INTERTASK COMMUNICATIONS IN AN INTEGRATED MULTI-ROBOT SYSTEM[1]

Kang G. Shin

Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-1109

May 1985

CENTER FOR RESEARCH ON INTEGRATED MANUFACTURING

Robot Systems Division

COLLEGE OF ENGINEERING

THE UNIVERSITY OF MICHIGAN

ANN ARBOR, MICHIGAN 48109-1109

---

[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 view of ONR. A subset of this paper was presented at the 1985 IEEE International Conference on Robotics and Automation.

TABLE OF CONTENTS

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. Although the IMRS is motivated by an interesting application, it is essentially a distributed, real-time processing system with various heterogeneous processes.

In order to support a distributed, modular architecture of the IMRS, we propose in this paper low-level *communication primitives* and their *supporting language syntax* which are typical of real-time concurrent programming languages. This is done by (i) carefully examining the generic structure and interactions of IMRS processes, (ii) comparing and analyzing the primitives and syntax developed/proposed for general concurrent programming, and (iii) using port-directed communications.

We have chosen a complex chip insertion process to illustrate the selected primitives. Our solution shows the effectiveness of the IMRS communication mechanisms.

Index Terms - multi-robot system, robot languages, real-time constraints, (robot) processes, subprocesses and (computational) tasks, module architecture, concurrent programming, communications and synchronization, port-directed communications, blocking and nonblocking primitives.

1. INTRODUCTION

There are numerous advantages in using a multi-robot system(MRS) for manufacturing applications. 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[BROW84]. 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 [SHIN85] 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,
- the tasks are executing in true parallelism,

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

- 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 [SHIN85] a high-level abstraction of IMRS communications, called a *module architecture*, to support an IMRS; this will be discussed briefly in Section 2.

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 jumping into our 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.

There are numerous robot languages (see [BONN82] for a survey) which can control more than one robot simultaneously, the most advanced being AL[MUJT79]. AL allows one program to control two robots at once. By using **cobegin-coend** pairs, a programmer can initiate two pseudo-concurrent tasks. They can be synchronized using the *event* data type (integer semaphores). The principal motive behind this design was to allow cooperation via *serializing* 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 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[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. The furnace system is controlled by a real-time concurrent language called "Multicomputer PEARL." PEARL allows the transmission of information from one task to another by message passing and remote procedure calls. Each furnace is controlled by its own microcomputer system, and the microcomputer systems are logically *paired* so should one system fail, the corresponding mate computer system would control two furnaces. This system is reported to be highly fault-tolerant, having only 11 hours of down time in more than 24,000 hours of use. This figure is indeed impressive, but the classes of parallelism involved in the furnace application are far less complex than the classes of parallelism needed in an IMRS. The action of one IMRS process could completely alter the action of another IMRS process, or robots might have to work on one common process, requiring tightly-coupled communications and synchronization. Because of the more dynamic

nature of the IMRS, a more intricate, flexible communications structure is needed.

There has been a considerable amount of research in concurrent programming languages. Many languages have been created, each utilizing different primitives to allow communication and synchronization. Andrews [ANDR83] 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) [HANS78], Communicating Sequential Processes (CSP) [HOAR78], Thoth [GENT81], and Ada [DOD82]. Although each language uses different communications mechanism, it is claimed that for common concurrent benchmarks (i.e. dining philosophers problem, bounded buffer) the mechanisms of any of these languages can be used [ANDR83], 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 problems 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) IMRS's will probably be programmed by groups of people, and thus a structured interface between their respective pieces, while hiding implementation details, will be necessary.

(b) Debugging 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, but except for this, none of the other concurrent programming languages provide structured channels. Later in this paper, we use ports [SILB81] to structure the intermodule communications of an IMRS.

This paper is organized as follows. We briefly review in Section 2 the module architecture that we have developed in [SHIN85]. We advocate the port directed communications for an IMRS in Section 3. In Section 4 we identify first communication needs for each process class and then propose the primitives most suitable for an IMRS. In Section 5 we combine all notions of the module architecture and the primitives into a communication structure. Section 6 is concerned with demonstrating how the selected primitives and structure can be used by using an example IMRS which is typical of manufacturing applications. Section 7 concludes the paper with a brief mention of the remaining work needed to implement an IMRS.

2. 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 in [SHIN84]. Our development toward a module architecture began with the classification of IMRS processes, which is given in Table 1. 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.

