The Synchronization of Multiple Manipulators in Kali *

Ajit Nilakantan and Vincent Hayward
McGill Research Center for Intelligent Machines, 3480 University Street, Montréal, Québec, Canada H3A 2A7

This paper presents a strategy in multi-manipulator synchronization that treats the motions as finite state machines. We use the concept of a motion-system as a convenient abstraction for programming explicitly coupled motions. Motions, treated as processes, can communicate/affect one another through the use of control signals and the dynamics of the system are taken into account during the transitions between different motion states. Using examples, we show that such a scheme is general enough to cover diverse situations as two cooperating arms in a multi-manipulator environment, synchronizing motion of the feet of a legged robot for simple gaits and synchronizing the fingers of an anthropomorphic end-effector for simple grasping strategies.

Keywords: Robot programming, Robot control, Multi-manipulator control, Multi-manipulator synchronization.

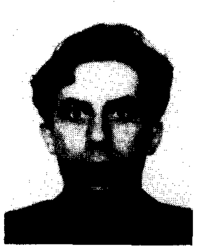
Ajit Nilakantan was born in Bangalore, India, on Sept 20, 1963. He recieved a B. Eng. degree in electrical engineering with a minor in mathematics at McGill in 1985. This was followed by a M. Eng. Degree ('87) which involved the development of a VLSI-based tactile sensor. He then spent a year contributing to the KALI project at McGill. He is currently working with Les Systèmes Cimmetry, an R & D company located in Montréal. His research interests include mathematical methods in manufacturing, image processing and robotics applications.

* Received May 12, 1988; revised January 6, 1989; accepted April 10, 1989. The original version of this paper was presented at the 2nd International Symposium on Robotics and Manufacturing (ISRAM), Albuquerque, New Mexico, November 16–18, 1988. The published proceedings of this meeting may be ordered from: CAD Laboratory for Systems/Robotics, EECE Dept., UNM, Albuquerque, NM 87131, USA.

1. Introduction

The mutual synchronization of the various arm motions and the synchronization of these motions with respect to external events is one major concern in the area of multi-manipulator control. In particular, in order to achieve cooperation in a multi-manipulator environment, one must provide mechanisms that can ensure accurate and predictable time-space rendezvous between the arms and capture these mechanisms into software primitives. These primitives must also provide means of handshaking between the motions of manipulators so that one motion can be predicated by the outcome of another motion.

There is, unfortunately, not a lot of work in this area. One can, however, find an analogy between the multi-robot synchronization problem and that of the synchronization of jobs by a multi-tasking operating system. The latter problem is better understood and more studied. There are well established theories and ample results provided by computer science in the design of operating systems [2].



Vincent Hayward was born in Paris, France, on January 5, 1955. He received the Diplôme d'Ingénieur from Ecole Nationale Supérieure de Mécanique de Nantes, Nantes, France, and the Diplôme de Docteur-Ingénieur from Université de Paris XI at Orsay in computer science, in 1978 and 1981, respectively.
From December 1981 to December 1983, he was at Purdue University, in the Department of Electrical Engineering, the first year as a Visiting Scholar sponsored by CNRS's ARA program (Automatique et Robotique Avancée), and the second year as a Visiting Assistant Professor. There he developed RCCL, a robot control and programming system. He then joined CNRS at the LIMSI laboratory in Orsay, where he worked as Attaché de Recherche on trajectory planning and spatial reasonning until May 1985. He is now Assistant Professor with the Department of Electrical Engineering at McGill University, where he teaches a course on Artificial Intelligence, and a Research Associate with the McGill Research Center for Intelligent Machines. His research interests and publications are in the following areas: robot programming and control, spatial reasoning, computational architectures, and space and remote applications of robotics and telerobotics. Dr. Hayward is member of IEEE.

In general, robot systems have tended to adopt some of the simpler concepts of operating systems; such as notions like critical section protection via atomic events and semaphores [1,8]. While simple semaphore interaction can achieve synchronization in some cases, they would not prove useful in truly cooperative, tightly coupled tasks where the phenomena have to be analyzed at a very fine grain. Semaphores, in effect, attempt to decouple the tasks by blocking out other processes which, paradoxically, opposes the intended aim of having the arms perform tasks *together*.

There is also another very important feature that distinguishes multi-manipulator systems from the multi-tasking operating system metaphor. This is that manipulators are physical *dynamical* systems. That is to say, they "remember" their past. Consequently, *preview* information must be used and taken into account in order to meet time-space constraints.

