# Message Ordering based on Colorful Forbidden Predicates

*V. V. Murty  and  V. K. Garg*

**TR  ECE-PDS-97-005  April  1997**

**Parallel & Distributed Systems group**

**Department of Electrical & Computer Engineering**

**University of Texas at Austin**

**Austin, Texas 78712**

# Message Ordering based on Colorful Forbidden Predicates

V. V. Murty and V. K. Garg
Department of Electrical & Computer Engineering
University of Texas at Austin
email: {murty, vijay}@pine.ece.utexas.edu

**Abstract**

In this paper we study a class of message ordering guarantees that can be implemented by tagging information to the user messages and without any control messages. To specify the message ordering, we use the method called *forbidden predicates* introduced in [13]. In this paper we extend the concept of forbidden predicates by defining three attributes for each message, that is, color, sending and receiving process. All existing message ordering specifications such as FIFO, flush channels, causal ordering, and logically synchronous ordering as well as many new message orderings can be concisely specified using forbidden predicates.

We present a general algorithm to implement the message ordering that can be specified using forbidden predicates and are implementable without control messages. This shows the similarities in all the specification that are implementable without control messages. We further present techniques to generate efficient protocol for a given specification.

## 1 Introduction

A distributed computation or a run describes an execution of a distributed program. At an abstract level, a run can be defined as a partially ordered set $(H, \triangleright)$, where $H$ is the set of events in the system and $\triangleright$ the "happened before" relation [11] between events. It is often easier to develop distributed programs when the partially ordered set $(H, \triangleright)$ is guaranteed to satisfy certain message ordering properties. For example, many distributed algorithms work correctly only in the presence of FIFO channels. This guarantee on ordering of messages is either provided explicitly by communication primitives such as causal ordering [2] and logically synchronous ordering [4, 6], or built into the algorithm itself as with global snapshot and recovery algorithms. In this paper, we propose a general algorithm for existing and new message orderings that can be implemented by message tagging.

A fair amount of research has been done for efficient algorithms to implement different message orderings. Birman and Joseph [2], Raynal, Schiper and Toueg [14], Schiper, Eggli and Sandoz [15], have presented algorithms for the causal ordering of messages. These algorithms tag knowledge of processes about messages sent in the system with the message. For example, process $P_i$ in the algorithm by Raynal, Schiper and Toueg [14] tags a message with the matrix $m$ where $m[j,k]$ is the knowledge of process $P_i$ about the messages sent from $P_j$ to $P_k$. It is natural to ask whether the message ordering can be further restricted by sending higher-levels of knowledge (for example, by using three dimensional matrices: what $P_i$ knows that $P_j$ knows about messages sent from $P_k$ to $P_l$). It is an easy consequence of the results of this paper, that no additional tagging of information can restrict the message ordering further.

Variants of FIFO ordering have been studied under F-channels [1]. The implementation of F-channels, provides us with some basic synchronization primitives for sending messages: *two-way-flush* send, *forward-flush* send, *backward-flush* send, and *ordinary* send. Similar flush primitives can be defined for causal ordering. These message orderings can be specified using forbidden predicates. By constructing predicate graphs of these predicates it can be derived that these orderings can be implemented without using any control messages.

Many asynchronous consistent-cut protocols [16] such as global snapshot algorithms [5, 8, 12], check-pointing and rollback recovery [7, 10, 9], and deadlock detection [3] require special messages to find consistent-cuts in a computation. These protocols require some form of inhibition of the special messages in order to guarantee correctness.

In a broad setting, a message ordering specification can be characterized as a set of acceptable runs, that is, a subset of $\mathcal{X}$, where $\mathcal{X}$ is the set of all runs. For example, a system satisfying causal ordering can be viewed as the set of runs, say $\mathcal{X}_{co}$, such that for all runs in $\mathcal{X}_{co}$, and for all pairs of messages, $s_1 \triangleright s_2 \Rightarrow \neg(r_2 \triangleright r_1)$, where $s_j$ is the send of a message and $r_j$ is its corresponding receive. Since $\mathcal{X}$ is an infinite set, in [?] we presented a finite representation for its subsets that specify message ordering. We presented a method called *forbidden predicates*, that can be used to describe a large class of message specifications. All existing message ordering guarantees such as FIFO, flush channels, causal ordering, and logically synchronous ordering as well as others can be concisely specified using forbidden predicates. For example, the specification for causal ordering $\mathcal{X}_{co}$ can be stated as: for all runs in $\mathcal{X}_{co}$, and for all pairs of messages, $\neg\left(\left(s_1 \triangleright s_2\right) \wedge \left(r_2 \triangleright r_1\right)\right)$. The forbidden predicate for $\mathcal{X}_{co}$ is $\exists\,(s_1, r_1), (s_2, r_2)\ :\ (s_1 \triangleright s_2) \wedge (r_2 \triangleright r_1)$. In general, a forbidden predicate can be stated as a conjunction of causality relationships between the events (send and receive).

In this paper we are interested in the class of specifications that can be implemented using tagged user messages.

## 2 Model

A distributed system is defined as a set of $n$ sequential processes and a set of messages $M$. In this paper we will not consider internal events since they do not effect the partial order. A message $x$ consists of *send* and *receive* events and denoted as $(x.s, x.r)$.

**Definition 2.1:** A *run* is a pair $(H, \triangleright)$, where $\triangleright$ is an order relation on $H$ called the causality relation. For a run, $h \triangleright h'$ if:

1. $h$ and $h'$ are events in the same process and $h$ precedes $h'$, or
2. $(h, h') \in M$, or
3. $\exists\, h''$ such that $h \triangleright h''$ and $h'' \triangleright h'$.

■

The relation $\trianglerighteq$ is defined as $\triangleright \cup \equiv$, that is, if $h \trianglerighteq h'$ then either $h \triangleright h'$ or $h$ and $h'$ are the same events. Thus, $h \trianglerighteq h$ is true, but $h \triangleright h$ is false. If $(f \triangleright g)$ and $(g \triangleright h)$, then we also write it as $(f \triangleright g \triangleright h)$.

**Notation:** We represent events by letters $f, g, h$. Letters $F, G, H$ represent sets of events and letters $x, y, z$ represent messages and $a, b$ to represent message instances. A message $x \in M$ is a tuple $(x.s, x.r)$ where $x.s$ represents the send event and $x.r$ the corresponding receive event. $x_i.s$ and $x_i.r$ are the send and receive events of the message $x_i$ and is sometimes represented as $s_i$ and $r_i$, respectively. The letters $p$ and $q$ stand for either $s$ or $r$, therefore, $x.q$ is either $x.s$ or $x.r$ and

$q_i$ is either $s_i$ or $r_i$. $\mathcal{X}$ is always used to represent the set of all possible runs, and $\mathcal{Y}, \mathcal{Z}$ represents subsets of $\mathcal{X}$. □

### 2.1 Message Ordering Specifications

In general a specifications can be stated as a subset of $\mathcal{X}$. In this section we describe a method of stating a specification using *forbidden predicates.*

**Definition 2.2:**

1. A *forbidden predicate* $P$ is defined as,

$$P \equiv \exists\, x_1, x_2, \ldots, x_m \in M \;:\; P(x_1, x_2, \cdots, x_m)$$

   where,
$$P(x_1, x_2, \cdots, x_m) = \bigwedge_{i,j \neq i \in 1,\cdots,m} (x_i.p \triangleright x_j.q),$$

   and $p$ and $q$ represent $s$ or $r$.