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

Table 1 - The Four Basic Process Classes

In Table 1 table 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:

A. *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* [GENT81]).

B. *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 into 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.

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

D. *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 is an example of this.

E. *Work-Coupled Processes:* This class is not listed in Table 1 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. It is obvious that 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 [STEU84] 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 (i) the structure of a module, and (ii) 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.

Vertical communications are defined as communications between a task and any of its descendant tasks. *Simple* vertical communications are those which occur between

[2]When a task begins executing.

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 (i) the number of child tasks is small, (ii) the IMRS processes are not apt to be modified often, (iii) the parent task is very reliable, and (iv) the child tasks are not computationally intensive. However, if (i) is invalidated then communications bottlenecking occurs; if (ii) is invalidated then changes will have to be made to more than one task in the system and will result in down-time; if (iii) is invalidated then the system is vulnerable to a single failure in the parent; and if (iv) is invalidated then parallelism is not being exploited.

Horizontal communications ameliorate the IMRS significantly. Allowing tasks to communicate directly without a central controller (i) reduces the chances of a bottleneck by exchanging messages among children, (ii) keeps all the code for each subprocess local to one module, (iii) increases reliability because all the subprocesses do not rely on one central control task, and (iv) allows more parallelism because each child task is not blocked as often as in the vertical case, where 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 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 [McQU77]), 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 *expert system*[3] 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 [SHIN85]. Particularly, we have stressed that there are 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).

3. 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 [SILB81].

[3]This is purely conjecture; the development of such a system is not a subject of this paper.

We first discuss ports before proposing in Section 4 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 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[4] 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

[4]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.

(**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.

4. 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, there are wide variations in their elegance and efficiency [WELS81]. Our work is geared toward IMRS processes which can be categorized under the five process classes of Section 2 and [SHIN85]. 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 need for each IMRS process class which will then lead to selection of the primitives.

4.1. 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 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.[5] 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).[6]

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 discussions in Sections 3 and 4.1.

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

[6]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.

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 2. 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)

[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.

can be attained by using a port user restriction. 1-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 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 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.) 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

[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

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. 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 an Integrated Multi-Robot 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.

5.1. 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 [SHIN85], this recursive subdivision of processes leads to a tree-like 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 ideally 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 Chapter 6 of the Ada Reference Manual [DoD82]).

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 follow, an item in "{}" can be used zero or more times, an item in "[ ]" is optional (i.e., can appear zero or one time). Statements within two "--" are comments. A module specification is as follows.

---

debugging.

```
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 3. A Module Specification.

The body of a module will take on the following form.

