Efficient Distributed Detection of Conjunctions of Local Predicates

Michel Hurfin, Masaaki Mizuno, Michel Raynal, and Mukesh Singhal, *Member*, *IEEE Computer Society*

Abstract—Global predicate detection is a fundamental problem in distributed systems and finds applications in many domains such as testing and debugging distributed programs. This paper presents an efficient distributed algorithm to detect conjunctive form global predicates in distributed systems. The algorithm detects the first consistent global state that satisfies the predicate even if the predicate is unstable. Unlike previously proposed run-time predicate detection algorithms, our algorithm does not require exchange of control messages during the normal computation. All the necessary information to detect predicates is piggybacked on computation messages of application programs. The algorithm is distributed because the predicate detection efforts as well as the necessary information are equally distributed among the processes. We prove the correctness of the algorithm and compare its performance with respect to message, storage, and computational complexities with that of the previously proposed run-time predicate detection algorithms.

Index Terms—Distributed systems, on-the-fly global predicate detection.

1 Introduction

Development of distributed applications requires the ability to analyze their behavior at run time to debug or control the execution. In particular, it is sometimes essential to know if a property is satisfied (or not) by a distributed computation. Properties of the computation, which specify desired (or undesired) evolutions of the program's execution state, are described by means of predicates over local variables of component processes.

A *basic* predicate refers to the program's execution state at a given time. These predicates are divided into two classes called *local predicates* and *global predicates*. A local predicate is a general boolean expression defined over the local state of a single process, whereas a global predicate is a boolean expression involving variables managed by several processes. Due to the asynchronous nature of a distributed computation, it is impossible for a process to determine the total order in which the events occurred in the physical time. Consequently, it is often impossible to determine the global states through which a distributed computation passed through, complicating the task of ascertaining if a global predicate became true during a computation.

Basic predicates are used as building blocks to form more complex classes of predicates such as linked predicates [17], simple sequences [1], [6], [10], interval-constrained sequences [1], regular patterns [5], or atomic sequences [9], [10]. The above class of properties are useful in characterizing the evolution of a program's execution state, and protocols exist for detecting these properties at run time by way of language recognition techniques [2].

When the property (i.e., a combination of the basic properties) contains no global predicate, the detection can be done locally without introducing any delays, without defining a centralized process, and without exchanging any control messages. Control information is just piggybacked to the existing message of the application. However, if the property refers to at least to one global predicate, then all possible observations of the computation must be considered. In other words, the detection of the property requires the construction and the traversal of the lattice of consistent global states representing all observations of the computation. When the property reduces to one global predicate, the construction of the lattice can be avoided in some cases. If the property is expressed as a disjunction of local predicates, then obviously no cooperation between processes is needed in order to detect the property during a computation. A form of global predicate, namely, the conjunction of local predicates, has been the focus of research [6], [7], [8], [15], [20] recently. In such predicates, the number of global states of interest in the lattice is considerably reduced because all global states that include a local state where the local predicate is false need not be examined.

1.1 Previous Work

The problem of global predicate detection has attracted considerable attention lately, and a number of global predicate detection algorithms have been recently proposed. In the centralized algorithm of Cooper and Marzullo [3], every process reports each of its local states to a designated process, which builds a lattice of the global computation and checks if a state in the computation satisfies the global

- M. Hurfin and M. Raynal are with IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France. E-mail: {hurfin, raynal}@irisa.fr.
- M. Mizuno is with the Department of Computing and Information Sciences, Kansas State University, Manhattan, KS 66506. E-mail: masaaki@cis.ksu.edu.
- M. Singhal is with the Department of Computer and Information Science, Ohio State University, Columbus, OH 43210. E-mail: singhal@cis.ohio-state.edu.

Manuscript received 7 Dec. 1995; revised 9 Sept. 1997.
Recommended for acceptance by K. Marzullo.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 105629.

predicate. The power of this algorithm lies in generality of the global predicates it can detect; however, the algorithm has a very high overhead. If a computation has n processes and if m is the maximum number of events in any process, then the lattice consists of $O(m^n)$ states in the worst case. Thus, the worst case time complexity of this algorithm is $O(m^n)$. The algorithm in [13] has linear space complexity; however, the worst case time complexity is still linear in the number of states in the lattice.

Since the detection of generalized global predicates by building and searching the entire state space of a computation is utterly prohibitive, researchers have developed faster, more efficient global predicate detection algorithms by restricting themselves to special classes of predicates. As indicated previously, a form of global predicates that is expressed as the conjunction of several local predicates has been a focus of research [6], [7], [8], [15], [20]. Detection of such predicates can be done during a replay of the computation [15], [20] or at runtime during the computation [6], [7], [8]. This paper focuses on the second kind of solution which allows one to detect the predicate even before the end of the computation. In the Garg-Waldecker centralized algorithm to detect such predicates [7], a process gathers information about the local states of the processes, builds only those global states that satisfy the global predicate, and checks if a constructed global state is consistent. In the distributed algorithm of Garg and Chase [8], a token is used that carries information about the latest global state (cut) such that the local predicates hold at all the respective local states. The worst case time complexity of both these algorithms is $O(mn^2)$ which is linear in m and is much smaller than the worst case time complexity of the methods that require searching the entire lattice. However, the price paid is that not all properties can be expressed as the conjunction of local predicates.

Recently, Stoller and Schneider [19] proposed an algorithm that combines the Garg-Waldecker approach [7] with any approach that constructs a lattice to detect Possibly(Φ). (A distributed computation satisfies Possibly(Φ) iff predicate Φ holds in a state in the corresponding lattice.) This algorithm has the best features of both the approaches—it can detect Possibly(Φ) for any predicate Φ and it detects a global predicate expressed as the conjunction of local predicates in time linear in m (the maximum number of events in m any process).

1.2 Paper Objectives

This paper presents an efficient distributed algorithm to detect conjunctive form global predicates in distributed systems. Unlike previously proposed run-time algorithms, our algorithm does not require exchange of additional control messages during the normal computation. All the necessary information to detect predicates are piggybaked on computation messages of the application program. In addition to minimizing the message traffic, this property leads to a major advantage in reliability; even though application programs run on unreliable network environments, as long as they can tolerate message loss, the algorithm will capture causal relations correctly and detect solutions. On the other hand, in the previously proposed run-time algorithms that require additional control messages, if control messages are lost, the detection algorithms do not work. We prove the correctness of the algorithm and compare its performance with that of the previous algorithms to detect conjunctive form global predicates.

The rest of the paper is organized as follows: In Section 2, we define the system model and introduce necessary definitions and notations. Section 3 presents our global predicate detection algorithm and gives its correctness proof. In Section 4, we compare the performance of our algorithm with that of the previously proposed run-time algorithms for detecting conjunctive form global predicates. Finally, Section 5 contains concluding remarks.

2 System Model, Definitions, and Notations

2.1 Distributed Computations

A distributed program consists of n sequential processes denoted by $P_1, P_2, \cdots, P_n$. The concurrent execution of all the processes on a network of processors is called a distributed computation. The processes do not share a global memory and a global clock. Message passing is the only way for processes to communicate with one another. The computation is asynchronous; each process evolves at his own speed and messages are exchanged through communication channels, whose transmission delays are finite but arbitrary. We assume that no messages are altered or spuriously introduced. No assumption is made about the FIFO nature of the channels. Channels can get severed and messages can be lost during the computation.

2.2 Events, Local States, and Intervals

2.2.1 Communication Events

During a computation, each process P_i can execute internal, send, and receive statements. An internal statement does not involve communication. When P_i executes a statement "*send(m, j)*," it puts the message m on the channel from P_i to P_j. When P_i executes the statement "*receive(m)*," one message in a channel to P_i is removed and delivered to P_i. If no message exists in a channel to P_i, P_i is blocked until a message arrives on a channel. Execution of internal, send, and receive statements are modeled by internal, send, and receive events, respectively.

In order to evaluate a conjunction of local predicates, internal events are not important; therefore, we focus our attention on only send and receive events, called *communication events*. We use e_i^x to denote the xth send or receive event which occurs at process P_i. Thus, during a computation, the execution of process P_i is characterized by a sequence of communication events $E_i \equiv e_i^1\ e_i^2 \cdots e_i^x \cdots$.

For each process P_i, we define an additional event, denoted as e_i^0, that occurred at process P_i at the beginning of the computation. Furthermore, if the computation terminates, the last send or receive event executed at process P_i (denoted as $e_i^{l_i}$) is followed by an imaginary event, denoted as $e_i^{l_i+1}$.

The "happened before" causal precedence relation of Lamport induces a partial order on the events of a distributed computation. This transitive relation, denoted by $\prec$, is defined as follows:

$$\forall e_i^x,\ \forall e_j^y,\ e_i^x \prec e_j^y \Leftrightarrow \begin{cases} (i = j) \wedge (x < y) \\ \text{or} \\ \text{There exists a message } m \\ \text{such that} \\ e_i^x \text{ is the send event } \mathit{send}(m, j) \text{ and} \\ e_j^y \text{ is the receive event } \mathit{receive}(m) \\ \text{or} \\ \text{There exists an event } e_k^z \\ \text{such that} \\ e_i^x \prec e_k^z \text{ and } e_k^z \prec e_j^y \end{cases}$$

This relation is extended to a reflexive relation denoted $\preccurlyeq$.

