Communication and Synchronization in Distributed Systems

ABRAHAM SILBERSCHATZ

Abstract—Recent advances in technology have made the construction of general-purpose systems out of many small independent microprocessors feasible. One of the issues concerning distributed systems is the question of appropriate language constructs for the handling of communication and synchronization. In his paper, "Communicating sequential processes," Hoare has suggested the use of the input and output constructs and Dijkstra's guarded commands to handle these two issues. This paper examines Hoare's concepts in greater detail by concentrating on the following two issues:

1) allowing both input and output commands to appear in guards,
2) simple abstract implementation of the input and output constructs.

In the process of examining these two issues we develop a framework for the design of appropriate communication and synchronization facilities for distributed systems.

Index Terms—Communication, distributed systems, guarded commands, input/output commands, programming languages, synchronization.

I. INTRODUCTION

In his paper, "Communicating sequential processes," [1] (CSP), Hoare has introduced a language concept for concurrent processing which is suitable for a microcomputer network environment with distributed storage. Central to the language are the following concepts.

1) A CSP program consists of a fixed number of sequential processes that are mutually disjoint in address spaces.
2) Communication and synchronization are accomplished through the input and output constructs.
3) The sequential control structures are based on Dijkstra's guarded commands [2].

We now briefly elaborate on points 2) and 3) above.

Communication in CSP occurs when one process names another as destination for output and the second process names the first as source for input. When this happens, the output values are copied from the first process to the second. Transfer of information occurs only when both the source and destination processes have invoked the output and input commands, respectively. This implies that either the source or the destination process may be suspended until the other process is ready with the corresponding output or input.

Thus the I/O facility serves both as a communication mechanism and a synchronization tool.

In CSP sequential control is accomplished through the use of Dijkstra's guarded commands [2]. A guarded command has the form:

\[ <\text{guard}> \rightarrow <\text{command-list}> \]

A guard consists of a list of declarations, Boolean expressions, and an input command (each of these is optional). A guard fails if any of its Boolean expressions have the value False, or if the process named in its input command has terminated. If a guard fails then the guarded command has no effect; control is transferred to the statement following that command. If a guard does not fail, then the command-list is executed. This takes place only after the input command (if present) has been completed.

Guarded commands may be combined into an alternative command that has the form:

\[ [G_1 \rightarrow C_1 \sqcap G_2 \rightarrow C_2 \sqcap \cdots \sqcap G_n \rightarrow C_n] \]

An alternative command specifies execution of one of its constituent guarded commands. Consequently, if all guards fail, the alternative command fails. If more than one guarded command can be executed successfully, an arbitrary one is selected for execution.

Alternative commands can be executed as many times as possible via the use of the repetitive command that has the form:

\[ *[G_1 \rightarrow C_1 \sqcap G_2 \rightarrow C_2 \sqcap \cdots \sqcap G_n \rightarrow C_n] \]

The alternative command is executed repeatedly as long as it does not fail. When the alternative command fails because all of its guards fail, then the repetitive command terminates and control is transferred to the following statement.

Hoare emphasized in his paper that his proposal is at best only a partial solution to the problem of finding an appropriate language for concurrent programming. One of the reservations that Hoare has had about his proposal concerns the restriction that only input commands may appear in guards. Hoare has presented two arguments in favor of removing this restriction.

1) This will allow some useful simplification of certain programs, particularly in the case where subroutines are called by result.
2) This will ensure that the externally visible effect and

Manuscript received February 26, 1979; revised May 16, 1979. This work was partially supported by the National Science Foundation under Grant MCS 7702463.

The author is with the Department of Mathematical Sciences, University of Texas at Dallas, Richardson, TX 75080.
behavior of every parallel command can be modeled by some sequential command.

It is the aim of this paper to examine the ramifications of allowing output commands to appear in guards. In particular, we concentrate on the question of whether such a modification to CSP will impend upon the possibility of obtaining a simple and efficient implementation of the input and output constructs. In the process of doing so we develop a framework for the design of appropriate communication and synchronization facilities for distributed systems.

