

ON THE SPACE COMPLEXITY OF COLOURLESS TASKS

by

Leqi Zhu

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

© Copyright 2019 by Leqi Zhu

Abstract

On the Space Complexity of Colourless Tasks

Leqi Zhu

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2019

In this thesis, we prove lower bounds on the number of registers needed to solve colourless tasks in asynchronous shared memory systems. Many fundamental synchronization tasks, such as consensus, k -set agreement, and ϵ -approximate agreement, are colourless.

We show that it is possible to transform any nondeterministic solo-terminating algorithm (including any randomized wait-free algorithm) into an obstruction-free algorithm that uses the same number of registers. This result extends to algorithms using any finite number of deterministic objects that support read operations. Hence, we can focus on proving lower bounds for obstruction-free algorithms.

We prove a tight lower bound on the number of registers needed to solve obstruction-free consensus. We also prove the first non-constant lower bounds on the number of registers needed to solve obstruction-free k -set agreement and obstruction-free ϵ -approximate agreement. The bound for k -set agreement is asymptotically tight when k is a constant and the bound for ϵ -approximate agreement is asymptotically tight when ϵ is sufficiently small. To prove these bounds, we introduce a new technique, *revisionist simulations*. This technique allows us to prove a general theorem that yields lower bounds on the number of registers needed to solve any colourless task in an obstruction-free manner.

Finally, we define the class of *extension-based proofs* and show that no extension-based proof can establish the impossibility of deterministically solving k -set agreement among $n > k > 1$ processes in a wait-free manner using registers.

In loving memory of my grandparents, 袁诚 and 王蕴华.

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Faith Ellen. She has been extremely patient, helpful, and understanding with me throughout my graduate studies (especially in the writing of this thesis). On more than one occasion, I've half-jokingly remarked that she is like a second mother to me. This thesis and our papers together have undoubtedly benefited enormously from her selflessly spending hours and hours with me, carefully working on the exposition and proofs. Most importantly, she has imparted to me many lessons about life and research, which I will faithfully try to live up to, though I've always been a poor pupil... 😊

I must also thank my friends and colleagues at the theory group: Sepehr, Jai, Suraj, Rob, Eric, Alex, Ksenija, Jeremy, Chris, Rati, Kevan, Mike, Lily, Calum (George), Rahim, Ian, James, Tyrone, Ammar, and Ximo. I am grateful to have met each and every one of you and I will fondly remember the many interesting conversations we shared. For other members of the theory group, it's a shame that our paths did not cross as often! Special thanks to Rati, for all the wonderful excursions (all done for the sake of research, of-course) and fun collaborations.

I don't think I would have even considered walking this path, without the guidance of my parents, 朱福民 and 袁晓芬. I am thankful for their love and support as well as their commitment to our biweekly family dinners together on the weekends, rain or shine. It was a comforting tradition.

I also thank the members of my committee for taking the time to read this thesis and their great comments: Benjamin Rossman, Vassos Hadzilacos, Sergio Rajsbaum, and Sam Toueg. Support is also gratefully acknowledged from NSERC.

Last, but not least, I would like to thank a certain mysterious stranger, who gave me some very nice gifts. Including, some might say, the greatest gift of all 😊.

Contents

1	Introduction	1
2	Preliminaries	6
2.1	Colourless Tasks	6
2.2	Asynchronous Shared Memory	7
2.2.1	Objects	7
2.2.2	Processes	8
2.2.3	The Scheduler	9
2.3	Protocols	9
2.4	Implementations	10
2.5	General Progress Conditions	10
2.6	Snapshots of Readable Objects	11
3	Nondeterministic Solo-termination vs Obstruction-freedom	13
3.1	Related Work	13
3.2	The Transformation	14
3.2.1	Sketch	14
3.2.2	Formal Description	15
3.3	Proof of Correctness/Linearizability	16
3.3.1	B is correct/linearizable	16
3.3.2	B is obstruction-free	16
4	Consensus	18
4.1	Related Work	18
4.2	Buffer Lower Bound	20
4.3	Buffer Upper Bound	23
4.4	Test-and-Set Objects	27
4.5	Max-Registers	28
4.6	Swap Objects	31
5	Augmented Snapshot Objects	35
5.1	Related Work	35
5.2	Implementing block-update for a Single Process	36
5.3	Simple Augmented Snapshot Object	38

