

## Architecture for a Human-Robot Symbiotic System

Fred W. DePiero, Wayne W. Manges, Reid L. Kress, Mike R. Kedl, William R. Hamel

Telerobotic Systems Section  
Instrumentation and Controls Division  
Oak Ridge National Laboratory  
Oak Ridge, TN 37831-6005

### ABSTRACT

**This paper describes a human-robot symbiont that is under development at Oak Ridge National Laboratory. An overview of the symbiotic system is described that motivates the architecture that has been developed. The architecture is a hierarchical structure that consists of several expert systems which reside above a robot control interface. This interface allows the manipulator to be operated in both a teleoperated and autonomous mode. All these processes coexist with the lowest level of the hierarchy, which is a numerically intensive control algorithm. The architecture is implemented on five processors in a coarsely parallel system.**

### INTRODUCTION

This architecture was developed for the Man-Robot Symbiosis project at ORNL. The symbiont itself has two components, or resources. The human operator is one resource. He operates in the symbiont's task space via teleoperation. The other resource is the robot operating in an autonomous mode. Each resource provides unique contributions to the system, and this produces the symbiotic effect. This effect is symbiotic because the overall system capability is greater than that of either resource alone. During teleoperation, the system benefits from the human operator's ability to respond to unexpected events. During autonomous operation the operator benefits from the robot's tireless precision during repetitious work.

Another facet of the symbiotic relationship called for a craftsman-apprentice association between the two resources. This was intended to allow the capabilities of the autonomous mode to grow as the system experiences more examples of teleoperated subtasks.

A task was needed for the symbiont to perform for demonstration purposes. The "Cranfield Assembly Benchmark"[1] was chosen. This task is fairly representative of one-handed assemblies. It involves part placement into a jig and pin insertion.

### SYSTEM OVERVIEW

The system first plans the steps required to assemble the Cranfield using a linear job planner. See Figure 1. The Job Planner[2] uses a set of preconditions and postconditions for each assembly step in order to produce a sequence of subtask primitives. These primitives are expressed in a high level of description.

Grasp(Lever) and  
Move\_Arm(Lever:Hover\_pos)

are examples of these primitives. This assembly plan is displayed to the operator for approval. The steps are then assigned for execution in either an autonomous or teleoperated mode by the Dynamic Task Allocator [3]. This process uses one of two user selectable allocation policies. Either an allocation which minimizes time of assembly or one that maximizes the anticipated success of each step is used. Both the teleoperating human and the autonomous robot have ratings for each primitive. These ratings vary dynamically as the assembly is performed. This allows the allocation of a particular primitive to change as the abilities of each resource improve or degrade. The allocation of each step is displayed to the operator, who has the option to either change or approve it. Once the allocation of each primitive is determined, the assembly steps are performed.

A primitive that is executed via teleoperation is displayed to the operator, who then may use a six degree-of-freedom (DOF) force-reflecting Kraft master to complete the assembly step [4]. During teleoperation, trajectory commands for the slave are produced from forward kinematic calculations on the master. The difference from the master's previous position is taken, applied to a low-pass filter, and sent to the arm as a velocity request for that iteration. Indexing and motion scaling are provided. The master's grip also incorporates a control for the slave's end effector tong. The slave position is fed back to the master along with force/torque data from a sensor mounted on the slave's wrist. These data are used in force feedback calculations.

A primitive that is to be executed autonomously is sent to a process known as the Intelligent Controller. The command is decomposed via simple string comparisons, and its syntax is checked. Some state information on the position of the manipulator

is retained so that the logical sequence of commands may be verified. For example, the arm must be above an object before that object can be grasped. The Intelligent Controller can request arm movement by specifying the endpoints of a path segment. These endpoints are passed to the robot position control algorithm.

The robot arm chosen for the project was the CESARm. This is a seven DOF manipulator with a spherical wrist. It is approximately three times the size of a human arm and has a maximum tip speed of 120 inches per second. Six DOF are required to arbitrarily position and orient a manipulator in space. The redundancy introduced by the CESARm's extra DOF is resolved, and its configuration is optimized by a gradient projection method [5]. This is a numerically intensive algorithm. It involves first finding a particular solution of joint angle velocities that will satisfy the requested end-effector motion. This solution is projected into the Null space to find the component of the velocity vector which does not contribute to end-effector motion. This Null space component is then subtracted from the particular solution to find the set of joint angle velocities having a minimum Euclidean norm. The configuration of the arm is optimized by using a cost function of joint angles. A typical cost function would penalize joint motion near joint limits. The gradient of this cost function is projected into the Null space, scaled, and added to the least norm solution. The cost is minimized with Null space motion only; in this way, the optimization component does not alter the desired motion of the end effector. This technique is executed at a 100Hz rate and is applied in each mode of operation.

