SYNCHRONIZATION OF ASYNCHRONOUS PROCESSES IN CSP

Rajive Bagrodia

November 1988
CSD-880089

SYNCHRONIZATION OF ASYNCHRONOUS PROCESSES

IN CSP[1]

Rajive Bagrodia

April, 1987 (revised August, 1988)

Computer Science Department,
University of California at Los Angeles,
Los Angeles, CA 90024

Arpanet address: rajive@cs.ucla.edu

---

[1]A preliminary version of this paper titled "A Distributed Algorithm to Implement the Generalized Alternative Command of CSP' was presented at the 6th ICDCS held at Cambridge in May, 1986. This work was partially supported by a grant from the National Science Foundation under grant No. CCR-8810376

ABSTRACT

Many concurrent programming languages including CSP and Ada use synchronous message-passing to define communication between a pair of asynchronous processes. Suggested primitives like the generalized alternative command for CSP and the symmetric select statement for Ada allow a process to non-deterministically select one of several communication statements for execution. The communication statement may be either an input or an output command. We propose a simple algorithm to implement the generalized alternative command and show that it uses fewer messages than existing algorithms.

1. Introduction

Many concurrent programming languages [Hoare 78], [Gehani 83], [Andrews 81] use synchronous message-passing for communication between a pair of concurrent, asynchronous processes. In synchronous message-passing, the sender and receiver must both be ready to communicate for a communication between them to occur; we refer to the above form of communication as a **binary rendezvous**. The generalized alternative command [Kieburtz 79] allows a process to select one of several binary rendezvous for execution. A number of algorithms [Bernstein 80, Buckley 83, Van De Snepscheut 81, Schwarz 78, Schneider 82] have been designed to implement the above command for a group of concurrently executing processes. Buckley & Silberschatz [Buckley 83] present four criteria to determine the 'effectiveness' of algorithms that implement this construct. In the light of these criteria, they point out major drawbacks of many previously published algorithms and present an algorithm that meets the criteria. In section 2, we present an algorithm that satisfies the above criteria and is simpler and more efficient than the algorithm presented in [Buckley 83].

More recently, researchers have proposed the use of multiway rendezvous [Charlesworth 87, Forman 87, Francez 86, Milne 85] to model synchronous communication among an *arbitrary* number of asynchronous processes. Multiway rendezvous is a generalization of binary rendezvous. Existing solutions [Chandy 88, Bagrodia 87] for the multiway rendezvous, subdivide the problem into two parts, synchronization and exclusion, and implement multiway rendezvous by combining solutions to each of the subproblems. Although the above algorithms can be used to implement the special case of binary rendezvous, such solutions are more complex than algorithms designed specifically for binary rendezvous.

Section 2 presents a simplified version of the binary rendezvous problem. In the simplified description, we ignore the syntax and semantics of communication in CSP and restrict our attention to the issue of pairwise synchronization of asynchronous processes. In sections 3 and 4, we describe the algorithm in the simplified context. The correctness proof for the algorithm is presented in section 5. In section 6, we indicate how the algorithm presented in this paper is extended to implement the generalized alternative command of CSP.

2. Problem Description

We consider a group of concurrent, asynchronous processes. Each process in the system has a unique identifier, called its **process_id**. A synchronization between two processes in the system is referred to as an **interaction**. Each interaction is represented by a unique pair (p_i,p_j), where p_i and p_j are the process_ids of the two processes involved in the corresponding synchronization. We use the term process p_i to mean the process with process_id p_i. Each process in the system is either *idle* or *active*. We assume that processes do not terminate. An *idle* process is waiting to **commit** to any one of a set of interactions; this set is referred to as its **interaction_set**. An *idle* process p_i becomes *active*, only after it **commits** to some interaction (p_i,p_j). An *active* process becomes *idle* autonomously. The algorithm ensures that a process p_i **commits** to an interaction (p_i, p_j), only when it determines that process p_j will also do so. We assume that each interaction in the system is assigned a unique identifier, referred to as its **interaction_id**[2].