```
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
```

Table 4. A Module Body.

The module specification declares (i) the tasks that are created when the module is created (i.e., all the bodies begin execution concurrently), (ii) the response handlers, (iii) the ports that it owns, along with the needed restrictions, and (iv) the ports that it uses, along with the needed restrictions. Each declared response handler must have a corresponding handler in the module body, otherwise a load error will result. The port owner and user options are the restrictions discussed in Section 3. 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 [STEU84] 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, i.e., 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 (i.e., 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, etc. 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 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 rational 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. In order to be able to view the structure of the subprocess and module, we should be able to easily locate 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. Additionally, 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 [SHIN85] and summarized in Section 2 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 [SHIN85] we explain the need for a **costart** primitive. 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).

5.2. The Communication Primitives

We now are ready to discuss the 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 *actual_parameter_part* [DoD82]

Table 5. Syntax of the Communication Primitives.

In the above commands, 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 in order to correctly **reply** to it at a later time. This is similar to Thoth [GENT81] and the V Kernel [CHER84]. 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 four 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 [CHER84], except he lumped tasks into groups. Our method unfortunately calls for many ports to be declared, but other advantages results (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.

6. EXAMPLE IMRS

In this section we demonstrate how *LIMS*, coupled with our previous work in [SHIN85], can be used to program a complex IMRS easily, and with a high degree of reliability. The example process does not include all the five process classes, nor does its solution include all the communication primitives, but it does illustrate many of the salient features of our port directed communications. Before we begin with the example, we should mention that we "idealize" the capabilities of robots. That is, our process uses robots for chores that would be more effectively, inexpensively, and reliably solved with other equipment. This is because we wish to focus on the intertask com-

munications, and not on detailed device drivers.

The IMRS we use is the three workcell chip insertion assembly line. The first workcell is simply a relay that release pallets onto a roller conveyor.[9] The second workcell uses two robots to anchor a blank computer card to the pallet. The first robot moves a card to the pallet, then the second robot screws the card onto the pallet. The third workcell consists of a single robot that inserts various chips onto the card. To complicate the last workcell, assume that a similar assembly line exists next to this line, and that the last robots of each line share a common set of feeders. This reduces the hardware costs and the space requirements.

This IMRS consists of three independent processes. The first involves no subprocesses. The second consists of two subprocesses that make a serialized motion process. The third involves no subprocesses, but forms a loosely coupled process with another process for the other assembly line. We thus need five modules, gate, load_it, and chip_it which correspond to the controlling modules for the three workcells, and load_card and screw_in_card which are the two modules for the two subprocesses of load_it. Therefore, we expect to need the following communication channels (we omit the code needed for the higher levels of the process tree):

(1) A port that is owned by gate that receives messages from load_it and chip_it containing information about their speeds.

(2) Another port owned by gate that communicates with a task in its module that releases a pallet onto the conveyor.

[9]A roller conveyor uses metal rollers instead of a rubber mat to move the cargo. The advantage of this is that cargo can be stopped at one workcell by a gate while the rollers can be left on to keep cargo moving on the other sections of the conveyor.

(3) Two ports owned by load_it that issues status request messages to its child processes (controlled by the modules load_card and screw_in_card).

(4) A port owned by load_card that communicates between it and a device driver that freezes the pallet in a known location in the workspace.

(5) Another port owned by load_card which sends a message to screw_in_card when a card is in place.

(6) A port owned by chip_it that communications between it and a device driver which freezes that pallet with the card in a known location in the workspace.

(7) Another port owned by chip_it that it uses to ask the other robot for permission to enter the critical region about the feeders. (Likewise, this task also uses a port that the other robot's module owns which handles permissions coming from the other module).

A detailed code of this example IMRS is given in the Appendix, where we use a notation similar to Ada [DoD82]. Code that is omitted is described in braces ("{}"). We also omit procedures not crucial to the intertask communications (the hardware dependent device drivers).

There are many interesting features of the solution which warrant special attention. These include:

(1) Bounded buffers can be utilized to simulate nonblocking **send**'s. Notice that the approach of using a filter task does not complicate either end of the port. Both GATE and the two users LOAD_IT and CHIP_IT are ignorant to the internal actions of BOUNDED_BUFFER. Because BOUNDED_BUFFER is a filter, being spliced onto a port, some mechanism must be used to indicate from which end of

the splice each communication comes. The built-in port name FILTER is the mechanism preferred. The communications via FILTER go to the port owner. Correct usage between the owner, users, and filter can still be verified at load time. Each primitive using FILTER must complement the usage declared by the port owner, and the usage declared by the filter must complement the usage declared by each user.

(2) There is valuable redundancy in the timeout error exceptions. The failure of any module or task will be caught eventually by a timeout, and correct action can be taken. This could even include killing a task or module and creating a new copy.

(3) When device drivers are used, we prefer to restrict the number of users rather than the one user via a USERLIST. This allows accidental usage, but makes it easier to kill a copy of the device driver should a hardware error occur, which can often happen when relays are trying to freeze a moving pallet.

