

On-Demand Coordination of First-Order Multiparty Interactions

Peiyi Tang

Department of Mathematics and Computing
University of Southern Queensland
Toowoomba QLD 4350 Australia

Yoichi Muraoka

School of Science and Engineering
Waseda University
Tokyo 169 Japan

Abstract

First-order multiparty interaction is a key abstraction in the distributed programming model, called Interacting Processes (IP). In this paper, we propose an efficient algorithm for coordinating first-order multiparty interactions on demand. By taking advantage of multi-threading, this algorithm requires less messages than the algorithm proposed by Joung and Smolka [1]. It also supports independent compilation of modules in IP, allowing them to be used as software components for open distributed and parallel computing.

1 Introduction

First-order multiparty interaction is one of the fundamental abstractions of the powerful distributed programming model called Interacting Processes (IP) proposed by N. Francez and I. R. Forman [2]. We found recently that the IP model is also extremely suitable for parallel programming due to its ease-of-programming, high degree of parallelism and support for modular programming [3]. It is our belief that the IP model may become a mainstream programming model for the future parallel and distributed computing on computational grids [4].

The key to the acceptance and success of the IP model is efficient implementation of its first-order multiparty interaction. A process can be ready to participate in several multiparty interactions expressed in guard selection or iteration commands, but is allowed to participate in only one interaction at a time. The processes need to coordinate to decide which interactions to execute.

The first-order multiparty interaction in IP allows participants to change dynamically at run time. The existing algorithm for coordinating first-order multiparty interactions by Joung and Smolka [1] is based on the set of all processes which **may** participate in an interaction. The cost of reaching consensus to select the interaction is $4m^2 - 2m$ messages, where m is the

size of this set which is usually large.

In this paper, we propose an on-demand algorithm using multi-threading for coordinating first-order multiparty interactions in IP. Our algorithm is more efficient and the cost of establishing the consensus to select and execute the interaction is $2lw$ messages, where l is the number of parties of the interaction and w the number of interactions to be participated by the processes in conflict. Both l and w are expected to be much smaller than m described above in most applications.

Our algorithm also supports independent compilation of modules in IP, allowing them to be used as software components for open distributed and parallel computing.

Section 2 introduces the first-order multiparty interactions in IP and defines the guard scheduling problem for them. Our on-demand coordination algorithm for the problem is described in Section 3. The correctness and the message complexity of the algorithm are presented in Sections 4 and 5, respectively.

2 Guard Scheduling Problem for First-Order Multiparty Interactions

A multiparty interaction is not enabled until all the participating processes are ready to execute it. In the IP model, a process can participate in a multiparty interaction through an *interaction statement* of format: $a[.]$, where a is the name of the interaction and the square brackets include the statements to be executed by this process when the interaction is executed.

The IP model extends the CSP [5] guard statement with multiparty interactions. The enhanced guard selection statement in IP is of format:

$$[B_1 \& a_1[.] \rightarrow S_1 \square \dots \square B_n \& a_n[.] \rightarrow S_n]$$

where B_i ($i = 1, \dots, n$) are boolean expressions of local states, called *guarding predicates*, and a_i ($i = 1, \dots, n$) are the multiparty interactions in which the process can participate, called *guarding interactions*; both are optional

A guard is *ready* if its guarding predicate is true. The guarding interaction of a ready guard is *enabled* if all the guards with the same interaction in other processes are also ready. An enabled guarding interaction can be selected for execution. If many guarding interactions in a guard selection statement are enabled, only one of them can be selected. Whether an enabled guarding interaction is actually selected depends on the result of coordination among the processes involved. After the guarding interaction a_i is selected and executed, the corresponding statement S_i is executed.