An interaction (p_i,p_j) is *enabled* if both processes p_i and p_j are *idle*. The interaction is said to be *disabled* if at least one of the two processes is *active*. We assume that when two processes commit to the same interaction, they are synchronized for a finite period of time. We define a relation *conflict* between a pair of interactions, where *conflict*(e_k,e_l) is *true* if and only if $k \neq l$ and interactions e_k and e_l contain a common process. It is required to construct a distributed algorithm that satisfies the following two properties:

1. **Safety** property: Processes do not (simultaneously) commit to interactions that conflict with each other.
2. **Liveness** property:
    - a. If process p_i **commits** to interaction (p_i, p_j), process p_j will eventually do so.
    - b. Every *enabled* interaction is eventually *disabled*.

The next section gives an informal, intuitive description of the algorithm. The properties of the algorithm are formally defined and proven in section 5.

[2] Unique identifiers for interactions are not strictly required in this algorithm. The pair of process_ids can be used to uniquely identify each interaction. The assumption of unique interaction_ids is being made only for simplicity in exposition.

3. Algorithm : Informal Description

The algorithm associates a unique token with each interaction. The token contains the process_ids of the two processes involved in the interaction. When a process p_i becomes *idle*, it determines if any interaction (p_i, p_j) can be executed by requesting that interaction. An *idle* process requests interactions from its interaction_set in increasing order of priority. The unique interaction_id of each interaction determines its priority, where a higher interaction_id implies a higher priority. A process p_i may request only those interactions, for which it possesses the corresponding token. When p_i requests an interaction (p_i,p_j), it sends the corresponding token to p_j. A process may request at most one interaction at any time.

On receiving a token, a process p_j may either **commit** to the corresponding interaction, **refuse** to do so or **delay** its response. If p_j is *idle* and has not itself requested another interaction, it **commits** to this interaction. A process **commits** to an interaction by sending the token back to the requesting process. On the other hand, if p_j is *active*, it **refuses** the interaction. A process **refuses** an interaction by capturing the token and sending a **cancel** message to the requesting process, in this case p_i. This implies that the process that last **refused** an interaction has the responsibility to initiate the next **request** for the interaction. Henceforth, a process that has requested an interaction, but has not received a response to its request, is referred to as a *grey* process.

Due to the asynchronous nature of the requests for interactions, conflicts can arise when a *grey* process receives the token (request) for another interaction before receiving a response to its request. These conflicts can be easily resolved by using the unique interaction_ids. Specifically, if a *grey* process p_i receives a token with interaction_id e_k, it **delays** the token if e_k is less than the interaction_id of the token requested by p_i; otherwise, p_i **refuses** the interaction and sends a **cancel** message as described above. This guarantees that the requests in the system will never be delayed in a manner that can cause the system to deadlock. If a *grey* process, say p_i delays an interaction, the algorithm can guarantee that p_i will either commit to its requested interaction or to the delayed interaction. Thus, it is only necessary for a process to delay at most one interaction. Tokens received by a *grey* process that has delayed an interaction, can immediately be refused by the process independently of the relative priority of the two interactions. A *grey* process responds to the delayed interaction, after it receives a reply to its request. Consider an interaction (p_i, p_j) with interaction_id e_k. Process p_i requests interaction e_k and sends the corresponding token to p_j. Subsequently, p_i receives some token and delays it in accordance with the rule for delaying interactions described above. In reply to its request, eventually p_i receives either the token for interaction e_k(i.e. p_j has committed to interaction e_k) or a *cancel* message (i.e. interaction e_k is refused by p_j). In the first case, p_i commits to interaction e_k and refuses the delayed interaction. In the second case, p_i relinquishes interaction e_k and commits to the delayed interaction.

The key idea of the algorithm is that an *idle* process continues to request interactions in increasing order of priority, until it either commits to an interaction or runs out of tokens. If any process delays an interaction, it is guaranteed to eventually become *active*. If an *idle* process runs out of tokens, it will commit to the first token it receives from another process. If it does not receive any tokens, it follows that all interactions in its interaction_set must be *disabled*. This property is proven subsequently. We first give a precise description of the algorithm in the next section.

