Global Predicate Detection under Fine-Grained Modalities

Punit Chandra and Ajay D. Kshemkalyani

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

Abstract. Predicate detection is an important problem in distributed systems. Based on the temporal interactions of intervals, there exists a rich class of modalities under which global predicates can be specified. For a conjunctive predicate ϕ, we show how to detect the traditional $Possibly(\phi)$ and $Definitely(\phi)$ modalities along with the added information of the exact interaction type between each pair of intervals (one interval at each process). The polynomial time, space, and message complexities of the proposed on-line detection algorithms to detect *Possibly* and *Definitely* in terms of the fine-grained interaction types per pair of processes, are the *same as* those of the earlier on-line algorithms that can detect *only* whether the *Possibly* and *Definitely* modalities hold.

1 Introduction

Predicate detection in a distributed system is useful in many contexts such as monitoring, synchronization and coordination, debugging, and industrial process control [2,4,6,7,8,14,16,17]. Marzullo et al. defined two modalities under which predicates can hold for a distributed execution [4,14].

- $Possibly(\phi)$: There exists a consistent observation of the execution such that ϕ holds in a global state of the observation.
- $Definitely(\phi)$: For every consistent observation of the execution, there exists a global state of it in which ϕ holds.

The formalism and axiom system given in [9] identified an orthogonal set $\Re$ of 40 fine-grained temporal interactions between a pair of intervals in a distributed execution. It was shown in [10] that this formalism provides much more expressive power than the *Possibly* and *Definitely* modalities, and a mapping from $\Re$ to the *Possibly* and $Definitely$ modalities was given. A *conjunctive* predicate is of the form $\bigwedge_i \phi_i$, where ϕ_i is any predicate defined on variables local to process P_i. We show that for a conjunctive predicate ϕ (e.g., $x_i = 2 \wedge y_j > 8$), $Possibly(\phi)$ and $Definitely(\phi)$ can be detected along with the added information of the exact interaction type between each pair of intervals, one interval at each process. This provides flexibility and power to monitor, synchronize, and control distributed executions. The time, space, and message complexities of the

V.A. Saraswat (Ed.): ASIAN 2003, LNCS 2896, pp. 91–109, 2003.
© Springer-Verlag Berlin Heidelberg 2003

Table 1. Comparison of space, message and time complexities. n = number of processes, M = maximum queue length at P_0, p = maximum number of intervals occurring at any process, m_s = total number of messages exchanged between all the processes. Note: $p \geq M$, as all the intervals may not be sent to P_0.

	Avg. time complexity at P_0	Total number of messages	Space at P_0 (= total msg. space)	Avg. space at P_i, $i \in [1, n]$
GW94 [6] (Possibly)	$O(n^2M)$ or $O(nm_s)$	$O(m_s)$	$O(n^2M)$ or $O(nm_s)$	$O(n)$
GW96 [7] (Definitely)	$O(n^2M)$ or $O(nm_s)$	$O(m_s)$	$O(n^2M)$ or $O(nm_s)$	$O(n)$
$Fine_Poss$, $Fine_Def$, $Fine_Rel$	$O(n^2M)$ or $O(n[min(4m_s, np)])$	$O(min(4m_s, np))$	$O(min[(4n-2)np, 10nm_s])$	$O(n)$

proposed on-line, centralized detection algorithms (Algorithms $Fine_Poss$ and $Fine_Def$ - the main results) to detect $Possibly$ and $Definitely$ in terms of the fine-grained modalities per pair of processes, are the *same as* those of the earlier on-line, centralized algorithms [6,7] that can detect *only* whether the $Possibly$ and $Definitely$ modalities hold. Table 1 compares the complexities. $Fine_Rel$, which is an intermediate problem we need solve, is introduced later.

The power of our approach stems from the use of intervals as opposed to individual events in the distributed execution. The intervals at each process are identified to be the durations during which the local predicate is true [10,12]. We now state Problems $Fine_Poss$ and $Fine_Def$.

Problem $Fine_Poss$ **Statement.** For a conjunctive predicate ϕ, determine on-line if $Possibly(\phi)$ is true. If true, identify the fine-grained pairwise interaction between each pair of processes when $Possibly(\phi)$ first becomes true.

Problem $Fine_Def$ **Statement.** For a conjunctive predicate ϕ, determine on-line if $Definitely(\phi)$ is true. If true, identify the fine-grained pairwise interaction between each pair of processes when $Definitely(\phi)$ first becomes true.

Section 2 gives the background and objectives. Section 3 presents the framework and data structures. Section 4 and Section 5 present the on-line algorithms. Section 6 gives the conclusions.

2 System Model, Background, and Objectives

2.1 System Model

We assume an asynchronous distributed system in which n processes communicate only by reliable message passing. We do not assume FIFO channels. To model the system execution, let $\prec$ be an irreflexive partial ordering representing the causality relation on the event set E. E is partitioned into local executions at each process. Let N denote the set of all processes. Each E_i is a linearly ordered set of events executed by process P_i. An event e at P_i is denoted e_i. The causality

Table 2. Dependent relations for interactions between intervals are given in the first two columns [9]. Tests for the relations are given in the third column [10].

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** $false$

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 [13]. A *cut* C is a subset of E such that if $e_i \in C$ then $(\forall e_i')e_i' \prec e_i \implies e_i' \in C$. A *consistent cut* is a downward-closed subset of E in $(E, \prec)$ and denotes an execution prefix. For event e, there are two special consistent cuts $\downarrow e$ and $e \uparrow$. $\downarrow e$ is the maximal set of events that happen before e. $e \uparrow$ is the set of all events up to and including the earliest events at each process for which e happens before the events.

Definition 1. *Cut* $\downarrow e$ *is defined to be* $\{e' \mid e' \prec e\}$ *and cut* $e \uparrow$ *is defined to be* $\{e' \mid e' \not\succeq e\} \bigcup \{e_i, i = 1, \ldots, |N| \mid e_i \succeq e \bigwedge (\forall e_i' \prec e_i, e_i' \not\succeq e)\}$.

The system state after the events in a cut is a global state; if the cut is consistent, the corresponding system state is a consistent global state. We assume that the popular vector clocks are available [5,15] – the vector clock V has the property that $e \prec f \iff V(e) < V(f)$.

