COMMUNICATION PRIMITIVES FOR A DISTRIBUTED MULTI-ROBOT SYSTEM[1]

Kang G. Shin[+] and Mark E. Epstein[++]

+Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109

++Advanced Robotics Systems Group
IBM Corporation
Boca Raton, Florida 33432

ABSTRACT

An *integrated multi-robot 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.

In order to support a distributed, modular architecture of an IMRS in [SHIN84], we propose in this paper low-level *communication* and *synchronization primitives* for the IMRS. This is done by comparing and analyzing the primitives developed/proposed for general concurrent programming, and carefully examining the generic structure and interactions of IMRS processes.

1. INTRODUCTION

Conventionally, 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 or the supervision of one central task. Although almost all manufacturing processes can be accomplished using a central controller, communications bottlenecking and unreliability (that occurs at the central controller) become major problems as systems become more sophisticated. For this reason we have defined in [SHIN84] a new MRS, called *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,[2]
- the tasks are executing in true parallelism,
- it is adaptable to either centralized or decentralized control, and
- tasks may be used for controlling other machinery, sensor I/O processing, communication handling, or just plain computations.

We have already taken the first step toward solving the difficult IMRS software system, by analyzing industrial processes and examining the high-level communications needed to solve these processes [SHIN84]. We now look at the low-level communications needed. Before doing so, we first present background research in related areas.

There are numerous robot languages designed (see [BONN82] for a survey) which can control more than one robot simultaneously (i.e. AL[MUJT79], AML/V[IBM81]). They both allow the control of two robots at once, by using concurrent tasks. The tasks can be synchronized using semaphores. The principal motive behind this design was to allow two operations to be controlled at once with synchronization points to allow serialized motion process [SHIN84]. This restricts the potential amount of parallelism and limits the classes of processes that can be attained. Another problem is that it does not lend itself to integrating different machinery via a structured communications approach.

Some work has been done on distributed industrial process control (e.g. [STEU84]), but the results are not easily transportable to an IMRS. [STEU84] has described a distributed, fault-tolerant system used for controlling soaking pit furnaces. Each furnace is controlled by its own microcomputer system, and the microcomputer systems are logically *paired* so should one system fail, the paired system would control two furnaces. This system is reported to have high reliability, but the classes of parallelism involved in the furnace application are far less complex than the classes of parallelism needed in an IMRS, as are the applications that need to be coded. The action of one IMRS process could completely alter the action of many other IMRS processes, or many robots might have to work on one common process, requiring tightly-coupled communication and synchronization. An IMRS needs a more intricate communications structure to handle this complex 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. One technique used is *message passing*. Some of the languages which utilize message passing are *Distributed Processes* (DP)

[1]The work reported here is supported in part by the NSF Grant No. ECS-8409938 and the U.S. AFOSR Contract No. F49620-82-C-0089. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies.

[2] "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.

[HANS78], *Communicating Sequential Processes* (CSP) [HOAR78], and *Ada* [DoD82]. The *Monitor* is another technique used to enforce exclusive access rights to common procedures and data. *Concurrent Pascal* [HANS77] uses monitors to link different tasks. Although the communication mechanisms in these languages are different, it is claimed that for several common applications that do not involve real-time constraints, either mechanism can be used [ANDR83], and that at an abstract level, their relative powers are equal. This has been shown theoretically and by benchmarking them against standard problems. However, these languages are recent advances in computer science, and have not been amply tested in distributed environments, especially those dealing with real-time control. With GM's work on MAP [BROW84], and the Department of Defense's work with Ada [DoD82], our knowledge will soon expand in this area. For the meantime, we explore the communications primitives needed in an IMRS by looking at the process structure and module architecture [SHIN84].

Consequently, we need to select the IMRS communication and synchronization primitives that can meet the conflicting demands of an IMRS (i.e. reliable, efficient, natural, easy-to-use, powerful). In this paper we intend to meet this need by carefully examining the nature of industrial processes and the primitives existing/proposed in general concurrent programming. It should be noted that there has been little work dealing with an IMRS and its intertask communications and synchronization.

This paper is organized as follows. In Section 2 we compare and critically review the existing communication and synchronization primitives in the context of IMRS applications. In Section 3 we discuss the need for different primitives for the five classes of industrial processes identified in [SHIN84]. In Section 4 we advocate first port-directed communications and then propose the primitives most suitable for an IMRS. Section 5 concludes the paper with a discussion on how our work solves the IMRS communication primitives problem.