2.2.2 *Local States and Intervals*

At a given time, the local state of a process P_i is defined by the values of the local variables managed by P_i. To evaluate conjunctions of local predicates, it is necessary to identify causal relations among local states of different processes, which are caused by communication events. To effectively capture such causal relations, we introduce the notion of *intervals*. An interval is a segment of time on a process that begins and ends with consecutive communication events. Formally, the xth interval of process P_i, denoted by θ_i^x, is a segment of the computation that begins at event e_i^x and ends at e_i^{x+1}.

The relation that expresses causal dependencies among intervals is denoted by $\rightarrow$. This relation induces a partial order on the intervals of distributed computation and is defined as follows:

$$\forall \theta_i^x,\ \forall \theta_j^y,\ \theta_i^x \rightarrow \theta_j^y \Leftrightarrow e_i^{x+1} \preccurlyeq e_j^y$$

A set of intervals is consistent if for any pair of intervals in the set, say θ_i^x and θ_j^y, $\neg(\theta_i^x \rightarrow \theta_j^y)$.

The example of distributed computation depicted in Fig. 1 illustrates the notations given previously. This example will be used several times in this paper.

2.3 Conjunctions of Local Predicates

2.3.1 *Local Predicates and Verified Intervals*

A local predicate defined over local states of process P_i is denoted by $\mathcal{L}_i$. $\mathcal{L}_i$ can be evaluated by process P_i at any time without communicating with any other process. Notation $\theta_i^x \models \mathcal{L}_i$ indicates that $\mathcal{L}_i$ is satisfied in interval θ_i^x. We call an interval during which its associated local predicate is verified a *verified interval*. In Fig. 2, we consider two local predicates and represent verified intervals by grey rectangles.

2.3.2 Cuts

Let Φ denote a conjunction of p local predicates, where $p \le n$. Without loss of generality, we assume that the p processes involved in conjunction Φ are $P_1, P_2, \cdots, P_p$. We write either Φ or $\mathcal{L}_1 \wedge \mathcal{L}_2 \wedge \cdots \wedge \mathcal{L}_p$ to denote the conjunction.

A set containing p intervals, one for each process P_k such that $k \le p$, is called a *cut*. We use notation C to denote a cut. A cut is consistent if the set of intervals is consistent. A dependency relation denoted by $\rightsquigarrow$ is defined over the set of all consistent cuts as follows. Let $C^x = \{\theta_1^{x_1}, \theta_2^{x_2}, \cdots, \theta_p^{x_p}\}$ and $C^y = \{\theta_1^{y_1}, \theta_2^{y_2}, \cdots, \theta_p^{y_p}\}$ be two consistent cuts, then:

$$C^x \rightsquigarrow C^y \Leftrightarrow (C^x \ne C^y) \wedge (\forall k, 1 \le k \le p, x_k \le y_k)$$

The set of all consistent cuts for a distributed computation is represented by a lattice structure whose minimal element corresponds to the cut $\{\theta_1^0, \theta_2^0, \cdots, \theta_p^0\}$ [3]. An edge exists from a cut $\{\theta_1^{x_1}, \cdots, \theta_j^{x_j}, \cdots, \theta_p^{x_p}\}$to a cut $\{\theta_1^{x_1}, \cdots, \theta_j^{x_j+1}, \cdots, \theta_p^{x_p}\}$ if the distributed computation can reach the

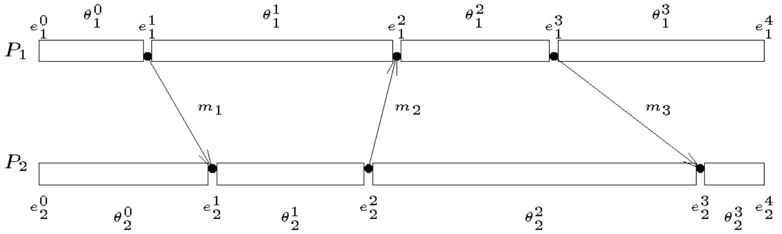

Fig. 1. A distributed computation.

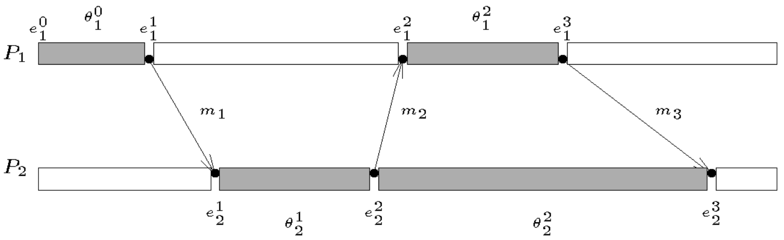

Fig. 2. Illustration of verified intervals.

latter from the former when process P_j executes its next communication event $e_j^{x_j+1}$. Each path of the lattice starting at the minimal element corresponds to a possible observation of the distributed computation. Each observation is identified by a sequence of events, where all events of the computation appear in an order consistent with the "happened before" relation.

2.3.3 Solutions

A cut $C^s = \{\theta_1^{s_1}, \theta_2^{s_2}, \cdots, \theta_p^{s_p}\}$ is a *solution* if and only if the cut is consistent and is composed exclusively of verified intervals:

$$\forall k,\ 1 \le k \le p, \begin{cases} \forall l,\ 1 \le l \le p,\ l \ne k,\ \neg(\theta_l^{s_l} \to \theta_k^{s_k}) \\ \wedge \\ \theta_k^{s_k} \models \mathcal{L}_k \end{cases}$$

Due to the relationship between local states and intervals, this definition of a solution is consistent with definitions proposed earlier [6], [7], [8].

First Solution. Let S denote the set of all cuts which are solutions. If S is not empty, the *first solution* is the unique element of S denoted by C^f such that every element C^s of S satisfies the following property:

$$(C^f \rightsquigarrow C^s) \vee (C^f = C^s)$$

That is, C^f is the first set of p intervals which verifies Φ. This particular solution, if it exists, is well defined in the computation.

2.3.4 Modal Operators on Predicates

Given a conjunction Φ, various modal operators have been defined [11]. Two well-known modal operators are Possibly(Φ) and Definitely(Φ), introduced by Cooper and Marzullo [3]. A distributed computation satisfies Possibly(Φ) if and only if the lattice of consistent cuts has a cut verifying the predicate Φ, whereas Definitely(Φ) is satisfied by the computation if and only if each observation (i.e., each path in the lattice) passes through a consistent cut verifying Φ. The algorithm we present in this paper detects the first solution for Possibly(Φ). Possibly(Φ) is particularly important to test and debug distributed executions.

3 The Detection Algorithm

To detect a consistent cut of intervals, each of which verifies its local predicate, the following two approaches are possible:

1) Processes always keep track of a set of intervals that form a consistent cut. For each such cut, each process checks whether its interval in the set verifies its local predicate.
2) A process always keeps track of a set of verified intervals. For each such set, the process checks whether the cut is consistent.

Note that these two approaches are complementary. We have developed two algorithms based on these two approaches that are dual of each other. Even though these two approaches are quite different, interestingly these two algorithms are strikingly similar. Therefore, this paper presents only the algorithm based on the first approach. A description of the algorithm based on the second approach can be found in [12].

In the algorithm, the detection of the property is performed at runtime without exchanging any control messages and without relying on a centralized process. All processes cooperate to detect a solution by piggybacking control information on the computation messages.

3.1 Data Structures

3.1.1 Dependency Vectors

To identify a set of p concurrent intervals, the algorithm keeps track of causal dependencies among intervals by using a vector clock mechanism similar to that described in [4], [16], [18]. Each process P_i ($1 \le i \le n$) maintains an integer vector $D_i[1..p]$, called the *dependency vector*. All elements *of* D_i are initialized to zero. Since causal relations between two intervals at different processes are created by communication events (and their transitive relation), D_i is advanced only when a communication event takes place at P_i. We use D_i^x to denote the value of vector D_i when process P_i is in interval θ_i^x. This value is computed at the time e_i^x is executed at process P_i.

Each process P_i executes the following protocol:

1) If P_i belongs to the set of the p processes directly implicated in the conjunction (i.e., $i \le p$):
   - When P_i executes a send event, it advances D_i by setting $D_i[i] := D_i[i] + 1$. P_i then sends the message with D_i.
   - When P_i executes a receive event, where the message contains D_m, it updates D_i by setting 1) $D_i[k] := \max(D_i[k], D_m[k])$ for $1 \le k \le p$ and then 2) $D_i[i] := D_i[i] + 1$.
2) Otherwise ($i > p$):
   - When P_i executes a send event, it includes D_i in the message.
   - When P_i executes a receive event, where the message contains D_m, it updates D_i by setting $D_i[k] := \max(D_i[k], D_m[k])$ for $1 \le k \le p$.

