COMPOSITIONAL REASONING IS NOT POSSIBLE IN DETERMINING THE SOLVABILITY OF CONSENSUS

Prasad Jayanti

Technical Report PCS-TR96-277

1/96

Compositional reasoning is not possible in determining the solvability of consensus*

Prasad Jayanti
Department of Computer Science
Dartmouth College
Hanover, NH 03755

prasad@cs.dartmouth.edu

Abstract

Consensus, which requires processes with different input values to eventually agree on one of these values, is a fundamental problem in fault-tolerant computing. We study this problem in the context of asynchronous shared-memory systems. In our model, shared-memory consists of a sequence of cells and supports a specific set of operations. Prior research on consensus focussed on its solvability in shared-memories supporting *specific* operations. In this paper, we investigate the following general question:

> Let OP_1 and OP_2 be any two sets of operations such that each set includes *read* and *write* operations. Suppose there is no consensus protocol for N processes in a shared-memory that supports only operations in OP_1 and in a shared-memory that supports only operations in OP_2. Does it follow that there is no consensus protocol for N processes in a shared-memory that supports only operations in $OP_1 \cup OP_2$?

This question is in the same spirit as the robustness question [7], but there are significant differences, both conceptually and in the models of shared-memory for which the two questions are studied. For deterministic types, the robustness question has been shown to have a positive answer [1, 10]. In contrast, we prove that the answer to the question posed above is negative even if operations are deterministic.

1 Introduction

1.1 Background

In an asynchronous system, processes progress at independent and arbitrarily varying speeds. Consequently, the view that a process holds of the global state of the computation does not necessarily coincide with either the reality or with the view of another process. Thus, it often becomes necessary for processes to reconcile their differences and arrive at a mutually acceptable common view. The desirable requirements of such reconciliation are captured by the *consensus problem*, which may be stated as follows. Each process is initially given a binary

*This work was supported by the NSF grant CCR-9410421 and the Dartmouth College Startup grant.

input and is required to eventually decide a value such that (i) no two processes decide different values, and (ii) the decision value is the input of some process.

In this paper we study the consensus problem in systems where asynchronous processes share a memory. The shared-memory consists of an infinite number of cells, each capable of holding an unbounded integer, and it supports a specific set of operations. Processes communicate by applying operations to shared-memory. Each operation has a well-defined specification that determines how the memory state is transformed and what response is returned to the invoking process. An operation may affect only a single memory cell (*e.g.*, *write*, *fetch&increment*, *compare&swap*) or multiple memory cells (*e.g.*, *move*, *n-register assignment*). We assume that operations are linearizable: each application of an operation appears to take effect at some instant between its invocation and response [6].

The importance of the consensus problem became explicit when Herlihy discovered the fundamental role of consensus in realizing implementations of arbitrary data-structures: Given (i) a wait-free protocol that (repeatedly) solves the consensus problem among N processes, and (ii) an array of registers, it is possible to have a wait-free implementation, shared by N processes, of any data-structure that has a sequential implementation [5]. (An implementation is *wait-free* if every process can complete every operation on the implemented object in a finite number of its own steps, regardless of the speeds of the remaining processes [8].) Thus, a consensus protocol can be regarded as a "universal" primitive.

Whether a consensus protocol is possible in a given system clearly depends on what operations are supported by the shared-memory of that system. Consequently, there has been an active study of the capabilities of shared-memories supporting different sets of common operations. Dolev, Dwork, and Stockmeyer [3], Loui and Abu-Amara [9], and Chor, Israeli, and Li [2] proved that it is not possible to solve consensus, even for two processes, if shared-memory supports only *read* and *write* operations. (These and most other impossibility results relating to consensus are proved using the bivalency technique introduced by Fisher, Lynch, and Paterson [4].) Loui and Abu-Amara proved that if shared-memory supports *test&set* operation, in addition to *read* and *write*, it is possible to solve consensus for two processes, but it is still impossible to solve for three processes [9]. Finally, Herlihy considered a host of common operations — *fetch&add*, *swap*, *move*, *n-register assignment*, *compare&swap etc.* — and analyzed, for each operation *op*, the maximum number of processes for which we can solve consensus if shared memory supports *op*, *read* and *write* [5].