The dynamics of the system are most readily apparent in the inertia properties of the robots and of their loads. These inertias contribute to delay the response of the system. To offset this delay, one can use sensors which can provide information about the environment with a certain amount of preview. One of the justifications for such an approach is provided in the context of tele-operation. Here, inertia contributes to sluggishness in response that a man–machine interface should account for [3], by means of on-line sensory preview.

However, this type of synchronization is a quite general problem that can be found in a variety of situations (grasping, walking, etc.).

1.1 The Notion of Motion System

Motion systems are convenient abstractions of physical situations where manipulators move together with kinematic and/or force relationships. A motion system consists of a set of manipulators whose kinematic loops share a common "drive transform". We recall here that the drive transform represents a frame transformation introduced into a kinematic loop in order to achieve motion from one position to another. In many cases this frame describes the "tool frame" or controlled frame. Initially, the drive transform contains the "difference" between the position we want to move to and the position we are currently at. Interpolating the drive transform linearly down to unity, for example, will cause a linear motion in cartesian coordinates towards the destination.

The conceptual advantage of a motion system is that it can be viewed as a unit. For example, sending a Sleep signal to a motion system would freeze all the manipulators comprising the motion system; a Go signal would awaken them all simultaneously. Thus synchronization is implicitly achieved between the arms involved in a motion system.

A motion system is specified by giving a set of closed kinematic loops. Each loop contains the coordinate frame of one robot, so that one can solve for the transform representing the robot. The loops may share any number of frames (e.g. common frames of reference), but must share a common drive transform.

By breaking a motion system, one allows the manipulators to move independently, while merging two motion systems causes the arms to move in tandem. The facilities provided by the Go/Sleep/End signals furnish the necessary means to perform these operations. These facilities are provided by a software package called Kali [4], a major redesign of the RCCL system [6].

1.2 Trajectory Generation

In their complete generality, trajectories contain many types of information and must take into account such concerns as the desired position and force set-points, velocities, arrival times, tracking accuracy as well as factors induced by the dynamics of the robot and the joint motors. To simplify the problem somewhat, we make several assumptions about the nature of the trajectory. First of all, it is assumed that during path segments, the main variables of concern are scheduled arrival time or velocity. During the *transition* from one segment to another, because of the change in velocity, we relax the constraints on the velocity and position, allowing the trajectory to wander off the desired path. If we insist on following the path exactly, then the manipulator must be brought to a stop at the transition point, possibly violating timing or acceleration constraints if insufficient lead time is allowed.

These trajectories must be computed with respect to possible time-varying frames (e.g. in tracking moving objects) and we assume that these frames behave in a similar fashion. That is to say, their trajectories consist of segments of approximately constant velocity separated by occasional periods of acceleration.

Finally, the trajectory generation takes into account the dynamics of the system and 'preview' information during the *transition* periods. The time it takes to change velocity from one path segment to a new velocity in the next path segment is determined on line by the dynamics of the robot (including, possibly, a load that could be shared by several manipulators) and limited by the maximum torques that the joint motors can provide in order to meet the requisite acceleration. With preview information, one can initiate the transition before the actual transition point so that, taking into account the dynamically determined transition-time, we can meet and arrive at the through-point without over- or under-shooting. The resulting path is obtained by blending path segments together [5].

2. Trajectory Generator as a Finite State Machine

To achieve synchronization with the above assumptions and constraints, we extent the analogy with the operating system model, going beyond simple semaphore handshaking. Here, we consider a motion as a finite state machine which can change states either through internally generated changes or by externally applied signals. This is really an extension of the crude finite state behavior of semaphores – they, in effect, create two states: Running and Waiting. In many cases, however, it is desirable to have a 'sleeping' state. In this state, the drive transform is frozen, freezing the *relative* motion of the manipulator. That is to say, if the manipulator were approaching a moving object, putting it to sleep would cause the manipulator to maintain the same relative distance from the moving object.

Because of the third state, we decided to do away with semaphores and instead use control signals to affect the state of the manipulator. Such an approach, in addition to greatly increasing the flexibility of the system, fully transforms the behavior of the motion-system into one of a finite state machine.

Motions are requested by the controlling program and sent to the motion generator through a queue. There is one queue assigned to each motion system. In a motion system, at all times, we keep track of three motion requests in its motion queue: the motion that has just been completed (mA), the current motion (mB), and the pending motion (mC). The motion mA, as well as all previous motions, is in the Terminated state and is immune from any further state changes. The pending motion, mC, is in the Considered state and all subsequent motions are in the Queued state. In the normal course of events, the current motion – the motion of interest, mB, is in the Running state. Once we have reached the destination of mB, a state change is triggered internally; mB becomes Terminated and gets shifted back to become mA, mC becomes the current Running motion mB and a new Considered motion mC is popped off the motion queue.