(4) The response handlers include a port name as well as the response name. This is to lower the amount of work that is done at run time. By doing this, we can compare compatibilities at load time.

(5) We have used a simple work scheduler in CHIP_IT. The work scheduler will choose the step with the highest priority (the lowest number). Since all the steps are given priority 1, a step will be chosen arbitrarily. Each statement of the step is then executed. By using this, it is trivial to change a chip number or a chip location. We do not have to go digging into separate files, complicated record structures, or into the task body. We simply change the appropriate step. Depending on the design of the work scheduler, GRASP, and INSERT, return codes could be allowed which determined if a step were successfully performed.

Thus if a feeder became empty, we could signal the operator and continue with another step.

(6) There is a complex **order** statement in CHIP_IT. When we were originally programming CHIP_IT (while also considering CHIP_IT2), we tried to guarantee exclusion of the critical region by using different **order** statements in CHIP_IT and CHIP_IT2. After several hours of frustration we realized it could not be done. This is because of the following. CRIT_SECTION must be ahead of the background code (either CHIP_IT or CHIP_IT2) in at least one of two modules, otherwise both tasks could simultaneously **query** to the other module, thus causing a deadlock. But if CRIT_SECTION is ahead of the background code in one of the modules, than one module could get permission to enter the critical section, and then because of an operating system context switch[10] remain hung in its background code. Meanwhile the other module could continue to execute, ask permission to enter the critical section (which would be granted because the other module still had not entered the GRASP procedure), and then a collision in the critical region could occur. This is unavoidable because once a task is given permission, it can lose execution cycles. Thus the **order** statement should not be used to try to perform mutual exclusion, and should be used as it was originally intended, to force real-time priorities. This is how we use it in CHIP_IT (and CHIP_IT2). A **query** can only interrupt the current thread when the robot is performing an insertion. Our solution can introduce a deadlock, however, because each module could **query** the other simultaneously (or any time after the INSERT but before the next GRASP). Thus we have to break the deadlock. This requires

[10]We want our solution to work on uniprocessors, a network of multiprocessors, or a network of uniprocessors.

asymmetrical timeout exception handlers, but the deadlock can be broken. Note that this deadlock will occur the first time through each module, but as soon as it is broken, each task will immediately get permission to enter the GRASP routine as soon as the other module enters the INSERT routine. By the time the other module gets done with the INSERT routine, the current module will be either in GRASP or INSERT, and in either case permission to enter GRASP will be granted (because INSERT has lower priority than the critical section and a call to INSERT follows a call to GRASP). Depending on how the arrival of the cards are synchronized, deadlock could occur on each pass through the modules. Certainly a more sophisticated synchronization scheme could be designed, but this would depend on how often the deadlock was occurring, and how accurate the deadlock timeout exception handler is. If the deadlock occurs often, or if a more complicated scheme is needed to implement an n-way critical region, the simplest approach is to create a monitor task with the existing primitives.

(7) LOAD_IT uses two parts, ASK_CHILD1 and ASK_CHILD2, to check on the status of its children LOAD_CARD and SCREW_IN_CARD. Instead of this, we can use one part, which both child modules would use. Then a **query** by the parent, LOAD_IT, would nondeterministically select one of the children to use. The only problem with this is that the selection may not be fair, and the same child could get queried each time. If an implementation were known to be fair, then this technique would not cause such a problem.

7. 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.* In order to support the module architecture in [SHIN84], we have developed (i) a set of communications and synchronization primitives needed for an IMRS, and (ii) 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:

- Developing a complete operating system kernel. The V System [CHER84] 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.
- Determining the message primitives 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 MRS 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.

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 [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*, March 1983, Vol. 15 No. 1, pp. 3-43.

[BROW84] Brown, A. D., "Using Communications Standards to link Factory Automation Systems", *Machine Design*, p. 123-126, August 23, 1984.

[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.

[BERN80] Bernstein, A. J., "Output Guards and Nondeterminism in Communicating Sequential Processes," *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 2, April 1980, pp. 234-238.

(1) Bourne, S. R., *The UNIX System*, Addison-Wesley, 1982.

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

(3) U. S. Dept. of Defense, "Reference Manual for the Ada Programming Language," July 1982.

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

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

(6) Gini, G. and Gini, M., "ADA: A Language for Robot Programming?," *Computers in Industry*, Vol. 3, No. 4, 1983, pp. 253-259.

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

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

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

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

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

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

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

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

(15) 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.

(16) Schoeffler, J. D., "Distributed Computer Systems for Industrial Process Control," *Computer*, Feb. 1984, pp. 11-18.

(17) 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 Robotics and Integrated Manufacturing (CRIM), The University of Michigan, Ann Arbor, MI, July 1984. Also appeared in the *Proc. 18th Hawaii Int'l Conf. on System Sciences*, Jan. 1985, pp. 120-129.

(18) 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.

(19) Silberschatz, A., "Port Directed Communication," *The Computer Journal*, Vol. 24, No. 1, 1981, pp. 78-82.

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

(21) Steusloff, H. U., "Advanced Real-Time Languages for Distributed Industrial Process Control," *Computer*, Feb. 1984, pp. 37-46.

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

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

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

APPENDIX: CODE OF AN EXAMPLE IMRS

