An \(\Omega(n \log n)\) Lower Bound on the Cost of Mutual Exclusion

Rui Fan
MIT Computer Science and AI Laboratory
Cambridge, MA 02139, USA
rfan@theory.csail.mit.edu

Nancy Lynch
MIT Computer Science and AI Laboratory
Cambridge, MA 02139, USA
lynch@theory.csail.mit.edu

ABSTRACT
We prove an \(\Omega(n \log n)\) lower bound on the number of non-busywaiting memory accesses by any deterministic algorithm solving \(n\) process mutual exclusion that communicates via shared registers. The cost of the algorithm is measured in the state change cost model, a variation of the cache coherent model. Our bound is tight in this model. We introduce a novel information theoretic proof technique. We first establish a lower bound on the information needed by processes to solve mutual exclusion. Then we relate the amount of information processes can acquire through shared memory accesses to the cost they incur. We believe our proof technique is flexible and intuitive, and may be applied to a variety of other problems and system models.

Categories and Subject Descriptors
F.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems

General Terms
Theory, Algorithms

Keywords
Mutual exclusion, Time complexity, Lower bound techniques, Information theory

1. INTRODUCTION
In the mutual exclusion (mutex) problem, a set of processes communicating via shared memory access a shared resource, with the requirement that at most one process can access the resource at any time. Mutual exclusion is a fundamental primitive in many distributed algorithms, and is also a foundational problem in the theory of distributed computing. Numerous algorithms for solving the problem in a variety of cost models and hardware architectures have been proposed over the past four decades. In addition, a number of recent works have focused on proving lower bounds for the cost of mutual exclusion. The cost of a mutex algorithm may be measured in terms of the number of memory accesses the algorithm performs, the number of shared variables it accesses, or other measures reflective of the performance of the algorithm in a multicomputing environment. In this paper, we study the cost of a mutex algorithm using the state change cost model, a simplification of the standard cache coherent model, in which an algorithm is charged only for performing shared memory operations causing a process to change its state. Let a canonical execution consist of \(n\) different processes, each of which enters the critical section exactly once. We prove that any deterministic mutex algorithm using registers must incur a cost of \(\Omega(n \log n)\) in some canonical execution. This lower bound is tight, as the algorithm of Yang and Anderson [13] has \(O(n \log n)\) cost in all canonical executions with our cost measure. To prove the result, we introduce a novel technique which is information theoretic in nature. We first argue that in each canonical execution, processes need to cumulatively acquire a certain amount of information. We then relate the amount of information processes can obtain by accessing shared memory to the cost of those accesses, to obtain a lower bound on the cost of the mutex algorithm. Our technique can be extended to show the same lower bound when processes are allowed access to comparison-based shared memory objects, in addition to registers. Furthermore, we believe that with some modifications, we can use the techniques to prove an \(\Omega(n \log n)\) lower bound on the cost of some canonical execution in the cache coherent model. A report on these results is in preparation.

We now give a brief description of our proof technique. Intuitively, in order for \(n\) processes to all enter the critical section without colliding, the “visibility graph” of the processes, formed by adding a directed edge from each process that “sees” another process, must contain a directed chain on all \(n\) processes. Indeed, if there exist two processes, neither of which sees the other, then an adversary can make both processes enter the critical section at the same time. To form a directed visibility chain, the processes must all together collect enough information to compute a permutation \(\pi \in S_n\). Such a permutation takes \(\Omega(n \log n)\) bits to specify. We show that in some canonical executions, each time the processes perform some memory accesses with cost \(C\), they gain only \(O(C)\) bits of information. This implies that in some canonical executions, the processes must incur \(\Omega(n \log n)\) cost. To formalize this intuition, we construct, for any permutation \(\pi \in S_n\), an execution \(\alpha_\pi\) in which a process ordered lower in \(\pi\) does not see any processes ordered higher...
in \( \pi \). In this execution, we can show that the processes must enter their critical sections in the order specified by \( \pi \). This implies that \( \alpha_\pi \) must be different for different \( \pi \), so that the set \( \{ \alpha_\pi \}_{\pi \in \Sigma_n} \) contains \( n! \) different executions. Then, we show that if the cost of execution \( \alpha_\pi \) is \( C_\pi \), we can encode \( \alpha_\pi \), that is, produce a string that uniquely identifies \( \alpha_\pi \), using \( O(C_\pi) \) bits. But since it takes \( \Omega(n \log n) \) bits to uniquely identify an element from a set of size \( n! \), some execution must have cost \( C_\pi = \Omega(n \log n) \).

The remainder of our paper is organized as follows. In Section 2, we describe related work on mutual exclusion and other lower bounds. In Section 3, we formally define the mutual exclusion problem and the state change cost model. We give a detailed overview of our proof in Section 4. In Section 5, we present an algorithm that, for every \( \pi \in \Sigma_n \), produces a different execution \( \alpha_\pi \) with some cost \( C_\pi \). We show in Section 6 how to encode \( \alpha_\pi \) as a string \( E_\pi \) of length \( O(C_\pi) \). In Section 7, we show \( E_\pi \) uniquely identifies \( \alpha_\pi \), by presenting a decoding algorithm that recovers \( \alpha_\pi \) from \( E_\pi \). Our main lower bound result follows as a corollary of this unique decoding. Lastly, in Section 8, we summarize our results and techniques, and discuss some future work and open problems.

2. RELATED WORK

Mutual exclusion is a seminal problem in distributed computing. Starting with Dijkstra's work in the 1960's, research in the area has progressed in response to, and has sometimes driven, changes in computer hardware and the theory of distributed computing. For interesting accounts of the history of this problem, we refer the reader to the excellent book by Raynal [12] and survey by Anderson, Kim and Herman [4].

The performance of a mutual exclusion algorithm can be measured in a variety of ways. An especially relevant measure for modern computer architectures is memory contention. In [1], Alur and Taubenfeld prove that for any non-trivial mutual exclusion algorithm, some process must perform an unbounded number of memory accesses to enter its critical section. This comes from the need for some processes to busywait until the process currently in the critical section exits. Therefore, in order for a mutex algorithm to scale, it must ensure that its busywaiting steps do not congest the shared memory. Local-spin algorithms were proposed in [8] and [11], in which processes busystay only on local or cached variables, thereby relieving the gridlock on main memory. Local-spin mutex algorithms include [13], [10] and [3], among many others. In particular, the algorithm of Yang and Anderson [13] performs \( \Omega(n \log n) \) remote memory accesses in an execution in which \( n \) processes each complete their critical section once. The state change cost model we propose discounts certain busywaiting steps by charging an algorithm only for memory accesses that change a process's state. Yang and Anderson's algorithm also has \( \Omega(n \log n) \) cost using the state change model.

A number of lower bounds exist on the memory complexity for solving mutual exclusion [6]. Recently, considerable research has focused on proving time complexity (number of memory accesses) lower bounds for the problem. Cypher [7] first proved that any mutual exclusion algorithm must perform \( \Omega(n \log \log \log n) \) total remote memory accesses in some canonical execution. An improved, but non-amortized lower bound by Anderson and Kim [2] showed that some process must perform at least \( \Omega(\log n) \) remote memory accesses.

