Detection of Orthogonal Interval Relations

Punit Chandra and Ajay D. Kshemkalyani

Dept. of Computer Science, Univ. of Illinois at Chicago
Chicago, IL 60607, USA
{pchandra,ajayk}@cs.uic.edu

Abstract. The complete set $\Re$ of orthogonal temporal interactions between pairs of intervals, formulated by Kshemkalyani, allows the detailed specification of the manner in which intervals can be related to one another in a distributed execution. This paper presents a distributed algorithm to detect whether pre-specified interaction types between intervals at different processes hold. Specifically, for each pair of processes i and j, given a relation $r_{i,j}$ from the set of orthogonal relations $\Re$, this paper presents a distributed (on-line) algorithm to determine the intervals, if they exist, one from each process, such that each relation $r_{i,j}$ is satisfied for that (i, j) process pair. The algorithm uses $O(n \min(np, 4mn))$ messages of size $O(n)$ each, where n is the number of processes, m is the maximum number of messages sent by any process, and p is the maximum number of intervals at any process. The average time complexity per process is $O(\min(np, 4mn))$, and the total space complexity across all the processes is $\min(4pn^2 - 2np, 10mn^2)$.

1 Introduction

Monitoring, synchronization and coordination, debugging, and industrial process control in a distributed system inherently identify local durations at processes when certain application-specific local predicates defined on local variables are true. To design and develop such applications to their fullest, we require a way to *specify* how durations at different processes are related to one another, and also a way to *detect* whether specified relationships hold in an execution. The formalism and axiom system formulated by Kshemkalyani [5] identified a complete orthogonal set $\Re$ of 40 fine-grained temporal interactions (or relationships) between intervals to *specify* how durations at different processes are related to one another. This gives flexibility and power to monitor, synchronize, and control distributed executions. Given a specific orthogonal relation that needs to hold between each pair of processes in a distributed execution, this paper presents a distributed (on-line) algorithm to *detect* the earliest intervals, one on each process, such that the specified relation between each pair of intervals is satisfied.

The pairwise interaction between processes is an important way of information exchange even in many large-scale distributed systems. Examples of such systems are sensor networks, ad-hoc mobile networks, mobile agent systems, and on-line collaborative motion planning and navigation systems. The various participating nodes can compute a dynamic global function (e.g., the classical distance-vector routing or AODV) or a dynamic local function such as the velocity of a mobile agent participating in a cooperative endeavor. Dynamic and on-line computation of local functions are used for centroidal Voronoi tessellations with applications to problems as diverse as image compression, quadrature, finite difference methods, distribution of resources, cellular biology, statistics, and the territorial behavior of animals [3].

To capture the pairwise interaction between processes, intervals at each process are identified to be the durations during which some application-specific local predicate is true. We introduce and address the following problem DOOR for the Detection of Orthogonal Relations.

Problem DOOR: Given a relation $r_{i,j}$ from $\Re$ for each pair of processes i and j, determine in an on-line, distributed manner the intervals, if they exist, one from each process, such that each relation $r_{i,j}$ is satisfied by the (i, j) pair.

The algorithm we propose uses $O(n \min(np, 4mn))$ messages of size $O(n)$ each, where n is the number of processes, m is the maximum number of messages sent by any process, and p is the maximum number of intervals at any process. The average time complexity per process is $O(\min(np, 4mn))$, and the total space complexity across all the processes is $\min(4pn^2 - 2np, 10mn^2)$.

A solution satisfying the set of relations $\{r_{i,j}(\forall i, j)\}$ identifies a global state of the system [2]. Note that a solution may exist in an execution only if the set of specified relations, one for each process pair, that need to hold, satisfies the axioms given in [5].

Section 2 gives the system model and preliminaries. Section 3 gives the theory used to determine when two given intervals at different processes can never be part of a solution set, and thus one of them can be discarded. Section 4 gives the data structures and local processing for tracking intervals at each process, and gives some tests used to determine the interaction type between a pair of intervals. Section 5 presents the distributed algorithm to solve problem **DOOR**. Section 6 gives concluding remarks.