When a process P_i ($1 \le i \le p$) is in interval θ_i^x, the following properties are observed:

1) $D_i^x[i] = x$ and it represents the number of intervals at P_i that precede interval θ_i^x.
2) $D_i^x[j] (j \ne i)$ represents the number of intervals at process P_j that causally precede the interval θ_i^x.
3) The set of intervals $\{\theta_1^{D_i^x[1]}, \theta_2^{D_i^x[2]}, \cdots, \theta_i^{D_i^x[i]}, \ldots, \theta_p^{D_i^x[p]}\}$ is consistent; that is, *each dependency vector represents a consistent cut*.

4) None of the intervals θ_j^y such that $y < D_i^x[j]$ (i.e., intervals at P_j that causally precede $\theta_j^{D_i^x[j]}$) can be concurrent with θ_i^x *as* $\theta_j^y \rightarrow \theta_i^x$. Therefore, none of them can form a set of intervals with θ_i^x that verifies Φ.

Let D_a and D_b be two dependency vector values. We use the following notations.

- $D_a = D_b$ *iff* $\forall i, D_a[i] = D_b[i]$
- $D_a \leq D_b$ *iff* $\forall i, D_a[i] \leq D_b[i]$
- $D_a < D_b$ *iff* $(D_a \leq D_b) \wedge \neg(D_a = D_b)$

The following result holds:

- $\theta_i^x \rightarrow \theta_j^y \Leftrightarrow e_i^{x+1} \preccurlyeq e_j^y \Leftrightarrow D_i^x < D_j^y$

3.1.2 Logs

Each process P_i $(1 \leq i \leq p)$ maintains a log, denoted by Log_i, that is a queue of dependency vector entries. Log_i is used to store dependency vectors representing all consistent cuts that: 1) P_i has detected, 2) are candidates for a solution, but 3) have not been tested globally yet. P_i considers a consistent cut to be a candidate for a solution if P_i*'s* interval in the cut verifies $\mathcal{L}_i$. Thus, when P_i finds that $\mathcal{L}_i$ becomes true while in interval θ_i^x, it enqueues its current dependency vector D_i^x in Log_i if this value has not already been stored.

As the computation progresses (through exchanged messages), each node P_i obtains more information about past intervals of other processes. Based on such information, each cut stored in Log_i is examined or updated.

3.1.3 Cut Vectors

In addition to the dependency vector, each process P_i $(1 \leq i \leq n)$ maintains an integer vector $C_i[1..p]$. Vector C_i represents a cut that P_i is currently checking to see if it verifies Φ. Such a cut is called a *candidate c*ut and is denoted by C_i. We use notation C_i^x to denote a candidate cut right after communication event e_i^x is executed.

The value of vector C_i changes only when a communication event takes place and remains unchanged in an interval. Let $C_i^x[j]$ denote the value of $C_i[j]$ after the communication event e_i^x has been executed at P_i. By definition:

$$C_i^x = \{\theta_1^{C_i^x[1]}, \theta_2^{C_i^x[2]}, \ldots, \theta_j^{C_i[j]}, \ldots, \theta_p^{C_i^x[p]}\}$$

Each P_i maintains vector C_i in such a way that P_i *is certain none of the intervals that precede interval* $\theta_j^{C_i[j]}$ *at* P_j *can form a set of intervals that verifies* Φ. Therefore, each process P_i $(1 \leq i \leq p)$ may discard any values D_i in Log_i such that $D_i[i] < C_i[i]$.

3.1.4 Attestation Vectors

Each P_i $(1 \leq i \leq n)$ also maintains a boolean attestation vector $A_i[1..p]$ in such a way that $A_i[j]$ holds if P_i knows that the interval $\theta_j^{C_i[j]}$ at P_j verifies its local predicate, that is,

$$A_i[j] \Rightarrow \theta_j^{C_i[j]} \models \mathcal{L}_j$$

Thus, if the system is not certain whether the interval verifies its local predicate, $A_i[j]$ is set to false. To disseminate the knowledge what intervals verify their local predicates, cut vector C_i and attestation vector A_i are exchanged among processes. More precisely, when P_i sends a message, it includes vectors C_i, A_i, and D_i in the message. In summary, the following three data structures are managed by each process P_i:

A_i : **array**[1..p] **of boolean** /* Attestation vector */
C_i : **array**[1..p] **of integer** /* Cut vector */
D_i : **array**[1..p] **of integer** /* Dependency vector */

3.2 Descriptions of the Algorithm

A formal description of the algorithm is given in Section 3.3. The algorithm consists of the following three procedures that are executed at a process P_i:

- **procedure** `Local_sat` that is executed each time local predicate $\mathcal{L}_i$ associated with P_i $(1 \leq i \leq p)$ becomes true.
- **procedure** `Sending` that is executed when P_i $(1 \leq i \leq n)$ sends a message.
- **procedure** `Receiving` that is executed when P_i $(1 \leq i \leq n)$ receives a message.

To avoid logging the same vector clock value more than once in Log_i, each process P_i $(1 \leq i \leq p)$ maintains a boolean variable *not_logged_yet*$_i$, which is true iff the vector clock value that is associated with the current interval has not been logged in Log_i.

Procedure `Local_sat` (called when the local predicate $\mathcal{L}_i$ becomes true):

Suppose P_i is currently in interval θ_i^x $(= \theta_i^{D_i^x[i]})$. The current vector clock value D_i^x is logged in Log_i if it has not been logged yet (*not_logged_yet* is true). Then, P_i sets variable *not_logged_yet*$_i$ to false.

Furthermore, if the current interval is in the candidate cut (i.e., $D_i^x[i] = C_i^x[i]$, $A_i^x[i]$ is set to true to indicate that $\theta_i^{C_i^x[i]}$ has satisfied $\mathcal{L}_i$.

Procedure `Sending` (called when P_i sends a message):

Since it is the beginning of a new interval, a process P_i $(1 \leq i \leq p)$ advances the vector clock by setting $D_i[i] = D_i[i] + 1$ and resets variable *not_logged_yet*$_i$ to true.

If the log is empty, none of the intervals that precedes this new interval can be in a solution. Thus, P_i discards all such intervals from consideration for a solution by setting $C_i[i]$ to $D_i[i]$. P_i sets $A_i[i]$ to false since it has not seen $\mathcal{L}_i$ to be verified in this new interval.

Finally, P_i $(1 \leq i \leq n)$ sends the message along with C_i, A_i, and D_i.

Procedure `Receiving` (called when P_i receives a message from P_j that contains D_j, C_j, and A_j):

Since this is also the beginning of a new interval, P_i advances D_i and resets variable *not_logged_yet*$_i$ to true. From the definition of a candidate cut, at any process P_k ($1 \leq k \leq p$), none of the intervals that precedes interval $\theta_k^{C_i[k]}$ or $\theta_k^{C_j[k]}$ can form a solution. Thus, C_i is advanced to the componentwise maximum of C_i and C_j. A_i is updated so that it contains most recent information from either A_i or A_j.

Then, if $i \leq p$, process P_i deletes log values for intervals that precede $\theta_i^{C[i]}$ since it is certain that these intervals cannot be in a solution.

After these operations, there are two possibilities:

Case 1. Log_i becomes empty. In this case, none of the intervals at P_i after $\theta_i^{C_i[i]}$ and before $\theta_i^{D_i[i]}$ can be a part of a solution. Thus, the algorithm needs to consider only future intervals θ_i^z such that $D_i[i] \leq z$.

Since none of the intervals $\theta_k^{y_k}$ such that $y_k < D_i[k]$ at other processes P_k can form a set of concurrent intervals with θ_i^z of P_i, the candidate cut C_i is advanced to D_i. When process P_i executes the receive action, it has no informations about intervals $\theta_k^{D_i[k]}$ ($1 \leq k \leq p$) (because they are concurrent with $\theta_i^{D_i[i]}$. Therefore, all components of vector A_i are set to false.

Case 2. Log_i contains at least one entry that was logged after event $e_i^{C_i[i]}$. Let the oldest such logged entry be D_i^{log}. From the properties of vector D_i and a candidate cut, at any process P_k, $1 \leq k \leq p$, none of the intervals preceding $\theta_k^{D_i^{log}[k]}$ *or* $\theta_k^{C_i[k]}$ can be a part of the solution. Thus, C_i is advanced to the componentwise maximum of D_i^{log} and C_i. Similar to Case 1, if the value $C_i[k]$ is modified (i.e., it takes its value from $D_i^{log}[k]$), P_i is not certain whether P_k *s* local predicate held in the interval $\theta_k^{D_i^{log}[k]}$. Thus, $A_i[k]$ is set to false. If the value $C_i[k]$ remains unchanged, the value $A_i[k]$ also remains unchanged.

Furthermore, since $\theta_i^{D_i^{log}[i]}$ verified its local predicate, $A_i[i]$ is set to true. At this point, P_i checks whether $A_i[k]$ is true for all k. If so, this indicates that each interval in the consistent cut[1] $\{\theta_1^{C_i[1]}, \theta_2^{C_i[2]}, \ldots, \theta_p^{C_i[p]}\}$ verifies its local predicate and thus, Φ is verified.

