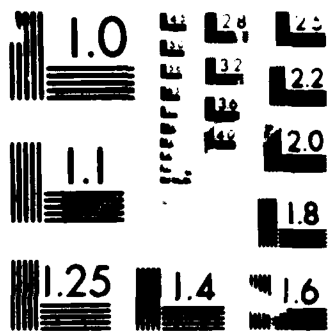
AD-A187 820 A SHARED MEMORY ALGORITHM AND PROOF FOR THE ALTERNATIVE 1/1
CONSTRUCT IN CSP (.. (U) UTAH UNIV SALT LAKE CITY DEPT
OF COMPUTER SCIENCE R M FUJIMOTO ET AL. 25 AUG 87
UNCLASSIFIED UUCS-87-021 N00014-87-K-0184 F/G 12/7 NL



MICROCOPY RESOLUTION TEST CHART

A Shared Memory Algorithm and Proof for the Alternative Construct in CSP

Richard M. Fujimoto
Hwa-chung Feng

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF UTAH

UUCS-87-021

TECHNICAL REPORT

Department of Computer Science

University of Utah
Salt Lake City, Utah

12

A Shared Memory Algorithm and Proof for the Alternative Construct in CSP

Richard M. Fujimoto
Hwa-chung Feng

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF UTAH

UUCS-87-021

A Shared Memory Algorithm and Proof for the Alternative Construct in CSP

Richard M. Fujimoto[1]
Hwa-chung Feng

Department Of Computer Science
University Of Utah
Salt Lake City, UT 84112

August 25, 1987

[1]This work was supported by ONR contract number N00014-87-K-0184.

Abstract

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. 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 alternative construct that exploits the capabilities of a parallel computer with shared memory. The algorithm assumes a generalized version of Hoare's original alternative construct that allows output commands to be included in guards. A correctness proof of the proposed algorithm is presented to show that the algorithm conforms to some *safety* and *liveness* criteria. Extensions to allow termination of processes and to ensure fairness in guard selection are also given.

Keywords: 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 [11,10]. 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 (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[6]. This construct enables a process to *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 [1,2,3,4,5,12,15,22]. It has been argued that the exclusion of output guards in the original definition of CSP is too restrictive and sometimes degrades performance [3,15]. Algorithms that allow output guards in the alternative construct have been proposed[1,2,3,4]. Others suggest a paradigm similar to that which was originally proposed [9,12,22]. 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 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 applications areas that are well suitable for shared memory machines. For example, distributed discrete event simulation algorithms are usually described in terms of message passing paradigms [13, 16], and implementations on shared memory architectures have been described [21]. Similarly, message passing is used extensively in object-oriented programming.

- Shared memory machines are widely available. Multiprocessors such as the BBN Butterfly™ [23] and Sequent Balance™ are available from the commercial sector, and numerous shared memory research machines such as IBM's RP3 [18] and the University of Illinois's Cedar [8] 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™ [20]. 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 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 CSP alternative construct that exploits the facilities afforded by shared memory and avoids the aforementioned system processes. This algorithm implements the "generalized" alternative construct that allows output guards.

The proposed algorithm uses the notion of total ordering among processes [3] to prevent deadlocks, but applies this principle *dynamically* on transactions (defined later) rather than statically as originally proposed. The shared memory architecture simplifies the task of maintaining globally unique IDs. 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. Therefore, we provide a proof of the correctness of the algorithm according to *safety* and *liveness* criteria [14]. Modifications are also suggested to achieve fairness [7].

Finally, the algorithm does not contain any inherent *hot spots* [19]. The few global variables that are shared by all processes are not accessed with sufficient frequency to constitute a hot spot. With the exception of these global variables, the algorithm is fully distributed and does not rely on any centralized controller.

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

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

the predicate to the left of the '→' 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:

$$[G_{i(i \in PB)} \;\rightarrow\; S_i \;\Box\; G_{j(j \in IO)} \;\rightarrow\; S_j].$$

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 primitives 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 $P_1, P_2, \ldots, P_N$. 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 blocked 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* primitives 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.

- **Sleep(T)** causes the process invoking it to block for at least T time units. A process will always eventually awake after calling *Sleep*.

- The amount of time between successive samples of a shared memory location by a busy wait loop (which does nothing but sample and test the value stored in this location for inequality) can be bounded, and is shorter than the time required to invoke either the *Send* or *Recv* primitives defined above.