5.4	Augmented Snapshot Object	40
5.5	Proof of Correctness and Termination	43
6	Revisionist Simulations	48
6.1	Related Work	48
6.2	Overview	49
6.3	Direct Simulators and a Simple Case of the Simulation	50
6.4	Covering Simulators	53
6.5	The Intermediate Execution of a Real Execution	57
6.6	Correctness of the Simulation	58
6.7	Wait-freedom of the simulation	62
6.8	Applications of the Simulation	64
7	Why Extension-based Proofs Fail	66
7.1	Related Work	66
7.2	Preliminaries	67
7.2.1	NIIS Model	67
7.2.2	Topological Representation of a Protocol	69
7.2.3	Non-uniform Chromatic Subdivision	70
7.2.4	Distances between Subcomplexes	71
7.3	Restricted Extension-Based Proofs	73
7.4	Why Restricted Extension-based Proofs Fail	75
7.5	Extension-Based Proofs	79
8	Conclusions and Future Work	83

Chapter 1

Introduction

An *asynchronous shared memory system* consists of a finite number, $n \geq 2$, of processes, running at possibly varying speeds, that can only communicate with each other by taking steps to access shared memory locations. Each memory location supports a set of operations that may be performed on it and each step by a process involves the process performing one such operation. Many modern multiprocessor computers are examples of asynchronous shared memory systems.

A well-studied type of distributed problem is a *task*, where each process has a private input value and each process must produce a single output value such that, collectively, the output values satisfy the specifications of the task. A task is *colourless* if the input of any process may be the input of any other process, the output of any process may be the output of any other process, and the specifications of valid outputs only depends on the set of inputs of the processes.

Many fundamental problems in distributed computing are colourless tasks. One example is the *consensus* problem, in which each process has a private input value and the processes must agree on a single input value to output. There are numerous applications of consensus, e.g. in synchronization, implementing shared data structures, and solving other tasks. Other examples of colourless tasks include the *k-set agreement* problem, for integers $1 \leq k < n$, which generalizes consensus so that the set of output values has size at most k , and the *ϵ -approximate agreement* problem, which requires any two output values are at most ϵ apart. All three problems have been the subject of much research.

The goal of this thesis is to study, for various colourless tasks, the *minimum* number of memory locations needed to solve the task in an asynchronous shared memory system. We call this the *space complexity* of the task (in that system). The space complexity depends on the operations supported at the memory locations in the system.

In this thesis, we primarily consider asynchronous shared memory systems where each memory location is a *register*, i.e. it supports only two operations: **read**, which allows the contents of the memory location to be read, and **write(v)**, which writes value v to the memory location, for arbitrary values v . Our decision to focus on these systems is motivated by two considerations. First, modern architectures typically provide a fixed set of operations, including **read** and **write**, that may be performed on all memory locations. Second, for many of the tasks we consider, there were large gaps in the upper and lower bounds on their space complexity, even in this simple setting.

A solution to a task is *wait-free* if each process outputs a value after taking sufficiently many steps, regardless of what other processes do (even if they stop taking steps). Many tasks have no deterministic,

wait-free solutions in asynchronous systems containing only registers. In particular, one of the most well-known results in the theory of distributed computing is that there is no deterministic wait-free solution to consensus in this setting [31, 46].

There are many approaches to circumventing this impossibility result. One approach is to relax the circumstances in which a process must eventually output a value. For instance, a process may only be required to output a value after taking sufficiently many *consecutive* steps (without any other process interfering). This is called *obstruction-freedom*. Another way is to strengthen the system so that each process has access to a private source of randomness. In this case, each process is required to output a value after taking finitely many steps, with probability 1, regardless of what other processes do. Solutions satisfying this property are called *randomized wait-free*. The formal definition are presented in Chapter 2.

The main contributions of this thesis are new space lower bounds for solving consensus, set agreement, and approximate agreement in a randomized wait-free or obstruction-free manner in asynchronous systems containing only registers. Proving the bounds required development of new techniques, which culminated in a general theorem that may be used to prove space lower bounds for any colourless task in such systems. We also present some evidence that new techniques may be needed in order to prove general, non-constant space lower bounds for set agreement. In the rest of this chapter, we summarize the results of this thesis.

