# Efficient Implementation of Multi-process Handshaking on Broadcasting Networks

M. Aspnäs [*] R.J.R. Back [*] R. Kurki-Suonio [†]

March 29, 1989

**Abstract**

Multi-process handshaking is a generalization of ordinary handshake communication between two processes. It allows an arbitrary number of processes to be synchronized in a common handshake, for the purpose of carrying out a joint action, consisting of a sequential statement that is executed in the combined state space of the processes participating in the handshake. The statement updates the local variables of the processes in a manner that depends on the values of the local variables of the other processes. The paper is concerned with implementing this communication mechanism on networks with a broadcasting facility. The implementation is described and its correctness is proved. The efficiency is calculated analytically and verified experimentally by simulation studies.

## 1 Introduction

Languages like CSP [Hoa], Occam [Inm] and Ada [DoD] use a *handshaking* (rendezvous) mechanism for communication. A process that wants to communicate with another process must wait until the other process is also ready for communication, before the actual message transmission can take place. Communication is thus carried out in two phases: first a *synchronization* phase (the handshake) and then a *message transmission* phase. A process may be ready for more than one communication, but it may only be engaged in one communication at a time. In CSP this is achieved by allowing input statements in guards, in Ada by a similar mechanism for accept statements. In both languages the mechanism is designed so that the decision whether a specific communication is to take place or not can be done locally by the process which is executing the alternative communication command.

The communication mechanisms of CSP/Occam and Ada thus have the following characteristics:

1. Communication involves exactly two processes.
2. Communication is asymmetric: the processes engage in the communication in different roles, one as the sender (or caller), the other as the receiver (or callee).

[*]Åbo Akademi, Department of Computer Science, Lemminkäinengatan 14, SF-20520 Turku, Finland
[†]Tampere University of Technology, Software Systems Laboratory, Box 527, SF-33101 Tampere, Finland

3. Choosing among different possible communications is restricted to one of the parties, the receiver (or callee).

## 1.1 Multi-process handshaking and action systems

Restriction 3 can be lifted in CSP by allowing output statements in guards, and in Ada by allowing alternative entry calls. This, however, introducs the problem of distributed agreement: the processes have to consult each other in order to reach agreement about which of the possible communications should be realized. If no agreement is sought, i.e., each process is allowed to decide for itself which communication it will participate in, then deadlock can occur.

Consider next the restriction 2, i.e., that communication between processes is asymmetric, with one process sending a message and the other receiving it (or one process calling an entry in the other process). We make the positions of the participants symmetrical with respect to communication by postulating that the two processes execute a joint statement after the synchronizing handshake, rather than just exchanging a message. The joint statement is a sequential statement that is executed in the combined state space of the processes involved in the handshake. The statement may thus refer to the local variables of both processes involved and may result in local variables of both processes being updated, in a manner that depends on the values of local variables in the other process.

Joint statements generalize both CSP and Ada. In the case of CSP, the joint statement executed after the synchronizing handshake is always an assignment "$x := e$", where $x$ is the input variable of the receiving process and $e$ is the output value of the sending process. In the case of Ada, the joint statement executed after the synchronizing handshake is of the form $z{:=}u;\ S;\ y{:=}v$, where $u$ is the list of values of the input parameters in the entry call, $z$ is the list of formal input parameters, $S$ is the statement executed at the entry, $y$ is the list of actual output parameters and $v$ is the list of formal output parameters. Here $u$ and $y$ only refer to local variables of the calling process, while $z$, $S$, and $v$ only refer to local variables of the called process. We assume that the caller and the callee do not share any common variables.

The joint statement approach has the additional advantage that it allows us to generalize handshaking to involve an arbitrary number of processes. Thus restriction 1 above can be lifted. This gives us the final generalization of the communication mechanism of CSP and Ada, which we refer to as *actions*. If an action is executed by two or more processes, then we call it a *joint action*. If there is only one process involved, then it is called a *private action*. Note that the communication mechanisms of CSP and Ada cannot be readily extended to more than two processes, because these mechanisms are asymmetric.

The private actions correspond to the local computation that processes do between communications in CSP and Ada. Hence, by using private actions, we can dispense with the bodies of the processes altogether: everyting can be seen as an action. This is the basis for the action system approach to describing parallel and distributed systems.

An *action system* consists of a set of processes $P$ and a set of actions $A$. A process in an action system is of the form

**process** $p$: **var** $y_p$; $S'_p$;

where $p$ is the name of the process, $y_p$ stands for the *local variables* of process $p$, and

$S'_p$ assigns initial values to the variables of $p$. Together the local variables in an action system constitute the *global state* $y$ of the system.

Actions in an action system are of the form

```
**action** $a$ **by** $processes_a$ : $G_a \rightarrow S_a$
```

where $a$ is the name of the action, $processes_a$ is the set of proceses participating in the action $a$, $G_a$ is the *guard* of the action and $S_a$ is the *action body.* If the guard of an action is satisfied, the action is said to be *enabled.* The action $a$ is to be jointly executed by the processes in $processes_a$, provided that it is enabled. Executing the action changes the global state of the action system in the way described in the action body. The guard and the body of an action may only refer to local variables of the processes participating in the action, i.e., to the set of variables $y_a = \cup\{y_p \mid p \in processes_a\}$.

We assume that each guard is a conjunction of *local guards* of the processes, where a local guard only refers to local variables of the process. This means that an action is enabled if each process involved in the action evaluates its own local guard for the action to true. (It is actually sufficient to require that guards in actions are *separable*, i.e., that a guard is a boolean statement where each atomic predicate referes to local variables of only one process).