This final "timing" assumption is perhaps the most distasteful aspect of the proposed algorithm. It is not necessary to ensure the safety of the algorithm, i.e., if it were relaxed, no "invalid" rendezvous will result. The assumption is primarily a theoretical requirement that is necessary to prove liveness and has only limited practical implications. If this assumption is relaxed, specific scenarios requiring a prolonged, highly synchronous behavior between independent processes must develop to violate liveness. Such scenarios are unlikely to occur in practice, as will be discussed in detail after the algorithm has been described, and precautions can be take to reduce the likelihood of such occurrences if the timing assumption cannot be guaranteed.

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. 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 which each 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. After aborting, the process goes to sleep for some predetermined period of 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 figure 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 must pass 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.

One procedure merits special attention. **CheckAndCommit(AltList$_r$, 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_r (r denotes the *remote* process), and if so, to attempt to commit to P_r. If a commit was attempted and succeeded, then *CheckAndCommit* returns a positive integer indicating the corresponding guard in the *remote* process P_r. Otherwise, *CheckAndCommit* returns a non-positive integer, denoted by the constant FAILED. This procedure is shown in figure 2.

CheckAndCommit uses a procedure **CheckGuard(AltList$_r$, g$_i$): INTEGER** that scans the remote alternative list $AltList_r$ 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. *CheckGuard* returns j, 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_r by testing if $WakeUp_r$ is zero, and if so, incrementing it. An ordinary addition is used rather than the *AtomicAdd* primitive to increment $WakeUp_r$ because $AltLock_r$ guarantees atomicity — every "test-and-set" operation performed on $WakeUp_r$ occurs while $AltLock_r$ is set. If P_l is the first process to commit to P_r, i.e., if $WakeUp_r$ was previously zero, then P_l successfully commits. *CheckAndCommit* returns the number of the corresponding guard, and rendezvous is imminent. Otherwise, *CheckAndCommit* returns FAILED. $AltLock_r$ ensures serial access to $AltList_r$. 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.

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.
- **CommunicantID(g_i)** is a function that returns the ID of the process listed in the I/O command portion of guard g_i.
- **Communicate(g_i)** executes the I/O command in guard g_i.
- **TimeOut** is a constant indicating the number of time units a process should sleep after an aborted attempt. More will be said about this later.

4.4 Description Of The Algorithm

The alternative algorithm is shown in figures 3 and 4. The *Alternative* procedure shown in figure 3 is a "front end" that is responsible for retrying aborted attempts. The heart of the algorithm is in the *TryAlternative* procedure shown in figure 4. The parameters passed to both procedures 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 is only called after non I/O guards have been evaluated and are found to be FALSE. This procedure does not return until a rendezvous has been completed at which time it returns an integer indicating the guard ($g_1, g_2, \ldots, g_n$) that was eventually satisfied.

The *Alternative* procedure obtains a unique transaction ID by performing an *AtomicAdd* operation on the global *NextTransID* variable. It then attempts to rendezvous by calling *TryAlternative*. *TryAlternative* either returns the number of the guard on which a rendezvous occurred, or the FAILED flag indicating the attempt must be retried. Each time *TryAlternative* fails, the process enters the SLEEPING state for at least *TimeOut* time units before retrying. The same transaction ID remains in use despite one of more failed attempts. It will be shown that *TryAlternative* cannot fail an unbounded number of times within a single transaction.

The heart of the alternative algorithm is embodied in the *TryAlternative* procedure (figure 4). In this procedure, l refers to the local process P_l, and r refers to the remote process P_r associated with the guard that is being scanned.

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_r.

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

If P_r 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_r) will eventually retry its alternative operation. If P_l and P_r are destined to eventually rendezvous on this transaction, P_l will typically proceed to the WAITING state, and P_r will later retry, commit, and rendezvous with P_l.

If P_r is WAITING, then P_r has already reached the rendezvous point so P_l attempts to rendezvous. $AltList_r$ 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_r (by sending a signal) and rendezvous. Otherwise, P_l advances to the next guard.

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

If two processes see each other in the ALT state, one will be forced to abort and retry the alternative, while the other pauses within the current operation until the first aborts. The transaction IDs of the two processes are used to determine the process that will abort and the process that will proceed. A process with a smaller, i.e., older, transaction ID is given higher priority. This protocol avoids deadlock situations in which two processes attempting to communicate with each other both advance to the WAITING state.