Nondeterministic Solo-termination vs Obstruction-freedom. A typical way to obtain space lower bounds for both obstruction-free and randomized wait-free solutions is to obtain space lower bounds for solutions that satisfy *nondeterministic solo termination*, which is weaker than both conditions. Informally, a solution is nondeterministic solo-terminating if, from every point, a process has a sequence of steps in which it will output a value, if it takes these steps consecutively. This approach was pioneered by Ellen, Herlihy, and Shavit [28].

A *readable* object is a memory location that supports a read operation. In Chapter 3 of this thesis, we prove that, if there is a nondeterministic solo terminating solution for a task in a system that uses only deterministic readable objects, then there is an obstruction-free solution for the task that uses the same objects. This result appears in joint work with Ellen and Gelashvili [27]. A register is an example of a readable object.

The theorem implies that, for asynchronous systems containing only registers, any space lower bound for obstruction-free solutions is also a space lower bound for nondeterministic solo terminating solutions and, hence, for randomized wait-free solutions. Consequently, researchers can focus on proving space lower bounds for obstruction-free solutions, which are simpler to analyze than randomized wait-free solutions.

Consensus. There are numerous randomized wait-free and obstruction-free solutions to consensus [7, 19, 55], which all use at least n registers. Ellen, Herlihy, and Shavit [28] proved that any nondeterministic solo terminating solution to consensus must use $\Omega(\sqrt{n})$ registers. They also introduced the approach of proving space lower bounds for *anonymous* solutions, in which each process is identical (i.e. have no identifiers) and run the same code. Much later, Gelashvili [32] proved that any anonymous nondeterministic solo terminating solution to consensus must use $\Omega(n)$ registers.

In Chapter 4 of this thesis, we give a simple proof that any obstruction-free solution for consensus

must use at least $n - 1$ registers. The proof carefully combines a covering argument [21] with a valency argument [31]. In particular, we use a stronger notion of bivalency, called *solo bivalency*, and show that solo bivalency can be maintained while performing a covering argument. We also present an anonymous obstruction-free solution that uses n registers. These results appear in [56]. Other anonymous solutions exist, but are either more complex to describe or use more registers. In joint work with Ellen, Gelashvili, and Shavit [26], we show how to generalize the techniques in [56] to prove space bounds in systems containing more powerful objects, such as buffer objects, test-and-set objects, max-registers, and swap objects.

A *b-buffer* object, for integers $b \geq 1$, is a memory location that supports two operations, `write(v)` and `read`. The `write(v)` operation writes value v to the memory location while the `read` operation returns a vector containing the last b values written to the memory location, from most recent to least recent. (The vector contains some initial default values, if fewer than b values were written.) In this thesis, we prove that, for asynchronous systems containing only b -buffer objects, $\lceil \frac{n-1}{b} \rceil$ such objects are needed to solve obstruction-free consensus. We also show a matching upper bound whenever $n - 1$ is not divisible by b .

A *test-and-set* object is a memory location, with initial value 0, that supports two operations, `test-and-set` and `read`. The `test-and-set` operation writes 1 to the memory location and returns its previous value, while the `read` operation returns the current value in the memory location. In this thesis, we prove that, for asynchronous systems containing only test-and-set objects, an unbounded number of such objects are necessary to solve obstruction-free consensus among $n \geq 3$ processes.

A *max-register* is a memory location, with initial value 0, that supports two operations, `write-max(v)`, for non-negative integers v , and `read`. The `write-max(v)` operation writes value v to the memory location if and only if v is larger than the current value in the memory location, while the `read` operation returns the current value in the memory location. In this thesis, we prove that, for asynchronous systems containing only max-registers, 2 max registers are necessary and sufficient to solve obstruction-free consensus among $n \geq 2$ processes.

A *swap* object is a memory location that supports two operations, `swap(v)` and `read`. The `swap(v)` operation writes value v to the memory location and returns the previous value in the memory location, while the `read` operation returns the current value. In this thesis, we prove that, in asynchronous systems containing only swap objects, $n - 1$ such objects are sufficient to solve anonymous obstruction-free consensus. This is a generalization of our anonymous solution for registers.

In [26], we also present a fairly complex generalization of the space lower bound for systems containing b -buffer objects, where a `write` operation may be performed on multiple memory locations at once, which is called *multi-assignment*. Gelashvili's thesis [33] also contains details about this generalization.

Augmented Snapshots. An m -component *atomic snapshot* object consists of a sequence of m memory locations. Each memory location j may be updated using an `update(j, v)` operation, for arbitrary values v , and the contents of *all* m memory locations may be *atomically* viewed using a single operation, `scan`. Afek *et al.* [2] showed that an m -component atomic snapshot object may be implemented in a wait-free manner using only m registers.