2. Given a forbidden predicate $P(x_1, \cdots, x_m)$, the corresponding *specification set* $\mathcal{X}_P \subseteq \mathcal{X}$ is defined as,

$$\mathcal{X}_P = \{\, (H, \triangleright) \;:\; \neg P(x_1, \cdots, x_m),\;\; \forall\, x_0, x_1, \cdots, x_m \in M \subseteq I \times S \times R \,\}.$$

■

**Note on notation**: Let $P \equiv \exists\, x, y \in M \;:\; (x.s \triangleright y.s)$. We will write the predicate $P$ as $(x.s \triangleright y.s)$ dropping the quantifier $\exists$ for ease of use. $P(a, b)$ implies the evaluation of $(x.s \triangleright y.s)$ for the instances $a$ and $b$ in $M$. Therefore, $P(a, b)$ is true if and only if $a.s \triangleright b.s$. In case of ambiguity we express the predicate as $P \equiv \exists\, x, y \in M \;:\; P(x, y)$.

Given two forbidden predicates $P$ and $P'$ for the sets $\mathcal{X}_P$ and $\mathcal{X}_{P'}$, respectively, $P' \Rightarrow P$ iff $\mathcal{X}_P \subseteq \mathcal{X}_{P'}$. If a protocol for $P''$ guarantees that all the allowable partial orders belong to the set $\mathcal{X}_{P''}$, then the same protocol guarantees that all the allowable partial orders belong to the set $\mathcal{X}_{P'}$.

Consider, the example of causal ordering. The predicate can be stated as, $P \equiv (x.s \triangleright y.s) \wedge (y.r \triangleright x.r)$. For each element $(H, \triangleright)$ of $X_{co}$ (the corresponding specification set),

$$\forall\, x, y \in M \;::\; \neg\left((x.s \triangleright y.s) \;\wedge\; (y.r \triangleright x.r)\right).$$

Further, we can define two attributes with each message, that is, receiving and sending processes, and color. The function *process* $(h)$ returns the process identifier of the process in which event $h$ is executed. The attribute color is associated with a message and *color* $(a)$ returns the color associated with message $a$. We can use these attributes to define a range for the variables of the predicate. For example, the forbidden predicate for FIFO can be stated as,

$$\exists\, x, y \in M \;:\; \text{process}\,(x.s) = \text{process}\,(y.s) \wedge \text{process}\,(x.r) = \text{process}\,(y.r) \;:\; (x.s \triangleright y.s) \wedge (y.r \triangleright x.r),$$

or we may be interested in runs where a series of messages should not overtake the marker message,

$$\exists\, x, y \in M \;:\; \text{color}\,(y) = \mathit{red} \;:\; (x.s \triangleright y.s) \;\wedge\; (y.r \triangleright x.r).$$

## 2.2 Previous Results

In this section we classify the forbidden predicates to determine the type of algorithm necessary and sufficient to guarantee the safety and liveness.

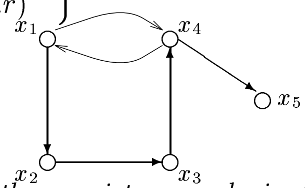
**Definition 2.3:** Let $P \equiv \exists x_1, \ldots, x_m \in M \; : \; P(x_1, \ldots, x_m)$ be a forbidden predicate. A *predicate graph* $G_P(V, E)$ is a multi-graph, such that,

$$\begin{aligned} V &= \{x_1, \ldots, x_m\} \\ E &= \{(x_i, x_j) \mid (x_i.p \triangleright x_j.q) \text{ is a conjunct of } P \text{ where } p, q \text{ is } s \text{ or } r \;\} \end{aligned}$$

■

**Example 1:** Let a predicate be,

$$P \equiv \left\{ \begin{array}{c} (x_1.r \triangleright x_2.s) \;\wedge\; (x_2.s \triangleright x_3.s) \;\wedge\; (x_3.r \triangleright x_4.r) \;\wedge \\ (x_4.s \triangleright x_1.r) \;\wedge\; (x_4.s \triangleright x_5.r) \;\wedge\; (x_1.s \triangleright x_4.r) \end{array} \right\},$$




then $G_P(V, E)$ is,

$$\begin{aligned} V &= \{x_0, x_1, x_2, x_3, x_4, x_5, x_6\} \\ E &= \{(x_1, x_2), (x_2, x_3), (x_3, x_4), (x_4, x_1), (x_4, x_5), (x_1, x_4)\}, \end{aligned}$$

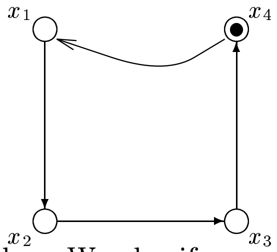
■

**Theorem 1** *There exists a protocol for a predicate $P$ if and only if there exists a cycle in the corresponding predicate graph $G_P(E, V)$.*

We are interested in cycles in the specification graphs. Pick any cycle $G_c(V^c, E^c) \subseteq G(V, E)$ in the specification graph and let the corresponding forbidden predicate be $P_c$.

**Example 2:** Consider the forbidden predicate and the graph from example 1. A possible cycle and the corresponding predicate is shown below. It is easy to see that $P \Rightarrow P_c$, since, $P_c$ is same as $P$ with some conjuncts removed.

$$\begin{aligned} V^c &= \{\, x_1, x_2, x_3, x_4 \,\} \\ E^c &= \{\, (x_1, x_2), (x_2, x_3), (x_3, x_4), (x_4, x_1) \,\} \\ P_c &= \left\{ \begin{array}{c} (x_1.r \triangleright x_2.s) \;\wedge\; (x_2.s \triangleright x_3.s) \wedge \\ (x_3.r \triangleright x_4.r) \;\wedge\; (x_4.s \triangleright x_1.s) \end{array} \right\} \end{aligned}$$




■

The specification graphs can contain a number of cycles. We classify a cycle into different categories based on the number of $\pi$ vertices it contains.

**Definition 2.4:** Given, a cycle $G_c(V^c, E^c)$ in the graph $G(V, E)$, we say $x \in V^c$ is a $\pi$ vertex with respect to the cycle $G_c(V^c, E^c)$ if the incoming edge is either $y.s \triangleright x.r$ or $y.r \triangleright x.r$ and the outgoing edge is either $x.s \triangleright z.s$ or $x.s \triangleright z.r$. The *order* of a cycle is equal to the number of $\pi$ vertices it contains. ■

**Theorem 2** *Let $\mathcal{X}_P$ be a specification with $P$ as the corresponding forbidden predicate. Let the predicate graph be $G_P(V, E)$. If there exists a cycle $G_c(V^c, E^c) \subseteq G_P(V, E)$ of order 0, then a trivial protocol (do nothing) can guarantee safety and liveness.*

**Theorem 3** *Let $\mathcal{X}_P$ be a specification with $P$ as the corresponding forbidden predicate. Let the predicate graph be $G_P(V, E)$ without a cycle of order 0. Then there exists a protocol with only tagged user messages if and only if there exists a cycle of order 1.*

In this paper we will be interested in predicate with their predicate graphs having a cycle of order 1 and not having a cycle of order 0.

## 3 Forbidden Predicates

The focus of this paper is finding a general algorithm that can implement a large class of message ordering, specified using forbidden predicates, without using any control messages. In the previous section we showed that this is possible only if the predicate graph, derived from the forbidden predicate, has a cycle with zero or one $\beta$ vertex. We will not consider the trivial case where there exists a cycle with zero $\beta$ vertex.

In this paper, we will consider forbidden predicates with the corresponding graphs being a cycle and deriving an optimal algorithm. The algorithm is optimal in the sense it does not restrict a run if it is legal. The reasons for focusing on these predicates are:

1. as we have seen in the previous section cycles in the graph charaterize whether the specification is implementable without using control messages.
2. many of the examples of message ordering specifications refered in literature are cycles,
3. we can argue about the optimality of the algorithm,
4. the same algorithm can be used for all forbidden predicates, but may not be optimal, and
5. we believe that a general (optimal) algorithm to consider all forbidden predicates is not possible and any algorithm that is better than this algorithm will be a variant of this algorithm.