If the process does not abort, it pauses in a busy wait loop until the remote process moves out of the ALT state. The remote process will either abort, changing to the SLEEPING state, or rendezvous.

changing to the RUNNING state. Later, it will be shown that one of these two possibilities must eventually occur. 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_r may change immediately after P_l examines $State_r$. It will be proven that the algorithm operates correctly despite this potential 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 is 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 figures 2, 3, and 4 be performed in *exactly* the order in which they appear. Seemingly minor changes such as swapping the order of the statements

```
WakeUpl := 0;
Statel := 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. This is guaranteed by the locking protocol used in *CheckAndCommit*.

5 Discussion

Several aspects of the alternative algorithm presented above 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 is done in figure 3, and *not* when an existing operation is retried. The use of dynamic transaction IDs is further justified by the fact that global variables are relatively inexpensive in shared memory architectures, and the *NextTransID* variable is not referenced with sufficient frequency to become a hot spot.

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 The Timing Assumption

We earlier required the following assumption to ensure liveness:

> The amount of time between successive samples of a shared memory location by a busy wait loop (which does nothing but sample and test the value stored in this location for inequality) can be bounded, and is shorter than the time required to invoke either the *Send* or *Recv* primitives.

This assumption is necessary because the algorithm uses a polling loop to detect another process leaving the ALT state. Suppose P_i is waiting for P_j to change to a new state. It is possible, albeit unlikely, that P_j (1) modifies $State_j$, (2) rendezvous and resumes execution of user code or goes to sleep for *TimeOut* units of time, and (3) reenters *TryAlternative* and changes $State_j$ back to ALT; all of this must occur *without* P_i noticing $State_j$ had been modified, so this activity must occur *between* successive samples of $State_j$ by P_i's polling loop. While it is true that this might occasionally occur if P_i is interrupted during its polling loop, it is necessary that this scenario be repeated ***an unbounded number of times*** within a single execution of the polling loop to compromise the liveness of the algorithm. We conjecture that it is highly improbable that such a scenario will occur even a few times within a single transaction. Further, we emphasize that safety remains guaranteed even if the above assumption is relaxed, so no ill effects, other than delays, will result should this scenario occur some (finite) number of times.

As can be seen from figure 4, P_j must execute either the *Sleep*, *Send*, or *Recv* primitive *after* the state of P_j is changed (to SLEEPING or RUNNING), i.e., during step (2) above. Therefore, as stated in the above assumption, ensuring that the minimum execution time of each of these primitives exceeds the time between successive samples of P_i's polling loop is sufficient to avoid the above scenario (actually, the *Sleep* primitive is excluded because its minimum execution time is trivially set). If the time between successive samples of the polling loop can be bounded, the minimum amount of time required by the *Send* and *Recv* primitives can be easily modified to adhere to the timing assumption through the introduction of a timed delay (e.g., by calling *Sleep*). However, one would not expect introduction of such a delay to be necessary in most practical situations.

Assuming the time required by a remote memory reference is bounded, the time between successive samples by the busy wait loop can be bounded by disabling interrupts during the polling loop. If this is not a viable alternative, one can reduce the likelihood of entering the above scenario by introducing randomness into the program's temporal behavior. For example, a random sleeping period may be selected (with some minimum value, as described below) when a process is forced to abort. This will reduce the likelihood of excessive delays caused by synchronized behavior between processes.

5.3 Setting the Sleeping Period

The "sleep period" before a retry is attempted, i.e., *TimeOut* in figure 4, must be sufficiently long to allow the "winning" process to observe that the sleeping process is indeed in the SLEEPING state. In particular, *TimeOut* cannot be shorter than the interval between successive samples in the busy wait loop executed by the winner.

On the other hand, an excessively long sleeping period will lead to an inefficient implementation. A reasonable *TimeOut* value is the time required for a few remote memory references.

5.4 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.

5.5 Termination

Termination is another important issue facing real implementations. This was not treated in the previous discussion because it introduces obscurities into the description. The termination semantics play an important role in CSP because it is the basis of the termination of the *repetitive* command [11]. 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 for 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 communicate 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 figure 4). Identical constraints 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 figure 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* [14,17]. As described above, 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 figures 2, 3, and 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. $P_i(T_r)$ denotes that fact that P_i is in T_r. 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. **The function $\mathbf{prev(T_r)}$ returns the ID of the transaction executed by the process which immediately preceded T_r. The existence of T_r implies the termination of prev(T_r). Also, $prev^0(T_r)$ refers to T_r itself and $prev^m(T_r)$ corresponds to the *mth* previous transaction entered by P_j.**