In Chapter 5 of this thesis, we introduce an m -component *augmented snapshot* object, which is a generalization of an m -component atomic snapshot object. In addition to `update` and `scan`, our object allows each process to *attempt* to `update` multiple memory locations atomically using a `block-update`

operation. The main purpose of the augmented snapshot object is to facilitate our simulations in Chapter 6.

In the special case when there is only a single, fixed process performing **block-update** operations, the **updates** in a **block-update** operation by this process always occur atomically. Moreover, the **block-update** returns a view of the m memory locations at an earlier point in time such that only **update** operations by other processes occur between that point and the **block-update** operation. In this thesis, we show that, using only registers, it is possible to implement this simple augmented snapshot object in a *non-blocking* manner, i.e. if some processes keep taking steps, then some operation eventually completes. This is somewhat surprising since the special process may use a simple augmented snapshot object to implement multi-assignment and it is provably impossible to implement multi-assignment in a non-blocking manner from registers [37]. The key, however, is that *only* the special process can perform multi-assignment operations.

The general definition of the augmented snapshot object is fairly complex. To implement an augmented snapshot object in a non-blocking manner from registers, we weaken the general definition of a **block-update**. Roughly speaking, we assign processes fixed, distinct priorities and we allow a **block-update** operation to return a special “yield” value, which indicates that the **updates** in the **block-update** did not occur atomically, but there was a concurrent **block-update** by a higher priority process whose **updates** did occur atomically. Any **block-update** operation that does not return “yield” must satisfy the same properties as in the special case.

Revisionist Simulations. The space complexity of k -set agreement, which is a generalization of consensus, is still unresolved. Recall that, in this problem, each process begins with an input and the processes must collectively output at most $k < n$ different input values. In three concurrent papers by Borowsky and Gafni [16], Herlihy and Shavit [41], and Saks and Zaharoglou [51], it was shown that there is no deterministic, wait-free solution to k -set agreement. The best obstruction-free solutions use $n - k + 1$ registers and are anonymous [55, 19]. Delporte-Gallet, Fauconnier, Kuznetsov, and Ruppert [23] proved that it is impossible to solve obstruction-free k -set agreement using 1 register. For anonymous solutions, they also proved a lower bound of $\sqrt{\frac{n}{k} - 2}$ registers, which still leaves a large gap between the lower and upper bounds.

The space complexity of approximate agreement is also unresolved. In ϵ -approximate agreement, for any fixed $\epsilon > 0$, each process has an input in $\{0, 1\}$. The processes are required to output values in the interval $[0, 1]$ that are all within ϵ of each other. Moreover, each output value must lie between the smallest input and the largest input. Interestingly, this problem *can* be deterministically solved in a wait-free manner. The only space lower bound for this problem, $\Omega(\log(\frac{1}{\epsilon}))$, was in a restricted setting with single-bit registers, due to Schenk [52]. The best upper bounds are $\lceil \log_2(\frac{1}{\epsilon}) \rceil$ registers, due to Schenk [52], and n registers, due to Attiya, Lynch, and Shavit [10].

In Chapter 6 of this thesis, we prove that any obstruction-free solution to k -set agreement must use at least $\lceil \frac{n}{k} \rceil$ registers. As a corollary, we get a tight lower bound of n registers for consensus. We also prove that any obstruction-free solution to ϵ -approximate agreement must use at least $\min\{\lceil \frac{n}{2} \rceil + 1, \sqrt{\log_2 \log_2(1/\epsilon)/2}\}$ registers. Hence our bound is asymptotically tight, for sufficiently small ϵ .

Our approach is to use a simulation argument. Specifically, if a solution uses too few registers, then we show that it is possible to simulate it in a wait-free manner so that all processes output values after taking a certain number of steps, which is a function of the number of registers used. Then we show

that this violates some known impossibility result or complexity lower bound.

To bound the number of steps taken by processes before they output a value, we introduce a new simulation technique, which we call a *revisionist simulation*. Using an augmented snapshot object, processes have the ability to indistinguishably revise the past of the simulated execution (by inserting hidden simulated steps using the view returned by an atomic `block-update` operation). This allows the processes to simulate executions similar to the ones constructed in our space lower bound for consensus. The key is that it is possible to bound the lengths of these executions and, hence, the number of steps taken by any process, in terms of the number of registers used in the execution.