II. Master-Slave Communication Partners

In order to examine the issues of I/O commands implementation, and the feasibility of allowing output commands in guards, it is necessary to examine the complexity issue concerning synchronization that is inherent in a communication scheme.

When processes wish to communicate, some information about the state of each process must be exchanged in the course of executing an I/O command in order to determine when and whether communication can take place. A state information exchange may be viewed as a signal which carries no information relevant to a particular program, but is needed for synchronization purpose. When looking for a simple implementation of the I/O commands one seeks an algorithm that will reduce the number of such synchronization signals. Moreover, the signals should not result in deadlocks which could not otherwise occur.

In CSP one can obtain a simple uniform implementation of the I/O commands by taking advantage of the asymmetry of the input and output constructs. This asymmetry is the result of Hoare’s requirement that only input commands may appear in guards. Thus, if one chooses a rule that for each pair of communicating processes the process attempting input (the destination process) always waits for a request from the process attempting output (the source process), a simple algorithm for handling the I/O commands can be obtained which is both efficient and deadlock free. In particular, guarded commands can be handled very effectively, because of the fact that a process executing a guard always waits for the initiation of I/O requests by some other processes. With such a scheme the effective signaling that must take place between a source and destination process to establish the fact that both are ready to proceed is minimal. An example of a possible abstract implementation of this scheme can be found in [3].

If output commands are to be permitted to appear in guards the above proposed simple rule for the handling of the I/O commands cannot be immediately effective. This is because a source process always initiates a request and it is suspended until the time the corresponding destination process has completed the transfer of I/O. Thus no gain is achieved by allowing output commands to appear in guards. One way to resolve this difficulty is to require both the source and destination processes to exchange signals in order to determine their respective states. In this case, the implementation will not be simple; moreover, it is prone to deadlocks [4], [5].

What we have pointed out is that if one wishes to allow both input and output commands to appear in guards, one must sacrifice the simple implementation discussed above. We thus wish to examine whether one can reasonably restrict the type of programs that can be written in CSP so that both symmetry and simple implementation can be obtained. In the following we propose such a restriction and argue that it is indeed reasonable.

We propose to replace the asymmetry of the input and output constructs by another kind of asymmetry; namely, by defining some ordering relation between the system processes. Suppose that we define the notion of a master-slave relation between communicating processes. That is, for each pair of processes $P_1$ and $P_2$ that can communicate, we will consider one of these processes as being the master of the other. We also require that for each such pair there exists a unique single master. The requirement of a unique master implies that the master-slave relation is antisymmetric. We avoid for the moment explaining the criteria for deciding how master-slave relations are formed.

The motivation for having the master-slave model with the above requirement is as follows.

1) One can obtain a simple uniform deadlock-free implementation of the I/O commands. In particular, one may choose a rule in which the slave of each pair of communicating processes always waits for a request for an I/O service. This simple rule will be adopted in this paper and an abstract implementation of the I/O commands using this rule will be presented in the next section.

2) With such a uniform implementation one can allow both input and output commands to appear in guards. This, however, is restricted to the following case. A slave can have I/O commands in guards, but these commands may involve only those processes which are its masters.

One should at this point in time examine our proposal in greater detail to justify the suggested restrictions. We will do so by demonstrating that the modified CSP language is at least as powerful as language supporting processes, classes (abstract data types), and monitors hierarchically organized (e.g., Concurrent Pascal [6], Modula [7], etc.). Furthermore, since no suspension takes place after an output message has been delivered to a lower level process, one can conclude that it is indeed more powerful than those above-mentioned languages.

Consider a program written in Concurrent Pascal. Each module in this program can be implemented as a CSP process. The access-graph of the program (as defined by Brinch Hansen [6]) must be acyclic. Thus if one considers the arcs of the access-graph to be master-slave relations rather than access-rights, then our requirement that the master-slave relation be antisymmetric is preserved in this system. It is simple to show that the restrictions imposed on I/O commands in guards will allow the simulation of monitors and condition queues [6], [8]. One method for simulating these constructs was presented in [9]. Thus we have informally substantiated our claim.