1.2 Problem and the result

As described above, the feasibility of solving consensus in shared-memories supporting *specific operations* has been well-studied. In this paper we ask the following general question. Let M_1, M_2, and M_3 be multiprocessors where M_1's shared-memory supports some set of operations OP_1, M_2's shared-memory supports some set of operations OP_2, and M_3's shared-memory supports operations $OP_1 \cup OP_2$. Suppose that we know that there is no consensus protocol for N processes in systems M_1 and M_2. Based *only* on this knowledge, can we deduce that there is no consensus protocol for N processes in system M_3? The answer is no, as the following simple argument shows. Let OP_1 consist of a single operation called *sticky-write*(v, loc), with the following specification: the operation changes the value of cell at location *loc* to v if and only if its previous value is $\perp$, and returns *ack* as the response. Since a process gets no useful

information through the response of a *sticky-write* operation, it is impossible to solve consensus, even for two processes, if shared-memory supports only OP_1. Let OP_2 consist of the single operation, *read(loc)*, which returns the value of the memory cell at location *loc*. It is obvious that it is impossible to solve consensus, even for two processes, if shared-memory supports only OP_2. However, if shared-memory supports $OP_1 \cup OP_2 = \{\textit{sticky-write}(v, loc), \textit{read}(loc)\}$, we can solve consensus for N processes, for any N, as follows: a cell is initialized to $\perp$; each process writes its input to that cell using *sticky-write* and then reads the value of that cell and decides on that value.

The question becomes interesting if we require that each of OP_1 and OP_2 includes *read* and *write* operations. Such a requirement is reasonable from a practical point of view since *read* and *write* operations are very basic and are supported in all real multiprocessors. Thus, we consider the following property of pairs of sets of operations:

> PROP$_N(OP_1, OP_2)$: If each of OP_1 and OP_2 is a set of operations that includes *read* and *write* operations, and there is no consensus protocol for N processes in a shared-memory that supports only operations in OP_1 and in a shared-memory that supports only operations in OP_2, then there is no consensus protocol for N processes in a shared-memory that supports only operations in $OP_1 \cup OP_2$.

To the best of our knowledge, for all sets of operations OP_1, OP_2 studied in the literature, PROP$_N(OP_1, OP_2)$ holds. The natural question is:

> QUESTION: Does PROP$_N(OP_1, OP_2)$ hold for all OP_1, OP_2?

The motivation for studying this question is clear: if the answer is yes, compositional reasoning becomes possible. For instance, based solely on our knowledge that that there is no consensus protocol for four processes if shared-memory supports only {*read*, *write*, *2-register assignment*} or only {*read*, *write*, *fetch&increment*}, we will be able to conclude the impossibility of a consensus protocol even if shared-memory supports {*read*, *write*, *2-register assignment*, *fetch&increment*}. On the other hand, if the answer is negative, it follows that such compositional reasoning is impossible in general: each time a new operation is added to a set of operations, the power of the expanded set needs to be evaluated from scratch.

We prove that the answer to the above question is a strong no. Specifically, we exhibit two (deterministic) operations, *reduce* and *conditional-multiply*, with the following properties:

1. If shared-memory supports only operations in {*read*, *write*, *reduce*} or it supports only operations in {*read*, *write*, *conditional-multiply*}, then it is not possible to solve consensus, even for two processes.
2. If shared-memory supports operations in {*read*, *write*, *reduce*, *conditional-multiply*}, then it is possible to solve consensus for N processes, for any N.

Thus, for $OP_1 =$ {*read*, *write*, *reduce*} and $OP_2 =$ {*read*, *write*, *conditional-multiply*}, PROP$_N(OP_1, OP_2)$ is false for all $N \geq 2$.