In our simulation, each simulator assigns its own input to the processes it simulates. It outputs the value that one of its simulated processes output. Colourless tasks allow the input or output of any process to be the input or output of any other process. Hence, our technique can also be used to prove space lower bounds for arbitrary colourless tasks.

Extension-based Proofs. The early proofs of the impossibility of deterministic, wait-free k -set agreement [16, 41, 51] all relied on sophisticated machinery from combinatorial topology, which they used to model the space of all reachable configurations of the system. Later on, Attiya and Castañeda [8] and Attiya and Paz [11] showed how to obtain the same results using combinatorial techniques, without explicitly using topology. A common feature of these impossibility proofs is that they are non-constructive. In particular, they show that, if every execution has finite length, then there is some execution in which $k+1$ different values are output. Hence, there is an infinite execution, but it is not explicitly constructed.

In contrast, impossibility proofs for deterministic, wait-free consensus explicitly construct an infinite execution by repeatedly extending a finite execution by the steps of some processes. The valency argument introduced by Fischer, Lynch, and Paterson [31] shows that such extensions are always possible for consensus. Is there a proof of the impossibility of k -set agreement that explicitly constructs an infinite execution by repeated extensions?

In Chapter 7 of this thesis, we formally define the class of *extension-based proofs*. Valency arguments and covering arguments are examples of extension-based proofs. We prove that there is no extension-based proof of the impossibility of a deterministic, wait-free solution for k -set agreement among $n > k \geq 2$ processes in asynchronous systems containing only registers.

Chapter 2

Preliminaries

In this chapter, we formally define colourless tasks, describe our primary model of computation, the asynchronous shared memory model, and define some terminology and notation used throughout the thesis.

2.1 Colourless Tasks

In a distributed task, each process has a private input and should produce a single output value such that, collectively, the output values satisfy the specifications of the task. It is convenient to describe tasks in terms of combinatorial topology. We give some basic definitions, which will also be used in Chapter 7, and use them to define colourless tasks.

An *(abstract) simplex* is the set of all subsets of some finite set. An *(abstract) simplicial complex* is a finite collection of sets, \mathbb{S} , that is closed under subset: for any set $\sigma \in \mathbb{S}$, if $\tau \subseteq \sigma$, then $\tau \in \mathbb{S}$. In other words, \mathbb{S} is the union of a finite number of simplices. Each set $\sigma \in \mathbb{S}$ is called a *face*. If $|\sigma| = 1$, then σ is called a *vertex*. If $|\sigma| = 2$, then σ is called an *edge*. A *subcomplex* of \mathbb{S} is a subset of \mathbb{S} that is also a simplicial complex.

A *colourless* task is described by a triple $(\mathbb{I}, \mathbb{O}, \Delta)$, where \mathbb{I} is the *input complex*, \mathbb{O} is the *output complex*, and Δ maps each simplex in \mathbb{I} to a subcomplex of \mathbb{O} such that, for any simplices $\sigma, \sigma' \in \mathbb{I}$, $\Delta(\sigma \cap \sigma') \subseteq \Delta(\sigma) \cap \Delta(\sigma')$. The interpretation is that each set in \mathbb{I} specifies an allowable combination of inputs and each set in \mathbb{O} specifies an allowable combination of outputs. Initially, each process is assigned an input such that the set of all their inputs, σ , is in \mathbb{I} . The set of outputs, τ , produced by the processes must be in $\Delta(\sigma)$. If $\tau \in \Delta(\sigma)$, then we say τ is a *correct set of outputs for σ* . Finally, the condition on Δ implies that if a set of outputs is correct for a subset of an input set, then it is also correct for the input set. The condition is called *monotonicity*.

Colourless tasks were introduced by Borowsky, Gafni, Lynch, and Rajsbaum [15], who called them *convergence tasks*. Herlihy and Rajsbaum [40] later called them colourless tasks and studied their topological properties in detail. Herlihy, Kozlov, and Rajsbaum [38] survey results about colourless tasks. Typically, the number of processes in the system, as well as the identifier of each process (which is sometimes referred to as the *colour* of the process), may be used in the definition of a task. However, colourless tasks may be defined without reference to these properties.