A conjunctive predicate is of the form $\bigwedge_i \phi_i$, where ϕ_i is a predicate defined on variables local to process P_i. The intervals of interest at each process are the durations in which the local predicate is true. Such an interval at process P_i is identified by the (totally ordered) subset of adjacent events of E_i for which the local predicate is true.

2.2 Pairwise Interactions

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 [9]. Informally speaking, with dense time, $\forall x, y$ in interval A, $x \prec y \implies \exists z \in A \mid x \prec z \prec y$. These orthogonal interaction types were identified by first using the six relations given in the first two columns of Table 2. Relations R1 (strong precedence), R2 (partially strong precedence), R3 (partially weak precedence), R4 (weak precedence) define *causality conditions*

Table 3. The 40 independent relations in $\Re$ [9]. X and Y are intervals. 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

whereas S1 and S2 define *coupling conditions*. Assuming that time is dense, it was shown in [9] that there are 29 possible interaction types between a pair of intervals, as given in the upper part of Table 3. Of the 29 interactions, there are 13 pairs of inverses, while three are inverses of themselves. 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 interaction types are illustrated in [9]. The nondense time model, whose importance is given in [9], permits 11 interaction types between a pair of intervals, defined in the lower part of Table 3, besides the 29 identified before. Of these, there are five pairs of inverses, while one is its own inverse. These interaction types are illustrated in [9]. The set of 40 relations is denoted as $\Re$.

2.3 Modalities for Global Predicates

Observe that for any predicate ϕ, three orthogonal relational possibilities hold under the *Possibly*/ *Definitely* classification: (i) $Definitely(\phi)$, (ii) $\neg Definitely(\phi) \wedge Possibly(\phi)$, (iii) $\neg Possibly(\phi)$.

Table 4. Refinement mapping [10]. The upper part shows the 29 mappings when the dense time model is assumed. With the nondense time model, the 11 additional mappings in the lower part also apply.

$Definitely(\phi)$	$Possibly(\phi) \wedge \neg Definitely(\phi)$	$\neg Possibly(\phi)$
ID and IX	IB and IR	IA and IQ
ID′ and IU	IC and IV	
IE and IW	IG	
IE′ and IT	IH and IK	
IF and IS	II and IJ	
IO and IL		
IP and IL′		
IM		
IM′ and IN		
IN′		
ID″ and IUX	IM″ and IMN	
IE″ and ITW	IMN″	
IL″ and IOP		
IN″ and IMN′		

Conjunctive predicates form an important class of predicates and have been studied extensively [2,6,7,8]. Based on the definitions of the orthogonal temporal interactions [9], the 3 orthogonal relational possibilities based on the *Possibly*/*Definitely* definitions were refined into the exhaustive set of 40 possibilities [10]. Table 4 shows this refinement mapping, assuming that the conjunctive predicate is defined on two processes. When conjunctive predicate ϕ is defined over variables that are local to $n > 2$ processes, one can still express the three possibilities (i) $Definitely(\phi)$, (ii) $\neg Definitely(\phi) \wedge Possibly(\phi)$, and (iii) $\neg Possibly(\phi)$, in terms of the fine-grained 40 independent relations between C_2^n pairs of intervals. Note that not all $40^{C_2^n}$ combinations will be valid – the combinations have to satisfy the axiom system given in [9].

For $n > 2$ processes, the refinement mappings of the $Possibly$ and $Definitely$ modalities are given by Theorem 1 [10].

Theorem 1. *[10] Consider a conjunctive predicate $\phi = \wedge_i \phi_i$. The following results are implicitly qualified over a set of intervals, containing one interval per process.*

- $Definitely(\phi)$ *holds if and only if* $\bigwedge_{(\forall i \in N)(\forall j \in N)} [Definitely(\phi_i \wedge \phi_j)]$
- $\neg Definitely(\phi) \wedge Possibly(\phi)$ *holds if and only if*
  - $(\exists i \in N)(\exists j \in N) \neg Definitely(\phi_i \wedge \phi_j) \bigwedge (\bigwedge_{(\forall i \in N)(\forall j \in N)} [Possibly(\phi_i \wedge \phi_j)])$
- $\neg Possibly(\phi)$ *holds if and only if* $(\exists i \in N)(\exists j \in N) \neg Possibly(\phi_i \wedge \phi_j)$

Consider the following example (from [10]) of the extra information provided by the fine-grained modalities. Let ϕ be $a_i = 2 \wedge b_j = 3 \wedge c_k = 5$. Let a_i, b_j, and c_k be 2, 3, 5 respectively, in intervals X_i, Y_j, and Z_k, respectively, and

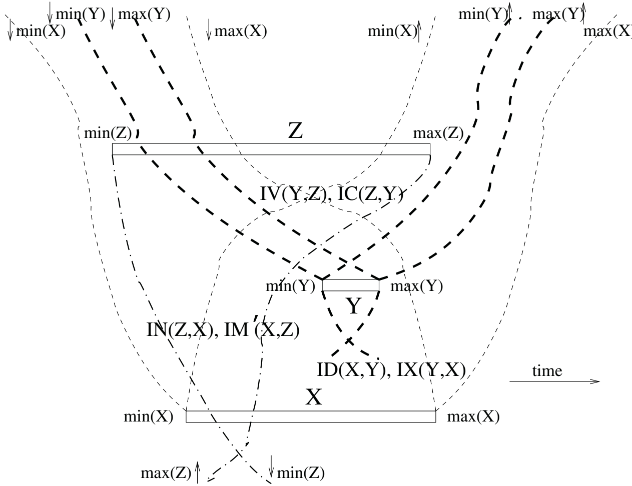

Fig. 1. Example [10] to show fine-grained relations across $n > 2$ processes.

let $ID(X_i, Y_j)$, $IV(Y_j, Z_k)$, and $IN(Z_k, X_i)$ be true. This is shown in Figure 1. Then by Theorem 1, we have (i) $Definitely(a_i = 2 \wedge b_j = 3)$, (ii) $Possibly(b_j = 3 \wedge c_k = 5)$ and $\neg Definitely(b_j = 3 \wedge c_k = 5)$, and (iii) $Definitely(a_i = 2 \wedge c_k = 5)$. By Theorem 1, we have the modality $Possibly(\phi) \wedge \neg Definitely(\phi)$. Conversely, if $Possibly(\phi) \wedge \neg Definitely(\phi)$ is known in the classical course-grained classification, the fine-grained classification gives the added information: $ID(X_i, Y_j)$, $IV(Y_j, Z_k)$, and $IN(Z_k, X_i)$.