2 System Model and Preliminaries

We assume an asynchronous distributed system in which n processes communicate by reliable message passing. Without loss of generality, we assume FIFO message delivery on the channels. A poset event structure model $(E, \prec)$, where $\prec$ is an irreflexive partial ordering representing the causality relation on the event set E, is used to model the distributed system execution. E is partitioned into local executions at each process. E_i is the linearly ordered set of events executed by process P_i. An event e executed by P_i is denoted e_i. The causality relation on E is the transitive closure of the local ordering relation on each E_i and the ordering imposed by message send events and message receive events [8]. This execution model is analogous to that in [5, 6, 9].

Table 1. Dependent relations for interactions between intervals are given in the first two columns [5]. Tests for the relations are given in the third column [7]

Relation r	Expression for $r(X, Y)$	Test for $r(X, Y)$
R1	$\forall x \in X \forall y \in Y, x \prec y$	$V_y^-[x] > V_x^+[x]$
R2	$\forall x \in X \exists y \in Y, x \prec y$	$V_y^+[x] > V_x^+[x]$
R3	$\exists x \in X \forall y \in Y, x \prec y$	$V_y^-[x] > V_x^-[x]$
R4	$\exists x \in X \exists y \in Y, x \prec y$	$V_y^+[x] > V_x^-[x]$
S1	$\exists x \in X \forall y \in Y, x \not\preceq y \bigwedge y \not\preceq x$	**if** $V_y^-[y] \not< V_x^-[y] \bigwedge V_y^+[x] \not> V_x^+[x]$ **then** $(\exists x^0 \in X$: $V_y^-[y] \not\leq V_x^{x^0}[y] \wedge V_x^{x^0}[x] \not\leq V_y^+[x])$ **else** $false$
S2	$\exists x_1, x_2 \in X \exists y \in Y, x_1 \prec y \prec x_2$	**if** $V_y^+[x] > V_x^-[x] \bigwedge V_y^-[y] < V_x^+[y]$ **then** $(\exists y^0 \in Y : V_x^+[y] \not< V_y^{y^0}[y] \wedge V_y^{y^0}[x] \not< V_x^-[x])$ **else** $false$

We assume vector clocks are available [4, 10]. The vector clock V has the property that $e \prec f \iff V(e) < V(f)$. The durations of interest at each process are the durations during which the local predicate is true. Such a duration, also termed as an interval, at process P_i is identified by the corresponding events within E_i.

Kshemkalyani showed in [5] that there are 29 or 40 possible mutually orthogonal ways in which any two durations can be related to each other, depending on whether the dense or the nondense time model is assumed. Informally speaking, with dense time, $\forall x, y$ in interval A, $x \prec y \implies \exists z \in A \mid x \prec z \prec y$. The orthogonal interaction types were identified by first using the six relations given in the first two columns of Table 1. Relations R1 (strong precedence), R2 (partially strong precedence), R3 (partially weak precedence), R4 (weak precedence) defined *causality conditions* whereas S1 and S2 defined *coupling conditions*.

Assuming that time is dense, it was shown in [5] that there are 29 possible interaction types between a pair of intervals, as given in the upper part of Table 2. The twenty-nine interaction types are specified using boolean vectors. The six relations R1-R4 and S1-S2 form a boolean vector of length 12, (six bits for $r(X, Y)$ and six bits for $r(Y, X)$). The nondense time model is significant because clocks which measure dense linear time use a nondense linear scale in practice. This model is also significant because actions at each node in a distributed system are a linear sequence of discrete events. This model permits 11 interaction types between a pair of intervals, defined in the lower part of Table 2, in addition to the 29 identified before. The interaction types are in pairs of inverses. For illustrations of these interactions and explanation of the table, the reader is requested to refer to [5]. The set of 40 relations is denoted as $\Re$.

