RSD-TR-10-84

A MODULE ARCHITECTURE FOR AN INTEGRATED MULTI-ROBOT SYSTEM

Kang G. Shin

Mark E. Epstein

R. A. Volz

Department of Electrical Engineering and Computer Science

The University of Michigan

Ann Arbor, Michigan 48109-1109

July 1984

CENTER FOR ROBOTICS AND INTEGRATED MANUFACTURING

Robot Systems Division

COLLEGE OF ENGINEERING

THE UNIVERSITY OF MICHIGAN

ANN ARBOR, MICHIGAN 48109-1109

A MODULE ARCHITECTURE FOR AN INTEGRATED MULTI-ROBOT SYSTEM[1]

ABSTRACT

An integrated multi-robot system (IMRS) consists of two or more robots, machinery and sensors. Industrial processes can be categorized into five classes; independent, loosely-, tightly-, work-coupled, or serialized motion processes.

In this report, we present first a formulation of new IMRS communication and coordination concepts. Then, we propose an IMRS module architecture which can execute each of these five process classes with efficiency, flexibility, and reliability. The modules within this architecture interact in a decentralized and/or centralized manner, depending on the nature of applications. This is in a sharp contrast to most existing multi-robot systems (MRS's) which utilize centralized control *only*. The purpose of this new architecture is to (i) improve performance by employing the inherent parallelism in IMRS processes, (ii) make the IMRS more fault-tolerant, and (iii) provide the user with more flexibility to facilitate a wider spectrum of applications.

[1]The work reported here is supported in part by the U.S. AFOSR Contract No. F49620-82-C-0089 and Robot Systems Division, Center for Robotics and Integrated Manufacturing (CRIM), The University of Michigan, Ann Arbor, Michigan. All correspondence should be addressed to Prof. Kang G. Shin at the above address.

TABLE OF CONTENTS

1. INTRODUCTION

Conventionally, multi-robot systems (MRS's) are all centrally controlled; that is, control tasks for an MRS may be distributed over a network of processors or reside in a uniprocessor but are all executed under directives of one central task.[2] Although all of the five process classes to be discussed in Section 3 can be accomplished using a central controller, communications bottlenecking and unreliability (that occurs at the central controller) become major problems. For this reason we need to develop a new architecture which can accommodate both centralized and decentralized controls. To differentiate this new architecture from the conventional MRS, define an *integrated multi-robot system* (IMRS) as a collection of two or more robots, sensors, and other computer controlled machinery, such that

- each robot is controlled by its own set of dedicated tasks, which communicate to allow synchronization and concurrency between robot processes,[3]
- the tasks are executing in true parallelism,
- *both* centralized and decentralized control concepts are used, and
- tasks may be used for controlling other machinery, sensor I/O processing, communication handling, or just plain computations.

We make no assumptions concerning the computers and robots used for an IMRS.

In this report we present a high-level architecture for an IMRS that explores and integrates both the centralized and the decentralized control concepts. Emphasis is placed on flexibility to facilitate a wide spectrum of manufacturing applications.

---

[2] This is based on (i) several multi-robot systems that were recently displayed at the Robots 8 Exposition in Detroit and (ii) works published in open literature.

[3] "Process" will be used to denote the industrial output of the IMRS, which is accomplished by a set of "tasks" executing on one or more processors.

improved productivity by exploiting the inherent parallelism, and reliability/graceful degradation to provide non-stop operation of the IMRS.

The general IMRS would encompass many areas of research, some of which include sensors, collision and obstacle avoidance, artificial intelligence, and communications and concurrency. One of the most important steps is the design of the communication system that links the tasks of an IMRS. The design includes the high-level abstractions (we term this here the ***module architecture***), as well as the low-level communication primitives. In the rest of this report, we will develop a modular architecture by taking a top-down approach; (i) examining the problems we wish to solve, (ii) investigating how unreliability, unflexibility, and communications bottlenecking can occur while trying to solve these problems, and then (iii) designing an architecture capable of handling these problems elegantly. We will not discuss issues concerning the actual primitives needed for the intertask communications. For this the reader is referred to [1], [3], [5], [19], and [20].

There are numerous robot languages designed (see [2] for a survey) which can control more than one robot simultaneously, the most advanced being AL[14]. AL allows one program (and hence one task) to control two robots at once. By using **Cobegin-Coend** pairs, a programmer can initiate two pseudo-concurrent tasks. They can be synchronized using the *EVENT* data type (integer semaphores). The principal motive behind this design was to allow cooperation via ***serializing*** each robot's motions by using *EVENTS*. This restricts the potential amount of parallelism that can be attained. It would be more efficient to let each robot process run under the control of its own tasks, with synchronization (or rendezvous) at designated points in the programs.

Some work has been done on distributed industrial process control[18], but the results are not easily transportable to an IMRS. Steusloff[18] has described a distributed, fault-tolerant system used for controlling soaking pit furnaces. The furnace system is controlled by a real-time concurrent language called "Multicomputer PEARL." PEARL allows the transmission of information from one task to another by message passing and remote procedure calls. Each furnace is controlled by its own microcomputer system, and the microcomputer systems are logically *paired* so should one system fail, the corresponding mate computer system would control two furnaces. This system is reported to be highly fault-tolerant, having only 11 hours of down time in more than 24,000 hours of use. This figure is indeed impressive, but the classes of parallelism involved in the furnace application are far less complex than the classes of parallelism needed in an IMRS. The action of one IMRS process could completely alter the action of another IMRS process, or robots might have to work on one common process, requiring tightly coupled communication and synchronization. Obviously an IMRS needs a more intricate communications structure to handle this more dynamic environment.

There has been considerably more research in the area of communicating concurrent tasks. Numerous languages have been designed which contain primitives that allow tasks to synchronize and communicate via various techniques. We will discuss the issues and ideal communication primitives for an IMRS in a future report. For some actual concurrent languages, see [3], [8], [9], [10], [17], [18], [20], and [21].

