Intertask Communications in an Integrated Multirobot System

KANG G. SHIN, SENIOR MEMBER, IEEE, AND MARK E. EPSTEIN

Abstract—**An integrated multirobot system (IMRS) consists of two or more robots, machinery, and sensors and is capable of executing almost all industrial processes with efficiency, flexibility, and reliability. Although the IMRS is motivated by an interesting application, it is essentially a distributed real-time processing system with various heterogeneous processes. To support a distributed modular architecture of the IMRS, low-level *communication primitives* are proposed along with their supporting *language syntax* which are typical of real-time concurrent programming languages. This is done by 1) carefully examining the generic structure and interactions of IMRS processes, 2) comparing and analyzing the primitives and syntax developed/proposed for general concurrent programming, and 3) using port-directed communications.**

I. Introduction

Using a multirobot system (MRS) for manufacturing applications has numerous advantages. For example, the system throughput or productivity can be increased by exploiting the inherent parallelism; structural flexibility can be accommodated for diversified applications; and even system reliability is achievable via the multiplicity of robots.

Conventionally, MRS's are all centrally controlled; that is, control tasks for an MRS may be distributed over a network of processors but are all executed under the supervision of one central task. Heterogeneous controllers in such an MRS are made to converse via a standard communication protocol, e.g., GM's MAP [2]. By using a network to tie MRS's components together, it is possible to have robots working together to solve processes,[1] instead of working independently. Although almost all manufacturing processes can be handled by the conventional central controller, communications bottlenecking and unreliability (that occurs at the central controller) become major problems. For this reason we have defined in [16] a new MRS, called integrated multirobot 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,
- the tasks are executing in true parallelism,
- it is adaptable to either centralized or decentralized control concepts,
- tasks may be used for controlling other machinery (e.g., intelligent device driver for CNC's), sensor I/O processing, communication handling, or just plain computations.

Further, we have developed in [16] a high-level abstraction of IMRS communications, called a *module architecture,* to support an IMRS; this will be discussed briefly in Section II.

The reasons for needing a structured solution to the IMRS problem are fundamental. The distributed nature of the IMRS will make programming more difficult, error prone, and subject to complicated communications control techniques. Two immediate places where structuring is needed is in the design of a *module* (the computational entity that controls an indivisible subprocess in the system) and the intermodule communications. Borrowing notions from software architecture, it is important to encapsulate the implementation of a subprocess into a single module, showing only interface information to other modules. If the software is structured in this manner, then

1) the system is easily adaptable,
2) the system is maintainable,
3) taking the step from the process structure to its module implementation is easier, and
4) the complexity of the system is reduced.

These are just some of the benefits of having a well-structured module and intermodule communication structure. Before delving into the presentation of the proposed solutions, we first look at other approaches used in related areas and discuss why these cannot be ported to the IMRS.

Numerous robot languages (see [4] for a survey) have been written which can control more than one robot simultaneously, the most advanced being AL [14]. AL allows one program to control two robots at once. By using COBEGIN–COEND pairs, a programmer can initiate two pseudoconcurrent tasks. They can be synchronized using the event data type (integer semaphores). The principal motive behind this design was to allow cooperation via serializing the execution of tasks for each robot's motions by using events. This restricts the potential amount of parallelism that can be attained. It would

Manuscript received May 5, 1985; revised June 30, 1986. This work was supported in part by the NSF under Grant ECS-8409938 and in part by the US AFOSR under Contracts F49620-82-C-0089 and F33615-85-C-5105. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the view of funding agencies.

K. G. Shin is with the Computer Science and Engineering Division, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109, USA.

M. E. Epstein was with the Computer Science and Engineering Division, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109. He is now with the T. J. Watson Research Center, IBM Corporation, Yorktown Heights, NY 10598, USA.

IEEE Log Number 8613008.

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

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 [19], but the results are not easily transportable to an IMRS. This work 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 h of downtime in more than 24 000 h 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 heavy communications and tight synchronization. Because of the more dynamic nature of the IMRS, a more intricate, flexible communications structure is needed.

Considerable research has been done in concurrent programming languages. Many languages have been created, each utilizing different primitives to allow communication and synchronization. Andrews [1] has classified concurrent programming languages into three classes: *procedure-oriented, message-oriented,* and *operation-oriented.* The last two classes are most suitable for an IMRS because the IMRS is inherently distributed. Some languages which fall into this class are distributed processes (DP) [10], communicating sequential processes (CSP) [11], Thoth [8], and Ada [7]. Although each language uses different communications mechanisms, it is claimed that for common concurrent benchmarks (e.g., dining philosophers problem, bounded buffer) the mechanisms of any of these languages can be used [1], and that at an abstract level, their powers are equal. However, for real-time industrial process control, these languages are virtually untested. We expect real-time process control languages to evolve over time and making a statement as to the best primitives to use would be futile. Different applications place different demands on the underlying language, and as computers and robots are used to automate more complex processes in the future, needs will be generated for new communications primitives. Regardless of the communication primitives used, it will be necessary to structure the communications into well-defined channels for many reasons.

a) Because a complex IMRS will be programmed by programming teams working independently, a structured interface between their respective pieces, while hiding implementation details, will be necessary.

b) Debugging and maintaining the IMRS will be easier if declared communication channels exist.

c) Heterogeneous components will need to be linked via communications, and a clear flexible design will allow easier integration.

d) Implementation of distributed communications will be easier.

Certainly one could come up with more reasons than these to justify the need for structured communication channels. Ada's entry-accept mechanism can be viewed as a channel and Digital Equipment Corporation's VAXELN [6] is the first real language that uses ports [7]. However, VAXELN supports neither multiprocessing nor port options that are needed for an IMRS. Except for these two, none of the other concurrent programming languages provide structured channels. In this paper, we propose to use ports and various port options to structure the intermodule communications of an IMRS.

This paper is organized as follows. We briefly review in Section II the module architecture that we have developed in [16]. We advocate the port directed communications for an IMRS in Section III. In Section IV we identify first communication needs for each process class and then propose the primitives most suitable for an IMRS. In Section V we combine our notions of the module architecture and the primitives into a communication structure. Section VI concludes the paper with a brief mention of the remaining work needed to implement an IMRS.

