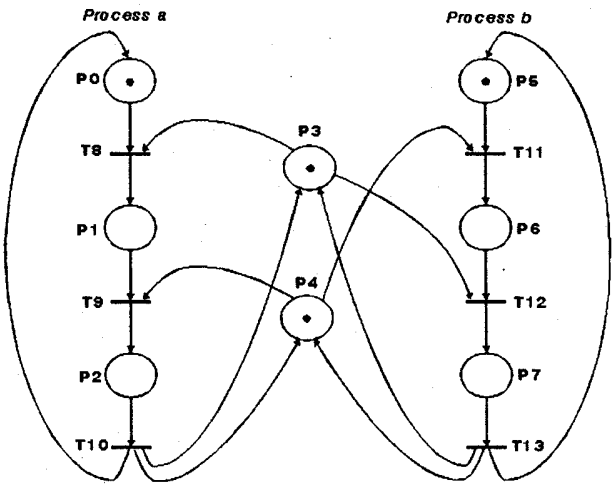


Fig. 4 An Exemplar Petri net being composed of Two Computing Agents

Then, top-down assignment of node numbers is done on every marking state from the left branch to the right, as in Fig. 4. An arc, labeled by the fired transition, leads from the initial marking to each of the new markings which are immediately reachable from the initial marking. Here, when initial marking state reappears in a branch of the tree, new markings are not further generated on the branch. In Fig. 5, notice that the marking M(3) is dead; no transitions are enabled, and thus no new markings are produced in the tree by this dead marking.

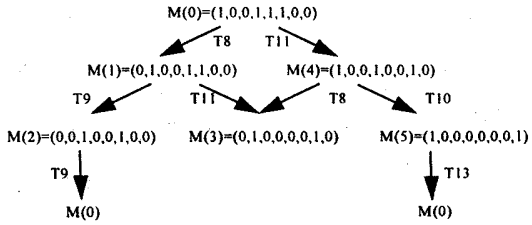


Fig. 5. Reachability Tree for Petri net in Fig. 4

To implement such a reachability tree generator, a map which represent relations between places and transitions is firstly produced. In Table 1, the '1' means that there is an arc from the i-th place, pi to the j-th transition, tj, the '-1' means that there is an arc to pi from tj, and the '0' implies that there are no arcs between pi and tj. Then, reachability-tree can be generated by incorporating TRM as follows;

Table 1 Task(Place) Relation Map (TRM) for the Job shown in Fig. 6

	T1	TB	T3	T4	T5	T6	T7	T8	T9	T10
P1	1	0	0	0	0	0	0	0	0	0
P2	-1	1	0	0	0	0	0	0	0	0
P3	0	-1	1	0	0	0	0	0	0	0
P4	0	0	-1	1	0	0	0	0	0	0
P5	0	0	0	-1	1	0	0	0	0	0
P6	0	0	0	0	-1	1	0	0	0	0
P7	0	0	0	0	0	-1	1	0	0	0
P8	0	0	0	0	0	0	-1	1	0	0
P9	0	0	0	0	0	0	0	-1	1	0
P10	0	0	0	0	0	0	0	0	-1	1
P11	1	0	0	0	0	0	0	0	0	-1
P12	-1	0	0	0	0	0	1	0	0	0
P13	1	0	0	0	0	0	0	-1	0	0
P14	0	1	0	0	0	0	0	0	0	0

Reachability Tree Generation Algorithm

```
n = the number of places
m = the number of transitions

Receive initial marking state, M(0), from PGRL
  ( where, M(0) = (number of tokens in place 1,
        number of tokens in place 2, ... ,number of
        tokens in place n) ).

M = ( child marking );
T = ( parent marking );
RTREE = (single linked list);
child_num = 1;   /* initial number of childs,M[0], = 1 */
M = M(0);
 FOR p = 0 to child_num
   T  =  M[p]
   FOR i = 1 To n
     If( tokens  >  0 )
     THEN {
       FOR j = 1 To m
         If( j-th column in i-th row in TRM[i][j] = 1 )
         THEN{
           possible transitions   = j-th column
           FOR k = 1 to n
            If( k-th row in j-th column in TRM[k][j] = -1 )
            THEN{
              pass a token to k-th place
              iter = iter + 1
              M(iter) = T + (0,0,0, ... ,1k, ... -1i, .... )
              INSERT (M[iter], p, RTREE)
              /* insert M[iter] to RTREE linked list with
              pointer p pointing previous(parent) marking*/
            }
           NEXT
         }
       NEXT
     }
   NEXT
   child_num = iter;
   If( M(iter) = M(0) ) { break; }
NEXT
```

Now, safeness, liveness, and deadlock can be simply checked by using reachability tree as follows; First, by investigating a place having more than 2 tokens from all markings in the tree, safeness can be tested. Second liveness can be checked by investigating that there exists a transition which is not included in any path of all possible paths. Finally, deadlock can be found by checking the existence of a marking state which has not been completely progressed. These can be summarized by the following algorithm;