Our design approach consists of several distinct phases, beginning from the generic nature of an IMRS to the design and evaluation of the communications system based on process classes. The organization of this report conforms to these phases except for the low-level communication primitives and their evaluation, which is not included due

to the limited size of this report. Section 2 describes the generic nature of an IMRS such as motivations, and design goals. In Section 3 we identify types of parallelism in an IMRS. In Section 4 we propose a modular architecture to facilitate such parallelism. Section 5 concludes the report with a discussion of the remaining work to be done for IMRS's.

2. GENERIC ASPECTS OF AN IMRS

This section presents the reasons why IMRS's are needed, and outlines the design goals of an IMRS. These considerations form a foundation for the module architecture to be proposed later in this report.

2.1. Why an IMRS ?

There are many reasons why a multi-robot system(MRS) is desirable. Two robots could share a single set of tools, a tightly-coupled motion could exist where each robot's actions are not independent, collision avoidance can be performed, one robot can be used as a generalized fixture for another, or robots could help each other when a failure occurs to reduce down time. As we shall see later, these example processes require the use of multiple robots, but using only centralized control cannot efficiently solve all these processes. By efficiently utilizing both the centralized and decentralized control of an IMRS, these processes can be accomplished.

The goal of an IMRS is to *outperform* its counterparts, i.e. several single robot systems or a centrally controlled MRS, including a better utilization of physical space and computer capabilities, and increased throughput, flexibility, fault-tolerance and capability of handling diversified manufacturing processes. Some of these are discussed below in detail.

(1) Due to the necessity of inter-component interactions in a manufacturing cell and the limited size of the workspace, other devices are usually located in the vicinity of each robot for its use. These devices could include robots, automatic feeders, gripper adapted tools, and sensors. These raise the minimum space requirement for the manufacturing cell, and even further increase the set-up cost. With an IMRS, a common set of tools or feeders could easily be shared by two or more robots. By using an "intelligent" compiler (based on a schedule-driven robot language),[4] the work for each robot could be optimally ordered to reduce the likelihood of a resource (tool) conflict. Similarly, several robots could share one vision system, i.e. a global sensor. This could amount to a significant space and hardware savings, without a drop in productivity.

(2) The automation of certain processes require multiple robots. An example would be two robots picking up a large airplane panel and moving it to an airplane wing, after which two other robots could traverse predetermined paths riveting the panel to the wing. This process could be coded in a single task if a language were available that permitted the control of four or more robots, but would be undesirable because: i) a serial program would be used for a process that is an obvious choice for parallel computations, ii) the program would not be easily adaptable because the code to perform four subprocesses would have to be combined into one task, making it hard to isolate the code for one of the four subprocesses, iii) if a failure were to occur (either caused by a software bug or hardware malfunction), the one task would have to be aborted, causing a halt to four subprocesses, thus the whole MRS would fail. These problems can be

[4]A schedule-driven robot language is one which allows the programmer to list each step of the process, giving some steps, a higher priority than others so they are performed first. Steps may be given an equal priority to allow them to be performed in an arbitrary order.

partially improved by using a central task giving directives to each of four subtasks responsible for the four subprocesses, but then new problems are introduced: a) bottlenecking could occur between the robot tasks and the controller, b) the reliability of the system depends on the reliability of the controller, which may be error prone due to possible failures of sensors, communication media, processors, etc., c) a modification to one of the robot process would probably cause a change in two tasks, the controller and the subtask. Changes in both tasks may be difficult to locate, especially if the change is done long after the original coding. It would be preferable to keep the changes local to one task only. The judicious use of both centralized and decentralized controls in an IMRS would eliminate most of these problems. Computational parallelism would be increased, changes to a robot task could probably be kept local to the single robot task, synchronization between robot processes is possible and efficient, and the reliability of the system and communications is increased because there is no one vulnerable central controller for the multiple robot tasks. Another advantage is that using different tasks makes it easier to allow another robot to dynamically change its work schedule to help a failed robot process. Thus processes could be made fault-tolerant.

(3) Single robot systems waste a lot of CPU time, because the processor is blocked while robot motion is being performed, which could typically take anywhere from a tenth to over five seconds. IBM's AML [11] provides an **Amove** which allows the processor to continue executing instructions up to but not including the next motion command, so computations can proceed in parallel with robotic motion. This allows only a minimal amount of parallelism, in comparison to using multi-

ple tasks.[5]

The IMRS approach of utilizing both centralized and decentralized control between tasks offer many advantages over the pure centralized control currently utilized in industry. The design of an IMRS and a corresponding task structure is not a simple problem. Not only must an IMRS allow simple, efficient, and modifiable solutions to a broad class of problems, but it must be fault-tolerant, lend itself to distributed processing, and be predictable in meeting the real-time requirements (i.e. the failure of one task should never induce unpredictable behavior in another task).

2.2. Design Goals

There are many goals which govern the design of an IMRS. In this section we will present the general design goals that we feel are important to take into account when designing the software for an IMRS. Depending on the particular configuration, these goals could have different weights.

(a) The software that controls an IMRS must be able to handle *heterogeneous* components. Users may need to use different robots for different applications, or different computers, or different intelligent machinery in the system.

(b) The IMRS must yield a productive output in comparison to several single robot systems. The goal of an IMRS is to provide extended capabilities beyond that of several independently run single robot systems, but not at the expense of a productivity decrease.

(c) At all costs, the system must be reliable and degradable. This requirement is no different than the requirement of a distributed operating system. The failure of

[5]Even if several tasks are executing on a uniprocessor, it would be simple to use a context switching

one process should neither cause the whole system to fail, nor cause a serious performance degradation. Since an IMRS actually deals with moving machinery, the consequences of an improperly designed IMRS could be drastic. The system should degrade gracefully by first reconfiguring robot processes and tasks, and then informing an operator of the failure and its cause.

(d) An efficient, easily understandable language for programming an IMRS must be available. Since parallel computations and actions are much more difficult to program, it is imperative that a concise, complete language be available. The IMRS system programming language should also adhere to the design goals of a general programming language. These goals include simplicity, reliability, adaptability, efficiency, portability, and generality[8].

(e) Automatic collision avoidance should be provided between robots with overlapping workspaces. This design goal does not change our developments for a modular architecture, but this is a necessary feature of IMRS's. Facilities must be provided to allow convenient collision avoidance programming for the application programmers.

