Pending Conflicts Make Progress Impossible

Petr Kuznetsov[a], Pierre Sutra[b], Guillermo Toyos-Marfurt[a,b,*]

[a] *LTCI, Télécom Paris, Institut Polytechnique de Paris, 91120 Palaiseau, France*
[b] *SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, 91120 Palaiseau, France*

Abstract

In this work, we study progress conditions for commutativity-aware, linearizable implementations of shared objects. Motivated by the observation that commuting operations can be executed in parallel, we introduce *conflict-obstruction-freedom*: a process is guaranteed to complete its operation if it runs for long enough without encountering step contention with conflicting (non-commuting) operations. This condition generalizes obstruction-freedom and wait-freedom by allowing progress as long as step contention is only induced by commuting operations. We prove that conflict-obstruction-free universal constructions are impossible to implement in the asynchronous read-write shared memory model for $n \geq 3$ processes. This result exposes a fundamental limitation of conflict-aware universal constructions: the mere invocation of conflicting operations imposes a synchronization cost. Progress requires eventual resolution of pending conflicts.

Keywords: Distributed computing, Shared memory, Commutativity

1. Introduction

Context. Synchronizing operations that access shared data is a crucial task in distributed systems. The conventional safety property, linearizability [1], requires that each operation takes effect instantaneously, at some point in time between its invocation and response. Jointly with safety, one also needs to define a progress property specifying under which circumstances an operation should return. The strongest of such guarantees is wait-freedom [2], which demands that every operation completes in a finite number of steps of its invoking process.

We consider the standard asynchronous shared-memory model, where processes communicate via read-write primitives and may crash at any time. In this model, it is well-established that wait-free linearizable implementations of many standard data structures are impossible [2, 3]. Because linearizability is usually retained to preserve usability, much research has focused on exploring weaker progress guarantees. A prominent example is obstruction-freedom [4]: a

*Corresponding author

process that runs *alone* (without encountering steps of concurrent processes) for long enough is guaranteed to complete its operation. Although obstruction-free algorithms guarantee progress only without contention, they often behave as wait-free under typical scheduling assumptions and admit simpler implementations [5]. In particular, one can build an obstruction-free *universal construction*, which is a universal recipe to create linearizable objects, using just read-write communication primitives [4, 6].

Motivation. To ensure they apply to a wide variety of shared objects, progress properties are typically agnostic to their semantics. However, a key observation is that the main cost in synchronizing concurrent accesses comes from computing a total order among *conflicting* (non-commuting) operations [1]. In contrast, commuting operations [7], such as increments on a shared counter, peek operations on a list, or updates to distinct keys in a hash table, may proceed independently, in a coordination-free manner. Some works leverage this to reduce the cost of data replication [8], boost transactional memory [9], and define smart locks [10]. In light of these results, a natural question is whether commuting operations may enjoy stronger progress than regular, conflicting ones.

Contributions. We introduce a novel progress condition, *conflict-obstruction-freedom* (COF). COF guarantees that a process that encounters no concurrent steps of conflicting operations must be able to complete its operation. This progress condition generalizes obstruction-freedom in a natural sense. Moreover, for objects that only provide commuting operations, COF is identical to wait-freedom. For example, consider a collection of *queues*, where each of them is implemented by an obstruction-free algorithm [4] and each process can only enqueue its own identifier. The resulting composition of queues is conflict-obstruction-free, as any two concurrent operations are in conflict if and only if they are applied to the same queue.

Further, we investigate whether conflict-obstruction-freedom admits a universal construction in the read-write shared memory model. This model captures minimal communication assumptions in a concurrent system, making it a natural starting point for studying a new progress condition. When all operations conflict, one can build obstruction-free universal constructions [4]. On the other hand, when all operations commute, wait-freedom is trivially achievable [11]. Thus, our study aims to characterize whether such a construction is possible in the intermediate cases, that is, when some operations commute and some do not.

Our main result is negative. We show that such a universal construction can be used to solve a weaker version of consensus in which processes must terminate when only processes with the same input remain active. We prove that this task is impossible to solve using only reads and writes for three or more process.

This impossibility exposes a fundamental limitation in leveraging operation commutativity in concurrent systems. The mere presence of conflicting operations incurs a synchronization cost, even if these operations later become inactive. While commutativity eases coordination, it does not eliminate the

need for synchronization: once a conflict appears, the system must eventually resolve it to make progress. Thus, to date, obstruction-freedom remains the strongest known progress condition that enables a universal construction in the read-write model.