3.3 A Formal Description of the Algorithm

Procedure `Init` /* executed at initialization by any process P_i

D_i := [0, 0, ⋯, 0]; C_i := [0, 0, ⋯, 0]; A_i := [*false*, *false*, ⋯, *false*];
s**if** ($i \leq p$) **then**
Create(Log_i); *not_yet_logged*$_i$:= true;
endif

Procedure `Local_sat` /* executed by a process P_i when the local predicate $\mathcal{L}_i$ becomes true

if (($i \leq p$) **and** (*not_yet_logged*$_i$)) **then**
Enqueue(Log_i,D_i); *not_yet_logged*$_i$:= *false*;
if ($C_i[i] = D_i[i]$) **then** $A_i[i]$:= *true*; **endif**
endif

Procedure `Sending` /* executed by a process P_i when it sends a message

if ($i \leq p$) **then**
$D_i[i] := D_i[i] + 1$; *not_yet_logged*$_i$:= true;
if *Empty*(Log_i) **then** $C_i[i] := D_i[i]$ **endif**
endif
Append vectors D_i, C_i, and A_i to the message;
Send the message;

Procedure `Receiving` /* executed by a process P_i when it receives a message from P_j

Extract vectors D_j, C_j, and A_j from the message;
D_i := *max*(D_i, D_j);
(C_i, A_i) := *Combine_Maxima*((C_i, A_i),(C_j, A_j));
if ($i \leq p$) **then**
$D_i[i] := D_i[i] + 1$; *not_yet_logged*$_i$:= true;
while ((**not** (*Empty*(Log_i))) **and** (*Head*(Log_i)[i] < $C_i[i]$)) **do**
Dequeue(Log_i);
/* Delete all those logged intervals that form the current
/* knowledge do not lie on a solution.
if (*Empty*(Log_i)) **then**
$C_i := D_i$; A_i := [*false*, *false*, ⋯ *false*];
/* Construct a solution that passes through the next local interval.
else
(C_i, A_i) := *Combine_Maxima*((C_i, A_i), (*Head*(Log_i), [*false*, *false*, ⋯, *false*]))
$A_i[i]$:= true;
/* Construct a solution that passes through the logged interval.
if (A_i = [*true*, *true*, ⋯, *true*]) **then** "Possibly(Φ) *is* verified" **endif**
endif
endif
Deliver the message;

Function *Combine_Maxima* ((*C*1, *A*1), (*C*2, *A*2))
A: *vector* [1..*p*] of boolean;
C: *vector* [1..*p*] of integers;
for j := 1 **to** p **do**
case
$C1[j] > C2[j] \rightarrow C[j] := C1[j]$; $A[j] := A1[j]$;
$C1[j] = C2[j] \rightarrow C[j] := C1[j]$; $A[j] := (A1[j]$ **or** $A2[j])$;
$C1[j] < C2[j] \rightarrow C[j] := C2[j]$; $A[j] := A2[j]$;

1. We will prove in Section 3.6 that the intervals denoted $\theta_1^{C_i[1]}$, $\theta_2^{C_i[2]}$, …, θ_1, and $\theta_p^{C_i[p]}$ are concurrent.

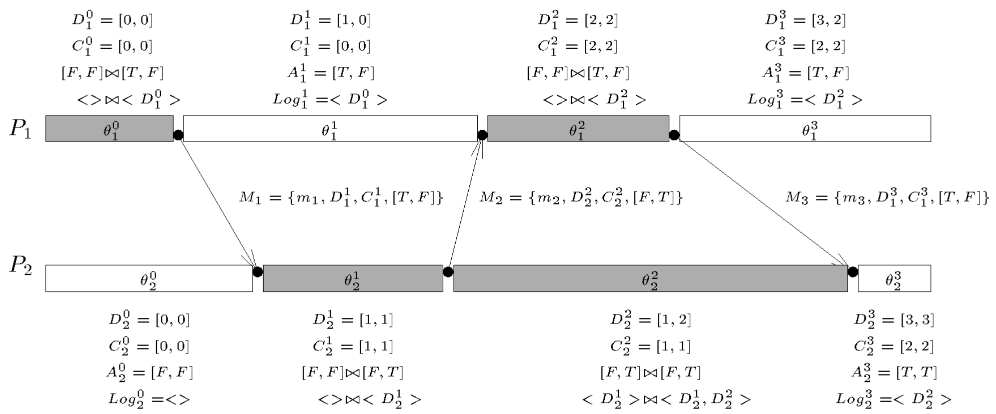

Fig. 3. An example to illustrate the algorithm.

endcase
r**eturn(**C, A**)**;

3.4 An Example

Since the algorithm is quite involved, we illustrate its operation with the help of an example. In Fig. 3, a verified interval is represented by a grey rectangle. We indicate the values of major variables used to detect Possibly($\mathcal{L}_1 \wedge \mathcal{L}_2$) at various points in the computation. The contents of vectors D and C remain unchanged during an entire interval. On the contrary, the contents of both vector A and control variable *Log* may change once during a verified interval. Therefore, their values at the beginning and at the end of every verified interval are depicted (they are separated by the symbol $\bowtie$).

Initial value of the interval number at two processes is 0 and the C [0, 0] at both processes. When the local predicate holds in interval θ_1^0, process P_1 enqueues the D_1 vector into Log_1. Process P_1 also sets $A_1[1]$ to true because it is certain that $\theta_1^{C_1[1]}$ satisfied $\mathcal{L}_1$. When P_1 sends message M_1, it increments $D_1[1]$ to 1 and sends vectors A_1, C_1, and D_1 in the message.

When P_2 receives message M_1, it updates its A, C, and D vectors as follows:

$$D_2 = max(D_2^0, D_1^1) = [1, 0]$$

$$(C_2, A_2) = Combine\_Maxima((C_2^0, [F, F]), (C_1^1, [T, F])) = ([0, 0], [T, F])$$

Then, it increments $D_2[2]$ to 1. P_2 finds its log empty and constructs a potential solution using its D vector and stores it into its C vector:

$$C_2^1 = D_2^1$$

and

$$A_2 = [F, F]$$

When the local predicate becomes true in interval θ_2^1, P_2 adds D_2 to Log_2 and sets $A_2[2]$ to true. When P_2 sends message M_2, it increments $D_2[2]$ to 2 and sends vectors A_2, C_2, and D_2 in the message.

When P_1 receives message M_2, it updates its A, C, and D vectors as follows:

$$D_1 = max(D_1^1, D_2^2) = [1, 2]$$

$$(C_2, A_2) = Combine\_Maxima((C_1^1, [T, F]), (C_2^2, [F, T])) = ([1, 1], [F, T])$$

After incrementing $D_1[1]$ to 2, P_1 finds that $C_1[1]$ (= 1) > $Head(Log_1)[1]$ (= 0) and discards this entry from Log_1. Since Log_1 is empty, P_1 constructs a potential solution using its D vector and stores it into its C vector:

$$C_1^2 = D_1^2$$

and

$$A_1 = [F, F]$$

When the local predicate becomes true in interval θ_1^2, P_1 logs vector D_1 to Log_1 and sets $A_1[1]$ to true. When P_1 sends message M_3, it increments $D_1[1]$ to 3 and sends vectors A_1, C_1, and D_1 in the message.

In the meantime, the local predicate holds in interval θ_2^2 and consequently, P_2 logs vector D_2 to Log_2.

When P_2 receives message M_3, it updates its A, C, and D vectors as follows:

$$D_2 = max(D_2^2, D_1^3) = [3, 2]$$

$$(C_2, A_2) = Combine\_Maxima((C_2^2, [F, T]), (C_1^3, [T, F])) = ([2, 2], [T, F])$$

After merging with the vectors received in the message and incrementing $D_2[2]$ to 3, P_2 finds that $C_2[2]$ (= 2) > $Head(Log_2)[2]$ (= 1) and discards this entry. Since the next entry in Log_2 cannot be discarded, P_2 constructs a potential solution using $Head(Log_2)$ vector (i.e., D_2^2) and stores it into its C and A vectors:

$$C_2, A_2) = Combing\_Maxima(([2, 2], T, F]), (D_2^2, [F, F])) = ([2, 2], [T, F])$$

The potential solution goes through interval θ_1^2 and the fact that this interval satisfies $\mathcal{L}_1$ is known by process P_2 ($A_2[1]$ is true). After P_2 sets $A_2[2]$ to *true*, it finds that all entries of vector A_2 are *true* and declares the verification of the global predicate.

3.5 Extra Messages

The algorithm detects a solution without exchanging any control messages in a completely distributed manner without requiring a centralized process. Basically, it suppresses control messages used in other protocols by requiring application messages to piggyback control information. Since the protocol depends only on computation messages, a solution may not be detected immediately when processes encounter intervals of the first solution. As a consequence, if a solution exists, enough computation messages may not be exchanged to detect the solution before the computation terminates. For example, if the first solution is the cut consisting of the last intervals of the p processes, $\{\theta_1^{l_1}, \ldots, \theta_i^{l_i}, \ldots, \theta_p^{l_p}\}$, the algorithm will not detect it.

