Reprinted from:
INTERNATIONAL JOURNAL OF PARALLEL PROGRAMMING

Vol. 16, No. 3, June 1987
Printed in Belgium

A Shared Memory Algorithm and Proof for the Generalized Alternative Construct in CSP[1]

Richard M. Fujimoto[2] and Hwa-chung Feng

Received August 1987; Accepted November 1987

Communicating Sequential Processes (CSP) is a paradigm for communication and synchronization among distributed processes. The alternative construct is a key feature of CSP that allows nondeterministic selection of one among several possible communicants. A generalized version of Hoare's original alternative construct that allows output commands to be included in guards has been proposed. Previous algorithms for this construct assume a message passing architecture and are not appropriate for multiprocessor systems that feature shared memory. This paper describes a distributed algorithm for the generalized alternative construct that exploits the capabilities of a parallel computer with shared memory. A correctness proof of the proposed algorithm is presented to show that the algorithm conforms to some *satefy* and *liveness* criteria. Extensions to allow termination of processes and to ensure fairness in guard selection are also given.

KEY WORDS: Communicating sequential processes; alternative operation; shared memory multiprocessor; parallel processing.

1. INTRODUCTION

Communicating Sequential Processes (CSP) is a well known paradigm for communication and synchronization of a parallel computation.[1,2] A CSP program consists of a collection of processes $P_1, P_2, \ldots, P_N$ that interact by exchanging *messages*. These message passing primitives, called input and output commands, are synchronous—a process attempting to output

[1] This work was supported by ONR Contract Number N00014-87-K-0184.

[2] Department of Computer Science, University of Utah, Salt Lake City, Utah 84112.

(input) a message to (from) another process must wait until the second process has executed the corresponding input (output) primitive.

An important feature of CSP is the *alternative* construct which is based on Dijkstra's guarded command.[3] This construct enables a process of *nondeterministically* select one communicant among many. Each alternative operation specifies a list of guards. Each guard has a set of actions associated with it that cannot be executed until the value of the corresponding guard becomes TRUE. Each guard consists of a sequence of Boolean expressions and an optional input command (output guards were not allowed in the original specification of CSP). A guard is said to be *enabled* if each of the Boolean expressions preceding the input command evaluates to TRUE. The value of a guard is TRUE if the guard is enabled and its input action has successfully completed.

Implementation of the alternative construct on a multiple processor computer has been the subject of much research.[4-11] It has been argued that the exclusion of output guards in the original definition of CSP is too restrictive and can degrade performance.[6,10] A *generalized* alternative construct that allows output guards has since been proposed, and algorithms to implement it have been developed.[4-7] However, all of the algorithms reported thus far assume a message-based computer architecture; no shared memory is assumed. The principal contribution of this paper is to present an algorithm for implementing the generalized alternative construct on a shared memory multiprocessor and to prove its correctness. To the authors' knowledge, no such algorithm has previously been reported.

CSP does not assume shared memory between constituent processes, so one might ask why implementation on a shared memory machine is an issue. Implementation of CSP on a shared memory architecture is an important question for several reasons:

- CSP has clean semantics that simplify proving the correctness of programs. It is a worthwhile programming paradigm in its own right, independent of the underlying machine architecture.
- The message passing paradigm is a natural means of expressing programs in many application areas that are well suited for shared memory machines. For example, distributed discrete event simulation algorithms are usually described in terms of message passing paradigms,[12,13] and implementations on shared memory architectures have been described.[14] Similarly, message passing is used extensively in object-oriented programming.
- Shared memory machines are widely available. Multiprocessors such as the BBN Butterfly™ [see Ref. 15] and Sequent Balance™ are available from the commercial sector, and numerous shared

memory research machines such as IBM's RP3 [see Ref. 16] and the University of Illinois's Cedar [see Ref. 17] have also been developed.

- Shared memory architectures provide fast interprocessor communications. A complete interconnection among processors is provided, avoiding costly store-and-forward communication software in message-based architectures such as the Intel iPSC™ [see Ref. 18]. At present, parallel processors using shared memory are more appropriate for applications requiring frequent communication among the constituent processes.

Although one can clearly "retrofit" any message-based algorithm to a shared memory architecture by building a suitable interface, this will often lead to an inappropriate and awkward implementation. Existing message-based algorithms for the generalized alternative construct are not appropriate for a shared memory machine because (1) they do not exploit the facilities afforded by shared memory, leading to an inefficient implementation; and (2) they require additional "system" processes to respond to incoming messages (e.g., requests for rendezvous) resulting in unnecessary context switching overhead. We will describe an algorithm for the generalized CSP alternative construct that exploits the facilities afforded by shared memory and avoids the aforementioned system processes.

The algorithm is fully distributed and does not rely on any centralized controller. The notion of total ordering among processes [Ref. 6] is used to prevent deadlocks, but is applied *dynamically* on transactions (defined later) rather than statically as originally proposed. The status of a remote process can be interrogated directly, in contrast to the message-based algorithms where message handshake and context switching overheads reduce the efficiency of the implementation. However, because processes in the proposed algorithm concurrently access shared data, great care must be taken to avoid race conditions. An "abort-and-retry" protocol is used to avoid certain race conditions, and a proof is also included to verify that the algorithm operates correctly according to *safety* and *liveness* criteria.(19) Modifications are also suggested to achieve fairness.(20)

The remainder of this paper is organized as follows. The semantics of the generalized alternative construct are discussed first, followed by a description of the assumed machine architecture. The proposed algorithm and a discussion of its operation is then presented. Other important issues related to the algorithm are then discussed, and an extension to handle termination of processes is described. We conclude the paper with a proof of the correctness of the algorithm followed by a discussion of fairness issues.

2. THE ALTERNATIVE CONSTRUCT

A guard of the alternative construct can appear in one of two possible forms. The first, called the *pure Boolean* form, contains no I/O command. For example, in

$$(x = 1 \text{ and } y > 5) \rightarrow z := z * 3$$

the predicate to the left of the '$\rightarrow$' operator is a pure Boolean guard. The second form, called the *I/O guard* form, contains an I/O command as well as an (optional) Boolean part. For example, in

$$P_1?x \rightarrow z := z + 1$$

the input guard $P_1?x$ requests input from process P_1. The received data is assigned to the variable x. Guards such as this which do not contain a Boolean part are referred to as *pure* I/O guards. In effect, the boolean part is the constant TRUE. An I/O guard is said to be *enabled* if the Boolean part is TRUE, so a pure I/O guard is *permanently* enabled.

Consider the following alternative construct:

$$[G_{i(i \in PB)} \rightarrow S_i \,\square\, G_{j(j \in IO)} \rightarrow S_j]$$

Where *PB* stands for the set of indices of all of the pure boolean guards and *IO* the set of indices of all of the I/O guards. Whenever this alternative construct is executed, exactly one guard is selected and the corresponding action (S_i or S_j) is executed. The selection is made according to the *availability* of the guards. For pure Boolean guards, the guard is said to be available if it is enabled, i.e., if the Boolean part evaluates to TRUE. For I/O guards, the guard is available if it is enabled and the process associated with the guard is also ready to communicate using the complementary I/O command. Because we assume I/O commands only appear in guards of alternative operations, this implies the remote process is executing an alternative operation in which the corresponding I/O operation is part of an enabled guard. If more than one guard is available, one is chosen arbitrarily. The application program cannot control this selection.