However, this result does not give a nontrivial lower bound for the total number of remote accesses performed by all the processes. The techniques in these papers involve keeping the set of processes contending for the critical section "invisible" from each other, and eliminating certain processes when they become visible. Our technique is fundamentally different in that we do not eliminate processes, nor do we try to keep all processes invisible from each other. Intuitively, we show that in order for \( n \) processes to solve mutual exclusion, they must collectively gather enough information to compute a directed "visibility chain", in which each process sees the next process in the chain. We then relate the amount of information processes can acquire with the cost they must incur to obtain that information. Information-based arguments have also been used by Jayanti [9] and Attiya and Hendler [5], among others, though in quite different forms.

3. MODEL

In this section, we define the formal model for proving our lower bound. We first describe the general computational model, then define the mutual exclusion problem, and the state change cost model for computing the cost of an algorithm.

3.1 The Shared Memory Framework

In the remainder of this paper, fix an integer \( n \geq 1 \). A system consists of a set of processes \( p_1, \ldots, p_n \), and a collection \( L \) of shared variables. A shared variable consists of a type and an initial value. In this paper, we restrict the types of all shared variables to be multi-reader multi-writer registers. Each process is modeled as a deterministic automaton, consisting of a state set, an initial state, and a deterministic transition function that computes a step (e.g., a memory access) for the process to execute based on its current state. We write \( \delta(s, i) \) for the transition function of process \( p_i \), where \( s \) is a state of \( p_i \). For every \( i \in [n] \), a read step of \( p_i \) is \( \text{read}(\ell) \), where \( \ell \in L \), and represents a read by process \( p_i \) on register \( \ell \). We let \( \text{own}(\text{read}(\cdot, i)) = i \) be the process performing the read. A write step of \( p_i \) is \( \text{write}(\ell, v) \), where \( \ell \in L, v \in V \), and represents a write of value \( v \) by process \( p_i \) on register \( \ell \). Here, \( V \) is some arbitrary fixed set. We let \( \text{own}(\text{write}(\cdot, i)) = i \). We define \( \text{val}(\text{write}(\cdot, v)) = v \) to be the value written by a write step. Let \( e \) be a step. We define \( \text{type}(e) \) to be \( R \) if \( e \) is a read step, and \( W \) if \( e \) is a write step. We say that a step \( \text{read}(\ell) \) or \( \text{write}(\ell, \cdot) \) accesses register \( \ell \). An algorithm specifies a process automaton \( p_i \), for each \( i \in [n] \).

A system state is a tuple consisting of the states of all the processes and the values of all the registers. We assume that all systems have a default initial state \( s_0 \), consisting of the initial values of all the registers and the initial states of all the processes. An execution consists of a (possibly infinite) alternating sequence of system states and process steps. That is, an execution is of the form \( s_0 p_1 s_1 p_2 \ldots S_n \), where each \( s_i \) is a system state, and each \( e_i \) is a step by some process. For any execution \( \alpha \), we define
\( \alpha(t) = s_{0e1}...e_{it} \) to be the length \( t \) prefix of \( \alpha^t \). We let the projection of \( \alpha \) on a process \( p_i \) be the sequence \( \alpha[i \text{ \ if \ it \ exists} \) consisting only of the states and steps of \( p_i \). If \( \alpha \) is a finite execution, we define \( st(\alpha) \) to be the final system state of \( \alpha \), and, for \( i \in [n] \), \( st(\alpha, i) \) to be the state of process \( p_i \) in \( st(\alpha) \).

We say an execution \( \beta \) is an extension of \( \alpha \) if \( \beta \) contains \( \alpha \) as a prefix. For convenience, we sometimes represent an execution simply as a sequence of process steps, \( e_1e_2... \). Since we assume that the system has a unique initial state, and that all the processes and variables are deterministic, we can uniquely identify the system state after any sequence of process steps, and therefore both representations of executions are equivalent. Given an algorithm \( \mathcal{A} \), we let \( \text{execs}(\mathcal{A}) \) denote the set of all executions of \( \mathcal{A} \).

Given a permutation \( \pi \in S_n \), we think of \( \pi \) as a bijection from \([n]\) to itself, and we write \( \pi^{-1}(i) \) for the element that maps to \( i \) under \( \pi \), for \( i \in [n] \). We write \( i \leq_j j \) if \( \pi^{-1}(i) \leq \pi^{-1}(j) \); that is, \( i \) equals \( j \), or \( i \) comes before \( j \) in \( \pi \). Lastly, if \( S \subseteq [n] \), we write \( \min(S) \) for the minimum element in \( S \), where elements are ordered by \( \leq \).

3.2 The Mutual Exclusion Problem

Let \( \mathcal{A} \) be an algorithm. For each process \( p_i \), the steps of \( p_i \) contains the following critical steps: try, enter, exit, rem. For any critical step \( e \), we define type(\( e \)) = 0. For simplicity, we assume that these steps, and the read and write steps of \( p_i \) are the only steps performed by \( p_i \). We say a process \( p_i \) is in its trying section if its last critical step in an execution is try. We say it is in its critical section if the last critical step is enter. We say it is in its exit section if the last critical step is exit. Finally, we say it is in its remainder section if the last critical step is rem or there are no critical steps.

We say that \( \mathcal{A} \) solves the livelock-free mutual exclusion problem if any finite execution \( \alpha \in \text{execs}(\mathcal{A}) \) satisfies the following properties.

- **Well Formedness:** Let \( p_i \) be any process, and consider the subsequence \( s \) of \( \alpha \) consisting only of \( p_i \)'s critical steps. Then \( s \) forms a prefix of the sequence try, enter, exit, rem, ...  
  - **Mutual Exclusion:** For any two processes \( p_i, p_j \) if the last occurrence of a critical step by \( p_i \) in \( \alpha \) is enter, then the last critical step by \( p_j \) in \( \alpha \) is not enter.

  In addition, every fair execution\(^4\) \( \alpha \) of \( \mathcal{A} \) satisfies:

- **Livelock Freedom:** For any process \( p_i \), and any try \( j \) step in \( \alpha \), there exists a later step enter \( i \), for some process \( p_j \). In addition, for any exit \( i \), there exists a later step rem \( i \), for some process \( p_k \).

The well formedness condition says that every process progresses cyclically through its trying, critical, exit and remainder sections. The mutual exclusion property says that no two processes can be in their critical sections at the same time. The livelock freedom property says that if a process is in its trying section, then eventually, some process, perhaps not the same one, enters its critical section. Additionally, if a process is in its exit section, then eventually some process enters its remainder section. This means that the overall system always makes progress.

In addition to satisfying the three properties above, we want the mutex algorithm to be nontrivial, so that each process may request to enter the critical section anytime it is in the remainder section. For simplicity, we assume that the initial step of each process \( p_i \) is try.

3.3 The State Change Cost Model

In this section, we define the state change cost model for measuring the cost of a shared memory algorithm. In [1], it was proven that the cost of any shared memory mutual exclusion algorithm is infinite if we count every shared memory access. To obtain a more meaningful measure for cost, researchers have focused on models in which some memory accesses are free. Two important models that have been studied are the distributed shared memory (DSM) model and the cache coherent (CC) model. We define a new cost model, called the state change (SC) cost model, which is a simplification of the cache coherent model. Informally, the state change cost model charges an algorithm for a memory access only when the process performing the access changes its state. In particular, we charge the algorithm for each write performed by a process\(^6\). Additionally, the state change cost model allows unit cost busywaiting reads, but only on one variable at a time. However, the model is sufficiently generous that it permits algorithms to incur \( O(n \log n) \) in all canonical executions. Formally, the cost model is defined as follows.