2.4 Objective

Our objective is to solve $Fine_Poss$ and $Fine_Def$, i.e., to detect $Possibly(\phi)$ and $Definitely(\phi)$, for conjunctive predicates, with the added information of the exact interaction type between each pair of intervals (one at each process) when $Possibly(\phi)$ and $Definitely(\phi)$ are true. The extra information about the pairwise interaction type is useful, as shown in [10] by considering various applications. Another use of the extra information is in multi-player distributed games. The overheads of our algorithms are the same as those of the earlier algorithms, [GW94] [6] for $Possibly(\phi)$ and [GW96] [7] for $Definitely(\phi)$, that can detect **only** whether $Possibly(\phi)$ is true and whether $Definitely(\phi)$ is true. Tables 1 compares all the performance metrics.

3 Detecting Predicates: Framework and Data Structures

Given a conjunctive predicate, for each pair of intervals belonging to different processes, each of the 29 (40) possible independent relations in the dense (non-dense) model of time can be tested for using the bit-patterns for the dependent

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. 2. The vector clock and *Interval Clock*.

relations, as given in Table 3. The tests for the relations $R1$, $R2$, $R3$, $R4$, $S1$, and $S2$ are given in the third column of Table 2 using vector timestamps. Recall that the interval at a process is used to identify the period when some local property (using which the predicate ϕ is defined) holds. V_i^- and V_i^+ denote the vector timestamps at process P_i at the start of an interval and the end of an interval, respectively. V_i^x denotes the vector timestamp of event x_i at P_i.

The tests in Table 2 can be run by a central server along the lines of the algorithms in [4,6,7,8,14]. Processes P_1, P_2, ..., P_n send the timestamps of their intervals and certain other local information to the server P_0, which maintains queues Q_1, Q_2, ... ,Q_n for each of the processes. We require that the central server P_0 receive the updates from each P_i, $1 \leq i \leq n$, in FIFO order. For each of the problems to be solved, the server runs different algorithms to process the interval information in the queues. We assume that interval X occurs at P_i and interval Y occurs at P_j. 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. Also, there are a maximum of p intervals at any process.

The operations and data structures required by the algorithms to solve Problems $Fine_Poss$ and $Fine_Def$ can be divided into two parts. The first, common to all the algorithms, runs on each of the n processes P_1 to P_n, and is given in this section. The second part of each algorithm runs on the central process P_0 and is presented in later sections.

3.1 Log Operations

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 $Vector\ Clock$. (2) I_i : array[1..n] of integer. This is a *Interval Clock* which tracks the latest intervals at processes. $I_i[j]$ is the timestamp $V_j[j]$ when ϕ_j last became true. (3) Log_i: Contains the information to be sent to the central process. Figure 2 shows how to maintain the vector clock and *Interval Clock*.

To maintain Log_i, the required data structures and operations are defined in Figure 3. Log_i is constructed and sent to the central process using the protocol shown. The central process uses the Log to determine the relationship between two intervals.

```
type Event_Interval = record                type Log = record
    interval_id : integer;                      start: array[1..n] of integer;
    local_event: integer;                       end: array[1..n] of integer;
end                                             p_log: array[1..n] of Process_Log;
                                            end
type Process_Log = record
    event_interval_queue: queue of Event_Interval;
end

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:
    if (change in I_i) then     //Store local component of vector clock and interval_id
        for each k such that I_i[k] was changed      //which caused 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 event occurs between start of previous interval and end of
    current interval) then
        Send Log_i to central process.
```

Fig. 3. Data structures and operations to construct *Log* at P_i $(1 \leq i \leq n)$.

3.2 Complexity Analysis at P_i $(1 \leq i \leq n)$

Space complexity of *Log*. Each *Log* stores the start (V^-) and the end (V^+) of an interval, which requires a maximum of $2np$ integers per process. Consider the construction algorithm for *Log*. Besides the start and the end of each interval, an *Event_Interval* is added to the *Log* for every component of *Interval Clock* which is modified due to the receive of a message. As a change in a component of *Interval Clock* implies the start of a new interval on another process, the total number of times the component of *Interval Clock* can change is equal to the number of intervals on all the processes. Thus the total number of *Event_Interval* which can be added to the *Log* of a single process is $(n-1)p$. This takes $2(n-1)p$ integers per process. The total space needed for *Log*s corresponding to all p intervals on a single process is $2(n-1)p + 2np$. This gives an average of $4n-2$ integers per *Log*. As only one *Log* exists at a time, the average space requirement at a process P_i $(1 \leq i \leq n)$ at any time is the sum of space required by vector clock, *Interval Clock*, and *Log*, which is $6n-2$ integers.

Message complexity of control messages sent to the central process P_0 by processes P_1 to P_n. This can be determined in two ways. As one message is sent per interval, the number of messages is $O(p)$ for each P_i $(i \neq 0)$. This gives a total message complexity of $O(np)$. On the average, the size of each message is $4n-2$ as each message contains the *Log*. The total message space overhead for a particular process is the sum of all the *Log*s for that process, which was shown to be $4np-2p$. Hence the total message space complexity is $4n^2p - 2np = O(n^2p)$.

An optimization of message size. The *Log* corresponding to an interval is sent to the central process 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). 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'. For each message exchanged between processes, a maximum of four interval *Log*s need to be sent to the central process, because two successive intervals (Y and Y') will have different relationships if a receive or a send occurs between the start of Y and end of Y'. This makes it necessary to send two interval *Log*s for a send event and two for a receive event. Hence if there are m_s number of messages exchanged between all processes, then a total of $4m_s$ intervals need to be sent to the central process in $4m_s$ control messages, while the total message space overhead is $2m_s.n + 4m_s.2n = 10m_sn$. The term $2m_s.n$ arises because for every message sent, each other process eventually (due to transitive propagation of $Interval\ Clock$) may need to insert a $Event_Interval$ tuple in its *Log*. This can generate $2nm_s$ overhead in *Log*s across the n processes. The term $4m_s.2n$ arises because the vector clock at the start and end of each interval is sent with each message.

Hence, the total number of control messages sent to the central process and the total message space overhead is the lesser of when either four intervals are sent for each message exchanged or when all the intervals are sent. Thus the total number of messages sent is $O(min(4m_s, np))$ and the total message space overhead is $O(min(4n^2p - 2np, 10m_sn))$.