The model and results presented in this work have been formalized in Lean 4, available at [12]. In particular, every numbered lemma, theorem, and corollary is accompanied by a corresponding machine-checked proof. The Lean formalizations and proofs were developed with the assistance of Claude Opus 5.5. We provide these formalizations as supplementary material, offering an additional layer of machine-checked verification for the results of this work. They complement the natural-language proofs in the paper, which remain the primary presentation of the mathematical arguments.

2. Preliminaries

Objects. A sequential object is defined as a tuple (Q, q_0, O, R, σ), where Q is a set of *states*, $q_0 \in Q$ is the *initial state*, O is a set of *operations*, R is a set of *responses*, and $\sigma : O \times Q \to R \times Q$ is a *state transition function* specifying the *response* and *next state* for each operation and state. Given a state q and a sequence of operations $s \in O^*$, we denote by $s(q)$ the final state after applying the operations in s sequentially starting from q. For two sequences s and s', $s \cdot s'$ denotes their concatenation. Given two operations $o, o' \in O$, we say that they *commute* in a state q if $o \cdot o'(q) = o' \cdot o(q)$ and each operation returns the same result in both sequences. The object defines a symmetric *conflict relation*, written $\asymp \subseteq O \times O$, where $o \asymp o'$ if there exists a state q in which o and o' do not commute. When this happens, we say that operations o and o' *conflict*.

System Model. We consider a system with n asynchronous processes, $\{p_i\}_{i \in [n]}$, that communicate through shared memory using atomic read-write registers. Processes are deterministic and may fail by crashing, i.e., stopping prematurely and taking no further steps; an arbitrary number of processes may crash. There is no bound on relative process speeds, and no assumptions are made about timing or scheduling fairness.

Algorithms and executions. To *implement* a shared object, processes follow an *algorithm* consisting of deterministic automata, one per process. When a process p *invokes* an operation on the object, it takes *steps*, according to its automaton. Each step consists of local computation and one shared memory event – either a read or a write to an atomic register. After each step, p updates its local state according to its state machine and optionally returns a response on the pending high-level operation. In an *operation instance* $\Phi = (x, o, p)$, process p performs the operation o on object x. An *execution fragment* is a (possibly infinite) sequence of steps by the processes, according to their respective state machines. A *configuration* represents a complete system state, that is, the local states of all processes together with the state of the shared memory. An *execution* is an execution fragment starting from the algorithm's initial configuration. If the

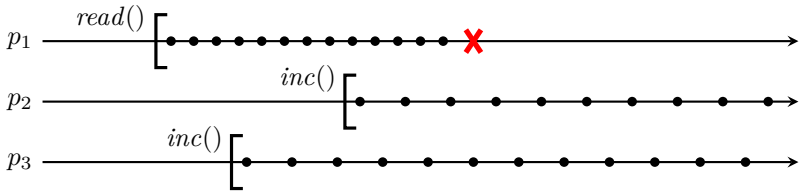

Figure 1: Execution of three operations on a shared counter. Open brackets mark invocations and dots mark steps. Process p_1 crashes after some steps, while p_2 and p_3 continue concurrently. A read conflicts with an increment. Two increments commute. Conflict-obstruction-freedom guarantees completion of the two increments, while obstruction-freedom does not.

last step of an operation instance Φ in execution α returns a response, we say that Φ *completes* in α, otherwise Φ is *pending* in α. We say that Φ takes a step in an execution α when the process p that invoked Φ takes a step and Φ has not yet returned a response in α. In an infinite execution, a process is *correct* if it takes an infinite number of steps or it has no pending operation; otherwise, it is *faulty*.

Progress conditions. An implementation is *wait-free* (WF) if, in every execution, each operation instance invoked by a correct process completes in a finite number of its own steps. An operation instance Φ is *eventually step-contention free* in an execution α if either Φ completes in α, or there exists a suffix of α in which only Φ takes steps (i.e., from some point on in α, Φ runs solo). An implementation is obstruction-free (OF) if, for every infinite execution α, any operation instance Φ invoked by a correct process completes in α whenever Φ is eventually step-contention-free in α [6].