8. $\mathbf{GuardList_j(T_r)}$ lists the guards that are passed as parameters to the alternative operation executed by P_j 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 through 5 lead to theorem 1 which states 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. ∎

This lemma implies that $WakeUp_j$ must be set to 0 before a signal can be sent to P_j. In addition, at most one signal is sent to P_j each time $WakeUp_j$ is set to 0.

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 m, the number of transactions entered by P_j.
>
> Consider the first transaction ($m = 1$) executed by P_j. $WakeUp_j$ is initialized to 1. Because $WakeUp_j$ can only be set to 0 by P_j during a transaction, $WakeUp_j$ must remain nonzero up to at least the beginning of P_j's first alternative operation. No process can commit to P_j until $WakeUp_j$ becomes 0, so by lemma 1, no signals can be sent to P_j before its first transaction, and therefore none can be pending. Thus, (a) and (b) are both true at the beginning of P_j's first transaction.
>
> During any transaction, and in particular the first, P_j will either reset $WakeUp_j$ to 0 exactly once (just before entering the WAITING state), or not at all. If P_j does not reset $WakeUp_j$, then obviously $WakeUp_j$ is still nonzero at the end of the alternative operation. No signal can be sent to P_j because no process can commit, so none are pending.
>
> If P_j does reset $WakeUp_j$ to 0, then at most one process can commit (and send a signal) to P_j during this transaction. This is because (1) $WakeUp_j$ is set to 0 at most one time during this transaction; (2) each process must obtain the lock $AltLock_j$ before it can examine $WakeUp_j$ (see the *CheckAndCommit* procedure); (3) as soon as one process reads a zero in $WakeUp_j$, it increments it *before* releasing $AltLock_j$; so (4) two processes cannot both read a zero value from $WakeUp_j$ during a single transaction in P_j. Because

no two processes can see a zero value in $WakeUp_j$ during a single transaction, no two processes can commit to P_j during this (or any) transaction. Therefore, according to lemma 1, at most one signal will be sent to P_j during this transaction.

P_j always calls *WaitForSignal* after setting $WakeUp_j$ to zero. Therefore, the only signal that could have been sent to P_j must have been absorbed by the *WaitForSignal* operation, so none can be pending when the transaction completes (if it completes) satisfying condition (a). Condition (b) must also be satisfied at the end of the transaction because a process must commit *before* sending a signal to P_j, so $WakeUp_j$ must be nonzero before the process can resume execution after calling *WaitForSignal*. 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 mth transaction entered by P_j. We will now show it is also true on the $m+1st$ transaction. According to the inductive hypothesis, no signals are pending at the end of the mth operation, and $WakeUp_j$ is nonzero. Therefore, these conditions will remain true until the beginning of the $m+1st$ transaction because no process can commit to P_j until $WakeUp_j$ becomes 0. As noted in the proof for $m = 1$, if (a) and (b) are true at the beginning of any transaction, they will be true at the end of the transaction if it terminates. Therefore, (a) and (b) are true at the end of the $m+1st$ transaction entered by P_j. ∎

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. The proof relies on the fact that $WakeUp_k$ is not zero at the beginning of the alternative operation and can be set to zero at most one time during a single transaction. The atomicity of the commit operation (i.e., two read-modify-write sequences cannot be inappropriately interleaved) guarantees that only a single process can commit to P_k during T_t.

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: According to the algorithm, P_i checks 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 when P_i commits, as well as when the signal is received. This must be the case.

however, because once P_j enters the WAITING state, it cannot change 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 this transaction. 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.

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

Proof: By lemma 2a, no signals are pending at the beginning of each transaction. By lemma 3, at most one process can commit during a transaction, so at most one signal is sent (and therefore received) during a transaction. Thus, a signal can never arrive during a transaction while another has already been received but is still pending, so no signals are ever lost during a transaction.

No signals destined for a process P_j are lost between successive transactions of P_j because none can be sent to P_j while it is in the RUNNING state. This is true because (1) a signal is only sent to P_j following a commit operation (lemma 1), (2) Pj must have been in the WAITING state when the commit occurred (lemma 4), and (3) P_j must remain in the WAITING state until the signal is received and absorbed by a *WaitForSignal* operation (lemma 4). ∎

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 only 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, 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 $P_i(T_r)$ or $P_k(T_t)$ rendezvous with $P_j(T_s)$.*

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, showing $P_j \in GuardList_i(T_r)$, can be proved by contradiction. Suppose $P_j \notin GuardList_i(T_r)$. Then P_i would not have committed to P_j because P_i only scans those processes in $GuardList_i(T_r)$ (see the **FOR** loop in the *TryAlternative* procedure), contradicting our original assumption that P_i committed to P_j.