To solve this problem, if a solution has not been found when the computation terminates, messages containing vectors D, C, and A are exchanged among the p processes until the first solution is found. To guarantee the existence of at least one solution, we artificially make the set of intervals $\{\theta_1^{l_1+1}, \ldots, \theta_i^{l_i+1}, \ldots, \theta_p^{l_p+1}\}$ a solution. To reduce the overhead caused by these extra messages among processes, a token passing scheme may be used. At the end of computation, if a solution has not be found, one node initiates a special token that circulates around a logical ring consisting of the p processes, disseminating the information about the three vectors. Another solution is to send the token to a process that may know the relevant information (i.e., a process P_j such that $A_i[j]$ is false). Note that this additional computation does not alter the behavior of the application.

3.6 Correctness of the Algorithm

LEMMA 1. *Let V_1 and V_2 be two vector timestamps such that the sets of intervals represented by $\{\theta_1^{V_1[1]}, \theta_2^{V_1[2]}, \ldots, \theta_p^{V_1[p]}\}$ and $\{\theta_1^{V_2[1]}, \theta_2^{V_2[2]}, \ldots, \theta_p^{V_2[p]}\}$ are both concurrent. Then, the set of intervals represented by $\{\theta_1^{V_3[1]}, \theta_2^{V_3[2]}, \ldots, \theta_p^{V_3[p]}\}$ is concurrent, where $V_3[i] = \max(V_1[i], V_2[i])$ for $1 \le i \le p$.*

PROOF. We show that a pair of intervals $(\theta_i^{V_3[i]}, \theta_j^{V_3[j]})$ for any combination of i and j, $1 \le i, j \le p$, and $i \ne j$ is concurrent. Renumbering the vectors V_1 and V_2 if necessary, suppose $V_3[i] = V_1[i]$. There are two cases to consider:

1) $V_3[j] = V_1[j]$. This case is obvious because, from the assumption, $\theta_i^{V_1[i]}$ and $\theta_j^{V_1[j]}$ are concurrent.

2) $V_3[j] = V_2[j]$. Suppose on the contrary $\theta_i^{V_3[i]}$ and $\theta_j^{V_3[j]}$ are not concurrent. There are two cases to consider:

- $\theta_j^{V_3[j]} \rightarrow \theta_i^{V_3[i]}$ ($\equiv \theta_j^{V_2[i]} \rightarrow \theta_i^{V_1[i]}$). In this case, $e_j^{V_2[j]+1} \preccurlyeq e_i^{V_1[i]}$. Since $V_1[j] \le V_2[j]$, this implies $e_j^{V_1[j]+1} \preccurlyeq e_i^{V_1[i]}$ and so $\theta_j^{V_1[j]} \rightarrow \theta_i^{V_1[i]}$. This contradicts the assumption that $\theta_j^{V_1[j]}$ and $\theta_i^{V_1[i]}$ are concurrent.
- $\theta_i^{V_3[i]} \rightarrow \theta_j^{V_3[j]}$ ($\equiv \theta_i^{V_i[i]} \rightarrow \theta_j^{V_2[j]}$). By applying the same argument as before, we have a contradiction to the assumption that $\neg(\theta_j^{V_2[i]} \rightarrow \theta_j^{V_2[j]})$.

Thus, $\theta_i^{V_3[i]}$ and $\theta_j^{V_3[j]}$ are concurrent. □

The following lemma guarantees that a cut C_i always keeps track of a set of concurrent intervals.

LEMMA 2. *At any process P_i, at any given time, a set of intervals $\{\theta_1^{C_i[1]}, \theta_2^{C_2[2]}, \ldots, \theta_p^{C_i[p]}\}$ is concurrent.*

PROOF. C_i is updated only in one of the following three ways:

Case 1. When a send event occurs, entry $C_i[i]$ is set to $D_i[i]$.

Case 2. When a receive event is executed and log_i is empty, C_i is set to D_i.

Case 3. When a receive event is executed and log_i is not empty, by taking maximum of C_i and C_j (the cut contained in the message sent by process P_j), and then by taking maximum of C_i and D_i^{log} (the oldest value of the dependency vector still in the log).

We prove the lemma by induction on the number of intervals that causally precede the current interval (i.e., intervals θ_j^y such that $\theta_j^y \rightarrow \theta_i^x$).

Induction Base. Since the cut $\{\theta_1^0, \theta_2^0, \ldots, \theta_p^0\}$ is consistent, initially the lemma holds for any C_i^0 (i.e., in any interval causally preceded by no intervals).

Induction Hypothesis. Assume that the lemma holds for any interval causally preceded by at most t intervals.

Induction Steps. Suppose that $t + 1$ intervals causally precede θ_i^x. The following three cases correspond to the three cases in which C_i is updated (discussed above):

Case 1. e_i^x is a send event and $C_i^x[i]$ is updated to $D_i^x[i]$. Thus, $C_i^x[i] = D_i^x[i]$ and $\forall j$ such that $j \ne i$, $C_i^x[j] = C_i^{x-1}[j]$.

Suppose no receive event occurs at process P_i before the send event e_i^x. All the entries of vector D_i^x and vector C_i^x, except $D_i^x[i]$ and $C_i^x[i]$, are zero. Therefore, $C_i^x = D_i^x$. Since any dependency vector D_i represents a consistent cut (i.e., the set of p intervals $\{\theta_1^{D[1]}, \theta_2^{D[2]}, \ldots, \theta_p^{D[p]}\}$ is consistent), C_i represents a consistent cut.

Suppose now that e_i^y is the last receive event which occurred before e_i^x. Because logged entries are discarded only when *a* receive event occurs, no entry in Log_i has been discarded since e_i^y occurred (and the corresponding algorithm for a receive event was executed). Since $C_i[i]$ is updated to $D_i[i]$ only when Log_i is empty, Log_i is empty when e_i^x occurs. Therefore, we can conclude that the log was also empty when e_i^y occurred and hence, $C_i^y = D_i^y$ after the receive algorithm was executed. As e_i^y is the last receive event that occurs before e_i^x, we conclude that $\forall j$ such that $j \neq i$, $D_i^y[j] = D_i^x[j]$, and $C_i^y[j] = C_i^x[j]$. Therefore, $C_i^x = D_i^x$, and C_i represents a consistent cut.

Case 2. e_i^x is a receive event and C_i^x is set to D_i^x (Log_i is empty at e_i^x).

Since any D_i represents a consistent cut, C_i^x represents a consistent cut.

Case 3. e_i^x is a receive event and $C_i^x = max(max(C_i^{x-1}, C_j^y), D_i^{log})$.

Since at most $t + 1$ intervals causally precede θ_i^x, at most t intervals causally precede θ_i^{x-1} and θ_j^{y-1} (recall that e_j^y is the corresponding send event). Thus, from the induction hypothesis, both C_i^{x-1} and C_j^{y-1} represent consistent cuts. Furthermore, C_j^y represents a consistent cut since if $C_j^y \neq C_j^{y-1}$, Case 1 guarantees that C_j^y is also consistent.

Thus, from Lemma 1, $max(C_i^{x-1}, C_j^y)$ represents a consistent cut. From the definition of vector clocks, D_i^{log} (the oldest entry still in the log) represents a consistent cut. Therefore, C_i^x represents a consistent cut (from Lemma 1). □

The following lemma shows that if there is a solution, a cut C_i will not pass beyond the solution, even though D_i has passed the solution (i.e., $\neg(C^f \leadsto C_i)$ is an invariant).

LEMMA 3. *Consider a particular solution C^s identified by an integer vector S: $C^s = \{\theta_1^{S[1]}, \ldots, \theta_p^{S[p]}\}$. Let θ_i^x denote any interval. If $C_j^y \leq S$ for all intervals θ_j^y such that $\theta_j^y \rightarrow \theta_i^x$, then $C_i^x \leq S$.*

PROOF. Proof is by contradiction. Suppose there exists an interval θ_i^x such that $\neg(C_i^x \leq S)$, and for any interval θ_j^y such that $\theta_j^y \rightarrow \theta_i^x$, $C_j^y \leq S$. That is, e_i^x is the first communication event that advances C_i beyond S.)

There are two cases to consider:

1) $C_i^x[i] \leq S[i]$ and $S[j] < C_i^x[j]$ for some $j \neq i$. From hypothesis, $C_i^{x-1}[j] \leq S[j]$ holds. Therefore, entry $C_i[j]$ is modified during execution of event e_i^x. This event is necessarily a receive event since only a receive event may advance the jth entry of C_i ($j \neq i$). Let e_k^z be the corresponding send event.

Case 1. $C_i^x = D_i^x$. Note that $\theta_i^{C_i^x[i]}$, $\theta_i^{D_i^x[i]}$, and θ_i^x denote the same interval. Since $S[j] < D_i^x[j]$ holds, from the definition of dependency vectors, $\theta_j^{S[j]} \rightarrow \theta_i^x$ holds. However, either $(\theta_i^{C_i^x[i]} = \theta_i^{S[i]})$ *or* $(\theta_i^{C_i^x[i]} \rightarrow \theta_i^{S[i]})$ and therefore, $\theta_j^{S[j]} \rightarrow \theta_i^{S[i]}$. This contradicts the hypothesis that $\{\theta_1^{S[1]}, \ldots, \theta_p^{S[p]}\}$ is a set of concurrent intervals.