The predicate graph for the predicates considered in this paper is a cycle with one $\beta$ vertex. Without loss of generality we will assume that the vertices of the graph are $V = \{x_1, x_2, \ldots, x_m\}$, the edge set as $E = \{\, (x_i, x_{(i \bmod m)+1}) \,:\, i = 1, 2, \ldots, m\}$ and the $x_1$ is the $\beta$ vertex. In addition for each $i$ the message $x_i$ has it satisfy $c_i(x_i)$ where $c_i(\cdot)$ is a trivial predicate, i.e a function of color of the message or the sending process identifier. Thus a general forbidden predicate can be written as,

$$\begin{aligned}\exists\, x_1, \ldots, x_m \in M \;:\quad & c_1(x_1) \wedge \cdots \wedge c_m(x_m)\\ & :\; (x_1.s \triangleright x_2.p) \wedge \left( \bigwedge_{i=2,\ldots m-1} (x_i.p' \triangleright x_{i+1}.q') \right) \wedge (x_m.q \triangleright x_1.r),\end{aligned}$$

where $p, p', q'$ and $q$ stand for either $s$ or $r$.

Some examples of forbidden predicates that can be implemented without using control messages are:

**FIFO** : The messages are received in the same order that they are sent between any pair of processes.

$$(\text{process}\,(s_1) = \text{process}\,(s_2)) \wedge (\text{process}\,(r_1) = \text{process}\,(r_2)) \;:\; (s_1 \triangleright s_2) \wedge (r_2 \triangleright r_1)$$

**Colored FIFO** : The messages of different kind (or different color) are received in the same order that they are sent between any pair of processes.

$$(\text{process}(s_1) = \text{process}(s_2)) \wedge (\text{process}(r_1) = \text{process}(r_2)) \wedge (\text{color}(x_1) \neq \text{color}(x_2)) \ : \ (s_1 \triangleright s_2) \wedge (r_2 \triangleright r_1)$$

**Causal Ordering** : A series of messages cannot overtake another message.

$$(s_1 \triangleright s_2) \wedge (r_1 \triangleright r_2).$$

**Colored Causal Ordering** : The ordering among messages with different colors is maintained. Similar predicates are used in many consistent-cut protocols.

$$(\text{color}(x_1) \neq \text{color}(x_2)) \ : \ (s_1 \triangleright s_2) \wedge (r_1 \triangleright r_2).$$

$k$–**Weaker Causal Ordering** : The messages can be out of order by atmost $k$ messages.

$$(s_1 \triangleright s_2) \wedge (s_2 \triangleright s_3) \wedge \cdots \wedge (r_{k+2} \triangleright r_1).$$

**Local Forward–Flush** : All messages sent before a *red* message are received before the *red* message between any pair of processes.

$$\text{process}(s_1) = \text{process}(s_2) \wedge \text{process}(r_1) = \text{process}(r_2) \wedge \text{color}(x_2) = \textit{red} \ : \ (s_1 \triangleright s_2) \wedge (r_2 \triangleright r_1).$$

**Local Backward–Flush** : All messages sent after a *red* message are received after the *red* message between any pair of processes

$$\text{process}(s_1) = \text{process}(s_2) \wedge \text{process}(r_1) = \text{process}(r_2) \wedge \text{color}(x_1) = \textit{red} \ : \ (s_1 \triangleright s_2) \wedge (r_2 \triangleright r_1).$$

**Global Forward–Flush** : All messages sent before a *red* message are received before the *red* message.

$$\text{color}(x_2) = \textit{red} \ : \ (s_1 \triangleright s_2) \wedge (r_2 \triangleright r_1).$$

**Global Backward–Flush** : All messages sent after a *red* message are received after the *red* message.

$$\text{color}(x_1) = \textit{red} \ : \ (s_1 \triangleright s_2) \wedge (r_2 \triangleright r_1).$$

## 4 An informal argument for the Algorithm

Informally, a forbidden predicate states that a run is illegal if a set of events satisfy certain causality relations. Therefore, an algorithm firstly, has to track all possible combination of events that may eventually satisfy the causality relations as specified by the forbidden predicate, and secondly, delay some events to gaurantee safety.

There are two steps in the algorithm,

1. the detection of different stages of the predicate, and

2. avoidance of one of the clauses and in our case its always the last clause.

The predicate can be viewed as a causality chain, and the first step keeps track of all events that have satisfied part of the causality chain. This is achevied by keeping $m$ sets where the length of the causality chain in the predicate is $m$. Second step is acheived by delaying some events such that the last causality relation in the causality chain is not true.

In this section we expand on these two steps by using the following predicate as an example,

$$\text{color}\,(x) = \textit{red} \ \wedge \ \text{color}\,(y) = \textit{green} \ : \ (x.s \triangleright y.p) \ \wedge \ (y.q \triangleright x.r),$$

where $p, q$ stand for either $s$ or $r$.

## 4.1 Detection

The algorithm associates $m$ sets (in this case $m = 2$), $L_1, L_2, \ldots, L_m$, with each process that represent the local information at a process, where $m$ depends on the forbidden predicate. These sets are called *level sets*. Whenever a process sends a message $x$ it tags its local information along with the message. The new local information of the receiving process is a function for its old local information, that is $L_i$s, and the information tagged along with the message $x$, denoted as $x.L_i$s. The value of the set $L_i$ just after the event $g$ is $L_i[g]$. The detection of the predicate is done in stages. For example, in the above predicate its done in two stages:

1. Detection of the first part of the predicate, i.e., detect the state $g$ such that

$$\text{color}\,(x) = \textit{red} \ : \ (x.s \triangleright g).$$

   We can rewrite the same as,

$$a \in L_1[g] \Leftrightarrow \exists\, a \ : \ \text{color}\,(a) = \textit{red} \ : \ (a.s \trianglerighteq g).$$

   Thus, $L_1$ keeps track of all messages that are *red* . In this case it is easy to see that if a message $a$ satisfies the condition, that is, color $(a) = \textit{red}$ , then the update of $L_1$ is done as,

| **sending process** | **receiving process** |
|---|---|
| $L_1 = L_1 \cup \{a\}$ | $L_1 = L_1 \cup a.L_1 \cup \{a\}$ |

2. Detection of the first part followed by the second part, i.e.,

$$a \in L_2[g] \Leftrightarrow \exists a, b \ : \ \text{color}\,(a) = \textit{red} \ \wedge \ \text{color}\,(y) = \textit{green} \ : \ (a.s \triangleright b.p) \ \wedge \ (b.q \triangleright g).$$

   Note that part of the predicate is same as the definition of $L_1$. Thus, we can rewrite $L_2$ using $L_1$ as,

$$a \in L_2[g] \Leftrightarrow \exists f, b \ : \ \text{color}\,(b) = \textit{green} \ : \ (f \triangleright b.p) \ \wedge \ (b.q \triangleright g) \ \wedge \ (a \in L_1[f]).$$

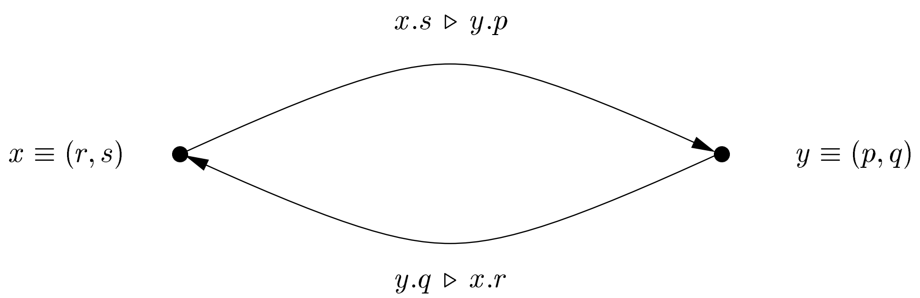



Figure 1: Predicate Graph for $[\text{color}\,(x) = \textit{red} \;\wedge\; \text{color}\,(y) = \textit{green} \;:\; (x.s \triangleright y.p) \wedge (y.q \triangleright x.r)]$.

