Correctness of Mutual Exclusion Algorithms

Neelam Soundarajan and Ten-Hwang Lai
Computer and Information Science
Ohio State University, Columbus, OH 43210
e-mail: {neelam, lai}@cis.ohio-state.edu

Abstract

One way of characterizing non-token-based mutual exclusion (m.e.) algorithms is in terms of the underlying *information structure*. The information structure for a given m.e. algorithm specifies which particular processes interact with which other processes before entering their critical sections, and which processes they interact with when leaving their critical sections. By focusing on the information structure of an m.e. algorithm, we can compare different algorithms by comparing the respective information structures. Further, we can characterize the correctness of an m.e. algorithm in terms of conditions that its information structure must satisfy. In this paper, we propose a necessary and sufficient condition for ensuring the correctness of the m.e. algorithm and show that this condition is superior to the currently accepted condition.

1 Introduction

Mutual exclusion (m.e.) is a central problem in distributed systems. Over the years, numerous algorithms have been proposed for solving this problem. These algorithms may be classified [4, 8] as token-based and non-token-based. Our focus in this paper is on non-token-based algorithms. In these algorithms [1, 3, 5, 7], each process keeps some information about the status of the system such as which processes currently are waiting to enter their critical sections. When a process wishes to enter its critical section, it sends '*request*' messages to a certain set of processes; it then waits for '*grant*' messages from a certain set of processes granting it permission, at which point it enters its critical section; and when it leaves the critical section, it sends '*release*' messages to a certain set of processes. Information about the set of processes to which each process sends requests to or sends release messages to can be considered as the *information structure* [6] associated with this algorithm. Different non-token-based m.e. algorithms can be considered as special cases of a generalized algorithm, the differences between the different algorithms being captured entirely in the differences between the corresponding information structures.

Our goal in this paper is to consider necessary and sufficient conditions that the information structure of an m.e. algorithm must satisfy in order ensure the correctness of the algorithm. The main contributions of the paper are: First it shows that the standard condition [6, 4, 8] that is accepted in the literature to be the necessary and sufficient condition to guarantee correctness of the m.e. algorithm is, in fact, not necessary. This is proved by considering some simple and correct m.e. algorithms and showing that they do not satisfy the standard condition. Second, it proposes

a new condition and proves that this condition is both necessary and sufficient.

The paper is organized as follows: In Section 2, we specify the system model, describe the information structure associated with an m.e. algorithm, and consider the generalized m.e. algorithm. In Section 3, we present the currently accepted condition and show that it is not necessary for the correctness of the algorithm. In Section 4, we propose a new condition and prove that it is both sufficient and necessary. In the final section, we summarize the work and consider pointers for future work.

2 System model, Information structure, and Generalized M.E. algorithm

2.1 System model

Our system model is a fairly typical one. We assume that the system consists of N processes, each with a unique *process id*. Processes communicate with other processes via asynchronous message passing. Communication is assumed to be reliable; message transit time is arbitrary but finite. Processes exchange three types of messages: a *request* message, which we will denote RQ, indicates that the sending process wishes to enter its critical section (c.s.) and is requesting permission to do so from the receiving process; a *grant* message, denoted GR, means the sending process is granting permission to the receiver to enter the c.s; and a *release* message, RL, indicates that the sending process is 'releasing' the c.s., i.e., is leaving its c.s.

2.2 Information structure

The *information structure* consists of two sets for each process P_i; the *request set* R_i consisting of the id's of those processes to which P_i sends an RQ message when it wishes to enter its c.s; and the *inform set* I_i consisting of the id's of the processes to which P_i sends an RL message when it leaves its c.s. It is also useful to define the *status set* S_i as the set $\{j | i \in \mathsf{I}_j\}$. This information structure is the one proposed by Sanders [6].

2.3 Generalized m.e. algorithm

The generalized algorithm works as follows: Each process P_i has a boolean variable Free_i which is true if according to the information that P_i has, no process is in its critical section and false otherwise. When a process P_i wishes to enter its c.s., it sends a request message to each process P_j whose identity j is in its request set R_i; it then waits for GR messages from each of these processes before entering its critical section. When P_i leaves its critical section, it sends a RL message to every process whose id is in its inform set, I_i.