2. COMPARISON OF EXISTING PRIMITIVES

Stotts[STOT82] has pointed out thirteen design issues that need to be considered when designing a concurrent language. The issues most pertinent to an IMRS are task creation, task destruction, communication and synchronization mechanisms, and real-time support, and reliability. In this section, we present the concepts and issues underlying the communication and synchronization primitives, by looking at three different concurrent programming languages, *Communicating Sequential Processes* (CSP) [HOAR78], *Distributed Processes* (DP) [HANS78], and *Ada* [DoD82]. Many of the ideas discussed in this section are drawn from [WEGN83], [ANDR83], and [WELS81].

Andrews [ANDR83] classifies concurrent programming languages into three different classes, *procedure-oriented, message-oriented,* and *operation-oriented.* Procedure-oriented languages (e.g. Concurrent PASCAL, Modula, Mesa, Edison) use *monitors* as the basis for intertask communications. These are not well suited in a distributed environment because an efficient implementation requires shared variables. Message-oriented languages (e.g. CSP, Gypsy, Plits, Thoth) use **send** and **receive** primitives for communications and synchronization between tasks. An implementation does not require shared variables, and thus is appropriate for a distributed environment[GENT81]. Exclusion on a variable needed by different tasks can be achieved by using a *proprietor* task which accepts messages from different tasks in exclusive fashion. The last class, operation-oriented languages (e.g. DP, Ada), use the remote procedure call as the means for intertask communications. One task can call a procedure (i.e. entry) in another task by sending an implicit message to the called procedure. While the called procedure is executing, the calling task is blocked. When the called procedure has completed execution, a return message is sent, and the blocked task may then resume. We say that a *rendezvous* has occurred between the two tasks, when the two threads of execution have been united into one while the remote procedure call is executing. Since an implicit **send** and **receive** operation is performed in a remote procedure call, this class is a rigid structuring of the message-oriented class.

Due to the distributed nature of an IMRS and also most automated systems, it is best to use either the message-oriented or operation-oriented classes of concurrent programming [ANDR83]. These classes are appropriate because message passing between processors does not require shared memory. Message passing can also be reliably implemented. In this section we discuss issues underlying message based primitives. We use CSP, DP, and Ada for examples. These three languages have vastly different mechanisms, and will serve well.

(A) Blocking vs. nonblocking primitives

(B) Task addressing

(C) Message format

(D) Reliability and communications failure.

Each of these issues is discussed comparatively in the following subsections.

2.1. Blocking vs. Nonblocking

A communication primitive is said to be *blocking* if the task executing the primitive is halted until the communication is performed, or *nonblocking* if the communications requested by the primitive do not cause the task to halt. The blocking **send** and **receive** are the most commonly seen primitives,[3] e.g. CSP and Thoth. However, in many cases (especially in an IMRS) blocking is undesirable due to its delay in response and limitation to parallelism. To this end, slight modifications to the blocking primitives have been made in some concurrent languages. For example, Ada provides the ability to cancel the **entry call** or **accept** when blocking results, whereas CSP does not. DP uses a blocking **send** only to a remote, passive procedure (e.g. monitor-like routines that do not require a **receive**). A call is performed as soon as all the threads of control in the called task have either terminated or been suspended at a guarded region.

Advantages and disadvantages of blocking and nonblocking primitives are:

- The blocking primitives perform *implicit* synchronization, and thus no new notation is needed to provide

[3]The nonblocking send is not needed, since it can be simulated by using a bounded buffer task. The nonblocking receive is not sensible, since the task receiving the message will perform some operation based on the message. Interrupts are usually used instead of a nonblocking receive.

synchronization. However, this convenience does not come without disadvantage; namely, blocking primitives *restrict* parallelism. A task that is blocked at a **send** or **receive** cannot continue execution. The parallelism that could exist is reduced to a single thread of control.

- Blocking primitives need no message buffering, whereas nonblocking ones do. Thus, the implementation of blocking primitives need not deal with updating message buffer pointers and testing for buffer overflow. In case of nonblocking primitives, hiding the above implementation details from the user may sometimes lead to incorrect programs.

Implementation of blocking primitives is straightforward. By using operating system kernel routines and queues, a task can easily be blocked and unblocked. *Busy waiting* (spin locking) could also be used, but would only be appropriate when the communications bottleneck the computations [ROBE81].

As alternatives to nonblocking primitives, [GENT81] proposes two semantics. The first is to use a bounded buffer to simulate a nonblocking **send**. The second is to use *conditional* primitives. A conditional primitive is one that sends (or receives) the message only if the other communicator is already blocked. This is exactly what Ada uses by *conditional entry calls* and *else* clauses in a **select** construct.