4 Algorithm *Fine_Rel*: Detecting Fine-Grained Relations

To solve Problems *Fine_Poss* and *Fine_Def*, we first state and solve an intermediate problem.

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

Note that the given relations $\{r_{i,j}, \forall i, j\}$ need to satisfy the axioms on $\Re$ [9] for a solution to potentially exist. A distributed and more complex method to solve *Fine_Rel*, using the data structures of Figure 3 and Theorem 2 below, without proofs or a complexity analysis, is presented in [3].

Algorithm Overview: The algorithm detects a set of intervals, one on each process, such that each pair of intervals satisfies the relationship specified for that pair of processes. If no such set of intervals exists, the algorithm does not return any interval set. The central process P_0 maintains n queues, one for *Log*s from each process and determines which orthogonal relation holds between pairs of intervals. The queues are processed using "pruning", described later. If there exists an interval at the head of each queue and these intervals cannot be pruned, then these intervals satisfy $r_{i,j}\ \forall\ i, j$, where $i \neq j$ and $1 \leq i, j \leq n$, and hence these intervals form a solution set.

For $S2(X, Y)$

1. // Eliminate from *Log* of interval Y (on P_j), all receives of messages //which were sent by P_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) delete *event_interval*.

2. // Select from the pruned *Log*, the earliest message sent from interval 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.

For $S1(Y, X)$

1. Same as step 1 of the algorithm to determine $S2(X, Y)$.
2. Same as step 2 of the algorithm 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. Tests for coupling relations $S1(X, Y)$ and $S2(Y, X)$ at P_0.

We first define the function $S(r_{i,j})$ and the relation $\vdash$. Recall that X and Y are intervals on P_i and P_j, respectively, and Y' is any interval that succeeds Y.

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

Intuitively, for each $r_{i,j} \in \Re$, we define a *prohibition function* $S(r_{i,j})$ 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.

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

Definition 3. $\vdash$ *is a relation on* $\Re \times \Re$ *such that* $R' \vdash R''$ *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, $IC \vdash IB$ 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.

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

Proof. If $R \in S(r_{i,j})$, using Definition 2, 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 3.

Table 5. $S(r_{i,j})$ for the 40 independent relations in $\Re$. The upper part of the table gives the function S on 29 relations assuming dense time. The lower part of the table gives function S for the 11 additional relations assuming non-dense time.

Interaction Type $r_{i,j}$	$S(r_{i,j})$	$S(r_{j,i})$
$IA\ (= IQ^{-1})$	ϕ	$\Re - \{IQ\}$
$IB\ (= IR^{-1})$	$\{IA, IF, II, IP, IO, IU, IX, IUX, IOP\}$	$\Re - \{IQ, IR\}$
$IC\ (= IV^{-1})$	$\{IA, IB, IF, II, IP, IO, IU, IX, IUX, IOP\}$	$\Re - \{IQ, IV\}$
$ID\ (= IX^{-1})$	$\Re - \{IQ, IS, IR, IJ, IL, IL', IL'', ID, ID', ID''\}$	$\Re - \{IQ, IX\}$
$ID'\ (= IU^{-1})$	$\Re - \{IQ, IS, IR, IJ, IL, IL', IL'', ID, ID', ID''\}$	$\Re - \{IQ, IU\}$
$IE\ (= IW^{-1})$	$\Re - \{IQ, IS, IR, IJ, IL, IL', IL'', ID, ID', ID'', IE\}$	$\Re - \{IQ, IW\}$
$IE'\ (= IT^{-1})$	$\Re - \{IQ, IS, IR, IJ, IL, IL', IL'', ID, ID', ID'', IE'\}$	$\Re - \{IQ, IT\}$
$IF\ (= IS^{-1})$	$\Re - \{IQ, IS, IR, IJ, IL, IL', IL'', ID, ID', ID'', IF\}$	$\Re - \{IQ, IS\}$
$IG\ (= IG^{-1})$	$\Re - \{IQ, IR, IJ, IV, IK, IG\}$	$\Re - \{IQ, IR, IJ, IV, IK, IG\}$
$IH\ (= IK^{-1})$	$\Re - \{IQ, IR, IJ, IV, IK, IG, IH\}$	$\Re - \{IQ, IR, IJ, IK\}$
$II\ (= IJ^{-1})$	$\Re - \{IQ, IR, IJ, IV, IK, IG, II\}$	$\Re - \{IQ, IR, IJ\}$
$IL\ (= IO^{-1})$	$\Re - \{IQ, IR, IJ, IL\}$	$\Re - \{IQ, IR, IJ, IO\}$
$IL'\ (= IP^{-1})$	$\Re - \{IQ, IR, IJ, IL'\}$	$\Re - \{IQ, IR, IJ, IP\}$
$IM\ (= IM^{-1})$	$\Re - \{IQ, IR, IJ, IM\}$	$\Re - \{IQ, IR, IJ, IM\}$
$IN\ (= IM'^{-1})$	$\Re - \{IQ, IR, IJ, IN\}$	$\Re - \{IQ, IR, IJ, IM'\}$
$IN'\ (= IN'^{-1})$	$\Re - \{IQ, IR, IJ, IN'\}$	$\Re - \{IQ, IR, IJ, IN'\}$
$ID''\ (= (IUX)^{-1})$	$\Re - \{IQ, IS, IR, IJ, IL, IL', IL'', ID, ID', ID''\}$	$\Re - \{IQ, IUX\}$
$IE''\ (= (ITW)^{-1})$	$\Re - \{IQ, IS, IR, IJ, IL, IL', IL'', ID, ID', ID'', IE''\}$	$\Re - \{IQ, ITW\}$
$IL''\ (= (IOP)^{-1})$	$\Re - \{IQ, IR, IJ, IL''\}$	$\Re - \{IQ, IR, IJ, IOP\}$
$IM''\ (= (IMN)^{-1})$	$\Re - \{IQ, IR, IJ, IM''\}$	$\Re - \{IQ, IR, IJ, IMN\}$
$IN''\ (= (IMN')^{-1})$	$\Re - \{IQ, IR, IJ, IN''\}$	$\Re - \{IQ, IR, IJ, IMN'\}$
$IMN''\ (= (IMN'')^{-1})$	$\Re - \{IQ, IR, IJ, IMN''\}$	$\Re - \{IQ, IR, IJ, IMN''\}$

Hence $R \not\vdash r_{i,j}$. 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 3 and hence $R \vdash r_{i,j}$. □

Table 5 gives $S(r_{i,j})$ for each of the 40 interaction types in $\Re$. The table is constructed by analyzing each interaction pair in $\Re$. We now state an important result between any two relations in $\Re$ that satisfy the "allows" relation, and the existence of the "allows" relation between their respective inverses. Specifically, if R' allows R'', then Theorem 2 states that R'^{-1} necessarily does not allow relation R''^{-1}. The theorem can be observed to be true from Lemma 1 and Table 5 by a case-by-case analysis.