These are our design goals, which we will often refer to in our future discussions. Note, however, that we are addressing in this report only the module architecture problem, not such subjects as the optimal solution for collision avoidance or the ideal robot language.

---

mechanism based on language primitives that introduced blocking, as done in AL [14] and DP [9].

3. THE CLASSES OF PARALLELISM IN AN IMRS

Parallelism in an IMRS exists between robot processes as well as between the computational tasks. The robot process is the actual work being done by the robotic manipulator, e.g., mechanical assembly, part transfer, paint spraying. Two robots might be working on the same process, or they could be working independently. In either case parallelism exists because two robots are simultaneously in operation. Parallelism between tasks occurs when two tasks are executing concurrently, e.g. one task controlling a robot motion while a second task is sampling an A/D port. Since the productivity of an IMRS is measured by the processes being completed, it is more natural to begin by classifying the parallelism at the robot process level. The parallelism and structure of the robot processes will lead us to a system architecture, i.e., a *module architecture.*

3.1. Parallelism Between Robot Processes

There are many ways in which parallelism could occur in an IMRS. Following is a list of the different classes of parallelism between robot processes with an illustrative example process containing each class of parallelism. We feel that all robot processes can be expressed as a combination of these process classes. The term "subprocess" is used to denote a logical component of a process, i.e. a process can be divided into many subprocesses.

A. *Independent Processes:* the work of each subprocess is independent, and the actions taken by each subprocess to accomplish their goals are also independent. If both subprocesses involve robots, then this means that the action of one robot in no way directly influences the action of another robot. Since robots exist together in the same environment, "state" variables (e.g. the speed of a gate

releasing parts onto a conveyor, or the conveyor belt speed) parameterize each process. Thus indirect influence through state variables is the only way the subprocesses of an independent process may be related. The values of these state variables are determined by many different tasks, and thus simultaneous changes must be handled reliably. For this reason, either *proprietor* or *administrator* tasks [5] are needed. These tasks are used to provide exclusive access to shared resources (e.g. the right to change a state variable) and resolve conflicts among different concurrent tasks.

B. *Loosely Coupled Processes:* the subprocesses perform independent work, but the actions taken by each subprocess depend on the actions of the other subprocess. One example of loose coupling between robot subprocesses is tool sharing. If robot A is using tool T, another robot B may be forced into either waiting for tool T, or into performing another action not involving tool T. The work being done by each subprocess is unrelated, but the actions taken at each step depend on each others individual steps. Collision avoidance between two subprocesses is another example of how two subprocesses can be loosely coupled.

C. *Tightly Coupled Processes:* the work of the subprocesses depend on each other, and the actions taken to achieve each subprocess also depend on each other. One example of a tightly coupled process are two robots which must grab a long steel beam off a conveyor belt. The action of one robot subprocess must be tightly coupled to the action of the other robot subprocess. If the robots subprocesses were controlled independently, then the robots would move at different speeds along independent paths, which could cause the beam to slip, or cause damage to one of the robots.

D. *Serialized Motion Processes:* the work of the subprocesses depend on each other, yet the actions taken to accomplish each subprocess are independent. We have chosen the name ***serialized motion*** because the most practical robot process illustrating this interaction involves serializing the action of different robots. The actions taken by the different robot subprocesses are independent (i.e. they can run under their own control), yet one subprocess must wait for the other to complete before it can commence. The use of one robot as a generalized fixture for another robot is an example of this class of process.

E. *Work Coupled Processes:* monitor each other. Should one process crash (due to a robot, computer, or external failure), the other process attempts to compensate. This is identical to how PEARL[18] is used in the furnace application already mentioned. It is obvious that the individual processes themselves will be one of the four aforementioned processes, yet extra work must be done by an IMRS to handle work coupled processes, so it seems appropriate to consider work coupled processes separately. Work coupling may be *one-way* or *two-way*, depending on the ability of the equipment to be used toward either process. Work coupling can even be generalized to allow purely computational tasks to be monitored.

The first four of these process classes are summarized in Table 1. Work-coupled processes are omitted because it is not a primitive process class, i.e. any work-coupled process must also belong to another process class.

Subprocesses	*Actions*	*Process Class*
Independent	Independent	Independent
Independent	Dependent	Loosely-Coupled
Dependent	Dependent	Tightly-Coupled
Dependent	Independent	Serialized-Motion

Table 1 - The Four Primitive Process Classes

3.2. An Example

To illustrate the five process classes, consider the IMRS of Figure 1. This IMRS is one that requires two robots to simultaneously move an airplane panel from one conveyor belt to a slowly moving airplane wing on a different conveyor belt, after which two other robots rivet the panel to the wing. The entire process can be broken into two subprocesses; the first subprocess is moving the panel to the wing, and the second is the riveting. These two subprocesses are both independent and serialized motion processes. They are independent processes because they are coupled via common state variables; the speed of the conveyor carrying the body is one such state variable. They are serialized motion processes because the riveting cannot begin until the panel is in place. Now each of these subprocesses can be broken into two further subprocesses. The first subprocess yields two tightly coupled subprocesses that control the robots performing the simultaneous grasping of the panel. The second subprocess yields two loosely coupled subprocesses that control the riveting robots which require collision avoidance. Also, suppose that should either of the riveting robots fail,

the panel moving robot on the same side of the conveyor would finish the riveting, this results in two separate one-way work coupled process for fault-tolerance.