Given a set of orthogonal relations, one between each pair of processes, that need to be detected, each of the 29 (40) possible independent relations in the dense (nondense) model of time can be tested for using the bit-patterns for the dependent relations, as given in Table 2. The tests for the relations $R1$, $R2$, $R3$,

Table 2. The 40 independent relations in $\Re$ [5]. The upper part of the table gives the 29 relations assuming dense time. The lower part of the table gives 11 additional relations if nondense time is assumed

Interaction Type	Relation $r(X,Y)$						Relation $r(Y,X)$					
	R1	R2	R3	R4	S1	S2	R1	R2	R3	R4	S1	S2
$IA(=IQ^{-1})$	1	1	1	1	0	0	0	0	0	0	0	0
$IB(=IR^{-1})$	0	1	1	1	0	0	0	0	0	0	0	0
$IC(=IV^{-1})$	0	0	1	1	1	0	0	0	0	0	0	0
$ID(=IX^{-1})$	0	0	1	1	1	1	0	1	0	1	0	0
$ID'(=IU^{-1})$	0	0	1	1	0	1	0	1	0	1	0	1
$IE(=IW^{-1})$	0	0	1	1	1	1	0	0	0	1	0	0
$IE'(=IT^{-1})$	0	0	1	1	0	1	0	0	0	1	0	1
$IF(=IS^{-1})$	0	1	1	1	0	1	0	0	0	1	0	1
$IG(=IG^{-1})$	0	0	0	0	1	0	0	0	0	0	1	0
$IH(=IK^{-1})$	0	0	0	1	1	0	0	0	0	0	1	0
$II(=IJ^{-1})$	0	1	0	1	0	0	0	0	0	0	1	0
$IL(=IO^{-1})$	0	0	0	1	1	1	0	1	0	1	0	0
$IL'(=IP^{-1})$	0	0	0	1	0	1	0	1	0	1	0	1
$IM(=IM^{-1})$	0	0	0	1	1	0	0	0	0	1	1	0
$IN(=IM'^{-1})$	0	0	0	1	1	1	0	0	0	1	0	0
$IN'(=IN'^{-1})$	0	0	0	1	0	1	0	0	0	1	0	1
$ID''(=(IUX)^{-1})$	0	0	1	1	0	1	0	1	0	1	0	0
$IE''(=(ITW)^{-1})$	0	0	1	1	0	1	0	0	0	1	0	0
$IL''(=(IOP)^{-1})$	0	0	0	1	0	1	0	1	0	1	0	0
$IM''(=(IMN)^{-1})$	0	0	0	1	0	0	0	0	0	1	1	0
$IN''(=(IMN')^{-1})$	0	0	0	1	0	1	0	0	0	1	0	0
$IMN''(=(IMN'')^{-1})$	0	0	0	1	0	0	0	0	0	1	0	0

$R4$, $S1$, and $S2$ in terms of vector timestamps are given in the third column of Table 1 [7]. V_i^- and V_i^+ denote the vector timestamp at process P_i at the start of an interval and at the end of an interval, respectively. V_i^x denotes the vector timestamp of event x_i at process P_i. The tests in Table 1 can be run by each process in a distributed manner. Each process P_i, $1 \leq i \leq n$, maintains information about the timestamps of the start and end of its local intervals, and certain other local information, in a local queue Q_i. The n processes collectively run a token-based algorithm to process the information in the local queues and solve problem **DOOR**.

We note some assumptions and terminology. (1) There are a maximum of p intervals at any process. (2) Interval X occurs at P_i and interval Y occurs at P_j. (3) For any two intervals X and X' that occur at the same process, if $R1(X, X')$, then we say that X is a predecessor of X' and X' is a successor of X.

1. When an internal event or send event occurs at process P_i, $V_i[i] = V_i[i] + 1$.
2. Every message contains the vector clock and *Interval Clock* of its send event.
3. When process P_i receives a message msg, then $\forall\ j$ do,

   if $(j == i)$ **then** $V_i[i] = V_i[i] + 1$,
   else $V_i[j] = \max(V_i[j], msg.V[j])$.

