In his recent paper, "Communicating Sequential Processes" (Comm. ACM 21, 8 (Aug. 1978), 666-677), C.A.R. Hoare outlines a programming language notation for interprocess communication in which processes are synchronized by the messages they exchange. The notation carries with it certain implications for the synchronization protocols required in a message transfer. These are not at all obvious and are made explicit here. An alternative convention is suggested in which communication and synchronization are partially uncoupled from one another.

Key Words and Phrases: concurrency, interprocess communication, message-passing, synchronization

CR Categories: 4.20, 4.22, 4.32

In "Communicating Sequential Processes" (CSP) [1], Hoare has given us the outline of a new programming style, designed to cater to the needs of concurrent programming. As with any new concept, there are several aspects that seem to deserve further exploration and discussion.

1. CAUSALITY, SYNCHRONIZATION, AND AUTOMATIC BUFFERING

The activities of parallel processes are synchronized by interprocess communications. The fundamental law of causality requires that a communication cannot be received before it is sent.

In CSP, it is further required that a source process cannot send a communication until a destination process is ready to accept it. Obviously, this restriction means that the parallel execution of CSP programs will be less than that of a system in which processes are synchronized only by the causality constraint. Furthermore, the additional synchronization is in a sense implicit, as it is an effect
accompanying the explicit action of delivering output. Implicit synchronization that restricts parallel execution does not seem to meet any of the stated design goals of CSP, and we should examine carefully the reasons for its inclusion.

One reason stated for this restriction is that otherwise an output message would need to be buffered automatically for subsequent input at its destination. It is argued that automatic buffering is not a primitive operation and should not be mandated by the programming language, since it can always be accomplished (if desired) by software buffering at the destination, or by interposing a buffering process between the ultimate source and destination. We can think of three objections to this argument.

(1) In order to program software buffering at the destination, the structure of a destination process must usually be somewhat complicated in order to anticipate input messages before they are to be acted upon.

(2) Underlying the notion that buffered communications are not primitive is the assumption that memoryless communications channels are the norm, rather than channels with memory. Although this may be true of many current computer architectures, it need not be true of future architectures. Specifically, communication via a time-shared, common bus provides no in-channel memory. However, direct processor-to-processor links can easily have memory, and memory is inherent when indirect, store-and-forward transmission is used. With current LSI technology, these kinds of channels are reasonable alternatives to a common bus, and in VLSI technology, they may be greatly preferred.

(3) A further assumption is that all communications are of such importance that none may be lost. This depends upon the application. For an example in which a few communications can be overlooked, consider a pair of processes: one which computes the location of a moving point to be presented on a display screen, and one which continuously refreshes the display.

\[\text{Update} :: [x, y : \text{integer};
\quad \begin{array}{l}
\quad \text{\bullet compute coordinates of the point;}
\quad \quad \text{Display!} (x, y);
\quad \end{array}
\]

\[\text{Display} :: [u, v : \text{integer}; j : 0..N\text{lines};
\quad \begin{array}{l}
\quad \quad \begin{array}{l}
\quad \quad \quad \text{\bullet} j := 0;
\quad \quad \quad \begin{array}{l}
\quad \quad \quad \quad \begin{array}{l}
\quad \quad \quad \quad \quad j < N\text{lines} \rightarrow \text{Update?} (u, v) \rightarrow \text{skip}
\quad \quad \quad \quad \quad \quad \text{\quad true} \rightarrow \text{refresh the } j \text{th line, using}
\quad \quad \quad \quad \quad \quad \quad \quad (u, v) \text{ as the current position}
\quad \quad \quad \quad \quad \quad \quad \quad \text{of the point; } j := j + 1
\quad \quad \quad \quad \quad \quad \quad \end{array}
\quad \quad \end{array}
\quad \end{array}
\quad \end{array}
\]

In this example, it is unimportant whether a particular set of coordinates is lost or not. If one set is lost, it will shortly be replaced by a more recent estimate of the position of the point. In fact, the process Update will make better use of its time by beginning the computation of a new set of position coordinates immediately rather than by waiting to transmit a set of data already inaccurate.

2. CLEAN TERMINATION OF PROCESSES

A CSP process is a unit of activity which is intended to terminate after some finite time. A process terminates when execution reaches the end of its command
list, with no further activity specified. A process may, of course, contain repetitive commands. A repetitive command terminates when all of its component commands fail in an attempt to execute it one more time. However, since an unfulfilled output (input) command does not fail immediately, simply because no corresponding input (output) is ready, it is necessary to provide for the exigency in which the partner process named in an I/O command has already terminated. When this happens, the I/O command is said to fail, and the process does not wait further for it to be completed. Thus it is necessary, in an interprocess communication of CSP, that each process be afforded a glimpse of the state of its communication partner, and herein lies a potential problem.