If a process executes a message $b$ such that $\text{color}\,(b) = \textit{green}$ then the sending process can update the level sets $L_1, L_2$ using its old values and the knowledge that message $b$ satisfies the condition. Similarly, the receiving process can update its level sets based on its level sets and the level sets tagged with the message $b$. All the element of $L_1$ that are in the causal past of $b.p$ are added to $L_2$ in the causal future of $b.q$. Alternatively, if $a \in L_1[f]$, where $f$ is in the causal past of $b.p$ then $a \in L_2[g]$, where $g$ is in the causal future of $b.q$.

In the above predicate $(p, q)$ can be one of the four possibilities: $(s, s)$, $(s, r)$, $(r, r)$, or $(r, s)$. Figure 2 shows the corresponding predicate graph. The vertex $x$ is of type $(r, s)$ and $y$ is of type $(p, q)$. The meaning of level sets in this case is,

$$\begin{aligned} a \in L_1[g] \;&\Leftrightarrow\; \exists\, a \;:\; \text{color}\,(a) = \textit{red} \;:\; (a.s \trianglerighteq g), \\ a \in L_2[g] \;&\Leftrightarrow\; \exists\, f, b \;:\; \text{color}\,(b) = \textit{green} \;:\; (f \triangleright b.p) \wedge (b.q \trianglerighteq g) \wedge (a \in L_1[f]) \end{aligned}$$

Whenever a message $a$ satisfies the condition ($\text{color}\,(a) = \textit{red}$ ) then $L_1$ is updated. Similarly, if a message $a$ satisfies the condition ($\text{color}\,(a) = \textit{green}$ ) then $L_2$ is updated It is easy to see that the level sets satisfy the monotonic property,

$$(f \triangleright g) \Rightarrow L_i[f] \subseteq L_i[g].$$

To maintain the monotonic property the update for sending process is always like $L_i = L_i \cup \cdots$ and for the receiving process $L_i = L_i \cup b.L_i \cup \cdots$, for a message $b$.

We will focus on a message $b$ that satisfies the condition ($\text{color}\,(b) = \textit{green}$ ) and the process of updating the level set $L_2$ in each of the four possible cases of the above marker message predicate. If $\underline{p = s \text{ and } q = s}$ then the update condition is,

$$a \in L_2[g] \Leftrightarrow \exists\, f, b \;:\; \text{color}\,(b) = \textit{green} \;:\; (f \triangleright b.s) \wedge (b.s \trianglerighteq g) \wedge (a \in L_1[f]).$$

and the update is done ($\text{color}\,(b) = \textit{green}$ ) as,

| **sending process** | **receiving process** |
|---|---|
| $L_2 = L_2 \cup L_1$ | $L_2 = L_2 \cup b.L_2 \cup b.L_1$ |

If $\underline{p = s \text{ and } q = r}$ then the update condition is,

$$a \in L_2[g] \Leftrightarrow \exists\, f, b \;:\; \text{color}\,(b) = \textit{green} \;:\; (f \triangleright b.s) \wedge (b.r \trianglerighteq g) \wedge (a \in L_1[f]).$$

and the update is done ($\text{color}\,(b) = \textit{green}$ ) as,

| sending process | receiving process |
|---|---|
| $L_2 = L_2$ | $L_2 = L_2 \cup b.L_2 \cup b.L_1$ |

If $\underline{p = r \text{ and } q = r}$ then the update condition is,

$$a \in L_2[g] \Leftrightarrow \exists\, f, b \;:\; \text{color}\,(b) = \mathit{green} \;\;:\; (f \rhd b.r) \wedge (b.r \unrhd g) \wedge (a \in L_1[f]).$$

and the update is done (color $(b) = \mathit{green}$ ) as,

| sending process | receiving process |
|---|---|
| $L_2 = L_2$ | $L_2 = L_2 \cup b.L_2 \cup L_1 \cup b.L_1$ |

If $\underline{p = r \text{ and } q = s}$ then the predicate graph corresponding to the forbidden predicate color $(x) = \mathit{red} \wedge \text{color}\,(y) = \mathit{green} \;:\; (x.s \rhd y.r) \wedge (y.s \rhd x.r)$ is of order 2 (that is, two $\pi$ vertices). Therefore, there does not exists a protocol with only tagged messages (from theorem **??**).

The condition for update of $L_2$ can be rewritten as,

$$a \in L_2[g] \Leftrightarrow \exists\, b, f \;:\; \text{color}\,(b) = \mathit{green} \;\;:\; (f \rhd l_2(b) \unrhd g) \wedge (a \in L_1[f]),$$

where

$$l_2(b) = \begin{cases} b.s & \text{if } p = s \text{ and } q = s \\ b.s \rhd b.r & \text{if } p = s \text{ and } q = r \\ b.r & \text{if } p = r \text{ and } q = r \end{cases}.$$

### 4.2 Safety

The algorithm maintains the safety property by maintaining the invariant

$$\mathit{wait}(g) \equiv (x \in L_m[g]) \wedge (\text{process}\,(x.r) = \text{process}\,(g)) \Rightarrow (x.r \rhd g).$$

Informally, this states the waiting condition for the event $g$, that is, wait until all the message have been received that will belong the set $L_m[g]$.

Let us consider the above forbidden predicate. The $L_i$s are,

$$\begin{aligned} a \in L_1[g] &\Leftrightarrow \mathit{color}\,(a) = \mathit{red} \;\;:\; (a.s \unrhd g), \\ a \in L_2[g] &\Leftrightarrow \exists f, b \;:\; \text{color}\,(b) = \mathit{green} \;\;:\; (f \rhd b.p) \wedge (b.q \rhd g) \wedge (a \in L_1[f]). \end{aligned}$$

Consider a case when the invariant $\mathit{wait}(g)$ is not maintained. That is, $a \in L_m[g] \wedge (\text{process}\,(a.r) = \text{process}\,(g) \Rightarrow \neg(a.r \rhd g)$. Since, the events $a.r$ and $g$ are in the same process, $\neg(a.r \rhd g) \Rightarrow (g \unrhd a.r)$. From $(g \unrhd a.r)$ and $a \in L_2[g]$, i.e,

$$\exists\, b \;:\; \text{color}\,(a) = \mathit{red} \;\wedge\; \text{color}\,(b) = \mathit{green} \;\;:\; (a.s \rhd b.p) \wedge (b.q \unrhd g),$$

we have $\exists\, a, b \;:\; (a.s \rhd b.s) \wedge (b.r \unrhd g) \wedge (g \unrhd a.r)$. Therefore,

$$\exists\, a, b \in M \;:\; \text{color}\,(a) = \mathit{red} \;\wedge\; \text{color}\,(b) = \mathit{green} \;\;:\; (a.s \rhd b.p) \wedge (b.q \rhd a.r)$$

. Thus, $P(a, b)$ is true and the run is illegal.