The guard iteration statement is similar and of format:

$$*[B_1 \& a_1[\dots] \rightarrow S_1 \square \dots \square B_n \& a_n[\dots] \rightarrow S_n]$$

The difference is that the enclosed guard selection will be executed repeatedly until none of the guarding predicates is true.

```

1 team TABLE(value  $n$  : integer) ::
2 [
3    $\parallel_{i=0, n-1}$  role  $R_i$  ::
4      $s_i := 'thinking'$ ;
5      $*[s_i = 'thinking' \rightarrow s_i = 'hungry']$ 
6      $\square$ 
7      $s_i = 'hungry' \& get\_forks_i[s_i := 'eating']$ 
8      $\rightarrow give\_forks_i[]$ 
9   ]
10  $\parallel_{i=0, n-1}$  process  $F_i$  ::
11    $*[get\_forks_i[] \rightarrow give\_forks_i[]]$ 
12    $\square$ 
13    $get\_forks_{(i+1) \bmod n}[] \rightarrow give\_forks_{(i+1) \bmod n}[]$ 
14 ]
15 ]

```

Figure 1: IP Module for Dining Philosophers Problem

An IP program module (called **team**) for the dining philosophers problem is shown in Figure 1. There are n fork processes, F_i , and n formal processes, R_i , (called **roles**) to be invoked by actual dining philosopher processes outside of the module. There are n three-party interactions, get_forks_i . The participants of interaction get_forks_i are the fork processes F_i and $F_{(i-1) \bmod n}$ and the dining philosopher process invoking R_i . When all the dining philosophers are hungry and all the forks are on the table to be picked up, the coordination among all the processes can be illustrated by the bipartite graph in Figure 2, assuming roles R_i are invoked by dining philosophers processes P_i , respectively. The thick bars I_i in the graph represents the multiparty interactions get_forks_i and the circles the coordinating processes. The edges represent the possible participations and a process can participate in only one interaction.

The guard scheduling problem for coordinating first-order multiparty interactions in IP can now be formally

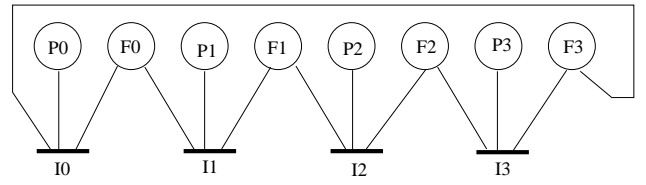


Figure 2: Bipartite Graph of Processes and Interactions

described as follows:

Given n multiparty interactions I_i ($i = 1, \dots, n$) each of which has l_i parties to be participated by distinct processes from m processes P_j ($j = 1, \dots, m$) whose identities are not known until the run time, the guard scheduling problem is to select (nondeterministically) a subset of the multiparty interactions for execution, subject to the following constraints:

1. Each interaction selected for execution must have the required number of processes to participate in it.
2. No process can participate in executions of more than one interaction.
3. If there are interactions which can be selected for execution, the selection must be finished in finite time.

3 On-demand Coordination for Guard Scheduling

We use a separate process for each multiparty interaction, also denoted I_i , to coordinate the participating processes at run-time.

When a process is ready to participate in multiparty interactions, it starts a separate thread for each of them. Let there be k_j interactions, $I_{i_1}, \dots, I_{i_{k_j}}$, in which process P_j is ready to participate. The thread started by P_j for I_{i_r} ($1 \leq r \leq k_j$) is denoted $T_{i_r}^j$. In addition, process P_j starts a master thread denoted M^j to manage all the inter-process communications and coordinate threads $T_{i_1}^j, \dots, T_{i_{k_j}}^j$.

We assume that the underlying communications between processes and threads are asynchronous and carried out on reliable FIFO channels. Each thread or process also maintains a queue for incoming messages.

Our guard scheduling algorithm consists of three protocols for $T_{i_r}^j$, I_{i_r} and M^j . Thread $T_{i_r}^j$ communicates with process I_{i_r} and thread M^j . Threads $T_{i_r}^j$ ($r = 1, \dots, k_j$) also communicate with each other. The pseudo codes of the protocols for $T_{i_r}^j$, I_{i_r} and M^j are shown in Figures 3, 4 and 5, respectively.