4. The Algorithm

A process autonomously makes the transition from *active* to *idle*. On becoming *idle*, it negotiates with processes named in its interaction_set to **commit** to an interaction. We assume that when two processes commit to an interaction, they remain synchronized until the process that requested the interaction sends a signal to the other process to terminate the synchronization. (This assumption is not necessary for the algorithm. In general, the synchronization may be terminated by either of the two processes.) Each process p_i in the system has the following local variables:

token_q$_i$: Ordered queue to store the tokens owned by process p_i; the queue is ordered on the interaction_ids of the tokens. Standard queue operations, namely *enqueue*, *dequeue*, and *empty* are assumed to be defined for this ordered queue.

color$_i$: used to describe the state of the process, which may be any one of the following:

- *white*: the process is *idle* and does not have any outstanding request.
- *grey*: the process has requested an interaction, but not received a reply to its request;
- *black*: the process has committed to an interaction.
- *yellow*: the process is *active*.

rno$_i$: if **color**$_i$=*grey*, it contains the id of the interaction requested by the process; otherwise it is set to 0.

delay$_i$: set to *true if and only if* p_i has delayed a token.

delay_token$_i$: if **delay**$_i$ is *true*, then **delay_token**$_i$ contains the token delayed by p_i.

com$_i$: **vector of booleans**: **com**$_i[e_k]$ is *true* if and only if p_i commits to interaction e_k.

sync$_i$: **vector of booleans**: **sync**$_i[e_k]$ is *true* if e_k was requested by p_i and both processes have committed to e_k.

The algorithm uses the following types of messages:

token: RECORD
ino: unique id of this interaction. (Every *ino* is greater than 0)
process_list: process_ids of the two processes named in the interaction;

END_RECORD;

cancel: sent by a process to **refuse** an interaction.

done: sent by a process to terminate an interaction.

Initially, the variable **color** for each process is set to *yellow*; variable **delay** and vectors **com** and **sync** are set to *false*; **rno** is initialized to 0. The token for each interaction is arbitrarily assigned to one of the two processes named in the interaction. Each process executes the following rules which constitute a single guarded command [Dijkstra 75]. The subscripts on variables have been dropped for simplicity.

```
R1: On transition to idle state:

      (color=yellow) ∧ (process is idle)
                                 ===>   color:=white;

R2: Requesting an interaction:

      (color=white ∧ ¬empty(token_q))
                                 ===>  [ REQUEST_TOKEN;
                                         color:=grey;
                                       ]

R3: Committing to  an interaction:

   R3.1 On receiving a token:
   (color=white) ∨ ((color=grey) ∧ (token.ino=rno))
                                 ===> [ COMMIT_TOKEN(token);
                                        color:=black;
                                      ]

   R3.2 On receiving a cancel message:
   (color=grey) ∧ delay
                                 ===>  [ delay:=false;
                                         COMMIT_TOKEN(delay_token);
                                         color:=black;
                                       ]

R4: Refusing an Interaction:

   On receiving a token:
   (color=yellow) ∨ (color=black) ∨ delay ∨ ((color=grey) ∧ (token.ino>rno))
                                 ===> REFUSE_TOKEN(token);

R5: Delaying an interaction:

   On receiving a token:
   (color=grey) ∧ (token.ino<rno) ∧ ¬delay
                                 ===>  [ delay:=true;
                                            delay_token:=token;
                                       ]
```

```
R6: Relinquishing an Interaction:

    On receiving a cancel message:
    (color=grey) ∧ ¬delay
                                ===> [ rno:=0;
                                       color:=white;
                                     ]

R7: Transition to active state:

    7.1 Terminate Interaction:
    (∃e_k, sync[e_k])        ===> [ sync[e_k]:=false;
                                    send done message to other process;
                                    Set delay and com to false;
                                    color:=yellow;
                                  ]

    7.2 On receiving a done message:
    (color=black)            ===> [ Set delay and com to false;
                                    color:=yellow;
                                  ]

Request_Token

    token:=dequeue(token_q);
    rno:=token.ino;
    send token to other process in token.process_list;

Refuse_Token(token)

    send cancel message to requesting process;
    enqueue(token_q, token);

Commit_Token(token)

    com[token.ino]:=true;
    If delay then Refuse_Token(delay_token);
    [ token.ino <> rno
        ==>Send token to requesting process;
    [] token.ino = rno
        ==>sync[token.ino]:=true;
           enqueue(token_q, token);
    ]
```