Pure Boolean guards can be resolved without any interaction with other processes. Therefore, to simplify the discussion which follows, we will restrict attention to the resolution of I/O guards.

3. THE MACHINE ARCHITECTURE

The machine is assumed to be a shared memory multiprocessor. The algorithm is well suited for machines such as BBN's Butterfly or Sequent's Balance, among others. Several primitive are used in the algorithm.

None are unusual in a multiprocessor environment, and all can be easily constructed using a test-and-set and standard scheduling primitives.

The CSP program contains processes $P_1, P_2, ..., P_N$. Process P_i is assigned the unique *process ID i* to distinguish it from others.

We will assume the following:

- Communications are reliable. An error free communications mechanism exists so that two distinct processes can communicate by exchanging a message. In particular, *Send*(M, R) and *Recv*(R): *Message* provide the same semantics as CSP's output and input commands, respectively. M is the message which is transmitted and R is the ID of the remote process with which communications is to take place. *Recv* returns the received message (of type *Message*). In accordance with CSP semantics, we assume the process invoking the primitive blocks until process P_R executes the complementary I/O primitive.
- Read and write accesses to shared memory are atomic, as is normally the case with a shared memory multiprocessor. *AtomicAdd*(X): *INTEGER* atomically increments the integer variable X and returns the original value of X.
- *WaitForSignal* and *Signal* primitives are available to block and unblock the process, respectively. A signal contains a single, user defined integer value. *WaitForSignal*( ): *INTEGER* causes the process invoking the primitive to block until a signal becomes available to it from *any* other process and returns the integer value stored within the signal. *Signal*(R, i) sends a signal containing integer i to process P_R. The *Signal* primitive wakes up the signaled process if it is block on *WaitForSignal.* Otherwise, the signal remains in effect until P_R executes a *WaitForSignal* primitive. If a second signal is sent to P_R before the first is absorbed by a call to *WaitForSignal*, the first signal is discarded.
- *Lock* and *Unlock* primtives provide exclusive access to shared data structures. *Lock*(L) will block until the lock L becomes zero, at which time L is set to one. The "test-and-set" operation must be atomic. *Unlock*(L) sets the lock L to zero. Further, we assume the *Lock* primitive is fair, i.e., if a process is blocked while attempting to obtain a lock, it does not remain blocked an unbounded amount of time unless the lock is not unlocked for an unbounded amount of time.

It is assumed that all input and output commands occur within guards of the alternative construct. Simple CSP input and output primitives are

special cases of the alternative construct. Simple CSP input and output primitives are special cases of the alternative construct. We also assume that the variables used in the alternative algorithm are not modified by processes except as indicated in the algorithm. Finally, it is assumed that processes do not terminate. The algorithm can be extended to handle termination, as will be discussed later.

4. THE ALTERNATIVE ALGORITHM

Each invocation of an alternative operation is referred to as a *transaction.* A transaction begins when an alternative operation is initiated and ends when a successful communication has been completed. A process will usually engage in many transactions during its lifetime. A total ordering is imposed among all transactions entered by *all* processes of a given CSP program. A unique sequence number, referred to here as a *transaction ID*, is associated with each transaction.

Two processes, each of which initiates an alternative operation that results in a communication between them, are said to *rendezvous.* More precise definitions of rendezvous and other terminology introduced in this section will be presented later. Each rendezvous always involves exactly two distinct processes. In a *typical* rendezvous, the first process to enter the alternative will block, waiting for a signal from the second. When the second process enters the alternative, it will *commit* to the first in order to obtain "permission" to rendezvous; the "committing" process will then signal and exchange a message with the blocked process, and both will complete their respective alternative operations.

A *commit* operation is, in effect, a request for rendezvous. It will be shown that a rendezvous will occur only after a successful commit operation has taken place, and every successful commit results in a rendezvous. A process will not attempt to commit until it has determined that the process with which it is committing is a suitable candidate for rendezvous, i.e., each lists the other in their respective guard lists, and the two processes are not both trying to execute the *same* I/O operation (*Send* or *Recv*). The commit operation resolves conflicts when two different processes attempt to simultaneously rendezvous with a third. The algorithm uses an "abort and retry" mechanism to avoid race conditions when two potential communicants simultaneously enter the alternative command.

4.1. Process States

Each process can be in one of the following states:

- WAITING. The process is blocked on a *WaitForSignal* operation, waiting for another process to rendezvous with it.

- ALT. The process has begun an alternative operation, and is scanning through its list of guards to find a process with which it can rendezvous.
- SLEEPING. The process was forced to abort an alternative operation. Each time the process aborts, it goes to sleep for some time before retrying. While blocked in this way, the process is in the SLEEPING state. This state differs from the WAITING state because a process may remain in the latter for an unbounded amount of time.
- RUNNING. The process is executing user or system code not related to the alternative operation. The process is in the RUNNING state if it is not in any of the other states listed above. Once the process initiates an alternative operation, it can only be in the WAITING, ALT, or SLEEPING state until the alternative operation completes with a rendezvous.

It is possible to combine the RUNNING and SLEEPING states into a single state. Two states are used to simplify the description of the algorithm and its proof.

A state transition diagram for each process is shown in Fig. 1. Initially, a process is in the RUNNING state. Once the process initiates an alternative operation, it enters the ALT state. If the process is forced to abort the alternative it switches to the SLEEPING state, and returns to the ALT state when it retries. If the process is able to commit and rendezvous with another process, it returns to the RUNNING state. Otherwise, the process moves to the WAITING state until some other process commits to it, at which time it rendezvous and returns to the RUNNING state.

The ALT and SLEEPING states should be viewed as "transitory" states through which a process passes while trying to commit or move into the WAITING state. It will be shown that a process cannot remain in either the ALT or the SLEEPING state for an unbounded amount of time on a single transaction.

4.2. Shared Variables

Each process P_j maintains a number of variables that may be examined, and in some cases modified, by other processes:

- $AltList_j$ lists the guards associated with the last alternative operation initiated by P_j that caused P_j to enter the WAITING state.
- $AltLock_j$ is a lock used to control access to $AltList_j$. It is initialized to 0 (unlocked).

- $State_j$ holds the current state of P_j. It may be set to WAITING, ALT, SLEEPING, or RUNNING, and is initialized to RUNNING.
- $WakeUp_j$ is initialized to 1 and is set to zero by P_j whenever it enters the WAITING state. It is incremented (atomically) by processes trying to commit to P_j. This variable prevents two processes from both successfully committing to a third on a single transaction.

There is also one system wide global variable used by the algorithm:

- *NextTransID* is initialized to zero and is incremented each time a process initiates an alternative operation. This variable ensures a unique transaction ID can be generated for each instance of an alternative operation.

Use of a global variable to generate unique transaction IDs is not strictly necessary. It is possible to generate unique transaction IDs that conform to the requirements of the algorithm without use of any shared variables. This will be discussed later.