It can easily happen that a process will be able to terminate only after one or more of its partner processes has terminated. CSP provides very little support for the programmer to assure himself that the processes of a parallel command will always terminate in an orderly and predictable manner. In fact, it appears to be fraught with possibilities for deadlocking processes, and this seems to us a great weakness. For instance, consider

```
[ [A :: [B!msg1 (); C!msg2 ()]
 || [B :: [C?msg3 (); A?msg1 ()]
 || [C :: [A?msg2 (); B!msg3 ()]
```

which is a deadlocking parallel command in CSP, but would be deadlock free if a source process could continue execution immediately upon executing an output command.

In Section 7.9 of [1], Hoare has briefly discussed an important aspect of the clean termination problem, but the restriction he proposes there seems to run counter to the definition he gives in Section 2.4.

3. COMMUNICATION AND THE PROCESS STATE

It has been mentioned that 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. Let us examine the states of a process, relative to a single input command. We are only interested in a projection of the total state vector of the process, reflecting state changes that occur as a direct consequence of requesting and accepting an input message. The state transition diagram of Figure 1 is illustrative of the case in which an input command appears in a guard. Circles in the diagram represent states of the destination process:

R—ready to receive input;
A—active, but not ready to receive;
T—terminated.

The state-transition arcs are labeled to indicate the steps in the execution of an input command that bring them about. An A-to-R transition occurs when an input command is encountered in the execution sequence of a process. An R-to-A transition occurs if either the communication is successfully completed, or if it is aborted. Abortion of an input command can occur in either of two ways:
(1) if the source process has terminated, then the input command fails, or
(2) when the input command occurs in a guard, and execution of the guard of
some other component of the same guarded command succeeds, then the as
yet unsatisfied command is aborted.

In case (2) the R-to-A transition occurs "spontaneously," without any signal from
the source process. The states of a source process are similar, but there is no
"spontaneous" R-to-A transition for a source, as an output command cannot
occur within a guard.

The effective signaling that takes place between a source and a destination
process must establish that both are ready to proceed. Unfortunately, this is
complicated by the possibility of a spontaneous R-to-A transition by the desti-
nation. For an effective rendezvous, the R state of the destination process must
be split, as shown in Figure 2. This resolves the problem, since the only R2-to-A
transition is by completion of the communication. However, it requires an
additional signal, here called confirmation, to be exchanged. At the time the
destination process sends a confirmation signal and makes the transition R1 to
R2, the input command has succeeded in the sense that it cannot subsequently
be aborted by either of the communicating processes. Thus, if the input command

happens to be within a guard, the guard can be said to succeed at that point in time.

In the signaling diagram of Figure 3, the initial input request is presumed to contain the "pattern" that is to be matched by the output command. The request for confirmation and the confirmation itself are both pure signals, incorporated into the sequence solely for the purpose of exchanging state information. Thus, insofar as these signals carry no information relevant to the particular program, they may be regarded as overhead of the communications channel. (If the signaling were assumed to be initiated by a source, rather than by a destination process, then the initial message-request signal shown in Figure 3 would be omitted, and the pattern which gives the message type would accompany the first signal sent by the source process. A confirmation signal from the destination is still necessary, however.)

Note, however, that the illustration is independent of any assumptions about architecture or implementation; it demonstrates the inherent complexity of the communications scheme. And if output commands were allowed to appear in guards, a possibility suggested in Section 7.8 of [1], then a spontaneous R-to-A transition would also be possible for source processes. In consequence, a reliable rendezvous could no longer be accomplished by fixed-message signals exchanged between finite-state controls! In order for asynchronous processes to rendezvous, one process or the other must honor a commitment not to change the state it has reflected to the partner process, until the partner has had an opportunity to arrive at the expected rendezvous and complete the anticipated transaction.

4. TOPOLOGY OF INTERPROCESS COMMUNICATIONS CHANNELS

One of the more powerful aspects of the language notation presented in CSP is the ability to define an indexed array of processes. A source or a destination process for a communications command can then be identified merely by specifying a value for the index of a process. If an index is specified by a constant expression (or by an expression that differs by at most a constant from the index of the process in whose code the command appears), then the binding of the communications channel is clearly static, and this is the use made of indexed
processes in the examples given by Hoare. However, the notation appears to allow the use of more complex indexing expressions, which could be used to simulate dynamic binding of a source or a destination process as well. Also, even in examples 4.5, 5.2, and 6.1 given in [1], we find instances of static bindings of communications between a single process and all the other processes of an indexed set.