3. Conflict-obstruction-freedom

We now introduce *conflict-obstruction-freedom* (COF), a natural generalization of obstruction-freedom that leverages operation commutativity. Intuitively, while obstruction-freedom guarantees termination in the absence of *step contention*, COF only requires the absence of step-contention with *conflicting* operations. Thus, any operation that takes sufficiently many steps completes, provided it eventually interleaves only with steps of operations that commute with it. Formally, we generalize the notion of *obstruction-freedom* from [6] by restricting step-contention to operations that do not commute.

Definition 1. *An operation instance $\Phi = (x, o, p)$ is eventually conflict-step-contention free in an execution α if either Φ completes in α, or there exists a suffix of α in which no other operation $\Phi' = (x, o', q)$ takes steps such that $o \asymp o'$.*

Definition 2. *An implementation is conflict-obstruction-free if, for every infinite execution α and every operation instance Φ invoked by a correct process such that Φ is eventually conflict-step-contention free in α, Φ completes in α.*

To grasp the new progress condition, consider the execution in Figure 1, where three processes access a shared counter. The object supports two types of operations: $read()$, which returns the current value of the counter, and $inc()$, which increments the counter by one. In the figure, process p_1 invokes $o_1 = read()$, while processes p_2 and p_3 invoke $o_2 = inc()$ and $o_3 = inc()$, respectively. Reading a counter conflicts with incrementing it ($o_1 \asymp o_2$ and $o_1 \asymp o_3$), but two increment operations — and likewise two read operations — commute with each other ($o_2 \not\asymp o_3$). After some steps of p_1, p_1 crashes while p_2 and p_3 continue concurrently. In Figure 1, no operation is eventually step-contention free. As a consequence, OF would not guarantee that the operations of the two correct processes complete. On the contrary, COF would ensure progress because o_2 and o_3 are in fact eventually conflict-step-contention free.

Lemma 1. *Let $\mathcal{I}$ be an implementation of a shared object (Q, q_0, O, R, σ). If the induced conflict relation satisfies $\asymp = O \times O$ (i.e., every pair of operations conflicts), then $\mathcal{I}$ is conflict-obstruction-free iff $\mathcal{I}$ is obstruction-free.*

Proof. An operation instance Φ is eventually conflict-step-contention free if either (1) Φ completes, or (2) there is a suffix of the execution in which no other operation conflicting with Φ takes steps. If $\asymp = O \times O$, any operation conflicts with Φ, so condition (2) holds iff no other operation takes steps in the suffix. Hence, in this case, eventual conflict-step-contention-freedom coincides with eventual step-contention-freedom. Since COF requires termination of every eventually conflict-step-contention free operation, and OF requires termination of every eventually step-contention free operation, the conditions coincide. □

Lemma 2. *Let $\mathcal{I}$ be an implementation of a shared object (Q, q_0, O, R, σ). If the induced conflict relation is empty, $\asymp = \emptyset$ (i.e., all operations commute), then $\mathcal{I}$ is conflict-obstruction-free iff $\mathcal{I}$ is wait-free.*

Proof. If $\asymp = \emptyset$, no two operations conflict. Hence, in every execution, every operation instance is eventually conflict-step-contention free. By definition of COF, every such operation invoked by a correct process must complete, which implies wait-freedom. Conversely, any wait-free implementation guarantees termination of every operation in every execution, satisfying COF. □

4. Impossibility of Conflict-obstruction-freedom

Our main result establishes that no algorithm can implement a conflict-obstruction-free universal construction in the read-write shared memory model. The proof proceeds by a reduction to the (binary) consensus problem requiring COF termination. Then, we show that this problem cannot be solved in the read-write shared memory model.

4.1. Conflict-obstruction-free Consensus

In the consensus problem [3], processes *propose* values by invoking a single-shot operation *propose(v)* and *decide* the values returned by these invocations. A consensus algorithm satisfies the following properties:

- **Validity:** If a process decides a value v, then some process proposed v;
- **Agreement:** No two processes decide differently;
- **WF termination:** Every correct process decides in finitely many steps.

Conflict-obstruction-free consensus retains Validity and Agreement, but replaces wait-free termination with the following weaker liveness condition:

- **COF termination:** If eventually only processes that proposed the same value take steps, then every correct process decides in finitely many steps.