The joint actions are implemented by a *multi-process handshaking mechanism*, which works as follows. At any time, a process is either waiting to participate in some action or is actually executing some action. A process may only participate in one action at a time. After finishing execution of an action, the process determines which actions it is willing to participate in next by evaluating the local guards for all the actions that it knows of. An action is *enabled* if all processes that are involved in it are willing to participate in it. Some enabled action is chosen for execution. The local variables of the processes involved in this action will then be updated in a manner specified by the action. A process *releases* the handshake after its local variables have been updated, and is then free to participate in a new action. There is no synchronization at the end of executing a joint action: each process will finish its share of the joint action independently of the other processes involved in the action. Any number of actions may be executing simultaneously, subject to the restriction that a process only participates in one action at a time.

## 1.2 An example

As an example of an action system we consider an extended version of the dining philosophers problem where starvation among philosophers is not possible.

The system consists of $n$ philosopher processes and $n$ fork processes. Each fork process has a variable *token* which is set to true when the fork holds the token, and a variable *reserved* which is set to true when a philosopher has made a reservation for this fork. The eating action passes the token from the left to the right fork, if there is a token at the left fork. A philosopher can make a reservation for his left fork by a special reserve action. When the token eventually arrives at this reserved fork, the left neighbor is forced to give the right neighbor a chance to eat, provided that the latter has not eaten since he reserved the left fork. Initially, the first fork has the token and no fork is reserved.

The action system is as follows:

```
process Philosopher[i:1..n] :
    var hungry: boolean;
    hungry := false;
process Fork[i:1..n] :
    var token, reserved: boolean;
    reserved := false;  token := (i=1);
action Thinking[i:1..n]  by Philosopher[i] :
$\neg$ Philosopher[i].hungry $\rightarrow$
    Think;
    Philosopher[i].hungry := true;
action Reserve[ i:1..n]  by Philosopher[i], Fork[i] :
Philosopher[i].hungry $\wedge \neg$ Fork[i].reserved $\rightarrow$
    Fork[i].reserved := true;
action Eating[i:1..n]  by Philosopher[i], Fork[i], Fork[i+1] :
Philosopher[i].hungry $\wedge \neg$ (Fork[i+1].token $\wedge$ Fork[i+1].reserved) $\rightarrow$
    Eat;
    Philosopher[i].hungry := false;
    if Fork[i].token
    then Fork[i].token, Fork[i+1].token := false, true;
    Fork[i].reserved := false;
```

A proof for starvation freedom of this algorithm is presented in [BaKu88].

## 1.3 Previous and related work

Different implementations of multi-process handshaking have been proposed in [BaKu84b], [BaHaKu85], [BaKu88], [Ekl], [Bag87] and [Ram]. The implementations in [Ekl], [Bag87] and [Ram] all assume a fully connected point-to-point communication network, wheras [BaKu84b], [BaHaKu85] assume a broadcasting network. The problems involved in implementing this mechanism are similar to the problem of implementing CSP with output guards, which has been studied in [BuSi83], [Sis], [BaEkKu], [Bor] and [Bag86].

The work in this paper is based on the broadcasting net implementations proposed in [BaKu84b] and [BaHaKu85]. In [BaKu84b] and [BaKu88] a process is assumed to be able to cancel an attempt to broadcast a select message if it simultaneously receives a conflicting selection message. This assumtion is quite strong, and [BaHaKu85] therefore derived two implementations of multi-process handshaking on broadcasting networks that do not depend on this assumption. These implementations work for many of the common broadcasting networks, like CSMA/CD and token ring networks. The implementation of multi-process handshaking presented here is close to the first of the two implementations presented in [BaHaKu85], which is based on a token passing scheme. It improves on the earlier implementation by putting an upper bound on the number of token passing messages per executed handshake and by changing the way in which systems are initiated and terminated. We also include here simulation results from an experimental implementation of this handshake mechanism, based on [Asp87]. The purpose of this implementation was to experimentally measure the efficiency of the multi-process handshaking mechanism and compare this to analytic worst case measures.

The action system approach to describing parallel and distributed programs was introduced in [BaKu83] and has been further developed in [BaKu84a], [BaKu84b] and [BaKu88]. A similar approach has also been proposed by Chandy and Misra [ChMi88]. The implementation presented here would also work for their UNITY programs, as they can be seen as a special case of action systems where the execution is assumed to be weakly fair with respect to the selection of enabled actions.

## 2 A model for broadcasting communication

A *broadcasting network* consists of a set of concurrent processes $P = \{P_1, \ldots, P_n\}$, together with a broadcast channel $B$ which is shared by the processes. The broadcast channel is modeled as a shared data structure as follows. $B$ is a tuple $B = (M, next)$, where $M$ is a sequence of messages and $next$ is a function $next : P \to N$ ($N$ the set of natural numbers). $M$ records the sequence of messages that have been broadcast on the channel. For each process $p$ in $P$, $next(p)$ indicates the next message that $p$ will read from the channel: if $next(p) = i, 1 \leq i \leq length(M) = k$, and $M = \langle M_1, M_2, \ldots, M_k \rangle$, then $M_i$ is the next message process $p$ will read. If $next(p) > k$, then $p$ has read all the messages that have been broadcast. Initially $M$ is empty and $next(p) = 1$ for each $p$ in $P$.

A process $p$ can access the channel $B$ using two operations, $send_p$ and $receive_p$. The operation $send_p(m)$, where $m$ is a message, has the effect of appending the message $m$ to the end of the sequence $M$. Send operations performed by different processes are mutually exclusive, i.e., if two or more processes simultaneously attempt to perform a send operation, one of them will succeed and the others will have to wait.

The operation $receive_p(m)$ has the effect

```
**await** $next(p) \leq length(M)$;
$m := M_{next(p)}$; $next(p) := next(p)+1$;
```

The receive operations can be performed simultaneously by different processes, i.e., no mutual exclusion is assumed for receive operations. However, receive and send operations are mutually exclusive.