```
-- Gate checks to see if there is room on the conveyor for another
-- pallet. It does so every n/2 seconds, where n is the faster of
-- load_it and chip_it. This guarantees that each cell will always
-- be busy, without requiring gate to do busy waiting.
--
MODULE GATE IS
        TASK BOUNDED_BUFFER;    -- Used to simulate a nonblocking
                                -- port for messages from chip_it and load_it
        TASK RELEASE;           -- Used to release a pallet onto the conveyor.
                                -- It does so only if the line is not backed
                                -- up (perhaps by looking at a photocell).
                                -- A RC=0 indicates a pallet was released.
        PORT FREQUENCY(TASK_NUM:INTEGER;TIME_FOR_ONE:REAL)
                                -- Receive cycle time messages.  TASK_NUM=0
                                -- for load_it, 1 for chip_it.
                USAGE=SEND-- Must ask BOUNDED_BUFFER for the next queued
                                -- message.  TASK_NUM=-1 implies empty buffer
                FILTER=BOUNDED_BUFFER

                USERLIST=(LOAD_IT,CHIP_IT);

        PORT LET_1_GO(RC:INTEGER)
                                -- Send a message to RELEASE to release a pallet
                USAGE=SEND
                #USERS=1        -- Should RELEASE fail we can start up
                                -- another task with a different name.
                TIMEOUT=2 sec;  -- If we don't get a response within 2 seconds,
                                -- then invoke an exception handler.
END GATE;


MODULE BODY GATE IS
        CYCLE_TIME,             -- Holds the delay time between pallets
        TEMP_TIME: REAL;        -- Holds the time from one message
        SPEEDS:ARRAY(0..1) OF REAL; -- Holds the estimated cycle times for
                                -- load_it (0) and chip_it (1)
        RC,                     -- The return code
        NUM_RELEASED,           -- Used for inventory purposes
        TASK_NUM;               -- Either 0 or 1 for a message.

        ORDER(LET_1_GO.TIMEOUT,GATE) -- Let a timeout interrupt the background code

BEGIN
        CYCLE_TIME := 10.0; -- Initialize variables
        SPEEDS(0)  := 20.0;
        SPEEDS(1)  := 40.0;

-- Loop indefinitely!!

        LOOP
        DELAY CYCLE_TIME ;      -- Delay so as to not kill cycles
```

```
        SEND LET_1_GO(RC); -- Let the next pallet go and get the return code

        IF RC=0 THEN NUM_RELEASED := NUM_RELEASED+1;
                ELSE {signal for help}

        -- See if a message is waiting in the bounded buffer, and if so, update
        -- CYCLE_TIME.

        SEND FREQUENCY(TASK_NUM,TEMP_TIME);

        IF TASK_NUM >= 0 THEN
                SPEEDS(TASK_NUM) := TEMP_TIME;  -- Update the correct speed.
                CYCLE_TIME=MIN(SPEEDS(0),SPEEDS(1))/2.0 ;
        END IF;

        END LOOP;

EXCEPTION

        WHEN LET_1_GO.TIMEOUT =>
                {Signal for help. Perhaps reset timer, perhaps kill
                 and restart RELEASE}

END GATE;


TASK BOUNDED_BUFFER IS -- Hold 20 messages. Do not test for overflow.
        PORT FREQUENCY(TASK_NUM:INTEGER; TIME_FOR_ONE:REAL)
                        -- A filter task also declares the port. The only
                        -- difference is that the USAGES may be different,
                        -- as is done here. The USAGE for the filter must
                        -- be compatible with those of the USERs.
                USAGE=RECEIVE     -- Receive messages from users
                FILTER=BOUNDED_BUFFER
                        -- By declaring this, the compiler knows this then
                        -- is the filter task, and thus the predefined port
                        -- FILTER is legal
                USERLIST=(CHIP_IT,LOAD_IT);
END BOUNDED_BUFFER;


TASK BODY BOUNDED_BUFFER IS
        TYPE MESSAGE IS                 -- The message type
                RECORD
                T_NUM : INTEGER ;  -- 0 for load_it, 1 for chip_it
                SPEED   : REAL ;        -- Its speed
                END RECORD;

        BUFFER : ARRAY(0..19) OF MESSAGE; -- The buffer of messages
        HEAD,                           -- The head of the queue
        TAIL,                           -- The tail of the queue
        TASK_ID,                        -- The runtime id for a task.
        TASK_NUM: INTEGER;              -- 0 for load_it, 1 for chip_it

        TEMP_TIME : REAL;  -- The cycle time for either load_it or chip_it
```

```
BEGIN
        -- Loop Indefinitely!