One procedure merits special attention. $CheckAndCommit(m, g_i)$: *INTEGER* is called by process P_l (l denotes the *local* process) to check that "valid" communications can take place between P_l using guard g_i and P_m (m denotes the *remote* process). If so, P_l attempts to commit to P_m. If successful, *CheckAndCommit* returns a positive integer indicating the corresponding guard in the *remote* process P_m. Otherwise, *CheckAndCom-*

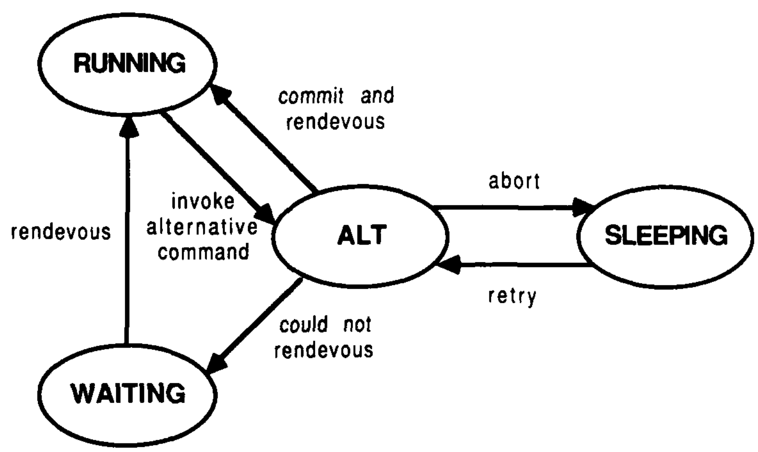

Fig. 1. State diagram of each process.

mit returns a nonpositive integer, denoted by the constant FAILED. This procedure is shown in Fig. 2.

CheckAndCommit uses a procedure *CheckGuard*($AltList_m, g_i$): *INTEGER* that scans the remote alternative list $AltList_m$ looking for a *matching* and *compatible* guard g_j to the local guard g_i. By *matching* we mean g_j contains an I/O operation with P_l. By *compatible* we mean g_i and g_j do not *both* contain input (output) commands. *ChechGuard* returns an integer j that denotes the number of a matching and compatible guard if one was found, and FAILED otherwise. If such a guard is found, P_l attempts to commit to P_m by testing if $WakeUp_m$ is zero, and if so, incrementing it. An ordinary addition is used rather than the *AtomicAdd* primitive to increment $WakeUp_m$ because $AltLock_m$ guarantees atomicity. If P_l is the first process to commit to P_m, i.e., if $WakeUp_m$ was previously zero, then P_l successfully commits, *CheckAndCommit* returns the number of the corresponding guard, and rendezvous is imminent. Otherwise, *CheckAndCommit* returns FAILED. $AltLock_m$ ensures serial access to $AltList_m$. As will be demonstrated later, it is crucial that this lock is not released until *after* the commit operation is attempted (if it is attempted) in order to avoid race conditions. This would be the case even if an *AtomicAdd* operation were used to increment the *WakeUp* variable.

```
/* m is the remote process */
PROCEDURE CheckAndCommit(m, gi): INTEGER;
VAR
     INTEGER GuardNumber; /* number of matching guard */
BEGIN
     Lock(AltLockm);
     /* check guard matches and is compatible */
     GuardNumber := CheckGuard(AltListm, gi);
     IF (GuardNumber = FAILED) THEN
         Unlock(AltLockm);
         RETURN (FAILED);
     /* try to commit */
     ELSEIF (WakeUpm = 0) THEN
         WakeUpm = WakeUpm + 1;
         Unlock(AltLockm);
         RETURN (GuardNumber);
     ELSE
         Unlock(AltLockm);
         RETURN (FAILED);
         END;
END CheckAndCommit;
```

Fig. 2. Procedure to check that a potential communication is valid and, if so, to commit. The *CheckGuard* function returns the number of a matching (and compatible) remote guard or returns FAILED if none was found.

4.3. Other Notation

For notational convenience, other variables and predefined functions are defined that are used in the algorithm. These include:

- TransID_l is a variable that contains the ID of the current transaction in which process P_l is engaged.
- $\text{CommunicantID}(g_i)$ is a function that returns the ID of the process listed in the I/O command portion of guard g_i.
- $\text{Communicate}(g_i)$ executes the I/O command in guard g_i.

4.4. Description of the Algorithm

The alternative algorithm is shown in Figs. 3 and 4. The *Alternative* procedure shown in Fig. 3 is a "front end" that is responsible for retrying aborted attempts. It does not return until a rendezvous has been completed at which time it returns an integer indicating the guard that was eventually satisfied. The heart of the algorithm lies in the *TryAlternative* procedure shown in Fig. 4. The parameters passed to this and the *Alternative* procedure are n enabled I/O guards $g_1, g_2, \ldots, g_n$. Each guard contains either a single output or a single input primitive.

The *Alternative* procedure first obtains a unique transaction ID by performing an *AtomicAdd* operation on the global *NextTransID* variable. It then attempts to rendezvous by calling the *TryAlternative* procedure. *TryAlternative* either returns the number of the guard on which a rendezvous occurred, or the FAILED flag indicating the attempt must be retried. The same transaction ID remains in use despite one or more failed

```
/* gi are enabled I/O guards */
PROCEDURE Alternative(g1, ..., gn):  INTEGER;
VAR
     INTEGER ReturnValue; /* indicates guard that rendezvoused */
BEGIN
     /* l is the local process id */
     TransIDl := AtomicAdd(NextTransID);
     ReturnValue := FAILED;
     WHILE (ReturnValue = FAILED) DO
         ReturnValue := TryAlternative(g1, ..., gn);
         END;
     RETURN (ReturnValue);
END Alternative;
```

Fig. 3. The "front end" procedure. *TryAlternative* returns the number of the guard on which a rendezvous took place or FAILED if it aborted.

attempts. It will be shown that *TryAlternative* cannot fail an unbounded number of times within a single transaction. In the discussed that follows, P_l again refers to the local process and P_m to the remote process with the guard that is being scanned.

```
PROCEDURE TryAlternative(g1, ..., gn): INTEGER;
VAR
    BOOLEAN flag;
    INTEGER GuardNumber; /* corresponding guard of Pm */
    INTEGER i, m, RemoteID;
BEGIN
    Statel := ALT;
    /* look for rendezvous with a waiting process. */
    FOR i:=1 TO n DO
        m := CommunicantID(gi);
        flag := TRUE;
        WHILE (flag) DO
            CASE Statem DO /* The remote process state. */
            RUNNING: flag := FALSE;
            SLEEPING: flag := FALSE; /* try next guard */
            WAITING: GuardNumber := CheckAndCommit(m, gi);
                IF (GuardNumber = FAILED) THEN
                    flag := FALSE; /* try next guard */
                ELSE /* Wake up Pm */
                    Statel := RUNNING;
                    Signal(m, GuardNumber);
                    Communicate(gi);
                    RETURN (i);
                    END;
            ALT:RemoteID := TransIDm;
                IF (TransIDl < RemoteID) THEN
                    WHILE ((Statem = ALT) AND (RemoteID = TransIDm)) DO END;
                ELSE
                    Statel := SLEEPING;
                    WHILE ((Statem = ALT) AND (RemoteID = TransIDm)) DO END;
                    RETURN (FAILED); /* abort...*/
                    END; /* if-then-else */
                END; /* case statement */
            END; /* while loop */
        END; /* for statement */
    /* couldn't find guard to rendezvous */
    Lock(AltLockl); AltListl:=(g1, ..., gn); Unlock(AltLockl);
    WakeUpl := 0; /* first to commit gets rendezvous */
    Statel := WAITING; i := WaitForSignal(); Statel := RUNNING;
    Communicate(gi);
    RETURN (i);
END TryAlternative;
```

Fig. 4. The *TryAlternative* procedure attempts to rendezvous.