   It only remains to be proven that $P_i \in GuardList_j(T_s)$. It is seen from the algorithm that P_i checks $AltList_j$ just before committing to P_j, and $AltList_j$ is set to hold $GuardList_j(T_s)$ just before P_j enters the WAITING state, and therefore before the commit. However, an arbitrarily long delay may elapse from the time P_i checked $AltList_j$ to the time it committed. We therefore need to confirm that the value of $AltList_j$ that P_i checked is $GuardList_j(T_s)$ rather than $GuardList_j(prev^m(T_s))$ for some $m > 0$.

   This will be proven by contradiction. Suppose P_i checked $GuardList_j(prev(T_s))$. This would imply that the following sequence of events must have occurred:

   (a) $P_i(T_r)$ checks $GuardList_j(prev(T_s))$ (stored in $AltList_j$);

   (b) $P_j(T_s)$ modifies $AltList_j$ so that it becomes $GuardList_j(T_s)$;

   (c) $P_j(T_s)$ sets $WakeUp_j(T_s)$ to 0; and

   (d) $P_i(T_r)$ commits to $P_j(T_s)$.

   Event (a) must take place by the aforementioned assumption, and event (d) must take place by our original assumption that $P_i(T_r)$ commits $P_j(T_s)$. Event (c) must precede (d) because $WakeUp_j(T_s)$ must be reset to 0 before any commitment to $P_j(T_s)$ can occur (see definition of commit). Event (b) must precede (c) according to the order in which operations are performed in the algorithm. Event (b) must follow (a) in order to satisfy our supposition that P_i checked $GuardList_j(prev(T_s))$. However, this sequence of events is not possible because the locking protocol of the procedure *CheckAndCommit* (used by P_i when checking $AltList_j$) ensures that $AltList_j$ is not modified after P_i checks it (event (a) above), but before P_i commits (event (d)). Therefore, event (b) could not have occurred between (a) and (d), so our assumption that $P_i(T_r)$ examined $GuardList_j(prev(T_s))$ must be incorrect. Similarly, it is not possible that $P_i(T_r)$ examined $GuardList_j(prev^m(T_s))$ for any $m > 0$.

3. Compatibility is checked when $P_i(T_r)$ checks that it is in $AltList_j(T_s)$. Similarly, this information is implicitly updated whenever $AltList_j$ is updated. Therefore, this condition is satisfied using the same proof as was used in (2) to show P_i is in $GuardList_j(T_s)$.

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) above 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 (definition of rendezvous), 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 $P_j(T_s)$.

   We need not consider signals sent before T_r, T_s, or T_t but received during these respective transactions because none can be pending when the transaction begins (lemma 2a).

   (a) Suppose $P_k(T_t)$ rendezvoused because it received a signal from P_i *during* T_r (signals generated by P_i outside T_r are not relevant). This implies $P_i(T_r)$ sent signals to *two* processes because our original assumption is that $P_i(T_r)$ committed to (and therefore signaled according to lemma 1) $P_j(T_s)$. It is clear from the algorithm that a process can signal at most one other process on any given transaction because any time a signal is generated, the transaction always completes without calling the *Signal* procedure again (see figure 4). Therefore, $P_k(T_t)$ could not have received a signal from $P_i(T_r)$.

(b) Suppose $P_k(T_t)$ received a signal from P_j *during* T_s (signals generated by P_j outside T_s are not relevant). This implies $P_j(T_s)$ both sent a signal to P_k and received a signal from P_i within a single transaction. If $P_j(T_s)$ sent a signal, then, according to the algorithm in figure 4, P_j must have rendezvoused and completed the transaction without ever entering the WAITING state or setting $WakeUp_j(T_s)$ to zero. This contradicts our original assumption that $P_i(T_r)$ committed to $P_j(T_s)$.

(c) Suppose $P_k(T_t)$ signaled $P_i(T_r)$. This implies $P_i(T_r)$ both sent a signal to P_j and received a signal from P_k within a single transaction. This latter signal must have been preceded by $P_k(T_t)$ committing to P_i (lemma 1). This commit must have occurred *during* or *before* T_r. But, $P_k(T_t)$ could not have committed to P_i during T_r because $WakeUp_i$ is never equal to zero during T_r. This is because, by assumption, $P_i(T_r)$ commits to $P_j(T_s)$, so $P_i(T_r)$ never enters the WAITING state (It is only then that the *WakeUp* variable is set to 0.) Also, $P_k(T_t)$ could not have committed to P_i before T_r and signaled P_i during T_r because this would violate lemma 4. Therefore $P_k(T_t)$ could not have sent a signal to $P_i(T_r)$.

(d) Finally, $P_k(T_t)$ could not have committed (and therefore could not have signaled) P_j *during* T_s because this would imply both P_k and P_i committed to P_j within a single transaction, violating lemma 3. $P_k(T_t)$ could not have committed to P_j before T_s and signaled P_j during T_s because this would again violate lemma 4. Thus, $P_k(T_t)$ could not have signaled $P_j(T_s)$ either. Therefore, $P_k(T_t)$ could not have rendezvoused with either $P_i(T_r)$ or $P_j(T_s)$, so the proof is complete. ∎

Note from the proof of (2) in the Safety theorem that it is crucial that accesses to *AltList* are controlled by locks, and that the act of checking the *AltList* and committing is atomic to ensure correct operation. Also note that the status of P_j may change immediately after P_i checks it. The algorithm operates correctly despite this inconsistency.

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 through 11 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 $P_j(T_s)$ or (b) after $P_i(T_r)$ receives a signal from $P_j(T_s)$. In either case, $P_i(T_r)$ rendezvoused with $P_j(T_s)$. ∎