PGRL Analyzer Algorithm

```
FOR i = 0 TO n        /* Initialze buffer for liveness test */
   buf[i] = 0;        /* n is the number of places */
NEXT
FOR all possible down-paths in RTREE
   FOR all nodes in a path
      FOR i = 0 TO n
        tn = TOKENS(i);   /* returns number of sum of
                             tokens in the i-th place */
        IF ( tn > 1)
        THEN{
             ERROR (" Safeness Error ");
        }
        buf[i] = 1;          /* i-th place in a tree RTREE
                                for liveness test */
      NEXT
   IF ( Last node cannot transit to init node)
   THEN{
       ERROR (" Deadlock Error ");
   }
   NEXT
NEXT
FOR i = 0 TO n
  IF(buf[i] = 0 )
  THEN{
          ERROR (" Liveness Error ");
  }
NEXT
```

3.3 Task Coordination Unit(TCU)

TCU is responsible for the coordination among ta while monitoring degree of completion of each t Specifically, TCU consists of Task Scheduler(TS) Formal model Coordinator(FMC). Firstly, TS ch whether currently active tasks are completed by DSI and determines next places to be executed by using T The number of places to be executed depends conditions of output transitions for currently active pl Then, TS sends the scheduled task to FMC.

task(place) is represented by a simple module, FMC directly send all information on the module to the device controller in charge of the module execution. Otherwise, each nested module of the complex module is allocated to its corresponding DSPU. And then, computation is performed in a distributed fashion by a number of concurrent computing agents in a simple robotic device-controller under the support of a real time operating system. Task scheduling algorithm is here summarized as follows; In the algorithm, TRM[m][n] is defined as m × n task relation matrix for an m unit of place and an n unit of transition.

Task Scheduling Algorithm

```
Integer TRM[m][n] ;        /* 2 dimension array for TRM
                              , where  m:place, n:trasition */
M = ( child marking );
T = ( parent marking );

SEND task command in starting-place
        TO Formal Model Coordinator(FMC)
WHENEVER ( TS receives a signal of task completion
                for place pi from FMC )

DO
  FOR x=1 TO n                  /* x imply parent place */
    IF (TRM[x][i] = 1)
    THEN {
      FOR y=1 TO m              /* y imply y-th transition */
        IF (TRM[x][y]=-1)
        THEN {
          FOR k=1 TO n          /* k imply child place */
            IF (TRM[k][y]=1)
            THEN {
              transmit y-th place, task command
              for the y-th place, to FMC
              M[x][k] = T[x] + 1; /* M is a place have tokens
                                     receiving from each
                                     parent marking      */
            }
          NEXT
        }
      NEXT
    }
    token buffer T[x] = T[x] - 1; /* x is place firing token */
  NEXT
DO_END
```

A data base is made or updated to identify how a module is related to a device controller, whenever a basic module is produced. Thus, FMC can identify a task described by the Formal model language as "simple module" or "complex module" by investigating whether all nested modules are related by one device controller or not. In case of simple module, FMC directly transfer input, data handling, and output modules to the corresponding device. However, in case of complex module, nested modules are classified according to their device controllers and are coordinated by FMC.

Formal Model Coordination Algorithm

```
WHENEVER ( FMC receives a task command from TS )
DO WHILE( there are no complex modules )
  FOR all modules
    IF (  Current module is complex modules )
    THEN{
      DECOMPOSE the module
            into several modules;
      BREAK LOOP;
    }
    IF ( module is sequentially connected with others)
    THEN{
      INSERT BINARY Semaphores between the
      module and the others
    }
  NEXT
  CREATE all modules as parallel processes
  in corresponding DSPU
END
```

4. Experiments

To verify convenience and efficiency of PGRL in describing a job for a multiple robotic system, a grinding job is considered as an example of sensor-based robot controls, where one SCARA robot and one 5-axis articulated robot with 6-axis force/torque sensor are employed. To successfully perform the job, the overall grinding process is represented by PGRL as in Fig. 6. :