1.3 Differences with the robustness question

The question raised here is in the same spirit as the following question, raised in the context of investigating robust wait-free hierarchies [7].

> ROBUSTNESS QUESTION: Suppose that there is no consensus protocol for N processes in the following two situations: (i) when registers and objects of type T_1 are all shared-objects available for process communication, and (ii) when registers and objects of type T_2 are all shared-objects available for process communication. Does it follow that there is no consensus protocol for N processes even if registers, objects of type T_1, *and* objects of type T_2 are available for process communication?

The differences between this question and the one studied in this paper are the following:

1. The model of shared-memory considered in the study of the robustness question is different from the model considered in this paper. In studying the robustness question, shared-memory is viewed as a collection of objects. For each object, there is a well-defined set of operations which are the *only* means of accessing that object. In this paper, we view shared-memory as a sequence of cells on which some set of operations, including *read* and *write*, may be applied. This view of shared-memory accurately models real multiprocessors, but it cannot model objects such as queues, supporting only *enq* and *deq* operations (because the data hiding inherent in such abstract data objects is not possible when shared-memory is simply a linear sequence of cells all of which can be read and written by all processes).
2. Aside from the model, there is an important conceptual difference. Suppose that neither a shared-memory that supports OP_1 nor a shared-memory that supports OP_2 is good enough for solving N-process consensus (see the top two pictures in Figure 1). The present question asks if it is necessarily the case that a shared-memory supporting $OP_1 \cup OP_2$ is also not good enough for solving N-process consensus (see bottom left in Figure 1). In contrast, the robustness question asks if it is necessarily the case that two banks of shared-memory, one supporting *only* OP_1 and the other supporting *only* OP_2, are together not good enough for solving N-process consensus (see bottom right picture in Figure 1).

If objects are deterministic, Borowsky, Gafni, and Afek [1] and Peterson, Bazzi, and Neiger [10] showed that the answer to the robustness question is positive. In contrast, we prove that the answer to the question posed in this paper is negative even if operations are deterministic.

2 Model and definitions

2.1 Shared-Memory

A *shared-memory* consists of an infinite number of *cells*, numbered $0, 1, 2, \ldots$, each capable of holding any natural number as its value. A state of memory is a tuple $[v_0, v_1, v_2, \ldots]$, where each v_i is a natural number that denotes the value of the cell numbered i. We let