5. Correctness Proof

In this section, we prove that our algorithm satisfies the properties listed below.

- $\neg(\mathbf{com}_i[e_k] \wedge \mathbf{com}_i[e_l] \wedge (l \neq k))$ I1
- $\neg(\mathbf{sync}_i[e_k] \wedge \mathbf{sync}_j[e_l] \wedge conflict(e_k, e_l))$ I2
- $enable(e_k) \mapsto disable(e_k)$. P1

The symbol $\mapsto$ stands for 'leads-to' [Chandy 88], and means that if the left-hand side of the relation holds, the right-hand side holds or will eventually hold. Invariants I1 and I2 represent the safety properties for the algorithm, where I1 ensures that a process commits to at most one interaction, and I2 guarantees that processes do not commit to interactions that conflict with one another. Property P1 ensures that if an interaction e_k is *enabled* (both processes named in the interaction are *idle*), then eventually e_k is *disabled* (at least one of p_i or p_j becomes *active*).

Safety

Theorem 1: The following is an invariant:

$$\neg(com_i[e_k] \wedge com_i[e_l] \wedge (l \neq k)) \qquad \text{I1}$$

Proof: Without loss of generality, assume $\mathbf{com}_i[e_k]$. From the algorithm, for any e, $\mathbf{com}_i[e]$ is set to *true* only due to execution of R3 which also sets $\mathbf{color}_i = black$. Subsequently, $\mathbf{color}_i = black$ leads-to $\mathbf{color}_i \neq black$, only due to R7, which also sets array $\mathbf{com}_i$ to *false*. As the preconditions to R3 exclude $\mathbf{color}_i = black$, we have $\mathbf{com}_i[e_k] \Rightarrow \forall l \neq k, \neg \mathbf{com}_i[e_l]$ □

Lemma S1: Given interaction (p_i, p_j) with interaction_id e_k, $\mathbf{sync}_i[e_k] \Rightarrow \mathbf{com}_i[e_k] \wedge \mathbf{com}_j[e_k]$.

Proof: Given $\mathbf{sync}_i[e_k]$. From the text of Commit_Token, $\mathbf{sync}_i[e_k] \Rightarrow \mathbf{com}_i[e_k]$. Also due to the text of Commit_token, $\mathbf{sync}_i[e_k]$ is set to *true*, only when a process p_i receives a token e_k that it had requested. From R2 and R3, a process sends a token to another either to request an interaction or commit to an interaction requested by another process. Since interaction e_k was requested by p_i, the token must have been sent by p_j to p_i only after p_j committed to e_k. Due to R3, when p_j committed to e_k, $\mathbf{com}_j[e_k]$ must have been set to *true*. □

We use invariants I1, and lemma S1 to show that the algorithm maintains invariant I2.

Theorem 2: The following is an invariant:

$$\neg(\mathbf{sync}_i[e_k] \wedge \mathbf{sync}_j[e_l] \wedge conflict(e_k, e_l))$$

Proof: As $conflict(e_k, e_l)$, interactions e_k and e_l must have a process in common; let the process be p_c. Assume that interactions e_k and e_l are defined by the pairs (p_i, p_c) and (p_j, p_c) respectively. We present a proof by contradiction. Assume $(\mathbf{sync}_i[e_k] \wedge \mathbf{sync}_j[e_l])$.