This models broadcasting as a readers-writers problem, with the readers being the processes that attempt to receive and the writers being the processes that attempt to send. As in the original readers-writers problem, writers exclude each other and all readers, but readers may read simultaneously. Fairness assumptions about broadcasting can be made in the same way as in the original problem: we assume that receivers have priority over senders, and that senders are treated fairly. The latter means that a sender that wants to send will eventually be able to perform the send operation.

The model includes, besides the actual broadcast transmission, also the buffering of incoming messages. The messages of $M$ in the range $next(p) \ldots length(M)$ correspond to messages that have been broadcast on the channel but which have not yet been received by process $p$. Hence, these messages must be in a buffer associated with process $p$. We will assume that each process has an unbounded message buffer in which to store the incoming messages (the value of $next(p)$ can be arbitrarily much smaller than $length(M)$).

This model entails a number of properties that we assume about the broadcasting network. First, broadcasting is reliable, i.e., no message is lost, corrupted or duplicated. Secondly, the messages from a process are received by all processes in the same order as they were sent by the process. Third, every process receives the same (global) sequence of broadcast messages. This implies that a process also receives its own messages, i.e., that messages are echoed back to the sender.

These assumptions are realistic for present LAN architectures, such as contention based bus architectures or token ring architectures, but usually require some higher level

Figure 1: Process states

protocol to guarantee reliable transmission and echoing of messages back to the sender. Such a higher level protocol can also be used to enforce the global ordering property (see e.g. [CM]). The assumptions may also be satisfied in point-to-point networks by a suitable choice of protocol (see e.g. [Sch]). A more detailed study of how the model is realized on different network architectures is beyond the scope of this paper.

The case when each process $p$ only has a bounded buffer of length $b(p)$ can be modeled by postulating that $next(p)$ can be at most $b(p)$ positions behind, i.e., that only elements in the range $M_{k-b(p)+1}, \ldots, M_k$ can be received by a process $p, k = length(M)$. This is achieved by changing the receive operation to

```
await $next(p) \leq length(M)$;
$next(p) := max\ (next(p), length(m) - b(p) + 1)$;
$m := M_{next(p)}$; $next(p) := next(p) + 1$;
```

In this model, messages may be lost due to buffer overflow (incoming messages push old messages out of the buffer, before they are received by the process).

For simplicity we will assume that the processes have unbounded message buffers. In practice, this is not, however, a very realistic assumption. We will therefore require that the implementation proposed for multi-process handshaking is reasonable in the sense that it also works when buffers are bounded, provided that the processes are "reasonably fast". Essentially, this assumption means that a processes is fast enough to be able to carry out the computations triggered by a message during the time that the next message is being received in the buffer. This means that messages do not accumulate in the buffers and hence that only a small buffer is needed for each process.

## 3 The multi-process handshaking mechanism

In this section we describe an implementation of multi-process handshaking for networks where a brodcasting facility is available. We assume that processes are uniquely identified by a process identifier, and actions uniquely identified by an action identifier. A process is assumed to have knowledge of those processes which participate in actions that it itself can participate in.

Consider a system of processes which communicate by executing joint actions. A process $p$ that participates in the joint action system will circulate through four states: *Ready, Waiting, Collecting* and *Executing.* Its behaviour is as follows:

1. Process $p$ starts off in the *Executing* state by executing its initialization statements, after which it goes to the *Ready* state.

2. Process $p$ evaluates the guards for the action calls in the *Ready* state. It is willing to participate in all the actions for which the guard evaluates to true. It then sends a Willing message to all other processes in the system, informing them of which actions it is willing to participate in. Process $p$ remains in the Ready state until it receives its own Willing message. All messages it receives before its own Willing message are ignored.

3. When $p$ receives its own Willing message it goes to the *Waiting* state, where it receives Willing messages from other processes in the system. For each received Willing message, $p$ records those actions that the sending process has indicated its willingness to participate in.

4. A process can choose an action to be executed only when it holds a token, which circulates among the processes in the Waiting state. When $p$ receives a Token message it checks whether there are any enabled actions, and if one is found it sends a Select message for this action. The process $p$ must also take measures to pass the token to an other process in the Waiting state.

5. If $p$ receives a Select message and it is itself involved in the chosen action, it commits itself to this action and goes to the *Collecting* state. If $p$ is not involved in the selected action it just notices which processes were committed to the selected action and records the fact that these processes are not willing to participate in any action.

6. When $p$ has become committed to an action it goes to the *Collecting* state, where the processes participating in the selected action exchange the values of the variables needed in the action. If $p$ has not yet been able to pass the token to a free process in the Waiting state, it remains in the Collecting state until it receives a Willing message from some free process, and then sends the token to this process.

7. When $p$ has received a Collect message from each process participating in the selected action it goes to the *Executing* state, where it executes the body of the action, using the variable values received in the previous stage for the nonlocal variables of the action body. After this it continues to the *Ready* state again and the cycle is repeated.

## 3.1 Messages

The processes participating in the action system reach agreement about which action should be executed by sending and receiving messages over the broadcasting channel in a multi-process handshaking protocol. A process stores the information received from other processes in its local variables. The following types of messages are used in the handshaking protocol:

**Willing [p,A]** : $p$ is the identifier of the sending process and $A$ is the set of actions $p$ is willing to participate in.

**Select [p,s,P]** : $p$ is the identifier of the sending process, $s$ is the identifier of the selected action and $P$ is the set of processes participating in $s$.

**Collect [p,V]** : *p* is the identifier of the sending process and *V* is the set of local variables in process *p* that are needed in the selected action.

**Token [p,Q,H]** : *p* is the identifier of the sending process, *Q* is a list of processes that will receive the token after *p*, and *H* is a list of processes that have held the token without being able to select any action. When a process *p* receives a Token message it checks if it is the first process in the token list *Q*, i.e., if *head(Q) = p*, and accepts the token only if this is the case.