## 5 Properties of the Algorithm

Consider a predicate $P$ with the corresponding graph $G(V,E)$ being a cycle. Let the vertex set be $V = \{x_1, x_2, \cdots, x_m\}$ and the edge set be $E = \{ (x_i, x_{i \bmod m+1}) : i = 1, \cdots m \}$. Without loss of generality we will assume $x_1$ is the $\pi$ vertex. The conditions satisfied by messages are $C = \{c_1, c_2, \cdots, c_m\}$. The forbidden predicate can be written as:

$$\exists\, x_1, \ldots x_m \in M \;:\; c_1(x_1) \wedge \cdots \wedge c_m(x_m) \;:\; (x_1.s \triangleright x_2.p) \wedge \cdots \wedge (x_m.q \triangleright x_1.r).$$

We will drop the quantifier $\exists$ for ease of use. For example, consider the forbidden predicate from the last section,[1]

$$\text{color}\,(x_1) = \textit{red} \;\wedge\; \text{color}\,(x_2) = \textit{green} \;:\; (x_1.s \triangleright x_2.p) \wedge (x_2.q \triangleright x_1.r),$$

here there vertex set $V = \{x_1, x_2\}$ and $E = \{(x_1, x_2), (x_2, x_1)\}$ The conditions satisfied by the messages are $C = \{c_1, c_2\}$, where $c_1(x) \equiv (\text{color}\,(x) = \textit{red}\,)$ and $c_2(x) \equiv (\text{color}\,(x) = \textit{green}\,)$.

The definition for the level set $L_1$ for the example predicate is,

$$a \in L_1[g] \Leftrightarrow \exists\, a \;:\; \text{color}\,(a) = \textit{red} \;:\; (a.s \trianglerighteq g).$$

Similarly, for the general predicate we get,

$$a \in L_1[g] \Leftrightarrow \exists\, a \;:\; c_1(a) \;:\; (a.s \trianglerighteq g).$$

The definition of the level set $L_2$ for the example predicate is,

$$a \in L_2[g] \Leftrightarrow \exists\, f, b \;:\; \text{color}\,(b) = \textit{green} \;:\; (f \triangleright l_2(b) \trianglerighteq g) \wedge (a \in L_1[f]).$$

Similarly, for the general predicate we get,

$$a \in L_2[g] \Leftrightarrow \exists\, a \;:\; c_2(b) \;:\; (f \triangleright l_2(b) \trianglerighteq g) \wedge (a \in L_1[f]).$$

This can be generalized to other level sets and we get,

$$a \in L_i[g] \Leftrightarrow \exists\, f, b \;:\; c_i(b) \;:\; (f \triangleright l_i(b) \trianglerighteq g) \wedge (a \in L_{i-1}[f]),$$

where

$$l_i(b) = \begin{cases} b.s & \text{if vertex } x_i \text{ is of type } (s,s) \\ b.s \triangleright b.r & \text{if vertex } x_i \text{ is of type } (s,r) \\ b.r & \text{if vertex } x_i \text{ is of type } (r,r) \end{cases}.$$

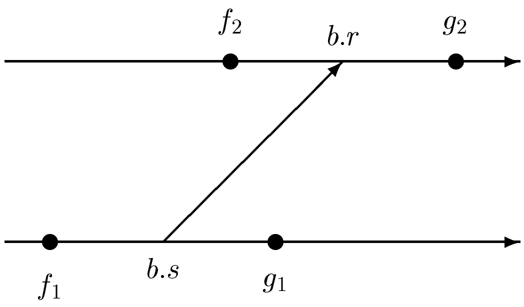
Informally, the meaning of $L_i[g]$ is a set of messages where $a \in L_i[g]$ if there exists a set of messages $\{b_2, b_3, \ldots, b_i\}$, such that, $a.s \triangleright l_1(b_1) \triangleright l_2(b_2) \triangleright \cdots \triangleright l_i(b_i) \trianglerighteq g$ and $c_1(a) \wedge c_2(b_2) \wedge c_3(b_3) \wedge \cdots \wedge c_i(b_i)$. A messages $a$ is moved up the level sets from $L_1$ to $L_2$ and so forth. Eventually $a \in L_m$ and an event $e$ has to wait for the message $a$ before executing, satisfying the waiting condition,

$$\textit{wait}(g) \equiv (x \in L_m[g]) \wedge (\text{process}\,(x.r) = \text{process}\,(g)) \Rightarrow (x.r \triangleright g).$$

The update of level set $L_i$ $(i \neq 1)$ is done depending on the vertex type for $x_i$. (see figure **??**).

The conditions meet by the algorithm are as follows:

[1] instead of using $x, y$ as free variable we are using $x_1, x_2$, respectively.




**Vertex type is $(s, s)$** :

$$a \in L_i[g] \Leftrightarrow \exists\, b, f \;:\; c_i(b) \;:\; (f \triangleright b.s) \wedge (b.s \trianglerighteq g) \wedge (a \in L_{i-1}[f])$$

$$x \in L_i[g_1] \quad \Leftrightarrow \quad \begin{cases} x \in L_i[f_1] & \vee \quad \text{Monotonic} \\ x \in L_{i-1}[f_1] & \quad \text{Update Condition} \end{cases}$$

$$x \in L_i[g_2] \quad \Leftrightarrow \quad \begin{cases} x \in L_i[f_2] & \vee \quad \text{Monotonic} \\ x \in L_i[f_1] & \vee \quad \text{Monotonic} \\ x \in L_{i-1}[f_1] & \quad \text{Update Condition} \end{cases}$$

**Vertex type is $(s, r)$** :

$$a \in L_i[g] \Leftrightarrow \exists\, b, f \;:\; c_i(b) \;:\; (f \triangleright b.s) \wedge (b.r \trianglerighteq g) \wedge (a \in L_{i-1}[f])$$

$$x \in L_i[g_1] \quad \Leftrightarrow \quad \begin{cases} x \in L_i[f_1] & \quad \text{Monotonic} \end{cases}$$

$$x \in L_i[g_2] \quad \Leftrightarrow \quad \begin{cases} x \in L_i[f_2] & \vee \quad \text{Monotonic} \\ x \in L_i[f_1] & \vee \quad \text{Monotonic} \\ x \in L_{i-1}[f_1] & \quad \text{Update Condition} \end{cases}$$

**Vertex type is $(r, r)$** :

$$a \in L_i[g] \Leftrightarrow \exists\, b, f \;:\; c_i(b) \;:\; (f \triangleright b.s) \wedge (b.s \trianglerighteq g) \wedge (a \in L_{i-1}[f])$$

$$x \in L_i[g_1] \quad \Leftrightarrow \quad \begin{cases} x \in L_i[f_1] & \quad \text{Monotonic} \end{cases}$$

$$x \in L_i[g_2] \quad \Leftrightarrow \quad \begin{cases} x \in L_i[f_2] & \vee \quad \text{Monotonic} \\ x \in L_i[f_1] & \vee \quad \text{Monotonic} \\ x \in L_{i-1}[f_2] & \vee \quad \text{Update Condition} \\ x \in L_{i-1}[f_1] & \quad \text{Update Condition} \end{cases}$$

Figure 2: Update of level Sets

**Waiting Condition (WC)** : When is an event delayed ?

$$(x \in L_m[g]) \wedge (\text{process}\,(x.r) = \text{process}\,(g)) \Rightarrow (x.r \rhd g).$$

**Entry Condition (EC)** : What messages are of interest ?

$$c_1(a) \wedge (a.s \unrhd h) \Leftrightarrow a \in L_1[h].$$

**Update Condition (UC)** : When should a message $a$ become a member of level set $L_i, 1 < i \leq m$ ?

$$\exists\, b, f \;:\; c_i(b) \wedge (f \rhd l_i(b) \unrhd g) \wedge (a \in L_{i-1}[f]) \Leftrightarrow (a \in L_i[g]).$$

We can show that UC $\Rightarrow$ **Monotonic Property (MP)** where, MP is

$$x \in L_i[g] \Rightarrow \forall\, h \;:\; g \rhd h \;:\; x \in L_i[h].$$

Let $a \in L_1[g]$ and $g \rhd h$.

$$\begin{array}{rcll} a \in L_1[g] & \Rightarrow & c_1(a) \wedge (a.s \unrhd g) & \text{Using EC} \\ & \Rightarrow & c_1(a) \wedge (a.s \rhd h) & \text{Using } g \rhd h \\ & \Rightarrow & a \in L_1[h] & \text{Using EC} \end{array}$$

Let $a \in L_i[g], i > 1$ and $g \rhd h$.

$$\begin{array}{rcll} a \in L_i[g] & \Rightarrow & \exists\, b, f \;:\; c_i(b) \wedge (f \rhd l_i(b) \unrhd g) \wedge (a \in L_{i-1}[f]) & \text{Using UC} \\ & \Rightarrow & \exists\, b, f \;:\; c_i(b) \wedge (f \rhd l_i(b) \rhd h) \wedge (a \in L_{i-1}[f]) & \text{Using } g \rhd h \\ & \Rightarrow & a \in L_i[h] & \text{Using UC} \end{array}$$

The algorithm satisfying the above conditions (WC, EC, MP and UC) is given in the figure 3. Here we give an informal argument to show that the algorithm satisfies the above four properties.