If $\mathbf{sync}_i[e_k]$, due to lemma S1, $\mathbf{com}_i[e_k] \wedge \mathbf{com}_c[e_k]$. Similarly, if $\mathbf{sync}_j[e_l]$, due to lemma S1, $\mathbf{com}_j[e_l] \wedge \mathbf{com}_c[e_l]$. However, $\mathbf{com}_c[e_k] \wedge \mathbf{com}_c[e_l]$ violates invariant I1. □.

Liveness

Lemma L0: For some e_k, if $\mathbf{com}_i[e_k]$, then eventually p_i becomes *active*.

Due to R3, if $\mathbf{com}_i[e_k]$, then $\mathbf{color}_i = black$. Due to R7, $\mathbf{color}_i = black$ implies that eventually $\mathbf{color}_i = yellow$, which by definition implies that p_i is *active*.

Lemma L1: The interaction delay graph is a finite, acyclic graph.

The interaction delay graph is a finite, directed graph G, each of whose vertices represent an interaction. A directed edge from vertex e_j to vertex e_k exists in G if and only if interaction e_j was delayed by a process that had requested interaction e_k. Since a process may request or delay at most one interaction, each vertex in G may have at most one incoming or outgoing edge.

Initially this graph is empty and hence trivially acyclic. Subsequently, a cycle in this graph can be formed only due to the addition of an edge. In the algorithm, a process p_i can delay an interaction e_k only if p_i has requested an interaction e_j, such that $e_k < e_j$. Therefore, all edges in the graph go from a vertex with a lower id to one with a higher id. It follows that G must be acyclic. □

Lemma L2: If a process p_k delays an interaction, eventually all interactions in the interaction_set of p_k are *disabled*.

<u>Proof</u>: Consider a process p_k that has requested an interaction (p_k, p_j). For simplicity in notation, let this interaction have an interaction_id e_k. Assume $\mathbf{delay}_k$, i.e. p_k delays some interaction e_l in accordance with rule R5. Thus, $\mathbf{delay\_token}_k = e_l$. An arc exists from e_l to e_k in G. If p_k becomes *active*, by definition, all interactions in its interaction_set are *disabled*. Due to L0, if for some e, $\mathbf{com}_k[e]$, then eventually p_k is *active*. We prove that given $delay_k$, eventually $\mathbf{com}_k[e_k] \vee \mathbf{com}_k[e_l]$.

From lemma L1, G is acyclic. Hence, there exists at least one vertex in G, called the *sink* vertex, that has an incoming arc and no outgoing arc. Let the *sink* vertex be represented by e_s. Either e_k is a *sink* vertex or a *sink* vertex is reachable from e_k. In either case, let the incoming arc to e_s be from vertex e_r. This implies that some process p_s has requested interaction e_s and delayed interaction e_r. Since vertex e_s has no outgoing arcs, interaction e_s has not been delayed. Only two possibilities remain: either p_s commits to interaction e_s ($\mathbf{com}_s[e_s]$), or e_s is refused. In the second case, due to R3.2, $\mathbf{com}_s[e_r]$. Due to lemma L0, ($\mathbf{com}_s[e_r] \vee \mathbf{com}_s[e_s]$) implies that eventually p_s is *active*, and by definition e_r is *disabled*.

When interaction e_r is *disabled*, the corresponding arc (e_r, e_s) is deleted from G. Since each vertex in G can have at most one outgoing arc, deleting arc (e_r, e_s) from the graph will result in vertex e_r becoming a sink vertex (no outgoing arcs). Since G is finite and e_s is reachable from e_k, we can inductively conclude that eventually e_k will become a *sink* vertex. By using the reasoning applied above for p_s, it follows that eventually ($\mathbf{com}_k[e_k] \vee \mathbf{com}_k[e_l]$). □

We use lemmas L1 and L2 to prove property P1.