## 3.2 Variables

Each process maintains a set of local variables when executing the joint action protocol. We denote a variable in process *p* by indexing the variable with the identifier of the process. For simplicity, indices are left out where the meaning is obvious. Variables of set or list type are identified by capital letters. A process *p* has the following local variables:

**running** : A boolean variable, initially *true*. When it is set to *false* in the *Terminate* action it forces the processes to terminate.

**Known** : The set of actions that process *p* can participate in.

**Processes** : An array containing, for each action *a* (that *p* participates in), the set of processes participating in the action *a*.

**WillingFor** : An array containing, for each action *a* (that *p* participates in), the sets of processes that *p* knows are willing to participate in the action *a*.

**HasToken** : Boolean variable, *true* when *p* holds the token.

**selected** : The identifier of the action that process *p* is committed to (this variable has value *nil* when *p* is not committed to any action).

**A** : The set of actions that *p* is willing to participate in.

**Free** : A list of processes that *p* knows are not committed to any action.

**Seen** : The set of processes that have held the token without being able to select any action.

**M** : Variable used for receiving messages.

## 3.3 Algorithms

We describe the algorithm for a process *p* participating in the multi-process handshaking protocol. A process can only access its own local variables. Processes communicate via the broadcasting network using two predefined procedures *Send(M)* and *Receive(M)*, where *M* is a message. A process manipulates its variables of set type using the set operators $\cup$ (set union) and $\setminus$ (set subtraction). By $\langle A \rangle$ we denote the list containing the same elements as the set A, with the elements in any order. For simplicity, we write $\langle r \rangle$ for a list containing only one element r, instead of $\langle \{r\} \rangle$.

The processes use the following predefined functions for manipulating variables of list type:

**append(L,K)** : *L* and *K* are lists of process or action identifiers. The function returns the list formed by appending the elements of *K* to the list *L*.

**remove(L,K)** : *L* and *K* are lists of process or action identifiers. The function returns the list formed by removing the elements of *K* from the list *L*. If an element in *K* is not included in *L*, then the elements in *L* are unaffected.

**head(L)** : *L* is a list of process or action identifiers. The function returns the first element of *L*.

**empty(L)** : returns the value *true* if the list *L* is empty.

The algorithm is as follows:

```
var
    running: boolean := true;
    WillingFor: array [ActionId] of set of ProcessId;
    Seen, ReceiveFrom, H, P: set of ProcessId;
    A : set of ActionId;
    HasToken: boolean;
    Free, Q: list of ProcessId;
    s, t, selected: ActionId;
    p, r: ProcessId;

procedure CheckEnabled();
if $\exists$ t $\in$ Known : WillingFor[t] = Processes[t]  then
    Free := remove(Free,$\langle$Processes[t]$\rangle$);
    selected := t;
    Send (Select[p,t,Processes[t]]);
    Q := append(Q,Free);
    if $\neg$empty(Q) then HasToken := false;  Send (Token[p,Q,Seen]) fi;
else
    Q := append(Q,Free);  Q := remove(Q,$\langle p \rangle$);  Q := append(Q,$\langle p \rangle$);
    Seen := Seen $\cup$ $\{p\}$;
    if p $\neq$ Head(Q) $\wedge$ $\langle$Seen$\rangle$ $\neq$ Free then
        HasToken := false; Send (Token[p,Q,Seen]);
    fi;
fi;
end CheckEnabled;
```

```
while running do
```

Ready state:

```
A := the set of actions p is willing to participate in;
for $\forall$ t$\in$A  do  WillingFor[t] := $\{p\}$ od;
for $\forall$ t$\notin$A  do  WillingFor[t] := $\{\}$ od;
Send (Willing[p,A]);
do Receive (M) until M = Willing[p,A];
```

Waiting state:

```
    Free := ⟨p⟩;  Selected := nil;
    repeat
        Receive (M);
        if M = Willing[r,A] then
            Free := append(Free, ⟨r⟩);
            for ∀ t∈A  do  WillingFor[t] := WillingFor[t] ∪ {r} od;
            if HasToken then Seen := {};  CheckEnabled() fi;
        fi;
        if M = Select[r,s,P] then
            Free := remove(Free,⟨P⟩);
            for ∀ t∈Known  do  WillingFor[t] := WillingFor[t] \ P od;
            if p ∈ P then selected := s fi;
        fi;
        if M = Token[r,Q,H] ∧ head(Q) = p then
            HasToken := true;  Q := remove(Q,⟨p⟩);  Seen := H;
            CheckEnabled();
        fi;
    until Selected ≠ nil;
```

Collecting state:

```
    V := the set of variables used in the selected action;
    Send (Collect [p,V]);
    ReceiveFrom := Processes[Selected];
    while HasToken ∨ (ReceiveFrom≠{}) do
        Receive (M);
        if M = Collect[r,V] then
            ReceiveFrom := ReceiveFrom \ {r};
            Save the values received in V in temporary variables;
        fi;
        if M = Willing[r,A] ∧ HasToken then
            HasToken := false;  Send (Token[p,⟨r⟩,{}]);
        fi;
    od;
```

Executing state:

```
    Execute the selected action, using the values received in Collect messages;

od;
```

## 3.4 Initiation and termination

As in all systems based on token passing, there is a problem with initiation and termination of the system. When the execution of an action system is started one of the processes must initially hold the token. If the system is to terminate, the last process must be able to enter the Executing state even though there is no other process it can pass the token to.

Initiation can be implemented by choosing one of the processes in a distributed election, and initially giving the token to this process. However, to avoid deadlock, the process that holds the token must have knowledge about all the other processes in the Waiting state. This means that the process who first broadcasts its Willing message when the system is started must initially be given the token, because this is the only process that will receive all Willing messages from the other processes.