**Definition 3.1. The State Change Cost Model** Let \( \mathcal{A} \) be an algorithm, and let \( \alpha = s_{0e1}...e_{it} \in \text{execs}(\mathcal{A}) \) be a finite execution.

1. Let \( p_i \) be a process, and \( j \in [t] \). We define \( sc(\alpha, i, j) \) to be 1 if \( e_j \) is a shared memory access step by \( p_i \), and \( st(\alpha(j - 1), i) \neq st(\alpha(j), i) \); it is 0 otherwise.

2. We define the cost of execution \( \alpha \) to be \( C(\alpha) = \sum_{i \in [n]} \sum_{j \in [t]} sc(\alpha, i, j) \).

Notice that this model charges only for steps of \( p_i \) accessing the shared memory, and not for the critical steps of \( p_i \) (even though \( p_i \) may change its state after a critical step). The cost of \( \alpha \) is simply the number of times a process changes state following shared memory steps, summed over all the processes. The SC cost model allows a limited form of busywaiting reads with bounded cost. For example, suppose the value of a register \( \ell \) is currently 0, and process \( p_i \) repeatedly reads \( \ell \), until its value becomes 1. As long as \( \ell \)'s value is not 1, the process does not change its state, and thus, continues to read \( \ell \). Then, the algorithm is charged one unit for all reads up to when \( p_i \) reads \( \ell \) as 1.

The cost of most practical algorithms is higher in the state change cost model than in the standard cache coherent model. For example, the CC model allows bounded cost busywaits on multiple registers at the same time, while the SC model does not. However, we believe the SC cost model is a mathematically clean and interesting model which allows a clear demonstration of our proof techniques, with few nonessential technical complications.

4. OVERVIEW OF THE LOWER BOUND

In this section, we give a detailed overview of our lower bound proof. Let \( \mathcal{A} \) be any livelock-free mutual exclusion algorithm. The proof consists of three steps, which we call the construction step, the encoding step, and the decoding step. The construction step builds a finite execution
\( \alpha_e \in \text{execs}(A) \) for each permutation \( \pi \in S_n \), such that different permutations lead to different executions. The encode step produces a string \( E_{\pi} \) of length \( O(C(\alpha_e)) \) for each \( \alpha_e \). The decode step reproduces \( \alpha_e \) using only input \( E_\pi \). Since each \( E_{\pi} \) uniquely identifies one of \( n! \) different executions, some \( E_{\pi} \) must have length \( \Omega(n \log n) \). Therefore, the corresponding execution \( \alpha_e \) must have cost \( O(n \log n) \).

Fix a permutation \( \pi = (\pi_1, \ldots, \pi_n) \in S_n \). We say that a process \( p_i \) has lower (resp., higher) index (in \( \pi \)) than process \( p_j \) if \( i \) comes before (resp., after) \( j \) in \( \pi \). Roughly speaking, the construction step works as follows. We construct, in \( n \) stages, \( n \) different finite executions, \( \alpha_1, \ldots, \alpha_n \in \text{execs}(A) \), where \( \alpha_n = \alpha_e \). In each \( \alpha_i \), only the first \( i \) processes in the permutation, \( p_{\pi_1}, \ldots, p_{\pi_i} \), take steps. Thus, \( \alpha_i \) is a solo execution by process \( p_{\pi_i} \). Each process runs until it has completed its critical and exit sections. We will show that the processes complete their critical sections in the order given by \( \pi \), that is, \( p_{\pi_1} \) first, then \( p_{\pi_2} \), etc., and finally, \( p_{\pi_n} \). Next, we construct execution \( \alpha_1+1 \) in which process \( p_{\pi_1+1} \) also takes steps, until it completes its critical and exit sections. \( \alpha_1+1 \) is constructed by starting with \( \alpha_1 \), and then inserting steps by \( p_{\pi_1+1} \), in such a way that \( p_{\pi_1+1} \) is not seen by any of the the lower indexed processes \( p_{\pi_1}, \ldots, p_{\pi_i} \). Informally, this is done by placing some of \( p_{\pi_1+1} \)'s writes immediately before writes by lower indexed processes, so that the latter writes overwrite any trace of \( p_{\pi_1+1} \)'s presence. Of course, there are many possible ways to make \( p_{\pi_1+1} \) unseen by the lower indexed processes. For example, we can place all of \( p_{\pi_1+1} \)'s steps after all steps by lower indexed processes. But doing that, we may not be able to encode the execution using only \( O(C_e) \) bits. The key to the construction is to produce an execution that both ensures higher indexed processes are unseen by lower indexed ones, and that can also be encoded efficiently.

While the above describes the intuition for the construction step, it is not exactly how we actually perform the construction. Instead of directly generating an execution \( \alpha_i \) in stage \( i \), we actually generate a set of metasteps \( M_i \) and a partial order \( \leq_i \) on \( M_i \), in stage \( i \). A metastep consists of three sets of steps, the read steps, the write steps, and the \textit{winning} step, which is a write step. All steps access the same register. A process performs at most one step in a metastep. We say a process is \textit{encoded} in the metastep if it takes a step in the metastep, and we say the \textit{winner} of the metastep is the process performing the winning step. The purpose of a metastep is to hide the presence of all processes contained in the metastep, except possibly the winner. Given a set of metasteps \( M_i \) and a partial order \( \leq_i \) on \( M_i \), we can generate an execution by first ordering \( M_i \) using any total order consistent with \( \leq_i \), to produce a sequence of metasteps. Then, for each metastep in the sequence, we expand the metastep into a sequence of steps, consisting of the write steps of the metastep, ordered arbitrarily, followed by the winning step, followed by the read steps, ordered arbitrarily. Notice that this sequence hides the presence of all processes except possibly the winner. The sequence of steps resulting from totally ordering \( M_i \) and then expanding each metastep is an execution which we call a \textit{linearization} of \((M_i, \leq_i)\). Of course, there may be many total orders consistent with \( \leq_i \), and many ways to expand each metastep, leading to many different linearizations. However, we will show that for the particular \( M_i \) and \( \leq_i \), we construct, all linearizations are “essentially the same”. For example, in any linearization, all processes \( p_{\pi_1}, \ldots, p_{\pi_i} \) complete their critical sections once, and they do so in that order. It is the set \( M_i \) and partial order \( \leq_i \), generated at the end of stage \( i \) in the construction step, that we eventually encode in the encoding step. The reason we construct a partial order of metasteps instead of directly constructing executions is that the partial order \( \leq_i \) affords us more flexibility in stage \( i+1 \) when we add steps by process \( p_{\pi_{i+1}} \), which in turn leads to a more efficient encoding. We can show that all linearizations of \((M_n, \leq_n)\) have the same cost, and we call this cost \( C_e \).