State changes, however, can also be explicitly induced by signals applied by the user. The End signal applied to the current motion mB causes it to terminate prematurely and immediately begin a transition into the next motion mC. A Sleep signal applied to the current motion would cause its drive transform to stay constant and remain that way, in the Sleeping state, until the motion is awakened with the Go signal or prematurely terminated with an End signal.

Note that all state changes go through a transition period whose duration reflects the dynamic manipulability of the motion system. In each state, the drive transform is assumed to have an approximately constant velocity; during the transition period, blending is used to ensure a smooth interpolation between the two velocities. The constraints on the duration of this period of transition are: the dynamics of the manipulators and the load in the motion system (the dynamics comprising the gravitational, velocity and inertial forces) and the maximum torques that the joint motors can supply (in effect the maximum physically achievable accelerations).

According to the desired behavior, the user must also provide a preview factor. With no preview, the transition is initiated at the moment of the state change and the resulting trajectory will overshoot. With preview, one can look ahead and initiate a transition before the expected time of the state change; with a 100% preview factor, the end of the transition period coincides with the time of the state change. Finally, the user can also

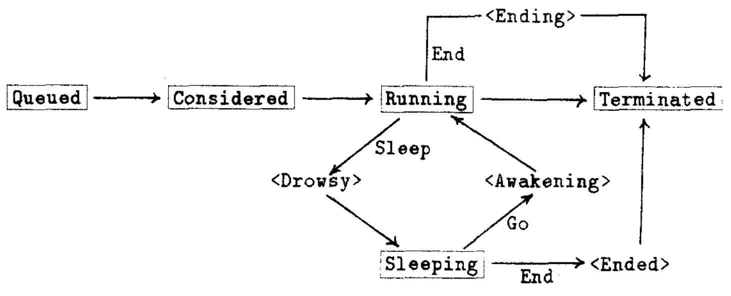


Fig. 1. Finite state model of a motion. Unlabeled arrows denote implicit state changes.

specify the profile of the acceleration, as a flat step, a ramp or something in between which will have an impact on the trade-off between path wander and transition duration.

The finite state model of the motion generator is illustrated on *Fig. 1.*

For instance, the robot can be in the **Running** state. By applying the **Sleep** signal to the motion, it will change state to **Drowsy** and an internally triggered event will cause it to change state to **Sleeping**. This state causes the robot to freeze in mid-flight – the manipulator's drive transform is kept constant so the manipulator will continue to maintain the same relative position with respect to its goal. While the manipulator is **Sleeping**, if a **Go** signal is applied then the motion will proceed to the **Awakening** state which will internally trigger another transition back into the **Running** state from where the manipulator will resume its interrupted motion. On the other hand, applying an **End** signal to a **Sleeping** motion will cause it to change state to **Ended** from where, once again, the **Ended** motion will trigger a transition into the **Terminated** state, causing a new motion request to be popped from the queue

We shall illustrate four possible scenarios, giving the position as a function of time. In *Fig. 2* we have a normal, uninterrupted motion from A to B. If no external signals are applied to the motion, then we can expect it to follow the trajectory

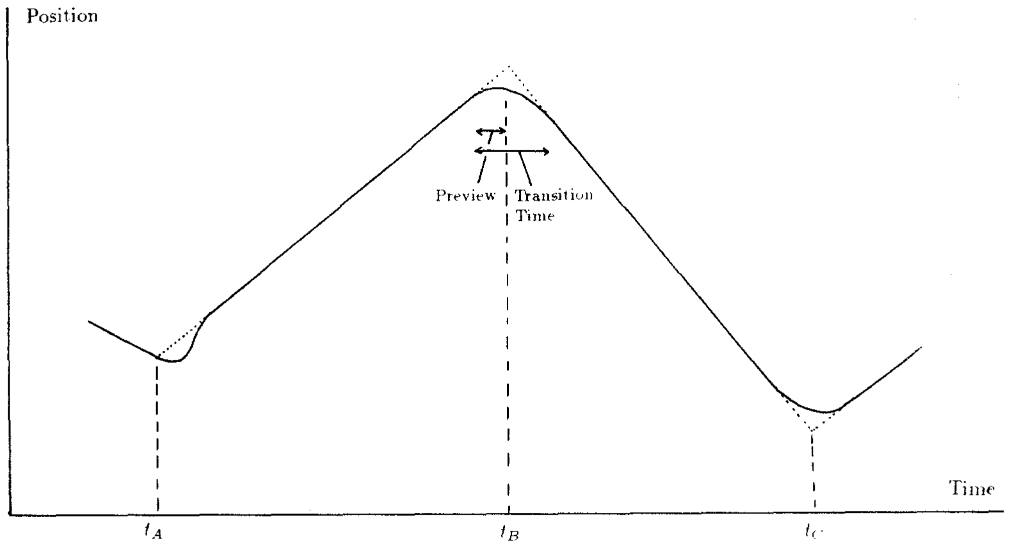


Fig. 2. Uninterrupted motion from A to B, then C.