We now formally define k -set agreement problem, for integers $k \geq 1$, as a colourless task. Recall

that, in this problem, every process begins with an arbitrary input and must output a value such that each output value is the input of some process (validity) and the processes collectively output at most k different values (k -agreement). More formally, k -set agreement is the colourless task $(\mathbb{I}, \mathbb{O}, \Delta)$, where \mathbb{I} is the simplex whose vertices are the possible inputs, \mathbb{O} is the subcomplex of \mathbb{I} consisting of all subsets of cardinality at most k , and $\Delta(\sigma) = \{\tau \subseteq \sigma : |\tau| \leq k\}$, for each $\sigma \in \mathbb{I}$. In the m -valued version of k -set agreement, \mathbb{I} is the simplex whose vertices are $\{0, 1, \dots, m-1\}$. If we do not specify the domain of the inputs, then we assume that there are at least $k+1$ different inputs, including $\{0, 1, \dots, k\}$. The 1-set agreement problem is also called *consensus* and we refer to 2-valued consensus as *binary consensus*.

We also define the ϵ -approximate agreement problem, for real numbers $\epsilon > 0$, as a colourless task. Recall that, in this problem, every process begins with an input in $\{0, 1\}$ as input and must output a real number between the smallest and largest input values (validity) and the absolute value of the difference between any two output values is at most ϵ (ϵ -agreement). More formally, ϵ -approximate agreement is the colourless task specified by the triple $(\mathbb{I}, \mathbb{O}, \Delta)$, where \mathbb{I} is the simplex with vertices $\{0, 1\}$, \mathbb{O} is the collection of all finite subsets $\tau \subseteq [0, 1]$ with $\max(\tau) - \min(\tau) \leq \epsilon$, and $\Delta(\sigma) = \{\tau \subseteq [\min(\sigma), \max(\sigma)] : \tau \in \mathbb{O}\}$, for each non-empty $\sigma \in \mathbb{I}$.

2.2 Asynchronous Shared Memory

An asynchronous shared memory system consists of a set of $n \geq 2$ processes, a collection of shared objects, and a scheduler. The processes can only communicate with each other by accessing (i.e. performing operations on) the shared objects. The order in which they do so is determined by an (adversarial) scheduler.

2.2.1 Objects

An *object* has a set of possible values, called its *domain*, and a set of supported *operations*, each of which takes some fixed number of inputs, returns a response, and possibly modifies the value of the object. For example, a *read* operation simply returns the current value of the object and does not change the object's value. Each *instance* of an object may be *initialized* to an arbitrary value in the object's domain. Henceforth, we will denote an object instance using an uppercase letter in sans serif font and we will denote operations on an object using lower case words in the same font. For example, $X.op$ denotes the operation *op* on an instance X of some object.

An object is *deterministic* if the response of each operation and the resulting value of the object after the operation are functions of the parameters of the operation and the object's value immediately prior to the operation. In this thesis, we only consider deterministic objects.

We now formally define the objects considered in this thesis. (An additional type of object, which may be viewed as a collection of objects, is presented in Section 2.6.)

- For $b \geq 1$, a b -buffer object, B , with domain D^b , supports two operations, $write(v)$, for any value $v \in D$, and $read$. If the current value of B is (v_1, \dots, v_b) , then a $write(v)$ operation sets the value of B to (v, v_1, \dots, v_{b-1}) and returns an acknowledgement, ACK. A *register* is a 1-buffer object. We say a register is *single-writer* if it is only ever written to by a single process.
- A *swap* object with domain D supports two operations, $swap(v)$, for any value $v \in D$, and $read$. The $swap(v)$ operation sets the value of the object to v and returns the previous value of the object.

- A *test-and-set* object, T , supports two operations **test-and-set** and **read**. The domain of T is $\{0, 1\}$. A **test-and-set** operation sets the value of T to 1 and returns the previous value of T .
- A *max-register*, M , with a totally ordered domain D supports two operations, **write-max**(v), for any value $v \in D$, and **read**. The **write-max**(v) operation sets the value of M to v if and only if the current value of M is less than v and returns an acknowledgment, **ACK**.

In many of our protocols using these objects, D is the set of non-negative integers, which we denote as \mathbb{N} . We use $[m]$ to denote the set $\{1, \dots, m\}$. Finally, we denote a vector of values as \vec{v} and use the notation \vec{v}_i to mean the i 'th component of \vec{v} .