We now describe the encoding step. In this step, an encoding algorithm takes as input \((M_n, \leq_n)\), produced after stage \( n \) of the construction step. For any process \( p_i \), we can show that all the metasteps containing \( p_i \) in \( M_n \) are totally ordered in \( \leq_n \). Thus, for any metastep containing \( p_i \), we can say the metastep is \( p_i \) \textit{in} \( j \)'th metastep, for some \( j \). The encoding algorithm uses a table with \( n \) rows and an infinite number of columns. In the \( j \)'th row and \( i \)'th column of the table, which we call cell \( T(i, j) \), the encoder records what process \( p_i \) does in its \( j \)'th metastep. However, to make the encoding short, we only record, roughly speaking, the type, either read or write, of the step that \( p_i \) performs in its \( j \)'th metastep. In addition, if \( p_i \) is the winner of the metastep, we also record a \textit{signature} of the entire metastep. The signature basically contains a count of how many processes in the metastep perform read steps, and how many perform write steps (including the winning step). Note that the signature does not specify which processes read or write in the metastep, nor the register or value associated with any step. Now, if there are \( k \) processes involved in a metastep, the total number of bits we use to encode the metastep is \( O(k) + O(\log k) = O(k) \). Indeed, for each non-winner process in the metastep, we use \( O(1) \) bits to record its step type. For the winner process, we record its step type, and use \( O(\log k) \) bits to record how many readers and writers are in the metastep. Notice that the cost to the algorithm for performing this metastep is also \( O(k) \). Informally, this shows that the size of the encoding is proportional to the cost incurred by the algorithm. The final encoding of \((M_n, \leq_n)\) is formed by iterating over all the metasteps in \( M_n \), each time filling the table as described above. Then, we concatenate together all the nonempty cells in the table into a string \( E_n \).

Lastly, we describe how, using \( E_n \) as input, the decoding step constructs an execution \( \alpha_e \) that is a linearization of \((M_n, \leq_n)\). Roughly speaking, at any time during the decoding process, there exists an \( m \in M_n \) such that the decoder algorithm has produced a linearization of all the metasteps that \( \leq_n m \) (as well as some metasteps that are incomparable to \( m \)). We say all such metasteps have been \textit{executed}. This linearization is a prefix of \( \alpha_e \). Using this prefix, the decoder tries to find a minimal (with respect to \( \leq_n \)) unexecuted metastep, which it then executes, by appending the steps in the metastep to the prefix; then the decoder restarts the decoding loop. To find a minimal unexecuted metastep, the decoder applies the transition function \( \delta \) of \( A \) to the prefix to compute, for each process \( p_i \), the step that \( p_i \) takes in the smallest unexecuted metastep containing \( p_i \). Then, by reading \( E_n \), the decoder finds the signature, that is, the number of readers and writers, of some minimal unexecuted metasteps. Suppose the decoder finds the signature of a metastep \( m' \) accessing some register \( \ell \) (recall that all steps in a metastep access the same register), and suppose

\[ \text{Note that even though our discussion involves } \pi, \text{ the decoder does not know } \pi. \text{ The only input to the decoder is the string } E_\pi. \]
the signature indicates that \( m' \) contains \( r \) read steps and \( w \) write steps. Then, since the decoder knows each process’s next step, it can check whether there are \( r \) processes whose next step reads \( \ell \), and \( w \) processes whose next step writes to \( \ell \). If so, the decoder executes all these steps on \( \ell \). That is, it appends them to the current execution, placing all the write steps before all the read steps, and placing the winning write — the write performed by the process whose cell in \( E_n \) contains the signature — last among the writes. We can show that these steps are exactly the steps contained in \( m' \), and that \( m' \) is a minimal unexecuted metastep. After executing \( m' \), the decoder tries to find a minimal metastep \( \not\leq m' \). By repeating this process until it has read all of \( E_n \), the decoder produces an execution that is a linearization of \( \langle M_n, \preceq_n \rangle \).

5. THE CONSTRUCTION STEP

5.1 Description of the Construction

In this section, we present the algorithm for the construction step. For the remainder of this paper, fix an arbitrary permutation \( \pi \). For the remainder of this paper, we fix an arbitrary \( \pi \). In stage \( i \), CONSTRUCT builds \( M_i \) and \( \preceq_i \) by calling the procedure GENERATE with inputs \( M_{i-1} \) and \( \preceq_{i-1} \) (constructed in stage \( i-1 \)) and \( \pi_i \). We define \( M_0 = \emptyset = \emptyset \). We now describe GENERATE\( (M_n, \preceq_n, \pi_i) \). For simplicity, we write \( M \) for \( M_n \) and \( \leq \) for \( \preceq_n \), and \( j \) for \( \pi_i \) in the remainder of this section. GENERATE proceeds in a loop, and terminates when process \( p_j \) performs its \( \text{rem}_j \) action, that is, completes its critical and exit sections. In each iteration of the loop, we compute the next step \( e_j \) that \( p_j \) takes. Then, we either insert \( e \) into an existing metastep of \( M \), or create a new metastep containing only \( e \).

Let \( m' \) be the metastep that was modified or created in the previous iteration of the loop. Then, roughly speaking, we will insert \( e \) into a metastep if we can find a write metastep that \( \not\preceq e \), and which accesses the same register as \( e \). If we cannot find such a metastep, we create a new metastep for \( e \). If \( e \) is a write step, we make \( e \) the winning step of the new (write) metastep. Then we check whether there are any read metasteps in \( M \) on \( e \)'s register that \( \not\preceq e \). If so, we order all such metasteps before \( e \)'s metastep. We also add these metasteps to the preread set of \( e \)'s metastep. If \( e \) is a read step, we simply create a new read metastep containing \( e \). Now, we set \( m' \) to be the metastep we just modified or created, add \( m' \) to \( M \) if \( m' \) is a newly created metastep, and change \( \preceq \) to order \( m' \). Then, we let execution \( \alpha \) be a linearization of all metasteps \( \preceq m' \). Using \( \alpha \), we can now compute \( p_j \)'s next step, and the loop continues.

We now describe the construction in more detail. We will refer to specific line numbers in Figure 1 in angle brackets. For example, \((8)\) refers to line 8. In \((8)\), GENERATE begins by creating a new metastep \( m \) containing only the critical step \( \text{try}_j \), indicating that \( p_j \) starts in its trying section. We add \( m \) to \( M \), and set \( m' \) to \( m \). \( m' \) keeps track of the metastep we created or modified during the previous iteration of the main loop. We then begin the main repeat loop, which ends when \( p_j \) performs its \( \text{rem}_j \) step. The loop begins at \((10)\) by setting \( \alpha \) to be a linearization of all metasteps \( \preceq m' \). This is computed by the function \( \text{PLIN}(M, \preceq m') \). Using \( \alpha \), we can now compute \( p_j \)'s next step \( e \) as \( \alpha(e, j) \). Let \( k \) be the register that \( e \) accesses, if \( e \) is a read or write step.

We split into two cases, depending on \( e \)'s type. If \( e \) is a write step \((13)\), then we set \( m_w \) to be the minimum write metastep in \( M \) that accesses \( k \), and that \( \preceq m' \). For any registerto, we can show that the set of all write metasteps accessing that register are totally ordered. Thus, if \( m_w \) exists, it is unique. When \( m_w \) exists, we insert \( e \) into \( m_w \), by adding \( e \) to \( \text{write}(m_w) \) \((16)\). The idea is that this hides \( p_j \)'s presence, because \( e \) will immediately be overwritten by another write step in \( m_w \) when we linearize any set of metasteps including \( m_w \). Next, we add the relation \((m', m_w) \to \preceq \)8, indicating