The **reply** primitive in [GENT81],[CHER84] is valuable for distributed systems. The idea behind the **reply** is to require that every **send-receive** message transfer issue a **reply** from the callee back to unblock the caller. By reversing the role of **send** and **receive**, nonblocking semantics are virtually attained (i.e. *administrator* in [GENT81]). When compared to remote procedure calls, this provides additional parallelism because the caller is unblocked as soon as the critical section of code for **receive** in the called task has completed. This also can be implemented more reliably that the standard **send-receive** because every message transaction is bidirectional and implementation can assume this [BIRR84].

2.2. Task Addressing

Task addressing refers to how the tasks in a message transfer are named. CSP uses *2-way naming*; both the **receive** and **send** primitives must name the tasks to participate in the message transfer. Ada and DP both use *1-way naming*, an entry or external procedure call needs only name the task and the procedure to be executed. Ada's **accept**, as well as the common procedures in a DP task, have "open ears" to a call from any task. Advantages of each are:

- *Nondeterminism* in the receiver is attained with 1-way naming. If the called procedure were required to name a source task for input, the named source might not be ready, even though the called procedure was ready. A conditional **receive** may alleviate this problem. In such a case, a repetitive looping with a conditional **receive** for all the possible sources is needed. After the first **accept**, the **accept** body would then be executed. In the mean time, many other **entry calls** could arrive and would thus be queued. Rather than being unqueued randomly, the repetitive looping would impose ordering of the **accepts**, and nondeterminism is lost. Complete nondeterminism is a way to exploit parallelism [GENT81], but looping around conditional **receives** leads to determinism and inefficiency.
- 1-way naming facilitates the creation and updating of library routines. It would be nearly impossible to write a library routine using 2-way naming. A library routine must accept calls from any source, and this cannot easily be done if the library procedure must name each source procedure.
- 2-way naming allows a task to accept **sends** only from a particular caller. This protection may be needed if the system is being used in a malicious environment.

Other possibilities exist to the 1- and 2- way naming schemes discussed above. A *port* is a symbolic name that allows an extra level of indirection [SILB81]. Messages are sent to a port rather than a task. Typically, many tasks may send messages to a port, but only one task is allowed to receive a message from a port. Port directed communications are preferable for IMRS's for many reasons, and will be discussed further in Section 4.1.

2.3. Message Format

The message format refers to how a message is represented. The most common message format is simply a list of typed parameters. The parameters contain values which are passed between tasks, provided parameter types agree. Another message format is a string of characters, which each task may interpret differently by using a different template (i.e. variant record in PASCAL). The message format plays an important role in any distributed system, because assumptions concerning the message format allows a more efficient implementation. For this reason, most distributed systems opt for using fixed length messages. If a message is shorter than the fixed length, then dummy bytes are padded. If a message is longer, then it is broken into *packets*. transmitted a packet at a time, and then reconstructed at the destination processor. It has been shown that 32 bytes is the most appropriate size for a message [CHER84],[SHOC80]. CSP, DP, and Ada all provide messages of arbitrary length via parameters to the message primitives, but real-time systems such as IMRS's should avoid using multiple packet messages because of their potential overhead. Some systems provide additional primitives for transferring large messages. For example, Thoth provides a **.Transfer** primitive [GENT81], the V kernel provides the **CopyTo** and **CopyFrom** primitives [CHER84]. We prefer to fix the message size of each transaction.

2.4. Reliability and Communications Failure

This issue can never be avoided, yet most published works entirely sidestep this issue. This may be for several reasons. Reliability primarily depends on the implementation and is difficult to treat. Another related topic is fault tolerance, i.e. how to perform error detection and recovery. The reliability and fault tolerance issue does propagate up to the user interface, particularly in the comprehension aspect. The user must be able to invoke his own error handlers when time limits are not being met, a task dies, or the messages become corrupted. If the primitives provide the user a simple model, then it will be easier to program fault-tolerant systems. To this end, we advocate port-directed communications, see Section 4.1.

3. COMMUNICATIONS NEEDED FOR INDUSTRIAL PROCESSES

The process structure of an IMRS is hierarchical. An industrial process can be decomposed into several *subprocesses*, which are further divided, and so on. Eventually, the industrial process is divided into many indivisible subprocesses. Each of these processes is programmed with a a *module*. Each module consists of computational tasks. We define the *module architecture* as (i) the structure of a module, and (ii) the logical structure and/or communication paths that connect the modules in an IMRS (see [SHIN84] for a detailed description).