When a process P_i receives a request, it either immediately grants permission to the requesting process by sending a GR message, or adds the request to its collection of pending requests, depending on whether Free_i is true or false at the time the request was received. If P_i sends a GR message to another process P_j, then it sets Free_i to false if $j \in \mathsf{S}_i$, i.e., $i \in \mathsf{I}_j$.

When P_i receives a RL message, it does the following: Sets Free_i to true. Next, if its collection of

pending requests is non-empty, it picks the highest priority request from the collection and sends a GR message to the requesting process P_j, removes the request from its collection, and if $j \in \mathsf{S}_i$, sets Free_i to false; it repeats this until its collection is empty or Free_i is false. A standard time-stamp based notion of priority [2], with ties being resolved by an ordering among process id's, is used. The time-stamp and process id are part of each request.

There is a special case corresponding to $i \in \mathsf{R}_i$. In this case, when P_i decides to enter its critical section, before sending its requests to other processes in R_i, either it grants permission to itself if Free_i is true and sets Free_i to false if $i \in \mathsf{I}_i$, or it adds the request to its collection of pending requests if Free_i is false. Similarly, if $i \in \mathsf{I}_i$, when P_i leaves its critical section, before sending RL messages to the other processes in I_i, P_i does the following: sets Free_i to true and proceeds to grant the highest priority request(s) from its collection of pending requests, as described in the last paragraph.

The generalized algorithm above does not necessarily implement mutual exclusion correctly for arbitrary values of R_i and I_i. For example, if we defined each R_i to be just {i}, i.e., each process had to just get permission from itself before entering its c.s., m.e. would clearly be violated. At the other extreme, if we defined each R_i and I_i to contain the id's of *all* the processes in the system, m.e. would indeed be guaranteed but far more messages than necessary would be exchanged. It is therefore important to have a condition that is both necessary and sufficient to ensure that m.e. is implemented correctly.

3 Standard Condition for Correctness

In this section we first describe the condition that is accepted in the literature as the necessary and sufficient condition for correctness of the generalized m.e. algorithm. We will show that while this condition is sufficient, it is not necessary; indeed, that many simple and correct algorithms do not satisfy the condition.

Sanders [6] presents the following result: If $(i \in \mathsf{I}_i)$ for all i, then the following is a necessary and sufficient condition for the correctness of the m.e. algorithm:

$$[(\forall i.\, \mathsf{I}_i \subseteq \mathsf{R}_i) \wedge (\forall i, j.(\, (\mathsf{I}_i \cap \mathsf{I}_j \neq \Phi) \vee (j \in \mathsf{R}_i \wedge i \in \mathsf{R}_j)\,))\,] \qquad \text{(A)}$$

The sufficiency of (A) may be seen as follows: Consider two processes P_i, P_j. Suppose $(\mathsf{I}_i \cap \mathsf{I}_j \neq \Phi)$ is satisfied, say, $(k \in \mathsf{I}_i \cap \mathsf{I}_j)$. Then, since $(\mathsf{I}_i \subseteq \mathsf{R}_i)$ and $(\mathsf{I}_j \subseteq \mathsf{R}_j)$, P_i and P_j will each seek permission from P_k before entering its c.s.; P_k will, since $(k \in \mathsf{I}_i \cap \mathsf{I}_j)$, set its Free_k to *false* once it grants permission to either of them; hence P_k will *not* grant permission to the other until it receives a release message from the first.

The argument in the case of the other alternative, $(j \in \mathsf{R}_i \wedge i \in \mathsf{R}_j)$, is a bit more involved. Suppose P_i and P_j decide to enter their c.s.'s at about the same time. Each will grant itself permission to do so, set its Free to false (since $i \in \mathsf{I}_i$, $j \in \mathsf{I}_j$) then receive the request from the other; hence neither will grant permission to the other[1]. What if, say, P_j decides to enter its c.s. somewhat before P_i does so, and sends its request to P_i? In that case, P_i will grant permission to P_j and will *not* set Free_i to *false* (since $j \notin \mathsf{I}_i$). Next, P_i decides to enter its c.s., and P_i will grant its own request; but P_j will not grant P_i's request, hence m.e. will be ensured.