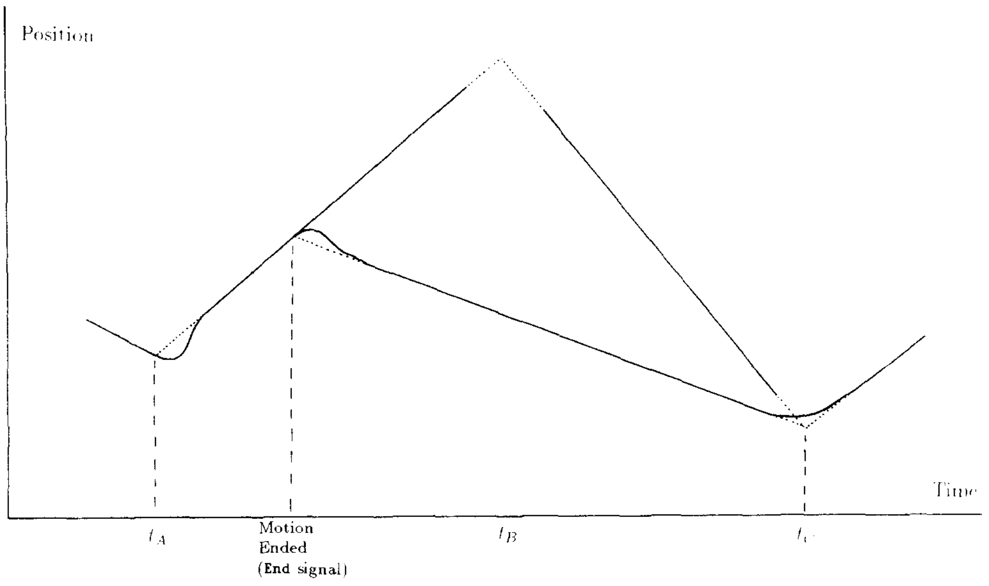


Fig. 3. The motion from *A* to *B* is Ended so the manipulator moves directly to *C*.

shown; during the transition phase between motion *A* and motion *B*, we have a transition time that is computed, based on the dynamics of the motion system and a preview time that is supplied by the user.

In *Fig. 3*, however, we illustrate the case where the End signal is applied to same motion, from *A* to *B*. By applying the End signal to the motion system, the user indicates that she or he wants to forget about moving to *B* and go immediately to position *C*. Thus, the motion now will follow the path indicated by the arrows.

The two next cases both illustrate a Sleep signal being applied to the motion from *A* to *B*. Once the motion is Sleeping, it will remain so indefinitely until it is Awakened with a Go signal or is Ended with an End signal. If a motion is Awakened then it will continue its motion towards *B* as if nothing had happened. If the Sleeping motion is Ended then, as in the case above, the motion towards *B* will be aborted and the motion system will wake up and go directly to *C* (see *Fig. 4*)

Note that in all the state transitions, one computes the transition durations based on the dynamics of the robot and the desired change in velocity, while blending ensures that the transitions are indeed smooth. Thus at run time, the actual scheduling may vary, but the relevant constraints are always enforced. In addition, in every transition, including those induced by Go, Sleep and End signals, the user has control over the preview. This, in effect, means that the user can cause these signals to have a delayed effect. For example, if a robot, equipped with a range sensor, is approaching a table, the user can supply a 100% preview and send a Sleep signal ahead of time so that the robot will fall asleep just as it comes to the level of the table.

Finally, this finite state model of a motion begs comparison with the job control used by Unix [1] where processes can be in states such as Runnable, Stopped, Waiting, Sleeping or Idle, etc. Furthermore, a change of state of a process can be triggered by internal events (such as completion of the job) or by an externally induced events (such as completion of the job) or by an externally induced event (provided by Unix signals). In Unix, all processes can be identified by a unique process id; here the motions all have unique motion id's.

[1] Unix is a trademark of AT & T

3. Software and Hardware Considerations

The first implementation of Kali consists of a multi-processor system described as follows. Five Heurikon single board computers with the Motorola MC 68020 and floating point coprocessor communicate over a common VME backplane via a dual ported shared memory. The development is done in C and programs are created on a SUN workstation under Unix connected to the VME backplane via Ethernet. To off load VME bus traffic, we have selected a system which features the VSB secondary bus. The VSB bus serves to access the shared memory for all asynchronous communications. Further details of the hardware requirements and implementation are presented in [5].