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, once any process obtains a lock on any *AltList*, it must eventually release that lock because no unbounded loop or blocking primitive is executed before the corresponding *Unlock* is performed. Therefore, the lock cannot remain in place for an unbounded amount of time. No process will remain blocked attempting to obtain a lock for an unbounded amount of time because every lock will eventually be unlocked, and the 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 enter the* WAITING *state during transactions T_r and T_s, respectively.*

Proof: Proof by contradiction. Suppose both P_i and P_j enter 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 (If P_i successfully committed, they would have rendezvoused and completed the transaction according to theorem 2.) Consider the third case. We will now show that P_j must have been in a transaction *preceding* T_s for this case to apply. $WakeUp_j(T_s)$ is set to 0 *before* $State_j$ is set to WAITING. Therefore, if P_i saw P_j in the WAITING state while P_j was in transaction T_s, and P_i failed when it tried to commit, then it must be that some third process must have committed to P_j *during* T_s (after $WakeUp_j(T_s)$ is set to 0 but before P_i attempted to commit). But this successful commit must have resulted in a rendezvous, contradicting our original assumption that P_j blocked indefinitely in the WAITING state while in T_s. 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, or 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. Therefore, one scanned the other first. 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. ∎

Lemma 9 *A process P_i cannot remain continuously in the* ALT *state during a single transaction T_r for an unbounded amount of time.*

Proof: A process remains in the ALT state while it is scanning the processes in its *GuardList* trying to find one which is ready to rendezvous. If none is found, the process proceeds to the WAITING state. Because *GuardList* is necessarily bounded in length, we must show that a process does not spend an unlimited amount of time scanning a particular guard.

P_i moves on to the next *GuardList* entry or eventually changes state when it finds the process corresponding to the current guard is in either the SLEEPING, RUNNING, or WAITING state. Therefore, we only need to consider scanning a process P_j which is also in the ALT state. If $TransID_j < TransID_i$, then P_i aborts *TryAlternative* and changes to the SLEEPING state. Thus we need only examine the case $TransID_i < TransID_j$ (both cannot have the same ID). In this case, P_i enters a loop waiting for $State_j$ to change. In order for P_i to remain in this loop an unbounded amount of time, P_i must continually sample P_j while $State_j$ is ALT. There are three ways P_i's samples can indicate P_j remains in the ALT state for an unbounded amount of time: (1) P_j is also locked into the ALT state for an unbounded amount of time; (2) P_j repeatedly aborts *TryAlternative*, changes to the SLEEPING state, and then retries *TryAlternative* (changing back to the ALT state) in perfect synchrony with P_i's samples of $State_j$; or (3) P_j repeatedly rendezvous, changes to the RUNNING state, and then initiates a new alternative operation in perfect synchrony with P_i's samples of $State_j$. These are exhaustive because a process can only return from *TryAlternative* after a rendezvous or after an aborted attempt. Case (2) cannot occur, however, because the sleep period is set to a time sufficiently large that successive samples by P_i will detect that P_j is in the SLEEPING state. Similarly, case (3) cannot occur because the minimum execution time of the *Send* and *Recv* primitives are assumed to be larger than the time between successive samples of the polling loop. Therefore, only case (1) remains.

The previous discussion shows that P_i can only remain in the ALT state scanning P_j an unbounded amount of time if the following conditions hold: (1) $TransID_i < TransID_j$, and (2) P_j remains continuously in the ALT state on the same transaction an unbounded amount of time. By the same argument presented above, P_j will only remain in the ALT state on a single transaction 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. Continuing this logic, because the number of processes is bounded, the original process P_i will only remain in the ALT state for an unbounded time if a *cycle* of processes exists such that each is waiting for the next process in the cycle to leave the ALT state. This would require that $TransID_i < TransID_j < TransID_k < \cdots < TransID_i$, which is clearly not possible. Therefore, no such cycle can exist, so P_i *cannot remain continually* in the ALT state for an unbounded amount of time. ∎

Lemma 10 *The* TryAlternative *procedure cannot return* FAILED *an unbounded number of times during a single transaction* T_r *in some process* P_i.

Proof: *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 if *TryAlternative* fails an unbounded number of times, it must be that for some process P_j, the conditions $State_j =$ ALT and $TransID_j < TransID_i$ persist for an unbounded amount of time.

P_j cannot remain continually in the ALT state for an unbounded amount of time in a single transaction (lemma 9). Therefore, it must be the case that either (1) P_i finds P_j in the ALT state for a *different* transaction an unbounded number of times; or (2) within a single transaction, P_j repeatedly switches back and forth between the ALT and SLEEPING states for an unbounded number of times, and it so happens that every time P_i retries *TryAlternative* and scans P_j, P_i finds that P_j is in the ALT state. In case (2) *TryAlternative* must fail an unbounded number of times in P_j as well as P_i

Case (1): This is not possible because each new transaction ID is larger than all previous IDs. If P_i finds P_j in the ALT state for a new transaction an unbounded number of times, this would imply there are an unbounded number of transaction IDs less than $TransID_i$. This cannot be the case because transaction IDs are positive integers.