Theorem 2. *For $R', R'' \in \Re$, if $R' \vdash R''$ then $R'^{-1} \not\vdash R''^{-1}$.*

Taking the same example, $IC \vdash IB \Rightarrow IV(= IC^{-1}) \not\vdash IR(= IB^{-1})$, which is indeed true. Note that $R' \neq R''$ in the definition of relation $\vdash$ is necessary; otherwise $R' \vdash R'$ will become true and from Theorem 2, we get $R'^{-1} \not\vdash R'^{-1}$ which leads to a contradiction.

Lemma 2. *If the relationship $R(X, Y)$ between intervals X and Y (belonging to process 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 resp.) is not equal to $r_{i,j}$, then either interval X or interval Y is removed from the queue.*

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 $R \vdash r_{i,j}$ and $R^{-1} \vdash r_{j,i}$. As $r_{i,j}^{-1} = r_{j,i}$, we get $R \vdash r_{i,j}$ and $R^{-1} \vdash r_{i,j}^{-1}$. This is a contradiction as by Theorem 2, $R \vdash r_{i,j} \Rightarrow R^{-1} \not\vdash r_{i,j}^{-1}$. Hence $R \in S(r_{i,j})$ or $R^{-1} \in S(r_{j,i})$ or both, and thus at least one of the intervals can be deleted. □

Theorem 3. *Algorithm Fine_Rel run by P_0 in Figure 5 solves Problem Fine_Rel.*

Proof. Lemma 2 which allows queues to be pruned correctly is implemented in the algorithm at P_0. The algorithm deletes interval X as soon as $R(X,Y) \in S(r_{i,j})$ (lines 13-17). Similarly, Y is deleted if $R(Y,X) \in S(r_{j,i})$ (lines 15-17). Thus, an interval gets deleted only if it cannot be part of the solution. Also clearly, each interval gets processed unless a solution is found using a predecessor interval from the same process. Lemma 3 gives the unique property that if $R(X,Y) \neq r_{i,j}$, then either interval X or interval Y is deleted. A consequence of this property is that if *every* queue is non-empty and their heads cannot be pruned, then a solution exists and the set of intervals at the head of each queue forms a solution.

The set *updatedQueues* stores the indices of all the queues whose heads got updated. In each iteration of the **while** loop, the indices of all the queues whose head satisfy Lemma 2 are stored in set *newUpdatedQueues* (lines (13)-(16)). In lines (17) and (18), the heads of all these queues are deleted and indices of the updated queues are stored in the set *updatedQueues*. Observe that only interval pairs which were not compared earlier are compared in subsequent iterations of the **while** loop. The loop runs until no more queues can be updated. If at this stage *all* the queues are non-empty, then a solution is found (follows from Lemma 3). If a solution is found, then for the intervals X (at P_i) and Y (at P_j) stored at the heads of these queues, we have $R(X,Y) = r_{i,j}$. □

For processes P_1 to P_n, the space complexity was shown in Section 3.2 to be on average $O(n)$ at each process. Using the optimization in Section 3.2, the total number of messages sent is equal to $O(min(4m_s, pn))$ and the total message space complexity is $O(min((4n-2)np, 10nm_s))$.

Theorem 4. *Algorithm Fine_Rel has the following complexities.*

- *The total message space complexity is $O(min((4n-2)np, 10nm_s))$.*
- *The total space complexity at process P_0 is $O(min((4n-2)np, 10nm_s))$.*
- *The average time complexity at P_0 is $O((n-1)min(4m_s, pn))$. This is equivalent to $O(n^2M)$, where M is maximum number of entries in a queue.*

Proof. For the central process P_0, the total space required is $O(min((4n-2)np,$-$10nm_s))$ because the total space overhead at P_0 is equal to the total message space complexity, which was computed in Section 3.2.

The time complexity is the product of the number of steps required to determine a relationship and the number of relations determined.

Consider the first part of the product.

```
queue of Log: Q_1, Q_2, ... Q_n = ⊥
set of int: updatedQueues, newUpdatedQueues = {}
On receiving interval from process P_z at P_0
(1) Enqueue the interval onto queue Q_z
(2) if (number of intervals on Q_z is 1) then
(3)       updatedQueues = {z}
(4)       while (updatedQueues is not empty)
(5)             newUpdatedQueues={}
(6)             for each i ∈ updatedQueues
(7)                   if (Q_i is non-empty) then
(8)                         X = head of Q_i
(9)                         for j = 1 to n
(10)                              if (Q_j is non-empty) then
(11)                                    Y = head of Q_j
(12)                                    Test for R(X,Y) using the tests in Fig. 4 and Tab. 2
(13)                                    if (R(X,Y) ∈ S(r_{i,j})) then
(14)                                          newUpdatedQueues = {i} ∪ newUpdatedQueues
(15)                                    if (R(Y,X) ∈ S(r_{j,i})) then
(16)                                          newUpdatedQueues = {j} ∪ newUpdatedQueues
(17)            Delete heads of all Q_k where k ∈ newUpdatedQueues
(18)            updatedQueues = newUpdatedQueues
(19)      if (all queues are non-empty) then
(20)            solution found. Heads of queues identify intervals that form the solution.
```

Fig. 5. On-line algorithm *Fine_Rel* at P_0.