# 6 Proof for the Correctness of the Algorithm

**Lemma 1** *If UC and EC are true, then*

1. $x \in L_i[g] \Rightarrow x \in L_{i-1}[g]$.
2. $x \in L_i[g] \Rightarrow (x.s \unrhd g) \wedge c_1(x)$.

**Proof:** The first statement follows from MP and UC.

$$\begin{array}{rcll} a \in L_i[g] & \Rightarrow & \exists\, f \;:\; (f \rhd g) \wedge (a \in L_{i-1}[f]) & \text{Using UC} \\ & \Rightarrow & a \in L_{i-1}[g] & \text{Using } g \rhd f \text{ and MP} \end{array}$$

The second statement follows from 2 and EC.

$$a \in L_i[h] \Rightarrow a \in L_{i-1}[h] \Rightarrow \cdots \Rightarrow L_1[h] \Rightarrow (x.s \unrhd h) \wedge c_1(x).$$

$\square$

**Enabling condition for event** $g$ :

E1 $(x \in L_m[g]) \ \wedge \ (\text{process}\,(a.r) = \text{process}\,(g)) \Rightarrow (x.r \triangleright g)$.

**sending message** $x$ :

S1 Message Tag $= \{L_1, L_2, \ldots, L_m\}$
S2 **for** $i$ **in** $\{m, m-1, \cdots, 2\}$ **do**
S3     **if** $(c_i(x))$ **then**
S4         **if** $v_i$ is of type $(s, s)$ **then**
S5             $L_i = L_i \cup L_{i-1}$.
S6 **if** $(c_1(x))$ **then**
S7     $L_1 = L_1 \cup \{x\}$

**Receiving message** $x$ :

R1 $L_i = L_i \cup x.L_i$
R2 **for** $i$ **in** $\{m, m-1, \cdots, 2\}$ **do**
R3     **if** $(c_i(x))$ **then**
R4         **if** $v_i$ is of type $(s, s)$ **then**
R5             $L_i = L_i \cup x.L_{i-1}$.
R6         **if** $v_i$ is of type $(s, r)$ **then**
R7             $L_i = L_i \cup x.L_{i-1}$.
R8         **if** $v_i$ is of type $(r, r)$ **then**
R9             $L_i = L_i \cup L_{i-1} \cup x.L_{i-1}$.
R10 **if** $(c_1(x))$ **then**
R11     $L_1 = L_1 \cup \{x\}$

Figure 3: Pseudo Code for the Algorithm

**Lemma 2** *Let there exists an event $g$ and a message $a$ such that the following conditions are true.*

1. $(a \in L_m[g]) \wedge (g \triangleright a.r)$.
2. $x \in L_i[h] \Rightarrow \exists\, b, f \;:\; c_i(b) \wedge (f \triangleright l_i(b) \trianglerighteq h) \wedge (x \in L_{i-1}[f])$.
3. $x \in L_1[h] \Rightarrow (x.s \trianglerighteq h) \wedge c_1(x)$.

*Then there* $\exists\, a_1, a_2, \ldots, a_m \;:\; P(a_1, a_2, \ldots, a_m)$ *is true.*

**Proof:** Let $a_1 \equiv a$. From condition 1, we have $a_1 \in L_m[g]$. On expanding the definition of $L_m[g]$ we have,

$$a_1 \in L_m[g] \quad \Rightarrow \quad \exists\, a_m, f \;:\; c_m(a_m) \wedge (f \triangleright l_m(a_m) \trianglerighteq g) \wedge (a_1 \in L_{m-1}[f]).$$

On exapnding $L_{m-1}[f]$ we get,

$$\begin{aligned} a_1 \in L_m[g] \quad \Rightarrow \quad & \exists\, a_m, f \;:\; c_m(a_m) \wedge (f \triangleright l_m(a_m) \trianglerighteq g) \wedge \\ & (\exists\, a_{m-1}, h \;:\; c_{m-1}(a_{m-1}) \wedge (h \triangleright l_{m-1}(a_{m-1}) \trianglerighteq f) \wedge (a_1 \in L_{m-2}[h]))\,. \end{aligned}$$

On simplification we get,

$$\begin{aligned} a_1 \in L_m[g] \quad \Rightarrow \quad & \exists\, a_m, a_{m-1}, h \;:\; c_m(a_m) \wedge c_{m-1}(a_{m-1}) \wedge \\ & (h \triangleright l_{m-1}(a_{m-1}) \triangleright l_m(a_m) \trianglerighteq g) \wedge \\ & (a_1 \in L_{m-2}[h]). \end{aligned}$$

On repeated process of expanding $L_i$s we eventually get,

$$\begin{aligned} a_1 \in L_m[g] \quad \Rightarrow \quad & \exists\, a_m, a_{m-1}, \cdots a_2, h \;:\; c_m(a_m) \wedge c_{m-1}(a_{m-1}) \wedge \cdots \wedge c_2(a_2) \wedge \\ & (h \triangleright l_2(a_2) \triangleright \cdots \triangleright l_{m-1}(a_{m-1}) \triangleright l_m(a_m) \trianglerighteq g) \wedge \\ & (a_1 \in L_1[h]). \end{aligned}$$

If $(a_1 \in L_1[h])$ then $(a_1.s \trianglerighteq h) \wedge c_1(a_1)$. Therefore,

$$\begin{aligned} a_1 \in L_m[g] \quad \Rightarrow \quad & \exists\, a_m, a_{m-1}, \cdots a_2 \;:\; c_m(a_m) \wedge c_{m-1}(a_{m-1}) \wedge \cdots \wedge c_1(a_1) \wedge \\ & (a_1.s \triangleright l_2(a_2) \triangleright \cdots \triangleright l_{m-1}(a_{m-1}) \triangleright l_m(a_m) \trianglerighteq g) \end{aligned}$$

Since $(a_1 \in L_m[g]) \wedge (g \triangleright a_1.r)$,

$$\begin{aligned} \exists \quad & a_m, a_{m-1}, \cdots a_1 \;:\; c_m(a_m) \wedge c_{m-1}(a_{m-1}) \wedge \cdots \wedge c_1(a_1) \wedge \\ & (a_1.s \triangleright l_2(a_2) \triangleright \cdots \triangleright l_{m-1}(a_{m-1}) \triangleright l_m(a_m) \triangleright a_1.r) \end{aligned}$$

Therefore $P(a_1, a_2, \ldots, a_m)$ is true. □

**Lemma 3** *Let there exist messages $a_1, a_2, \ldots, a_m$, such that*

1. $P(a_1, a_2, \ldots, a_m)$ *is true.*
2. $c_1(a) \Rightarrow L_1[a.s]$.
3. $\exists\, b, f \;:\; c_i(b) \wedge (f \triangleright l_i(b) \trianglerighteq g) \wedge (a \in L_{i-1}[f]) \wedge \neg(a.r \triangleright g) \Rightarrow a \in L_j[g]$.