This leads us to the following method for system initiation: when a process starts its execution it must first check if it is the first process starting. It does this by sending a question, asking if there are any other processes in the system. The process holding the token is responsible for sending an answer within a fixed unit of time. If the process does not receive any answer within this time unit, it must be the first process in the system, and so it assigns the token to itself and goes on to the Ready state. The length of the time unit a starting process remains waiting for an answer must be chosen to be long enough with respect to the time it takes for a process to act upon a question.

Processes in an action system can terminate by executing a predefined private action called *Terminate*. This action is considered to be a part of the implementation and does not have to be declared by the user. Each process has a local boolean variable *running* which controls its termination. When the system is started, the variable is initialized to *true* for each process. A process that wants to terminate executes the termination action and as a result of this, the variable *running* will be given the value *false*, which will terminate the process.

The other processes in the system must be informed that a process is terminating. This is because the last process must be able to enter the Executing state, even though there is no other process it can pass the token to. This can be implemented by including the number of processes in the token message. The process holding the token is responsible for decrementing this value by one each time it receives a terminating message. When the number of processes reaches one, the token holder can enter the Executing state without first sending the token to a free process.

It is also possible to implement synchronized termination in action systems by implementing the termination action as an action in which all processes in the system participate. The guard of a synchronized termination action requires that all processes in the system are willing to terminate. This will synchronize all processes in a handshake, and then terminate the action system.

## 4 Correctness of the implementation

We will now present some properties that will be needed in order to show that the implementation is correct. A correct implementation of the multi-process handshaking must satisfy the following criteria:

1. A process must participate in at most one action at a time.

2. An action is executed only if all needed processes are willing to participate in it.

3. If an action is chosen for execution, then all processes involved in that action do in fact participate in the action and execute it to completion.

4. If one or more actions are enabled, then some action will eventually be chosen for execution.

Instead of our last correctness criteria, *weak fairness*, we could require *strong fairness* to hold, i.e., if a process is infinitely often willing to participate in some enabled action, then it will infinitely often participate in some action.

The proofs are based on the broadcasting model presented earlier. The sequence $M$ of broadcast messages introduces a sequence of *virtual global states* in the system. We denote the i$^{th}$ local state of process $p$ as the values of its local variables when it receives the i$^{th}$ message in $M$. The initial local state is determined by the initial values of the local variables in $p$. The i$^{th}$ virtual global state is defined as the collection of the i$^{th}$ local states of all the processes in the system. The global state is virtual, because processes do not necessary receive the i$^{th}$ message at the same time, as measured by some global clock. The behaviour of the system does not depend on the actual global time when messages are received, and hence it is sufficent to reason about the sequence of virtual global state when proving properties of the system.

We can show that the following properties are satisfied by the protocol.

LEMMA 1 *When a process $p$ in the Ready state broadcasts a willingness message, it is not included in any willingness set $WillingFor_r$ for any process $r \neq p$ in the Waiting state.*

**Proof** Initially, when the process $p$ broadcasts its first Willing message, the statement is obviously true, as all processes initialize their *WillingFor* sets to contain only its own identifier. Consider the situation when $p$ broadcasts its n$^{th}$ willingness message, n>1. It must have reached the Ready state from the Waiting state via the Collecting and Executing states, as a result of being committed to an action $a$. When the processes in the Waiting state receive the Select message for action $a$ they either move to the Collecting state (if participating in the action) or remove the processes participating in $a$ from their *WillingFor* sets (if not participating in the action). In either case, $p$ will be removed from all willingness sets before it again enters the Ready state, and so the stated property also holds for the n$^{th}$ Willing message from process $p$. □

LEMMA 2 *If process $p$ is included in a set $WillingFor_r[a]$ in some process $r \neq p$ in the Waiting state and for some action $a$, then $p$ is in the Waiting state and is willing to participate in the action $a$.*

**Proof** According to Lemma 1, process $p$ is not included in any willingness sets, except its own, when it in the Ready state broadcasts a Willing message for the actions it is willing to participate in. It then goes to the Waiting state and remains there until it becomes committed to an action. All the other processes in the Waiting state will insert $p$ into their *WillingFor* sets when they receive the willingness message. Process $p$ can exit from the Waiting state only if it becomes comitted to some action $a$. In that case, a Select message for the action $a$ must be sent, either by $p$ or by some other process. The processes in the Waiting state will either remove $p$ from their *WillingFor* sets or go to the Collecting state when they receive the Select message. Thus, process $p$ can only be included in a set $WillingFor_r[a]$ in some other process $r$ in the Waiting state if $p$ is in the Waiting state and willing to participate in the action $a$. □

LEMMA 3 *All processes included in the list $Free_p$ in a process $p$ are in the Waiting state and not committed to any action.*

**Proof** When a process $p$ enters the Waiting state it initializes the list *Free* to contain only its own identifier. For every Willing message $p$ receives in the Waiting state, the sender is appended to the list *Free*. When $p$ receives a Select message, it removes all participating processes from *Free*. Also if $p$ itself selects an action it removes all participating processes from the list *Free*. Therefore, if a process $r$ is included in the list $Free_p$ in a process $p$, $r$ must be in the Waiting state and not committed to any action. □

Lemma 4 *A process p that enters the Waiting state will eventually either be included in the token list Q, or become committed to an action and move to the Collecting state.*

**Proof** When the process $p$ broadcasts its Willing message and enters the Waiting state all the other processes in the Waiting state will receive the Willing message from $p$ and insert $p$ in their list of free processes. If there is a processes in the Waiting state which holds the token when it receives the willingness message from $p$, it appends the list *Free* to the token list, and $p$ will thus be inserted in $Q$. If there is no process holding the token in the Waiting state, the token holder must be in the Collecting state and waiting for a willingness message. When it receives the Willing message from $p$ it inserts $p$ into the token list $Q$ and sends the token to $p$. If $p$ becomes committed to an action it moves to the Collecting state and will not be included in the token list. Thus, process $p$ will eventually either be included in the token list $Q$ or become committed to an action. □