Case 2. $C_i^x = max(max(C_i^{x-1}, C_k^z), D_i^{\log})$. From the assumption, $C_i^{x-1} \leq S$ *and* $C_k^z \leq S$. Therefore, $C_i^x[j] = D_i^{log}[j]$. Since $S[j] < D_i^{log}[j]$ holds, from the definition of dependency vectors, $\theta_j^{S[j]} \rightarrow \theta_i^{D_i^{log}[i]}$ holds. Because of the *max* operation, $D_i^{log}[i] \leq C_i^x[i]$, and so $\theta_j^{S[j]} \rightarrow \theta_i^{C_i^x[i]}$. Then, the above Case 1 applies.

2) $C_i^x[i] > S[i]$.

Case 1. $C_i^x[i] = D_i^x[i]$. e_i^x is either a send or a receive event. From the algorithm, it is clear that this case occurs only if none of the intervals that occurred between $\theta_i^{C_i^{x-1}[i]}$ (including this) and $\theta_i^{D_i^x[i]}$ verifies $\mathcal{L}_i$. This contradicts the fact that $\theta_i^{S[i]}$ verifies $\mathcal{L}_i$ since $C_i^{x-1}[i] \leq S[i] < C_i^x[i] (= D_i^x[i])$.

Case 2. $C_i^x[i] = max(max(C_i^{x-1}[i], C_k^z[i]), D_i^{log}[i]$. e_i^x is a receive event and e_k^z is the corresponding send event. From the assumption, $C_i^{x-1}[i] \leq S[i]$ and $C_k^z[i] \leq S[i]$. Thus, $max(C_i^{x-1}[i]), C_k^z[i] \leq S[i]$ and therefore $C_i^x[i] = D_i^{log}[i]$.

From the algorithm, it is clear that this case occurs only if none of the intervals that occurred between $\theta_i^{max(C_i^{x-1}[i], C_k^z[i])[i]}$ (including this) and $\theta_i^{D_i^{log}[i]}$ verifies $\mathcal{L}_i$. This contradicts the fact that $\theta_i^{S[i]}$ verifies $\mathcal{L}_i$ since $max(C_i^{x-1}[i], C_k^z[i]) \leq S[i] < C_i^x[i] (= D_i^{log}[i])$. □

The following lemma proves that the algorithm keeps making progress if it has not encountered a solution.

LEMMA 4. *Suppose process P_i has just executed the algorithm at the xth communication event e_i^x (i.e., P_i is in the interval θ_i^x) and that the set $\{\theta_1^{C_i^x[1]}, \ldots, \theta_i^{C_i^x[i]}, \ldots, \theta_p^{C_i^x[p]}\}$ does not verify Φ. Then, there exists an event e_j^y such that $C_i^x < C_j^y$.*

PROOF. There are two reasons for $\{\theta_1^{C_i^x[1]}, \ldots, \theta_i^{C_i^x[i]}, \ldots, \theta_p^{C_i^x[p]}\}$ not verifying Φ:

1) $\theta_i^{C_i^x[i]}$ does not verify $\mathcal{L}_i$. In this case, at e_i^x, P_i could not find an interval $\mathcal{L}_i$, and therefore, $C_i^x[i]$ was set to x (i.e., the value of $D_i^x[i]$). At the next communication event e_i^{x+1}, P_i updates the ith entry of C_i by setting $C_i^{x+1}[i]$ *to* $D_i^{x+1}[i]$. Thus, $C_i^x < C_i^{x+1}$.

2) There exists at least one process P_k ($1 \le k \le p$) such that $\theta_k^{C_i^x[k]}$ does not verify $\mathcal{L}_k$. In this case, P_k will eventually advance $C_k[k]$ to a value greater than $C_i^x[k]$ (refer to Case 1). This new value will propagate to other processes.[2] Extra messages exchanged at the end of the computation guarantee that there will eventually be a communication event e_j^y at a process P_j such that $e_k^z \prec e_j^z$ and $C_i^x < C_j^y$. □

Finally, the following theorem shows that Φ is verified in a computation iff the algorithm detects a solution.

THEOREM.

1) *If there exists an interval θ_i^x on a P_i such that during this interval, $A_i[k]$ holds for all k ($1 \le k \le p$), then $\{\theta_1^{C_i^x[1]}, \ldots, \theta_p^{C_i^x[p]}\}$ verifies Φ.*

2) *Conversely, if $\{\theta_1^{C_i^x[1]}, \ldots, \theta_p^{C_i^x[p]}\}$ verifies Φ for an event e_i^x on some processor P_i, then there exists a communication event e_j^y such that for all k ($1 \le k \le p$), $C_j^y[k] = C_i^x[k]$ and $A_j[k]$ holds.*

PROOF.

1) Proof is by contradiction. Suppose $\theta_k^{C_i^x[k]}$ does not verify $\mathcal{L}_k$ for some k. We show that as long as $C_i[k]$ is not changed, $A_i[k]$ is false. There are two cases to consider:

- $i = k$ (i.e., $\theta_i^{C_i^x[i]}$ does not verify $\mathcal{L}_i$). $C_i[i]$ is updated to $D_i[i]$ when communication event e_i^x occurs. *If* e_i^x is a receive event, $A_i^x[i]$ is set to false at the same time and remains false during the interval θ_i^x. Log_i is necessarily empty for the entire duration of interval $\theta_i^{C_i^x[i]}$. $C_i[i]$ is modified only when event e_i^{x+1} occurs.

  If e_i^x is a send event, the value of $A_i[i]$ is unchanged since the last receive event or since the beginning of the computation if no receive event occurs at process P_i before the send event e_i^x. In both cases, Log_i remains empty during this entire period and $A_i[i]$ remains false.

- $i \ne k$. P_i updates $C_i[k]$ to $C_i^x[k]$ because there existed a process P_j that advanced $C_j[k]$ to $C_i^x[k]$, and the value was propagated to P_i. P_j must have set $A_j[k]$ to false and this information must have propagated to P_i. This value was propagated to P_i without going through P_k (else P_k would have advanced $C_k[k]$ to a value greater than $C_i^x[k]$). It is easy to see that $A_i^x[k]$ is false: Since P_k is the only process that can change $A_k[k]$ to true, P_i will never see $A_i[k]$ = true together with $C_i[k] = C_i^x[k]$.

2) Assume that $\{\theta_1^{C_i^x[x]}, \ldots, \theta_p^{C_i^x[p]}\}$ verifies Φ. Message exchanges guarantee that there will eventually be a communication event e_j^y such that for all k, $1 \le k \le p$, $e_k^{C_i^x[k]+1} \preccurlyeq e_j^y$. When process P_k is in interval $\theta_k^{C_i^x[k]}$, $A_k[k]$ is set to true.

   From Lemma 3, once a process P_h sets $C_h[k]$ to $C_i^x[k]$, it does not change this value in the future. This implies that all the processes P_h that are on the path of the message exchange from $e_k^{C_i^x[k]+1}$ to e_j^y, set $C_h[k]$ to $C_i^x[k]$ and $A_h[k]$ to true—none of such processes P_h sets $A_h[k]$ to false by advancing $C_h[k]$ beyond $C_i^x[k]$. Thus, all information is eventually propagated to P_j and $A_j^y[k]$ holds for all k, $1 \le k \le p$. □

2. The following property is obviously satisfied $:\forall \theta_k^z, \forall \theta_j^y, \theta_k^z \to \theta_j^y \Rightarrow C_k^z \le C_j^y$.

4 COMPARISON WITH RELATED WORK

Cooper-Marzullo's algorithm [3] addresses the detection of global predicates that are more general than conjunctive form global predicates. This algorithm explicitly constructs and checks every possible consistent cut. Consequently, its time complexity can be exponential. Since the proposed algorithm only deals with specialized global predicates, for fairness, we do not compare it with Cooper-Marzullo algorithm.

The predicate detection algorithms proposed by Manabe-Imase [15] and Venkatesan-Dathan [20] perform offline predicate evaluation. Manabe-Imase algorithm uses execution replay techniques: during an initial execution, useful information is logged and is then used during subsequent executions to guide them and enforce a deterministic behavior. Such replayed executions are immune to probe effects and are then perfect subject for a detailed analysis. In particular, the detection of a conjunctive form predicate during a replay of the execution can be done by carefully controlling the progress of each process in order to obtain a consistent cut satisfying the predicate. Venkatesan-Dathan algorithm uses a different approach. During an initial execution, each process locally records information that is relevant to predicate detection. After this run, this information is used to evaluate the predicate (note that this information is not used to obtain executions equivalent to the original one). They use a notion, called a spectrum, that is very similar to the interval notion. A spectrum is as a sequence of consecutive events at the same process. They also use the notion of spectra which is similar to the notion of verified interval introduced in Section 2. They proposed two detection algorithms (a sequential and a distributed) to analyze the recorded information and to detect if events that belong to different spectra could have coexisted during the analyzed computation.

In this section, we focus only on works that address online evaluation of conjunctive form global predicates. We compare our algorithm with such algorithms with respect to message, storage, and computational complexities.

4.1 Garg-Waldecker's Algorithm

Garg and Waldecker presented a centralized algorithm [7]. Unlike our algorithm, an interval in this algorithm is a segment of time between two consecutive send operations. The algorithm assumes the existence of a centralized checker process. A process reports every interval that satisfies its local predicate to the checker process. Based on this information, the checker process constructs possible combinations of sets of verified intervals and checks whether each set is consistent.