After setting the state of the process to ALT, P_l examines each guard listed in the alternative operation one after the other. Some action is then performed depending on the state of P_m.

If P_m is in the RUNNING state, P_l simply advances to the next guard. In this case, P_m has not yet entered a transaction and is not yet ready to rendezvous.

If P_m is in the SLEEPING state, P_l again advances to the next guard. P_l advances because the *Alternative* procedure guarantees that the SLEEPING process (P_m) will eventually retry its alternative operation. It will be shown later that a process cannot remain in the SLEEPING state for an unbounded amount of time. If P_l and P_m are destined to eventually rendezvous on this transaction, P_l will typically proceed to the WAITING state, and P_m will later retry, commit, and rendezvous with P_l.

If P_m is WAITING, then P_m has already reached the rendezvous point so P_l attempts to rendezvous. $AltList_m$ is examined to make sure a valid communication can take place, and if so, P_l attempts to commit. If successful, P_l will awaken P_m (by sending a signal) and rendezvous. Otherwise, P_l advances to the next guard.

Finally, if P_m is in the ALT state, some spcial precautions must be taken to avoid race conditions. This situation could result, for example, when P_l and P_m initiate an alternative operation at approximately the same time. The two processes may or may not be destined to rendezvous, however. In fact, P_m's alternative operation may not even contain a guard with P_l as a communicant.

If P_l sees P_m in the ALT state, P_l will pause in a busy wait loop until P_m either changes to another state or advances to a new transaction. To avoid deadlock (e.g., two processes each waiting for the other to leave the ALT state), P_l will first change to the SLEEPING state if its transaction ID is larger than that of P_m's. In this case, P_l must abort and retry the operation after P_m changes state in order to avoid race conditions (discussed later, in the proof of Lemma 8). Because higher priority is given to the process with a *smaller* transaction ID, the priority of each transaction tends to increase with time. This is necessary to ensure liveness in the algorithm.

Although the busy wait loop and abort retry scenario might initially appear to cause wasted time that could be better spent pursuing other activities, it is anticipated that this situation will arise infrequently in practice. Performance evaluations using empirical techniques are currently in progress to verify that this is the case.

It is interesting to note that the state of P_m may change immediately after P_l examines $State_m$. It will be proven that the algorithm operates correctly despite this apparent inconsistency. In fact, it will be shown that

the only locking that must be performed in the entire algorithm is that associated with *AltLock*.

If P_l goes through its entire guard list without rendezvousing with another process, P_l enters the WAITING state and calls *WaitForSignal* to block until another process commits to it. Before calling *WaitForSignal*, however, P_l also sets $AltList_l$ to contain the current guard list and "activates" $WakeUp_l$ by setting it to zero. After some process later commits to P_l, a signal is received, a communication takes place, and *TryAlternative* returns the identity of the (local) guard that rendezvoused. This information was sent to P_l in the signal that awakened it.

We should emphasize at this point that it is crucial that the operations listed in Figs. 2–4, be performd in *exactly* the order in which they appear. Seemingly minor changes such as swapping the order of the statements

$\mathrm{WakeUp}_1 := 0;$
$\mathrm{State}_1 := \mathrm{WAITING};$

introduces a race condition that invalidates the correctness proof.

We note that the *Lock* operation preceding the statement that modifies *AltList* must remain even if modification can be done atomically. The locking protocol in this and the *CheckAndCommit* procedure are carefully designed to avoid race conditions. Finally, it is noteworthy that the statement that sets $WakeUp_l$ to zero need not be executed while $AltLock_l$ is locked. The correctness proof only requires that two processes do not both read a zero value from $WakeUp_l$ during a single transaction of P_l.

5. DISCUSSION

Several aspects of the alternative algorithm merit further discussion. These are discussed next.

5.1. Transaction IDs

The algorithm uses dynamically assigned transaction IDs to determine the "winner" when a process finds another in the ALT state. Dynamic IDs are used rather than static, process IDs to ensure liveness. Intuitively, *liveness* means that two processes that "should" rendezvous eventually will, while *safety* means that any rendezvous that occurs is valid. The proposed approach avoids scenarios in which a process is repeatedly forced to abort and retry its alternative operation an unbounded number of times; this is because the priority of a transaction automatically increases with time as

other transactions are allowed to complete and new ones, with higher IDs and correspondingly lower priorities, are initiated. Dynamic transaction IDs guarantee this property while static IDs do not. It is important that a new transaction ID is only allocated when an alternative is first initiated, as in done in Fig. 3, and *not* when an existing operation is retried.

One can avoid using a global variable (*NextTransID*) to generate transaction IDs if contention is a concern. This function can be performed locally, within each process. Process P_l can create a new, unique, transaction ID by concatenating a *local* sequence number with l, the unique ID for the process. The sequence number is incremented each time a new transaction ID is created by that process. It is imperative that the process ID occupy the *least* significant portion of the transaction ID to ensure liveness, as was discussed earlier.

A second concern is overflow of the *NextTransID* variable. Overflow invalidates the liveness property of the algorithm because a transaction's priority does not necessarily increase with time. Also, because transaction IDs cannot be guaranteed to be unique after overflow has occurred, the arbitration protocol could fail (this could be circumvented by appending the process ID to the least significant portion of the transaction ID, however). In any event, overflow can be easily avoided by using a variable of large precision. For example, a 64 bit variable will not overflow with 1000 processes, each initiating a new alternative construct every microsecond, in over 500 years!

5.2. Channel I/O

In many CSP implementations, interprocess communication is based on pre-allocated *channels*. Each channel is a unilateral link between two communicating processes. The channel model facilitates modularity, reusability, and hierarchical construction of programs because a program can be "constructed" by interconnecting a group of constituent processes. The algorithm presented above can be adapted to the channel I/O model by modifying the *Send* and *Recv* primitives and translating port identifiers to process IDs. The *CheckAndCommit* procedure, for instance, must be modified to check for matching *channels* rather than matching process IDs. These modifications are a simple extension of the proposed algorithm.

5.3. Termination

Termination is another important issue facing real implementations. This was not treated in the previous discussion to simply the presentation. The termination semantics play an important role in CSP because it is the

basis of the termination of the *repetitive* command.[1] If an alternative operation is embedded within a repetitive command and no guard of the alternative can become true, e.g., because all processes associated with enabled guards have terminated, the repetitive command terminates. If no such repetitive command surrounds the alternative operation and it is found that no guards can become true, an error results.