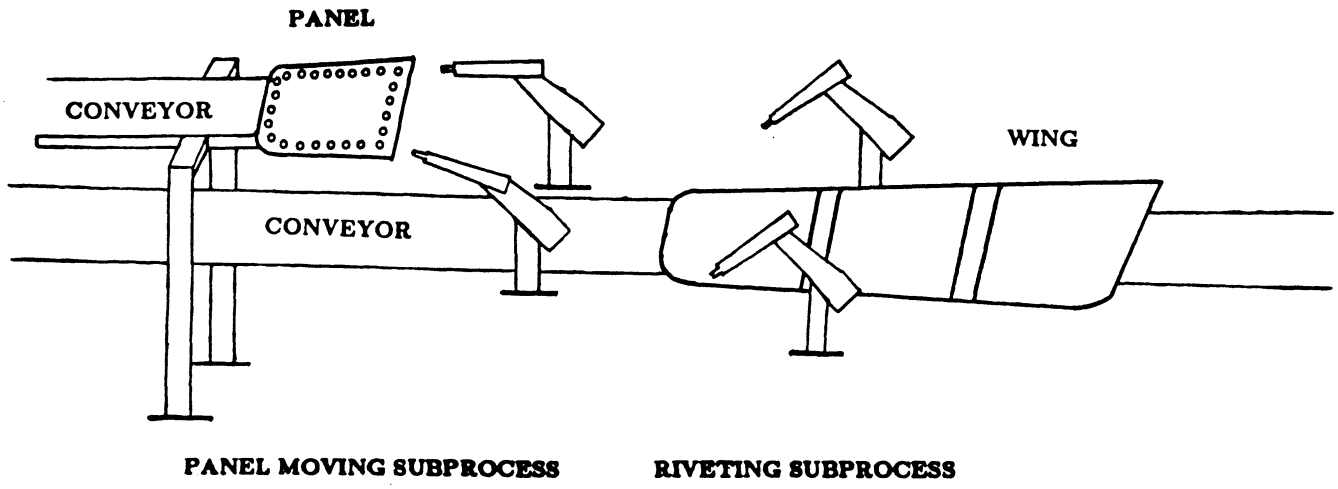

Figure 1 - An Example Multi-Robot System

The five types of parallelism between processes described above encompass a wide variety of applications in an IMRS. If we design our module architecture to handle each type of process, our system should be general enough to handle almost all (if not all) practical applications for an IMRS.

4. THE MODULE ARCHITECTURE

An IMRS process may require several subprocesses to be performed in parallel and/or in sequence. Each subprocess usually needs to execute a number of concurrent tasks, each of which in turn consists of a few procedures. The module architecture refers to the following.

i) The form of a module. A module will have many similarities to Ada task definitions, by consisting of a specification section and a body. The specification section will not only contain standard declarations, but also parameters used to modularize the IMRS. The body will contain a major task that executes in parallel with the other tasks in the IMRS, as well as special interrupt routines that are called when certain conditions arise (i.e. receipt of a message).

ii) The logical structure and/or communication channels that connect the modules in an IMRS.

The module architecture of a real-time or concurrent language is far more complex than the architecture of a sequential language. The architecture of a sequential language (e.g. PL/I [12]) generally deals with procedures instead of modules, and is concerned with the static organization of procedures, scoping rules, and access rights. The real-time, concurrent language required for an IMRS can refer to the static organization of procedures, as in the sequential language case, the task invocation and priority strategy, or the communication channel topology between concurrent tasks. Most of our module architecture discussion focuses on the last concept. We will also need to discuss task creation which forms a backbone of an IMRS, upon which the communication channel topology is overlaid..

The module architecture needs to be well structured so IMRS's can be efficiently, accurately, and reliably programmed. The structure must also provide flexibility, so the application programmer can use the structure toward *any* particular application. Andrews [1] discusses issues pertaining to well structured concurrent programs. And as expected, there are tradeoffs between structured concurrent constructs (e.g. **Cobegin-Coend**) and unstructured constructs (e.g. **fork/join**). As we discuss the communica-

tion topology aspect of the module architecture, we will present two structured approaches, the *vertical* and *horizontal* approaches. We will discuss the issues affecting the use of each approach, in the hopes that IMRS application programmers use these structured approaches correctly. Just as a sequential language programmer can forego **if-then-else**'s, **do-while**'s, and **repeat-until**'s in lieu of **goto**'s, so may our structures be misused.

We propose the module architecture for an IMRS to be an n-ary tree, that is formed by *task creation.*[6] A module will consist of a specification section, an implementation (i.e. a main task responsible for the module's function), and exception handlers. When a task is created, it becomes a child task of the task that created it.[7] This parent-child relationship between the tasks always exists, but the amount of communications between the two will be different according to the class of process that the tasks are controlling. Under most circumstances, communication channels among child tasks will be directly established, with the parent task playing a minor role. This is deemed *horizontal communications.* Note in this case that despite the parent-child relationship via task creation, there is little need of communications between the parent and its child tasks. However, in some cases the parent must tightly control its child tasks. This is deemed *vertical communications,* which is characterized by a close-knit relationship between a parent and its children. Note that these approaches represent centralized and decentralized controls, respectively.

[6]When a task begins executing.

[7]This is a slight abuse of terminology, since the creation of a new task really creates a module, which implies a whole lot more than a task. However, our future discussions will be clearer if we talk about child and parent tasks, since the tasks are responsible for the modules' function.

4.1. High-Level Communications in an IMRS

In this section, we will discuss the vertical and horizontal modes of communications. Vertical communications is defined as communications between a task and any of its descendant tasks. *Purely* vertical communications are those which occur between a task and its immediate child tasks. If the standard n-ary tree is drawn with children placed under their parents with an arc connecting them, communications between a parent and a descendant occur vertically in the tree. Horizontal communications is defined as communications that occur between tasks that are not related vertically (i.e. a cousin or uncle relationship exists between the tasks). *Purely* horizontal communications are those which occur among the children of a common parent. For simplicity, we only deal with the pure forms of communication here. However, our results and arguments can be generalized to IMRS's not constrained in this manner.

Vertical communications use one task to control its child tasks. This allows convenient synchronization of many child tasks, by making them await a directive from the parent. This scheme is easy to program, and provided that i) the number of child tasks is small, ii) the IMRS processes are not apt to be modified often, iii) the parent task is very reliable, and iv) the child tasks are not computationally intensive, then vertical communications can be utilized to solve the five classes of robot processes. However, if i) is invalidated then communications bottlenecking could occur; if ii) is invalidated then changes will have to be made to more than one task in the system, which may be difficult to locate; if iii) is invalidated then the system is not fault-tolerant; and if iv) is invalidated then parallelism is not being exploited. Horizontal communications ameliorate the IMRS significantly. Allowing tasks to communicate

directly without a central controller i) reduces the chances of a bottleneck by exchanging messages among children, ii) keeps all the code for each subprocess local to one module, iii) increases reliability because all the subprocesses do not rely on one central control task, and iv) allows more parallelism because each child task is not blocked as often as in the vertical case, when a child must always await a directive from the parent. Some processes are more conveniently and efficiently programmed using vertical communications, while others are best expressed horizontally. It will be shown in the next two sections that *tightly coupled processes* are best expressed vertically, while *independent, work-coupled, loosely-coupled,* and *serialized motion processes* are best expressed horizontally (with a minimum use for vertical communications).