2.2.2 Processes

Each process has a set of states, which depends on the algorithm it is running. A process begins by waiting for an input value from the environment. Once it is assigned an input value, it performs operations on the shared objects in order to communicate with the other processes. The state of the process determines the next operation that it performs or the value that it outputs. After performing an operation on a shared object and receiving a response, the process performs local computation to determine its next state. Once a process outputs a value, it may either terminate or wait for another input value (depending on the problem it is trying to solve).

More formally, a process is a *nondeterministic state machine*, which is a 6-tuple $(S, I, O, \iota, \nu, \delta)$, where S is a set of *states*, $I \subseteq S$ is a set of *input* states, $O \subseteq S - I$ is a set of *output* states, $\iota \in I$ is the *initial* state, ν is a function that specifies the *next step* of the process at each state, and δ is a *transition* function that maps each state $s \in S$ and each possible response a of step $\nu(s)$ to a subset $\delta(s, a)$ of S . The transition function δ has the property that, if s is an output state, then $\delta(s, a)$ is a (possibly empty) subset of the input states, I . Otherwise, $\delta(s, a)$ is a *non-empty* subset of $S - I$. The state machine is *deterministic* if $|\delta(s, a)| \leq 1$ for each state $s \in S$ and each possible response a of step $\nu(s)$.

There are three types of steps: **input** steps, **output** steps, and *shared memory* steps. At each input state, the next step of the process is to perform an **input** step, which returns an input value as its response when it is performed. At each output state, the next step of the process is to perform an **output**(v) step, for some value v , which always returns an acknowledgment, **ACK**, as its response when it is performed. At all other states, the next step of the process is to perform a shared memory step, i.e. an operation on some shared object. Once the process performs the shared memory step, it receives the response of the operation that it performed.

Initially, the process starts at the initial state, ι , which is an input state. At each state $s \in S - O$, the process performs its next step, $\nu(s)$, and receives a response, a . Since s is not an output state, $\delta(s, a)$ is a non-empty subset of $S - I$. The process nondeterministically chooses a state $s' \in \delta(s, a)$ and transitions to state s' . Observe that, by definition of the transition function, after performing an **input** step, the process performs shared memory steps until it enters an output state. Once the process enters an output state $s \in O$, it performs $\nu(s)$, which is **output**(v), for some value v , and it receives an acknowledgment, $a = \text{ACK}$, as a response. If $\delta(s, a) = \emptyset$, then the process stays in state s and *terminates*, i.e. does not take any more steps. We require the process to stay in state s to ensure that, at all times, each process is in some state. If $\delta(s, a) \neq \emptyset$, the process nondeterministically chooses an input state $s' \in \delta(s, a) \subseteq I$ and transitions to state s' .

Notice that the definition of a nondeterministic state machine is very powerful. Specifically, if we view the state of a process as the contents of its local variables, then a transition allows the process to apply *any* function to its local variables. Hence, the processes are computationally unbounded. This is justified by the fact that we wish to understand the theoretical limits of the objects that the processes communicate with and we are unconcerned with local computation time.

2.2.3 The Scheduler

A *configuration* describes the state of the entire system, i.e. the state of each process and the contents of each shared object. In the *initial* configuration, each process is in its initial state and each shared object contains an initial value. A step e by a process p is *applicable* at a configuration C if it is the step associated with p 's state in C . In this case, we say that p is *poised* to perform step e in configuration C and we use Ce to denote the configuration resulting from p taking step e . A sequence of steps e_1, e_2, \dots is *applicable* at configuration C if there exists a sequence of configurations $C = C_0, C_1, C_2, \dots$ such that e_i is applicable at C_{i-1} and $C_i = C_{i-1}e_i$ for all $i \geq 1$.

An *execution* of the system from a configuration, C_0 , is an alternating sequence of configurations and steps, $C_0, e_1, C_1, e_2, \dots$, such that, for each $i \geq 1$, step e_i is applicable at configuration C_{i-1} and $C_i = C_{i-1}e_i$. An *empty* execution from C_0 consists of only C_0 . We say that an execution α is *P-only*, for some set of processes P , if each step in the execution is taken by some process in P . If $P = \{p\}$, for a single process p , then we say that α is a *solo* execution by p . Given a finite execution α from a configuration C , we use $C\alpha$ to denote the last configuration in the execution. A configuration C is *reachable* if there exists a finite execution α from the initial configuration I such that $C = I\alpha$.