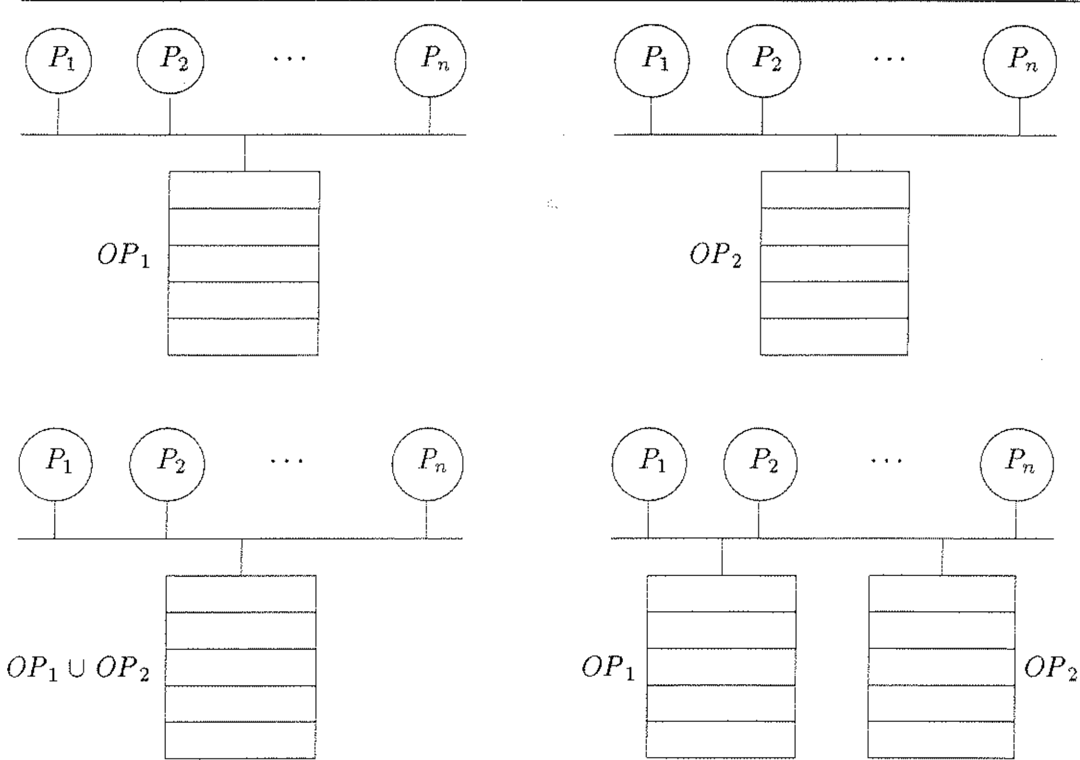

Figure 1: Illustration to contrast present question with robustness question

$Q = N^N$ denote the set of all possible states of memory (N is the set of natural numbers). Each memory is characterized by the set of operations that it supports. Each operation is a tuple (op-$name, n, ARG_1, \ldots, ARG_n, RES, \delta$), where *op-name* is a unique name by which the operation is referenced, n is the number of arguments that the operation takes, ARG_i is the set of possible values for the ith argument, RES is the set of possible responses, and $\delta : Q \times ARG_1 \times \cdots ARG_n \rightarrow Q \times RES$ is the *sequential specification* of the operation. Intuitively, if $\delta(\sigma, arg_1, \ldots, arg_n) = (\sigma', res)$, it means the following: Applying the operation with arguments $arg_1, \ldots, arg_n$ to a memory in state σ changes the memory's state to σ' and returns the response res. δ is required to satisfy two properties:

Finite effect: If $\delta(\sigma, arg_1, \ldots, arg_n) = (\sigma', res)$, the number of memory cells that have different values in σ and σ' is finite.

Computability: There is a two tape Turing machine M such that, given a state σ of memory on the first tape (*i.e.*, first tape contains an infinite string $v_0\#v_1\#v_2\# \ldots$, where v_i is the value of ith cell, and a string $arg_1\#arg_2\# \ldots \#arg_n$ encoding n arguments on the second tape, M eventually halts with σ' on the first tape and res on the second tape, where $(\sigma', res) = \delta(\sigma, arg_1, \ldots, arg_n)$.

We assume that every shared-memory supports *read* and *write* in addition to any other operations. *read*(loc) returns the value of the location loc, and *write*(v, loc) writes the value v at location loc, returning ack.

2.2 Concurrent system

We define a concurrent system informally. A formal definition using I/O automata was given in [5].

A *concurrent system* is specified by a finite set of processes $\{P_1, P_2, \ldots, P_N\}$ and a shared-memory M, where

- processes have distinct names. (All names are known to all processes).
- each cell of M is assigned an initial vale.
- each process is specified by a deterministic program. Some internal variables are distinguished as *input variables* and some are distinguished as write-once *output variables*. (Output variables are initialized to $\perp$.) Each instruction of the program specifies which, among the operations supported by M, should be applied and how the response should alter the values of the internal variables of the program.

We denote such a concurrent system as $(P_1, \ldots, P_N; M)$.