4. When an interval starts at P_i (local predicate ϕ_i becomes true), $I_i[i] = V_i[i]$.
5. When process P_i receives a message msg, then $\forall\ j$ do,
   $I_i[j] = \max(I_i[j], msg.I[j])$

Fig. 1. The vector clock V_i and *Interval Clock* I_i at process P_i

3 Conditions for Satisfying Given Interaction Types

A critical aspect of any distributed algorithm to solve problem **DOOR** is to design an efficient way to prune the intervals from the queues of the n processes. This section gives the condition for pruning an interval and the property which makes efficient pruning possible. We introduce the notion of prohibition function $S(r_{i,j})$ and relation $\lhd$ which give the condition for pruning of intervals. We also show that if the given relationship between a pair of intervals does not hold, then at least one of the intervals is deleted. This property makes the pruning and hence the algorithm efficient. Theorem 1 identifies this basic property.

For each $r_{i,j} \in \Re$, we define $S(r_{i,j})$ as the set of all relations R such that if $R(X, Y)$ is true, then $r_{i,j}(X, Y')$ can never be true for some successor Y' of Y. $S(r_{i,j})$ is the set of relations that prohibit $r_{i,j}$ from being true in the future.

Definition 1. Prohibition Function *$S : \Re \rightarrow 2^{\Re}$ is defined to be $S(r_{i,j}) = \{R \in \Re \mid R \neq r_{i,j} \wedge\ if\ R(X,Y)\ is\ true\ then\ r_{i,j}(X, Y')\ is\ false\ for\ all\ Y'\ that\ succeed\ Y\ \}$.*

Two relations R' and R'' in $\Re$ are related by $\lhd$ if the occurrence of $R'(X, Y)$ does not prohibit $R''(X, Y')$ for some successor Y' of Y.

Definition 2. *$\lhd$ is a relation on $\Re \times \Re$ such that $R' \lhd R''$ if (1) $R' \neq R''$, and (2) if $R'(X, Y)$ is true then $R''(X, Y')$ can be true for some Y' that succeeds Y.*

For example, $IC \lhd IB$ because (1) $IC \neq IB$ and, (2) if $IC(X, Y)$ is true, then there is a possibility that $IB(X, Y')$ is also true, where Y' succeeds Y.

Theorem 1. *For $R', R'' \in \Re$, if $R' \lhd R''$ then $R'^{-1} \not\lhd R''^{-1}$. (Proof is in [1].)*

Taking the same example, $IC \lhd IB \Rightarrow IV(= IC^{-1}) \not\lhd IR(= IB^{-1})$, which is indeed true. Note that $R' \neq R''$ in the definition of relation $\lhd$ is necessary; otherwise $R' \lhd R'$ leads to $R'^{-1} \not\lhd R'^{-1}$, a contradiction.

Lemma 1. *If $R \in S(r_{i,j})$ then $R \not\lhd r_{i,j}$ else if ($R \notin S(r_{i,j})$ and $R \neq r_{i,j}$) then $R \lhd r_{i,j}$.*

```
type Event_Interval = record
    interval_id : integer;
    local_event: integer;
end

type Log = record
    start: array[1..n] of integer;
    end: array[1..n] of integer;
    p_log: array[1..n] of Process_Log;
end

type Process_Log = record
    event_interval_queue: queue of Event_Interval;
end
```

Fig. 2. The *Event_Interval*, *Log*, and *Process_Log* data structures at P_i ($1 \leq i \leq n$)

Proof. If $R \in S(r_{i,j})$, using Definition 1, it can be inferred that $r_{i,j}$ is false for all Y' that succeed Y. This does not satisfy the second part of Definition 2. Hence $R \not\lhd r_{i,j}$. If $R \notin S(r_{i,j})$ and $R \neq r_{i,j}$, it follows that $r_{i,j}$ can be true for some Y' that succeeds Y. This satisfies Definition 2 and hence $R \lhd r_{i,j}$. □

$S(r_{i,j})$ for each of the interaction types in $\Re$ is given in [1]. The following lemmas are used to show the correctness of the algorithm in Figure 6.