II. Review of Module Architecture

As was pointed out in the Introduction, the term "process" will be used to mean an *industrial* (but not computational) process, which could be decomposed into several *subprocesses.* Each subprocess may be accomplished by executing a *module* in a computerized controller. Each module can be decomposed into computational tasks.

For completeness we briefly review the module architecture proposed in [16]. Our development toward a module architecture began with the classification of IMRS processes, which is given in Table I. Each process is broken into two or more subprocesses, whose intended work may or may not be dependent. The actions taken (in both the software and hardware) to achieve each subprocess also may or may not be dependent. In Table I we have named each of the four possible process classes appropriately. The formal definitions of each process class conform to the different interactions between subprocesses and their actions. Examples of each class are as follows.

1) Independent Processes: Two robots exist on the same plant floor, but the work for each robot is independent of the other's and is blind to the other's existence. Each robot may depend on common state variables (e.g., conveyor belt). The values of these state variables are determined by many different tasks, and thus simultaneous changes must be handled reliably (e.g., by use of a *proprietor* or *administrator* [8]).

2) Loosely Coupled Processes: Tool sharing is an example of this class. If robot A is using tool T, another robot B may be forced into either waiting for tool T or performing another action not involving tool T. The work of each robot is independent, but the individual actions taken are not. Collision avoidance between two robots executing independent processes but sharing the same workspace is another example of a loosely coupled process.

TABLE I
THE FOUR BASIC PROCESS CLASSES

Subprocesses	Actions	Process Class
independent	independent	independent
independent	dependent	loosely coupled
dependent	dependent	tightly coupled
dependent	independent	serialized motion

3) Tightly Coupled Processes: One example of a tightly coupled process is two robots which must grab a long steel beam off a conveyor belt. The action of one process must be tightly coupled to the action of the other process, otherwise the beam could slip or damage could occur to a robot.

4) Serialized Motion Processes: We have chosen the name *serialized motion* because the most practical process illustrating this interaction involves serializing the action of different robots. If subprocess A must be executed before subprocess B can commence, then A and B form a serialized motion process. The use of one robot as a generalized fixture for another robot in a two robot system is an example of this.

5) Work-Coupled Processes: This class is not listed in Table I because it is not a basic process class. If two processes are work-coupled, then should one process fail, the other will perform error recovery and take over the responsibilities of the failed process. Obviously, the process will also be one of the four aforementioned processes. Work coupling may be *one-way* or *two-way*, depending on the ability of the equipment to be used toward either process. The furnace pit operation described in [19] utilizes two-way work coupling.

The process structure of an IMRS is *hierarchical*. The main process is divided into many subprocesses, which are further divided, and so on. Eventually, the industrial process is divided into many indivisible subprocesses. Each of these subprocesses will be programmed with a *module*. The *module architecture* refers to the structure of a module and the logical structure and/or communication channels that connect the modules in an IMRS. Note that, when a distributed network is used, the hierarchical structure offers several advantages, e.g., easier implementation and better adaptability.

Because a module controls an indivisible subprocess, we will often use "task" instead of "module." We do this when we are concerned with the module's function, and thus the work of the primary, not auxiliary, tasks. This notation makes our later discussions more comprehensible.

We have proposed the module architecture for an IMRS to be an n-ary tree, that is formed by *task creation*.[2] When a task is created, it becomes a child of the task that created it. 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 termed *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 termed *vertical communications*, which is characterized by a close-knit relationship between a parent and its children. Note that these approaches represent *centralized* and *decentralized* control, respectively.

[2] When a task begins executing.

Vertical communications are defined as communications between a task and any of its descendant tasks. *Simple* 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 are defined as communications that occur between tasks that are not related vertically (i.e., a sibling, cousin, or uncle relationship exists between the tasks). *Simple* horizontal communications are those which occur among the children of a common parent.

Vertical communications are used in most currently existing MRS's. Synchronization of child tasks is achievable by having the parent issue directives, i.e., interrupts. This scheme is easy to program and efficient, provided that 1) the number of child tasks is small, 2) the IMRS processes are not apt to be modified often, 3) the parent task is very reliable, and 4) the child tasks are not computationally intensive. However, if 1) is invalidated, then communications bottlenecking occurs; if 2) is invalidated, then changes will have to be made to more than one task in the system and will result in downtime; if 3) is invalidated, then the system is vulnerable to a single point failure in the parent; and if 4) is invalidated, then parallelism is not being exploited.

Horizontal communications ameliorate the IMRS significantly. Allowing tasks to communicate directly without a central controller reduces the chances of a bottleneck by exchanging messages among children, keeps all the code for each subprocess local to one module, increases reliability because all the subprocesses do not rely on one central control task, and allows more parallelism because each child task is not blocked as often as in the vertical case, where the child must always await a directive from the parent.

How may horizontal message communications be realized? Each module will contain a message handler (MH), which receives and, if necessary, forwards messages among other modules horizontally. The MH would be part of the operating system and would not only act as an *interface message processor* (performing all the detailed work of the communications as in Arpanet [13]), but also as a *real-time scheduler*. The MH for each module will have to decide (based on task, message, and communication channel priorities) what is the most urgent thread of control to resume. Naturally, the structure of an MH depends on the system being used. For example, if all the tasks of one module executed on a uniprocessor, then the MH would have to decide if it was more important to let the current task continue or to unblock a blocked task. One possibility for giving the user control of the MH is to use a rule-based system for the MH which allows an application programmer to provide the dynamic decision rules. Horizontal communications between two tasks involve having a message traverse the links from the source MH to the destination MH (naturally the best route would be chosen). By providing multiple links to every module, reliability is achieved.

We have briefly summarized the results of our previous work contained in [16]. Particularly, we have stressed the two types of communications: vertical (centralized) and horizontal (decentralized) communications. A more complete justification and examples of our ideas are contained in that work, as well as several other facets not discussed here (e.g., task creation and destruction).