We now model consensus as a deterministic sequential object, restricting attention to the binary case. Its state space is $Q = \{0, 1, \perp\}$ and initial state $q_0 = \perp$, where 0 and 1 represent the states where values 0 and 1 have been decided, respectively, and $\perp$ represents the initial undecided state. The set of operations is $O = \{propose(v)\}_{v \in \{0,1\}}$, each returning a value in $\{0, 1\}$. The transition function σ sets the state to the argument of *the first propose* operation and returns that value; subsequent operations return the decision value associated with the state without modifying it. Hence, any pair of *propose* operations *commute* in states 0 and 1, and two operations $propose(v)$ and $propose(v')$ commute in state $\perp$ iff $v = v'$. Thus, two proposals conflict if and only if they propose different values: $propose(0) \asymp propose(1)$. Thus, a COF consensus implementation satisfies COF termination: every correct process completes its invocation in executions where, after some point, only processes with the same input value take steps.

Conflict-obstruction-free consensus is implementable in the read–write model for two processes. The consensus algorithm in [6] satisfies COF termination in this case, as it guarantees termination when both processes have the same input. As we show below, this is no longer the case for more than two processes.

4.2. Proof of the Impossibility Result

We show that COF consensus cannot be implemented in the read-write shared memory model for three or more processes. To establish this result, we consider a minimal model instance with three processes $\{p_0, p_1, p_2\}$, where p_0 proposes 0 and both p_1 and p_2 propose 1. The proof adapts the classical FLP-style bivalency argument [3, 13], using an appropriate definition of valency that accounts for COF termination.

The next step of each process is fully determined by its local state. We denote a step $e = (p, m, a)$, where p is the process performing the step, m is the memory location accessed, and $a \in \{\mathrm{R}, \mathrm{W}\}$ indicates whether p is reading or writing register m. If e is the next step of p in some configuration C, we say that e is *enabled in* C.

An *initial configuration*, specifying the state of each process and the shared memory, is characterized by the vector assigning inputs to each process. In our consensus instance, we consider the initial input vector to be $I = (0, 1, 1)$. We write $C \xrightarrow{e} C'$ or $C \to C'$ if there exists an enabled step $e = (p, m, a)$ in C such that process p takes step e, as a result, the configuration of the system changes

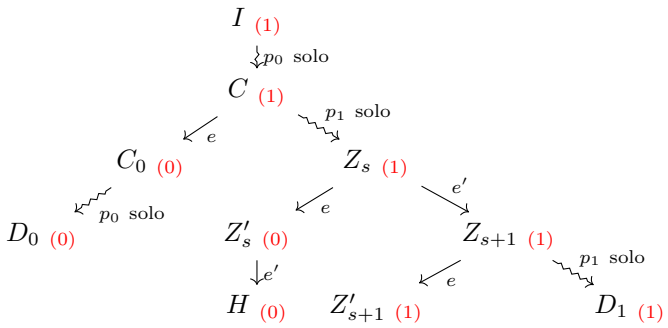

Figure 2: Valency diagram showcasing the contradiction in Lemma 4.

from C to C'. Note that the only difference between C and C' is the local state of process p, its next enabled step, and possibly the state of the shared register m. We write $C \rightsquigarrow C'$ or $C \overset{\pi}{\rightsquigarrow} C'$ if $C = C'$ or there exists a sequence of steps $\pi = e_1, \ldots, e_{k \geq 1}$ such that $C \xrightarrow{e_1} \ldots \xrightarrow{e_k} C'$ and say that C' is *reachable* from C, or C' is a *descendant* of C. For a set $P \subseteq \{p_i\}_{i \in [n]}$, we say that an execution fragment π is P*-only* if π only contains steps of processes in P.

We define the valency of a configuration according to the *decisions* processes p_1 and p_2 take in a subset of reachable configurations. A configuration C is v*-deciding* if some process has decided v in C. We say that a configuration C is v*-valent* if there is no $\{p_1, p_2\}$-only sequence π, such that $C \overset{\pi}{\rightsquigarrow} C'$ and C' is $(1-v)$-deciding. We say that a configuration is *univalent* if it is 0-valent or 1-valent. Otherwise, we say that the configuration is *bivalent*. Clearly, if a configuration C is v-valent and it has a descendant C', $C \overset{\pi}{\rightsquigarrow} C'$, where π is $\{p_1, p_2\}$-only, then C' is also v-valent.