4.1.1. Vertical Communications

In an IMRS, a parent will spawn child tasks to perform different subprocesses in the work cell. Naturally the single parent could control the subprocesses without creating additional tasks, but then parallelism is sacrificed, and reliability is decreased (since a failure in the parent would cause the subprocesses to fail). It is better to have concurrent child tasks with their own control logic, to perform the subprocesses. If the parent must block until the children complete execution, then the concurrency can be modeled in terms of concurrent subroutine calls. In most cases, the parent will create child tasks to handle subprocesses in the IMRS, and will continue doing other work. A message and a run-time priority will be given to the child tasks when they are created. The child tasks will perform their function independently of the parent. Depending on the urgency of the subprocess controlled by the child task, the parent may wish to check on the status/progress of the child task before the child task is completed. This may be done by having the child main-

tain a status information that the parent can read.

Consider a tightly coupled process of moving an airplane panel from one conveyor to another (Figure 1). The two controlling tasks (one for each robot subprocess) must be kept in tight synchronization, one task cannot run arbitrarily ahead of the other. Suppose the synchronization is to be performed by having each robot move a small distance at a time. After each step is performed, the tolerances of the position and forces on each gripper can be sampled, with appropriate action being taken. This would be difficult to perform using two independently executing tasks. If each task knows the ideal amount for each step in the (position and force trajectory) interpolation, then the tasks could synchronize by using a simple signal-wait binary semaphore. This is unrealistic because (i) it requires compile-time knowledge of all the moves so the step size can be determined before creating the two controlling tasks, (ii) neither task has absolute control over the whole process, and thus resolving conflicts in concurrent tasks becomes necessary (and is hard to perform correctly), and (iii) the frequency of messages between the two tasks will be high, because the very nature of a tightly coupled process requires many messages to keep two independently executing tasks tightly synchronized. The best way to solve this problem is by utilizing a master/slave relationship between tasks. How can this master/slave relationship best be expressed? Since the master controls the slave, the master is in charge of creating the slave, sending the slave commands, and destroying the slave when it becomes unnecessary. The module architecture makes a created task a child of its creator, and thus it is natural for the slave to become a child of the master. The master/slave relationship required to perform tightly coupled processes is best expressed by making the parent a master, and the children slaves, with verti-

cal communications used between the parent and its children.[8]

Under what circumstances should vertical communications be used in solving the other classes of processes? Vertical communications (that are time critical) should only be allowed when the parent is in charge of the communications between itself and the children. If all the children could initiate communications with the parent, then a bottleneck could result and the parent probably may not be able to respond quickly enough (if the number of children is large). We should not allow vertical communications that are initiated by a child task which expects a time-critical response. If the other four classes of processes are to be realized with vertical communications only, then the parent must be responsible for initiating all the communications between itself and its children. Of course, one could use vertical communications initiated by children in some IMRS's where the application programmer knows for certain that vertical bottlenecking will not occur. But this is only acceptable for small IMRS's, and since in general IMRS's will be large (e.g. automating an entire assembly line), we do not allow a child to initiate a time-critical communication transaction with its parent. Thus if the other four process classes are to utilize vertical communications, then the processes will have to be programmed in such a way that the parent can initiate all the communications. Controlling independent, serialized motion, work-, and loosely- coupled processes with vertical communications can be done, using the child tasks in this fashion has contradicted the very reasons child tasks were spawned; the child tasks were created to maximize parallelism, increase reliability, and to increase adaptability. Because the parent would have to be given the control and synchronization logic of both tasks, the child would become

[8]Depending on the nature of the tightly coupled process, a different number of child tasks may be spawned. In some cases the process is best controlled by using the parent and a single slave child, while in other cases it is better for the parent to control two or more slave children.

a dumb slave, simply awaiting directives from the parent to perform its next step in the process. The child tasks will almost always be blocked, the control of a subprocess depends on two tasks instead of one, and the code for one subprocess is scattered between two tasks. The problems of using the master/slave vertical communications for these process classes can perhaps be intuitively grasped from Table 1. The (centralized) master/slave relationship implies high dependency between processes and actions, and is only appropriate for process classes requiring this high dependency, i.e. tightly coupled processes only.

4.1.2. Horizontal Communications

Horizontal communications are used to avoid the above problems that arise when using the centralized control of the vertical scheme. Child tasks are allowed to send messages directly among each other, each being able to initiate a transfer. Horizontal communications are used between child tasks of the same parent, but not between the parent and a child. Horizontal communications are in fact preferred over vertical communications for all but tightly coupled processes because of improved reliability, communications efficiency, source-code adaptability, and truer parallelism. Horizontal communications can take on many forms: IMRS state variables can be controlled by proprietors which receive requests from other child tasks asynchronously, two tasks can query or inform each other as to their current status, or synchronization signals can be exchanged between tasks.

How may horizontal message passing be realized? From a user interface perspective, message passing and remote procedure calls should suffice, but the underlying system implementation can be modeled as follows: Each module will contain a message routing handler, called a *message handler* (MH), which receives and forwards

messages among other modules horizontally. A message would originate from a child task, be encoded by its MH, be relayed between sibling MH's until it reached the MH of the destination module, upon which that MH would decode the message and relay it to the destination task. Small, fixed length messages would be used so the communications could be efficiently handled.

The message handlers will have to be fast, "intelligent" nodes, capable of deciding what is most important at any given time for the IMRS, as quickly as possible (in many ways like a scheduler). As a message is received, the task executing in the module is interrupted to allow the MH to execute. The MH will have to decide (based on task, message, and communication channel priorities) whether it is best to allow the task to continue executing (thus queueing the message), or to suspend the task while the message is processed. The MH will also have to decide the best route to send a message to another module, and will have to utilize acknowledgements and timeouts to assure reliability of communications. How the MH is controlled is a difficult problem, because an application programmer should not need to program the implementation details. One possibility is to use a rule-based *expert system* which allows an application programmer to simply provide the rules for making the proper decision in real-time. The expert system will have the power to interrupt the current thread of control, as well as dynamically change priorities of the tasks and messages so the real-time constraints can be met. The MH's would have to be linked together in a certain topology, i.e. ringed or clustered. These communication links are different from the logical links connecting a parent to its children (which just model task creation). By providing multiple links to every module, the communications become more reliable (dynamic redundancy [18]) because the failure of one channel still allows communication by another channel.