Let us illustrate our concepts by an example. Suppose that M is the name of a CSP process that simulates a monitor, with $U_1$ and $U_2$ the names of its users. Central to M is the guarded
command:

\[
\begin{align*}
\text{X} &::= \text{B}(0..9) \text{ portion;} \\
\text{In}, \text{Out} &::= \text{integer}; \text{In} := 0; \text{Out} := 0; \\
\text{*}[\text{In} < \text{Out} + 10; \text{Producer}] \text{ B (In} \mod 10) \rightarrow \text{In} := \text{In} + 1 \\
\text{□[Out < In; Consumer]} \text{ B (Out} \mod 10) \rightarrow \text{Out} := \text{Out} + 1
\end{align*}
\]

for handling procedure calls, where \( U_1(\ ) \) and \( U_2(\ ) \) are optional depending on whether a result value is expected.

The user processes (e.g., \( U_1 \) and \( U_2 \)) simulate procedure calls to \( M \) by the sequence:

\[
M(\ ) \cdots M(\ )
\]

If only input commands are allowed to appear in guards the above sequence must be observed even when no output is required from the user process to the simulated monitor. In this case \( M(\ ) \) is nothing more than a pure signal to avoid the loss of efficiency. This could be avoided if output commands were allowed in guards. With our approach this is possible. For example, Hoare's bounded buffer could be implemented as follows:

\[
\text{X} ::= \text{B} ::= (0..9) \text{ portion; } \\
\text{In}, \text{Out} ::= \text{integer}; \text{In} := 0; \text{Out} := 0; \\
\text{*}[\text{In} < \text{Out} + 10; \text{Producer}] \text{ B (In} \mod 10) \rightarrow \text{In} := \text{In} + 1 \\
\text{□[Out < In; Consumer]} \text{ B (Out} \mod 10) \rightarrow \text{Out} := \text{Out} + 1
\]

In this example, the process \( X \) is a slave of both the Producer and Consumer processes. Therefore, it can have both input and output commands in guards involving these processes (e.g., Consumer and Producer). It is interesting to note that our solution to the bounded buffer problem is quite analogous to the solution using conventional monitors. This is precisely what Hoare has commented about in Section 7.8 of his paper.

One final note concerning I/O commands in guards should be made at this point. All the examples we have studied thus far, including all those presented by Hoare [1] obey our requirement that a slave can only have I/O commands in guards involving its masters. In particular, we note that the example of Section 7.8:

\[
\text{Z} ::= \text{X!2} \rightarrow \text{Y!3} \quad \text{□} \quad \text{Y!3} \rightarrow \text{X!2}
\]

obeys our requirement if we consider \( Z \) to be the slave of processes \( X \) and \( Y \). This example was furnished by Hoare to illustrate one of the reasons for having output commands in guards; namely, to ensure that the externally visible effect of a parallel command can be reproduced by a sequential command.

III. ABSTRACT IMPLEMENTATION

In this section we illustrate our concepts by presenting a simple abstract implementation of the I/O commands. This abstract implementation will be independent of any assumption concerning architecture and will be mainly used to illustrate our concepts. In order to concentrate on the main issues of communication and synchronization, we will restrict our attention here to a simplified version of Hoare's constructs; namely, that processes can exchange only fixed size messages. This will relieve us from the burden of handling pattern-matching, an issue which is not directly related to the issues of communication and synchronization.

We also do not discuss here the question as to how information (e.g., message, signal) is transferred from one process to another. This can be handled either by copying within a common store or by input/output between separate stores. Thus we will assume here that the system supports some form of primitives to allow the transfer of information from one process to another (as will be explained below). The reason for ignoring this problem is that it will allow us to obtain some general protocols for handling communication independent of any particular architecture.