- The total number of interval pairs between any two processes P_i and P_j is p^2. To determine $R1(X,Y)$ to $R4(X,Y)$ and $R1(Y,X)$ to $R4(Y,X)$, as eight comparisons are needed for each interval pair, a total of $8p^2$ comparisons are necessary for any pair of processes.
- To determine the number of comparisons required by $S1$ and $S2$, consider the maximum number of *Event_Intervals* stored in $Log_j.p_log[i]$ that are sent over the execution lifetime to the central process as part of the *Log*s. This is the maximum number of *Event_Intervals* corresponding to P_i stored in Q_j over P_j's execution lifetime. An *Event_Interval* is added to $Log_j.p_log[i]$ only when there is a change in the ith component of *Interval Clock* at the receive of a message. As the ith component of *Interval Clock* changes only when a new interval starts, the total number of times the ith component of *Interval Clock* changes is at most equal to p, the maximum number of intervals occurring on the other process P_i. From Figure 4, it can be observed that for each *Event_Interval*, there is one comparison. Thus, to determine the relationship between an interval on P_i and all other intervals on P_j, the number of comparisons is equal to p. As there are p intervals on P_i, a total of p^2 comparisons are required to determine $S1$ or $S2$. Hence the total number of comparisons to determine $S1(X,Y)$, $S2(X,Y)$, $S1(Y,X)$, and $S2(Y,X)$ is $4p^2$.

This gives a total of $8p^2 + 4p^2 = O(p^2)$ comparisons to determine the relation between each pair of intervals on a pair of processes. As there are a total of p^2 intervals pairs between two processes, the average number of comparisons required to determine a relationship is $O(1)$.

To analyse the second part of the product, consider Figure 5. For each interval considered from one of the queues in *updatedQueues* (lines (6)-(12)), the number of relations determined is $n-1$. Thus the number of relations determined for each iteration of the **while** loop is $(n-1)|updatedQueues|$, where $|updatedQueues|$ denotes the number of entries in *updatedQueues*. The cumulative $\sum |updatedQueues|$ over all iterations of the **while** loop is less than the total number of intervals over all the queues. Thus, the total number of relations determined is less than $(n-1)min(4m_s, pn)$, where $min(4m_s, np)$ is the upper bound on the total number of intervals over all the queues. As the average time required to determine a relationship is $O(1)$, the average time complexity of the algorithm is equal to $O((n-1)min(4m_s, pn))$.

The average time complexity can be equivalently expressed using M, the maximum number of entries in a queue, as follows. The total number of intervals over all the queues is $O(nM)$. As the total number of relations determined is $(n-1)\sum |updatedQueues|$ over all the iterations of the **while** loop, this is equivalent to $(n-1).nM = O(n^2M)$. This is also the average time complexity because it takes $O(1)$ time on the average to determine a relationship. □

Table 1 compares the complexities of *Fine_Rel* with those of GW94 [6] and GW96 [7]. GW94 and GW96 computed their time complexity at P_0 as only $O(n^2M)$, not in terms of m_s or p. They did not give the space complexity at P_0. As each control message in GW94 and GW96 carries a fixed size $O(n)$ message overhead and a control message is sent to P_0 for every message send/receive event, we have computed their total space complexity and average time complexity at P_0 as $O(nm_s)$. This enables a direct comparison with the complexities of our algorithm. Further, we have also computed our average time complexity using M, as $O(n^2M)$. In our algorithm, note that $M \leq p$; $M = p$ if the message overhead optimization is not used. We do not express the total space at P_0 in terms of M because the queue entries are of variable size, with an average size of $(4n-2)$ integers.

5 Algorithms *Fine_Poss* and *Fine_Def*

By leveraging Theorem 1 and the mapping of fine-grained modalities to *Possibly* and *Definitely* modalities, as given in Table 4, we address the problems of determining whether *Possibly*(ϕ) and *Definitely*(ϕ) hold. If either of these two coarse-grained modalities holds, we can also determine the exact fine-grained orthogonal relation/modality between each pair of processes, unlike any previous algorithm. Further, the time, space, and message complexities of the proposed on-line (centralized) detection algorithms (Algorithms *Fine_Poss* and *Fine_Def*) to detect *Possibly* and *Definitely* in terms of the fine-grained modalities per pair of processes, are the *same as* those of the earlier on-line

(centralized) algorithms [6,7] that can detect only whether the *Possibly* and *Definitely* modalities hold.

Recall that $\Re$ is a set of orthogonal relations and hence one and only one relation from $\Re$ must hold between any pair of intervals. Consider the case where, for each pair of processes (P_i, P_j), we are given a set $r^*_{i,j} \subseteq \Re$ such that we are satisfied if some relation in $r^*_{i,j}$ holds. Now consider the objective where we need to identify one interval per process such that for each process pair (P_i, P_j), some relation in $r^*_{i,j}$ holds for that (P_i, P_j). Such an objective would be useful if we can leverage the coarse-to-fine mapping of modalities, given in Table 4. We formalize such an objective by generalizing the detection problem *Fine_Rel* to problem *Fine_Rel′*, as follows.

Problem *Fine_Rel′* **Statement:** *Given a set of relations $r^*_{i,j} \subseteq \Re$ for each pair of processes P_i and P_j, determine on-line the intervals, if they exist, one from each process, such that any one of the relations in $r^*_{i,j}$ is satisfied (by the intervals) for each (P_i, P_j) pair. If a solution exists, identify the fine-grained interaction from $\Re$ for each pair of processes in the first solution.*

To solve *Fine_Rel′*, given an *arbitrary* $r^*_{i,j}$, a solution based on algorithm *Fine_Rel* (Figure 5) will not work because in the crucial tests in lines (13)-(14), neither interval may be removable, and yet none of the relations from $r^*_{i,j}$ might hold between the two intervals. This leads to deadlock! To see this further, let $r1, r2 \in r^*_{i,j}$ and let $R(X, Y)$ hold, where $R \not\in r^*_{i,j}$. Now let $R \in S(r1)$, $R^{-1} \not\in S(r1^{-1})$, $R \not\in S(r2)$, $R^{-1} \in S(r2^{-1})$. Interval X cannot be deleted because $r2(X, Y')$ may be true for a successor Y'. Interval Y cannot be deleted because $r1^{-1}(Y, X')$ may be true for a successor X'. Therefore, a solution based on Algorithm *Fine_Rel* will deadlock, and a more elaborate (and presumably expensive) solution will be needed.

We now identify and define a special property, termed $\mathcal{CONVEXITY}$, on $r^*_{i,j}$ such that the deadlock is prevented. Informally, this property says that there is no relation R outside $r^*_{i,j}$ such that for any $r1, r2 \in r^*_{i,j}$, $R \vdash r1$ and $R^{-1} \vdash r2^{-1}$. This property guarantees that when intervals X and Y are compared for $r^*_{i,j}$ and $R(X, Y)$ holds, either X or Y or both get deleted, and hence there is progress. The sets $r^*_{i,j}$, derived from Table 4, that need to be detected to solve Problems *Fine_Poss* and *Fine_Def* satisfy this property. We therefore observe that problems *Fine_Poss* and *Fine_Def* are special cases of Problem *Fine_Rel′* in which the property $\mathcal{CONVEXITY}$ on $r^*_{i,j}$ is necessarily satisfied. To solve Problems *Fine_Poss* and *Fine_Def*, we then use the generalizations of Lemmas 2 and 3, as given in Lemmas 4 and 5, respectively, to first solve *Fine_Rel′*.