Lemma 3. *The initial configuration I with input vector $(0, 1, 1)$ is 1-valent.*

Proof. Consider any execution π from I in which only p_1 and p_2 take steps. This execution is indistinguishable to p_1 and p_2 from an execution starting from the input vector $(1, 1, 1)$; therefore, by Validity, no process decides 0 in π. □

Lemma 4. *There exists a bivalent configuration reachable from I.*

Proof. Suppose, by contradiction, that every configuration reachable from I is univalent. Figure 2 illustrates the key configurations and critical steps considered below. Suppose an execution σ where p_0 runs solo from I. As p_1 and p_2 take no steps, p_0 decides 0 in σ by COF Termination and Validity. Let D_0 be the configuration reached in σ where p_0 decides 0. By Agreement (and linearizability), D_0 is 0-valent.

By our assumption, all configurations reachable from I are univalent (in particular, all configurations in σ, from I to D_0). By Lemma 3, I is 1-valent. Hence, there exist configurations C, C_0 and a *critical* step $e = (p_0, m, a)$ such that $I \rightsquigarrow C \xrightarrow{e} C_0$, where C is 1-valent and C_0 is 0-valent. Now consider an execution π where p_1 runs solo from C until it decides. By Termination and the fact that C is 1-valent, π ends at a configuration D_1 where p_1 decides 1.

Let $C = Z_0 \dots Z_{k\geq 1} = D_1$ be the sequence of configurations obtained by applying the steps in π. Since p_0 takes no step in π, e is enabled in each $Z_{i\in[0,k]}$. Let $Z'_i : Z_i \xrightarrow{e} Z'_i$ be the resulting configuration from applying step e from Z_i. Then $Z'_0 = C_0$ is 0-valent and Z'_k is 1-valent by Agreement. Hence, there exists $s \in [0, k-1]$ such that Z'_s is 0-valent, and Z_s, Z_{s+1} and Z'_{s+1} are 1-valent.

Let $e' = (p_1, m', a')$ be the step such that $Z_s \xrightarrow{e'} Z_{s+1}$, and H the configuration such that $Z'_s \xrightarrow{e'} H$. We observe that step $e = (p_0, m, a)$ must be a write. Otherwise, H and Z_{s+1} are indistinguishable to p_2, however, H is 0-valent and Z_{s+1} is 1-valent. Similarly, e' must also be a write; otherwise Z'_s and Z'_{s+1} are indistinguishable to p_2, yet p_2 would decide differently from Z'_s and Z'_{s+1}.

Now there are two cases to consider, both leading to a contradiction. If $m \neq m'$, then e' and e commute, leading to $H = Z'_{s+1}$, and contradicting that H is 0-valent while Z'_{s+1} is 1-valent. If instead $m = m'$, then p_2 cannot distinguish Z'_{s+1} from Z'_s, yet Z'_s is 0-valent while Z'_{s+1} is 1-valent; a contradiction. □

Lemma 5. *Any bivalent configuration C with $I \rightsquigarrow C$ has an enabled step e by a process in $\{p_1, p_2\}$ such that the resulting configuration C' with $C \xrightarrow{e} C'$ is bivalent.*

Proof. Let e_1 and e_2 be the steps by, resp., p_1 and p_2 that are enabled in C. Let C_1 and C_2 be the configurations obtained by applying, resp., e_1 and e_2 to C. By contradiction, assume that both C_1 and C_2 are univalent. Moreover, C_1 and C_2 must have different valencies, otherwise C would not be bivalent. Without loss of generality, assume that C_1 is 0-valent and C_2 is 1-valent. Let C'_1 and C'_2 be the resulting configurations of applying steps e_2 and e_1 from C_1 and C_2 respectively ($C_1 \xrightarrow{e_2} C'_1$ and $C_2 \xrightarrow{e_1} C'_2$). If both e_1 and e_2 are read steps, configurations C'_1 and C'_2 are indistinguishable to any process and have opposite valencies—a contradiction since p_1 has to decide differently in any $\{p_1\}$-only execution right after C'_1 and C'_2. The same argument applies if e_1 and e_2 access different registers.