Theorem 3: Liveness: An *enabled* interaction is eventually *disabled.*

Proof: Consider interaction (p_i, p_j) with interaction_id e_k. Due to L0, the interaction is *disabled* if ($\mathbf{com}_i[e_k] \vee (\mathbf{com}_j[e_k]$). Without loss of generality, assume that the token for interaction e_k is owned by p_i. Due to R2, if p_i remains *idle*, it must eventually request interaction e_k and send the corresponding token to p_j. On receiving the token, if p_j commits to or delays the interaction, then e_k will be *disabled* due to R3 or L2. Assume p_j refuses the interaction and captures the token.

If p_j remains *idle*, due to R2, it must eventually request interaction e_k and send the token to p_i. Once again, if p_i commits to or delays the interaction, the interaction is *disabled.* Assume p_i refuses the interaction. Due to R4, it must be that

$$(\mathbf{color}_i = \mathit{yellow}) \vee (\mathbf{color}_i = \mathit{black}) \vee \mathit{delay}_i \vee ((\mathbf{color}_i = \mathit{grey}) \wedge (e_k > \mathbf{rno}_i))$$

If $\mathbf{color}_i = \mathit{black}$, due to R7, eventually $\mathbf{color}_i = \mathit{yellow}$; If $\mathbf{color}_i = \mathit{yellow}$, interaction e_k is *disabled* by definition; if delay_i, e_k will eventually be *disabled* due to L2. Thus, it must be that $((\mathbf{color}_i = \mathit{grey}) \wedge (e_k > \mathbf{rno}_i))$. However, a process requests interactions in increasing order of priority and e_k has already been requested by p_i. Thus, if p_i is *grey* and has currently requested some interaction e_l, $\mathbf{rno}_i = e_l$ and it is impossible for e_k to be greater than $\mathbf{rno}_i$. □

Corollary 1: An interaction generates at most four messages before it is *disabled.*

In the above theorem, when interaction e_k is requested by p_i, if the interaction is disabled, exactly two messages are generated -- a token message from p_i to p_j and either a token or a **cancel** message from p_j to p_i. However, if the interaction is refused by p_j but not disabled, then eventually p_j must request the interaction. From the above theorem, the interaction is bound to be disabled and at most two more messages may be generated -- a token message from p_j to p_i and either a token or a **cancel** message from p_i to p_j. □

5.1. Algorithm Efficiency

Buckley and Silberschatz [Buckley 83] present four useful criteria to measure the efficiency of algorithms that implement the generalized alternative command. We compare the algorithm presented in this paper with respect to these criteria.

The first criterion states that the *number of processes* involved in determining whether any given interaction can be executed should be as small as possible. This criterion ensures that processes do not access global information. In the algorithm presented in this paper, for each interaction (p_i, p_j), the only processes involved in making this decision are p_i and p_j. Second, the *amount of system information* that each process needs in order to make a decision to commit to an interaction should be small. In our algorithm, each process needs to know only whether the other process named in an interaction, is *idle*. This information is communicated by the exchange of messages (**token** or **cancel**) between the two processes, and no global state information need be saved by a process. The amount of local state information is also small, and apart from the tokens, consists of exactly three variables, **color**, **rno**, and **com**. As proved in invariant I1, variable **com** need not be an array, as each process commits to at most one event. Variable **sync** is an auxiliary variable that was used to simplify the correctness proofs and is not needed for the operation of the algorithm itself. The third and fourth criteria impose a bound on the *time* and the *number of messages* needed to ensure that two processes named in an interaction do not stay *idle* indefinitely. The corollary to the liveness theorem establishes the necessary bounds by showing that each interaction can generate only a finite number of messages before it is disabled. We now compute the maximum number of messages needed to establish synchronization between two *idle* processes named in an interaction.