To have a better understanding, consider an example case of achieving independent processes with horizontal communications. Suppose that 20 child tasks all depended on the value of a particular state variable (and thus we have a 20-way independent process). A local copy of the state variable can be kept in each child, but changes to the state variable must be handled by communicating horizontally with a proprietor (which can modify the state variable).[9] The twenty children could be linked in a ring-like manner, with the two[10] child tasks most likely to communicate with the proprietor having additional links to the proprietor (this would have to be specified by the application programmer as an MH priority for messages destined to the proprietor). If these two child tasks are uniformly spaced among the ring, then no message will have to traverse more than four intermediate MH's. If the messages are not being handled quickly enough due to propagation delays in the messages, then extra direct channels to the proprietor can be created. If the proprietor receives too many requests too quickly, then the proprietor's MH could send a message to the parent, asking the parent to spawn a redundant copy of the proprietor. The parent would make a decision based on the feasibility of the request, and would dynamically alter the horizontal communication links if another proprietor were spawned. Redundant copies of the proprietors may simultaneously decide to alter the same state variable, so these conflicts must somehow be resolved. To handle this, we let the parent *verify* any modification to a state variable by proprietors. The parent would check the feasibility of the new value, and would send a message to each child of the change. Thus using proprietors to handle independent processes horizontally still requires a small amount of vertical communications. For increased

[9]A normal child task is not allowed to modify state variables.

[10]For reliability reasons, there must be a minimum of two-way connectivity.

reliability, multiple proprietor tasks would be work-coupled together, and a single proprietor task can be work-coupled to any sibling. The work coupling of two tasks would require each task to maintain a database for its mate, based on messages sent between the tasks. Should one task notice a lapse in messages from another task, action would be taken to find out if the other task was still functioning properly. If not, then the work coupled task would invoke a handler to recover.

An important issue is whether or not this proposed method of communications allows the real-time constraints to be met. If the messages must traverse paths consisting of too many intermediate MH's, then message delays could become critical. There are two ways to overcome this problem; either include extra horizontal channels between tasks of a common process (that are not normally connected due to nonproximity in the n-ary tree), or to arrange the child tasks in the tree so that tasks controlling each process are located next to each other in the n-ary tree so that the channels comprising the ring become direct channels between tasks of the common process. The latter is preferable, since it does not introduce extra communication channels, but is not always feasible. Thus a combination of both should be utilized.[11]

Horizontal message passing is utilized by processes which do not require tight synchronization. We have discussed how independent processes can be accomplished using horizontal communications with a proprietor(s). Loosely-coupled, work-coupled, and serialized motion processes can likewise be programmed horizontally, by sending messages directly between child tasks. The basic concepts presented thus far bear many similarities to work done in networking [3], [13], [16]. In Arpanet, for example, messages are routed between hosts by interface message processors (IMP's).

[11]A mechanism for doing this is the **Costart** system call, which will be discussed in Section 4.3.

The IMP handles disassembling and encoding the messages into packets, shipping the packets reliably to the destination IMP, and assembling and decoding the packages back into a single message. The differences are i) that we are dealing with tasks, which may be executing on a distributed network, as opposed to host processors, ii) our MH's must be user programmable to allow real-time constraints to be met, and iii) packet switching is not employed to meet real-time constraints.

4.2. An Example Module Architecture

Figure 2 illustrates the module architecture for the airplane panel-riveting process of Figure 1. The root task is the highest level task that creates two child tasks to perform the two subprocesses. Task A performs the panel moving subprocess, and task B performs the riveting subprocess. Tasks A and B each create two other tasks to actually perform the robot subprocesses. Tasks a1 and a2 control the panel moving robots, tasks b1 and b2 control the riveting robots. Tasks a1 and b1 control two of the robots on one side of the conveyors, while tasks a2 and b2 control the two robots on the other side of the conveyors. Task C is either a *proprietor* or *administrator* used to listen to requests from A and B to monitor and modify the system state variables. In this small example, one could delegate the work of the proprietor to the parent since a vertical bottleneck will not result between two children, yet for consistency with our prior developments, as well as with larger IMRS's, we perform the independent process horizontally instead of vertically. This proprietor task is work coupled to the riveting process, so a failure in the proprietor will cause task B to execute an error handler that will recover for the proprietor. Note that the tightly coupled panel moving subprocess utilizes vertical communications, while all the other processes utilize horizontal communications. The panel moving tasks (a1 and a2) receive messages

from the riveting tasks (b1 and b2), in order to perform work coupling. Tasks b1 and b2 communicate directly between themselves to allow riveting without colliding (loosely coupled). Tasks A and B use a horizontal communication channel so as to allow synchronization between the two subprocesses. Each of the tasks indicated is really a module, consisting of a declaration of communication channels needed, an MH, priority rules, exception handlers, and its primary task.