The basic idea behind the algorithm is simple. $T_{i_r}^j$ first sends a request with its identity to I_{i_r} and then waits for a message called *All-Met* from it. I_{i_r} records the request from $T_{i_r}^j$ and will not send *All-Met* back until it receives all the I_{i_r} requests it needs. The protocol of $T_{i_r}^j$ enters the second phase upon receiving *All-Met*. Depending on the states of other threads of P_j , $T_{i_r}^j$ can

```

*[ 1.0
  state = 'init' → send Request( $P_j, p$ ) to  $I_{i_r}$ ;
  state := 'req-sent'
□ 1.1
  state = 'req-sent'; receive All-Met from  $I_{i_r}$  →
  synchronized( $a[ ]$ ) {
    if ( $a[1..k] = 0$ )
       $a[r] := 1$ ; send Commit( $P_j, p$ ) to  $I_{i_r}$ ;
      state := 'commit-sent'
    else if ( $a[1..(r-1)] \neq 0 \wedge a[(i+1)..k] = 0$ )
      send Withdraw( $P_j, p$ ) to  $I_{i_r}$ ;
      send Re-try( $T_{i_r}^j$ ) to  $M^j$ ;
      state := 're-try'
    else if ( $a[(r+1)..k] \neq 0$ )
       $a[r] := 1$ ; state := 'pending'
    endif
  }
□ 1.2
  state = 'req-sent'; receive Stop from  $M^j$  →
  send Abort( $P_j, p$ ) to  $I_{i_r}$ ;
  send Ready-to-Die( $T_{i_r}^j$ ) to  $M^j$ ;
  state := 'ready-to-die'
□ 1.3
  state = 'pending'; receive Continue from  $T_{i_{r'}}^j$  ( $r < r'$ ) →
  send Commit( $P_j, p$ ) to  $I_{i_r}$ ; state := 'commit-sent'
□ 1.4
  state = 'pending'; receive Stop from  $M^j$  →
  send Withdraw( $P_j, p$ ) to  $I_{i_r}$ ;
  send Ready-to-Die( $T_{i_r}^j$ ) to  $M^j$ ;
  synchronized( $a[ ]$ ) { $a[r] := 0$ }; state := 'ready-to-die'
□ 1.5
  state = 'commit-sent';
  receive Succeed( $P_{j_1}, \dots, P_{j_q}$ ) from  $I_{i_r}$  →
  record process list ( $P_{j_1}, \dots, P_{j_q}$ );
  send Finish( $T_{i_r}^j$ ) to  $M^j$ ;
  state := 'success'
□ 1.6
  state = 'commit-sent'; receive Fail from  $I_{i_r}$  →
  synchronized( $a[ ]$ ) {
     $a[r] := 0$ ;
    if ( $a[1..(r-1)] \neq 0$ )
      let  $r'$  be the largest integer such that
         $1 \leq r' \leq (r-1) \wedge a[r'] = 1$ ;
      send Continue to  $T_{i_{r'}}^j$ 
    endif
    send Re-Try( $T_{i_r}^j$ ) to  $M^j$ ; state := 're-try'
  }
□ 1.7
  state = 're-try'; receive Stop from  $M^j$  →
  send Ready-to-Die( $T_{i_r}^j$ ) to  $M^j$ ; state := 'ready-to-die'
□ 1.8
  state = 're-try'; receive Try-Again from  $M^j$  →
  state := 'init'
□ 1.9
  state = 'ready-to-die'; receive Kill from  $M^j$  → kill itself
]

```

Figure 3: Protocol of Thread $T_{i_r}^j$

either send a commitment or withdrawal to I_{i_r} , or suspend the decision until late. Only one thread $T_{i_r}^j$ of P_j is allowed to send commitment at a time. After receiving commitment from $T_{i_r}^j$, I_{i_r} sends message *Fail*