*4.* $a \in L_m[g] \Rightarrow \forall h \ : \ (g \triangleright h) \wedge \neg(h \triangleright a.r) \ : \ a \in L_m[h]$.

*Then* $\exists g \ : \ wait(g)$ *is false.*

**Proof:** Since $P(a_1, a_2, \ldots, a_m)$, we have

$$c_1(a_1) \wedge c_2(a_2) \wedge \cdots \wedge c_m(a_m), \text{ and}$$

$$(a_1.s \triangleright l_2(a_2) \triangleright \cdots \triangleright l_{m-1}(a_{m-1}) \triangleright l_m(a_m) \triangleright a_1.r).$$

Let $t_i(x)$ represent the top element of $l_i(x)$, that is,

$$t_i(x) = \begin{cases} x.s & \text{if} \quad l_i(x) = x.s \\ x.r & \text{if} \quad l_i(x) = x.s \triangleright x.r \\ x.r & \text{if} \quad l_i(x) = x.r \end{cases}$$

$$\begin{array}{rcl} 2 \text{ and } c_1(a_1) & \Rightarrow & a_1 \in L_1[a_1.s]. \\ 3 \text{ and } c_2(a_2) \wedge (a_1.s \triangleright l_2(a_2) \trianglerighteq t_2(a_2) \triangleright a_1.r) \wedge (a_1 \in L_1[a.s]) & \Rightarrow & a_1 \in L_2[t_2(a_2)]. \\ 3 \text{ and } c_2(a_2) \wedge (t_2(a_2) \triangleright l_2(a_2) \trianglerighteq t_2(a_2) \triangleright a_1.r) \wedge (a_1 \in L_1[t_2(a_2)]) & \Rightarrow & a_1 \in L_3[t_2(a_2)]. \end{array}$$

By repeated application of condition 3, we eventually get

$$a_1 \in L_m[t_m(a_m)].$$

Since $(t_m[a_m] \triangleright a_1.r)$ and $(a_1.s \triangleright t_m(a_m))$ we get

$$\begin{array}{r} process\,(t_m[a_m]) = \text{process}\,(a_1.r) \quad \vee \\ \exists g \ : \ (t_m[a_m] \triangleright g \triangleright a_1.r) \wedge (\text{process}\,(g) = \text{process}\,(a_1.r)). \end{array}$$

From condition 4, we have $a_1 \in L_m[g]$ for all $g$ such that $t_m(a_m) \trianglerighteq g$ and $\neg(a.r \triangleright g)$. Therefore,

$$\exists g \ : \ (a_1 \in L_m[g]) \wedge (\text{process}\,(a_1.r) = \text{process}\,(g)) \wedge (g \triangleright a_1.r).$$

Therefore $wait(g)$ is false. $\square$

**Theorem 4 (Safety)** *Let the data structures, i.e.,* $L_i$*s satisfy UC and EC. Then there exists a set of messages* $\{a_1, a_2, \ldots, a_m\}$ *such that* $P(a_1, \ldots, a_m)$ *is true if and only if* $\exists g \ : \ wait(g)$ *is false.*

**Proof:** First, we prove "if" part. Let there exist an event $g$ such that $wait(g)$ is false. Therefore, $\exists a \ : \ a \in L_m[g]) \wedge (\text{process}\,(x.r) = \text{process}\,(g)) \wedge \neg(x.r \triangleright g)$. Since, (process $(x.r) = \text{process}\,(g)$,

$$\exists a, g \ : \ (a \in L_m[g]) \wedge (g \triangleright a.r).$$

The three conditions of lemma 2 are true.

1. $(a \in L_m[g]) \wedge (g \triangleright a.r)$, since $\exists g \ : \ wait(g)$ is false.
2. $x \in L_i[h] \Rightarrow \exists b, f \ : \ c_i(b) \wedge (f \triangleright l_i(b) \trianglerighteq h) \wedge (x \in L_{i-1}[f])$, follows from UC,
3. $x \in L_1[h] \Rightarrow (x.s \trianglerighteq h) \wedge c_1(x)$, follows from lemma 1.

. Therefore, $\exists\, a_1, a_2, \ldots, a_m : P_(a_1, a_2, \ldots, a_m)$ is true.

We proceed to prove "only if" part. Let there exists $\{a_1, a_2, \ldots, a_m\}$ such that $P(a_1, a_2, \ldots, a_m)$ is true. The four conditions of lemma 3 are true.

1. $P(a_1, a_2, \ldots, a_m)$ is true, given.
2. $c_1(a) \Rightarrow L_1[a.s]$, follows from EC.
3. $\neg(x.r \triangleright g) \wedge (\exists\, b, f : c_i(b) \wedge (f \triangleright l_i(b) \trianglerighteq h) \wedge (x \in L_{i-1}[f])) \Rightarrow x \in L_i[g]$, follows from UC,
4. $x \in L_m[g] \Rightarrow \forall\, h : g \triangleright h) \wedge \neg(a.r \triangleright h) : x \in L_m[h]$, follows from MP.

Therefore, $\exists\, g : wait(g)$ is false. □

**Theorem 5 (Liveness)** *Every event is eventually executed, follows from lemma 1.*

**Proof:** An event $g$ is delayed if only $a \in L_n[g]$ and process $(a.r) =$ process $(g)$ and has not been received. Since, $a \in L_i[g]$, $a.s \triangleright g$ Therefore, either $a.r$ has been received or the message is in transit. □

## 7 Garbage Collection

In the algorithm there is no way to purge an entry from any of the level sets. In this section we discuss two method to purge an entry from a level set.

In the algorithm an event $g$ is delayed until it receives all the messages in the set $L_m[g]$. If a process knows that a message has been already received then the level sets $L_i$s need not keep track of the message, in the causal future. Secondly, if a message $x \in L_j[g]$ it is redundant for a level set $L_i[g]$ where $i < j$ to keep track of that information, since the waiting condition of any event depends on the messages in the level set $L_m$. Thus, taking these ideas into consideration we get the following conditions to be met by the algorithm:

**Waiting Condition (WC)** : When is an event delayed ?

$$(x \in L_m[g]) \wedge (\text{process}\,(x.r) = \text{process}\,(g)) \Rightarrow (x.r \triangleright g).$$

**Modified Entry Condition (MEC)** : What messages are of interest ?

$$c_i(a) \wedge (a.s \trianglerighteq h) \wedge \neg(a.r \trianglerighteq h) \Leftrightarrow (a \in L_1[h]).$$

**Modified Update Condition (MUC)** :

$$\exists\, b, f : c_i(b) \wedge (f \triangleright l_i(b) \trianglerighteq g) \wedge (a \in L_{i-1}[f]) \wedge \neg(a.r \triangleright g) \Leftrightarrow a \in L_j[g].$$

We can prove similar results using WC, MEC and MUC as was done for WC, EC and UC.
**Lemma 4** *If MUC and MEC are true, then*

1. $x \in L_i[g] \Rightarrow x \in L_{i-1}[g]$.
2. $x \in L_i[g] \Rightarrow (x.s \trianglerighteq g) \wedge c_1(x)$.

*3.* $a \in L_i[h] \Rightarrow \neg(a.r \triangleright h)$.

**Proof Outline:** Similar to proof of lemma 1. The thrid part follows from MUC and MEC. $\square$

**Theorem 6 (Safety)** *Let the data structures, i.e., $L_i$s satisfy MUC and MEC. Then there exists a set of messages $\{a_1, a_2, \ldots, a_m\}$ such that $P(a_1, \ldots, a_m)$ is true if and only if $\exists\, g\ :\ wait(g)$ is false.*

**Proof Outline:** To prove "if" part show that MUC, MEC and $\exists\, g\ :\ wait(g)$ is false, implies the three statements of lemma 2.