If we insert \( e \) into a metastep \( m \), then we will still refer to the metastep as \( m \); that is, even though \( \text{own}(m) \) changes, \( m \) retains the same “name”. We do this for notational convenience.
1: procedure CONSTRUCT(σ)  
2: \( M_0 \leftarrow \emptyset \); \( \geq_n \leftarrow \emptyset \)  
3: for \( i = 1, \ldots, n \) do  
4: \( (M_i, \leq_i) \leftarrow \text{GENERATE}(M_{i-1}, \leq_{i-1}, n_i) \)  
5: end for  
6: return \( M_n \), and the reflexive, transitive closure of \( \leq_n \)  
7: procedure GENERATE(\( M, \leq_n \))  
8: \( m \leftarrow \text{new metastep}; \text{crit}(m) \leftarrow \text{try}_j \)  
9: \( M \leftarrow M \cup \{m\}; \ m' \leftarrow m \)  
10: repeat  
11: \( \alpha \leftarrow \text{PLAN}(M, \leq_n, m'); \ e \leftarrow \delta(\alpha, j); \ell \leftarrow \text{reg}(e) \)  
12: switch \( \text{type}(e) \)  
13: case \( \text{type}(e) = \text{W} \)  
14: \( m_w \leftarrow \min \{\mu; \mu \in M \land (\text{reg}(\mu) = \ell) \land (\text{type}(\mu) = \text{W}) \land (\mu \notin \leq_n)\} \)  
15: if \( m_w \) exists then  
16: \( \text{win}(m_w) \leftarrow \text{win}(m_w) \cup \{e\} \)  
17: \( m' \leftarrow \{\ell \in \text{Reg}(m_w) \mid m' \leftarrow m \} \)  
18: else  
19: \( m \leftarrow \text{new metastep}; \text{win}(m) \leftarrow \{e\} \)  
20: \( m' \leftarrow m \)  
21: end if  
22: \( \text{reg}(m) \leftarrow \text{reg}(m) \cup \{e\} \)  
23: \( M \leftarrow M \cup \{m\} \)  
24: end switch  
25: until \( \alpha \leftarrow \text{PLAN}(M, \leq_n, m'); \ e \leftarrow \delta(\alpha, j); \ell \leftarrow \text{reg}(e) \)  
26: end procedure  

5.2 Properties of the Construction  

We now present some results about the construction step. We give mostly proof sketches, and defer full proofs to a full version of the paper.  

**Lemma 5.2**. For any \( i \in [n] \), \( \leq_i \) defines a partial ordering on \( M_i \).  

**Lemma 5.3**. Let \( i \in [n] \), and let \( \ell \in L \) be any register. Then, all the write metasteps in \( M_i \) that access \( \ell \) are totally ordered in \( \leq_i \).  

Both lemmas can be verified by induction on the main loops in procedures CONSTRUCT and GENERATE.  

**Lemma 5.4**. Let \( 1 \leq i \leq j \leq k \leq n \), let \( \alpha_j \) be a linearization of \((M_j, \leq_j)\), and let \( \alpha_k \) be a linearization of \((M_k, \leq_k)\). Then \( \alpha_j \upharpoonright \alpha_k = \alpha_k \upharpoonright \alpha_k \).  

This lemma says that a process \( p_{e_i} \) cannot distinguish between a linearization from the \( j \)’th to the \( k \)’th stage of CONSTRUCT, for \( i \leq j \leq k \). This in turn implies that for \( i \leq j \leq k \), \( p_{e_i} \) cannot tell if process \( p_{e_j} \) is present or not in any linearization of \((M_k, \leq_k)\). This can be shown by considering how metasteps are created, ordered and linearized. Indeed, when a higher indexed process \( p_{e_j} \) performs a write step, this write step is placed either in a metastep so that it is immediately overwritten by the write of a lower indexed process, or placed after all reads by lower indexed processes on the same register (these reads become prereads of \( p_{e_j} \)’s write). Therefore, lower indexed processes never see any values written by higher indexed ones, and cannot tell if they are present. This lemma can be used to show the following.  

**Theorem 5.5**. Let \( 1 \leq i \leq n \). Then in any linearization of \((M_i, \leq_i)\), each process \( p_{e_1}, \ldots, p_{e_i} \) completes its critical section, and they do so in that order.  

**Proof**. (Sketch) The fact that processes complete their critical section follows from the livelock freedom property of the mutex algorithm, and Lemma 5.4. To show that they complete them in the order \( \pi \), consider the minimum \( j \) such
that there exists a process $p_{s_{b}}$ that enters its critical section before $p_{s_{a}}$, and $j < k$. Now, consider the moment when $p_{s_{a}}$ performs its $enter_{s_{a}}$ step. At this point, $p_{s_{a}}$ has not performed its $enter_{s_{a}}$ step yet. Now, since processes $p_{s_{a}}, \ldots, p_{s_{b}}$ all do not see $p_{s_{j}}$, by Lemma 5.4, then we can pause $p_{s_{j}}$, and run $p_{s_{a}}, \ldots, p_{s_{b}}$, and be guaranteed that $p_{s_{j}}$ will eventually perform its $enter_{s_{j}}$ step. However, then $p_{s_{j}}$ and $p_{s_{b}}$ have both performed their enter steps, but not their exit steps, which violates the mutual exclusion property, a contradiction. Thus, processes must enter their critical sections in the order $\pi$.

5.3 Additional Properties of the Construction

Finally, we present several lemmas which we use in Section 7.2 to prove the correctness of the decoding step. We begin with the following definitions.

**Definition 5.6.** Let $N \subseteq M$. We say $N$ is a prefix of $M$ if $\forall m \in N$, we have $\mu \in M \land (\mu \subseteq m) \subseteq N$.

Thus, a prefix of $M$ is a union of chains of $(M, \preceq)$.

**Definition 5.7.** Let $N$ be a prefix of $M$, and let $i \in \ell$ and $i \in [n]$. Define:

1. $\Gamma_{N}^{i}(N) = \{ \mu \mid (\mu \not\in N) \land (type(\mu) = \emptyset) \land (i \in own(\mu)) \}$.
2. $\Gamma_{N}^{i}(N, \ell) = \{ \mu \mid (\mu \not\in N) \land (reg(\mu) = \emptyset) \land (type(\mu) = \emptyset) \land (i \in own(\mu)) \}$.
3. $\Gamma_{N}^{i}(\ell, i) = \{ \mu \mid (\mu \not\in N) \land (reg(\mu) = \emptyset) \land (type(\mu) = \emptyset) \land (i \in own(\mu)) \}$.
4. $\Gamma_{N}^{i}(\ell, i) = \{ \mu \mid (\mu \not\in N) \land (reg(\mu) = \emptyset) \land (type(\mu) = \emptyset) \land (i \in own(\mu)) \}$.
5. $\gamma_{N}^{i}(\ell, i) = \min_{\ell} \Gamma_{N}^{i}(\ell, i)$, or $\perp$ if $\Gamma_{N}^{i}(\ell, i) = \emptyset$.
6. $\gamma_{N}^{i}(\ell) = \min_{\ell} \Gamma_{N}^{i}(\ell, i)$, or $\perp$ if $\Gamma_{N}^{i}(\ell) = \emptyset$.

These functions define various subsets of $M$ related to $N$. For example, $\Gamma_{N}^{i}(N)$ is the set of write metasteps not in $N$ that contain $p_{i}$, $\Gamma_{N}^{i}(N, \ell)$ is the set of write metasteps not in $N$ accessing $\ell$, and $\gamma_{N}^{i}(\ell)$ is the minimum write metastep not in $N$ containing $p_{i}$ that accesses $\ell$, if it exists, etc. We have the following lemmas.

**Lemma 5.8.** Let $N$ be a prefix of $M$, and let $i \in [n]$. Define $m^{*} = \gamma_{N}(\ell, i)$, $m_{i} = \gamma_{N}(i, \ell)$, and suppose $m^{*} \not\subseteq \emptyset$, $m_{i} \not\subseteq \emptyset$, and $type(step(m_{i}, i)) = \emptyset$. Then $m^{*} = m_{i}$.