Consider now the necessity of (A). Suppose for a particular i, j, (A)'s second conjunct is not

[1]There is a danger of deadlock in this case. That can be handled in a standard manner with a slightly more involved approach. Here we will only concern ourselves with the m.e. issue.

satisfied. Then we can create a scenario in which m.e. is violated, as follows. Suppose $i \notin \mathsf{R}_j$. Let P_i first decide to enter its c.s.; it gets permission from itself to enter the c.s., then sends requests to all processes in $(\mathsf{R}_i - \mathsf{I}_i)$; all of them will grant it permission; moreover, since these processes are not in I_i, their Free variables will remain *true*. Free_i will be *false* but that will not matter since $i \notin \mathsf{R}_j$, so P_j will not seek its permission in the next step; note also that Free_j will be *true* although P_i obtained permission from P_j (since $j \notin \mathsf{I}_i$). Next, P_j decides to enter its c.s.; it gets permission from itself, then sends requests to all processes in $(\mathsf{R}_j - \mathsf{I}_j)$; all of them will grant it permission since their Free variables are *true*; moreover, their Free variables will remain *true* at this point. Finally P_i can request permission from all processes in $(\mathsf{I}_i - \{i\})$ and P_j from all processes in $(\mathsf{I}_j - \{j\})$. All of these will grant permissions and P_i and P_j will both enter their c.s., thereby violating m.e.

The arguments above for the sufficiency of (A) and the necessity of the second conjunct of (A) are based on the ones in Sanders [6]. Sanders further claims that the first conjunct of (A) is also necessary without providing any detailed justification. However, as we will see next, there are correct m.e. algorithms that do not satisfy this clause so it is not a necessary condition. Further, as we see below, the condition $(i \in \mathsf{I}_i)$ that the above result imposes, is not satisfied by many correct m.e. algorithms.

The intuition behind the requirement $(i \in \mathsf{I}_i)$ is that, in conjunction with the clause $(\mathsf{I}_i \subseteq \mathsf{R}_i)$, it represents the idea that it is reasonable for a process wishing to enter its c.s. to get permission from itself (in addition to whichever other processes are in R_i) and similarly to inform itself (in addition to whichever other processes are in I_i) when it leaves its c.s. to help ensure m.e. But this intuition is not well founded. For example, consider a centralized m.e. algorithm in which a single process, say P_0, makes *all* decisions about when each process may enter its c.s. In other words, $\mathsf{R}_i = \mathsf{I}_i = \{0\}$ for all i. The requirement $(i \in \mathsf{I}_i)$ is not satisfied by this algorithm (except for $i = 0$). But it clearly implements m.e. correctly since no process can enter its c.s. without first sending a request to, and getting permission from, P_0; and when P_0 grants permission to a process P_i, it will set Free_0 to *false* and will not grant permission to any other process to enter its critical section until its Free_0 becomes *true* which in turn will happen only when P_i finally leaves its c.s. and sends a RL message to P_0. Thus the requirement $(i \in \mathsf{I}_i)$ is not appropriate in general.

But even if $(i \in \mathsf{I}_i)$ is satisfied, the first conjunct of (A) need not be. Consider the following:

$\mathsf{I}_i = \{i, 0, 1\}$; $\mathsf{R}_i = \{0\}$

This structure meets the requirement $i \in I_i$, but does not satisfy the first conjunct of (A). But the corresponding algorithm is correct: Since $0 \in \mathsf{R}_i$, before P_i enters its c.s., it will request permission from P_0. If Free_0 is *true*, P_0 will grant permission, and since $0 \in \mathsf{I}_i$, will set Free_0 to *false*. Now if P_j wishes to enter its critical section, it too will send a request to P_0 (since $0 \in \mathsf{R}_j$); P_0 will not grant permission since Free_0 is *false*. This situation will change only when P_i leaves its critical section and sends a release message to P_0 (which it will since $0 \in \mathsf{I}_i$); at that point, P_0 will set Free_0 to *true*, and send a grant message to another process, possibly P_j, from which it has received a RQ message. This ensures m.e. Note that P_i will also send RL messages to itself and to P_1 when it leaves its c.s., but those messages will not have any effect because those processes do not get any requests so their collection of pending requests is always empty. Thus, the clause $(\mathsf{I}_i \subseteq \mathsf{R}_i)$ is not necessary for the correctness of the m.e. algorithm.