In the context of the proposed algorithm, it is sufficient that the *Alternative* procedure determine when no guards can become satisfied and return an appropriate flag denoting this situation. The algorithm can be extended to handle termination by adding a shared variable called $GuardCount_i$ to each process P_i and a new process state called TERMINATED. $GuardCount_i$ indicates the number of I/O guards on which P_i might potentially rendezvous in the current transaction and contains a meaningful value whenever P_i is in the WAITING state. It is equivalent to the number of guards in $AltList_i$. The $GuardCount_i$ variable is used to detect situations in which P_i cannot rendezvous because all of the processes in its guards have terminated. This is the only case in which the *Alternative* procedure will return *without* rendezvous.

Whenever a process P_j terminates, it marks its state as TERMINATED and then examines the state of each of its neighboring processes, i.e., those processes which might communicated with P_j. If P_j finds another process P_i in the ALT state, it executes a busy wait loop until $State_i$ changes. This is necessary because P_j cannot know if P_i saw P_j had entered the TERMINATED state. If P_j finds P_i in the WAITING state and $AltList_i$ contains a guard listing P_j as a communicant, then P_j (atomically) decrements $GuardCount_i$ to indicate that one fewer guard is available for rendezvous. No further action is required unless the decrement operation causes $GuardCount_i$ to become zero. In this case, the terminating process must send P_i a special signal to indicate P_i's alternative operation can never rendezvous. Upon receiving this signal, the alternative operation in P_i will return a special flag indicating the alternative operation completed *without* rendezvous.

When looking for a process with which to rendezvous, i.e., when scanning the status of neighboring processes in the *TryAlternative* procedure, an I/O guard corresponding to a terminated process is skipped in the same way processes in the RUNNING or SLEEPING state are skipped. Such guards are excluded from $AltList_i$ and $GuardCount_i$ should the process fail to rendezvous and move into the WAITING state. If all I/O guards correspond to terminated processes, the alternative construct again returns a flag indicating the operation completed without rendezvous.

Finally, some precautions must be taken to avoid race conditions. The mechanism described above to notify a WAITING process that it cannot rendezvous on any of its guards bears some resemblance to the protocol used

to commit to a process—the *WakeUp* variable is analogous to *GuardCount* and committing (by incrementing *WakeUp*) is analogous to decrementing *GuardCount*. Therefore, it is not surprising that the precautions that are necessary to avoid race conditions are similar. In particular, $GuardCount_i$ must be set before P_i sets $State_i$ to WAITING but *after* P_i modifies $AltList_i$ (see Fig. 4). Identical containsts apply regarding the moment at which *WakeUp* to set to zero. Finally, when P_j wishes to decrement $GuardCount_i$, the same protocol that was used in the *CheckAndCommit* procedure (see Fig. 2) to lock $AltLock_i$ must be used to decrement $GuardCount_i$, i.e., $AltLock_i$ must *not* be released until *after* the decrement operation has completed.

6. Proof of Correctness

The correctness of the algorithm is established by proving that during the (potentially) infinite execution sequence, all processes and the interplay between them maintain invariant properties known as *safety* and *liveness*.[14,21] As previously described, safety means that any rendezvous which occurs is correct. For example, it is not possible for two processes to rendezvous which do not each list the other in some guard of their respective alternative lists. Liveness ensures that two processes which should rendezvous eventually will, provided of course each does not first rendezvous with some other process. These terms are defined more formally in Theorems 2 and 3. Intuitively, the safety property ensures that nothing "bad" will happen, while liveness ensures something "good" will eventually happen. Together they guarantee correct operation of the algorithm.

Before beginning the proof, terminology that has been used informally until now will be defined more precisely. These definitions are in terms of the alternative algorithm shown in Figs. 2–4. It is assumed throughout that the CSP program consists of a collection of processes $P_1, P_2, \ldots, P_N$.

6.1. Definitions

1. A process P_i is said to enter a transaction T_r when P_i calls the *Alternative* function. It exits transaction T_r when it returns from the function call. The notation $P_i(T_r)$ should be read "process P_i (while P_i is in transaction T_r)." It will be clear from the context that this notation is used that P_i must be in some transaction. Each transaction has a unique ID associated with it (r for transaction T_r) that is used to form a total ordering among all transactions. A transaction need not terminate. For example, the *application* program may deadlock.

2. A process P_i in transaction T_r is said to *commit* to process P_j if $P_i(T_r)$ increments $WakeUp_j$ from zero to one. The algorithm is such that every time $WakeUp_j$ is incremented, a commit operation takes place.
3. A transaction T_r executed by process P_i is said to rendezvous with transaction T_s for process P_j if either (a) P_i is in the WAITING state and receives a signal from P_j, or (b) P_i signals P_j after committing to P_j. It will be shown that once a process rendezvous, it will exchange a message, complete the current transaction and return to the RUNNING state.
4. A signal sent by P_i to P_j is said to be *pending* if (1) it was sent but has not yet been received by P_j, or (2) it was received, but has not yet been absorbed by P_j through a call to *WaitForSignal.*
5. A communication between P_i and P_j is *compatible* if one process wishes to send, and the other wishes to receive. Otherwise, the communication is said to be *incompatible.*
6. $VAR_i(T_r)$ denotes the value of state variable VAR of process P_i during transaction T_r. For example, $AltList_i(T_r)$ is the alternative list of P_i during transaction T_r. If significant, the point in time *during* the transaction that is referred to will be stated explicitly.
7. $GuardList_i(T_r)$ lists the guards that are passed as parameters to the alternative operation executed by P_i on transaction T_r. We will take the liberty of giving *GuardList* a dual meaning—it either refers to a list of *guards* or a list of *processes* that are designated in the I/O commands of these guards. The particular meaning that is intended will be clear from the context.

6.2. The Safety Property

Lemmas 1–5 and Theorem 1 state that no race conditions arise that might cause a process to mistakenly rendezvous with a second process that does not wish to rendezvous with the first. Theorem 2 subsumes Theorem 1 and ensures that the algorithm obeys the safety property.

Lemma 1. $P_i(T_r)$ signals P_j iff $P_i(T_r)$ commits to P_j.

Proof. This follows immediately from examination of the algorithm. A process only sends a signal after it commits, and always sends a signal after it commits. ▮

Lemma 2. At the beginning and at the end of each transaction entered by P_j, the following conditions must hold:

(a) No signals sent to P_j are pending.

(b) $WakeUp_j$ is nonzero.

Proof. Use induction on k, the number of transactions entered by P_j. Consider the first transaction ($k = 1$) executed by P_j. Condition (b) must be true at the beginning of this transaction because $WakeUp_j$ is initialized to 1 and is only modified by P_j during a transaction. Condition (a) is also true because no process can send a signal to P_j until $WakeUp_j$ is reset to 0.

If P_j does *not* reset $WakeUp_j$ to 0 during its first transaction, (a) and (b) are trivially true at the end of the transaction. If P_j does reset $WakeUp_j$ to 0 during its first transaction, (a) and (b) are true at the end of the transaction because (1) P_j resets $WakeUp_j$ to 0 at most once during any transaction; (2) the atomicity of the "test-and-increment $WakeUp_j$" operation in the *CheckAndCommit* procedure guarantees that at most one process will commit and send a signal to P_j as a result of $WakeUp_j$ being set to 0; and (3) P_j always calls *WaitForSignal* after resetting $WakeUp_j$ to 0, so the only signal that can be sent to P_j must be absorbed. Therefore, (a) and (b) are again true at the end of the first alternative operation as well as at the beginning.