Modern software engineering methods suggest the use of message passing mechanisms to implement complex software real-time application [7].

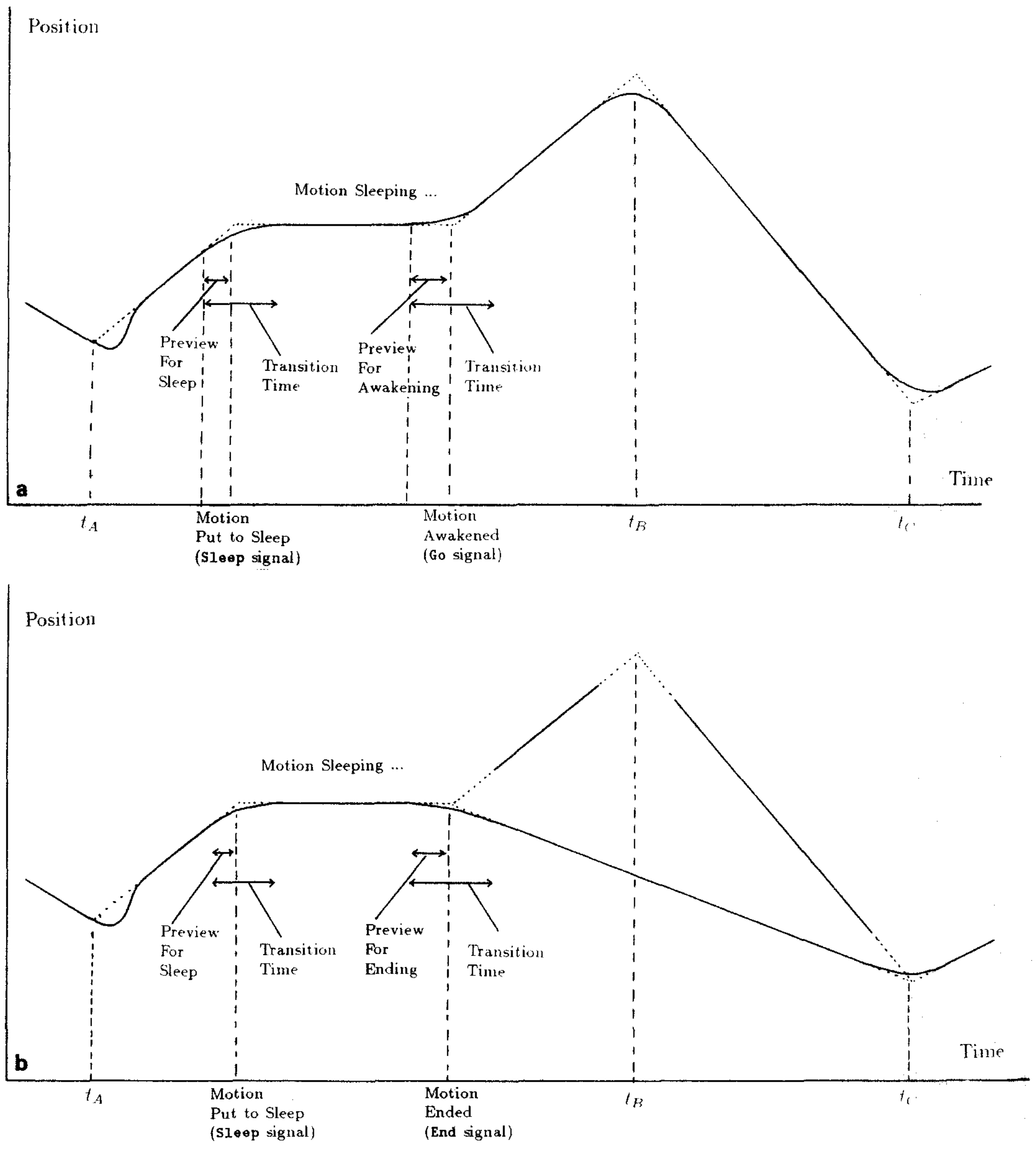


Fig. 4. In the first case, the motion from A to B is put to **Sleep** and, after some time, **Awakened** with a **Go** signal. In the second case, the sleeping motions is applied a **End** signal, the manipulator then proceeds directly toward C.