Consider two processes p_i and p_j that are named in an interaction e_k. Let T be the number of interactions in the interaction_set of each process. Further, let t be the number of interactions e_j in the interaction_set of each process, such that $e_j \leq e_k$. Assume that eventually, both p_i and p_j commit to interaction e_k. We compute the number of messages required, in the worst-case, for interaction e_k to be *disabled*. Without loss of generality, assume p_i originally owns the token for e_k. Process p_i can have at most T tokens. At worst, p_i successively requests the T interactions, all of which (including e_k) are refused by the other process(es). This generates at most 2*T messages. Subsequently, p_i does not receive any tokens, other than e_k (if it does, it would have to commit to the corresponding interaction). Process p_j requests tokens in increasing order of priority. This implies that p_j can request at most t-1 interactions before requesting e_k. The t-1 interactions must all be refused (since p_j eventually commits to e_k) generating exactly 2*(t-1) messages. It follows that in order for interaction e_k to be executed exactly 2*(T+t) messages are generated in the worst case. The algorithm described in [Buckley 83] requires at most 6*M + 2*Q messages for an interaction to be executed; where M is the number of interactions in the interaction_set of a process, say p_j; and Q is the number of processes in the interaction_set of p_j with process_ids that are smaller than p_j.

6. Discussion

In this section, we informally indicate how the algorithm is used to implement the generalized alternative command of CSP.

A CSP program consists of a collection of sequential processes that execute concurrently. The concurrent processes communicate with each other via synchronized message-passing. Non-deterministic process behavior is provided by means of the generalized alternative command. This command, as described in [Buckley 83], allows a process to arbitrarily select one of several statements for execution. Each statement in the command is protected by a **guard** (a boolean expression and/or a communication statement) which must be enabled for a statement to be considered for selection. A **guard** is enabled if the boolean expression evaluates to true and the process named in the communication statement has not terminated. The communication statement may be either an input or an output statement. In CSP, every input (output) statement must explicitly name the source (destination) process. Communication between two processes can occur when the output statement in a process *matches* the input statement of another process. A pair of communication statements are said to be *matched* [Hoare 78] if (1) an input command in one process specifies as its source the process name of the other process; (2) an output command in the other process specifies as its destination the process name of the first process; and (3) the target variable of the input command has the same type as the expression of the output command.

In the context of CSP, an interaction (p_i, p_j) represents a pair of *matched* communication statements in processes p_i and p_j respectively. The interaction_set of a process refers to the set of *matched* communication statements in an alternative command of the process. The term 'process p_i **commits** to interaction (p_i,p_j)' implies that p_i executes the communication statement that *matches* a corresponding communication statement in process p_j. The interaction is terminated when the processes have exchanged the appropriate message. We now indicate how the above algorithm is extended in minor ways to implement process communications in CSP.

In a CSP program, two processes p_i and p_j may contain more than one pair of matching communication statements, thus violating our assumption that a pair of process-ids uniquely identifies an interaction. This problem may be easily handled by associating a unique identifier with every pair of matched communication statements in the program. As CSP does not allow dynamic process creation, this association may be done statically (at compile time).

Second, a process in a CSP program may contain many alternative commands, each command comprising of different communication statements. In addition, each communication statement may be preceded by a boolean expression; the communication statement is executable only if the boolean expression evaluates to *true*. The two characteristics mentioned above imply that in general, the interaction_set of a process is dynamic, rather than static as assumed in this paper. To handle dynamic interaction_sets, for every p_i, we define a boolean array called $status_i$ which contains a single entry for each matched communication statement in process p_i. When p_i encounters an alternative command, $status_i$ is initialized to *false* (in the algorithm, *status* will be initialized to *false*, when rule R1 is executed). Each communication statement in the guards of the alternative command corresponds to a unique interaction; consider the communication statement corresponding to interaction e_k. If the communication statement is not preceded by a boolean expression, or if the boolean expression evaluates to *true*, $status_i(e_k)$ is set to *true*. The condition for refusing an interaction (rule R4) is modified such that on receiving a token, the process will also refuse the interaction e_k, if $\neg status(e_k)$. The rule for requesting tokens (rule R1) is also modified to prevent a process from requesting an interaction e_k, if $\neg status(e_k)$.