Case (2): An argument similar to that used in lemma 9 can be used here. Summarizing the arguments presented thus far in this lemma, *TryAlternative* in P_i will only fail an unbounded number of times if it also fails an unbounded number of times in some other process P_j, where $TransID_j < TransID_i$. Similarly, P_j will only continue to fail if some other process P_k exists which also continues to fail, 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 11 *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. In other words, a process cannot remain in the* ALT *state in the same transaction for an unbounded amount of time.*

Proof: The only way a process can *not* reach the WAITING state or rendezvous is to remain continually in the ALT 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. Neither is possible according to lemmas 9 and 10. ∎

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 11, 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 [7,24]. 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 figures 2, 3, and 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:

- The *Alternative* and *TryAlternative* procedures each receive *all* guards specified in the alternative command as parameters. The original procedures assumed only enabled guards are passed.

- A boolean flag is associated with each guard indicating whether or not it is enabled.

- 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 *mth* alternative construct in process P_i the variable $Alt_{i,m}$. Initially set to 0, this variable is incremented each time this particular alternative construct is executed. 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 $(Alt_{i,m} \bmod n) + 1$ rather than the first guard, where n is the number of guards in the alternative construct. The **FOR** loop is also modified to skip disabled guards. It 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 Algorithm*, 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 process 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 m times and contains n guards, one of which (g_v) contains a compatible communication with P_i. If P_j now executes A at least n more times and g_v is enabled on each of these n invocations of A, then P_i and P_j will rendezvous before the $(m+n)th$ execution of A completes.*

Proof: The theorem can be proved by contradiction. Assume P_i does not rendezvous with P_j before the $(m+n)th$ 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_v will become the *first* guard that is scanned. When g_v 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, this 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 16-processor BBN Butterfly parallel processor. Empirical performance evaluation of this implementation is in progress.

References

[1] A. A. Aaby and K. T. Narayana. A Distributed Implementation Scheme For Communicating Processes. *Proceedings of the 1986 International Conference On Parallel Processing*, 942–949, August 1986.

[2] R. Bagrodia. A Distributed Algorithm To Implement The Generalized Alternative Command In CSP. *The 6th International Conference On Distributed Computing Systems*, 422–427, May 1986.

[3] A. J. Bernstein. Output Guards and Nondeterminism in 'Communicating Sequential Processes'. *ACM Transactions on Programming Language and Systems*, 2(2):234–238, April 1980.

[4] 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.

[5] M. Collado, R. Morales, and J. J. Moreno. A Modula-2 Implementation of CSP. *ACM SIGPLAN Notices*, 22(6):25–37, June 1987.

[6] E. W. Dijkstra. Guarded Command, Nondeterminism and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[7] N. Francez. *Fairness*. Springer-Verlag, New York, 1986.

[8] D. D. Gajski, D. H. Lawrie, D. J. Kuck, and A. H. Sameh. Cedar. *COMPCON-84, IEEE Computer Society Conference*, 306–309, February 1984.

[9] J. E. Grass and R. H. Campbell. Mediators: A Synchronization Mechanism. *The 6th International Conference On Distributed Computing Systems*, 468–477, May 1986.

[10] C. A. R. Hoare. *Communicating Sequential Processes. Computer Science*, Prentice Hall, 1985.

[11] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[12] *OCCAM Programming Manual.* Inmos Ltd., 1982.

[13] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[14] 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.

[15] R. B. Kieburtz and A. Silberschatz. Comments on 'Communicating Sequential Processes'. *ACM Transactions on Programming Language and Systems*, 1(2):218-225, Oct. 1979.

[16] J. Misra. Distributed-Discrete Event Simulation. *ACM Computing Surveys*, 18(1):39-65, March 1986.

[17] S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Language and Systems*, 4(3):455-495, July 1982.

[18] G. F. Pfister et al. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. *Proceedings of the 1985 International Conference On Parallel Processing*. 764-771, August 1985.

[19] G. F. Pfister and V. A. Norton. "Hot Spot" Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-34(10):943-948, October 1985.

[20] D. A. Reed and R. M. Fujimoto. *Multicomputer Networks: Message-Based Parallel Processing. Computer Science*, MIT Press, 1987.

[21] D. A. Reed, A. D. Malony, and B. D. McCredie. Parallel Discrete Event Simulation: A Shared Memory Approach. *Proceedings of the 1987 ACM SIGMETRICS Conference on Measuring and Modeling Computer Systems*, 15(1):36-38, May 1987.

[22] Z. Sun and X. Li. CSM: A Distributed Programming Language. *IEEE Transactions On Software Engineering*, SE-13(4):497-500, April 1987.

[23] B. Thomas et al. *Butterfly Parallel Processor Overview*. BBN Report No. 6148, BBN Laboratories Incorporated, March 1986.

[24] D. Zobel. Transformations For Communication Fairness In CSP. *Information Processing Letters*, 25:195-198, May 1987.

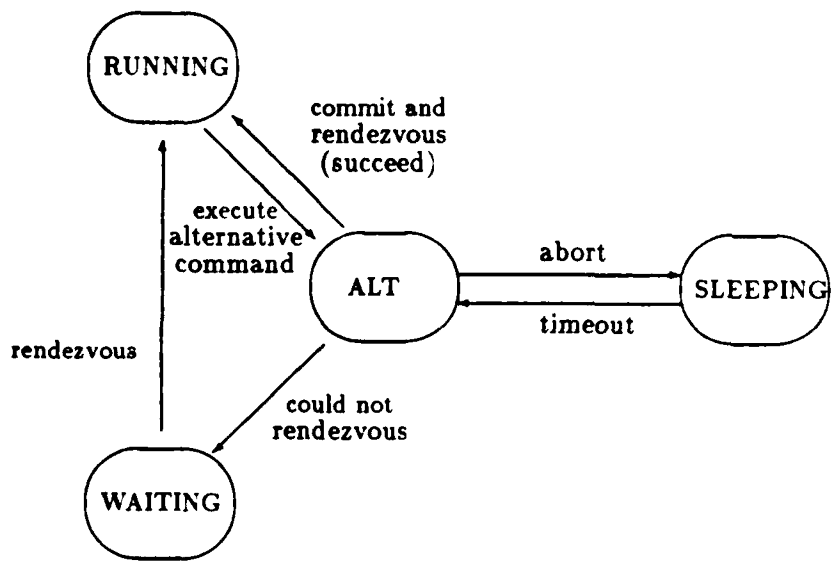


Figure 1: The State diagram of a process.

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

Figure 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.