Definition 4.

$$\mathcal{CONVEXITY}: \forall R \not\in r^*_{i,j} : (\forall r_{i,j} \in r^*_{i,j}, R \in S(r_{i,j}) \bigvee \forall r_{j,i} \in r^*_{j,i}, R^{-1} \in S(r_{j,i}))$$

Lemma 4. *If the relationship $R(X, Y)$ between intervals X and Y (belonging to processes P_i and P_j, respectively) is contained in the set $\bigcap_{r_{i,j} \in r^*_{i,j}} S(r_{i,j})$, then interval X can be removed from the queue Q_i.*

(13) **if** $(R(X,Y) \in \bigcap_{r_{i,j} \in r^*_{i,j}} S(r_{i,j}))$ **then**
(14) $newUpdatedQueues = \{i\} \cup newUpdatedQueues$
(15) **if** $(R(Y,X) \in \bigcap_{r_{j,i} \in r^*_{j,i}} S(r_{j,i}))$ **then**
(16) $newUpdatedQueues = \{j\} \cup newUpdatedQueues$

Fig. 6. Algorithm $Fine_Rel'$: Changes to algorithm $Fine_Rel$ are listed, assuming $r^*_{i,j}$ satisfies property $\mathcal{CONVEXITY}$.

Proof. From the definition of $S(r_{i,j})$, we infer that no relation $r_{i,j}(X,Y')$, where $r_{i,j} \in r^*_{i,j}$ and Y' is any successor interval of Y on P_j, can be true. Hence interval X can never be a part of the solution and can be deleted from the queue. □

Lemma 5. *If the relationship $R(X,Y)$ between a pair of intervals X and Y (belonging to processes P_i and P_j, respectively) does not belong to the set $r^*_{i,j}$, where $r^*_{i,j}$ satisfies property $\mathcal{CONVEXITY}$, then either interval X or interval Y is removed from the queue.*

Proof. We use contradiction. Assume relation $R(X,Y)$ ($\notin r^*_{i,j}(X,Y)$) is true for intervals X and Y. From Lemma 4, the only time neither X nor Y will be deleted is when both $R \notin \bigcap_{r_{i,j} \in r^*_{i,j}} S(r_{i,j})$, and $R^{-1} \notin \bigcap_{r_{j,i} \in r^*_{j,i}} S(r_{j,i})$. However, as $r^*_{i,j}$ satisfies property $\mathcal{CONVEXITY}$, we have that $R \in \bigcap_{r_{i,j} \in r^*_{i,j}} S(r_{i,j})$ or $R^{-1} \in \bigcap_{r_{j,i} \in r^*_{j,i}} S(r_{j,i})$ must be true. Thus at least one of the intervals can be deleted by an application of Lemma 4. □

The proof of the following theorem is similar to the proof of Theorem 3.

Theorem 5. *If the set $r^*_{i,j}$ satisfies property $\mathcal{CONVEXITY}$, then Problem $Fine_Rel'$ is solved by replacing lines (13) and (15) in algorithm $Fine_Rel$ in Figure 5 by the lines (13) and (15) in Figure 6.*

Proof. Analogous to the proof of Theorem 3. Use Lemmas 4 and 5 instead of Lemmas 2 and 3, respectively, and reason with $r^*_{i,j}$ instead of with $r_{i,j}$. □

Corollary 1. *The time, space, and message complexities of Algorithm $Fine_Rel'$ are the same as those of Algorithm $Fine_Rel$, which were stated in Theorem 4.*

Proof. The only changes to Algorithm $Fine_Rel$ are in lines (13) and (15). In Algorithm $Fine_Rel'$, instead of checking $R(X,Y)$ for membership in $S(r_{i,j})$ in line (13), $R(X,Y)$ is checked for membership in $\bigcap_{r_{i,j} \in r^*_{i,j}} S(r_{i,j})$. Both $S(r_{i,j})$ and $\bigcap_{r_{i,j} \in r^*_{i,j}} S(r_{i,j})$ are sets of size between 0 and 40. An analogous observation holds for the change on line (15). Hence, the time, space, and message complexities of $Fine_Rel$ are unaffected in $Fine_Rel'$. □

To detect $Possibly(\phi)$, $r^*_{i,j}$ is set to the union of the orthogonal interactions in the first two columns of Table 4. We can verify (by case-by-case enumeration) that $r^*_{i,j}$ does satisfy property $\mathcal{CONVEXITY}$. Similarly, to detect

$Definitely(\phi)$, $r^*_{i,j}$ is set to the union of the orthogonal interactions in the first column of Table 4. We can verify (by case-by-case enumeration) that $r^*_{i,j}$ does satisfy property $\mathcal{CONVEXITY}$.

The following two theorems about using algorithm *Fine_Rel′* (Figure 6) to solve Problems *Fine_Poss* and *Fine_Def* can be readily proved by using Theorem 1, the refinement mapping of Table 4, and Theorem 5. The two resulting algorithms are named *Fine_Poss* and *Fine_Def*, respectively.

Theorem 6. *Algorithm Fine_Rel modified to algorithm Fine_Rel′ (Figure 6) solves Problem Fine_Poss (about $Possibly(\phi)$) when $r^*_{i,j}$ is set to the union of the relations in the first and second columns of Table 4.*

Proof. From Theorem 1, $Possibly(\phi)$ is true if and only if $(\forall i \in N)(\forall j \in N)Possibly(\phi_i \wedge \phi_j)$. For any i and j, $Possibly(\phi_i \wedge \phi_j)$ is true if and only if $R(X_i, Y_j)$ is any of the temporal relations given in the first two columns of Table 4. When $r^*_{i,j}$ is set to the union of the relations in these two columns, we can verify (by case-by-case enumeration) that $r^*_{i,j}$ satisfies $\mathcal{CONVEXITY}$. As Algorithm *Fine_Rel′* is correct (by Theorem 6), when its $r^*_{i,j}$ is instantiated with the set above to get Algorithm *Fine_Poss*, we have that *Fine_Poss* is also correct. □