Before we propose communication and synchronization primitives, we examine the nature of communications and synchronization needed for industrial (sub)processes. This is done on the basis of the five classes of industrial processes that we have identified in [SHIN84]. In the following, we briefly reintroduce the definitions of the five classes and then discuss specific communications need of each 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 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. To no surprise, these are the communication primitives needed for the furnace application [STEU84], 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 for this process class. The communications must be quick, 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 of this process 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 [ROBE81] suggests 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 several 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 several 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 (i.e. the messages will have to be sequenced so the database can be correctly updated should the messages arrive in improper order[5]).

4. COMMUNICATION PRIMITIVES SUITABLE FOR AN IMRS

The five process classes utilize communications for different purposes. Independent processes send messages to common tasks. Loosely-coupled processes query each other or a common task for information about the process state. Tightly coupled processes use a master/slave control approach. Serialized motion processes use communications for synchronization and event signaling.

The variety of concurrent programming languages offer many primitives from which to choose the set suitable for an IMRS. In this section we select the primitives appropriate for an IMRS. Our discussion consists of two parts. The first part deals with structuring the communication channels into ports. The second part deals with the actual primitives that utilize the ports.

4.1. Ports

Ports are an alternative to the 1-way and 2-way task addressing discussed Section 2.2. A port is just a symbolic name that two tasks reference. Having messages address ports offers many advantages.

- 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.

[4] These processes are the ones handled in AL by using *events* [MUJT79].

[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.

- 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 1- and 2-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. [SILB84] 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. It should be noted 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. 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 [SILB81] 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:

(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 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 [GENT81] and the V-System [CHER84]), 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[6] can be checked when a task is loaded, and the intertask communication structure is easily observable. How this is done depends on the primitives. Ada declares the direction of parameters, since they use the remote procedure call (**accept**). Another way, the one we prefer, is to use **send-receive-reply** with the port being 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. When the port owner of users are loaded, system routines will have to test for conflicts and generate load errors if necessary.

(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 rational 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, a load error results if a conflict results.

(5) **Port Filter Restriction**: A filter is just a concurrently executing task that intercepts, processes, and relays the messages. It is as if the port was cut into two pieces, with the filter spliced in. A filter can be placed on either the user end, the owner end, or both. The filter task would declare 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 will reference the predefined port name FILTER. For example, suppose task T owns a port P with a filter F as a restriction. Then in task F, primitives addressing P will communicate with a user of port P, while primitives addressing FILTER will communicate with task T. A common filter will be a bounded buffer used to simulate a nonblocking **send**. A device driver is another use of a port filter. 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 needed its own filter, then it would be placed at the user's end. Thus the port declaration in the owner and user can each name filter tasks. The filter tasks can raise exceptions when necessary, invoking handlers in either the filter or the task using the port.

(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 would raise 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 will be sufficient. The owner end priority would be used to determine the highest priority nonempty port, for nondeterministic constructs. We could also allow user end priorities which would give a further degree of flexibility (and complexity). The overhead of this approach is not justifiable, and so we prefer a single priority per port.

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.

[6]Both a user and 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.

4.2. The Primitives Needed

Choosing the primitives for an IMRS is as, if not more, important than 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 real-time distributed systems. As distributed systems become more popular, we expect the communications to evolve. In this section we present the primitives that are appropriate for an IMRS. This is based on the discussion in Section 3.

Primitive	*Semantics*
send	blocking **send**.
receive	blocking **receive**.
reply	nonblocking **reply**.
query	Used to asynchronously invoke statments 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.

Table 1. Communication Primitives Needed For an IMRS

send, **receive**, and **reply** are used for both blocking and nonblocking message passing (see [GENT81] 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 **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. 2-way naming (CSP) can be attained by using a port user restriction. 1-way naming (DP, Ada) can be attained by using a port without user restrictions. Non-blocking semantics are attained via a bounded-buffer port filter. An advantage of these primitives is that the protocol is a 2-way message transfer so remote procedure calls are effectively simulated, and the work done by Birrell and Nelson in creating reliable communications is applicable[BIRR84].

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.* [ROBE81]. 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 of 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: (i) Ada does not allow parameters to be passed, and (ii) 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 [SILB84] has proposed an **order** statement, which is remotely similar to what we need. His **order** statment 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, The **order** statement 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

[7]However, we will not discuss the actual design of a robot programming language, which requires other developments such as a real-time distributed operating system, CAD/CAM interface, etc., and is expected to take several years to complete.