To prove "only if" part show that MUC, MEC and $\exists\, a_i$s $:\ P(a_1, a_2, \ldots, a_m)$ is true, implies the four statements of lemma 3. $\square$

**Theorem 7 (Liveness)** *Every event is eventually executed, follows from lemma 4.*

**Proof Outline:** Similar to earlier liveness proof. $\square$

It is interseting to note that if $x \in L_i[g]$ then $x \in L_{i-1}[g]$ (from lemma 4), therefore, it is sufficient to maintain $x$ as an element of the largest indexed set. Therefore, an efficient algorithm maintains the following properties:

**Uniqueness propery** : $x \in L_i[g] \Rightarrow x \notin L_j[g], \forall\, j < i$.

**Purge Condition** : $x.r \triangleright g \Rightarrow x \notin L_i[g], \forall\, i$.

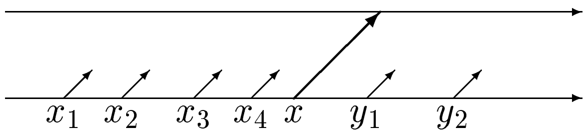
To maintain the unquiness condition is trivial. In this section we explain a method to do the purge. This is done by keeping vector clocks, and assigning a number in increasing order to each message sent by a process.

The algorithm is shown in figure 4.

# 8 Optimization

- Every message has to tag along $\{L_i\}$. And $|L_{n-1}|$ monotonically increases.
  - $x < y \;\Leftrightarrow\; x \in L_m[y.s] \;\wedge\;$ (process $(x.r) =$ process $(y.r)$)
  - $L_{m-1}[e] =$ maximal elements of $L_{m-1}[e]$
  - Useful simplification: $x < y \;\Leftrightarrow\; (x.s \triangleright y.s) \;\wedge\;$ (process $(x.r) =$ process $(y.r)$).
  - It results in optimal CO implementation.

- Use of equivalence class for $s.L_i$.
  - $x \sim y \;\equiv\; (x \in L_m[e] \;\Leftrightarrow\; y \in L_m[e]) \;\wedge\;$ (process $(x.r) =$ process $(y.r)$)
  - use of counter



**Enabling condition for event** $g$ :

E1 $(x \in L_m[g]) \ \wedge \ (\text{process}\,(a.r) = \text{process}\,(g)) \Rightarrow (x.r \triangleright g)$.

**sending message** $x$ :

S1 Message Tag $= \{L_1, L_2, \ldots, L_m, V\}$
S2 **for** $i$ **in** $\{m, m-1, \cdots, 2\}$ **do**
S3     **if** $(c_i(x))$ **then**
S4         **if** $v_i$ is of type $(s,s)$ **then**
S5             $L_i = L_i \cup L_{i-1}$, $L_{i-1} = \emptyset$
S6 **if** $(c_1(x))$ **then**
S7     $V[\text{process}\,(x.s)]++$
S8     $L_1 = L_1 \cup \{(x, V[\text{process}\,(x.s)]\}$

**Receiving message** $x$ :

R1 **for** $i$ **in** $\{m, m-1, \cdots, 1\}$ **do**
R2     **if** $((x,n) \in \ L_i) \ \wedge \ ((x,n) \notin \cup_j \, x.L_j) \ \wedge \ (n \leq x.V[\text{process}\,(x.s)])$ **then**
R3         $L_i = L_i - \{(x,n)\}$
R4     **if** $((x,n) \in \ x.L_i) \ \wedge \ ((x,n) \notin \cup_j \, L_j) \ \wedge \ (n \leq V[\text{process}\,(x.s)])$ **then**
R5         $x.L_i = x.L_i - \{(x,n)\}$
R6     **if** $((x,n) \in \ L_i) \ \wedge \ ((x,n) \in \cup_{j>i} \, x.L_j)$ **then**
R7         $L_i = L_i - \{(x,n)\}$
R8     **if** $((x,n) \in \ x.L_i) \ \wedge \ ((x,n) \in \cup_{j>i} \, L_j)$ **then**
R9         $x.L_i = x.L_i - \{(x,n)\}$
R10 **for** $i$ **in** $\{m, m-1, \cdots, 1\}$ **do**
R11     $L_i = L_i \cup x.L_i$
R12 **for** $i$ **in** $\{m, m-1, \cdots, 2\}$ **do**
R13     **if** $(c_i(x))$ **then**
R14         **if** $v_i$ is of type $(s,s)$ **then**
R15             $L_i = L_i \cup x.L_{i-1}$, $L_{i-1} = L_{i-1} - x.L_{i-1}$
R16         **if** $v_i$ is of type $(s,r)$ **then**
R17             $L_i = L_i \cup x.L_{i-1}$, $L_{i-1} = L_{i-1} - x.L_{i-1}$
R18         **if** $v_i$ is of type $(r,r)$ **then**
R19             $L_i = L_i \cup L_{i-1} \cup x.L_{i-1}$, $L_{i-1} = \emptyset$
R20 **if** $(c_1(x))$ **then**
R21     $L_i = L_i - \{(x,n)\}$
R22     $V[\text{process}\,(x.s)]++$

Figure 4: Pseudo Code for the Algorithm

- Reuse of class identifiers
  + If $[x] \subseteq L_m[e]$ and $\forall y \in [y] : [e] \subseteq L_m[y.s]$ then $[x]$ and $[y]$ can be given the same identifier,

– Receive 3 red messages before receiving this (marker type algo)

- Info that has to be sent along with message $x$.
  - If $\exists y \; : \; c_{m-1}(y) \; \wedge \; \text{process}\,(x.s) \neq \text{process}\,(y.s)$.
  – FIFO implementation does not need the whole matrix clock.

# References


[1] M. Ahuja. An implementation of F-channels. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):658–667, June 1993.

[2] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, January 1987.

[3] G. Bracha and S. Toeug. Distributed deadlock detection. *Distributed Computing*, 2(3):127–138, January 1987.

[4] G. Buckley and A. Silbershatz. An effective implementation of the generalized input-output construct of CSP. *ACM Transactions Programming Language Systems*, 2(2):223–235, April 1980.

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

[6] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous and asynchronous communication in distributed computations. Technical Report TR91.55, LITP, University Paris 7, September 1991.

[7] O. P. Damani and V. K. Garg. How to recover efficiently and asynchronously when optimism fails. Technical Report TR-PDS-1995-017, Parallel and Distributed Systems Laboratory, The University of Texas at Austin, 1995.

[8] E. W. Dijkstra. The distributed snapshot of K.M Chandy and L. Lamport. In M. Broy, editor, *Control Flow and Data Flow: Concepts of Distributed Programming*. Springer-Verlag, 1985.

[9] D. B. Johnson and W. Zwaenepeol. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181. ACM, 1988.

[10] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, January 1987.

[11] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications fo the ACM*, 21(7):95–114, July 1978.

[12] F. Mattern. Efficient distributed snapshots and global virtual time algorithms for non-FIFO systems. Draft Version, March 1990.

[13] V. V. Murty and V. K. Garg. Characterization of message ordering specifications and protocols. Technical Report TR-PDS-1996-004, Parallel and Distributed Systems Laboratory, The University of Texas at Austin, Febuary. 1996.

[14] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Inf. Process. Lett.*, 39(6):343–350, July 1991.

[15] A. Schiper, J. Eggli, and A. Sandoz. A new algorithm to implement causal ordering. In *Proceedings of the Third International Workshop on Distributed Algorithms*, pages 219–232. Springer-Verlay, 1989.

[16] K. Taylor. The role of inhibition in asynchronous consistent-cut protocols. In J.-C. Bermond and M. Raynal, editors, *Proc. of the 3rd International Workshop on Distributed Algorithms*, pages 280–291. Springer-Verlag, 1989.