The approach we take in presenting the abstract implementation will be to discuss these issues in terms of masters and slaves. We adopt the rule that a slave always waits for an I/O request from one of its masters. The presentation will be done in terms of Pascal notations [10]. The enumeration of all the masters of each process is accomplished via the scalar type declaration:

\[
\text{type Mymasters} = \langle \text{list of all the masters of a process} \rangle;
\]

For each process, say \( P \), the compiler needs to declare certain data structures to allow master processes to notify their intention for communication. In addition, some internal buffering may be required for handling the transfer of messages from one process to another. The following data structures are declared locally to process \( P \) by the compiler:

\[
\begin{align*}
\text{var Read} &::= \text{array [Mymasters] of Boolean}; \\
\text{Write} &::= \text{array [Mymasters] of Boolean}; \\
\text{Flag} &::= \text{Boolean}; \\
\text{Buffer} &::= \text{Message};
\end{align*}
\]

These data structures will be used in the implementation of the input and output commands. We would like to point out again that these structures are mainly intended to illustrate our concepts of communication and synchronization. They may be implemented as memory locations, hardware registers, hardware buffers, etc.

The system provides the following primitives for handling the transfer of data:

1. \( \text{Put}(Q,M) \equiv \text{transfer message} \ M \text{ to the Buffer of process} \ Q \).
2. \( \text{Get}(Q,M) \equiv \text{transfer content of Buffer of process} \ Q \text{ to} \ M \).
3. \( \text{Signal}(Q,F) \equiv \text{set the Boolean variable} \ F \text{ of process} \ Q \text{ to the value True} \) (where \( F \) is either \text{Read[ ], Write[ ], or Flag}).
4. \( \text{Wait}(F) \equiv \text{wait until the value of the Boolean variable} \ F \text{ is True} \). The waiting can be carried out by either busy waiting (if the process runs on a dedicated processor), or by process suspension (in case the processor is multiplexed).

We emphasize again that for the purpose of this paper we ignore the problem of how these primitives are handled. This clearly depends on the particular architecture. For example, if a processor can only transfer 32 bits at a time (hardware...
limitation), then the Put and Get primitives may have to be implemented as a sequence of 32 bits word transfer rather than one single transfer.

We are in a position now to describe how the input and output commands are handled. We will do so by describing their implementation in a slave process and a master process, respectively. Let P and Q be a pair of communicating processes such that P is the slave of Q. The translation of the input and output commands in P and Q is presented below:

<table>
<thead>
<tr>
<th>P (slave of Q)</th>
<th>Q (master of P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait(Read[Q])</td>
<td>Buffer := b</td>
</tr>
<tr>
<td>Get(Q,a)</td>
<td>Signal(P, Read[Q])</td>
</tr>
<tr>
<td>Read[Q] := false</td>
<td>Wait(Flag)</td>
</tr>
<tr>
<td>Signal(Q,Flag)</td>
<td>Flag := false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q!a</th>
<th>P!b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait(Write[Q])</td>
<td>Signal(P, Write[Q])</td>
</tr>
<tr>
<td>Put(Q,a)</td>
<td>Wait(Flag)</td>
</tr>
<tr>
<td>Write[Q] := false</td>
<td>b := Buffer</td>
</tr>
<tr>
<td>Signal(Q,Flag)</td>
<td>Flag := false</td>
</tr>
</tbody>
</table>

Note that the slave process always waits for the master to initiate I/O.

We describe now how I/O commands are handled in guards. Recall that we require that a slave can have I/O commands in guards, but these commands may involve only those processes that are his masters (a master cannot have I/O commands in guards involving its slaves). This implies that a process executing a guard is always waiting for the initiation of I/O requests by some other processes. We illustrate how the handling of guards is achieved by an example. Suppose that we have processes P, Q, and R such that P is the slave of both Q and R. Consider the following guarded command in process P:

\[
Q?a \rightarrow s_1  \quad \square \quad R!b \rightarrow s_2
\]

This statement would be translated by the compiler as follows:

```
while (¬Read[Q] and ¬Write[R]) do skip end;
if Read[Q] then begin
   Get(Q,a)
   Read[Q] := false
   Signal(Q,Flag)
   s_1
end
else if Write[R] then begin
   Put(R,b)
   Write[R] := false
   Signal(R,Flag)
   s_2
end
```

In this example we assume busy waiting in the while statement (in order to avoid the introduction of additional primitives to handle processor multiplexing). We also have avoided the question of nondeterminism (i.e., if both Q and R are ready to do I/O we always fulfill Q's request first).