III. Port-Directed Communications

As is often the case, there is a trade-off in complexity between the data structuring facilities and program complexity without the data structuring facilities. We have found that structuring the communication channels, as if they were a data type, leads to simpler programs. The basis for our intertask communications is ports [17]. We first discuss ports before proposing in Section IV the communication primitives that use the ports.

Although such structuring introduces an additional hidden overhead, we choose to use ports to achieve this structuring due to many advantages including

- accessing a port does not require the program to be dependent on the existence of a task (thus fault tolerance is improved since communications can be redirected by moving the end of a port);
- communications are structured into channels that are declared by the user (this is easier to use than direct naming, allows for more reliable and fault-tolerant computing, and lowers the number of needed primitives);
- the ports can be tailored to individual needs, providing the benefits of both one- and two-way naming.

One task declares the port and is said to *own* the port. The other tasks desiring to *use* the port must declare this intent in their specification sections (e.g., [18] employs a *use* statement in CELL). The declaration section of a port is allowed to include restrictions to tailor the port to individual needs. The primary benefits are 1) the declaration of ports allows for an adaptive communication system, 2) a smaller set of primitives can be used, and 3) interfacing different modules is easier. Note that ports are *logical* channels; the physical communication channels depend on the underlying implementation.

In the most general case, there are many users and one owner for each port. The number of users can, theoretically, be unbounded but is limited by the size of the memory buffers allocated. Bytes are sent between the users and the owner in free format, and it is the responsibility of the primitives that access the port to ensure compatibility. One of the primary values, however, is that when a port is declared, restrictions [17] can be included to configure the port to certain specifications. Restrictions can be placed on either the user end or owner end of the port, i.e., *port user restrictions* or *port owner restrictions.* Our proposed port restrictions are the following.

1) Message Format Restriction: This restricts the messages at compile time to a declared format. The owner and users of a port declare the message format that the port can handle, which would then be tested for compatibility at load time. The format could be a record or a typed formal parameter scheme as in Ada. The advantages of this restriction are that accidental misuse can be flagged at compile time, the declaration shows how the port is used, and the run-time mode is more efficient. Further, an underlying implementation may fix the packet size (i.e., 32 bytes in Thoth [8] and the V-System [5]), and this restriction allows compile-time warning of an inefficient size message, i.e., one requiring multiple packets.

2) Message Direction Restriction: By restricting the direction of messages through a port, incorrect local usage can be flagged at compile time, incorrect global usage[3] can be checked when a task is loaded, and the intertask communication structure is easily observable. Each end of a port is declared as a SEND or RECEIVE port.

3) Port User List Restriction: This is a port owner restriction that allows the owner to restrict the set of possible users. The advantages are 1) it is possible to create ports between only two tasks instead of the current many-to-one semantics, and 2) an efficient run-time implementation is possible, requiring only load-time verification for conflicts.

4) Number of Active Users Restriction: This is similar to the port user list restriction, except instead we limit the number of *active* communicating users of the port. The rationale behind this restriction is that it limits the run-time message buffer space permitting static buffer allocation instead of dynamic. As in the prior restriction, an error results if a conflict is detected during loading.

5) Port Filter Restriction: A filter is a concurrently executing task that intercepts, processes, and relays messages. A filter is placed on either the user end, the owner end, or both, as if the port was cut into two pieces, with the filter spliced in. The filter task declares the port along with the restrictions. Primitives in the filter referencing the port cause the messages to be transferred between the filter and the other module (or vice versa). To communicate with the module that declares the port and filter, the primitives in the filter reference the predefined port name FILTER. For example, suppose module M owns a port P with a filter F as a restriction. Then in filter F, primitives addressing P will communicate with a user of port P, while primitives addressing FILTER will communicate with module M. An example filter will be a bounded buffer used to simulate a nonblocking SEND. If all of a port's messages needed to be passed through the same filter, then the filter is placed on the owner's end. Likewise, if a particular user needs its own filter, then it would be placed at the user's end. The filter tasks raise exceptions when necessary, invoking handlers in either the filter or the task using the port.

[3] Both a user and the owner of a port may accidentally declare its "opening" as an input end of the port. Since we are allowing separate compilation, this cannot be flagged until the tasks are loaded, even though all the communications on the port are compatible with its definition. This is incorrect global usage but is correct local usage.

6) Timed Port Restriction: Since we are dealing with a real-time system, we provide a check that messages are delivered within a time limit. A timed port restriction can be placed at both the user and owner end. If either the one-way message or two-way rendezvous (depending on the primitives) is not completed by the designated time, then the operating system raises a timeout exception in the originating task.

7) Port Priorities: Port priorities are used to resolve queueing conflicts. A single port priority declared by the owner is sufficient. The owner end priority is used to determine the highest priority nonempty port for nondeterministic constructs. We could also allow user end priorities for an additional degree of flexibility, but we find the overhead of this approach is not justified.

These restrictions provide the user an easy way of tailoring and adjusting the communication channels the programs use. Rather than requiring inline code that fixes the communications to a task, the code fixes the communications to a port.

IV. Communication Primitives Suitable for an IMRS

Designing a set of primitives for a language is a difficult task. The primitives should be general enough to solve a broad class of problems on many architectures, yet at the same time provide efficient reliable structured elegant solutions to them. CSP, DP, and Ada have vastly different semantics, and although each language can be used to solve virtually any concurrent application, wide variations exist in their elegance and efficiency [20]. Our work is geared toward IMRS processes which can be categorized under the five process classes of Section II and [16]. These five classes are broad enough to include almost all manufacturing processes. The categorization places stringent demands on the communications system, since each class has different interactions between subprocesses and actions to be taken. We will begin with the identification of communications needed for each IMRS process class which will then lead to selection of the primitives.