        LOOP
        SELECT

        -- Receive a message if one is available and there is space for it

        WHEN (HEAD+1) MOD 20 <> TAIL =>
                RECEIVE FREQUENCY(TASK_ID,TASK_NUM,TEMP_TIME);

                -- Load into buffer

                HEAD := (HEAD+1) MOD 20; -- Increment HEAD
                BUFFER(HEAD).TNUM := TASK_NUM;
                BUFFER(HEAD).SPEED := TEMP_TIME;

        OR

        -- See if GATE is asking for a value

        RECEIVE FILTER(TASK_ID,TASK_NUM,TEMP_TIME);
                                        -- Use FILTER to designate communication with
                                        -- the port owner.
                -- Generate response

                IF HEAD = TAIL THEN REPLY FILTER(TASK_ID,-1,-1);
                ELSE
        REPLY FILTER(TASK_ID,BUFFER(TAIL).TNUM,BUFFER(TAIL).SPEED);
                TAIL := (TAIL+1) MOD 20 ;
                END IF;

        END SELECT;
        END LOOP;
END BOUNDED_BUFFER;


-- LOAD_IT just checks the status of its children every 4 seconds,
-- and once in a while sends an averaged cycle time message to
-- GATE.

MODULE LOAD_IT IS
        USE PORT FREQUENCY(TASK_NUM:INTEGER; TIME_FOR_ONE:REAL)
                        USAGE=SEND;
        PORT ASK_CHILD1(RC,NUM_DONE:INTEGER; TIME_FOR_ONE:REAL)
                                                -- Ask load_card for its status
                USAGE=QUERY                     -- Preempt the child if necessary
                USERLIST=(LOAD_CARD)            -- Only allow LOAD_CARD to use
                TIMEOUT=2 SEC;                  -- We better get a response, else the
                                                -- child could be dead.
        PORT ASK_CHILD2(RC:INTEGER)
                                                -- Ask screw_in_card for its status
                USAGE=QUERY                     -- Preempt the child if necessary
                USERLIST=(SCREW_IN_CARD)        -- Only allow screw_in_card to use
                TIMEOUT=2 SEC;                  -- We better get a response, else the
```

```
                                        -- child could be dead.
END LOAD_IT;

MODULE BODY LOAD_IT IS
        NUM_LOADED,                 --Used for inventory
        RC: INTEGER;            -- Return code
        CYCLE_TIME : REAL; -- The cycle time, according to load_card

        ORDER(ASK_CHILD1.TIMEOUT,ASK_CHILD2.TIMEOUT,LOAD_IT);
                                    -- Allow exceptions to preempt task
BEGIN
        COSTART(LOAD_CARD,SCREW_IN_CARD); -- Start other modules

        -- Loop Indefinitely!

        LOOP
        DELAY 4.0;                  -- Delay 4 seconds

      -- See If load_card is still ok

        QUERY ASK_CHILD1.STATUS(RC,NUM_LOADED,CYCLE_TIME);

        IF RC <> 0 THEN {Signal for Help}

        -- See if screw_in_card is still ok

        QUERY(ASK_CHILD2.STATUS(RC)

        IF RC <> 0 THEN {Signal for help}

      {Average many cycle_times and once in a while do
              SEND FREQUENCY(0,CYCLE_TIME); }
        END LOOP;

EXCEPTION
        WHEN ASK_CHILD1.TIMEOUT | ASK_CHILD2.TIMEOUT =>
              {Signal for help and take correct action}

END LOAD_IT;


MODULE LOAD_CARD IS
        TASK FREEZE_PALLET ;            -- Stops pallet when in correct place
                                        -- and sends message via PALLET_IN_PLACE
        USE PORT ASK_CHILD1(RC,NUM_DONE:INTEGER; TIME_FOR_ONE:REAL)
              USAGE=RESPONSE;  -- Must have a response handler

        -- PALLET_IN_PLACE is a signal from the driver FREEZE_PALLET that
        -- the next pallet is ready.