A *configuration* of a concurrent system is a tuple consisting of the states of processes and the shared-memory. Notice that the initial configuration is uniquely fixed by an assignment of values to input variables of processes. An *execution* of a concurrent system is a tuple $(\mathcal{A}, \Sigma)$, where $\mathcal{A}$ is an assignment of values to input variables and $\Sigma = C_0, C_1, C_2, \ldots$ is a sequence of configurations such that C_0 is the initial configuration corresponding to the assignment $\mathcal{A}$, and C_{i+1} is the configuration that results when some (single) process P executes a single instruction of its program in configuration C_i. We refer to the change of configuration from C_i to C_{i+1} as a *step* and associate this step with process P. The *execution is infinite* if Σ is infinite.

2.3 Consensus protocol

A *consensus protocol for processes* $P_1, \ldots, P_N$ is a concurrent system $(P_1, \ldots, P_N; M)$, where each P_i has a binary input variable $proposal_i$ and an output variable $decision_i$ such that every infinite execution $(\mathcal{A}, \Sigma)$ has the following properties:

Wait-freedom: If a process P_i has infinitely many steps in $\Sigma = C_0, C_1, C_2, \ldots$, then P_i *decides* in Σ: that is, there is a configuration C_k such that $decision_i$ has a non-$\perp$ value in C_k. (We refer to this non-$\perp$ value as P_i's *decision value* in Σ, and refer to the value assigned to $proposal_i$ by $\mathcal{A}$ as P_i's *proposal* in Σ.)

Agreement: If P_i and P_j decide in Σ, then their decision values are the same.

Validity: If P_i decides in Σ, then its decision value is the proposal of some process.

Definition 2.1 We say *there is a consensus protocol for N processes in shared-memory M* if there is a consensus protocol $(P_1, \ldots, P_N; M)$.

We let CONSENSUS$(P_i, v_i, \mathcal{P})$ denote process P_i's program in consensus protocol $\mathcal{P}$ when P_i's proposal is v_i.

3 Specification of operations `reduce` and `conditional-multiply`

In this section, we define two operations — *reduce* and *conditional-multiply*. Their sequential specifications are given in Figures 2 and 3. The operation *reduce*(loc) acts on the cell at location loc and changes the cell's value to its smallest prime factor. (In the figure, $M[loc]$ denotes the value of the memory cell at location loc.) Thus, for instance, if a cell has the value 6, *reduce* changes its value to 2. Similarly, if a cell has 35, *reduce*() changes it to 5. The operation *conditional-multiply*(u, loc) acts on the cell at location loc. It does not affect the cell if it holds a prime value (we consider 2 as the smallest prime). Otherwise, it multiplies the current value of the cell with u and writes the result in the cell. Both *reduce* and *conditional-multiply* return *ack* as the response.

```
reduce(loc)
    v := M[loc]
    if (v ≠ 0) ∧ (v ≠ 1)
        Let p be the smallest prime factor of v
        M[loc] := p
    endif
    return ack
```

Figure 2: Sequential specification of *reduce*

```
conditional-multiply(u, loc)
    v := M[loc]
    if v is not a prime
        M[loc] := u · v
    endif
    return ack
```

Figure 3: Sequential specification of *conditional-multiply*

---

4 The power of `reduce` and `conditional-multiply`

In this section, we show that a consensus protocol for N processes is possible in a shared-memory that supports all of *read*, *write*, *reduce*, and *conditional-multiply*, but is not possible in a shared-memory that supports *read*, *write*, and only one of *reduce* and *conditional-multiply*.

4.1 Consensus protocol

The consensus protocol is presented in Figure 4. It uses $N+1$ shared-memory cells: the array *input*$[1..N]$ for recording the proposals of processes, and the cell *synch-obj*, for determining the identity of the winning process. Each process P_i writes its proposal v_i in *input*$[i]$ and then participates in a race that determines which of $P_1, P_2, \ldots, P_N$ is the winner. P_i applies *conditional-multiply* in an attempt to multiply the value in *synch-obj* with q_i^2, where q_i is the ith prime (we regard 2 as the first prime). P_i then applies *reduce* to force *synch-obj* to assume a prime value. It then reads the value of *synch-obj*. If the value read is the jth prime, then P_j is regarded as the winner of the race. So P_i decides the value in *input*$[j]$.