```
/* 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);
        IF (ReturnValue = FAILED) THEN Sleep(TimeOut); END;
        END;
    RETURN (ReturnValue);
END Alternative;
```

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

```
PROCEDURE TryAlternative(g1, ..., gn): INTEGER;
VAR
    BOOLEAN flag;
    INTEGER GuardNumber; /* corresponding guard of Pr */
    INTEGER i, r;
BEGIN
    Statel := ALT;
    /* look for rendezvous with a waiting process.  */
    FOR i:=1 TO n DO
        r := CommunicantID(gi);
        flag := TRUE;
        WHILE (flag) DO
            CASE Stater DO /* The remote process state.  */
            RUNNING: flag := FALSE;
            SLEEPING: flag := FALSE; /* try next guard */
            WAITING: GuardNumber := CheckAndCommit(AltListr, gi);
                IF (GuardNumber = FAILED) THEN
                    flag := FALSE; /* try next guard */
                ELSE /* Wake up Pr */
                    Statel := RUNNING;
                    Signal(r, GuardNumber);
                    Communicate(gi);
                    RETURN (i);
                    END;
            ALT: IF (TransIDl < TransIDr) THEN
                    WHILE (Stater = ALT) DO END;
                ELSE /* busy wait loop.  */
                    Statel := SLEEPING;
                    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(); /* Blocks */
    Statel := RUNNING;
    Communicate(gi)
    RETURN (i);
END TryAlternative;
```

Figure 4: The *TryAlternative* procedure attempts to rendezvous with a process listed in an I/O guard, and does not return until rendezvous takes place.

END
FILMED
FEB. 1988
DTIC