More precisely, each process maintains a dependency vector. A process increments its own component of the vector by one when it executes a send event. When a process executes a receive event, it updates its vector by taking the component-wise maximum of its vector and the one contained in the message. Note that since an interval is defined to be a segment between two consecutive send operations, its own component of the vector clock is not incremented at a receive event. When a process detects its local predicate to become true for the first time since the last send event, it sends a control message carrying its current dependency vector to the checker process. The checker process maintains a separate queue of dependency vectors for each process involved in the predicate. The dependency vector carried in an incoming control message is enqueued in the associated queue. The checker process compares dependency vectors at the heads of the queues. If one dependency vector is less than another, the smaller one is dequeued and discarded (these intervals are not concurrent). The global predicate is verified if none of the dependency vectors at the heads are less than the others (i.e., the intervals represented by the dependency vectors are consistent).

This algorithm has the extra control message complexity of $O(M_s * p)$, where M_s is the maximum number of messages sent by any one process in the computation and p is the number of processes over which the global predicate is defined. The checker process maintains a queue of vectors for each process. It stores at most the number of vectors that it has received, and each vector consists of p integers. Thus, the storage complexity of the checker process is $O(M_s * p^2)$. Garg and Waldecker demonstrated that the checker processes computational complexity (the number of integer comparisons) is $O(M_s * p^2)$.

Garg and Waldecker presented a method to decentralize the above algorithm [7]. Processes involved in the conjunction are divided into g groups. Likewise, the conjunction of local predicates is divided into g subconjunctions (one per group). In addition to the centralized checker process, each group is assigned a local checker process. Within a group, its local checker process is responsible for detecting a solution to the corresponding subconjunction. Once such a solution is detected, the checker process sends the information to the centralized checker process. The centralized checker process checks for consistency across the groups. As indicated by Garg and Chase [8], this approach has a disadvantage that local checker processes may have to send huge number of control messages to the centralized checker process.

4.2 Garg-Chase's First Algorithm

In [8], Garg and Chase presented two distributed algorithms for detection of conjunctive form predicates. In both algorithms, all processes equally participate in the global predicate detection. Like our algorithm, an interval is defined to be a segment of time between two consecutive communication events.

In the first algorithm, each process maintains a dependency vector to keep track of the dependency relations among intervals (i.e., a process updates its own component when it sends and receives messages). A token is circulated to carry information about the latest global cut of verified intervals. When a process receives the token, it checks whether the cut represents concurrent intervals. If so, it claims detection of a solution; otherwise, it updates the cut based on its own information and forwards the token to the next process.

Let M_c be the maximum number of communication events at any one process. Garg and Chase showed that the message complexity of this algorithm is $O(M_c * p)$ since there are at most $M_c * p$ different intervals and a process eliminates at most one interval in the cut upon a receipt of the token. The storage and computational complexities are computed as follows: Each process logs a vector clock of p integers each time when the interval is verified. Thus, each process may have at most $M_c * p$ log entries, and the total storage complexity is $O(M_c * p^2)$. At each node, p integer comparisons are performed to eliminate an interval. Since there are at most $M_c * p$ intervals in the system, the computational complexity is $O(M_c * p^2)$. Thus the storage and computational complexities are $O(M_c * p^2)$, which is the same as that of Garg-Waldecker's [7] algorithm if $O(M_c) = O(M_s)$.

Furthermore, the message complexity per process is $O(M_c)$ and the storage and computational complexity per process is $O(M_c * p)$. Thus, the load distribution is equitable among all p processes. However, the predicate detection is still done sequentially since only a process with the token can update the cut. In order to introduce parallelism, the authors suggested the use of multiple tokens. In this approach, the processes are partitioned into groups, and the existence of a leader process is assumed. One token is assigned to each group and the leader process coordinates the activities of all the groups.

4.3 Garg-Chase's Second Algorithm

Garg and Chase's second algorithm [8] uses logical counters, instead of vector clocks, to identify direct dependency relations between intervals. Each process identifies its current interval by the value of its counter. That is, the counter is incremented when the process enters a new interval. The cut is not maintained by any single process. Instead, it is maintained in a distributed manner by the p processes; that is, each process P_i maintains only P_i s interval for the cut (here, we call P_i s interval for the cut P_i 's cut interval). Along with its cut interval, each process maintains a log. In a log, a process stores its verified intervals, which are future candidates for its cut interval. For each candidate cut interval in the log, a process also maintains a list of intervals at other processes that causally precede the interval.

When a process P_j sends a control message to P_i, it includes the current counter value, say d_j. When P_i receives the message, it stores (j, d_j) in the log. This indicates that the interval denoted by d_j at P_j directly precedes the next verified interval at P_i. As in the first algorithm, the logged information is used to eliminate some of its candidate intervals.

A process updates its cut if it detects that its cut causally precedes other processes' cuts. When a process P_i updates its cut, it checks whether there are direct dependencies between previous intervals of other processes and P_i's new candidate cut interval. This is done by checking whether there are entries, say (j, d_j), in the log, associated with P_i's new candidate. If such entries are found, control messages carrying values d_j are sent to all such processes P_j. Upon the receipt of control message (j, d_j), process P_j checks whether its current cut interval is d_j or precedes d_j. If so, P_j updates its cut interval and sends control messages to other processes if necessary. This process is repeated until all outstanding control messages have been received and processed. To make sure that no control message is in transit, a termination detection algorithm is used. If the termination is detected, the first solution is found. The cut is obtained by combining all the cut intervals of the p processes involved in the conjunction. In this algorithm, all n processes exchange control messages. A process that is not involved in the conjunction considers that every interval of the process is verified. Note that a control message does not carry the value of the cut (i.e., *dependency vector*); it carries only one integer value.

Message, storage, and computational complexities of Garg-Chase's algorithm are $O(M_s * n)$. These complexities are computed as follows: A direct causal relation is created by a message, and at most $M_s * n$ direct causal relations exist. Therefore, at most $M_s * n$ integer values are stored in all logs, and at most $M_s * n$ control messages are initiated. Note that the termination detection requires $O(n)$ messages. Upon a receipt of one control message, constant amount of work is performed. Thus, the computational complexity is $O(M_s * n)$. As far as the message complexity is concerned, their first algorithm may be desirable over this algorithm since $p \leq n$. However, this algorithm is desirable for the storage and computational complexities over the first algorithm when p^2 is greater than n.

4.4 Our Algorithm

Unlike other algorithms, the algorithm presented in this paper requires no additional control messages (except for messages exchanged after a computation terminates without detecting a solution). Control information is piggybacked on computation messages. Thus, the control message complexity is 0. In addition to minimizing the message traffic, this leads to a major advantage in reliability: even if application programs run on unreliable network environments, as long as they tolerate message loss, the algorithm captures causal relations correctly and detects a solution. Note that in algorithms that require control messages, if a control message is lost, the detection algorithms may not work.

In our algorithm, the volume of control information that an application message carries is bounded by O(p) integers (that includes integer vector C and boolean vector A). This applies to messages initiated not only by processes involved in the conjunction, but also by processes not involved in the conjunction. Thus, the worst case volume of control information exchanged among processes is $O(M_s * p * n)$. Since $p \leq n$, this volume may be greater than that of the Garg and Chase's first algorithm. However, a study by Lazowska et al. [14] showed that the overhead caused by sending and receiving messages can be considerably large due to context switching and execution of multiple communication protocol layers, and that it is desirable to exchange fewer bigger messages from a performance point of view. The maximum depth of the queue at each process is bounded by M_c. The size of each queue entry is $O(p)$, and there are p processes that maintain their queues. Thus, the total storage complexity is $O(M_s * p^2)$. At each process, the number of integer comparisons is bounded by the number of entries in its queue. Thus, the total computational complexity is $O(M_s * p^2)$.

Note that in the above comparisons of complexity, extra information carried in messages to keep track of the causal dependencies of intervals is not taken into account. A vector clock mechanism requires that all n processes include a vector clock of size p to every message they send (global cost is $O(Mnp)$). This additional cost was not considered in the Garg and Waldecker algorithm and the Garg and Chase first algorithm. In our algorithm, the order of the amount of the extra control information $O(M_s * p * n)$ is not affected by this additional cost.

Thus, our algorithm detects a solution without exchanging control messages in a completely distributed manner; control messages may need be transferred only in situations where a solution has not been detected before the computation terminates. The algorithm piggybacks the control information on application messages. In addition, this algorithm has a major advantage that it allows the system to tolerate message losses (as a lost message does not create any causal dependencies). However, there is a tradeoff: since the algorithm depends on computation messages for the advancement of the knowledge about the system state, a solution may not be detected immediately when processes encounter intervals of the first solution, thereby, possibly increasing the latency in the solution detection. In the worst case, if the predicate becomes true and sufficient application messages are not transferred to allow the "candidate cut" to progress, the application could terminate before the global predicate has been detected. We have discussed how to handle such situations in Section 3.5. The problem of increased latency in detection can be overcome by periodically sending a few control messages that allow a process to be informed that its own component (of its potential solution) has been discarded by other processes. Such control messages will reduce the latency of the detection by doing faster dissemination of the knowledge about the system state. Note that control messages can get lost without affecting the safety and the liveness of the detection; they will only expedite the detection.