Lemma 2. *If the relationship $R(X, Y)$ between intervals X and Y (belonging to processes P_i and P_j, resp.) is contained in the set $S(r_{i,j})$, then interval X can be removed from the queue Q_i.*

Proof. From the definition of $S(r_{i,j})$, we get that $r_{i,j}(X, Y')$ cannot exist, where Y' is any successor interval of Y. Hence interval X can never be a part of the solution and can be deleted from the queue. □

Lemma 3. *If the relationship between a pair of intervals X and Y (belonging to processes P_i and P_j respectively) is not equal to $r_{i,j}$, then interval X or interval Y is removed from its queue Q_i or Q_j, respectively.*

Proof. We use contradiction. Assume relation $R(X, Y)$ ($\neq r_{i,j}(X, Y)$) is true for intervals X and Y. From Lemma 2, the only time neither X nor Y will be deleted is when $R \notin S(r_{i,j})$ and $R^{-1} \notin S(r_{j,i})$. From Lemma 1, it can be inferred that $R \lhd r_{i,j}$ and $R^{-1} \lhd r_{j,i}$. As $r_{i,j}^{-1} = r_{j,i}$, we get $R \lhd r_{i,j}$ and $R^{-1} \lhd r_{i,j}^{-1}$. This is a contradiction as by Theorem 1, $R \lhd r_{i,j} \Rightarrow R^{-1} \not\lhd r_{i,j}^{-1}$. Hence $R \in S(r_{i,j})$ or $R^{-1} \in S(r_{j,i})$, and thus at least one of the intervals gets deleted. □

4 Tracking Intervals and Evaluating Relations

This section gives the operations and data structures to track intervals at each process. These are used by our algorithm given in the next section.

Each process P_i, where $1 \leq i \leq n$, maintains the following data structures. (1) V_i : array[1..n] of integer. This is the *Vector Clock* [4, 10]. (2) I_i : array[1..n] of integer. This is the *Interval Clock* which tracks the latest intervals at processes.

```
Start of an interval:
    $Log_i.start = V_i^-$.  //Store the timestamp $V_i^-$ of the starting of the interval.
On receiving a message during an interval:          //Store the local component of
    if (change in $I_i$) then              //vector clock and $interval\_id$ which caused
        for each $k$ such that $I_i[k]$ was changed                    //the change in $I_i$
            insert $(I_i[k], V_i[i])$ in $Log_i.p\_log[k].event\_interval\_queue$
End of interval:
    $Log_i.end = V_i^+$          //Store the timestamp $V_i^+$ of the end of the interval.
    if (a receive or send occurs between start of previous interval and end of
    present interval) then
        Enqueue $Log_i$ on to the local queue $Q_i$.
```

Fig. 3. The scheme for constructing *Log* at P_i $(1 \leq i \leq n)$

$I_i[j]$ is the timestamp $V_j[j]$ when ϕ_j last became true, as known to P_i. (3) Log_i: contains the information about an interval, needed to compare it with other intervals. Figure 1 shows how to update the vector clock and *Interval Clock*.

To maintain Log_i, the data structures are defined in Figure 2. The *Log* consists of vector timestamps *start* and *end* for the start and end of an interval, respectively. It also contains an array of *Process_Log*, where each *Process_Log* is a queue of type *Event_Interval*. *Event_Interval* consists of a tuple composed of *interval_id* and *local_event*. Let *local_event* be the local component of the clock value of a receive event at which the k^{th} component of *Interval Clock* gets updated — then the tuple composed of the *local_event* and the k^{th} component of *Interval Clock* is added into the *Process_Log* queue which forms the k^{th} component of *p_log*. Log_i is constructed and stored on the local queue Q_i using the protocol shown in Figure 3. Note that not all the intervals are stored in the local queue. The *Log* corresponding to an interval is stored only if the relationship between the interval and all other intervals (at other processes) is different from the relationship which its predecessor interval had with all the other intervals (at other processes). In other words, if two or more successive intervals on the same process have the same relationship with all other intervals, then *Log* corresponding to only one of them needs to be stored on the queue. Two successive intervals Y and Y' on process P_j will have the same relationship if no message is sent or received by P_j between the start of Y and the end of Y'.