Finally, it may be mentioned that the algorithm described in this paper is not fair: we do not assert that if a particular interaction, say (p_i, p_j) is *enabled* infinitely often, processes p_i and p_j will commit to the interaction infinitely often. In fact, as presented in this paper, the algorithm gives a higher priority to interactions with a smaller id. We indicate how an implementation may prevent this bias. For simplicity in the presentation, we have assumed in this paper, that interactions are always requested by a process in increasing order of an id number associated with the interactions. In fact, as shown in [Bagrodia 86], correctness of the algorithm does not require the interactions to have unique ids, nor to be requested in a particular order of priority. Unique process-ids are sufficient to allow processes to resolve conflicts. Although the modification cannot guarantee fairness, it will decrease the probability of an interaction with a higher id being always ignored in favor of an interaction with a lower one.

Acknowledgements

I am grateful to Professor Shmuel Katz and Dr. Ira Forman for initial discussions on the problem and the algorithm. I am indebted to Professor Chandy and Professor Snepscheut for their detailed comments which resulted in significant improvements in the presentation of the algorithm and its correctness proof. Thanks are also due to the referees for their helpful comments.

References

[Andrews 81] Andrews, G.R.
Synchronizing Resources.
ACM TOPLAS 3(4):405-430, October, 1981.

[Bagrodia 86] Bagrodia, R.
A Distributed Algorithm To Implement The Generalized Alternative Command of CSP.
In *Proccedings of 6th International Conference on Distributed Systems.* Cambridge, May, 1986.

[Bagrodia 87] Bagrodia,R.L.
A Distributed Algorithm to Implement N-Party Rendezvous.
In K.V.Nori (editor), *Foundations of Software Technology and Theoretical Computer Science, LNCS 287,* pages 138-152. December, 1987.

[Bernstein 80] Bernstein, A.J.
Output guards And Non-determinism in Communicating Sequential Processes.
ACM TOPLAS 2(2):234-238, April, 1980.

[Buckley 83] Buckley, G. and Silberschatz, A.
An Effective Implementation Of The Generalized Input-Output Construct of CSP.
ACM TOPLAS 5(2):223-235, April, 1983.

[Chandy 88] Chandy, K.M. and Misra, J.
A Foundation of Parallel Program Design.
Addison-Wesley, Reading, Massachusetts, 1988.

[Charlesworth 87] A. Charlesworth.
The Multiway Rendezvous.
ACM Trans. on Programming Languages and Systems 9(3):350-366, July, 1987.

[Dijkstra 75] Dijkstra, E.W.
Guarded commands, nondeterminacy, and formal derivation of programs.
Communications of the ACM 18(8):453-457, August, 1975.

[Forman 87] Forman, I.R.
On the Design of Large Distributed Systems.
Technical Report No. STP-098-86, Microelectronics and Computer Technology Corp, Austin, Texas, January, 1987.
Preliminary version in Proc. First Int'l Conf. on Computer Languages, Miami, Florida, October 25-27, 1986.

[Francez 86] Francez, N., Hailpern, B. and Taubenfeld, G.
Script:A Communication Abstraction Mechanism.
Science of Computer Programming 6(1), January, 1986.

[Gehani 83] Gehani N.
Ada: An Advanced Introduction.
Prentice-Hall, 1983.

[Hoare 78] Hoare, C.A.R.
Communicating Sequential Processes.
CACM 21(8):666-677, August, 1978.

[Kieburtz 79] Kieburtz, R.B., and Silberschatz, A.
Comments on 'Communicating Sequential Processes.
ACM Trans. Program. Lang. Syst. 1(2):218-225, October, 1979.

[Milne 85] Milne, George.
CIRCAL and the Representation Of Communication, Concurrency and Time.
ACM TOPLAS 7(2), April, 1985.

[Schneider 82] Schneider,F.
Synchronization In Distributed Programs.
ACM TOPLAS 4(2):125-148, April, 1982.

[Schwarz 78] Schwarz, J.S.
Distributed synchronization of Communicating Sequential Processes.
Technical Report, Dept. Of Artificial Intelligence, University of Edinburgh, Scotland, July, 1978.

[Van De Snepscheut 81]
Van De Snepscheut, J.L.A.
Synchronous Communication Between Asynchronous Components.
IPL 13(3):127-130, December, 1981.