However, our application has stringent timing requirements – trajectory setpoints are generated at rates varying from 20 to 100 Hz. Thus our option was to implement the entire trajectory generation algorithm, including the synchronization mechanism described here, as a single synchronous process bound to a clock interrupt. Since modern CPU's such as the MC 68020 with floating point coprocessors are capable of computing the trajectories for several "motion systems", we have made no attempt at distributing these computations over several processors, therefore diminishing the communication overhead. In addition, some vendors have announced single board computers five times faster than those we currently use. Since all code is in C this guarantees effortless upgrade in performance. We nonetheless envisage to make a use of true message passing mechanisms in subsequent versions of this project.

The trajectory generation process can handle several motion systems using a sort of "multitasking" analogous to operating systems. Each motion is associated with a "record" which contains the various applicable constraints. Whenever a motion is requested, such a record is created and assigned a unique identification number like a process id. The record is then stored in a circular buffer so it will remain accessible as long as needed. Thus an elementary motion can be thought of, to pursue the analogy, as a computer process with which it is associated. Given the identification number, the application program is able to send control signals to this process to perform the required synchronization. Sending signals is actually implemented by writing entries in the motion records. These entries are defined to be atomic variables at the hardware level. Thus an application program can read the current state of a particular motion without the need for explicit critical section protection.

Although a number of distributed processes are required to run the entire control algorithms, only two are relevant to this discussion: the user process, which contains all directives that make up a "robot program" and runs asynchronously with respect to the trajectory generator, and the trajectory generator, which contains all the algorithms to convert information available from the robot program and from external sensors into setpoints suitable to be tracked by feedback control.

The user process requests motions via the aforementioned records asynchronously with the actual robot motions; therefore it is capable of controlling several simultaneous "threads" of motions as illustrated by the examples in the next section.

4. Examples

In this section, we give various examples that demonstrate the usefulness of this motion synchronization system for a wide number of applications.

4.1. Two Arms

Many useful examples can be found in the synchronization of two manipulators. The easiest way to synchronize two arms is to have them belong to the same motion system; often, however, the arms do not share the same drive transform but one still desires such features as common departure or arrival times for the two independent motions. This is most easily achieved by having motions put themselves to Sleep. While they are asleep the motion states are repeatedly examined to see if the other motion is ready (i.e. is mB). When they both become so, they wake themselves up and can proceed towards the destination with the same departure time. In addition, just before the motions wake themselves up, several other useful functions can be performed. For instance, if we want to enforce a common arrival time, we can assign the same segment time or same arrival time. Going one step further, one can first see which arm is further from the destination, have it move at maximum velocity and let the closer arm have a motion segment time based on the expected time of arrival of the first arm.

Another example considers a situation where one robot passes an object to another robot. One strategy resembles the example shown above; the robots, using Sleep, synchronize themselves to start to move towards each other at the same time with a fixed arrival time. Then, using preview based on the expected time of arrival at the rendezvous point, they End themselves so they meet for an instant and then depart again without overshooting and hitting each other. Another strategy would have the two robots move towards some pre-determined rendezvous point. If one arrives too soon, it would put itself to Sleep and wait for the other arm to arrive. When the transfer of the object is detected, as perhaps, by a force sensor, then, as before, the motions End themselves and a new motion is popped from the queue. See a "Kali" code outline in Appendix A.

4.2. Cyclical Synchronized Motions: Walking

This example involves the control of a statically stable walking robot with six legs. The gait can be described by two motion systems; each comprises three legs so that at any point in time, three legs support the robot in a stable position. The synchronization of the motions involves the following factors: First of all, one constraint is that the feet touches the ground with zero velocity, without any overshoot, to avoid stomping. To obtain that effect, proximity sensors would avoid to have to rely on passive compliance, because preview time would be available. Once three feet have made solid contact with the ground, they should trigger the other three legs to lift up and move forward. Accurate transitions must occur so that the velocity at the end of the transition is zero and the position is at ground level, with no overshoot. Next, once contact has been made with the ground, the other set of three feet lift up and move forward. Meanwhile, the feet touching the ground move *backwards*. If the feet do not slip, this will have the effect of propelling the robot forward. If either motion in the two motion systems completes before the other, then it puts itself to Sleep and waits for the slower one to catch up and End it. Similarly, detecting contact with the ground will trigger one of two signals – if the other set of legs is asleep then this motion will End them as well as itself; otherwise the motion will put itself to sleep and wait.

4.3. The Motion of Fingers

The concept of a motion system can be quite convenient in instances such as controlling a multi-fingered hand. In such cases, using semaphores can be quite a headache but a motion system simply, effectively and explicitly achieves synchronization by having the fingers share their drive transform. Thus a single move request would queue a single motion involving all the fingers. First, a simultaneous move of three fingers towards the three hold points to grasp the object is requested. We assume that the position of the object is not known exactly, so the motion end is predicated upon force sensor readings. The grasped object is then swiveled around the z-axis. An outline of the code to perform a simple task is given in Appendix B.