Theorem 7. *Algorithm Fine_Rel modified to algorithm Fine_Rel′ (Figure 6) solves Problem Fine_Def (about $Definitely(\phi)$) when $r^*_{i,j}$ is set to the union of the relations in the first column of Table 4.*

Proof. From Theorem 1, $Definitely(\phi)$ is true if and only if $(\forall i \in N)(\forall j \in N)Definitely(\phi_i \wedge \phi_j)$. For any i and j, $Definitely(\phi_i \wedge \phi_j)$ is true if and only if $R(X_i, Y_j)$ is any of the temporal relations given in the first column of Table 4. When $r^*_{i,j}$ is set to the relations in this column, we can verify (by case-by-case enumeration) that $r^*_{i,j}$ satisfies $\mathcal{CONVEXITY}$. As Algorithm *Fine_Rel′* is correct (by Theorem 6), when its $r^*_{i,j}$ is instantiated with the set above to get Algorithm *Fine_Def*, we have that *Fine_Def* is also correct. □

In algorithm *Fine_Rel′*, when $r^*_{i,j}$ is set to the values as specified in Theorems 6 and 7 to detect *Possibly* and *Definitely*, respectively, set $\bigcap_{r_{i,j} \in r^*_{i,j}} S(r_{i,j})$ used in line (13) of the algorithm becomes $\{IA\}$ and $\{IA, IB, IC, IG, IH, II\}$, respectively. An identical change occurs to the set $\bigcap_{r_{j,i} \in r^*_{j,i}} S(r_{j,i})$ on line (15).

Corollary 2. *The time, space, and message complexities of Algorithms Fine_-Poss and Fine_Def are the same as those of Algorithm Fine_Rel (stated in Theorem 4) and of Algorithm Fine_Rel′ (stated in Corollary 1).*

Proof. Follows from Corollary 1 and the fact that $r^*_{i,j}$ for *Fine_Poss* and *Fine_Def* satisfy $\mathcal{CONVEXITY}$ and are instantiations of $r^*_{i,j}$ in *Fine_Rel′*. □

6 Discussion & Conclusions

This paper presented algorithms to detect conjunctive predicates under fine-grained modalities. Algorithms *Fine_Poss* and *Fine_Def* not only detect *Possibly*(ϕ) and $Definitely(\phi)$, respectively, *but also (unlike previous algorithms)*

return the pairwise fine-grained relations which exist between all the intervals in the solution set. The space, message, and computational complexities of the previous works for conjunctive predicate detection, GW94 [6] and GW96 [7], for detection of only $Possibly(\phi)$ and $Definitely(\phi)$, respectively, is compared with our algorithms in Table 1. All the complexity measures for algorithms $Fine_Poss$ and $Fine_Def$ are the same as those for GW94 [6] and GW96 [7]. Thus with the same overhead, Algorithms $Fine_Poss$ and $Fine_Def$ do the *extra work* of finding the fine-grained relations which exist between the intervals contained in the solution set for $Possibly$ and $Definitely$.

A detailed version of these results appears in [1]. Distributed algorithms can be devised for $Fine_Poss$ and $Fine_Def$ based on the distributed algorithm given in [3] to solve $Fine_Rel$. A discussion of how intervals might be identified when trying to use the fine-grained modalities on nonconjunctive predicates, i.e., general relational predicates, is given in [11].

Acknowledgements

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

References

1. Chandra, P., Kshemkalyani, A.D.: Algorithms for Detecting Global Predicates under Fine-grained Modalities, Technical Report UIC-ECE-02-05, University of Illinois at Chicago, April 2002.
2. Chandra, P., Kshemkalyani, A.D.: Distributed Algorithm to Detect Strong Conjunctive Predicates, Information Processing Letters, 87(5): 243-249, September 2003.
3. Chandra, P., Kshemkalyani, A.D.: Detection of Orthogonal Interval Relations, Proc. High-Performance Computing Conference, 323-333, LNCS 2552, Springer, 2002.
4. Cooper, R., Marzullo, K.: Consistent Detection of Global Predicates, Proc. ACM/ONR Workshop on Parallel & Distributed Debugging, 163-173, May 1991.
5. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed Systems Concepts and Design, Addison-Wesley, 3rd edition, 2001.
6. Garg, V.K., Waldecker, B.: Detection of Weak Unstable Predicates in Distributed Programs, IEEE Trans. Parallel & Distributed Systems, 5(3), 299-307, Mar. 1994.
7. Garg, V.K., Waldecker, B.: Detection of Strong Unstable Predicates in Distributed Programs, IEEE Trans. Parallel & Distributed Systems, 7(12):1323-1333, Dec. 1996.
8. Hurfin, M., Mizuno, M., Raynal, M., Singhal, M.: Efficient Distributed Detection of Conjunctions of Local Predicates, IEEE Trans. Software Engg., 24(8): 664-677, 1998.
9. Kshemkalyani, A.D.: Temporal Interactions of Intervals in Distributed Systems, Journal of Computer and System Sciences, 52(2): 287-298, April 1996.
10. Kshemkalyani, A.D.: A Fine-Grained Modality Classification for Global Predicates, IEEE Trans. Parallel & Distributed Systems, 14(8): 807-816, August 2003.

11. Kshemkalyani, A.D.: A Note on Fine-grained Modalities for Nonconjunctive Predicates, 5th Workshop on Distributed Computing, LNCS, Springer, Dec. 2003.
12. Kshemkalyani, A.D.: A Framework for Viewing Atomic Events in Distributed Computations, Theoretical Computer Science, 196(1-2), 45-70, April 1998.
13. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, Communications of the ACM, 558-565, 21(7), July 1978.
14. Marzullo, K., Neiger, G.: Detection of Global State Predicates, Proc. 5th Workshop on Distributed Algorithms, LNCS 579, Springer-Verlag, 254-272, October 1991.
15. Mullender, S.: Distributed Systems, 2nd Edition, ACM Press, 1994.
16. Stoller, S., Schneider, F.: Faster Possibility Detection by Combining Two Approaches, Proc. 9th Workshop on Distributed Algorithms, 318-332, LNCS 972, Springer-Verlag, 1995.
17. Venkatesan, S., Dathan, B.: Testing and Debugging Distributed Programs Using Global Predicates, IEEE Trans. Software Engg., 21(2), 163-177, Feb. 1995.