The *Log* is used to determine the relationship between two intervals. The tests in Table 1 are used to find which of $R1$, $R2$, $R3$, $R4$, $S1$, and $S2$ are true. Figure 4 shows how to implement the tests for $S1(Y, X)$ and $S2(X, Y)$ using the *Log* data structure.

$S2(X,Y)$:

```
1. // Eliminate from $Log$ of interval $Y$ (on $P_j$), all receives of messages
   //which were sent by $i$ before the start of interval $X$ (on $P_i$).
   (1a) for each $event\_interval \in Log_j.p\_log[i].event\_interval\_queue$
   (1b)     if ($event\_interval.interval\_id < Log_i.start[i]$) then
   (1c)         remove $event\_interval$

2. // Select from the pruned $Log$, the earliest message sent from $X$ to $Y$.
   (2a) $temp = \infty$
   (2b) if ($Log_j.start[i] \geq Log_i.start[i]$) then $temp = Log_j.start[j]$
   (2c) else
   (2d)     for each $event\_interval \in Log_j.p\_log[i].event\_interval\_queue$
   (2e)         $temp = \min(temp, event\_interval.local\_event)$

3. if ($Log_i.end[j] \geq temp$) then $S2(X,Y)$ is true.
```

$S1(Y,X)$:

```
1. Same as step 1 of scheme to determine $S2(X,Y)$.
2. Same as step 2 of scheme to determine $S2(X,Y)$.
3. if ($Log_i.end[j] < temp$) and ($temp > Log_j.start[j]$) then $S1(Y,X)$ is true.
```

Fig. 4. Implementing the tests for $S1(X,Y)$ and $S2(Y,X)$

5 A Distributed Algorithm

5.1 Algorithm DOOR

To solve problem **DOOR**, defined in Section 1, recall that the given relations $\{r_{i,j} \mid 1 \leq i,j \leq n \text{ and } r_{i,j} \in \Re\}$ need to satisfy the axioms on $\Re$, given in [5]. Thus, it is possible for a solution to exist in some execution.

Algorithm Overview: The algorithm uses a token-based approach. Intuitively, the process P_i which has the token triggers at each other process P_j, the comparison of the interval at the head of P_i's queue with the interval at the head of P_j's queue. The comparison may result in either the interval at the head of P_i's queue or P_j's queue being deleted – the corresponding queue index gets inserted in *updatedQueues*, which is a part of the token at P_i. Once such a comparison is done with all other process queues, the token is then sent to some P_j whose index j is in *updatedQueues*. This allows comparison of the new interval at the head of P_j's queue with the interval at the head of the queue of each other process. A solution is found when *updatedQueue* becomes empty.

The Algorithm: Besides the token (T), two kinds of messages are exchanged between processes – $REQUEST$ (REQ) and $REPLY$ (REP). The data structures are given in Figure 5. The proposed algorithm is given in Figure 6. Each procedure is executed atomically. The token is used such that only the token-holder process can send REQs and receive REPs. The process (P_i) having the

```
type $REQ$ = message
    $log$ : $Log$;                         //Contains the $Log$ of the interval at the queue
end                                        //head of the process sending the $REQ$
type $REP$ = message
    $updated$: set of integer;       //Contains the indices of queues updated after a test
end
type $T$ = token
    $updatedQueues$: set of integer;     //Contains the indices of all the updated queues
end
```