back to $T_{i_r}^j$ if it has ever received a withdrawal from its participating processes or message *Success* if it has received the commitments from all of them. Upon receiving message *Success*, $T_{i_r}^j$ signals M^j to kill all other threads of P_j . If $T_{i_r}^j$ receives message *Fail*, it continues to go to the next round of coordination.

```

*[ 2.0
  state = 'meeting'; receive Request( $P_j, p$ ) from  $T_{i_r}^j$  →
  record  $P_j$  as the participant for party  $p$ ;  $nReq + +$ ;
  if ( $nReq = q$ )
     $nReq := 0$ ;
    for each of the  $q$  recorded  $P_j$  send All-Met to  $T_{i_r}^j$ ;
    state := 'all-met'
  endif
□ 2.1
  state = 'meeting'; receive Abort( $P_j, p$ ) from  $T_{i_r}^j$  →
  discard  $P_j$  as the participant for party  $p$ ;
   $nReq - -$ ;
□ 2.2
  state = 'all-met'; receive Commit( $P_j, p$ ) from  $T_{i_r}^j$  →
  if ( $nW = 0$ )
    record  $P_j$ 's commitment for party  $p$ ;
     $nC + +$ ;
    if ( $nC = q$ )
      let  $P_{j_1}, \dots, P_{j_q}$  be the  $q$  recorded
         $P_j$  with commitment;
      for ( $k = 1$  to  $q$ ) send Succeed( $P_{j_1}, \dots, P_{j_q}$ ) to  $T_{i_r}^{j_k}$ ;
       $nC := 0$ ;
      state := 'success'
    endif
  else
    send Fail to  $T_{i_r}^j$ ;
     $nC + +$ ;
    if ( $nC + nW = q$ )
       $nW := 0$ ;  $nC := 0$ ;
      discard all the records;
      state := 'meeting'
    endif
  endif
□ 2.3
  state = 'all-met'; receive Withdraw( $P_j, p$ ) or
  Abort( $P_j, p$ ) from  $T_{i_r}^j$  →
  if ( $nW = 0 \wedge nC \neq 0$ )
    let  $P_{j_1}, \dots, P_{j_{nC}}$  be the  $nC$  recorded
       $P_j$  with commitment;
    for ( $k = 1$  to  $nC$ ) send Fail to  $T_{i_r}^{j_k}$ ;
  endif;
   $nW + +$ ;
  if ( $nC + nW = q$ )
     $nW := 0$ ;  $nC := 0$ ;
    discard all the records;
    state := 'meeting'
  endif
]

```

Figure 4: Protocol of Interaction Process I_{i_r}

In this paper, we use CSP-like iterative guard commands as follows to present the communication protocols:

$$*[g_1 \rightarrow S_1 \square \dots \square g_n \rightarrow S_n]$$

Guard commands are referred to as *rules* in this paper. All the rules in the three protocols are numbered for ease of reference.

We assume in our algorithm that there is a total order among all the interactions involved. Without loss of generality, we assume $I_1 < \dots < I_n$ and give the

highest priority to I_1 . To simplify the notation, we use k to denote k_j . Among the k interactions, I_{i_1}, \dots, I_{i_k} , we assume $I_{i_1} < \dots < I_{i_k}$, i.e. $i_1 < \dots < i_k$. This total order is essential both to prevent deadlock and to ensure progress in each round of coordination.

Process P_j maintains a bit map $a[1..k]$ shared by all its threads $T_{i_r}^j$ ($r = 1, \dots, k$). The initial value of every bit of $a[1..k]$ is 0. Bit $a[r]$ is set to 1 when thread $T_{i_r}^j$ has either committed P_j to the corresponding interactions I_{i_r} , or is in state '*pending*'. A pending thread will eventually commit to its corresponding interaction unless it is stopped and killed.

Since the bit map $a[]$ is shared by all the threads of process P_j , its accesses should be put into critical sections. We use *synchronized* block as in Java [6] to indicate critical sections in the protocols in this paper.

Message *Continue* is sent by $T_{i_r}^j$ to wake up a pending thread, upon receiving *Fail* from I_{i_r} (rule 1.6). Upon receiving *Continue*, the pending thread proceeds to send *Commit()* to its own interaction (rule 1.3).

In the protocol of I_{i_r} , we use q to denote l_{i_r} , the number of parties of interaction I_{i_r} .

Process I_{i_r} maintains three counters, $nReq$, nC , and nW , for the number of *Request()*s, *Commit()*s and *Withdraw()*s received, respectively. It has also data structures to record the identities of the participating processes.