While manipulation is being performed, several sensors are actively updating a sensor database. These include the force/torque sensor mentioned above and a tactile sensor that is mounted within the gripper. Arm position and other manipulator data are also fed into this database. A process called the Automated Monitor [2] is supplied with expectations of sensor conditions that should be true before, during, and after each move. Any abnormalities are flagged as exceptions to the operator. During autonomous execution, the Intelligent Controller also checks for abnormal conditions.

Upon completion of each step in the assembly, the operator is queried to find his opinion on the success or failure of the step. These data are used along with an evaluation provided by the Automated Monitor to determine the proper update of object position data that results after each step. Failed steps in the assembly cause a reallocation and sometimes a replan.

To provide the capability of learning subtasks, an effort was made to analyze sensor data during teleoperated moves. This process was known as the Learning System. Research in this area has yielded

an algorithm capable of very compactly expressing common traits of sensor data. This capability was considered to be a first step towards recognizing a teleoperated subassembly.

A separate obstacle detection routine runs whenever the arm moves, regardless of the control mode. This was implemented as a safety feature. It was thought that these computations should be totally separate from any part of the control scheme so that experimentation with new control algorithms would never interfere with this safeguard. In these computations the arm is represented as a stick figure surrounded by legal and illegal volumes. Entry into an illegal region sets an error condition and causes motion to stop. The isolation of the safety software meant that separate forward kinematic calculations had to be performed. This was computationally intensive; thus the calculations were interleaved so as to complete a full check at a 12Hz rate.

### HARDWARE ARCHITECTURE

This system has been implemented with coarsely parallel processors. The final configuration of this system will consist of four Motorola 133 processors, each with a 68020 CPU and 68881 FPU. The processors use the OS/9 (TM) operating system developed by Microware. This processor-OS combination has proved to be a high-performance, general-purpose platform for real-time systems. The processor's versatility has been advantageous because of the frequently changing requirements of the system. The fifth processor in the system is a separate Macintosh II computer which has been used to create a graphical Human-Machine Interface (HMI).

The Motorola processors reside on a VME bus, which provides the backplane connection required for various I/O devices. The VME bus is well standardized and supports a great many off-the-shelf I/O boards. The system uses parallel I/O cards for the CESARm's brakes, encoders, and safety interlocks, and also for the force/torque sensor. A D/A card is used to drive the pulse-width-modulated amplifiers for the actuators. Serial connections are made to the tactile sensor and to the Macintosh.

The hardware architecture of this system is somewhat similar to the CONDOR [6]. However, this system does not rely on a Sun host. Also, the 68020 control processors each have a full multitasking, real-time kernel on board. Having the full kernel on each processor has added flexibility to the system. Its scheduler permits some processors to multitask, and its I/O system supports interprocessor communication.

The symbiont is being implemented in C, and the OS/9 development environment includes both source and register level debuggers. An Ethernet link is also available for OS/9 systems that supports code development on a Sun and includes a cross compiler.

Having systems without an intimate reliance on a Sun-UNIX host has proved to be beneficial on past projects. These scaled down systems have been used as both a target and a development system and require fewer changes when producing the final embedded system. The OS is also ROMable, which has permitted the development of very compact and rugged diskless systems.

### SOFTWARE ARCHITECTURE

Due to the system's complexity and to the fact that several developers were to be working concurrently, it was readily apparent that the system would have to be developed as a set of independent processes. (The system currently consists of over 10,000 lines of code with approximately 6 man-years of effort.) This approach allowed more independent code development. It also provided increased flexibility because the processes could be moved onto separate CPUs, as dictated by performance and memory requirements; however, this also meant that a communication mechanism was required that was capable of spanning processor boundaries.

#### Off-line Communication and Control

A communication network of pipes was created along with a library of routines that allowed each process to link up and access the network. Direct process-to-process connections are provided with the capability of expanding to a fully interconnected network. These messages are used to exchange commands and data between the offline processes. Each message consists of a five byte header followed by a data portion of arbitrary length. The header contains the message's destination, its origin, a type, and the total message length. The origin field simplified the sending of messages to the HMI. All HMI message I/O is required to pass through the same serial link; thus, the origin field made it possible to differentiate the source of each message that arrived at the HMI through this same communication link.

The interconnections of the network have been defined in a common file. This allowed for easy expansion as more processes came on line. Each process that wishes to exchange messages calls a routine to initialize itself onto the network. This routine accesses the common file and finds all neighbors of the local task. The routine then opens the proper pipe to each of its neighbors and exchanges salutations. Routines are provided to send and receive messages from any specified process. Another function allows a process to sleep until a message arrives from one of its neighbors. This was important for several of the processes which multitask on the same CPU.

A convention was established that required each process on the network to respond to a KILL message. This was useful when the system shut down because it cleaned up the processes on each CPU.