Lemma 5 *Each process that enters the Waiting state will eventually either receive the token, or become committed to an action and move to the Collecting state.*

**Proof** According to Lemma 4, a process $p$ that enters the Waiting state will eventually either be included in the token list $Q$, or become committed to an action. When $p$ becomes included in the token list $Q$, there is only a finite number of processes in front of it in the token list. Each time the token is passed the receiving process removes itself from the head of $Q$. New processes are always appended to the rear of the list. Thus, unless $p$ becomes committed to an action and goes to the Collecting state, it will eventually receive the token. □

Lemma 6 *If process p holds the token, then* $Free_p$ *contains the identifiers of all processes in the Waiting state.*

**Proof** According to Lemma 3, the list *Free* in a process $p$ contains the identifiers of all processes $p$ knows are not committed to any action. Processes are appended to the rear of *Free* in the same order they enter the Waiting state, and the list *Free* is appended to the rear of the token list $Q$. This means that the first process in $Q$, i.e., the process holding the token, must have received the Willing messages from all other processes in the Waiting state, and have inserted these into its list *Free*. If $p$ moves to the Collecting state still holding the token, this is because the token list, and thus also its list *Free*, is empty and there are no processes in the Waiting state. Thus, the list $Free_p$ in the process $p$ holding the token will contain the identifier of every process in the Waiting state. □

Lemma 7 *When a process p sends a Token message, the token list Q contains at least one other process than p, and all processes in Q are in the Waiting state.*

**Proof** A process *p* passes the token either in the Waiting state after it has checked its willingness sets for an enabled action, or in the Collecting state after it has become committed to an action. If *p* finds an enabled action in the Waiting state it appends the list *Free* to the token list *Q*, and if *Q* is not empty the token is passed. The process *p* itself will not be included in *Q*, because it participates in the selected action. If *p* does not find any enabled action it appends the list *Free* to the token list and then inserts itself at the end of *Q*. It sends the token only if it is not the first process in *Q*, i.e., if there is at least one other process than itself in the token list. According to Lemma 3, all processes in *Free* are in the Waiting state, and when *p* inserts itself in the token list it also remains in the Waiting state, so all processes in *Q* are in the Waiting state. If *p* passes the token in the Collecting state to a process *r*, then *p* must have received a Willing message from *r*, which must be in the Waiting state, because *p* holds the token and no actions can be selected. Thus, whenever a Token message is sent, the token list contains at least one process different from the sender, and all the processes in *Q* are in the Waiting state. □

LEMMA 8 *The token cannot be lost.*

**Proof** This follows from Lemmas 5 and 7. All processes that enter the Waiting state will eventually either receive the token or become committed to some action. When the token is passed, there is at least one free process is in the Waiting state, other than the token sender, which will receive the token. This means that the process holding the token will eventually find another process it can send the token to and the receiving process is not committed to any action, so the token cannot be lost. □

Based on these lemmas we can show that our implementation satisfies the stated correctness criteria. We assume that each execution of an action eventually terminates, for every process participating in the action.

LEMMA 9 *A process will participate in at most one action at a time.*

**Proof** A process *p* can become committed to an action only if it is involved in some action *a* that is selected for execution. After *p* has become committed to a selected action, it moves to the Collecting state and will ignore all further Select messages, until it again enters the Waiting state. This can only happen after it has finished execution of the previous action, so a process can only participate in at most one action at a time. □

LEMMA 10 *An action is executed only if all the needed processes are willing to participate in it.*

**Proof** An action *a* will be chosen by a process *p* only if *p* holds the token and the set *WillingFor[a]* in *p* is equal to *Processes[a]*. According to Lemma 2, all the processes in *WillingFor[a]* are in the Waiting state and willing to participate in the action *a*. Process *p* will send the Select message for action *a* before it passes the token. This means that all processes participating in *a* will still be in the Waiting state and willing to participate in *a* when they receive the Select message, because no other selections can take place as long as *p* holds the token. Thus, an action can only be executed if all processes participating in it are willing to execute it. □

Lemma 11 *If an action is chosen for execution, then all processes involved in the action do in fact participate in it and carry out the action.*

**Proof** This follows directly from Lemma 10. When a process $p$ sends a Select message for an action $a$, all the participating processes are in the Waiting state and willing to participate in $a$. They will all receive the Select message and move to the Collecting state, where each process will send a Collect message containing the values needed for carrying out the action. After a process has received a Collect message from each participating process, it will execute the chosen action. The token holder can be delayed from entering the Executing state if there is no process in the Waiting state it can send the token to. However, some process will eventually finish its execution of a selected action and enter the Waiting state. Process $p$ will send the token to this process and then move to the Executing state. $\square$

Lemma 12 *If one or more actions are enabled, then some action will be chosen for execution within finite time.*

**Proof** Assume that there is an enabled action $a$, i.e., that all processes participating in $a$ are willing to execute $a$. This means that all the processes needed in the action are in the Waiting state, and not committed to any action. One of the processes participating in $a$, say $p$, was the first process to broadcast its Willing message. Thus, $p$ must have recorded the Willing messages of all the other processes participating in $a$, and included these processes in its *WillingFor* sets. According to Lemma 5, $p$ will eventually either receive the token or become committed to an action. If $p$ receives the token it will send a Select message for $a$ or some other enabled action. In either case, some action will eventually be chosen for execution. $\square$

We summarize these results in the following Theorem:

Theorem 1 *The implementation of multi-process handshaking satisfies the stated correctness criteria 1, 2, 3, and 4.*

**Proof** This follows directly from Lemmas 9 to 12. $\square$

The presented implementation is weakly fair, but not strongly fair. This is because a process $p$ can be continously willing to participate in some action which is infinitely often enabled, but this action will never be enabled when $p$ gets the token, and no other process will ever select this action.