Assuming that there is no deadlock in the protocol of $T_{i_r}^j$ (to be discussed in Section 4 shortly), $T_{i_r}^j$ will send either a *Commit()* or a *Withdraw()* to I_{i_r} , after receiving *All-Met* from it. I_{i_r} may receive *Abort()* from $T_{i_r}^j$ in state '*all-met*'. This is because $T_{i_r}^j$ may receive *Stop* from its M^j in state '*req-sent*' (due to the fact that another thread succeeds) before the message *All-Met* from I_{i_r} reaches it. This is the reason why I_{i_r} should be prepared to accept *Abort()* in state '*all-met*' (rule 2.3). An *Abort()* received in state '*all-met*' is treated as an *Withdraw()*. After I_{i_r} receives the responses from all the participating processes, it enters state '*success*' if it receives all *Commit()*s, or goes back to state '*meeting*' for the next round of coordination otherwise.

The main function of protocol of thread M^j in Figure 5 is to coordinate all the threads $T_{i_1}^j, \dots, T_{i_k}^j$. It also intercepts and relays messages between $T_{i_r}^j$ and I_{i_r} . In particular, it discards all the messages to $T_{i_r}^j$ if it has killed $T_{i_r}^j$.

Thread M^j maintains a counter, $nRetry$, to synchronize all the threads before entering the next round of coordination. The next round of coordination should not start until all the threads $T_{i_r}^j$ ($r = 1, \dots, k$) fail (rule 3.1).

After one of threads $T_{i_r}^j$ succeeds, thread M^j is responsible to send *Stop* to all the other threads (rule

```

* [ 3.0
  state = 'init' →
  spawn k threads  $T_{i_1}^j, \dots, T_{i_k}^j$  in 'init' state;
  state := 'working'; nRetry := 0; nDied := 0
□ 3.1
  state = 'working'; receive Re-Try( $T_{i_r}^j$ ) →
  nRetry ++;
  if (nRetry = k)
    nRetry := 0;
    send Try-Again to all  $T_{i_1}^j, \dots, T_{i_k}^j$ 
  endif
□ 3.2
  state = 'working'; receive Finish( $T_{i_r}^j$ ) →
  send Stop to all  $T_{i_1}^j, \dots, T_{i_{r-1}}^j, T_{i_{r+1}}^j, \dots, T_{i_k}^j$ ;
  state := 'finishing'
□ 3.3
  state = 'finishing'; receive Ready-to-Die( $T_{i_r}^j$ ) →
  send Kill to  $T_{i_r}^j$ ;
  nDied ++;
  if (nDied = k - 1)
    state := 'success'
  endif
]

```

(a) Protocol of Thread M^j

Figure 5: Protocol and State Digram of Thread M^j

3.2). Another counter, $nDied$, is used to make sure that all the other threads are killed before M^j enters state '*success*' (rule 3.3).

4 Correctness

A solution to the guard scheduling problem for coordinating first-order multiparty interactions must satisfy the requirements of *safety*, *liveness* and *progress*.

The safety requirement demands that (1) no interaction be selected for execution unless it has the required number of processes to participate in it (*interaction safety*) and (2) no process participate in more than one interaction at a time (*process safety*).

The interaction safety is obvious and can be derived from the protocol of I_{i_r} directly.

The process safety is ensured by Theorem 1.

Theorem 1 *Among the threads, $T_{i_1}^j, \dots, T_{i_k}^j$, started by P_j , only one can enter state '*success*'.*

Proof: Only the thread in state '*commit-sent*' can enter state '*success*' (rule 1.5). Furthermore, a thread can only enter state '*commit-sent*' either from state '*req-sent*' (rule 1.1) or state '*pending*' (1.3). The use of the shared bit map $a[1..k]$ in both rules guarantees that only one thread can be in state '*commit-sent*' at a time. □

The liveness requirement of the guard scheduling problem demands that there be no deadlock and no process or thread stay in a waiting state indefinitely.