Inductive step: Assume lemma 2 is true on the end of the kth transaction entered by P_j. It is easy to see that lemma 2 is also true at the beginning and end of the $k+1$st transaction entered by P_j using arguments identical to those presented before. ∎

Lemma 3. Two processes, P_i and P_j, cannot both commit to a third process P_k during a single transaction T_t entered by P_k.

This lemma was actually proven as part of the proof of Lemma 2, but we include it as a separate lemma for future reference.

Lemma 4. If $P_i(T_r)$ commits to P_j, then P_j must have been in the WAITING state when P_i committed to P_j, and P_j must remain in the WAITING state until P_j receives the signal sent by P_i that results from this commitment.

Proof. P_i check that P_j is in the WAITING state before trying to commit to P_j. Let us assume P_j is in transaction T_s when P_i sees P_j in the WAITING state. Therefore, it only remains to be shown that P_j is still in the WAITING state in transaction T_s when P_i commits, as well as when the signal is received.

Suppose P_j completed T_s before P_i committed. Then, P_j must have advanced to another transaction (T_t) and reset $WakeUp_j$ to 0 before P_i committed, or else P_i's commit would have failed. If $P_i \in GuardList_j(T_t)$, P_j would *not* have been able to scan past the guard containing P_i because P_i

is in the ALT state, so it must be that $P_i \notin GuardList_j(T_t)$. Then, it must be that (1) P_i checked $AltList_j$ while the list corresponded to some transaction preceding T_t (or again, the commit would have failed); (2) $P_j(T_t)$ modified $AltList_j$ and reset $WakeUp_j$ to 0; and (3) P_i successfully committed to P_j. However, the *CheckAndCommit* operation guarantees that checking *AltList* (step 1) and committing (step 3) are atomic, so $AltList_j$ could not have been modified between these two operations. Therefore, P_j must have still been in T_s when P_i committed.

P_j must also remain in the WAITING state until the signal is received because P_j cannot leave this state until it first receives a signal. By Lemma 2a, there were no signals pending when transaction T_s began. By Lemma 3 no process other than P_i will commit to P_j during this transaction, so no signal other than P_i's are sent to, or received by P_j during T_s. Therefore, P_j cannot unblock from the *WaitForSignal* operation and therefore cannot change state until receiving the signal sent by P_i. ▮

The preceding lemma shows that arbitrarily long delays may occur from the time P_i observes that P_j is in the WAITING state until P_i's signal actually arrives at P_j. If the commit succeeded, this lemma guarantees that nothing "interesting" will happen at P_j from the time P_i found it to be waiting until the signal was received. This lemma also highlights the necessity of ensuring that checking the remote guard list and committing are implemented as an atomic operation.

Lemma 5. No signals are lost in the alternative algorithm.

Proof. This follows immediately from the previous lemmas. No signals can be sent to a process while another signal is pending, so none are lost. ▮

Theorem 1. If $P_i(T_r)$ signals (rendezvous) P_j, then P_j must be in some transaction T_s both when the signal is sent and when it is received. Further, $P_j(T_s)$ rendezvous $P_i(T_r)$.

Proof. By Lemma 4, P_j must be in a transaction when the signal is sent and when it is received, and remain in the WAITING state during this period. By Lemma 5, P_i's signal cannot be lost. By Lemmas 1, 2a, and 3, this is the signal received by P_j during transaction T_s, eliminating the possibility of P_j accepting another signal instead of P_i's. Because P_j always executes *WaitForSignal* when in the WAITING state, the signal from P_i must be received and absorbed, implying P_j rendezvous with P_i. ▮

Theorem 2 *(Safety)*. If $P_i(T_r)$ commits to $P_j(T_s)$, then the following properties must be true:

1. (Mutual consent) $P_i(T_r)$ rendezvous $P_j(T_s)$ and $P_j(T_s)$ rendezvous $P_i(T_r)$. In other words, the two communicating parties agree each is rendezvousing with the other.
2. $P_j \in GuardList_i(T_r)$ and $P_i \in GuardList_j(T_s)$.
3. Communications between $P_i(T_r)$ and $P_j(T_s)$ are compatible.
4. P_i and P_j will eventually communicate, complete their transaction, and return to the RUNNING state.
5. There does not exist a third process P_k ($k \neq i$ and $k \neq j$) such that $P_k(T_t)$ rendezvous with either $P_i(T_r)$ or $P_j(T_s)$.

Proof.

1. $P_i(T_r)$ commits to $P_j(T_s)$, implying $P_i(T_r)$ signals $P_j(T_s)$ (Lemma 1). This in turn implies the mutual rendezvous according to Theorem 1.
2. The first part, $P_j \in GuardList_i(T_r)$, is trivially true because P_i would not have scanned P_j were this not the case. The second part, $P_i \in GuardList_j(T_s)$, must also be true because this condition is checked by the *CheckAndCommit* procedure *after* P_i discovers P_j is in the WAITING state. According the Lemma 4, P_j remains in the WAITING state until it receives the signal sent by P_i.
3. Compatibility is checked when $P_i(T_r)$ checks that it is in $AltList_j(T_s)$. Therefore, the proof of this part is identical to that used in part (2).
4. Once rendezvous occurs between $P_i(T_r)$ and $P_j(T_s)$, each process initiates a communication with the other. Properties (2) and (3) and the reliability assumption regarding the communication mechanism guarantee that the communication succeeds. Once this occurs, completion of the alternative operation immediately follows.
5. Suppose $P_k(T_t)$ rendezvoused with either $P_i(T_r)$ or $P_j(T_s)$. Recall a rendezvous occurs by either sending or receiving a signal to or from another process, so there are four possibilities:

   (a) $P_k(T_t)$ received a signal from $P_i(T_r)$;
   (b) $P_k(T_t)$ received a signal from $P_j(T_s)$;
   (c) $P_k(T_t)$ sent a signal to $P_i(T_r)$; or
   (d) $P_k(T_t)$ sent a signal to $P_j(T_s)$.

   However, (a) would imply P_i sent two signals during a single transaction. It is clear from the algorithm that this cannot occur. (b) and (c) imply that either P_i or P_j send and receive a signal

during a single transaction. Again, it is clear from the algorithm that this cannot occur. (d) implies P_j receives two signals during a single transaction. This is not possible because of Lemas 3 and 4. Therefore, none of these situations is possible. ∎

6.3. The Liveness Property

The liveness property guarantees that no deadlock or livelock situations can arise within the alternative algorithm. Such situations can only be caused by an erroneous *application* program. Lemmas 6–12 and Theorem 3 prove that the liveness property is maintained by the proposed algorithm.

Lemma 6. A process P_i will never return to the RUNNING state after entering a transaction unless a rendezvous occurred.

Proof. By inspection of the alternative algorithm, the process only returns to the RUNNING state when either: (a) $P_i(T_r)$ signals P_j or (b) after $P_i(T_r)$ receives a signal from P_j. In either case, $P_i(T_r)$ rendezvoused with P_j. ∎

Lemma 7. A process P_i cannot remain blocked on a Lock operation in the alternative algorithm for an unbounded amount of time.