4 Necessary and Sufficient Condition

We cannot simply omit ($\mathsf{I}_i \subseteq \mathsf{R}_i$), because then we would face the following problem. Suppose $k \in (\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{R}_j \cap \mathsf{I}_j)$. We would expect P_k to ensure mutual exclusion between the critical sections of P_i and P_j. But there is a problem if there is some i' such that $k \in (\mathsf{I}_{i'} - \mathsf{R}_{i'})$. In this case, mutual exclusion can be violated as follows: First, $\mathsf{P}_{i'}$ sends requests to all processes in $\mathsf{R}_{i'}$ (which does not include k), gets grants from them, and enters its c.s. Next, P_i sends a request to P_k which grants the request and sets Free_k to *false*. Next, $\mathsf{P}_{i'}$ leaves its c.s., and sends release messages to all processes in $\mathsf{I}_{i'}$, *including* P_k; P_k, mistaking this for a release message from P_i, sets Free_k to *true*. Next, P_j decides to enter its c.s., and sends a request to P_k; since Free_k is *true* when P_k receives this request, it grants it (and sets Free_k to *false*). At this stage, both P_i and P_j have received grants from P_k and in the absence of some other process ensuring mutual exclusion between them, each will enter its c.s.

Thus the question is, how do we make sure that for each pair of processes, we have some process that ensures m.e. between them, without imposing a complete ban on such "bad" release messages from "third-party" processes such as $\mathsf{P}_{i'}$, i.e., without requiring ($\mathsf{I}_i \subseteq \mathsf{R}_i$) for all i? Let us first introduce some further notation. Define $\mathsf{B}_i = \mathsf{I}_i - \mathsf{R}_i$ and $\mathsf{BB} = \cup_i \mathsf{B}_i$. We will call processes whose indices are in B_i "bad" processes, these being the processes that may receive spurious release messages from P_i; BB is the set of *all* bad processes. Let NN be the set of all processes. Let $\mathsf{GG} = \mathsf{NN} - \mathsf{BB}$; thus GG is the set of all "good" processes. We can now state our main result:

Theorem: The necessary and sufficient condition for the correctness of the generalized m.e. algorithm is:

$$\forall i, j.[\{(\mathsf{R}_i \cap \mathsf{R}_j \cap \mathsf{I}_i \cap \mathsf{I}_j \cap \mathsf{GG}) \neq \Phi\} \vee \{(i \in (\mathsf{R}_i \cap \mathsf{R}_j \cap \mathsf{I}_i \cap \mathsf{GG})) \wedge (j \in (\mathsf{R}_i \cap \mathsf{R}_j \cap \mathsf{I}_j \cap \mathsf{GG}))\}] \ (B)$$

□

In other words, the necessary and sufficient condition for the correctness of the m.e. algorithm is that, for each pair i, j, one of the following must be true:

(b1). Either there exists k that is in each of $\mathsf{R}_i, \mathsf{R}_j, \mathsf{I}_i, \mathsf{I}_j$, and GG;
(b2). Or [(i is in each of $\mathsf{R}_i, \mathsf{R}_j, \mathsf{I}_i, \mathsf{GG}$) *and* ($j$ is in each of $\mathsf{R}_i, \mathsf{R}_j, \mathsf{I}_j, \mathsf{GG}$)]

A special case is when each B_i is empty, i.e., ($\mathsf{I}_i \subseteq \mathsf{R}_i$); in other words, there are no bad processes that receive spurious release messages. Then GG will be the set of all processes and $\cap\, \mathsf{GG}$ may be omitted from each clause of (B).

The intuition behind this condition is as follows: In order to ensure mutual exclusion between P_i and P_j, one of the following must be satisfied:
b1. There must be a process P_k that sends grants to both P_i and P_j, and keeps track of these grants, i.e., k in $(\mathsf{R}_i \cap \mathsf{R}_j \cap \mathsf{I}_i \cap \mathsf{I}_j)$; in addition this k must be a good process, so that it is not misled by spurious release messages from third-party processes. Or,
b2. P_i and P_j must each be responsible for sending grants to both P_i and P_j; and each must keep track of grants to itself. In addition, both P_i and P_j must be good so they are not misled by spurious releases.

Lemma: If $k \in (\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{GG})$, and if P_k sends a grant message GR to P_i, then Free_k will not become *true* again until P_i enters its c.s., completes it, leaves the c.s., sends a release message to P_k, and P_k processes it.