The message-passing scheme worked quite well. Once implemented, it provided a very clean mechanism for exchanging data between processes. It was also a helpful debugging tool. A message generator/analyzer was created that could emulate messages from neighboring processes and formats the data portion of messages for convenient display. Another debugging option was also provided that dumped all selected messages to either a terminal or a file during run time. This was helpful for debugging interprocess synchronization.

The hierarchy of the symbiont contains components similar to NIST's proposed standard architecture, RCS [7]. However, the style of message passing implemented for the symbiont makes this system differ from the RCS type. Here, messages are only exchanged at times when new data must be transmitted. The RCS standard calls for each process in the system to handshake with its neighbors in the hierarchy at a regular rate, regardless of the need for communication. The processes here do not suffer from this raised communication overhead. Some processes do take on the responsibility of checking for messages or alert flags which may arrive while the processes are busy. These additional checks make sense for some symbiont processes but not for others. Hence, the concept of a fixed rate handshake was not implemented as a system-wide mandate. For example, CESARm control processes check common memory flags for alert conditions that may have been discovered by other processes. However, a process such as the Job Planner does not perform this sort of handshaking while it is working on a job plan.

#### On-line Communication and Control

The data rate through a pipe between two processes which reside on the same CPU is approximately 70us per byte or better, depending on the packet length. Although this data rate was satisfactory for coordinating the activities of offline processes, it was too slow for exchanging data between the tasks responsible for real-time control.

Unfortunately, this processor-OS combination supported few options for *interprocessor* communication and synchronization, other than pipes. Actually this is due to a hardware limitation. The Motorola 133 68020 processors cannot generate mailbox interrupts. The OS/9 solution to this was to have polling tasks on each CPU that access shared memory in order to implement a network across the VME backplane. The timing requirements of the run-time control processes, however, are too stringent to tolerate the performance of these backplane pipes. Hence, the decision was made to let the online control processes poll common memory in order to communicate and synchronize their activities.

#### Database Communication

The data storage and retrieval needs of the various portions of the system placed opposing requirements on the system architecture. The Automated Monitor

and Intelligent Controller both required high-speed access to sensor data during online control. Also, this sensor data was updated very rapidly compared to the rate at which it was read. Hence, it could be read asynchronously with respect to its writing without the use of a semaphore. These needs contrast with the needs of the Task Allocator and with other needs of the Intelligent Controller which both require access to object position data. First of all, these two processes only need access to this data during offline computations. Second, it was felt necessary to implement an exclusive access scheme for object position data, just in case one process attempted to update positions from a previous move while another attempted to access positions during the planning of the next move. The needs of accessing object data and accessing sensor data also differ in another way. The number of objects stored in the database was permitted to change, and hence the memory storage location of each object was not necessarily fixed. In contrast, the number of sensors was always fixed for a particular run, and the location of sensor data could easily be fixed also.

At first a common solution to the data access problem was considered, but no scheme seemed satisfactory in meeting the opposing requirements. It was then decided to split the real-time sensor data and manipulator position data into a separate database from the object position data.

A database controller called the Knowledge Manager stores each object's current position, storage position, and assembly position. The need for the system to deal with new objects during some future task was anticipated; therefore, this database not only had to search efficiently but also had to perform rapid insertions and deletions. AVL trees were chosen for this reason [8]. These are binary trees that maintain an optimum balance in their branches regardless of the order in which incoming data are inserted. Various portions of the system must access object data by specifying different key fields of the object. Some processes specify objects by name, others by a cartesian position, and some by another coarser position description known as a bin number. To facilitate these differing search needs, the object data was ordered by three AVL trees, each of which operate on a different field.

It was realized that circumstances might arise which require the exclusive access of the object database to extend beyond a single update message sent to the Knowledge Manager. For example, these update requirements occurred when a subassembly was repositioned. In this event each object in the subassembly had to be updated in the database. Rather than complicate the structure of a message, these logically indivisible update messages were preceded by a LOCK message. The LOCK message meant that the Knowledge Manager had to stop responding to messages from its other neighbors and fix its attention on the LOCK message sender. Once the Knowledge Manager's attention was "locked"

onto another process, that process was free to read and write object data as needed. An UNLOCK command was, of course, also provided.

#### Process Partitioning

The real-time control algorithm for the CESARm has been partitioned into three processes that each reside on their own CPU. The expert systems all multitask on one processor. An outline of the partitioning of computations is given below. The letters indicate that the process is active during either the (A) autonomous mode, (T) teleoperated mode, or (B) both. The letter (O) designates a process having offline activity only.