The ability to specify static bindings of input or output channels to an arbitrary number of partner processes, or to simulate dynamic bindings of I/O channels, should give us pause to reflect on the intended use of the language. If it were meant only to specify multiprocess execution on a multiplexed, single processor, then it would be easy to accept this very flexible communications topology, for we would expect that it could be realized by the use of shared memory segments, with the sharing managed by an operating system kernel. But the language notation is also meant to be applicable to multiprocessing with multiple processors, each having independent memory. With such a configuration, each communications channel between processors must be explicitly provided by some mixture of hardware and software technology. In the latter case, it is not presently feasible to presume an arbitrary interprocess communications topology among more than a handful of participating processes. A language notation that imposes no restriction on the communications topology provides little help to the programmer in distinguishing between that which is implementable, and that which is not.

5. SUMMARY OF OBJECTIONS

To summarize, CSP gives us a crisp and clean concept of process, in which the states of processes are isolated from one another, except at explicit communications points. This allows us to specify systems in which processes are driven by their inputs and synchronized by their outputs. However, the required synchronization at output commands seems very likely to degrade the performance of parallel systems, relative to their potential. Clean termination of processes is not always easy to assure in CSP programs and is left as a problem for the applications programmer to cope with. The communications mechanism turns out to require some unsuspected signals to be exchanged, and extension of the language to allow the inclusion of output commands in guards is not as simple as it might at first appear. The use of indexed process arrays allows one to specify interprocess communications of arbitrary topology, which may not be implementable in a multiprocessor configuration.

6. ALTERNATIVES

It is admittedly easier to voice criticism than it is to create a clean and elegant structure for others to criticize. Nevertheless, a critic should be willing to try to suggest alternatives to those aspects with which he finds fault. Let us consider one in particular.

I/O Ports With Memory

Within the framework of CSP, better isolation of process states can be obtained if communications are routed through ports. By allowing a port to have sufficient
memory to hold a single message, we can relax the strict synchronization of outputs with inputs. The states of a port are illustrated in Figure 4.

Ports can be declared in a parallel command, and the scope of a port declaration is the entire parallel command. Each port name can be used by only a single pair of processes, one of which may use it in output commands, and the other in input commands. A port may be closed by a destination process executing a terminal input command (denoted by ??) or marked to be closed by an output process executing a terminal output (!!). An input (output) command fails if the port to which it is directed is closed.

This convention avoids the need to reflect the state of one process to its communication partners in order to secure clean termination of a parallel command. Neither a source nor a destination needs to exchange confirmation signals with a port, because a port, once ready to accept output or deliver input, cannot “spontaneously” undergo a state transition. Concurrent execution of processes is limited only by the synchronization requirement of causality, unless a programmer explicitly specifies more synchronization. Ports can readily be implemented either in hardware as an interprocessor data pipe, or in software by the use of shared memory.

At this point we should realize that we have given up the principal synchronizer available for use in CSP by stripping the synchronization requirement from the output command. An explicit synchronization primitive can readily be provided. Let us define a Boolean-valued synchronizing function When-empty(.). If the state of the port given as its argument is Empty, then When-empty() evaluates to true; if the state of its argument is Closed then it evaluates to false, and its evaluation is delayed if the port is in any other state. Through the use of this synchronizer, the activity of a source process can be delayed until a destination process at the other terminal of a port has accepted a message previously placed there.

The alternative of port-directed I/O is appealing because it further isolates the states of processes, consistent with the design goal of CSP, still allows all process synchronization to be activated through the interprocess communications, yet
distinguishes the message transmission function from the synchronization aspect of communication and allows the programmer to specify exactly that which is needed. It further offers the programmer better means by which to assure that processes will terminate cleanly, by allowing the state of a port to be controlled and tested, rather than relying on the implicit communication of information about process states.

A further consideration is the required overhead of the communications facility. Communication through a port can be accomplished without the use of "hidden" signals. The communications facility offered by CSP not only mandates such signals in each communication, but further requires of a terminating process that it either remain active, in order to respond to its nonterminated partners that it can no longer participate in a communication, or else broadcast to all of its partners a notice of its demise. A process communicating through a port needs only to close the port at the time of its last communication.

As to the allowable interprocess communication topology, we have no patent solution. However, it seems reasonable that the expressions used to specify indices of communicating processes should be restricted to reflect the concept of proximity in an architecture on which the process array is to exist. Perhaps the topology should be reflected in a set of functions that can be particularized for any given hardware architecture. For instance, given a two-dimensional iterative array architecture one might allow two-dimensional process arrays and restrict the index expressions that can be used in an input or output command to use only the expressions that can be formed by use of functions Successor and Predecessor, relative to the index of the process in which the command appears. Further study of the problem of communications topology seems to be desirable.

REFERENCES

Received October 1978; revised April 1979