5 Concluding Remarks

Global predicate detection is a fundamental problem in the design, coding, testing and debugging, and implementation of distributed programs. In addition, it finds applications in many other domains in distributed systems such as deadlock detection and termination detection.

This paper presented an efficient distributed algorithm to detect conjunctive form global predicates in distributed systems. The algorithm detects the first consistent global state that satisfies the predicate and works even if the predicate is unstable.

The algorithm is distributed because the predicate detection efforts as well as the necessary information are equally distributed among the processes. Unlike previous algorithms to detect conjunctive from global predicates, our algorithm does not require exchange of any control messages during the normal computation; instead, it piggybacks all the control information on computation messages. Additional messages are exchanged only if the predicate remains undetected when the computation terminates. The proposed algorithm compares favorably with the existing algorithms in terms of storage and computational complexities.

Acknowledgments

The authors would like to thank three anonymous referees whose comments were helpful in improving the presentation of the paper. Michel Hurfin was supported, in part, by an INRIA grant (postdoctoral fellowship). M. Mizuno was supported, in part, by the National Science Foundation under Grant No. CCR-9201645 and Grant No. INT-9406785. Mokesh Singhal was supported, in part, by an INRIA grant when he was visiting IRISA.

References

[1] Ö. Babaoğlu and M. Raynal, "Specification and Verification of Dynamic Properties in Distributed Computations," *J. Parallel and Distributed Computing*, vol. 28, no. 2, pp. 173–185, Aug. 1995.

[2] Ö. Babaoğlu, E. Fromentin, and M. Raynal, "Unified Framework for Expressing and Detecting Run-Time Properties of Distributed Computations," *J. Systems and Software*, special issue on Software Eng. for Distributed Computing, vol. 33, no. 3, pp. 287–298, June 1996.

[3] R. Cooper and K. Marzullo, "Consistent Detection of Global Predicates," *Proc. ACM/ONR Workshop Parallel and Distributed Debugging*, pp. 163-173, Santa Cruz, Calif., May 1991.

[4] C.J. Fidge, "Timestamps in Message Passing Systems that Preserve the Partial Ordering," *Proc. 11th Australian Computer Science Conf.*, pp. 55–66, Feb. 1988.

[5] E. Fromentin, M. Raynal, V.K. Garg, and A.I. Tomlinson, "On the Fly Testing of Regular Patterns in Distributed Computations," *Proc. 23rd Int'l Conf. Parallel Processing*, pp. 73–76, St. Charles, Ill., Aug. 1994.

[6] V.K. Garg and B. Waldecker, "Detection of Unstable Predicates in Distributed Programs," *Proc. 12th Int'l Conf. Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, *Lecture Notes in Computer Science 652*, pp. 253–264, New Delhi, India, Dec. 1992.

[7] V.K. Garg and B. Waldecker, "Detection of Weak Unstable Predicates in Distributed Programs," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 3, pp. 299–307, Mar. 1994.

[8] V.K. Garg and C.M. Chase, "Distributed Algorithms for Detecting Conjunctive Predicates," *Proc. 15th Int'l Conf. Distributed Computing Systems*, pp. 423-430, Vancouver, Canada, June 1995.

[9] D. Haban and W. Weigel, "Global Events and Global Breakpoints in Distributed Systems," *Proc. 21st Hawaiin Int'l Conf. System Sciences*, pp. 166–175, Jan. 1988.

[10] M. Hurfin, N. Plouzeau, and M. Raynal, "Detecting Atomic Sequences of Predicates in Distributed Computations," *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 32–42, San Diego, Calif., May 1993. reprinted in *SIGPLAN Notices*, vol. 28, no. 12, Dec. 1993

[11] M. Hurfin and M. Mizuno, "A Complete Set of Satisfaction Rules for Property Detection in Distributed Computations," *Proc. Ninth Int'l Conf. Parallel and Distributed Computing Systems (PDCS'96)*, pp. 156–163, Dijon, France, Sept. 1996.

[12] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal, "Efficient Distributed Detection of Conjunction of Local Predicates," Technical Report 2,731, INRIA, Nov. 1995.

[13] R. Jegou, R. Medina, and L. Nourine, "Linear Space Algorithm for On-Line Detection of Global States," *Proc. Int'l Workshop on Structures in Concurrency Theory*, 1995.

[14] E.D. Lazowska, J. Zahorjan, D. Cheriton, and W. Zwaenepoel, "File Access Performance of Diskless Workstations," *ACM Trans. Computer Systems*, vol. 4, no. 3, pp. 238–268, Aug. 1986.

[15] Y. Manabe and M. Imase, "Global Conditions in Debugging Distributed Programs," *J. Parallel and Distributed Computing*, vol. 15, pp. 62–69, 1992.

[16] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Proc. Int'l Workshop Parallel and Distributed Algorithms*, North-Holland, pp. 215-226, Chateau de Bonas, France, Oct. 1988.

[17] B.P. Miller and J.-D. Choi, "Breakpoints and Halting in Distributed Programs," *Proc. Eighth IEEE Int'l Conf. Distributed Computing Systems*, pp. 316–323, San Jose, Calif., June 1988.

[18] M. Raynal and M. Singhal, "Logical Clocks: A Way to Capture Causality in Distributed Systems," *Computer*, vol. 29, no. 2, pp. 49-56, Feb. 1996.

[19] S.D. Stoller and F.B. Schneider, "Faster Possibility Detection by Combining Two Approaches," *Proc. Ninth Int'l Workshop Distributed Algorithms*. Springer-Verlag, *Lecture Notes in Computer Science 972*, pp. 318–332, Le Mont-Saint-Michel, France, Sept. 1995.

[20] S. Venkatesan and B. Dathan, "Testing and Debugging Distributed Programs Using Global Predicates, *IEEE Trans. Software Eng.*, vol. 21, no. 2, pp. 163-177, Feb. 1995.

Michel Hurfin received the PhD degree in computer science from the University of Rennes I, France in 1993. His dissertation topic addressed execution replay and property detection in distributed systems. During 1994, he continued his research on property detection at Kansas State University, Manhattan. Dr. Hurfin is currently a researcher at the INRIA research unit of Rennes. His primary research interests include distributed systems, software engineering, and operating systems. Recently, he initiated research on fault-tolerant mechanisms in open distributed systems.

Masaaki Mizuno received the BS and MS degrees in electrical engineering from Keio University, Japan, and the MS degree in computer science from Pennsylvania State University. In 1987, he received the PhD degree in computer science from Iowa State University. Since then, he has been with the Department of Computing and Information Sciences at Kansas State University, where he is currently an associate professor. His research interests are in operating systems and distributed computing systems.

Michel Raynal received his "doctorat d'Etat" in computer science from the University of Rennes, France, in 1981. In 1981 he moved to Brest (France) where he founded the Computer Science Department in a telecommunications engineer school (ENST). In 1984, he moved to the University of Rennes, where he has been a professor of computer science. At IRISA (CNRS-INRIA-University joint computing research laboratory located in Rennes), he is leading the research group "Algorithmes distribués et protocoles" (Distributed Algorithms and Protocols). His research interests include distributed algorithms, operating systems, computing systems, protocols, and dependability. His main interest lies in the fundamental principles that underly the design and construction of distributed systems. He has been a principal investigator for a number of research grants in these areas (grants from French private companies, France Telecom, CNRS, NFS-INRIA, and ESPRIT BRA).

Raynal has published more than 50 papers in journals and more than 100 papers in conferences. He has also written seven books devoted to parallelism, distributed algorithms and systems. Among them: *Distributed Computations and Networks* (MIT Press, 1988) and *Synchronization and Control of Distributed Programs* (John Wiley & Sons, 1990). Raynal has been invited to serve on program committees for more than 40 international conferences. In 1995, he was program chair of the WDAG workshop and of the IEEE international conference on Future Trends of Distributed Computing Systems. He has been invited by many universities to give lectures about operating systems and distributed algorithms in Europe, South American, North America, Asia, and Africa. He serves as an editor for two international journals. Raynal is currently a member of the executive board of CABERNET (ESPRIT Network of Excellence in Distributed Computing Architectures).

Mukesh Singhal received his PhD degree in computer science from the University of Maryland, College Park, in May 1986 for work on the design and analysis of concurrency control algorithms for replicated databases. Since February 1986, he has been with the Department of Computer and Information Science at Ohio State University, Columbus, where he is currently an associate professor. His current research interests include operating systems, distributed systems, computer security, mobile computing, high-speed networks, databases, and performance modeling. He has published over 40 papers in refereed journals and over 70 papers in refereed conferences in these areas. He has co-authored two books *Advanced Concepts in Operating Systems*, (McGraw-Hill, 1994) and *Readings in Distributed Computing Systems*, (IEEE CS Press, 1993). He has served on the program committees of several international conferences and symposiums. He was the program co-chair of the Sixth International Conference on Computer Communications and Networks, 1997 (IC3N'97) and is the program chair of the IEEE 17th Symposium on Reliable Distributed Systems, 1998.