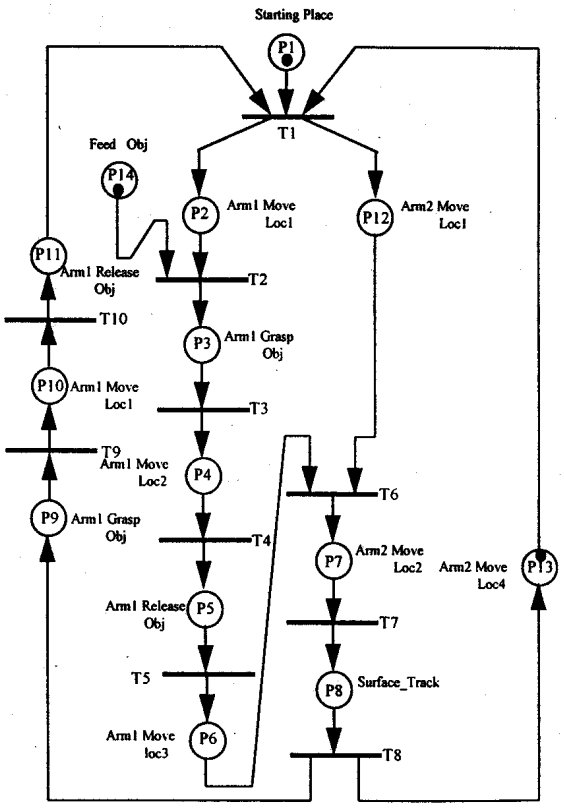


Fig. 6 A Grinding job represented by PGRL

Places and modules in the job are defined as follows;

```
MOVE( )( )(arm1,loc1) = [ R1_encoder()(p1,p2,p3,p4,p5, p6)( )
                        || R1_movl(p1, p2,p3,p4,p5, p6)(dx,dy,dz) (loc1)
                        || R1_act(dx,dy,dz( )( ) ]
FEED( )( )(obj)=[ F1_sense( )(din#1)( )
                        || F1_move(relay1)( ) (obj)
                        || F1_act( )( )( ) ]
GRASP( )( )(arm1)=[ R1_sense()(din#2)( )
                        || R1_grasp( )( )( grasp)
                        || R1_act( )( )( ) ]
RELEASE( )( )(arm1)=[ R1_sense()(din#3)( )
                        || R1_grasp( )( )( arm1)
                        || R1_act( )( )( ) ]
SURFACE_TRACK()()(f_dir, f_N, move_dir, move_dist) =
  [ FORCE()(fx,fy,fz,tx,ty,tz)()
  || TRACK(fx,fy,fz,tx,ty,tz)(d 1,d 2,d 3,d 4,d 5,d 6) (f_dir, f_N)
  || MOVE(d 1,d 2,d 3,d 4,d 5,d 6)( ) (move_dir, move_dist) ]
```

Here, SURFACE_TRACK task is composed of several modules; FORCE, TRACK, and MOVE. And their functional description is described as follows :

FORCE( )(fx,fy,fz,tx,ty,tz)( ); This module is a sensory module which measures 6 axis force/torque, fx, fy, fz, tx, ty, and tz, by using F/T sensor, and then outputs them to its designated output ports after a required processing such as noise filtering.

TRACK(fx,fy,fz,tx,ty,tz)(d1,d2,d3,d4,d5,d6)(f_dir,f_N); This module is a data handling module which takes charge of computing differential joint angle commands for surface tracking.

MOVE(d1,d2,d3,d4,d5,d6)( )(move_dir, move_dist); This module is an action module that drives motors to rotate by joint angle commands received from input ports.

It is remarked that a formal model controller should be designed in each device controller to perform an assigned simple module by creating necessary processes in a real time multi-tasking O.S. For this, every module to be performed in a specific device controller is represented as the following C-codes;

```
MOD_NAME(place_id,ps_sem,csem,i_input,d_input,i_output,d_output)
  int       place_id;  /* place to refer argument in data base          */
  SEM_ID ps_sem;  /* id of semaphore used in process switching   */
  SEM_ID csem;      /* id of semaphore for task completion signal */
  int    i_input[10];          /* integer type input port address     */
  double  d_input[10];       /* double type input port address      */
  int       i_output[10];      /* integer type output port address    */
  double  d_output[10];      /* double type output port address     */
  {
    Preprocessing Section   Specified in a module "MOD_NAME"
       semGive(csem);    /* Completion signal which informs that
                            preparation of execution is ended */
       while(End_Condition != TRUE) {
            semTake(ps_sem,WAIT_FOREVER);
            taskUnlock();
              Behavior Section of specified in   a
              module "MOD_NAME
            taskLock();
            semGive(ps_sem);
       }
  }
```

And, every module is enrolled as a sleeping process in real-time O.S. If a task command is transferred to a formal model controller in a device controller, then nested modules in the task are created by the formal model controller. Each nested module transfers semaphore(csem) signal to formal model controller to report task completion of preprocessing section after completing preprocessing. If the formal model controller receives semaphores from all modules composed of the task, it activates process switchings to allow each nested module to execute works specified in the behavior section, where process switchings are performed by using semaphore(ps_sem). If each nested module receives an semaphore from the formal model controller, the nested module begin to execute its behavior and then returns the semaphore after completion of its behavior.