Figure 2 illustrates one important feature of the module architecture of an IMRS. First notice that one task is not confined to one process only. Task A belongs to an independent, serialized motion, and tightly coupled process. Task B belongs to an independent, serialized motion, loosely-, and work- coupled process. Because of the work coupling, the child tasks a1, a2, b1, and b2 also belong to more than one class of

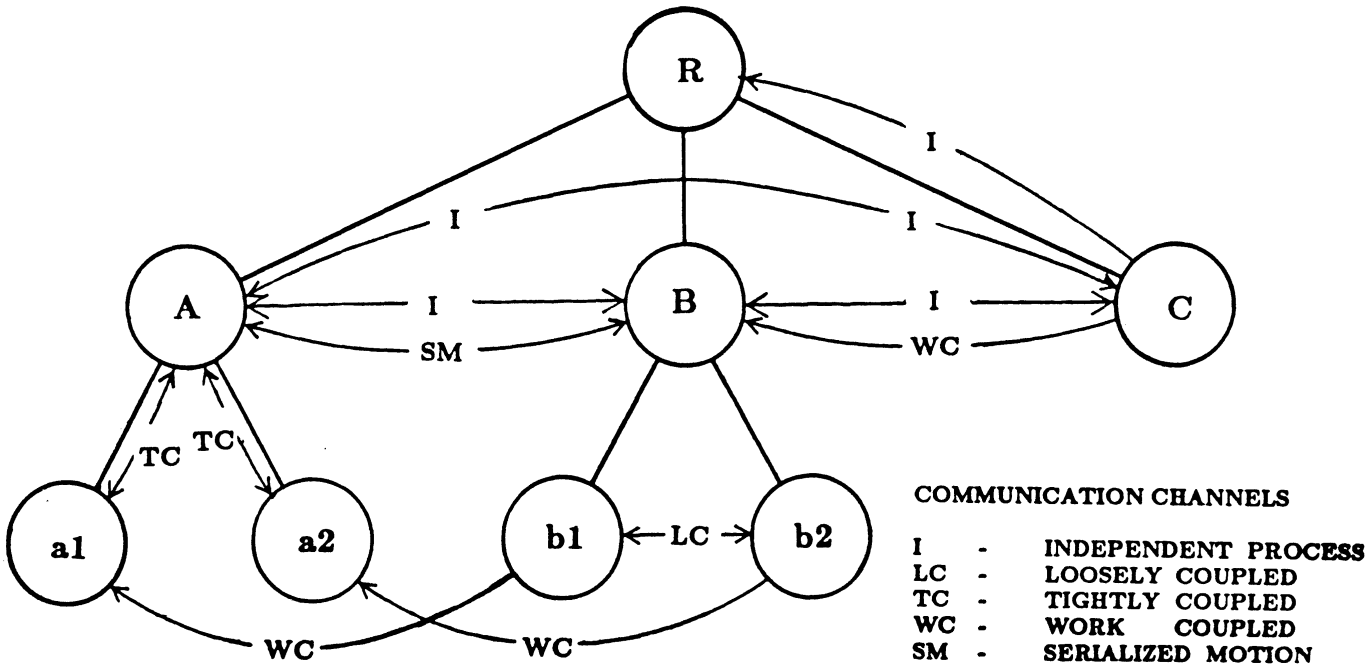

Figure 2 - The Module Architecture for Figure 1

process.

4.3. More on Module Architecture

Our proposed module architecture consists of an n-ary tree, formed by task creation. Parent tasks will create child tasks, which will communicate either horizontally or vertically. The module architecture provides an application programmer with the flexibility to choose a method of communications suitable for a given process.

There are a few last remarks to be made concerning vertical and horizontal communications.

- We have discussed how an independent process may still utilize vertical communications by having the father verify requests to change a state variable by its proprietors. This is why there is a unidirectional vertical transmission from task C to task R in Figure 2. Similar situations can be constructed for the loosely- and work- coupled, and serialized motion processes for which we advocated horizontal communications.
- Using a strict ring-like structure for horizontal implementation of independent processes may not be the best choice, but we chose it for illustrative purposes. Since IMRS tasks may be clustered (e.g. one sibling task coupled to another may not ever need to call the proprietor), a clustered topology similar to a cluster network[15] may be more appropriate. Regardless of the logical MH connections, the message handlers should try to choose the best message route dynamically.
- In general, no perfect suggestions can be made on the use of vertical and horizontal communications. For example, we discussed how adaptability of horizontal communications is easier since there is no need to also modify a central con-

troller. Although always true, more complex tasks are needed to perform communications horizontally. Depending on the language constructs chosen, it may be easier to change two separate tasks than one more complicated task. This depends on the particular IMRS application. The IMRS programmer should be given the flexibility of choosing the method best suited for each problem.

- Not all communications needed are purely horizontal or purely vertical. The example of Figures 1 and 2 show that there is work coupling horizontal communications between the task pairs a1-b1 and a2-b2 that are *not* pure because each task of the pair has a different parent.

For completeness, we need to discuss a few more general features, namely task definition, creation, and destruction.

Task Definition - We feel that tasks should be defined in a way similar to Ada. The benefit of doing so allows one to exploit Ada's architectural features. Several papers have been published suggesting Ada as a language for robot programming[6], [4]. Ada was designed to facilitate large software projects, just like programming an IMRS. The primary advantages are data encapsulation, data abstraction, and type abstraction. Consider the form of a package: a specification and an implementation. Pure Ada allows variables, types, tasks, or procedures to be declared in the specification. By generalizing this notion to allow each task to contain different specification sections for different IMRS features, we can entirely modularize each IMRS task. For example, one specification could declare what sensors a robot had, another could declare the MH priority rules for horizontal communications, another could specify the work schedule, another the communication channels, etc. Since the specification section would contain information to the operating system on how the module relates to

the entire IMRS the efficiency, reliability, and verifiability of an IMRS application program (or process) could be checked and changed by looking at the specifications that contain information pertinent to the module architecture. PEARL[18] uses this technique by providing a SYSTEM-division for declaring I/O connections, a PROBLEM-division for computation, a LOAD-division for fault-tolerance provisions, and a STATIONS-division describing the capabilities of each processor.

Task Creation - There are several ways that tasks can be created. One mechanism is the **Cobegin-Coend** construct, another is to let all declared tasks start simultaneously, another is to declare tasks in the declaration section of a procedure and let the tasks begin execution when the procedure begins execution (as in Ada). Ada even allows a task type, which can be used to dynamically create tasks. The **Cobegin-Coend** construct is appropriate when the task structure can be fixed at compile time[20]. This is almost suitable for an IMRS; the only problem is that the **Cobegin-Coend** causes blocking of the parent. What is needed is a **Costart** instruction, which is similar to a **Cobegin** except it is nonblocking. Besides this trivial difference, other advantages lie in using a **Costart**. If the communication channels could not be statically declared in the specification section, then the **Costart** could be used to specify the desired dynamic communication channels between the children (horizontally) or between the children and parent (vertically). A more important advantage is that repetitive uses of **Costart** would automatically adjust the n-ary tree to give a parent task more child tasks, and automatically readjust the MH communication topology to optimize horizontal communications. Thus **Costart** would not be a simple primitive, but rather an IMRS system call accepting many parameters.