Fig. 5. The *REQ*, *REP*, and *Token* data structures

token broadcasts a *REQ* to all other processes (line 3b). The *Log* corresponding to the interval at the head of the queue Q_i is piggybacked on the *REQ* (line 3a). On receiving a *REQ* from P_i, each process P_j compares the piggybacked interval X with the interval Y at the head of its queue Q_j (line 4d). According to Lemma 3, the comparison between intervals on processes P_i and P_j can result in three cases. (1) $r_{i,j}$ is satisfied. (2) $r_{i,j}$ is not satisfied and interval X can be removed from the queue Q_i. (3) $r_{i,j}$ is not satisfied and interval Y can be removed from the queue Q_j. In the third case, the interval at the head of Q_j is dequeued and process index j is stored in *REP.updated* (lines 4g, 4h). In the second case, the process index i is stored in *REP.updated* (line 4e). Note that both cases (2) and (3) can occur after a comparison. P_j then sends *REP* to P_i. The *REP* carries the indices of the queues, if any, which got updated after the comparison. Once process P_i receives a *REP* from all other processes, it stores the indices of all the updated queues in set *updatedQueues* which is a part of the token currently at P_i. Process P_i then checks if its index i is contained in *updatedQueues*. If so, it deletes the interval at the head of Q_i (line 8f). A solution is detected when *updatedQueues* becomes empty. If *updatedQueues* is non-empty, then the token is sent to a randomly selected process from *updatedQueues* (line 8g).

5.2 Complexity Analysis

The complexity is analyzed in terms of the maximum number of messages sent per process (m) and the maximum number of intervals per process (p).

- Space overhead:
  - Worst case space overhead per process is $\min(4np-2p, 4mn^2+2mn-2m)$.
  - Total space overhead across all processes is $\min(4pn^2 - 2np, 10mn^2)$.
- Average time complexity (per process) is $O(\min(np, 4mn))$.
- Total number of messages sent is $O(n \min(np, 4mn))$. Total message space overhead is $O(n^2 \min(np, 4mn))$.

The details of the complexity analysis are given in [1].

Note that in case of broadcast media, the number of *REQ*s sent for each *Log* is one because *REQ*s are broadcast by sending one message (line 3b). The message space complexity reduces to $O(n \min(np, 4mn))$, although the total number of messages sent stays at $O(n \min(np, 4mn))$.

```
(1) Initial state for process $P_i$
(1a)    $Q_i$ is empty

(2) Initial state of the token
(2a)    $T.updatedQueues = \{1, 2...n\}$
(2b)    A randomly elected process $P_i$ holds the token.

(3) $SendReq$ : Procedure called by process $P_i$ to send $REQ$ message
(3a)    $REQ.log = Log_i$ at the head of the local queue
(3b)    Broadcast request $REQ$ to all processes

(4) $SendReply$ : Procedure called by process $P_j$ to send a $REP$ message to $P_i$
(4a)    $REP.updated = \phi$
(4b)    $Y$=head of local queue $Q_j$
(4c)    $X$=$REQ.log$
(4d)    Determine $R(X, Y)$ using the tests given in Table 1 and Figure 4
(4e)    if $(R(X, Y) \in S(r_{i,j}))$ then $REP.updated = REP.updated \cup \{i\}$
(4f)    if $(R(Y, X) \in S(r_{j,i}))$ then
(4g)        $REP.updated = REP.updated \cup \{j\}$
(4h)        Dequeue $Y$ from local queue $Q_j$
(4i)    Send reply $REP$ to $P_i$

(5) $RcvToken$ : On receiving a token $T$ at $P_i$
(5a)    Remove index $i$ from $T.updatedQueues$
(5b)    if ($Q_i$ is nonempty) then $SendReq$

(6) $IntQue$ : When an interval gets queued on $Q_i$ at $P_i$
(6a)    if (number of elements in queue $Q_i$ is 1) and ($P_i$ has the token) then $SendReq$
(6b)    else
(6c)        if (number of elements in queue $Q_i$ is 1) and ($P_i$ has a pending request) then
(6d)            $REQ$ is not pending
(6e)            $SendReply$

(7) $RcvReq$ : On receiving a $REQ$ at $P_i$
(7a)    if ($Q_i$ is nonempty) then $SendReply$
(7b)    else $REQ$ is pending

(8) $RcvReply$ : On receiving a reply from $P_i$
(8a)    $T.updatedQueues = T.updatedQueues \cup REP.updated$
(8b)    if (reply received from all processes) then
(8c)        if ($T.updatedQueues$ is empty) then
(8d)            Solution detected. Heads of the queues identify the intervals.
(8e)        else
(8f)            if $(i \in T.updatedQueues)$ then dequeue the head from $Q_i$
(8g)            Send token to $P_k$ where $k$ is randomly selected from $T.updatedQueues$
```