Three comments concerning the above proposed abstract implementation should be made at this point. First, an informal argument for the correctness of this implementation can be found in [3]. Second, this implementation did not take into account the fact that processes may terminate. This can be easily handled by adding another data structure in each process. This data structure will be used by the process to record and remember which of its communicating partners have terminated [3]. Third, an abstract implementation of Hoare's original I/O constructs can be easily obtained from the above implementation. The main difference is that the initiation of an I/O request will be done by a source process rather than a master process and only two of the implemented commands (P!b, Q?a) are needed.

IV. Conclusion

We have presented a framework for the design of appropriate communication and synchronization facilities for distributed systems. We have done so by introducing the master-slave model and by proposing the rule that a slave always waits for one of its masters to initiate an I/O request. This simple rule allowed us to obtain a simple uniform deadlock-free implementation of the I/O commands. Moreover, it provided the means to include both input and output commands in guards. We have demonstrated the usefulness of these concepts by proposing an abstract implementation that is independent of architecture.

We have demonstrated that a language constraint by our model is more powerful than a language supporting the definition of processes and monitors hierarchically organized. We have also shown that our requirement concerning I/O commands in guards results in more natural (structured) algorithms.

We have avoided in this paper discussing the issue concerning the criteria for deciding as to how master-slave relations are formed. One method for handling this is to allow the programmer to specify (declare) such relations. The compiler can then verify that for each pair of communicating processes there is a single master.

Finally, we would like to stress again that this paper has only proposed an abstract implementation. The next step is to examine appropriate architectures that will allow the handling of our primitives Put, Get, Wait, and Signal in an efficient manner.

Acknowledgment

I would like to thank B. Johnson for his helpful comments concerning the material in this paper.

References

[2] E. W. Dijkstra, "Guarded commands, non-determinacy and
Exception Handling in CLU

BARBARA H. LISKOV AND ALAN SNYDER

Abstract—For programs to be reliable and fault tolerant, each program module must be defined to behave reasonably under a wide variety of circumstances. An exception handling mechanism supports the construction of such modules. This paper describes an exception handling mechanism developed as part of the CLU programming language. The CLU mechanism is based on a simple model of exception handling that leads to well-structured programs. It is engineered for ease of use and enhanced program readability. This paper discusses the various models of exception handling, the syntax and semantics of the CLU mechanism, and methods of implementing the mechanism and integrating it in debugging and production environments.

Index Terms—Exception handling, exit mechanisms, procedural abstractions, programming languages, structured programming.

I. INTRODUCTION

RECENTLY, there has been considerable emphasis on the development of programming language features that enhance the verifiability of programs [5]. While it is desirable that the task of developing correct programs be simplified as much as possible, another important goal of program construction is that programs behave "reasonably" under a wide range of circumstances. Such programs have been variously termed as reliable, robust, or fault tolerant.

In a reliable program, each procedure must be designed to behave as generally as possible. Its specifications should require a well-defined response to all possible combinations of legal inputs (inputs satisfying the type constraints), even when lower level modules on which this procedure is depending fail. Of course, different responses will be appropriate in the different cases. Note that even if the software has been verified, the possibility of hardware failure implies that software modules may fail, as does the presence of resource constraints.

This paper describes a linguistic mechanism that supports the construction of reliable software. The mechanism, called an exception handling mechanism, facilitates communication of certain information among procedures at different levels. The mechanism supports the view that different responses are appropriate in different situations. We assume that for each procedure there is a set of circumstances in which it will terminate "normally"; in general, this happens when the input arguments satisfy certain constraints and the lower level modules (implemented in both hardware and software) on which the procedure depends are all working properly. In other circumstances, the procedure is unable to perform any action that would lead to normal termination, but instead must notify some other procedure (for example, the invoking

Manuscript received March 8, 1979; revised June 25, 1979. This work was supported in part by the Advance Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under Contract N00014-75-C-0661, and in part by the National Science Foundation under Grants DCR74-21892 and MCS 74-21892.

B. H. Liskov is with the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

A. Snyder is with the Hewlett-Packard Corporation, Palo Alto, CA 94304.

Authorized licensed use limited to: Yale University. Downloaded on February 23,2022 at 15:26:58 UTC from IEEE Xplore. Restrictions apply.