Proof: From generalized m.e. algorithm described in Section 2.3, it is clear that Free_k will be set to *true* only when P_k receives a release message RL from some process, say, $\mathsf{P}_{i'}$. We will show that

$\mathsf{P}_{i'}$ must in fact be the same as P_i. Now the only time that $\mathsf{P}_{i'}$ sends a RL is when it leaves its c.s. and at that point, it sends RL messages to all processes in $\mathsf{I}_{i'}$. Hence, if P_k receives such a message, k must be in $\mathsf{I}_{i'}$ and since $k \in \mathsf{GG}$, this implies $k \in \mathsf{R}_{i'}$. Before it can leave its c.s., $\mathsf{P}_{i'}$ must, of course, have *entered* it, and it can do so only after receiving grant messages from all processes in $R_{i'}$ including P_k. P_k could not have sent its GR message *before* it sent the GR message to P_i because if P_k had done so, it would have set Free_k to *false*, and would not then have sent a GR message to P_i (until Free_k was *true* which in turn would happen only upon receiving, by inductive assumption, the release message from $\mathsf{P}_{i'}$). P_k could not have sent the GR to $\mathsf{P}_{i'}$ *after* sending its GR to P_i because at that point Free_k would be *false*. The only possibility then is that $\mathsf{P}_{i'}$ must in fact be the same as P_i, not some other process.□

4.1 Proof of sufficiency

Next we will show that if either (b1) or (b2) is satisfied, the algorithm will implement m.e. correctly, thereby proving the sufficiency of the condition (B).

Consider a particular pair i, j of process indices. Suppose for this pair, that (b1) is satisfied. Suppose k is in $\mathsf{R}_i, \mathsf{R}_j, \mathsf{I}_i, \mathsf{I}_j$, and GG. Suppose P_i has got into its c.s. That means P_k must have sent P_i a GR message. When it did that, P_k must have set Free_k to *false* since k is in I_i. Now in order for P_j to enter its c.s., it must receive grant messages from all processes in R_j, including P_k. But P_k will not send a GR to P_j (or to any other process) until it receives a release message from some process. Since $k \in \mathsf{GG}$, by our lemma, this release message can only come from P_i, i.e., only when P_i leaves its c.s. Hence P_j cannot get into its critical section while P_i is still in its critical section.

Suppose now (b2) is satisfied for this i, j pair. Suppose P_i has already got its c.s. before P_j decides to enter its c.s. Since $i \in \mathsf{I}_i$, Free_i will be set to *false* and will not become *true* until P_i gets a release message; and since $i \in \mathsf{GG}$, this will not happen until P_i gets out of its c.s. and "sends itself a release message", i.e., sets Free_i to *true*. Therefore, given that $i \in \mathsf{R}_j$ so P_j needs a GR message from P_i before entering its c.s., P_j will not enter its c.s. until P_i leaves its. We can argue similarly if P_j is already in the c.s. before P_i decides to enter its c.s.

What if P_i and P_j decide to enter their c.s.'s at about the same time? One request will have a higher priority, say P_i's request. P_i will process this request before that of P_j since, according to the algorithm in Section 2.3, if $i \in \mathsf{R}_i$, as soon as P_i decides to enter its c.s., it will either grant its own request or add it to the pending requests depending on whether Free_i is *true* or *false*. And in the latter case, it will grant its own request before granting P_j's since the former has higher priority. So P_i will first grant its own request, set Free_i to *false* and the rest of the argument from the last para will now apply to show that P_j cannot enter its c.s. until P_i leaves its. So mutual exclusion is ensured. □

Note: Consider again the argument that P_i will process its own request first if that request has higher priority. What if P_i received P_j's request *just before* deciding to make its own request? This is not possible because if it has received P_j's request, P_i will choose a higher time stamp for its own request, so its request will have a lower priority than that of P_j. So if P_i's request has a higher priority, it must be because P_i has not yet received P_j's request at the time P_i decides to make its request. Of course if P_j's request is higher priority, P_i may receive that request before or after it processes its own; but P_j would have processed its own request before receiving P_i's and we can go through the same argument exchanging the roles of i and j to show that mutual exclusion is ensured.