Formal model controllers are actually implemented by using MC68030 CPU board and real-time O.S. VxWorks. And for a measure of average process switching time, a task composed of 47 modules was chosen. A total process switching time for 47 modules was measured 23 msec, which implies that 0.5 msec switching time-delay for one module in an average sense. Since a single module usually consist of at most four or fire nest modules, about 2.5 msec switching time-delay may be maximally required for a task. But, such a switching time delay cannot be a burden when implementing a robotic device controller with a switching time greater than 20 msec.

All experiments have been successfully completed, where the process switching time is chosen as 2.5 msec in each device control unit. And the sampling time was chosen as 40 msec with which data transmission from the F/T sensor system and vision system to our prototype controller could be successfully performed based on the sensor data.

5. Concluding Remarks

We proposed a supervisory system to effectively coordinate multiple robot devices, where PGRL compiler and task coordination units were designed. And it was shown that a grinding job could be effectively described by the proposed PGRL and formal model languages and be also successfully performed by the proposed supervisory system. It is believed that as far as multiple robotic systems are concerned, our proposed supervisory scheme could be a good alternative to the traditional robot controller.

References

[1] J.E.Agapakis *et. al.*, "Programming and Control of Multiple Robotic Devices in Coordination Motion," *Proc. IEEE Int. Conf. on Robotics and Automation*, vol 1, 362-367, May 1990.

[2] M.A.Arbib, R.O.Eason and R.C.Gonzalez, "Autonomous Robotics Inspection and Mani-pulation Using Multisensor Feedback," *IEEE Trans. on Computer*, 24(4), 17-31, April 1991.

[3] B.Shimano, C.Gescheke, P.Smith - A Robot Programming System Incorporating Realtime and Supervisory Control: VAL-II. In K.Rathmill (Hrsg.), Robotic Assembly, Springer-Verlag, 1985.

[4] C.Blume,W.Jakob -PASRO-Pascal for Robots. Springer-Verlag, 1985.

[5] W.Kalkhoff,"Agent-OrientedRobot Task Transformation," *Proc. IEEE int. Conf on IROS*, 242-247, 1995.

[6] C.J.Paul et.al., "An Intelligent Reactive Monitoring and Scheduling System," *IEEE Control Systems*, 12(3), 78-86, June 1992.

[7] D.M.Lyons and M.A.Arbib, "A Formal Model of Computation for Sensor-Based Robotics," *IEEE Trans. on Robotics and Automation*, 2(3), 280-293, June 1989.

[8] H.Chu and H.A.Elmaraghy, "Integration of Task Planning and Motion Control in a Multi-Robot Assembly Workcell," *J. of Robotics and Computer Integrated Manu-facturing*, 10(3), June 1993.

[9] P.Maigret, "Reactive Planning and Control with Mobile Robots," *IEEE Control Systems*, 12(3), 95-100, June 1992.

[10] E.Rutten, *et.al.*, "A Task-Level Robot Programming Language and its Reactive Execution," *Proc. IEEE Int. Conf. on Robotics and Automation*, 3, 2751-2756, May 1992.

[11] E.D.Adamides and D.Bonvin, "Obtaining Synergetic Behavior by Exploiting Relations in Distributed Robot Plans," *Proc. IEEE Int. Conf. on Robotics and Automation*, 2, 1706-1712, May 1994.

[12] R.Zurawski and M.C.Zhou, "Petri Nets and Industrial Applications," *IEEE Trans. on Industrial Electronics*, 41(6), 567-583, Dec. 1991.

[13] I.H.Suh *et.al.*, "A Control System for Multiple Robot Manipulators ; Design and Implementation," *Proc. of ISRAM '94*, 5, 279-285, August 1994.

[14] J.L.Peterson, Petri net theory and the modeling of systems, PRENTICE-HALL, Englewood Cliffs, N.J., 1981.

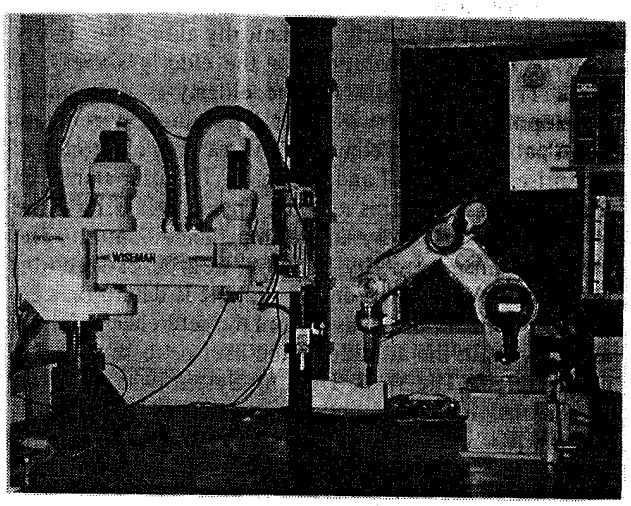


Fig. 7 Photograph of GRINDING Experiment.