Lemma 4.1 *For all $N \geq 1$, there is a consensus protocol for N processes in a shared-memory that supports the operations read, write, reduce, and conditional-multiply. Figure 4 presents such a protocol.*

Proof Sketch Consider any execution of the consensus protocol in which one or more processes have decided. Let P_r be the *first* process to apply *reduce* on *synch-obj* and $P_{j_1}, P_{j_2}, \ldots, P_{j_k}$ $(j_1 < j_2 < \cdots < j_k)$ be *all* the processes that applied *conditional-multiply* before P_r applied *reduce*. The key claim is that P_r's *reduce* operation causes the value of *synch-obj* to become q_{j_1}, and that the value of *synch-obj* never subsequently changes. The next paragraph proves this claim.

From the above, the first k operations on *synch-obj* are the *conditional-multiply* operations by processes $P_{j_1}, P_{j_2}, \ldots, P_{j_k}$ and the $(k+1)^{th}$ operation on *synch-obj* is the *reduce* operation by P_k. Since the initial value of *synch-obj* is 1 and since each of $P_{j_1}, P_{j_2}, \ldots, P_{j_k}$ attempts to multiply the value in *synch-obj* with a non-prime, their multiplications succeed. (This is obvious from the specification of *conditional-multiply*.) Thus, the value of *synch-obj*, just before P_r applies *reduce*, is $q_{j_1}^2 q_{j_2}^2 \cdots q_{j_k}^2$. Since $j_1 < j_2 < \cdots < j_k$, P_r's *reduce* operation causes the value of *synch-obj* to become q_{j_1}. (This follows directly from the specification of *reduce*.) Since q_{j_1} is a prime, subsequent *conditional-multiply* and *reduce* operations do not affect the value of *synch-obj*. Hence the claim.

From the claim, it follows that every process obtains q_{j_1} when it reads *synch-obj*. Thus, every process reads and returns the value in *input*$[j_1]$. Clearly, P_{j_1}'s writing of its proposal v_{j_1} in *input*$[j_1]$ precedes P_{j_1}'s application of *conditional-multiply*, which in turn precedes P_r's application of *reduce*. Since P_r's application of *reduce* precedes the reading of *synch-obj* by any process, every process obtains v_{j_1} when it reads *input*$[j_1]$. Thus, every process decides v_{j_1}. We conclude that the protocol satisfies validity and agreement. It is obvious that the protocol is wait-free. This concludes the proof of correctness of the protocol. □

4.2 Impossibility result

We now prove that there is no consensus protocol for two processes in a shared-memory that supports *read*, *write*, and only one of *conditional-multiply* and *reduce*. This impossibility result follows from a straightforward bivalency argument. Since bivalency arguments are standard, our definitions and the proof are informal. A configuration C of a consensus protocol is *v-valent* (for $v \in \{0,1\}$) if there is no execution from C in which $\overline{v}$ is decided by some process. In

```
shared-memory cells
```
$input[1..N]$, uninitialized
$synch\text{-}obj$, initialized to 1

```
internal variables of process P_i
```
$proposal_i \in \{0, 1\}$
$decision_i \in \{\perp, 0, 1\}$, initialized to $\perp$
$winner_i \in \{0, 1\}$, (uninitialized)

CONSENSUS($P_i, proposal_i, \mathcal{P}$) (for $1 \leq i \leq N$)

1. $input[i] := proposal_i$
2. conditional-multiply($P_i, q_i^2, synch\text{-}obj$)
3. reduce($P_i, synch\text{-}obj$)
4. $winner_i := synch\text{-}obj$
5. $decision_i := input[winner_i]$

Figure 4: Consensus protocol for processes $P_1, \ldots, P_N$

other words, once the protocol is in configuration C, no matter how processes are scheduled, no process decides $\overline{v}$. A configuration is *monovalent* if it is either 0-valent or 1-valent. A configuration is *bivalent* if it is not monovalent. If E is a finite execution of a consensus protocol $\mathcal{P}$ started in configuration C, $E(C)$ denotes the configuration at the end of the execution E.