## 5 Efficiency of the implementation

In this chapter we analyze the efficiency of the implementation as measured by the number of messages sent per executed action. First upper and lower bounds for the number of messages sent per executed action is presented. After that we present the results obtained from an implementation of the multi-process handshaking protocol on a simulated broadcasting network.

## 5.1 Upper and lower bounds on the number of messages

Consider an action system with $N$ processes. We denote by $P_a$ the number of processes participating in the action $a$. Thus, $P_a$ must be less than or equal to $N$.

According to Lemma 10 an action $a$ can only be executed if all participating processes are in the Waiting state and willing to participate in the action. Thus, $P_a$ Willing messages, one Select message and $P_a$ Collect messages must be transmitted before the action $a$ can be executed. Additionally, a number of Token messages will also be transmitted.

THEOREM 2 *A lower bound on the number of messages transmitted before an action a with $P_a$ participating processes can be executed is $2P_a + 2$*

**Proof** At least one Token messages must be transmitted before an action is choosen. The process that selects the action $a$ must hold the token when it sends the Select message for $a$. This process has received the token in a Token message. Thus, the lower bound on the number of messages transmitted before an action is executed is obtained by adding $P_a$ Willing messages, one Select message, $P_a$ Collect messages and one Token message. □

To obtain an expression for the upper bound on the number of messages transmitted before an action is executed we observe the following property:

LEMMA 13 *If all processes in the Waiting state have held the token without being able to select any action, the token will not be passed on before a new process enters the Waiting state.*

**Proof** When a process $p$ that holds the token checks its *WillingFor* sets in the Waiting state and does not find any enabled action, it inserts itself into the set *Seen*, which is included in the Token message. Thus, the set *Seen* in the token will contain the identifiers of all the processes that have held the token, but not been able to select any action. The set *Seen* is initialized to the empty set every time the token holder receives a Willing message, either in the Waiting state or in the Collecting state. Before $p$ passes the token it checks if the set *Seen* is equal to the set *Free*, and if this is the case it holds the token until it receives the next Willing message. According to Lemma 6, the set *Free* in the process holding the token contains the identifiers of all processes in the Waiting state. Thus, if all processes in the Waiting state have held the token without being able to select any action, the token holder will not pass the token until a new process enters the Waiting state. □

We can now give an upper bound on the number of Token messages that can be broadcast before an action is selected.

LEMMA 14 *An upper bound on the number of Token messages broadcasted before an action a is selected is given by*

$$\frac{1}{2}(N^2 - N) \tag{1}$$

**Proof** In the worst case all processes in the system must participate in the selected action. Consider a situation where only one process $p$ is in the Waiting state, and holding the token. Process $p$ has not been able to select any action when it received the token, and so it has inserted itself into the set *Seen* and remained waiting for a process to send the token to. The next process that enters the Waiting state will receive the token from $p$. This process cannot either select any action, so it inserts itself in the set *Seen*, moves itself to the rear of $Q$ and sends the token back to $p$. Process $p$ will not pass the token until it receives a Willing message from some process entering the Waiting state, because all processes in the Waiting state have held the token and none of them have selected any action. This will continue as processes one by one enter the Waiting state: the token is circulated among the processes in the Waiting state once every time a process enters the Waiting state, and each time the number of processes that will receive the token is increased by one. Thus, as we assume that all processes in the system must participate in the action, the upper bound on the number of Token messages is given by the sum $1 + 2 + \ldots + N - 1$, which can be written as

$$\sum_{i=1}^{N-1} i = \frac{1}{2}(N^2 - N) \tag{2}$$

□

We can now state an upper bound on the number of messages that are sent in the worst case before an action is executed. The worst case will occur if all processes in the system participate in the selected action, i.e., if $P_a = N$.

THEOREM 3 *An upper bound on the number of messages sent before an action is executed in a system with N processes is given by the expression*

$$\frac{1}{2}(N^2 + 3N) + 1 \tag{3}$$

**Proof** If all $N$ processes in the system participate in the selected action, in the worst case $N$ Willing messages are sent, one Select message, $N$ Collect messages and $1/2(N^2 - N)$ Token messages. If we add all the messages sent we get the expression

$$2N + \frac{1}{2}(N^2 - N) + 1 = \frac{1}{2}(N^2 + 3N) + 1 \tag{4}$$

□

Observe that the values transmitted in the Collect messages could be inserted into the Willing messages. This would mean that the minimum and maximum number of messages required for an action in the worst case would decrease by $N$. However, this does not change the asymptotic complexity of the algorithm.

## 5.2 Results from measurements

We have measured the efficiency of the implementation experimentally and compared it to the theoretical results presented in the previous section. A number of joint action systems were constructed and executed using the implementation, and the number of messages sent per executed action was measured.

The number of messages actually sent over the broadcasting network per executed action can be compared to the calculated minimum and maximum number of possible messages per action. In table I, $N$ denotes the number of processes in the system, $P_a$ the number of processes participating in an action $a$, $A$ the number of actions in the system, *min* and *max* the theoretical minimum respectively maximum number of messages sent per executed action, and $M_a$ the measured number of messages sent per action. For simplicity, the number of processes participating in an action, $P_a$, is the same for all actions in a test program. A collection of 11 test programs was constructed with different characteristics, and their performance was measured. The results are given in Table I.

| **Test** | **P**$_a$ | **N** | **A** | **min** | **max** | **M**$_a$ |
|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 2 | 6 | 6 | 6.0 |
| 2 | 2 | 10 | 10 | 6 | 66 | 6.2 |
| 3 | 2 | 20 | 20 | 6 | 231 | 6.2 |
| 4 | 3 | 3 | 3 | 8 | 10 | 10.0 |
| 5 | 4 | 4 | 4 | 10 | 15 | 13.1 |
| 6 | 4 | 12 | 6 | 10 | 91 | 11.2 |
| 7 | 5 | 5 | 2 | 12 | 21 | 15.7 |
| 8 | 5 | 5 | 5 | 12 | 21 | 16.0 |
| 9 | 6 | 6 | 6 | 14 | 28 | 18.9 |
| 10 | 10 | 10 | 10 | 22 | 66 | 29.5 |
| 11 | 10 | 12 | 3 | 22 | 91 | 29.5 |

Table I: Results of test runs

Each test program was executed 10 times with different numbers of actions executed in each run (by varying the problem size). No significant differences in the number of messages transmitted per executed action could be observed between large and small problems.