Proof. The only *Lock* operation performed by the algorithm is to serialize accesses to *AltList*. However, no unbounded loop or blocking primitive is executed before the corresponding *Unlock* is performed. No process will remain blocked attempting to obtain a lock for an unbounded amount of time because every lock will eventually by unlocked, and the *Lock* primitive is assumed to be fair. ∎

Lemma 8. Suppose $P_i \in GuardList_j(T_s)$ and $P_j \in GuardList_i(T_r)$, and their respective I/O guards are compatible. P_i and P_j cannot both block for an unbounded amount of time in the WAITUNG state during transactions T_r and T_s, respectively.

Proof. Suppose both P_i and P_j block in the WAITING state on T_r and T_s, respectively. Because P_i reached the WAITING state, it must be the case that the *last* time P_i scanned the state of P_j before P_i entered the WAITING state, $State_j$ was either (1) RUNNING, (2) SLEEPING, or (3) WAITING but P_i failed to commit to P_j. Consider the third case. P_j must have been in a transaction *preceding* T_s for this case to apply because if P_j had been in T_s, P_j would have rendezvoused with some other process and completed T_s, contradicting our initial assumption that $P_j(T_s)$ blocked in the WAITING state for an unbounded amount of time. Therefore, if case (3)

applies, P_j must have been in a transaction *previous* to T_s when P_i observed it to be in the WAITING state.

Similarly, P_j also reached the WAITING state, so P_i must have been in the RUNNING, SLEEPING, and WAITING state for a *previous* transaction the last time P_j scanned P_i before P_j entered the WAITING state. P_i and P_j could not have both scanned each other at the same instant because each would have found each other in the ALT state. Without loss of generality, let us assume P_i scanned P_j first. $P_i(T_r)$ was in the ALT state when it scanned P_j, and because it did not rendezvous or abort (the latter would require P_j to be scanned again, making this *not* the last time P_i scanned P_j), P_i must have remained in the ALT state until it changed to the WAITING state and blocked indefinitely. Therefore, when P_j later scanned P_i for the last time, P_j must have seen P_i in either the ALT or the WAITING state for transaction T_r. However, this contradicts the fact that P_j saw P_i in the RUNNING, SLEEPING, or WAITING state for a previous transaction. Therefore, the original hypothesis that P_i and P_j both entered the WAITING state must be false. ∎

The proof of Lemma 8 relies on the fact that processes leaving the SLEEPING state abort and retry the alternative operation rather than simply resume it. If resumption were used, a race condition would exist whereby P_i and P_j might *both* enter the WAITING state.

Lemma 9. The TryAlternative procedure cannot return FAILED an unbounded number of times during a single transaction T_r in some process P_i.

Proof. Suppose the TryAlternative procedure fails an unbounded number of times. TryAlternative returns FAILED if and only if P_i scans another process P_j and finds P_j is also in the ALT state, and $TransID_j <$ $TransID_i$. The number of guards in *GuardList* is finite, so these conditions persist in (some) P_j an unbounded amount of time. Because there are only a finite number of transactions with IDs less than $TransID_i$, this condition must persist within a *single* transaction T_s. P_i will not retry until P_j leaves the ALT state, so P_j must also abort (changing to the SLEEPING state) and retry (changing back to ALT) an unbounded number of times.

Similarly, P_j will only continue to abort if some other process P_k exists which also fails within a single transaction an unbounded number of times, and $TransID_k < TransID_j$. Because the number of processes is bounded, a cycle of processes must exist such that $TransID_i > TransID_j >$ $TransID_k > \cdots > TransID_i$, which of course, cannot occur. Therefore, a process cannot fail the *TryAlternative* procedure an unbounded number of times. ∎

Lemma 10. A process P_i cannot remain continuously in the ALT state during a single transaction T_r for an unbounded amount of time.

Proof. Because *GuardList* is bounded in length, we must show that P_i does not spend an unlimited amount of time scanning a particular guard. This can only occur if some other process P_j exists such that (1) P_i continually samples P_j while $State_j$ is ALT, (2) P_j remains in the same transaction with ID $TransID_j$, and (3) $TransID_i < TransID_j$.

According to the previous lemma, P_j cannot abort and retry the alternative operation within a single transaction an unbounded number of times. Therefore, P_j must also remain *continuously* locked in the ALT state an unbounded amount of time.

An argument similar to that used in the previous lemma can now be used. P_j will only remain continuously in the ALT state an unbounded amount of time if some other process P_k is in P_j's *GuardList*, $TransID_j < TransID_k$, and P_k remains continuously in the ALT state an unbounded amount of time. A *cycle* of processes must exist such that each is waiting for the next process in the cycle to leave the ALT state. This would require that $TransID_i < TransID_j < TransID_k < \cdots < TransID_i$, so no such cycle can exist. ∎

Lemma 11. A process P_i cannot remain continuously in the SLEEPING state during a single transaction T_r for an unbounded amount of time.

Proof. P_i can only remain in the SLEEPING state an unbounded amount of time waiting for some process P_j if (1) P_i continually samples P_j while $State_j$ is ALT, and (2) P_j remains in the same transaction T_s.

These conditions can only persist if either P_j aborts and retries the transaction T_s an unbounded number of times, or P_j remains continuously in the ALT state for an unbounded amount of time. Lemmas 9 and 10 proved that neither is possible, so P_i cannot remain in the SLEEPING state an unbounded amount of time. ∎

Lemma 12. For each alternative operation initiated by P_i, P_i eventually either rendezvous with some other process P_j and returns to the RUNNING state or moves to the WAITING state.

Proof. The only way a process can *not* reach the WAITING state or rendezvous is to remain continually in the ALT state, remain continually in the SLEEPING state, or switch back and forth between ALT and SLEEPING an unbounded number of times. The latter case implies *TryAlternative* fails an unbounded number of times within a single transaction. None of these is possible according to Lemmas 9–11. ∎

Theorem 3 *(Liveness).* Suppose two processes P_i and P_j each initiate an alternative operation and $P_j \in GuardList_i(T_r)$ and $P_i \in GuardList_j(T_s)$ and their communication requests are compatible. If neither P_i nor P_j rendezvous with another process during their respective transactions, P_i and P_j will eventually rendezvous with each other during T_r and T_s, respectively.

Proof. According to Lemma 12, P_i and P_j must each eventually either rendezvous or enter the WAITING state. They both cannot enter the WAITING state according to Lemma 8. Therefore, at least one of the two processes, say P_i, must rendezvous. By assumption, P_i cannot rendezvous with any process other than P_j, so P_i must rendezvous with P_j. By Theorem 2, P_j must also rendezvous with P_i. Therefore, P_i and P_j must eventually rendezvous with each other. ∎

7. FAIRNESS

One issue regarding the alternative construct that has received considerable attention is *fairness.* In particular, two types of fairness, *weak* and *strong* fairness, have been defined in Refs. 20 and 22. We call an implementation of the alternative construct *weakly fair* if it can be guaranteed that during the infinitely repetitive execution of an alternative command, a guard that remains *continuously* available (i.e., enabled and the neighboring process is ready to communicate) will eventually rendezvous. An implementation is said to be strongly fair if the implementation guarantees that any guard which is available *infinitely often* (though not necessarily continuously as is the case in weak fairness) will eventually rendezvous.