        PORT PALLET_IN_PLACE() #USERS=1     -- Allow only one driver
              TIMEOUT=30 SEC                -- Error if no pallet at all
              USAGE=RECEIVE;                -- Expects a message

        --CARD_IN_PLACE signals SCREW_IN_CARD that it can begin
```

```
        PORT CARD_IN_PLACE(RC:INTEGER)
                USAGE=SEND
                USERLIST=(SCREW_IN_CARD)
                TIMEOUT=30 SEC;          -- Error if no REPLY within 30 seconds
END LOAD_CARD;


MODULE BODY LOAD_CARD IS
        NUM_LOADED,                      -- Used for inventory
        TASK_ID,                         -- Used for REPLY, the runtime TASK id
        RC : INTEGER;                    -- A Return code

        CYCLE_TIME: REAL;  -- Time for the last cycle

        -- Allow preemptions

        ORDER(PALLET_IN_PLACE.TIMEOUT,ASK_CHILD1.STATUS,LOAD_CARD);
BEGIN
        NUM_LOADED := 0;

        -- Loop Indefinitely

        LOOP
        {start timer}
        {grab next card and move to ready position}

        -- Wait for signal that we can proceed, don't send REPLY until
        -- Pallet can be released.

        RECEIVE PALLET_IN_PLACE(TASK_ID);

        {move card to pallet}

        -- Signal SCREW_IN_CARD that the card is ready

        SEND CARD_IN_PLACE(RC);          -- IF RC=0 Then it was successfully anchored

        IF RC <>0 THEN {Signal for HELP}
                ELSE
                {stop timer}
                {compute cycle time and store in CYCLE_TIME}
                REPLY PALLET_IN_PLACE(TASK_ID);       -- Release pallet
                NUM_LOADED := NUM_LOADED+1;
                END IF;
        END LOOP;

-- Response Handler for Parental Query

RESPONSE
        WHEN ASK_CHILD1.STATUS(RC1,NUM1:INTEGER;CYCLE1:REAL) =>
                RC1    := 0;
                NUM1   := NUM_LOADED;
                CYCLE1 := CYCLE_TIME;
EXCEPTION
        WHEN PALLET_IN_PLACE.TIMEOUT | CARD_IN_PLACE.TIMEOUT =>
```

```
                {Signal for help}
END LOAD_CARD;


-- SCREW_IN_CARD awaits for a signal on CARD_IN_PLACE and then screws
-- the card to the pallet

MODULE SCREW_IN_CARD IS
        USE PORT ASK_CHILD2(RC:INTEGER) USAGE=RESPONSE; -- Status requests by load_it
        USE PORT CARD_IN_PLACE(RC:INTEGER) USAGE=RECEIVE
                TIMEOUT=30 SEC;             -- If no signal every 30 seconds then error
END SCREW_IN_CARD;


MODULE BODY SCREW_IN_CARD IS
        TASK_ID,                -- A run time TASK id for REPLY
        RC : INTEGER;           -- A return code

        -- Allow preemptions

        ORDER(CARD_IN_PLACE.TIMEOUT,ASK_CHILD2.STATUS,SCREW_IN_CARD);

BEGIN
        --Loop Indefinitely

        LOOP
 {Prepare to screw in next card}

        RECEIVE CARD_IN_PLACE(TASK_ID,RC);

        {screw card in, set RC=0 if OK, <>0 if error}

        REPLY CARD_IN_PLACE(TASK_ID,RC);
        END LOOP;

RESPONSE
        WHEN ASK_CHILD2.STATUS(RC1:INTEGER) =>
                RC1 := 0;
EXCEPTION
        WHEN CARD_IN_PLACE.TIMEOUT =>
                {Signal for help}
END SCREW_IN_CARD;


-- CHIP_IT inserts the chips onto the card.  It uses work scheduling
-- to perform this.  The work scheduler uses three procedures, REINSTATE
-- which initializes the internal variables, PERFORM which performs the
-- highest priority (lowest number) step not yet done, and MORE, which
-- returns the number of steps remaining.  We only postulated the usefulness
-- for work scheduling, and this is our attempt to show its value.
-- The locations and chip numbers can be easily changed, because they
-- are not embedded deeply inside the code, and the order of the insertion
-- is not predetermined, so an empty feeder still allows the performing
-- of other steps.
```