After I_i receives all the l_i requests it needs and enters state '*all-met*', it will receive the same number (l_i) of *Commit()*, *Withdraw()* or *Abort()* (rules 1.1, 1.2, 1.3 and 1.4), provided that each thread $T_i^{j_r}$ involved is live and responds eventually.

Similarly, if every thread $T_{i_r}^j$ ($1 \leq r \leq k$) is live, thread M^j is also live. Therefore, the liveness of the entire algorithm hinges on the liveness of the protocol of $T_{i_r}^j$. The following lemma is used to prove the liveness of $T_{i_r}^j$.

Lemma 1 *If a thread $T_{i_r}^j$ is in state '*pending*' indefinitely, there must be another thread $T_{i_{r'}}^j$ of P_j such that $r < r'$ in state '*commit-sent*' indefinitely.*

Proof: According to rule 1.1, $a[r] = 1$ only if $T_{i_r}^j$ is in states '*commit-sent*' or '*pending*', but the first thread $T_{i_r}^j$ with $a[r] = 1$ must be in state '*commit-sent*'.

To simplify the notation, we rename $T_{i_r}^j$ to be $T_{i_r}^{j_r}$. Let us assume that $T_{i_r}^{j_r}$ stays in state '*pending*' indefinitely.

When thread $T_{i_r}^{j_r}$ enters state '*pending*', $a[(r+1)..k] \neq 0$ must be held. Let $a[u_1], \dots, a[u_v]$ ($r+1 \leq u_1 < \dots < u_v \leq k$) be all the bits that either are 1 when $T_{i_r}^{j_r}$ enters state '*pending*' or ever become 1 while $T_{i_r}^{j_r}$ is in state '*pending*' (indefinitely).

Thread $T_{i_{u_v}}^{j_r}$ must be in state '*commit-sent*' when $T_{i_r}^{j_r}$ enters state '*pending*'. Other threads $T_{i_{u_1}}^{j_r}, \dots, T_{i_{u_{v-1}}}^{j_r}$ must be in state '*pending*' first.

We want to prove that based on the assumption above at least one of $T_{i_{u_1}}^{j_r}, \dots, T_{i_{u_v}}^{j_r}$ must be in state '*commit-sent*' indefinitely.

Consider thread $T_{i_{u_v}}^{j_r}$ first. If it does not stay in state '*commit-sent*' indefinitely, it must receive a *Success()* or a *Fail* in finite time. If it receives a *Success()*, $T_{i_r}^{j_r}$ would leave state '*pending*' in finite time (rules 1.5 \Rightarrow 3.2 \Rightarrow 1.4). This would contradict the assumption above.

If it receives a *Fail*, thread $T_{i_{u_{v-1}}}^{j_r}$ will enter state '*commit-sent*' in finite time (rules 1.6 \Rightarrow 1.3).

The same procedure also applies to threads $T_{i_{u_{v-1}}}^{j_r}, \dots, T_{i_{u_1}}^{j_r}$. Therefore, if none of $T_{i_{u_1}}^{j_r}, \dots, T_{i_{u_v}}^{j_r}$ can stay in state '*commit-sent*' indefinitely, $T_{i_r}^{j_r}$ will leave state '*pending*' in finite time. This proves the lemma. \square

There are four waiting states in the protocol of $T_{i_r}^j$: '*req-sent*', '*commit-sent*', '*pending*' and '*re-try*'. The waiting of $T_{i_r}^j$ in state '*req-sent*' is to ensure the interaction safety and should not be considered as a problem for liveness.

$T_{i_r}^j$ in state '*re-try*' will enter state '*init*' after all the threads started by P_j send *Withdraw()*s to their interactions (rules 1.1, 1.4 \Rightarrow 3.1 \Rightarrow 1.8). Therefore, for the liveness of the protocol of $T_{i_r}^j$, we only need to prove that no thread $T_{i_r}^j$ will stay in states '*commit-sent*' or '*pending*' indefinitely. This is done in the following theorem.

Theorem 2 *It is impossible for any thread T_i^j in the system to stay in states '*commit-sent*' or '*pending*' indefinitely.*

Proof: According to Lemma 1, we only need to prove that it is impossible for any thread T_i^j to stay in state '*commit-sent*' indefinitely.