As can be seen from the presented results, the number of messages transmitted for each executed action is very close to the theoretical minimum value. Even though the upper bound on the number of messages per executed action is proportional to the square of the number of participating processes, the actual number of messages transmitted is proportional to $P_a$, the number of participating processes.

The joint action protocol was implemented on a simulated broadcasting network. The implementation was written in Modula-2 on a Sun-3/160 workstation, and it consists of about 2000 lines of Modula-2 code. Processes and actions were implemented as objects whose internal structure is hidden from the user and can only be manipulated through a set of predefined procedures and functions. This set of datatypes, procedures and functions constitute the implementation.

## 6 Conclusions

We have presented a general and flexible communication mechanism called joint actions. A joint action is executed by synchronizing the participating processes in a multi-process handshake and executing the action in the combined statespace of the participating processes. An implementation of a multi-process handshaking protocol for processes communicating through a broadcasting network has been presented and proved to be correct. The efficiency of the algorithm has been studied by implementing it on a simulated broadcasting network. From the measurements made on the implementation of the protocol it can be seen that the number of messages transmitted over the broadcasting network is very close to the theoretical minimum number of messages per action, which is proportional to the number of processes involved in the actions.

## References

**[Asp87]** Aspnäs, M., Implementing joint actions in Modula-2, M.Sc. Thesis, Åbo Akademi, 1987 (in Swedish).

**[Bag86]** Bagrodia, R., A distributed algorithm to implement the generalized alternative command of CSP, Proc. 6th International Conf. on Distributed Computing Systems, May 1986, pp. 422–427.

**[Bag87]** Bagrodia, R., A distributed algorithm to implement N-party rendezvous, Proc. 7th Conf. on Foundations of Software Technology and Computer Science, Pune, India, Dec 1987, Lecture Notes on Computer Science 287, Springer Verlag, 1987.

**[BaKu83]** Back, R.J.R. and Kurki-Suonio, R., Decentralization of process nets with centralized control, Proc. Second Annual ACM Symposium on Principles of Distributed Computing, Montreal, Canada, 1983, pp. 131–142.

**[BaKu84a]** Back, R.J.R. and Kurki-Suonio, R., A case study in constructing distributed algorithms: distributed exchange sort, Proc. Winter School on Theoretical Computer Science, Finnish Society for Information Processing, Lammi, Finland, 1984, pp. 1 –33.

**[BaKu84b]** Back, R.J.R., and Kurki-Suonio, R., Co-operation in distributed systems using symmetric multi-process handshaking, Reports in Computer Science and Mathemathics 34, Åbo Akademi, 1984.

**[BaHaKu85]** Back, R.J.R., Hartikainen, E. and Kurki-Suonio, R., Multi-process handshaking on broadcasting networks, Reports on Computer Science and Mathemathics 42, Åbo Akademi, 1985.

**[BaKu88]** Back, R.J.R. and Kurki-Suonio, R., Distributed cooperation with action systems, ACM Trans. on Programming Languages and Systems, 10:4 (Oct. 1988), pp. 513–554.

**[BaEkKu]** Back, R.J.R., Eklund, P. and Kurki-Suonio, R., A fair and efficient implementation of CSP with output guards, Reports on Computer Science and Mathemathics 38, Åbo Akademi, 1984.

**[Bor]** Bornant, R., A protocol for generalized Occam, Software - Practice and Experience, 16:9 (September 1986), pp. 783–799.

**[BuSi83]** Buckley, G.N. and Silberschatz, A., An effective implementation for the generalized input-output construct of CSP, ACM Trans. on Programming Languages, 5:2 (April 1983), pp. 223-235.

**[CM]** Chang, J.M. and Maxemchuk, F., Reliable broadcast protocol, ACM Trans. on Computer Systems, 2:3 (Aug. 1984), pp. 251-273.

**[ChMi88]** Chandy, M. and Misra, J., *Parallel Program Design, a Foundation*, Addison–Wesley, Reading, Mass.,1988.

**[DoD]** Department of Defense, Ada Programming Language, ANSI/MIL-STD-1815A-1983.

**[Ekl]** Eklund P., Synchronization of multiple processes in common handshakes, Reports on Computer Science and Mathematics 39, Åbo Akademi, 1984.

**[Hoa]** Hoare, C.A.R., Communicating sequential processes, Communications of the ACM, 21:8 (August 1978), pp. 666-677.

**[Inm]** Inmos Limited, *Occam 2 Reference Manual*, Prentice Hall, New York, 1988

**[Ram]** Ramesh, S., A new and efficient implementation of multiprocess synchronization, Proc. PARLE Parallel Architectures and Languages Europe, Lecture Notes in Computer Science 259, Springer Verlag, 1987.

**[Sch]** Schneider, F.B., Synchronization in distributed programs, ACM Trans. on Programming Languages and Systems, 4:2 (April 1982), pp. 125-148.

**[Sis]** Sistla, A.P., Distributed algorithms for ensuring fair interprocess communication, Proc. 3rd ACM Conf. on Principles of Distributed Computing, 1984, pp. 266-277.