That is, if $p_{i}$ does a write step in the first write metastep on $\ell$ not in $N$ that contains $p_{i}$, then that metastep equals the first write metastep on $\ell$ not in $N$.

**Proof.** (Sketch) To prove this lemma, let $j = own(win(m^{*}))$. Notice that $p_{j}$ is the minimum index process (w.r.t. $\pi$) contained in $m^{*}$. We claim that $i \geq j$. Indeed, suppose $i < j$. Then, procedure $CONSTRUCT$ created the steps for process $p_{i}$ in iteration $\pi - 1(i)$, before it created the steps for process $p_{j}$ in iteration $\pi - 1(j)$. Consider iteration $\pi - 1(j)$ of $CONSTRUCT$, and the iteration of the main loop of $GENERATE$ where $p_{i}'s$ step in $m^{*}$ was created; $e$ is a write step. Since the write metastep $m_{i}^{*}$ has already been constructed, then in (14), $GENERATE$ will find $m_{i}^{*}$ as the minimum write metastep on $\ell$ not in $N$, and sets $m_{w} = m_{i}^{*}$; in (16), it adds $e$ to the write set of $m_{i}^{*} = m_{i}^{*}$. Thus, the minimum index of a process in $m^{*}$ is at most $i$, a contradiction. We have shown that $i \geq j$. Consider iteration $\pi - 1(i)$ of $CONSTRUCT$, and the iteration of the main repeat loop of $GENERATE$ where $p_{i}'s$ step in $m_{w}^{*}$ was created. Since $e$ is a write step, then in (14), $GENERATE$ sets $m_{w} = m_{w}^{*}$, and in (16), it adds $e$ to the write set of $m_{w}^{*} = m_{w}^{*}$. Thus, the lemma holds. □

**Lemma 5.9.** Let $N$ be a prefix of $M$, and let $i \in \ell$ and $i \in [n]$. Define $m_{i}^{*} = \gamma_{N}(\ell, i)$, and suppose $m_{i}^{*} \not\subseteq \emptyset$ and $type(step(m_{i}^{*}, i)) = \emptyset$. Then $m_{i}^{*} = \min_{\ell} \{ \mu \mid (\mu \not\in N) \land (type(\mu) = \emptyset) \land (\mu \not\in SC(PLIN(M, \preceq, \mu), \mu, i)) \}$.

Here, we used the functions $PLIN(M, \preceq, m)$ and $SC(\alpha, m, i)$, as defined in Figure 1. Recall that the former function returns a partial linearization of $(M, \preceq)$, consisting of all metasteps $\preceq m$. The latter function returns true exactly when process $p_{i}$, whose state is as in the final state of $\alpha$, will change its state upon reading the value written by write metastep $m$. This lemma states that, if $p_{i}$ does a read step in the first write metastep on $\ell$ not in $N$ that contains $p_{i}$, then that metastep equals the first write metastep on $\ell$ not in $N$ whose value causes $p_{i}$ to change its state.

The proof of this lemma is very similar to the proof for Lemma 5.8. If $e$ is the step $p_{i}$ takes in $m_{i}^{*}$, then in iteration $\pi - 1(i)$ of $CONSTRUCT$, and the iteration of $GENERATE$ where $e$ was created, (30) of $GENERATE$ adds $e$ to the read set of the minimum write metastep on $\ell$ not in $N$ that causes $p_{i}$ to change its state (if this metastep exists). We omit the details.

**Lemma 5.10.** Let $N$ be a prefix of $M$, and let $\ell \in \ell$ and $i \in [n]$. Suppose $\Gamma_{N}(\ell, i) \not= \emptyset$, $\Gamma_{N}(\ell, i) \not= \emptyset$, and $\Gamma_{N}(\ell) \not= \emptyset$. Then $\min_{\ell} \Gamma_{N}(\ell) \in \text{pred}(\gamma_{N}(\ell, i))$.

That is, if there exist write metasteps on $\ell$ not in $N$, but none of them contain $p_{i}$, and $p_{i}$ is contained in read metasteps on $\ell$ not in $N$, then the largest read metastep on $\ell$ not in $N$ containing $p_{i}$ is contained in the preread set of the minimum write metastep on $\ell$ not in $N$. This can be verified by considering lines (21–24) of $GENERATE$.

6. THE ENCODING STEP

6.1 Description of the Encoding

We can show that all linearizations of $(M, \preceq)$ have the same cost in the state change cost model, say $C$. In this section, we describe an algorithm that uniquely encodes $(M, \preceq)$ as a string of length $O(C)$. The encoding uses a two-dimensional grid of cells, with $n$ columns and an infinite number of rows. We fill some of the cells with strings. The contents of the cell in column $i$ and row $j$ denoted by $T(i, j)$. It represents the type of operation process $p_{i}$ performs in its $j$'th metastep, and possibly a count of how many reads, writes and prereads are in $p_{i}$'s $j$'th metastep. The complete encoding $E_{c}$ is produced by concatenating all nonempty cells $T(1, \cdot)$ (in order), then appending all nonempty cells $T(2, \cdot)$, etc., and finally appending all nonempty cells $T(n, \cdot)$. The encoder uses the helper function $\text{next}(T, i)$, which returns how many nonempty cells there are in column $i$ of $T$. Please see Figure 2 for the pseudocode.

The encoder works by iterating over every metastep $m \in M$. Suppose first that $m$ is a write metastep. Then for every step $e$ in $\text{read}(m) \cup \text{write}(m)$, let $p_{i}$ be the process that performs $e$, and let $m_{i}$ be $p_{i}$'s $q$'th metastep. The encoder writes $e$'s type, either $R$ or $W$, in cell $T(p_{i}, q)$. For the winning step, we record in $T(p_{i}, q)$ not only that it is a write, but also the signature of the metastep (9). The signature of $m$ records the number of read and write steps (including the winning step) in $m$, as well as the size of $m$'s preread set $\text{pread}(m)$. It is a string of the form $\text{PR}_{R}R_{\text{pr}}R_{W}$.
1: procedure Encode(M, ≤)
2: for all m ∈ M do
3: switch
4: case type[m] = ∀:
5: for all e ∈ read(m) \ uwrite(m)
6: p → own(e); q ← PC(p, m);
7: T(p, q) ← type(e)
8: end for
9: p ← own(rew(m)); q ← PC(p, m);
10: T(p, q) ← ∀, Pr|read(m) \ uwrite(m) + 1|
11: end case
12: end for

Figure 2: Encoding M and ≤ as a string E_α.

If m is a read metastep, then it contains only one step, in read(m), say by process p. If m is a preread of any other (write) metastep µ, then we write Pr in T(p, q) (13). Otherwise, we write SR (14). Lastly, if m is a critical step, we write C in T(p, q).

In the remainder of this paper, let E be the string Encode outputs given input M and ≤.

6.2 Properties of the Encoding

We now state some results about the efficiency of the encoding.

LEMMA 6.1. Let α1 and α2 be two different linearizations of (M, ≤). Then α1 and α2 have the same cost in the state change cost model.

Since all linearizations have the same cost, we let C be this cost.

THEOREM 6.2. The length of E_α is O(C).