Lemma 4.2 *There is no consensus protocol for two processes in a shared-memory that supports read, write, and only one of reduce and conditional-multiply.*

Proof This proof is a straightforward application of standard bivalency arguments. Suppose that there is a consensus protocol $\mathcal{P} = (P_0, P_1; M)$, where M is a shared-memory that supports *read*, *write*, and only one of *reduce* and *conditional-multiply*. Let C_0 be the initial configuration of $\mathcal{P}$ such that $proposal_0 = 0$ and $proposal_1 = 1$.

When P_0 runs by itself from C_0, the validity and the wait-freedom properties of $\mathcal{P}$ require that P_0 decides $proposal_0 = 0$. Similarly, when P_1 runs by itself from C_0, it decides $proposal_1 = 1$. Thus, C_0 is bivalent. Let E be a finite execution from C_0 such that (1) $\text{CRIT} = E(C_0)$ is bivalent, and (2) For all $i \in \{0, 1\}$, if P_i takes a step from CRIT, the resulting configuration is monovalent. (If such E does not exist, it is easy to see that there is an infinite execution E' in which no process decides. Thus, one of P_0 and P_1 takes infinitely many steps in E' without deciding, contradicting that $\mathcal{P}$ is a wait-free protocol.) For $i \in \{0, 1\}$, let X_i denote the shared-memory cell on which process P_i acts if P_i were to take a step from configuration CRIT, and let CRIT_i denote the configuration that results when P_i takes a step from CRIT. Since CRIT is bivalent and CRIT_0, CRIT_1 are both monovalent, it follows that one of CRIT_0 and

CRIT_1 is 0-valent and the other is 1-valent. Without loss of generality, let CRIT_0 be 0-valent and CRIT_1 be 1-valent.

Our convention in naming configurations is as follows: if σ is a finite sequence of 0's and 1's, CRIT_σ denotes the configuration that results when, starting from CRIT, processes are scheduled according to σ. For example, $\text{CRIT}_{0,1}$ is the configuration that results when, starting from CRIT, P_0 takes a step and then P_1 takes a step.

We claim that X_0 and X_1 must be the same shared-memory cell. For a proof, assume that the claim is false. Then, it is obvious that configurations $\text{CRIT}_{0,1}$ and $\text{CRIT}_{1,0}$ are identical. Since CRIT_0 is 0-valent, $\text{CRIT}_{0,1}$ must be 0-valent. Similarly, since CRIT_1 is 1-valent, $\text{CRIT}_{1,0}$ must be 1-valent. Since $\text{CRIT}_{0,1} = \text{CRIT}_{1,0}$, it follows that $\text{CRIT}_{0,1}$ must be both 0-valent and 1-valent, which is impossible. We conclude that X_0 and X_1 must be the same shared-memory cell. In the following, let $X_0 = X_1 = X$.

For $i \in \{0, 1\}$, let op_i denote the operation that process P_i will perform on cell X if P_i is scheduled from configuration CRIT. We claim that neither op_0 nor op_1 is a *read*. For a proof, suppose that the claim is false and op_i is *read* for some $i \in \{0, 1\}$. Clearly, the state of $P_{\bar{i}}$ and the state of the shared-memory are the same in configurations $\text{CRIT}_{i,\bar{i}}$ and $\text{CRIT}_{\bar{i}}$. Thus, the value that $P_{\bar{i}}$ decides when it runs by itself from $\text{CRIT}_{i,\bar{i}}$ is the same as the value that it decides when it runs by itself from $\text{CRIT}_{\bar{i}}$. This contradicts the earlier conclusion that $\text{CRIT}_{i,\bar{i}}$ is i-valent and $\text{CRIT}_{\bar{i}}$ is $\bar{i}$-valent.