Task Destruction - A task is destroyed either intentionally or unintentionally, and the action taken depends on which is the case. If two child tasks were work coupled and one was to be destroyed because of execution completion, then the work coupling would have to be disabled, and the intertask communications aborted. But if one child task was to abort abnormally, then the other child would have to notice this and alter itself to cover for the failed child. In general, the destruction of a task will cause a reconfiguration of the task tree, will cause different (predictable) actions in other tasks, and will alter the communication channels of an IMRS. The unintentional termination of a task (e.g. an emergency condition requiring immediate shutdown of a process and its controlling tasks) which is not work coupled would probably lead us to a graceful shutdown, but such decisions should be made by the application programmer. Error handlers could be invoked by simply signaling (or raising) a condition, as done in Ada [7] and PL/I [12].

5. CONCLUDING REMARKS

In this report we have first investigated the various communication demands brought about by five different types of processes, *independent, loosely coupled, tightly coupled, serialized motion,* and *work coupled processes.* Then, we have presented a module architecture which deals with the run-time relationship of tasks, including task creation, task destruction, and intertask communication channels. Two types of communications are needed, *vertical communications* to handle tightly coupled processes which require a parent task to control its child tasks in a master/slave relationship, and *horizontal communications* to handle communications directly between child tasks not requiring a central controller.

We feel that the work described in this report forms a foundation for developing high performance, flexible, and fault-tolerant automation systems. However, there are several important topics to be studied before a complete IMRS system is developed. Following are some of such topics.

- Determining what to include in the specification sections of a module.
- Choosing an appropriate set of communication primitives. They will be based on message and operation oriented languages [5] because of their applicability in distributed systems.
- Determining how to handle the priorities of messages in the message handlers for each task. As discussed in Section 4.2.2, this will probably have to be an expert system, which can be easily modified if messages are not properly handled.
- Evaluating the best way to define the communication topology in the source program. Port directed communication [20] may be a possibility.
- Designing the **Costart** system call. More specifically, the parameters it should accept and how the communication topologies should be adjusted on each call.
- Designing an IMRS system programming language which allows simple, efficient, and reliable programming of the IMRS processes. This will also be difficult; creating a robot programming language that can be used by people of different experiences, especially one with the power to control an IMRS as we have described, will be quite challenging.
- Developing a complete operating system kernel. The V System [3] has three major components, the interprocess communications (IPC), the kernel server, and the device server. Our discussion of horizontal communications has many similarities to Cheriton's IPC. The kernel will have to be developed along with the network

implementation of horizontal communications, and will prove to be difficult, because of all the different devices and sensors which must be incorporated into the system along with the real-time communications.

REFERENCES

[1] Andrews, G.R., and Schneider, F.B., "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, March 1983, Vol. 15 No. 1, pp. 3-43.

[2] Bonner, S., and Shin, K. G., "A Comparative Study of Robot Languages", *Computer*, Vol. 15, No. 12, December 1982, pp. 82-96.

[3] Cheriton, D. R., "The V Kernel: A Software Base for Distributed Systems," *IEEE Software*, April 1984, pp. 19-42.

[4] Gal, D., Mudge, T., and Volz, R., "Using ADA as a Robot System Programming Language," *Proceedings From the 13th International Symposium on Industrial Robots and ROBOTS 7*, 1983, pp. 12*42-57.

[5] Gentleman, W.M., "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept," *Software Practice and Experience*, Vol. 11, 1981, pp. 435-466

[6] Gini, G. and Gini, M., "ADA: A Language for Robot Programming?," *Computers in Industry*, 1982, Vol. 3, No. 4, pp. 253-259.

[7] Habermann, A., and Perry, D., *Ada For Experienced Programmers*, Addison-Wesley, 1983.

[8] Hansen, P. B., *The Architecture of Concurrent Programs*, Prentice-Hall, Inc., 1977.

[9] Hansen, P. B., "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM*, Nov. 1978, Vol. 21, No. 11, pp. 934-941.

[10] Hoare, C.A.R., "Communicating Sequential Processes," *Communications of the ACM*, Aug. 1978, pp. 666-677.

[11] IBM Corp., *IBM Robot System/1: AML Concepts and User's Guide*, Publication No. GA34-0180-1, 1981.

[12] Hughes, J. K., *PL/I Structured Programming,* the 2nd. Ed., John Wiley and Sons, 1979.

[13] McQuillan, J. M. and Walden, D. C., "The ARPA Network Design Decisions," *Computer Networks,* North-Holland Publishing Co., 1977, pp. 243-289.

[14] Mujtaba, S., and Goldman, R., "AL Users' Manual," *SAIL Report,* Jan. 1979.

[15] Schoeffler, J. D., "Distributed Computer Systems for Industrial Process Control," *Computer,* Feb. 1984, pp. 11-18.

[16] Shoch, J. F., and Hupp, J. A., "Measured Performance of an Ethernet Local Network," *Communications of the ACM,* Dec. 1980, Vol. 23, No. 12, pp. 711-721.

[17] Silberschatz, A., "Cell: A Distributed Computing Modularization Concept," *IEEE Transactions on Software Engineering,* March 1984, vol. SE-10, no.2, pp. 178-185.

[18] Steusloff, H. U., "Advanced Real-Time Languages for Distributed Industrial Process Control," *Computer,* Feb. 1984, pp. 37-46.

[19] Stotts, P. D. Jr., "A Comparative Study of Concurrent Programming Languages," *ACM SIGPLAN Notices,* Sept. 1982, vol. 17, no. 9, pp. 76-87.

[20] Wegner, P., and Smolka, S. A., "Processes, Tasks, and Monitors: A comparative Study of Concurrent Programming Primitives," *IEEE Transactions on Software Engineering,* Vol. SE-9, No. 4, July 1983, pp. 446-462.

[21] Welsh, J., and Lister, A., "A Comparative Study of Task Communication in Ada," *Software- Practice and Experience,* 1981, vol. 11, pp. 257-290. Springer-Verlag, 1982.