5. Conclusion

The motion synchronization scheme that we present has several advantages, the chief of which is its generality. The examples shown above indicate the wide variety of tasks that can be handled. The two main ideas introduced here are the concept of a motion system and that of the finite-state machine for robot motions. In addition, to account for the 'real-time dynamism' of the system, we compute transition times between the state changes, based on the the dynamics of the manipulators and loads. Finally, the code that implements this has been written as part of the Kali multi-robot programming and control system, and is presently running in a simulation mode. Work is under way at McGill University to install this code on a multi-processing real-time operating system environment.

Acknowledgements

The ideas described in this paper benefited invaluably of comments from Laeeque Daneshmend, Tony Topper, and John Lloyd.

This work has been made possible by a contract with the Jet Propulsion Laboratory (NASA sub-contract) and by a Grant from NSERC, the Natural Sciences and Engineering Research Council of Canada.

References

[1] R. Alami and H. Chochon, Programming of flexible assembly cells: Task modeling and system integration, *IEEE International Conference on Robotics and Automation*, St. Louis, MO (1985).

[2] F. André, D. Herman and J.-P. Verjus, *Synchronization of Parallel Programs* (MIT Press, 1986).

[3] V. Hayward, Autonomous control issues in a telerobot, *IEEE Conference on Systems Man and Cybernetics*, Peking, China (1988).

[4] V. Hayward and S. Hayati, Kali: An Environment for the Programming and Control of Cooperative Manipulators, *1988 American Control Conference*, Atlanta, GA (1988).

[5] V. Hayward, L. Daneshmend and A. Nilakantan, Model based trajectory planning using preview, *McGill Research Center for Intelligent Machines Technical Report*, CIM-88-9, McGill University, Montréal, Canada (1988). Also a Jet Propulsion Technical Report.

[6] V. Hayward and R.P. Paul, Robot manipulator control under Unix: RCCL a robot control 'C' library, *Int. J. of Robotics Research*, Vol. 5(4) (1987).

[7] W.M. Gentleman, Real-time applications: Multiprocessors in Harmony. *Proceedings of BusComp '88*, New York (1988).

[8] S. Mujtaba and R. Goldman, AL users's manual, AIM-344, Stanford, CA, Stanford University Artificial Intelligence Laboratory, (1981).