Let us assume that there is a thread $T_{i_1}^{j_1}$ staying in state '*commit-sent*' indefinitely. This means that $T_{i_1}^{j_1}$ receives neither *Success()* nor *Fail* in finite time. Therefore, none of the threads coordinating interaction I_{i_1} ever sends a *Withdraw()* or an *Abort()* to it (rules 2.2, 2.3). Furthermore, there is at least one of these threads that does not ever send a *Commit()* either. Let this thread be $T_{i_1}^{j_2}$. According to the protocol, $T_{i_1}^{j_2}$ must be in state '*pending*' indefinitely. From Lemma 1, there must be another thread $T_{i_2}^{j_2}$ from the same process P_{j_2} such that it stays in state '*commit-sent*' indefinitely and $i_1 < i_2$.

Continuing this way, we will have an infinite series

$$T_{i_1}^{j_1}, T_{i_1}^{j_2}, T_{i_2}^{j_2}, \dots, T_{i_{k-1}}^{j_k}, T_{i_k}^{j_k}, \dots$$

such that $T_{i_k}^{j_k}$ ($1 \leq k$) and $T_{i_{k-1}}^{j_k}$ ($2 \leq k$) are indefinitely in states '*commit-sent*' and '*pending*', respectively, and $i_1 < i_2 < \dots < i_k < \dots$. On the other hand, there are only a finite number (m) of interactions and we must have $i_1 < i_2 < \dots < i_k < \dots < m$. Therefore, the series above cannot be infinite. We have reached a contradiction. \square

The liveness of the system guarantees that an interaction process in state '*all-met*' will enter state '*meeting*' or state '*success*' in finite time. The progress requirement demands that at least one of those interactions in state '*all-met*' enter state '*success*'.

This requirement is satisfied in our algorithm. In order to prove this, we need the lemma as follows.

Lemma 2 *If thread $T_{i_r}^j$ sends a *Withdraw()* to I_{i_r} in state '*req-sent*' and enters state '*re-try*', there must be another thread $T_{i_{r'}}^j$ of P_j in state '*commit-sent*' such that $r' < r$.*

Proof: We prove this lemma by construction. Thread $T_{i_r}^j$ in state '*req-sent*' sends a *Withdraw()* to I_{i_r} only if it finds $a[1..(r-1)] \neq \mathbf{0}$ (rule 1.1). Let r' be the largest integer such that $r' < r$ and $a[r'] = 1$. According to the protocol, thread $T_{i_{r'}}^j$ is either the first thread in state '*commit-sent*' or a past pending thread which has been woken up by another thread and entered state '*commit-sent*' (rule 1.3). \square

The following theorem shows that in each coordination at least one selectable interaction will be selected. This ensures the progress of our algorithm.

Theorem 3 *Let I_{u_1}, \dots, I_{u_w} be the subset of all the interactions that enter state '*all-met*' after receiving all the requests they need. Then, at least one of them will enter state '*success*'.*

This theorem can be proved by contradiction and using Lemma 2. The technique of proof is similar to that used in proving Theorem 2. Due the limit of space of this paper we cannot include the proof here. The details of the proof can found in [7].