4.2 Proof of necessity:

Suppose for a given i, j pair, that neither (b1) nor (b2) is satisfied. We will then show that it is possible to construct a scenario under which m.e. is violated, thereby proving that (B) is a necessary condition.

First we note the following: $\mathsf{R}_i = (\mathsf{R}_i - \mathsf{I}_i) \cup (\mathsf{R}_i \cap \mathsf{I}_i)$ (by set theory).
Further, $(\mathsf{R}_i \cap \mathsf{I}_i) = (\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{BB}) \cup (\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{GG})$ (by set theory, given $\mathsf{BB} \cup \mathsf{GG}$ = set of all processes).
And, $(\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{GG}) = [((\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{GG}) - \{i\}) \cup ((\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{GG}) \cap \{i\})]$ (set theory).
Thus R_i is the union of the four disjoint sets $(\mathsf{R}_i - \mathsf{I}_i)$, $(\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{BB})$, $((\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{GG}) - \{i\})$, and $((\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{GG}) \cap \{i\})$. R_j can similarly be written as the union of four disjoint sets. The reason for expressing R_i and R_j in this fashion is that it will help in the construction of the m.e.-violating scenario.

We will describe the construction of the scenario in a series of steps. In each step P_i and P_j will send requests to, and get grants from, the processes in the different sets that make up R_i and R_j respectively. We assume that at the start of the scenario no process is in its critical section, that the Free variables of all the processes are *true*, and that no process has any pending requests.

1. P_i decides to enter its c.s., sends RQ messages to, and gets GRs from all the processes in $(\mathsf{R}_i - \mathsf{I}_i)$. P_j also decides to enter its c.s., sends RQs to, and gets GRs from all the processes in $(\mathsf{R}_j - \mathsf{I}_j)$. Note that there is no danger of any of these processes not sending GR messages to P_i or P_j no matter in what order the requests from P_i and P_j reach them because first, all the Free variables are initially *true* and second, since these processes are not in $\mathsf{I}_i/\mathsf{I}_j$, they leave their Free variables *true* when they send their GR messages. (We will see in step 4 that either P_i's or P_j's requests may have to be postponed until the other's requests have been sent to certain processes and corresponding grants received. The other steps will be unaffected by this consideration.)

2. Next we might consider having P_i send RQs to all the processes in $(\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{BB})$. But this will lead to trouble if there is some k that is in $(\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{BB})$ that is also in $(\mathsf{R}_j \cap \mathsf{I}_j)$; because in this case, P_k would have set Free_k to *false* when it sends a GR message to P_i, and so will not send a GR when P_j makes its request and we would not be able to achieve our goal of violating m.e.

   To take care of this, we will make use of the fact that $k \in B$ and proceed as follows: Suppose k' is such that $k \in (\mathsf{I}_{k'} - \mathsf{R}_{k'})$; such a k' must exist because $k \in B$. Before P_i (or P_j) sends its RQs, we will start by having $\mathsf{P}_{k'}$ send RQs to all processes in $\mathsf{R}_{k'}$; these processes will send GRs to $\mathsf{P}_{k'}$ (because all Free variables are initially *true*); next, P_i will send a RQ message to P_k; P_k will respond with a GR (since $k \notin \mathsf{R}_{k'}$, so Free_k is *true* before P_k receives P_i's request); and P_k will set Free_k to *false*; next, we will have $\mathsf{P}_{k'}$ get out of its c.s. and send RL messages to all processes in $\mathsf{I}_{k'}$, including P_k; when P_k receives this message, it will set Free_k to *true*. At this point, all Free variables are *true* and P_i has received a GR from P_k. We repeat this process for each $k \in (\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{BB})$; and at the end of this, P_i would have received GR messages from all the processes in $(\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{BB})$, and all their Free variables will still be *true*. Next we go through the same procedure with P_j for all processes in $(\mathsf{R}_j \cap \mathsf{I}_j \cap \mathsf{BB})$.