Next we claim that neither op_0 nor op_1 is a *write* operation. For a proof, suppose that op_i is *write* v for some $i \in \{0, 1\}$ and $v \in N$. Clearly, the state of P_i and the state of the shared-memory are the same in configurations $\text{CRIT}_{\bar{i},i}$ and CRIT_i. Thus, the value that P_i decides when it runs by itself from $\text{CRIT}_{\bar{i},i}$ is the same as the value that it decides when it runs by itself from CRIT_i. This contradicts the earlier conclusion that $\text{CRIT}_{\bar{i},i}$ is $\bar{i}$-valent and CRIT_i is i-valent.

Thus, each of op_0 and op_1 is a *conditional-multiply* operation or a *reduce* operation. Since the lemma states that the shared-memory supports only one of *conditional-multiply* and *reduce*, but not both, it follows that either op_0 and op_1 are both *reduce* operations or op_0 and op_1 are both *conditional-multiply* operations. We will now consider each of these possibilities.

We claim that op_0 and op_1 cannot both be *reduce* operations. For a proof, suppose that they are. Regardless of the value of X in configuration CRIT, from the specification of *reduce* it is obvious that the states of P_0, P_1 and X are identical in $\text{CRIT}_{0,1}$ and $\text{CRIT}_{1,0}$. Thus, $\text{CRIT}_{0,1}$ and $\text{CRIT}_{1,0}$ are identical. This contradicts that $\text{CRIT}_{0,1}$ is 0-valent and $\text{CRIT}_{1,0}$ is 1-valent.

Finally, we claim that op_0 and op_1 cannot both be *conditional-multiply* operations. For a proof, suppose that the claim is false, and op_0 is *conditional-multiply*(P_0, v_0, X) and op_1 is *conditional-multiply*(P_1, v_1, X). Let the value of X in configuration CRIT be x. There are two cases to consider: x is a prime or it is not. If x is a prime, then the value of X is not affected when P_0 or P_1 applies *conditional-multiply* on X. Thus, it is easy to verify that configurations $\text{CRIT}_{0,1}$ and $\text{CRIT}_{1,0}$ are identical. This, of course, contradicts that $\text{CRIT}_{0,1}$ is 0-valent and $\text{CRIT}_{1,0}$ is 1-valent. If x is not a prime, by the specification of *conditional-multiply*, the value of X is xv_0v_1 in both $\text{CRIT}_{0,1}$ and $\text{CRIT}_{1,0}$. It is again easy to verify that configurations $\text{CRIT}_{0,1}$ and $\text{CRIT}_{1,0}$ are identical, which is a contradiction.

We conclude that the protocol $\mathcal{P}$ cannot exist. Hence the lemma. □

5 The main theorem

Theorem 5.1 $\text{PROP}_N(OP_1, OP_2)$ *does not hold for arbitrary sets of operations* OP_1 *and* OP_2.

Proof Let $OP_1 = \{read, write, reduce\}$ and $OP_2 = \{read, write, conditional\text{-}multiply\}$. By Lemmas 4.1 and 4.2, for all $N \geq 2$, $\text{PROP}_N(OP_1, OP_2)$ is false. □

References

[1] E. Borowsky, E. Gafni, and Y. Afek. Consensus power makes (some) sense. In *Proceedings of the 13th Annual Symposium on Principles of Distributed Computing*, pages 363–372, August 1994.

[2] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 86–97, August 1987.

[3] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[4] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, 1985.

[5] M.P. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, 1991.

[6] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

[7] P. Jayanti. On the robustness of herlihy's hierarchy. Technical Report TR 93-1332, Cornell University, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, March 1993.

[8] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

[9] M.C. Loui and H.H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in computing research*, 4:163–183, 1987.

[10] G. Peterson, R. Bazzi, and G. Neiger. A gap theorem for consensus types. In *Proceedings of the 13th Annual Symposium on Principles of Distributed Computing*, pages 344–353, August 1994.