Appendix A

```
#include "relevant.h"

main()
{
        Link  *p0, *p1, *rv0, *rv1;      /* pointers to kinematic loops */
        Transform alice,  ralph,         /* The two manipulators        */
        drv_alice, drv_ralph,            /* Their drive transforms      */
        p0_pt, p1_pt,                    /* Two positions               */
        rv_pt;                           /* The rendez-vous point       */
        .
        .
        .
/*
**   Creation of the kinematic loops describing the positions of the two
**   manipulators, Alice and Ralph, at their start positions. Note
**   that they have different drive transforms, so they will move
**   independently.  "mmg" and "cmg" are built-in function for the
**   'Manipulator Motion Generator' and the 'Cartesian Motion
**   Generator'. "NoFun" indicates the transform is not bound to any
**   function.
*/
        p0 = ploop(&alice, mmg, &drv_alice, cmg, &p0_pt, NoFun, NULL);
        p1 = ploop(&ralph, mmg, &drv_ralph, cmg, &p1_pt, NoFun, NULL);
/*
** Assign Alice to MotionSystem #0 and Ralph to MotionSystem #1
** and move to the initial points.
*/
        move(&m0, p0);
        move(&m1, p1);
/*
**   The kinematic loops describing the position at the rendez-vous point.
*/
        rv0 = ploop(&alice, mmg, &drv_alice, cmg, &rv_pt, NoFun, NULL);
        rv1 = ploop(&ralph, mmg, &drv_ralph, cmg, &rv_pt, NoFun, NULL);
/*
**   Move to the rendez-vous point
*/
        move(&m0, rv0);
        move(&m1, rv1);
/*
**   Since the motions are not synchronised (they have different drive
**   transforms), we use the "wait()" function to make sure that
**   the arms do indeed meet at the rendez-vous point. If one arm
**   arrives at the rendez-vous point before the other, it puts itself
**   to sleep and wakes itsel up when the other arm arrives too.
*/
        wait(rv0, rv1);
/*
**   Do other motions, e.g. go back to the original points.
*/
```

```
        move(&m0, p0);
        move(&m1, p1);
        .
        .
        .
}

/*
**  The "wait" function is called from the user level. It adds two
**  motions to each queue and binds the first to the
**  interrupt-level function "_wait()". The function "_wait" does
**  the actual work of synchronising the motions; "wait()" just
**  sets things up for it.
*/

long id0, id1;                /* the id's of the motions to be synchronized */
int state_flag = 0;           /* state flag                                 */

void wait(l0, l1)
Link *l0, *l1;
{
        void _wait();           /* Interrupt level function          */

        m0.specs.control = Go;
        m0.specs.f0 = _wait;    /* Tie the function _wait to the motion. */
        id0 = movereq(&m0);     /* Queue the motion for motion system #0 */

        m1.specs.control = Go;
        m1.specs.f0 = _wait;    /* Do the same for motion system #1      */
        id1 = movereq(&m1);
/*
** Put two motions to act as ''buffers''.
*/
        m0.specs.control = Go;
        movereq(&m0);
        m1.specs.control = Go;
        movereq(&m1);
        state = 1;
}

/*
**  The interrupt level function "_wait". If the motion associated with
**  id0 becomes active (i.e. mB) before the motion associated with id1
**  does, then it puts itself to sleep and wakes itself up when the other
**  motion catches up. And, vice versa.
*/

void _wait(idd0, idd1)
long idd0, idd1;
{
        if (flag == 0) {
                return;                 /* ignore */
        }
```

```
/*
**  Case 1:  Both motions have arrived and are ready!
**          We kill the bogus motions and proceed.
*/
        if (flag && (m0.e.mB->id == id0) && (m1.e.mB->id == id1)
            && (current_system == &m0)) {
                m0.e.mB->control = End;
                m1.e.mB->control = End;
                flag = 0;
                return;
        }
/*
**  Case 2:  Motion System #0 is ahead of time---We put it to Sleep.
*/
        if (flag == 1  &&  m0.e.mB->id == id0  &&  m1.e.mB->id < id1)  {
                m0.e.mB->control = Sleep;
                flag = 2;
                return;
        }
/*
**  Case 3:  Motion System #1 is ahead of time---We put it to Sleep.
*/
        if (flag == 1  &&  m0.e.mB->id < id0  &&  m1.e.mB->id == id1)  {
                m1.e.mB->control = Sleep;
                flag = 2;
                return;
        }
}
```

Appendix B

```
main()
{
        Link *fl1, *fl2, *fl3;
        Transform finger1, finger2, finger3, drive,
                  hold1, hold2, hold3, object;
        Transform rot_increment, rot_decrement;
        long  id;
        int   i;
        extern long move3(); /* this rountine handles a motion with 3 loops */
        .
        .
        .
/*
** Creation of rotation matrices representing 0.1 radians rotations around
** the z-axis.
*/
        rot_increment = rot_to_rotm(0.1, zunit));
        rot_decrement = rot_to_rotm(-0.1, zunit));
        /*
**  Creation of the kinematic loops that describe the position of the fingers
**  when they are grasping the object. Note that they all share the same
**  drive transform.
*/
        fl1 = ploop(&finger1, mmg, &drive, cmg, &hold1, NoFun, &object, NoFun );
        fl2 = ploop(&finger2, mmg, &drive, cmg, &hold2, NoFun, &object, NoFun );
        fl3 = ploop(&finger3, mmg, &drive, cmg, &hold3, NoFun, &object, NoFun );

        kill_on_contact(move3(fl1, fl2, fl3));

        for (i=0; i<10; i++) {
                hold1.r = tr_mult(hold1.r, rot_increment);
                hold2.r = tr_mult(hold2.r, rot_increment);
                hold3.r = tr_mult(hold3.r, rot_increment);
                move3(fl1, fl2, fl3);
        }
        for (i=0; i<10; i++) {
                hold1.r = tr_mult(hold1.r, rot_decrement);
                hold2.r = tr_mult(hold2.r, rot_decrement);
                hold3.r = tr_mult(hold3.r, rot_decrement);
                move3(fl1, fl2, fl3);
        }
        .
        .
        .
}


void kill_on_contact(motion_id)
long motion_id;
{
        extern MotionSystem   *current_system;
        extern Boolean      made_contact();
        static long  id_of_interest;

        if (current_system == NULL) {
        /*
        ** Routine was called from the user level---Remember
        ** the motion id for future reference.
        */
                id_of_interest = motion_id;
        }
        else {
                if (current_motion->mB->id == id_of_interest) {
                /*
                ** Routine was called from interrupt level and we are servicing
                ** the motion we are interested in. If contact (with the object)
                ** is detected then we kill this motion.
                */
                        if (made_contact()) {
                                current_motion->mB->control = End;
                        }
                }
        }
}
```