Thus, the two steps access the same register and (at least) one of them is a write. Without loss of generality, suppose that e_1 is a write step. By termination, if we schedule $\{p_1\}$-only executions right after C_1 and C'_2, it should eventually decide. Let D_1 and D_2 be the corresponding deciding configurations. Process p_1 cannot distinguish C_1 from C'_2: if e_2 is a write step, it writes to the same location as step e_1 and occurs before it ($C \xrightarrow{e_2} C_2 \xrightarrow{e_1} C'_2$), and if e_2 is a read step, it affects only the local state of p_2. Hence, p_1 cannot distinguish between D_1 and D_2, and thus should decide the same value in both configurations—a contradiction with the fact that C_1 (and, thus, D_1) is 0-valent and C_2 (and, thus, D_2) is 1-valent. □

Theorem 6. *No algorithm implements conflict-obstruction-free consensus in the read-write shared memory model of three or more processes.*

Proof. By contradiction, suppose that there exists an algorithm $\mathcal{A}$ implementing COF consensus for three processes $\{p_0, p_1, p_2\}$. Let us consider the initial configuration I with the input vector $(0, 1, 1)$, i.e., p_0 proposes 0 and p_1 and p_2

propose 1. By Lemma 4, $\mathcal{A}$ has a bivalent configuration C reachable from I. Lemma 5 shows that from any bivalent configuration we can take a step by a process in $\{p_1, p_2\}$ to reach another bivalent configuration. By repeating this argument, we construct an infinite execution starting from I in which, from some point on, only $\{p_1, p_2\}$ take steps. Hence, at least one process $p_i \in \{p_1, p_2\}$ takes infinitely many steps and is therefore correct. As the execution goes through bivalent configurations only, no process can decide in it. However, because $\{p_1, p_2\}$ have the same input, p_i's operation is eventually conflict-step-contention free and is therefore required to complete —a contradiction. Hence no such algorithm exists. □

A COF universal construction is an algorithm that takes as input a sequential object specification and produces a linearizable shared-memory implementation satisfying COF termination — in particular it implements COF consensus.

Corollary 7. *No universal construction algorithm is conflict-obstruction-free in the read-write shared memory model of three or more processes.*

Acknowledgments

This work was supported by the CHIST-ERA European initiative (grant CHIST-ERA-22-SPiDDS-05, REDONDA project), and by Agence Nationale de la Recherche (grant ANR-23-CHR4-0009).

References

[1] M. Herlihy, J. M. Wing, Linearizability: A correctness condition for concurrent objects, ACM Trans. Program. Lang. Syst. 12 (3) (1990) 463–492.

[2] M. Herlihy, Wait-free synchronization, ACM Trans. Program. Lang. Syst. 13 (1) (1991) 124–149.

[3] M. J. Fischer, N. A. Lynch, M. S. Paterson, Impossibility of distributed consensus with one faulty process, JACM 32 (2) (1985) 374–382.

[4] M. Herlihy, V. Luchangco, M. Moir, Obstruction-free synchronization: Double-ended queues as an example, in: Proc. ICDCS, 2003, pp. 522–529.

[5] F. E. Fich, V. Luchangco, M. Moir, N. Shavit, Obstruction-free algorithms can be practically wait-free, in: DISC, 2005, pp. 78–92.

[6] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, The complexity of obstruction-free implementations, JACM 56 (4) (2009) 24:1–24:33.

[7] W. Weihl, Commutativity-based concurrency control for abstract data types, IEEE Trans. on Computers 37 (12) (1988) 1488–1505.

[8] L. Lamport, Generalized consensus and Paxos, Microsoft Research technical report MSR-TR-2005-33 (2005).

[9] M. Herlihy, E. Koskinen, Transactional boosting: a methodology for highly-concurrent transactional objects, in: Proc. PPoPP, 2008, pp. 207–216.

[10] A. Fekete, N. Lynch, M. Merritt, W. Weihl, Commutativity-based locking for nested transactions, J. Comput. Syst. Sci. 41 (1) (1990) 65–156.

[11] J. Aspnes, M. Herlihy, Wait-free data structures in the asynchronous PRAM model, in: F. T. Leighton (Ed.), Proc. SPAA, 1990, pp. 340–349.

[12] G. Toyos-Marfurt, Pending conflicts make progress impossible: a lean formalization (2026). doi:10.5281/zenodo.23037390.

[13] M. Loui, H. Abu-Amara, Memory requirements for agreement among unreliable asynchronous processes, Adv. in Comput. Research 4 (1987) 163–183.