The algorithm shown in Figs. 2–4 is not fair in either the weak or strong sense. However, weak fairness can be achieved by modifying the algorithm so that the order in which the *TryAlternative* procedure scans guards, which implies a certain prioritization of the guards, varies from one call to the next so that each guard is eventually scanned first. More precisely, we modify the algorithm as follows:

- Define a distinct integer variable for each alternative construct in a given CSP program. These variables could be defined by the compiler. Associate with the kth alternative construct in process P_i the variable $Alt_{i,k}$. Initially set to 0, this variable is incremented each time this particular alternative construct is invoked. It therefore indicates the number of times P_i has invoked the corresponding alternative construct.
- The FOR loop in the *TryAlternative* procedure is modified so that it begins scanning guard $(Alt_{i,k} \bmod n) + 1$ rather than the first

guard, where n is the number of guards in the alternative construct. The FOR loop is also modified to skip disable guards. If executes up to n iterations as before. The index variable of the FOR loop "wraps around" to 1 after scanning the nth guard.

The modified algorithm is referred to as the *Fair Algortihm*, and is assumed in the discussion which follows.

Theorem 4 *(Fairness).* Let P_i be blocked on an alternative operation (i.e., P_i is in the WAITING state) in which some proces P_j is listed in some enabled guard. Further, let us assume P_i does not become unblocked through a rendezvous with any process other than P_j. Consider an alternative construct A in P_j that has been executed u times and contains n guards, one of which (g_r) contains a compatible communication with P_i. If P_j now executes A at least n more times and g_r is enabled on each of these n invocations of A, then P_i and P_j will rendezvous before the $(u+n)$th execution of A completes.

Proof. The theorem can be proven by contradiction. Suppose P_i does not rendezvous with P_j before the $(u+n)$th execution of A. For this to happen, P_j must continually be rendezvousing with some other process(es) before it scans P_i because the moment it scans P_i, it will see that P_i is in the WAITING state and rendezvous with P_i. However, the *Fair Algorithm* guarantees that within n executions of A, g_r will become the *first* guard that is scanned. When g_r is scanned first, no other process can rendezvous with P_j before P_j scans P_i, so a rendezvous between P_i and P_j must take place. ∎

The following corollary follows immediately from this theorem:

Corollary 1. In an infinitely repetitive execution of an alternative construct, a guard cannot remain continually available for an unbounded amount of time without eventually rendezvousing.

This shows that the *Fair Algorithm* is weakly fair. It demonstrates, for instance, that a process waiting to be served by another process cannot be continuously denied service for an unbounded amount of time. The *Fair Algorithm* is *not* strongly fair, however. Modification of this algorithm to one which is strongly fair is an open question. None of the alternative algorithms that have been developed thus far (based on message-passing architectures) is strongly fair.

8. CONCLUSIONS

We have presented an algorithm that implements the generalized alternative construct in CSP. Unlike previous algorithms, it is based on a

shared memory architecture. It has been shown that the algorithm maintains the safety and liveness properties required by any correct implementation. Extensions to the algorithm that allow processes to terminate and guarantee weak fairness were also presented. An implementation, written in *C*, has been developed for a 18-processor BBN Butterfly parallel processor. Empirical performance evaluation of this implementation is in progress.

ACKNOWLEDGMENT

The authors wish to thank the anonymous referees for their helpful comments on earlier versions of this paper.

REFERENCES

1. C. A. R. Hoare, Communicating Sequantial Process, *Communications of the ACM* **21**(8):666–677 (August 1978).
2. C. A. R. Hoare, *Communicating Sequantial Processes, Computer Science*, Prentice Hall (1985).
3. E. W. Dijkstra, Guarded Commands, Nondeterminism and Formal Derivation of Programs, *Communications of the ACM* **18**(8):453–457 (August 1975).
4. A. A. Aaby and K. T. Narayana, A Distributed Implementation Scheme for Communicating Processes, *Proceedings of the International Conference on Parallel Processing*, pp. 942–949 (August 1986).
5. R. Bagrodia, A Distributed Algorithm to Implement the Generalized Alternative Command in CSP, *The 6th International Conference on Distributed Computing Systems*, pp. 422–427 (May 1986).
6. A. J. Bernstein, Output Guards and Nondeterminism in *Communicating Sequantial Processes, ACM Transactions on Programming Language and Systems* **2**(2):234–238 (April 1980).
7. G. N. Buckley and A. Silberschatz, An Efficient Implementation for the Generalized Input–Output Construct of CSP, *ACM TOPLAS* **5**(2):223–235 (April 1983).
8. M. Collado, R. Morales, and J. J. Moreno, A Modula-2 Implementation of CSP, *ACM SIGPLAN Notices* **22**(6):25–37 (June 1987).
9. *OCCAM Programming Manuael*. Inmos Ltd. (1982).
10. R. B. Kieburtz and A. Silberschatz, Comments on Communicating Sequential Processes, *ACM Transactions on Programming Language and Systems* **1**(2):218–225 (October 1979).
11. Z. Sun and X. Li, CSM: A Distributed Programming Language, *IEEE Transactions on Software Engineering* **SE-13**(4):497–500 (April 1987).
12. D. R. Jefferson, Virtual Time, *ACM Transactions on Programming Languages and Systems* **7**(3):404–425 (July 1985).
13. J. Misra, Distributed–Discrete Event Simulation, *ACM Computing Surveys* **18**(1):39–65 (March 1986).
14. D. A. Reed, A. D. Malony, and B. D. McCredie, Parallel Discrete Event Simulation: A Shared Memory Approach, *Proceedings of the ACM SIGMETRICS Conference on Measuring and Modeling Computer Systems* **15**(1):36–38 (May 1987).
15. B. Thomas, et al., *Butterfly Parallel Processor Overview*, BBN Report No. 6148, BBN Laboratories Incorporated (March 1986).

16. G. F. Pfister, et al., The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, *Proceedings of the International Conference on Parallel Processing*, pp. 764–771 (August 1985).
17. D. D. Gajski, D. H. Lawrie, D. J. Kuck, and A. H. Sameh, Cedar, *COMPCON-84, IEEE Computer Society Conference*, pp. 306–309 (February 1984).
18. D. A. Reed and R. M. Fujimoto, *Multicomputer Networks: Message-Based Parallel Processing. Computer Science*, MIT Press (1987).
19. R. A. Karp, Proving Failure-Free Properties of Concurrent Systems Using Temporal Logic, *ACM Transactions on Programming Language and Systems* **6**(2):239–253 (April 1984).
20. N. Francez, *Fairness*, Springer–Verlag, New York (1986).
21. S. Owicki and L. Lamport, Proving Liveness Properties of Concurrent Programs, *ACM Transactions on Programming Language and Systems* **4**(3):455–495 (July 1982).
22. D. Zobel, Transformations for Communication Fairness in CSP, *Information Processing Letters* **25**:195–198 (May 1987).

Printed by Catherine Press, Ltd., Tempelhof 41, B-8000 Brugge, Belgium