Proof. (Sketch) For any metastep m ∈ M, we compare the number of bits used to encode m, and the cost for the algorithm to execute m in the state change cost model. The cost to encode a read metastep is O(1), since we record either SR or PR for the metastep. Consider a write metastep m, and suppose m contains k processes. We first ignore the cost to encode the number of prereads in m. Let r be the number of reads in m, and w be the number of writes. If a process p is not the winner of m, then we use O(1) bits to encode the type of p’s step. If p is the winner, then we encode its type, and we also use O(log r) + O(log w) = O(k) bits encode the number of reads and writes. Now we count the bits used to encode the number of prereads. We notice that a read metastep can only appear as a preread in one write metastep. Thus, by counting every read metastep twice, we can account for the cost to encode all the prereads. Thus, if m is any metastep containing k processes, then the amortized cost of encoding m is O(k). We now claim that the algorithm incurs O(k) cost in the state change model to execute m. Indeed, all write steps have unit cost. A read is only added to a write metastep if the metastep’s value causes the reader to change its state. A read in a read metastep also has unit cost, as we have already argued the reader must change its state after the step. Therefore, |E_α| is proportional to the cost to the algorithm C.

7. THE DECODING STEP

7.1 Description of the Decoding

We now describe the decoding algorithm. We first give an informal description of the algorithm, and then give a more detailed one. The decoder creates an execution α, and repeatedly appends steps to α, until α equals a linearization of (M, ≤). Note that this means there is a particular total order on M consistent with ≤, and a particular way to expand each metastep in M via Seq, that produces an execution that equals α; that is, α equals the output of LIN(M, ≤) for some (nondeterministic) execution of LIN(M, ≤). Now, each time the decoder appends a sequence of steps, those steps are exactly the steps contained in some m ∈ M. We say the decoder has executed m. m has the property that it is a minimal (w.r.t. ≤) unexecuted metastep in M. Thus, the decoding algorithm essentially runs a loop where in each iteration, it finds and executes a minimal unexecuted metastep of M.

We now describe the decoding algorithm in more detail. Please see Figure 3 for the pseudo-code. We first describe the variables in Decode. α is the execution that the decoder builds. done ⊆ [n] is the set of processes that have completed their critical and exit sections. For i ∈ [n], pc_i is the number of metassteps the decoder has executed that contain p_i, and e_i is p_i’s step in the minimal unexecuted metastep containing p_i. We call e_i process p_i’s pending step. At certain points in the decoding, the decoder may not yet know the pending steps of some processes. If the decoder knows the pending step of process p_i, then it places i in wait. For i ∈ L, R_i (resp., W_i) contains the set of processes whose pending step is a read step (resp., write step) on register i. For any i ∈ [n], if i ∈ PR, then p_i has performed its last read on i in M. Lastly, if sig_i ≠ α, then sig_i contains the signature of the minimum unexecuted write metastep on i.

Each iteration of the main repeat loop of Decode consists of two sections; from (637), and from (3845). The purpose of the first section is to find the pending step of each process. The purpose of the second section is to find a set of processes whose pending steps together form the steps of a minimal unexecuted metastep. Consider any i /∈ done \ wait. That is, p_i has not finished its critical and exit sections yet, and the decoder does not know its pending step. In (7), the decoder increments pc_i, and calls the helper function.getTransaction(E, i, pc_i) to determine the type of p_i’s pending step. Recall that E is stored as the concatenation of type and signature information about each process’s steps in the metassteps containing that process. The decoder adds i to wait. It then switches based on the value of step.

Consider the case step_i = w. In (1112), the decoder loops until p_i’s next step is a write step. Let ℓ be the register that p_i’s pending step e_i writes to. In (15), the decoder adds i to W_ℓ. Also, if step_i contains a signature sig_i, the decoder sets sig_i to makesig(sig_i, i). If sig_i = PR or RW, then

\footnote{Note that we do this because p_i may do some critical steps before a write step.}
Next, consider the case \( \text{step}_1 = \text{SR} \). Then, \( p_i \)'s next unexecuted metastep is a read metastep. The decoder executes this metastep, and removes \( i \) from \( \text{wait} \) (31).

If \( \text{step}_i = \mathcal{C} \), then \( e_i \) is a critical step. The decoder appends \( e_i \) to \( \sigma \), and removes \( i \) from \( \text{wait} \). Finally, if \( \text{step}_i = \$ \), then \( p_i \) has finished all its steps in \( M \), and the decoder adds \( p_i \) to \( \text{done} \).

We now describe what \( \text{Decode} \) does in (38 – 45). In (38), the decoder finds some \( \ell \) for which it knows the signature. It then checks that the sizes of \( R, W \), and \( PR \) match the signature (39). If so, it sets \( \beta \) to be the concatenation, in an arbitrary order, of all the write steps \( e_i \), for \( i \in W \setminus \text{sig}(v) \). It sets \( \gamma \) to be the concatenation of all read steps \( e_i \), for \( i \in R \). Then, it appends \( \beta \circ e_{\text{sig}(v)} \circ \gamma \) to \( \alpha \). The decoder removes \( R, W, PR \) from \( \text{wait} \) (43), to indicate it needs compute pending steps for these processes in the next iteration of the repeat loop. It also resets \( sig, R, PR \) and \( W \).

The decoder performs the repeat loop until \( \text{done} = [n] \), indicating all processes have finished their critical and exit sections.

### 7.2 Properties of the Decomposition

In this section, we show that \( \text{Decode} \) produces an execution that is a linearization of \( (M, \preceq) \). In particular, we show that each time the decoder appends a sequence of steps to \( \alpha \), those steps are exactly the steps of some minimal unexecuted metastep \( m \), and that all the steps in \( \text{write}(m) \) are appended before \( \text{write}(m) \), which is appended before all the steps in \( \text{read}(m) \). Furthermore, we show that in each iteration of the main loop of \( \text{Decode} \), the decoder does append some steps to \( \alpha \). Thus, eventually \( \alpha \) equals a linearization of \( (M, \preceq) \). The proof uses induction on the execution of the decoder. Below, we define what it means for the decoder to behave correctly up to some point in its execution.

**Definition 7.1.** Let \( j \geq 0 \), and let \( e_1, e_2, \ldots, e_j \) be the value of \( \alpha \) at (6) in the \( j \)th iteration of the main repeat loop of \( \text{Decode} \). Note that each \( e_i \in \{0\} \), is a step. We say the decoder is correct up to iteration \( j \) if the following hold at (6) of iteration \( j \):

1. \( \alpha \in \text{SEQ}(m_1) \circ \ldots \circ \text{SEQ}(m_n) \), where \( m_1 \in \text{min}\{\mu | \mu \in M\} \), and \( \forall i \in \langle u - 1 \rangle; m_{i + 1} \in \min_{\geq} \{\mu | \mu \in M \land \mu \neq m_i \} \}. \) Let \( N = \bigcup_{i \in [u]} m_i \).
2. For all \( \ell \in L \), let \( m^* = \gamma^*(N, \ell) \). Then we have \( R, W, \subseteq \text{read}(m^*) \), \( W \subseteq \text{write}(m^*) \) and \( PR \subseteq \text{pread}(m^*) \).
3. For all \( \ell \in L \), if \( \text{sig} \neq \varepsilon \), then \( \text{sig} \circ \gamma^*(N, \ell) \), \( \text{sig} \circ \gamma^*(N, \ell) \).