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: (i) 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. (ii) 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**. (iii) The source code in all three tasks does not make clear what is really intended. (iv) 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 become 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 rational 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 insure this feasible at run-time, the user is made responsible for avoiding deadlock and insure correct usage.[8]

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 because of another task executing a **waitfor** is named in all the **waitfors** 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 3.1.) Silberschatz [SILB81] prefers complete nondeterminism in dequeueing messages from a port. This will not work in a real-time system. Alternatively, Gentleman[GENT81] proposes that port priorities can be simulated by using receive-specific messages (2-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.

5. CONCLUSION

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.* In order to support the module architecture in [SHIN84], we have developed a set of communications and synchronization primitives needed for an IMRS. We have also completed but not included here (due to the space limitation) 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.

Undoubtedly, the IMRS will play a significant role in future robotics and automation. Integrating all the workcells and devices leads to improvement of both manufacturing productivity, reliability, and safety. We feel that the primitives presented in this paper (and the language syntax completed but not included here) along with the module architecture in [SHIN84] should form a good foundation for developing such an IMRS.

REFERENCES

[ANDR83] Andrews, G.R., and Schneider, F.B., "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, Mar. 1983, vol. 15, no. 1, pp. 3-43.

[BIRR84] Birrell, A., and Nelson, B., "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, Feb. 1984, pp. 38-59.

[BONN82] Bonner, S., and Shin, K. G., "A Comparative Study of Robot Languages," *Computer*, vol. 15, no. 12, Dec. 1982, pp. 82-96.

[BROW84] Brown, A. D., "Using Communications Standards to Link Factory Automation Systems" *Machine Design*, Aug. 1984, pp 123-126.

[8] 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.

[CHER84] Cheriton, D. R., "The V Kernel: A Software Base for Distributed Systems," *IEEE Software*, Apr. 1984, pp. 19-42.

[DoD82] U. S. Dept. of Defense, "Reference Manual for the Ada Programming Language," July 1982.

[GENT81] 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.

[HANS77] Hansen, P. B., *The Architecture of Concurrent Programs*, Prentice-Hall, Inc., 1977.

[HANS78] Hansen, P. B., "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM*, Vol.21, No. 11, Nov. 1978, pp. 934-941.

[HOAR78] Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM*, Aug. 1978, pp. 666-677.

[IBM81] IBM Corp., *IBM Robot System/1: AML Concepts and User's Guide*, Publication No. GA34-0180-1, 1981.

[MUJT79] Mujtaba, S., and Goldman, R., "AL Users' Manual," *SAIL Report*, Jan. 1979.

[ROBE81] Roberts, E. S., et. al., "Task Management in Ada - A Critical Evaluation for Real-Time Multiprocessors," *Software Practice and Experience*, vol. 11, 1981, pp. 1019-1051.

[SHIN84] Shin, K. G., Epstein, M. E., and Volz, R. A., "A Module Architecture for an Integrated Multi-Robot System," *Technical Report*, RSD-TR-10-84, Robot Systems Division, Center for Research and Integrated Manufacturing (CRIM), The University of Michigan, Ann Arbor, MI, July 1984. Also appeared in the *Proc. of the 18th Hawaii Int'l Conf. on System Sciences*, Jan. 1985.

[SHOC80] Shoch, J. F., and Hupp, J. A., "Measured Performance of an Ethernet Local Network," *Communications of the ACM*, vol. 23, no. 12, Dec. 1980, pp. 711-721.

[SILB81] Silberschatz, A., "Port Directed Communication," *The Computer Journal*, vol. 24, no. 1, 1981, pp. 78-82.

[SILB84] Silberschatz, A., "Cell: A Distributed Computing Modularization Concept," *IEEE Transactions on Software Engineering*, vol. SE-10, no.2, Mar. 1984, pp. 178-185.

[STEU84] Steusloff, H. U., "Advanced Real-Time Languages for Distributed Industrial Process Control," *Computer*, Feb. 1984, pp. 37-46.

[STOT82] Stotts, P. D. Jr., "A Comparative Study of Concurrent Programming Languages," *ACM SIGPLAN Notices*, vol. 17, no. 9, Sept. 1982, pp. 76-87.

[WEGN83] 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, Jul. 1983, pp. 446-462.

[WELS81] Welsh, J., and Lister, A., "A Comparative Study of Task Communication in Ada," *Software Practice and Experience*, vol. 11, 1981, pp. 257-290.