A. Communications Needed for Each Process Class

Independent Process: Use and update of state variables through proprietors will be the most common communications need of an independent process. Another use of state variables for independent processes is a job reporting process that performs inventory, statistics, and material-handling operations. Depending on the urgency of the communication, different methods are required. Nonblocking message passing would be used when message receipt is not time critical or mandatory. Blocking message passing would be used when the sending task could not continue until it knew for certain that the destination task had received the message (e.g., sending a status update to a database in the console room). A task that is part of an independent process may even need a response to a message before it can continue (i.e., a state variable must be changed if the task is to continue operating). These needs require message passing and remote procedure calls. Not surprisingly, these are the communication primitives needed for the furnace application [19], which can be classified as an independent process.

Loosely Coupled Process: Because the actions depend on one another, the controlling tasks are constantly sending messages between themselves regarding their actions and status. When a task reaches a point in execution where it is about to perform the next step in the subprocess, it needs to know the status of the other subprocesses. It can either look into a local database, ask the other process for its status, or ask a server task for information about the state of the process. The first approach requires message passing between tasks, the second remote procedure calls, and the third a proprietor or monitor. Because the tasks control *independent* subprocesses, synchronization points between tasks are not needed. Thus nonblocking semantics are preferred to this process class. The communications must be fast, since actions in the process are delayed while the communications are being performed. Efficiency is less of a concern here because the frequency between messages is bound by the actions of the process, which are infrequent in comparison to processor cycles.

Tightly Coupled Process: The subprocesses in this class are controlled vertically, with the child being a slave of the parent. The child should always perform an action requested by the parent immediately. The child will probably have to return a status message after each directive from the parent, so the parent can decide the next directive to give to the child. Thus a remote procedure call is sufficient. An interrupt approach would lead to a more inefficient and unnatural solution for tightly coupled processes. Since the remote procedure calls will likely be executed often, it is crucial that its implementation not entail too much overhead. Roberts *et al.* [15] suggest that this may be difficult, and that lower level primitives should be used instead.

Serialized Motion Process: This class requires one or more subprocesses to be performed before another subprocess can commence. In the simplest of cases, this class simply requires SIGNAL/WAIT synchronization primitives.[4] In more complicated cases, information would have to be conveyed between tasks, so the blocking message passing could be used. We prefer to use messages for both cases, with null messages for SIGNAL/WAIT. The only difficulty is that synchronization between more than two tasks is difficult and a primitive for this is needed.

Work-Coupled Processes: Each task will have to maintain an updated database of all the other tasks to which it is work-coupled. Thus blocking message passing is needed (premature unblocking of a task would cause problems if a crash occurred before all of the sent messages were received). As soon as one of the tasks of the work-coupled process receives the update message, the original task may unblock. Care must be taken that the update messages are properly forwarded to each task involved in the work coupling. That is, the messages will have to be sequenced so the database can be correctly updated should the messages arrive in improper order.[5]

[4] These processes are the ones handled in AL by using *events* [14].

[5] This probably would not happen because the delay between the steps in an IMRS process are much greater than the message propagation delay but should nevertheless be performed for reliability.

B. The Primitives Needed

Choosing the primitives for an IMRS is as, if not more, important as the robot interface. Using ports takes major strides towards integrating individual modules, but the primitives dictate how easy it is to perform the communication and synchronization between modules. As mentioned in the Introduction, there are many concurrent programming languages, but the usefulness of each primitive has not been proven in distributed real-time systems. As distributed systems become more popular, we expect the communications to evolve. On the basis of the discussions in Sections III and IV-A, we have selected the primitives as shown in Table II that are appropriate for an IMRS. However, we will not discuss the actual design of a robot programming language, which requires other developments such as a distributed real-time operating system, CAD/CAM interface, etc., and is expected to take several years to complete.

TABLE II
COMMUNICATION PRIMITIVES NEEDED FOR AN IMRS

Primitive	Semantics
SEND	blocking SEND
RECEIVE	blocking RECEIVE
REPLY	nonblocking REPLY
QUERY	used to asynchronously invoke statements in one task from another task; preemption may occur depending on the priorities given in the ORDER statement
RESPONSE	a block of code at the end of a task that is asynchronously invoked by queries from other tasks
ORDER	used to prioritize conditions in a task
WAITFOR	multiple-task synchronization and communications

Primitives SEND, RECEIVE, and REPLY are used for both blocking and nonblocking message passing (see [8] for a good discussion on these primitives). The semantics are straightforward, as are their implementations. If task A issues a SEND to task B via a port, then task A will remain blocked until it has received a REPLY from task B. Task B executes a RECEIVE on a port. If task B executes its RECEIVE before the task A's SEND has occurred, it becomes blocked. Task A remains blocked until a REPLY is executed by task B, thus every SEND–RECEIVE sequence requires a REPLY to unblock tasks. The REPLY is nonblocking because task B knows that task A is already blocked at a SEND, thus when the REPLY is executed, task B does not need to block. Two-way naming (CSP) can be attained by using a port user restriction. On the other hand, one-way naming (DP, Ada) can be attained by using a port without user restrictions. Nonblocking semantics are attained via a bounded-buffer port filter. An advantage of these primitives is that the protocol is a two-way message transfer so remote procedure calls are effectively simulated, and the work done by Birrell and Nelson in creating reliable communications is applicable [3].

An efficient implementation of SEND–RECEIVE–REPLY is not difficult. By using queues for tasks blocked at a SEND or RECEIVE, tasks are removed from the active task pool and busy waiting is avoided. Using ports introduces additional run-time overhead (due to the extra level of indirection), but the implementation is not any more complex than the implementation discussed by Roberts *et al.* [15]. Roberts *et al.* also discuss why busy waiting might be preferred over queues (which involve context switches when implemented on a uniprocessor). They state that context switches are more expensive than busy waiting when the communications are significantly more frequent than the computations. Except in the tightly coupled processes of an IMRS, the intertask communications will occur relatively infrequently in comparison to the computations (i.e., at natural intervals in the IMRS process, which are few and far between). Thus ways need to be investigated to allow busy waiting for primitives using ports in a vertically controlled tightly coupled process. One possibility is to create a *process type restriction,* that allows the user to specify the process class in the port. The code generated for a port could then use the process type restriction to optimize the produced code. There are, of course, other ways to cause a compiler to produce different code (e.g., metacommands), and the advantages of each must be examined.