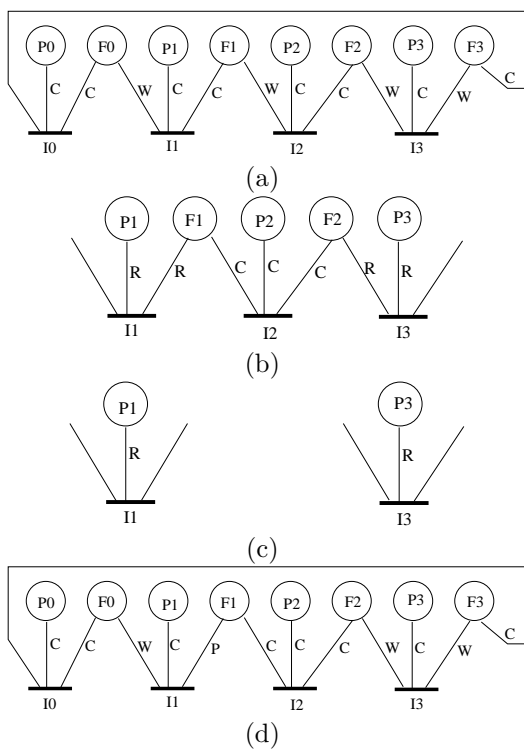


Figure 6: Progress of Nondeterministic Selection

5 Message Complexity

In our algorithm, I_{i_r} will re-try in the next round of coordination if it is not selected. Theorem 3 shows that at least one selectable interaction will be selected in each round of coordination. For a particular interaction selected eventually, the cost is obviously the number of rounds of coordination it has gone through times the number of messages required in each round of coordination.

According to the protocols of our algorithm, $4l_i$ messages between interaction I_i and its l_i participating processes are required in each round of coordination.

The average number of rounds of coordination required to select an interaction depends on many factors such as (1) the number of interactions connected through processes in conflict (processes ready to participate in more than one interactions) in the bipartite graph and (2) the average number of interactions in which a process participates.

The actual number of rounds is non-deterministic due to the asynchronous nature of the algorithm. Figure 6 shows two possible scenarios of selection of interactions I_0 and I_2 in our example. Edge labels C , W and R represent the situations where the process has just sent *Commit()*, *Withdraw()* and *Request()* to the interaction, respectively. Edge label P indicates that the process is in state 'pending' after it receives *All-Met* from the interaction. Figure 6(a) shows one possible situation where only interaction I_0 is about to be selected.

Figure 6(b) shows how I_2 is selected in the second round of coordination. Figure 6(c) shows the situation after I_2 is selected.

Let us go back to the situation shown in Figure 6(a) again. If the *All-Met* of I_2 reached thread $T_2^{F_1}$ of F_1 before the *All-Met* of I_1 reached thread $T_1^{F_1}$ of F_1 , we would have the situation shown in Figure 6(d). Both I_0 and I_2 would have been selected in the first round of coordination and the system would move to the situation shown in Figure 6(c) immediately.

The scenario shown above in Figure 6 (a), (b) and (c), is the worst case that can ever happen. This gives us the upper bound of the number of coordinations for an interaction to be selected: $\lfloor \frac{w}{2} \rfloor$, where w is the number of interactions connected through processes in conflict in the bipartite graph. This leads to the following theorem about the message complexity of our algorithm.

Theorem 4 *Given an l -party interaction I , the maximum number of interprocess messages required for I to be selected for execution is $4l \lfloor \frac{w}{2} \rfloor$, where w is the maximal number of interactions connected through processes in conflict in the bipartite graph of the problem.*

Here l is definitely much smaller than m in the complexity formula for Joung and Smolka's algorithm. Of course, w is application-dependent and we expect it to be also much smaller than m , particularly for real applications with large number of modules. The overhead of multi-threading used in this algorithm is negligible compared with the latency of message passing.

References

- [1] Yuh-Jzer Joung and Scott A. Smolka. Coordinating first-order multiparty interactions. *ACM Transactions on Programming Languages and Systems*, 16(3):954-985, May 1994.
- [2] Nissim Francez and Ira R Forman. *Interacting Processes - A Multiparty Approach to Coordinated Distributed Programming*. Addison-Wesley, 1996.
- [3] Peiyi Tang and Yoichi Muraoka. Parallel programming with interacting processes. In *Proceedings of the 12-th International Workshop on Languages and Compilers for Parallel Computing*, University of California at San Diego, USA, August 1999.
- [4] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1998.
- [5] C.A.R. Hoare. *Communication Sequential Processes*. Prentice Hall, 1985.
- [6] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman, Inc., 1996.
- [7] Peiyi Tang and Yoichi Muraoka. On-demand coordination of first-order multiparty interactions. Faculty of Sciences Working Paper Series (<http://www.sci.usq.edu.au/research/>) SC-MC-9902, Department of Mathematics and Computing, University of Southern Queensland, January 1999.