Fig. 6. Distributed algorithm to solve problem **DOOR**

5.3 Optimizations

The following two modifications to the algorithm increase the pruning and decrease the number of messages sent. However, the order of space, time, and message complexities remains the same for both modifications.

– When procedure *SendReply* is executed by P_j in response to P_i's REQ, instead of comparing the interval at the head of Q_j with P_i's interval (piggybacked on REQ), multiple comparisons can be done. Each time the comparison results in the interval at the head of Q_j being deleted, the next interval on Q_j is compared with the piggybacked interval. A REP is sent back only when either the comparison results in a relation equal to $r_{i,j}$ or the relationship is such that the interval at the head of Q_i (the piggybacked interval) has to be deleted. Thus each REQ can result in multiple intervals being pruned from the queue of the process receiving the REQ.

– In procedure *RcvReply* (lines 8f-8g), if $T.updatedQueues$ contains the index i, it means the interval at the head of queue Q_i needs to be deleted and that the token will be sent to process P_i again in the future. Hence, if index i is contained in $T.updatedQueues$, not only is the interval at the head of Q_i deleted but also the next token-holder is selected as P_i. This saves the extra message required to resend the token to P_i later.

6 Concluding Remarks

Pairwise temporal interactions in a distributed execution provide a valuable way to specify and model synchronization conditions and information interchange. This paper presented an on-line distributed algorithm to detect whether there exists a set of intervals, one at each process, such that a given set of pairwise temporal interactions, one for each process pair, holds for the set of intervals identified. Future work can be to explore how the orthogonal interaction types can formalize and simplify the exchange patterns for various applications.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9875617.

References

[1] P. Chandra, A. D. Kshemkalyani, Detection of orthogonal interval relations, *Tech. Report UIC-ECE-02-06*, Univ. of Illinois at Chicago, May 2002. 327, 328, 331

[2] K. M. Chandy, L. Lamport, Distributed snapshots: Determining global states of distributed systems, *ACM Transactions on Computer Systems*, 3(1): 63-75, 1985. 324

[3] Q. Du, V. Faber, M. Gunzburger, Centroidal Voronoi tessellations: applications and algorithms, *SIAM Review*, 41(4): 637-676, 1999. 324

[4] C. J. Fidge, Timestamps in message-passing systems that preserve partial ordering, *Australian Computer Science Communications*, 10(1): 56-66, February 1988. 325, 328

[5] A. D. Kshemkalyani, Temporal interactions of intervals in distributed systems, *Journal of Computer and System Sciences*, 52(2): 287-298, April 1996. 323, 324, 325, 326, 330

[6] A. D. Kshemkalyani, A framework for viewing atomic events in distributed computations, *Theoretical Computer Science*, 196(1-2), 45-70, April 1998. 324

[7] A. D. Kshemkalyani, A fine-grained modality classification for global predicates, *Tech. Report UIC-EECS-00-10*, Univ. of Illinois at Chicago, 2000. 325, 326

[8] L. Lamport, Time, clocks, and the ordering of events in a distributed system, *Communications of the ACM*, 558-565, 21(7), July 1978. 324

[9] L. Lamport, On interprocess communication, Part I: Basic formalism; Part II: Algorithms, *Distributed Computing*, 1:77-85 and 1:86-101, 1986. 324

[10] F. Mattern, Virtual time and global states of distributed systems, *Parallel and Distributed Algorithms*, North-Holland, 215-226, 1989. 325, 328