The QUERY, RESPONSE, and ORDER statements are used to allow one task to interrupt another task. When a task needs information from another task, it queries the other task through a port. This is similar to an exception being raised in Ada or PL/I, except it happens across task boundaries. This cannot be simulated by using multiple tasks, because tasks cannot share common variables. The appropriate RESPONSE handler at the other end of the port is then executed. Two differences between the QUERY–RESPONSE mechanism and Ada exceptions are 1) Ada does not allow parameters to be passed, and 2) after an exception handler has executed, control does not continue from the interrupted point. The QUERY is thus similar to a remote procedure call, except it preempts the current thread of control. The QUERY causes the RESPONSE to be raised in the task that owns the port Portname, provided the user is doing the QUERY. Alternatively, but less useful, the owner could execute the QUERY and one of the users would be interrupted. (A parent could query its children to check their status.)

A technical problem with the QUERY–RESPONSE is that in a real-time system, a more urgent operation should not be interrupted by a QUERY. Silberschatz [18] has proposed an ORDER statement, which is remotely similar to what we need. His ORDER statement is used in CELL to specify the priorities of threads of execution as they become unblocked. The ORDER statement is essentially a directive to a user programmable scheduler and contains a list of the different sections of a task arranged according to their priorities; a preemption requested by a QUERY will occur depending on the ORDER. The sections of a task that appear in the ORDER statement are the RESPONSE handlers, procedures, functions, and background code. This gives the programmer real-time control over the different sections of a task, which is needed in an IMRS and likely to be needed in other process control systems.

The last primitive is the WAITFOR primitive and is needed to allow more than two tasks to synchronize and communicate. Consider, for example, how to perform three-way synchronization and communication with the other primitives. One approach is to have one task issue two consecutive RECEIVES. The other two tasks would then issue SENDS to this task via a port. This simple solution unfortunately has flaws. First, the asymmetry allows communications only between the sending tasks and the receiving task. Even though three tasks are synchronized, the two sending tasks cannot directly communicate. Secondly, the solution is not very safe, since accidental misuse could easily occur if the wrong task entered the three-way synchronization by performing a SEND. Thirdly, the source code in all three tasks does not make clear what is really intended. Finally, this method is inefficient as the number of tasks grows. The problem is that the SEND–RECEIVE is designed for a two-way rendezvous only. The WAITFOR primitive is our proposed primitive to perform n-way rendezvous.

A call to WAITFOR includes a message, a function name, and a list of the tasks with which to synchronize. The semantics are as follows. When a task executes a WAITFOR, it remains blocked until all the tasks named in its WAITFOR list have executed a WAITFOR. When a set of tasks unblock because their WAITFOR list becomes satisfied, the named function in each WAITFOR would be executed. When the function is completed, execution of the task continues after the WAITFOR. The functions would have read access to all the messages pooled by the tasks involved in the synchronization via the WAITFOR. The rationale behind having these functions is that each task will have to respond differently according to the messages. The function would be written by the user and would return a single message by operating on the pooled messages. To be correctly used, if task A executes a WAITFOR, it should not be allowed to either unblock other tasks yet remain blocked or unblock itself yet have a task on one of the unblocking tasks' WAITFOR lists still remain blocked. Since it is too costly to ensure this feasible at run-time, the user is made responsible for avoiding deadlock and ensure correct usage.[6]

Note that this is not a language primitive but a system call that provides an easy-to-use method of multitask communications and synchronization. Further, note that since many tasks are involved in a symmetrical rendezvous, ports are not applicable, so the WAITFOR does not use ports. To implement the WAITFOR, a message will have to be sent to every processor that contains a task in its WAITFOR list. One message would originate and be relayed among the necessary processors. Except for an unavoidable framing window, the synchronization occurs simultaneously. Once again, it is intended that each task unblocking by another task's execution of a WAITFOR is named in all the WAITFOR's of the unblocking tasks. That is, each unblocking task has identical WAITFOR lists. To require this would need run-time testing, and thus the looser semantics are preferred.

How should we handle nondeterminism and dequeueing of messages? To obtain nondeterminism, Ada's SELECT statement is preferred. We do not really want complete nondeterminism in an IMRS, since we must always be able to predict what will occur in a given situation. Thus if more than one SELECT alternative is open (i.e., ready to communicate), we choose the message in FIFO fashion from the highest priority port. (See the port priority discussion in Section III.) Silberschatz [17] prefers complete nondeterminism in dequeuing messages from a port. This will not work in a real-time system. Alternatively, Gentleman [8] proposes that port priorities can be simulated by using RECEIVE-specific messages (two-way naming), or by using an additional task to receive the message. These alternatives can be used but lead to more unstructured solutions. The queueing and dequeueing should be handled by a systematic set of rules, not by burdening the application programmer. If ports do not have a priority, they are given a default priority lower than any user-specifiable priorities for ports. This scheme will cost slightly more to implement than nonpriority ports because the run-time efficiency can be spared at a cost of extra storage by appropriately using pointers into multiple linked lists.

[6] It may even be possible to define a predefined array or record of task names. Rather than giving a list of task names to WAITFOR, the record could be given. This could speed run-time efficiency and may help debugging.

V. Backbone for Programming Language for an IMRS

In this section we combine all our notions by proposing a new robot programming language, called LIMS (Language for Integrated Multirobot Systems), that is necessary to program an IMRS. LIMS will have similarities with Ada and AL, yet neither of these languages provide the features we need. Modifying either of these to our needs would cause more confusion than simply extracting the needed features. Our presentation of LIMS is incomplete, omitting details not pertinent to the IMRS intertask communications or module architecture. The presentation is broken into two subsections, the first deals with the module structure, and the second with the communication primitives.