   There is one special situation: What if the k' considered in the last paragraph turns out to be j? In other words, suppose k is in $(\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{BB} \cap (\mathsf{I}_j - \mathsf{R}_j))$ (and there is no other k' such that $k \in (\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{BB} \cap (\mathsf{I}_{k'} - \mathsf{R}_{k'}))$). In this case, there is no need to require P_j to do

any of the things we required of $\mathsf{P}_{k'}$. The point is that the reason for getting $\mathsf{P}_{k'}$ to enter its c.s. and exit it was to make sure that Free_k is *true* after it has sent a GR message to P_i since we may want P_j to be able to get a GR message from P_k. But here k is in $(\mathsf{I}_j - \mathsf{R}_j)$ which means that P_j does not need permission from P_k to enter its c.s. So there is no need to ensure that Free_k is *true* after P_k has sent the GR message to P_i. Indeed, in general, if $k \in (\mathsf{R}_i - \mathsf{R}_j)$, we do not need to worry about the value of Free_k after it has sent the GR message to P_i; similarly if $k \in (\mathsf{R}_j - \mathsf{R}_i)$, we do not need to worry about the value of Free_k after it has sent the GR message to P_j. Note also that if this situation occurs for P_i, i.e., $k \in (\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{BB} \cap (\mathsf{I}_j - \mathsf{R}_j))$, then it cannot simulatenously occur for P_j, i.e., we cannot for the same k, have $k \in (\mathsf{R}_j \cap \mathsf{I}_j \cap \mathsf{BB} \cap (\mathsf{I}_i - \mathsf{R}_i))$ (since the intersection of these two sets is empty).

3. P_i still needs to get grants from $(\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{GG})$ and P_j needs grants from $(\mathsf{R}_j \cap \mathsf{I}_j \cap \mathsf{GG})$.

   P_i now sends requests to all processes in $(\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{GG}) - \{i\}$ and receives GR messages from them. And P_j sends requests to all processes in $(\mathsf{R}_j \cap \mathsf{I}_j \cap \mathsf{GG}) - \{j\}$ and receives GR messages from them. Note that none of the processes in $(\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{GG}) - \{i\}$ can be in $(\mathsf{R}_j \cap \mathsf{I}_j \cap \mathsf{GG}) - \{j\}$ (because, otherwise, (b1) will be satisfied). Similarly, none of the processes in $(\mathsf{R}_j \cap \mathsf{I}_j \cap \mathsf{GG}) - \{j\}$ can be in $(\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{GG}) - \{i\}$. So these two sets of requests/grants can proceed (even simultaneously) without difficulty. Note also that a process k that is in $(\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{GG}) - \{i\}$ may well be in $(\mathsf{R}_j - \mathsf{I}_j)$; but P_j has already received grant messages from such processes, so there is no problem in P_i now receiving a GR from P_k and Free_k being set to *false*. Similarly if $k \in (((\mathsf{R}_j \cap \mathsf{I}_j \cap \mathsf{GG}) - \{j\}) \cap (\mathsf{R}_i - \mathsf{I}_i))$, P_i has already obtained its grant from P_k (in step 1), so P_j can get its grant from P_k without problem.

4. Next, suppose i is in $(\mathsf{R}_i \cap \mathsf{I}_i \cap \mathsf{GG})$; i may also be in $(\mathsf{R}_j - \mathsf{I}_j)$ (but not $(\mathsf{R}_j \cap \mathsf{I}_j)$ because then (b1) would be satisfied). In this case, if P_i were to start sending its requests before P_j does, we will have a problem because P_i would grant its own request immediately, and set Free_i to *false*; then it would not grant P_j's request when that arrives. We cannot get around this by assuming that P_i's request to itself is postponed while its requests to processes in $(\mathsf{R}_i - \mathsf{I}_i)$ etc. are sent out and grants received because, as noted before, if $i \in \mathsf{R}_i$ then P_i considers its own request to enter the c.s. as soon as it decides to enter the c.s. So in this case, we must have P_j send its RQ message to P_i (as part of step 1 for P_j since $i \in (\mathsf{R}_j - \mathsf{I}_j)$), P_i sends the GR to P_j; Free_i remains *true* since $i \in (\mathsf{R}_j - \mathsf{I}_j)$. Now P_i can decide to enter its c.s.; it will then grant itself permission to enter the c.s., set Free_i to *false*, then start sending out the other requests as described in earlier steps.

   Similar considerations apply if $j \in ((\mathsf{R}_j \cap \mathsf{I}_j \cap \mathsf{GG}) \cap (\mathsf{R}_i - \mathsf{I}_i))$; i.e., in this case we have to be sure to have P_i send its request to and receive the grant from P_j before P_j decides to enter its c.s. What if both of these situations apply? That cannot happen because then (b2) would be satisfied.