```
MODULE CHIP_IT IS
 TASK FREEZE_CARD;                          -- Stops the card
      PORT READY_TO_CHIP()                  -- FREEZE_CARD sends us a signal in this por
           #USERS=1                         -- Allow only 1 driver
           TIMEOUT=60 SEC                   -- Expect a response every minute
           USAGE=RECEIVE;                   -- We receive the signal
      PORT SHARE_FEEDER(CLEAR:INTEGER)      -- Ask the other process for permission
                                            -- to enter critical region, 0 means its OK
           USAGE=QUERY                      -- Preempt other task if it is inserting a chip
           USERLIST=(CHIP_IT2)              -- The other module
           TIMEOUT=3 SEC;                   -- Not only for module failure, but if both
                                            -- modules simultaneously query, than a deadlo
                                            -- occurs and this breaks the deadlock. A
                                            -- deadlock should rarely happen however.
      -- Port to allow CHIP_IT2 to ask us for permission
      -- CHIP_IT2 has a USE PORT identical to this except for SHARE_FEEDER

      USE PORT SHARE_FEEDER1(CLEAR:INTEGER) USAGE=RESPONSE;

      USE PORT FREQUENCY(TASK_NUM:INTEGER; TIME_FOR_ONE:REAL); -- Send

END CHIP_IT;


MODULE BODY CHIP_IT IS
      OK,                               -- OK to enter critical section (GRASP)
      STEPS: INTEGER;                   -- The number of work steps remaining

      CYCLE_TIME : REAL; -- The time to insert all the chips

WORK SCHEDULE
      -- Give all the steps equal priority, so they can be
      -- executed in arbitrary order

      1)    GRASP(1),INSERT(X,Y,Z);           -- One chip to be inserted
      1)    GRASP(4),INSERT(X1,Y1,Z1);  -- Another chip to be inserted
--    .
--    .
--    .
      1)    GRASP(12),INSERT(XN,YN,ZN);       -- The last chip on the card

      -- Use the following order -
      --    1) Allow timeouts the highest priority
      --    2) Don't allow the background code to be preempted,
      --       else a QUERY could be interrupted by a QUERY from CHIP_IT2
      --    3) Don't allow the critical region to be interrupted
      --    4) THEN allow preemption for a QUERY from CHIP_IT2
      --    5) The lowest priority is the insertion code. THUS a QUERY
      --       will only be answered when a robot has left the critical region.

      ORDER(READY_TO_CHIP.TIMEOUT,
            SHARE_FEEDER.TIMEOUT,
            CHIP_IT,
            GRASP,                      -- A subprogram whose code is omitted. Simply
                                        -- enters the critical region, grasps a chip
```

```
                                        -- from feeder N, and exits critical region.
                SHARE_FEEDER1.CRIT_SECTION,
                INSERT);                -- A subprogram whose code is omitted.  Simply
                                        -- inserts a chip at X,Y,Z.
BEGIN
        -- Loop Indefinitely

        LOOP
        {Start timer}
        RECEIVE READY_TO_CHIP(TASK_ID);         --Get signal to begin

        REINSTATE;                      -- Initialize work scheduler

        --Loop until no more steps to perform

        MORE(STEPS);
        WHILE STEPS <> 0
                LOOP
                QUERY SHARE_FEEDER.CRIT_SECTION(OK);        --Ask permission to enter
                                                            -- critical region
                IF OK <> 0 THEN PERFORM;
                ELSE {Signal operator}
                END IF;
                MORE(STEPS);                -- See if any more steps
                END LOOP;

        {stop timer, update and average cycle time, and once in a while do
                SEND FREQUENCY(1,CYCLE_TIME); }

        END LOOP;

-- Because of order statement, always give permission to enter critical section

RESPONSE
        WHEN SHARE_FEEDER1.CRIT_SECTION(CLEAR:INTEGER) =>
                CLEAR := 1;

-- card never arrived timeout.

EXCEPTION
        WHEN READY_TO_CHIP.TIMEOUT =>
                {Signal for help}

-- Share feeder timeout.  This could be because of a critical failure
-- in CHIP_IT2 or an unlucky deadlock occurrence.  There are many ways to
-- break the deadlock.  One would be to call a procedure which had a lower
-- priority than SHARE_FEEDER1.CRIT_SECTION.  If CHIP_IT2's timeout
-- than simply kept retrying the operation, eventually CRIT_SECTION
-- would have the higher priority than the low priority procedure,
-- and CHIP_IT2 would be given permission to enter the critical region.

END CHIP_IT;
```

q