A. The Module Structure

LIMS provides three distinct program units, *modules, tasks,* and *subprograms* which are hierarchically arranged. An IMRS consists of several large processes, which can be recursively divided into many subprocesses. As discussed in Section II and [16], this recursive subdivision of processes leads to a treelike structure. Eventually the leaf nodes are reached, which correspond to indivisible subprocesses. The physical process hierarchy now yields way to the software hierarchy. Each of the nodes in the process tree will be controlled by *modules,* which are executing in parallel. A module will consist of several concurrent tasks, each of which can contain subprograms (i.e., procedures and functions). We only discuss modules and tasks here, since these are the concurrently executing entities. The subprogram unit is identical to the Ada subprogram unit (see [7, ch. 6]).

Modules and tasks are very similar to each other and resemble Ada tasks. Each will contain two components, a *specification* and a *body*. In the grammars that follows, an item in braces $\{\cdot\}$ can be used zero or more times, an item in brackets $[\cdot]$ is optional (i.e., can appear zero or one time). Statements within two dashes (--) are comments. A module specification is given in Table III and the body of a module will take on the form as shown in Table IV.

The module specification declares 1) the tasks that are created when the module is created (i.e., all the bodies begin execution concurrently), 2) the response handlers, 3) the ports that it owns, along with the needed restrictions, and 4) the ports that it uses, along with the needed restrictions. Each declared response handler must have a corresponding handler

TABLE III
A MODULE SPECIFICATION

```
module_spec ::=          module mod_id
                             [is
                             {dec_option;}
                         end [mod_id]];
dec_option ::=           task task_id
                         |response resp_id [param_list]
                         |port port_id param_list {owneroption}
                         |useport port_id param_list {useroption}
param_list ::=           almost equivalent to Ada formal_part [DoD82]
                          --we allow NULL parameter lists.--
owneroption ::=          usage= usages
                         |#users= integer_const
                         |userlist= (task_or_mod {; task_or_mod})
                         |filter= task_id
                         |timeout= numeric_const timeunit
                         |priority= integer_const
useroption ::=           usage= usages
                         |filter= task_id
                         |timeout= numeric_const timeunit
usages ::=               send | receive | query | response
task_or_mod ::=          task_id | mod_id
timeunit ::=             msec | sec
process_type ::= INDEP | LOOSE | TIGHT | SERIAL | WORK
```

TABLE IV
A MODULE BODY

```
module_body ::=          module body mod_id is
                         [declarative_part]
                         [hardware
                                hardware_decl
                                {hardware_decl}]
                         [work_schedule
                                 work_step
                                 {work_step}]
                         [order_statement]
                         begin
                                sequence_of_statements
                         [response
                                 response_handler
                                 {response_handler}]
                         [exception
                                exception_handler
                                {exception_handler}]
                         end [mod_id];
hardware_decl ::=        --Implementation-dependent, these declarations
                         will contain information concerning physical
                         devices, I/O channels, etc. This is similar to
                         PEARL's divisions [STEU84] and AML's defio
                         [IBM81].--
work_step ::=            integer_const | subprogram_call {,
                         subprogram_call};
order_statement ::=      (priority_id {; priority_id})
priority_id ::=          mod_id | resp_id | excep_id | subprogram_id
response_handler ::=     when port_id.resp_id {|port_id.resp_id}
                         [actual_param_list] =>
                                 sequence_of_statements
exception_handler ::=    when excep_id {|excep_id} param_list =>
                                 sequence_of_statements
```

in the module body, otherwise a load error will result. The port owner and user options are the restrictions discussed in Section III. The *usage* (message direction) restriction cannot use the Ada modes *in, out,* and *in out* because they do not match uniquely to our primitives. Thus we must use modes which correspond to our communication primitives that use on the ports. When an owner declares a usage restriction, the owner can only access the port via the primitive named. A port user can also declare a usage restriction. By declaring usage restrictions, it is easy to examine the communications taking place through the port, and compile time checking can be done to make sure each port is correctly used. When the tasks are loaded, compatibility between the usages can be checked just once. Note that a task that issues a RECEIVE on a port must eventually issue a REPLY on the same port to complete the protocol. Also note that the userlist restriction syntax allows either a task or module to be named. This is because the two are identical as far as the communications go.

A module body is similar to an Ada task body, the only differences being that after the declaration section we include two divisions and an ORDER statement, and before the exception handlers there are response handlers.

The first division is a *hardware division*, which allows the programmer to define the process dependencies for the program. Examples might be the existence of a gripper switch, force sensor, or vision system. We leave this unspecified, for this is implementation dependent, i.e., a welding system will have one set of types or verbs, while an assembly system will have a different set.

We also provide a *work schedule division*. The philosophy behind this is to place all the statements that modify the process environment into one section at the beginning of the module body. By doing this, it is easy to examine and modify the function of a module and process. The control logic for the process would be included in the module's sequence_of_statements, but the actual work in the process would be performed by executing the next step in the work schedule via a PERFORM primitive. This is valuable when the steps of a process can be statically determined and are expressible in such a schedule. If this can be done, then a work schedule division can make the programming easier. If this cannot be done, then the standard approach of mixing computations with process control steps must be used.

The ORDER statement indicates the urgencies of each section of code. This is used when there is more than one legal thread of control in a module (e.g., several active response or exception handlers). The mod_id must correspond to the name of the module, and each named identifier must exist. The first identifier is given priority 1 (the highest priority), the second priority 2, and so on. This scheme requires all the background code for the module have the same priority as well as each entry block with the same name. We prefer this to the alternatives of providing a priority at the location of the definition of each prioritizable region or prioritizing according to labels. Our scheme allows easy comparison and modification of relative priorities. These priorities are not to be confused with task priorities. Task priorities are used to specify which task gets control if the tasks are executing on a

TABLE V
SYNTAX OF THE COMMUNICATION PRIMITIVES

send_command :: =	send port_id [actual_param_list]
receive_command :: =	receive port_id (runtime_tid, parameter {, parameter})
reply_command :: =	reply port_id (runtime_tid, parameter {, parameter})
query_command :: =	query port_id.resp_id [actual_param_list]
waitfor_command :: =	identifier :: = waitfor(function_id; task(or module)_id {, task_id}; parameter {, parameter})
getmess_command :: =	getmess(module_id)
actual_param_list :: =	equivalent to Ada's actual_parameter_part

uniprocessor. The ORDER priorities indicate when a QUERY should be handled.