   What if $i \in (\mathsf{R}_i \cap \mathsf{R}_j \cap \mathsf{I}_j \cap \mathsf{GG})$ and $j \in (\mathsf{R}_i \cap \mathsf{R}_j \cap \mathsf{I}_i \cap \mathsf{GG})$ are both satisfied? Note first that in this case, i must not be in I_i and j must not be in I_j (else (b1) or both (b1) and (b2) will be satisfied). In this case, P_i will get its grant from itself in the first step (since $i \in (\mathsf{R}_i - \mathsf{I}_i)$) and similarly P_j will get its grant from itself in the first step. Following that, Free_i and Free_j will still be *true*, and P_i and P_j can get grants from each other in step 3 without any difficulties.

In summary, we expressed each of R_i and R_j as unions of four disjoint sets each, described a scenario in which in a sequence of four steps, one corresponding to each of the four sets making up R_i (and similarly R_j), P_i gets grants from all the processes in R_i and P_j gets grants from all the processes

in R_j. At this point, both processes can enter their respective critical sections since they have all the grant messages they need, and mutual exclusion will be violated. This shows that condition (B) is a necessary condition for ensuring mutual exclusion.

5 Discussion

The importance of mutual exclusion was, of course, recognized from the earliest days of concurrency. Over the years, many non-token-based m.e. algorithms have been proposed. The correctness of these algorithms is often quite difficult to establish because of the complex interactions between the processes, with each process getting permission from a specific set of processes before entering its critical section and informing another set of processes when leaving it, etc. By treating the different algorithms as special cases of a general algorithm, with the differences between the individual algorithms being captured in terms of the differences between their corresponding information structure, we reduce the task of establishing correctness of an m.e. algorithm to checking whether its information structure satisfies a particular condition.

But in order for this approach to be useful, the condition we check for must be both necessary and sufficient. Sufficiency is needed because otherwise there would be no guarantee of correctness of the corresponding algorithm even if its information structure satisfied the condition. Necessity is needed because without it, the information structure of correct m.e. algorithms may not satisfy the condition. In this paper, we showed that the condition that is accepted in the literature as necessary and sufficient is in fact not necessary; proposed a new condition for the information structure; and showed that the proposed condition is necessary and sufficient.

We conclude with a pointer to future work. Throughout the paper, and in the literature on information structures, the information structure is assumed to be *static*. One important extension to consider would be the possibility of a *dynamic* information structure [7], i.e., one that changes as the system executes, perhaps as the relative frequencies with which the different processes need to access their critical sections change. Analyzing these algorithms by direct operational reasoning, for example via the construction of scenarios, is likely to be even more difficult than in the case of algorithms with static structures; hence having a necessary and sufficient condition to guarantee correctness would be essential. One important point to note in such algorithms is that even if in the steady state they satisfy the requirement $I_i \subseteq R_i$ so that in the steady state the information structure would satisfy the condition of theorem 1, *during* the time that the information structure is changing, we cannot expect this requirement to be met. Hence we believe that conditions similar to that of theorem 2 might have to be satisfied during this time. We hope to address this question in our future work.

References

[1] O. Carvalho and G. Roucairol. On mutual exclusion in computer networks. *Comm. ACM*, 26:3–5, 1983.

[2] L. Lamport. Time, clocks, and ordering of events in a distributed system. *Comm. ACM*, 21(7):558–564, 1978.

[3] M. Maekawa. A $\sqrt{n}$ algorithm for mutual exclusion in decentralised systems. *ACM Trans. on Compu. Sys.*, 3:145–159, 1985.

[4] M. Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *Operating Sys. Rev.*, 25:47–49, 1991.

[5] G. Ricart and A.K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Comm. ACM*, 24:9–17, 1981.

[6] B. Sanders. Information structure of distributed mutual exclusion algorithms. *ACM Trans. on Computer Systems*, 5(3):284–299, 1987.

[7] M. Singhal. A dynamic information structure mutual exclusion algorithm. *IEEE Trans. on Par. and Dist. Sys.*, 3(1):121–125, 1992.

[8] M. Singhal. A taxonomy of distributed mutual exclusion. *J. Par. and Dist. Computing*, 18:94–101, 1993.