#### CPU1:

- 1) CESARm's PID (B)
- 2) CESARm's Cable, joint, and velocity limit checks (B)
- 3) CESARm's path segment generation (A)
- 4) CESARm's forward kinematics, for force reflection in master (T)
- 5) CESARm's I/O for sensors and actuators (B)
- 6) Intelligent Controller (A,O)
- 7) Knowledge Manager (B,O)

#### CPU2:

- 1) CESARm's redundancy resolution and optimization (B)
- 2) CESARm's obstacle detection (B)

#### CPU3:

- 1) Master's forward kinematics (T)
- 2) Master's Inverse kinematics (T)
- 3) Master's I/O (T)

#### CPU4:

- 1) Job Planner (B,O)
- 2) Dynamic Task Allocator (B,O)
- 3) Automated Monitor (T,O)

#### Macintosh:

- 1) Human-Machine Interface (B,O)

The partitioning was not designed to exactly divide the system's computations between each processor. Actually, the processes listed above do not lend themselves particularly well to fine-grained parallel computation. Given the choice of a coarsely parallel system, it was decided to pack CPU2 and CPU3 with the processes that were reasonably stable in their content. CPU1 was left as underutilized as possible. This permitted quicker alterations to the system because it allowed CPU1 to accommodate any new tasks as they were identified while minimizing the chore of repartitioning. CPU2 requires 90% of the 0.01 sec. loop time to complete its computations. CPU3 runs flat out and can still only manage a loop rate of 60Hz. The bandwidth of the manufacturer's serial communication link to the master was a limiting factor here.

#### CURRENT STATE OF THE SYSTEM

To date, the system has not been tested in a fully integrated form. The Job Planner, Dynamic Task allocator, Human Interface, and Automated Monitor have all operated together with simulated sensor data. This configuration consisted of a Macintosh II that ran the graphical HMI and a single Motorola 68020 CPU that multitasked the other processes in the OS/9 environment. The CESARm tasks have been demonstrated in their partitioned form and are still undergoing development. The Learning System has demonstrated its ability to characterize simulated sensor data. The Intelligent Controller and Knowledge Manager have also been operated together successfully in a multitasking mode.

### CONCLUSION

Despite the absence of a fully integrated test to date, we feel that the Human-Robot Symbiosis concept is valid. It represents a cost-effective approach to robotic systems which must have robust operation in the presence of unexpected events. The VME-OS/9 type of system has proved to be a good platform for both the real-time processes and the expert systems. The message-passing scheme not only provided a clean communication interface but also became a useful debugging tool.

### ACKNOWLEDGEMENTS

This work was sponsored by the Engineering Research Program of the Office of Basic Energy Science, of the U.S. Department of Energy, under contract No. DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc. Other significant contributors to this effort include Bill Hamel and Chuck Weisbin, the project originators. Lynne Parker developed the Job Planner, Dynamic Task Allocator, and Automated Monitor. Ed Oblow developed the learning methodology. Carl Steidley, from Central Washington University, worked on portions of the Automated Monitor and Knowledge Manager. Susan Hruska, from Florida State University, worked on portions of the Dynamic Task Allocator, and Rick Glassell implemented the first version of the CESARm control algorithm.

### REFERENCES

- [1] K. Collins, A. M. Plamer, and K. Rathmill, "The Development of A European Benchmark for the Comparison of Assembly Robot Programming Systems," *Proceedings, June 1984 Robots Europe Conference*, Brussels, Springer-Verlag, 1985.
- [2] L. E. Parker, *Job Planning and Execution Monitoring for a Human-Robot Symbiotic System*, ORNL/TM-11308, Oak Ridge National Laboratory, November 1989.
- [3] L. E. Parker and F. G. Pin, *Dynamic Task Allocation for a Man-Machine Symbiotic System*,

ORNL/TM-10397, Oak Ridge National Laboratory, June 1987.

[4] J. F. Jansen and S. M. Babcock, *Bilateral Force Reflection for Teleoperators with Masters and Slaves with Dissimilar and Possibly Redundant Kinematics*, ORNL/TM-11326, Oak Ridge National Laboratory, November 1989.

[5] R. V. Dubey, J. A. Eular and S. M. Babcock, "An Efficient Gradient Projection Optimization Scheme for a Seven-Degree-of-Freedom Redundant Robot with Spherical Wrist," *Proceedings of the 1988 IEEE International Conference on Robotics and Automation*.

[6] S. Narasimhan, D. M. Siegel and J. M. Hollerbach, "CONDOR: An Architecture for Controlling the Utah-MIT Dextrous Hand," *IEEE Transactions on Robotics and Automation* vol. 5, no. 5 (1989).

[7] L. S. Haynes, et. al., "An Application Example of the NBS Robot Control System," *Robotics and Computer-Integrated Manufacturing* vol. 1, no. 1, pp. 81-95, 1984.

[8] R. Sedgewick, *Algorithms*, 2nd. ed., Addison-Wesley, Reading, Mass., 1988.