A response handler looks similar to an exception handler, except it provides a parameter list. One restriction must be placed on response handlers: they cannot change the value of a local variable. This restriction is placed to avoid erroneous results that could arise if a queried response alters the value of a local variable that was being used when the interrupt occurred. Hence the name "query" is given when accessing a RESPONSE handler, for the handler can QUERY a local variable but cannot change it.

A task in LIMS is defined almost exactly the same way as a module. The major differences are:

1) instead of the keyword *module* beginning the specification and body, the keyword *task* is used;
2) a task specification cannot declare another task as a dec_option.

The rationale behind this design is that an indivisible subprocess is being controlled by the module. This subprocess may require things to be done in parallel, so we allow concurrent tasks. To be able to view the structure of the subprocess and module, we should be able to locate easily a subprocess function in terms of its controlling program, i.e., module. If tasks could create other tasks, this would not be the case.

We have omitted many needed primitives from our definitions given here, for example, communication primitives, renaming declarations, representation clauses, and USE and WITH statements (to facilitate separate compilation). Variants of these and other primitives will have to be introduced to make this a complete language, but we only discuss the issues pertinent to the module architecture and intertask communications structure here.

Note that we omitted discussion concerning task and module priorities. Our work is based on the assumption that tasks and modules execute in true parallelism. With this assumption, task and module priorities are not needed. However, if each task does not execute on a dedicated processor, then true parallelism is unattainable, and priorities will have to be given. Besides, task and module priorities may not be independent of port priorities. A low priority task may need to send a crucial emergency message through a high priority port that preempts a task of higher priority. The message handler discussed in [16] and summarized in Section II will have to know how to resolve these conflicts arising from task priorities and message priorities.

The tasks of a module begin execution automatically when the module is created. In [16] the need for a COSTART primitive was discussed. The costart would be responsible for creating modules, establishing the link between the logical and physical communication channels, and allowing dynamic specification of ports (as opposed to statically declaring them as we have discussed here).

B. The Communication Primitives

We now are ready to discuss the syntax of the communication primitives as shown in Table V.

In the commands of Table V, the parameter lists are of the standard Ada form. The parameter lists for the RECEIVE and REPLY are identical to actual_parameter_list, except they must begin with a run time task identifier. Since the SEND-RECEIVE-REPLY sequence can allow more than one REPLY to be pending for the same RECEIVE, one must be able to identify the task whose SEND was just processed to REPLY correctly to it at a later time. This is similar to those in Thoth [8] and the V Kernel [5]. The semantics for the SEND, RECEIVE, and REPLY have already been discussed, and the syntax is easily understood.

The QUERY command behaves like a remote procedure call. The task executing the QUERY is blocked until the response handler has completed executing. The only difference between the QUERY and a remote procedure call is that preemption that takes place with the QUERY command.

The WAITFOR command has a syntax which allows the specified semantics, but a few final points must be made. The function that is executed when the synchronization is complete must return the same type as the identifier. The task_id's or module_id's are task or module names, not the run time task identifiers used in the RECEIVE and REPLY commands. When the function is executing, it must be able to access each message pooled by each task. We propose the GETMESS command to achieve this. The function can execute the GETMESS command giving a task_id. This will set the parameters given in the function definition to the parameters given by the designated task. The parameters are read-only to avoid errors if several tasks share a common database of messages on a single processor. By repeatedly performing GETMESS's, a function can correctly build a single response for the task executing the WAITFOR. The only disadvantage with this approach is that problems arise if each task pools a different size message. The function would then have to know the exact format of each message pooled by each task. A simple solution is to have the

GETMESS set a predefined record and size variable. The variable would hold the number of parameters pooled by the named task in the GETMESS. The record would hold the value and type of each parameter. We do not feel this extra power is warranted for our needs, and that requiring each task to pool the same format message is sufficient.

Using these primitives with port-directed communications as we have described here will yield a powerful communication system. Our work clearly provides one-to-one and many-to-one communication schemes. By reversing the roles of an owner and its users, the owner can send messages, or query response handlers in a one-to-many fashion. However, to be consistent, this one-to-many will still only send the message, or invoke the response handler in just one of the users. The user to take part in the communication is chosen nondeterministically. If we want true one-to-many semantics (i.e., a broadcast), we can adapt our scheme as follows. Define two new usages, SENDALL and CALLALL in addition to the already existing. Only a port owner can name these usages as a restriction, so a slight modification of the grammar will be required. When a port owner performs a SEND to a port declared as SENDALL, the SEND will be sent to all its users. Upon the first REPLY being sent back, the owner will unblock. This is almost identical to Cheriton's work with the V Kernel [5], except he lumped tasks into groups. Our method unfortunately calls for many ports to be declared, but other advantages result (i.e., not having to keep track of group id's, and restricting messages to different sets of users only requires one new statically declared port, as opposed to having many group id's). The semantics concerning multiple replies can be handled identically to Cheriton's approach. By combining his notions with our ports and restrictions, it is possible to attain powerful communications capabilities with only having to execute one primitive in the source program.

VI. Conclusion and Discussion

In this paper we have explored the various communication demands brought about by five different types of processes: independent, loosely coupled, tightly coupled, serialized motion, and work-coupled processes. To support the module architecture proposed in [16], we have developed a set of communications and synchronization primitives needed for an IMRS, and a concurrent language syntax using the selected primitives based on port-directed communications. The development is based on both the distinct complex nature of an IMRS and our knowledge of the existing concurrent languages.

However, our current accomplishment is just a beginning towards the final goal of developing a complete IMRS. Some of the remaining work includes the following.