Thus, the decoder is correct up to iteration \( j \) if three types of conditions are satisfied. First, the sequence of steps \( \alpha \) the decoder has produced up to iteration \( j \) is belongs to \( \text{SEQ}(m_1) \circ \ldots \circ \text{SEQ}(m_n) \). Note that this means there are some (nondeterministic) executions of \( \text{SEQ} \) on inputs \( m_1, \ldots, m_n \), whose concatenation equals \( \alpha \). In addition, \( m_1 \) is a minimal metastep of \( M \), and each \( m_i \) is a minimal metastep not preceding \( m_{i-1} \). This implies that \( \alpha \) is a prefix of a linearization of \( (M, \preceq) \). Also, we have that \( N = \bigcup_{i \in [u]} m_i \) is a prefix of \( M \). Second, for any \( \ell \in L \), the sets \( R, W, \subseteq \text{pread}(m^*) \).

**Lemma 7.2.** Let \( j \geq 0 \), and suppose \( \text{Decode} \) is correct up to iteration \( j \). Then \( \text{Decode} \) either terminates after iteration \( j \), or it is correct up to iteration \( j + 1 \).
Proof. (Sketch) Suppose the decoder does not terminate in iteration \( j \). We first verify that the conditions in the correctness definition continue to hold after lines (6–37). Later, we verify they hold after (38–45). Let \( i \in [n] \). Fix any \( \ell \in L \), and let \( m^* = \gamma_i^*(N, \ell) \). We consider four cases, depending on the type of step. If \( \text{step} = W \), then \( i \) is added to \( W_t \), where \( \ell = \text{reg}(e_i) \), and \( e_i \) is \( p_i \)'s pending step. Since we do not append any steps to \( \alpha \) in this case, condition 1 holds. To verify condition 2, note that the minimum write metastep on \( \ell \) containing \( p_i \) and not in \( N \), namely \( \gamma_i^*(N, \ell) \), is equal to \( m^* \), by Lemma 5.8. Thus, \( p_i \) performs a write step in \( m^* \), and so \( W_t \cup \{i\} \subseteq \text{write}(m^*) \cup \text{win}(m^*) \). If \( \text{step} \) contains a signature, then by the same argument, this is the signature for metastep \( m^* \), so condition 3 holds.

If \( \text{step} = R \) and \( i \) is added to \( R_t \), then \( \ell = \text{reg}(e_i) \), where \( e_i \) is \( p_i \)'s pending step. Let \( m^* = \gamma_i^*(N, \ell) \). Then \( m^* = \gamma_i^*(N, \ell) \), by Lemma 5.8. Also, since reading \( \text{val}(e_{\text{sig}_i}) \) causes \( p_i \) to change its state, we have by Lemma 5.9 that \( m^* = \gamma_i^*(N, \ell) \). Thus, \( p_i \) takes a read step in \( m^* \), so \( R_t \cup \{i\} \subseteq \text{read}(m^*) \), and condition 2 (and also 1 and 3) holds. If \( \text{step} = \text{Pr} \), then \( i \) is added to \( P_t \), where \( \ell = \text{reg}(e_i) \) and \( e_i \) is \( p_i \)'s pending step. By the correctness of the decoder, \( e_i \) is \( p_i \)'s final read on line \( f \in M \). Thus by Lemma 5.10, \( i \) belongs to \( \text{pred}(m^*) \), and condition 2 holds. Lastly, if \( \text{step} = \text{SR} \), it is easy to show that condition 1 (and 2, 3) continues to hold.

We now verify that the correctness conditions hold after lines (38–45). Suppose that the test on \( \text{Sr} \) succeeds. Let \( k = \text{sig}_{\text{R}^*} \); then \( k \) is the winner of \( m^* \). Because of conditions 2 and 3, we must have \( R_t = \text{read}(m^*) \), \( W_t = \text{write}(m^*) \cup \text{win}(m^*) \), and \( P_t = \text{pred}(m^*) \). Thus, \( R_t \cup W_t = \text{own}(m^*) \), and \( \text{concat}(\bigcup_{\ell \in R_t} e_{\ell}) \subseteq \text{seq}(m^*) \). Lastly, we have that \( m^* \in \min \{\mu | \{\mu \text{ is M} \land \mu \neq \text{m}_a\} \} \), because for every process in \( \text{own}(m^*) \), their pending step is their step in \( m^* \), and also, every metastep in \( \text{pred}(m^*) \) has been executed. Thus, condition 1 (and 2, 3) continues to hold, and the lemma is proved.

Lemma 7.2 shows a safety property of \( \text{Decode} \); that is, the decoder never executes a nonminimal unexecuted metastep. We now show a liveness property, that in every iteration of the main loop of the decoder, it executes some metastep.

Lemma 7.3. Suppose \( \text{Decode} \) is correct up to iteration \( j \), where \( j \geq 1 \). Then \( \text{Decode} \) either terminates or it executes some metastep in iteration \( j \).

Let \( m_a \) be the last metastep the decoder executed before iteration \( j \). Then we can easily show that the decoder executes some minimal \( m \) not preceding \( m_a \) in iteration \( j \). Informally, this is because the pending step of every process contained in \( m \) is its step in \( m \), and so \( \text{Decode} \) either appends a read or critical step in (27), (31) or (33), or the test on (39) succeeds and the decoder appends to \( \alpha \) in (42).

Together, Lemmas 7.2 and 7.3 imply the following.

Theorem 7.4. Let \( \alpha \) be the execution produced by \( \text{Decode} \). Then \( \alpha \) is a linearization of \( (M, \preceq) \).

7.3 A Lower Bound for Mutual Exclusion

Theorem 7.5. Let \( A \) be any livelock-free mutual exclusion algorithm. Then in some \( \alpha \in \text{exec}(A) \) in which processes \( p_1, \ldots, p_n \) all complete their critical sections once, we have \( C(\alpha) = \Omega(n \log n) \).

Proof. For each \( \pi \in S_n \), let \( (M^*, \preceq^*) = \text{Construct}(\pi) \), \( E_* = \text{Encode}(M^*, \preceq^*) \), and \( \alpha_* = \text{Decode}(E_*) \). By Theorem 7.4, \( \alpha_* \) is a linearization of \( (M^*, \preceq^*) \). Thus, by Theorems 5.5, we have that \( p_1, \ldots, p_n \) all complete their critical sections once in \( \alpha_* \), and they complete them in the order \( \pi \). Therefore, for \( p_1, p_2 \in S_n \), \( p_1 \not\preceq p_2 \) or \( p_2 \not\preceq p_1 \). Thus, the (deterministic) algorithm \( \text{Decode} \) produces \( n! \) different outputs, on input from the set \( \{E_\pi \mid \pi \in S_n \} \). Therefore, there exists \( \pi \in S_n \) such that \( |E_\pi| = \Omega(\log(n!)) = \Omega(n \log n) \). By Theorem 6.2, we have that \( |E_\pi| = O(C(\alpha_*)) \). Thus, we have \( C(\alpha_*) = \Omega(n \log n) \), and the theorem is proved.

8. CONCLUSIONS

In this paper, we have established a lower bound of \( \Omega(n \log n) \) memory accesses in the state change cost model for solving \( n \) process mutual exclusion. Our proof technique uses an information theoretic characterization of a necessary condition for solving mutual exclusion, and relates this to the information processes can gain through access to shared registers. Our proof technique can be extended to accommodate stronger memory primitives. We believe it also extends with minor modifications to the cache coherent cost model. We believe our proof technique is intuitive and flexible, and may be used to establish lower bounds for other problems for which current techniques are complex or insufficient.

9. REFERENCES