- A complete operating system kernel must be developed. The V System [5] has three major components, the interprocess communications (IPC), the kernel server, and the device server. Our discussion on the communication primitives is similar to Cheriton's IPC. We have only tackled one third of the work involved in designing a complete system like the V system, the kernel and device servers still need to be designed. This 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 software.
- The message primitives must be determined and how to process messages based on task priorities, message urgencies, and time limits and ports.
- An IMRS programming language must be designed which allows simple efficient and reliable programming of the IMRS processes. Creating a simple robot programming language that can be used by people of different experience (that also allows the power of an IMRS) will be quite challenging.
- Complete timing analysis with various IMRS implementations must be performed. There are many factors to consider for the timing analysis; for example, time for monitoring the IMRS environment, processing sensory data, and deciding an optimal action to take, time for communications, queueing, code execution, and even time for actuation. All these components must be considered together using both analytical models and simulation tools.

Undoubtedly, the IMRS will play a significant role in future robotics and automation, leading to improvement of both manufacturing productivity and robot safety. We feel that the communication structure presented in this paper along with the module architecture in [16] should form a good foundation for developing such an IMRS.

References

[1] G. R. Andrews and F. B. Schneider, "Concepts and notations for concurrent programming," *ACM Computing Surveys,* vol. 15, pp. 3–43, Mar. 1983.

[2] A. D. Brown, "Using communications standards to link factory automation systems," *Machine Design,* pp. 123–126, Aug. 23, 1984.

[3] A. Birrell and B. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.,* vol. 2, pp. 38–59, Feb. 1984.

[4] S. Bonner and K. G. Shin, "A comparative study of robot languages," *Computer,* vol. 15, pp. 82–96, Dec. 1982.

[5] D. R. Cheriton, "The V Kernel: A software base for distributed systems," *IEEE Software,* pp. 19–42, Apr. 1984.

[6] *Software Product Description: VAXELN Toolkit,* Digital Equipment Corporation, Version 2.0, SPD 28.02.02, May 1985.

[7] US Dep. of Defense, *Reference Manual for the Ada Programming Language,* July 1982.

[8] W. M. Gentleman, "Message passing between sequential processes: The reply primitive and the administrator concept," *Software Practice Experience,* vol. 11, pp. 435–466, 1981.

[9] P. B. Hansen, *The Architecture of Concurrent Programs.* Englewood Cliffs, NJ: Prentice-Hall, 1977.

[10] ——, "Distributed processes: A concurrent programming concept," *Commun. Ass. Comput. Mach.,* vol. 21, pp. 934–941, Nov. 1978.

[11] C. A. R. Hoare, "Communicating sequential processes," *Commun. Ass. Comput. Mach.,* pp. 666–677, Aug. 1978.

[12] *IBM Robot System/1: AML Concepts and User's Guide,* IBM Corp., Publication GA34-0180-1, 1981.

[13] J. M. McQuillan and D. C. Walden, "The ARPA network design decisions," in *Computer Networks.* Amsterdam: The Netherlands, North-Holland, 1977, pp. 243–289.

[14] S. Mujtaba and R. Goldman, "AL user's manual," *SAIL Report,* Jan. 1979.

[15] E. S. Roberts *et al.,* "Task management in Ada—A critical evaluation for real-time multiprocessors," *Software Practice Experience,* vol. 11, pp. 1019–1051, 1981.

[16] K. G. Shin, M. E. Epstein, and R. A. Volz, "A module architecture for an integrated multi-robot system," Robot Systems Division, Center for Research and Integrated Manufacturing (CRIM), The University of

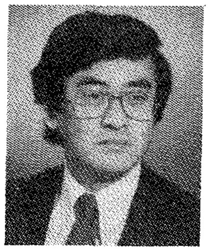
Michigan, Ann Arbor, Tech. Rep. RSD-TR-10-84, July 1984 (also in the *Proc. 18th Hawaii Int. Conf. System Sciences,* Jan. 1985, pp. 120–129).

[17] A. Silberschatz, "Port directed communication," *Comput. J.,* vol. 24, pp. 78–82, 1981.

[18] ——, "Cell: A distributed computing modularization concept," *IEEE Trans. Software Eng.,* vol. SE-10, pp. 178–185, Mar. 1984.

[19] H. U. Steusloff, "Advanced real-time languages for distributed industrial process control," *Computer,* pp. 37–46, Feb. 1984.

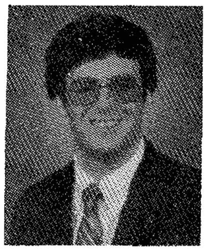
[20] J. Welsh and A. Lister, "A comparative study of task communication in Ada," *Software—Practice Experience,* vol. 11, pp. 257–290, 1981.



Kang G. Shin (S'75–M'78–SM'83) received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970, and both the M.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY, in 1976 and 1978, respectively.

From 1970 to 1972 he served in the Korean Army as an ROTC officer and from 1972 to 1974 he was on the research staff of the Korea Institute of Science and Technology, Seoul, Korea, working on the design of VHF/UHF communication systems. From 1978 to 1982 he was an Assistant Professor at Rensselaer Polytechnic Institute, Troy, NY. He was also a visiting scientist at the U.S. Airforce Flight Dynamics Laboratory in summer 1979 and at Bell Laboratories, Holmdel, NJ, in summer 1980. Since September 1982 he has been with the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, MI, where he is currently an Associate Professor. He was the Program Chairman of the 1986 IEEE Real-Time Systems Symposium and is the Guest Editor of the special issue of IEEE TRANSACTIONS ON COMPUTERS on real-time systems which is scheduled to appear in August 1987. As an initial phase of validation of architectures and analytic results, he and his students are currently building a 16-node distributed real-time system at the Real-Time Computing Laboratory (RTCL), the University of Michigan.

Professor Shin is a member of the Association for Computing Machinery, Sigma Xi, and Phi Kappa Phi.



Mark E. Epstein received the B.S. degree in electrical and computer engineering and the M.S. degree in computer, information, and control engineering from the University of Michigan, in 1983 and 1984, respectively.

He was with the Robotic Systems Software Group of the IBM Corporation, Boca Raton, FL, from 1984 to 1986, and since late 1986 he has been with the IBM T. J. Watson Research Center, Yorktown Heights, NY. His current interests include robotics, languages, vision and geometric modeling, and artificial intelligence.