An adversarial *scheduler* determines the order in which processes take steps. At any configuration, the scheduler *selects* a process that has not terminated, if one exists, and the process performs its next step, receives a response, and transitions to a new state. If the next step of the process is an *input* step, then the scheduler chooses the response of the *input* step. This response must be a valid input to the problem that the processes are trying to solve. The sequence of processes selected by the scheduler is called a *schedule*.

2.3 Protocols

A protocol defines a solution to a task by specifying the state machine of each process. Once a process outputs a value, it terminates. More formally, a *protocol* for a task in an asynchronous shared memory system specifies a nondeterministic state machine $M_p = (S_p, \{\iota_p\}, O_p, \nu_p, \delta_p)$ defining each process p . Notice that the initial state is the only input state. The transition function δ_p has the property that process p terminates immediately after taking an *output* step, i.e. for any output state s , $\delta_p(s, \text{ACK}) = \emptyset$. A protocol is *anonymous* if $M_p = M_q$, for all processes p and q . It is *deterministic* if each machine M_p is deterministic.

A protocol is *correct* if, in every execution from the initial configuration, the values *output* by the processes satisfy the specifications of the task. In other words, no matter how the scheduler assigns inputs to the processes and no matter how it schedules the processes, the processes never output values that violate the specification of the task.

2.4 Implementations

An implementation of an object specifies, for each process and each operation on the object, the procedure the process uses to carry out the operation, by specifying the state machine for each process. It also specifies the initial values of the shared objects. An input to a process is an invocation of an operation on the implemented object. Each value output by a process is considered to be the response to its most recent invocation. After a process outputs a value, it waits for another input.

More formally, an *implementation* of an object X specifies a nondeterministic state machine $M_p = (S_p, I_p, O_p, \iota_p, \nu_p, \delta_p)$ defining each process p . The transition function δ_p has the property that, once process p outputs a value, it transitions to another input state, i.e. for each output state s , $\emptyset \neq \delta_p(s, \text{ACK}) \subseteq I_p$. Notice that there may be multiple input states, which models the fact that a process may have persistent local variables. An implementation is *anonymous* if $M_p = M_q$, for all processes p and q . It is *deterministic* if each machine M_p is deterministic.

When p performs an input step, it receives, as a response, an operation to be performed by p on the implemented object, X , together with the input parameters for the operation. Each *invocation* of an operation on X by a process p *begins* with an input step by process p . The invocation *ends*, and the operation *completes*, when p next takes an `output(v)` step. The value, v , of the output step is the response of the operation. The *execution interval* of an invocation of an operation on X in an execution is the subsequence of the execution that begins when the invocation begins (with an input step) and ends when the invocation ends (with an output step). If an operation does not complete, for example, if the scheduler never allows the process that invoked the operation to take its output step, then the execution interval of the operation does not have an end.

An implementation of an object is *linearizable* if, for every execution, there is a point in the execution interval of each complete operation, called the *linearization point* of the operation, such that the operation can be said to have taken place atomically at that point. An incomplete operation may also have a linearization point. More formally, this means that it is possible to assign responses to a subset of the incomplete operations and order them together with the complete operations so that any operation which ends before another one begins is ordered earlier, and the responses of the operations are consistent with the specification of the object [42].

A deterministic implementation is *non-blocking* if, from any reachable configuration, there is no infinite execution in which only finitely many operations complete. Note that, in any infinite execution of a non-blocking implementation from a reachable configuration, there must be at least one process that completes infinitely many operations. However, the same is not necessarily true for other processes in the execution.

A *timestamp* is a label from a partially ordered set, which can be associated with an operation in an implementation, such that, if one operation completes before another operation begins, the first operation has a smaller timestamp [45]. A *vector timestamp* [30, 47, 14] is an n -component vector of non-negative integers, with one component per process. Vector timestamps are ordered lexicographically.

2.5 General Progress Conditions

An *algorithm* is either a protocol for a task or an implementation of an object.

For $x \in [n]$, a deterministic algorithm is *x -obstruction-free* if, for every reachable configuration C and

every set of processes P with $|P| \leq x$, there is no infinite P -only execution from C in which some process in P takes infinitely steps and does not output a value. A deterministic algorithm is *obstruction-free* if it is 1-obstruction-free and *wait-free* if it is n -obstruction-free. Taubenfeld [53] introduced x -obstruction-freedom as a termination condition that interpolates between obstruction-freedom and wait-freedom.

In a *randomized* algorithm, for every process, every state of the process, and every response to the step that the process takes at the state, there is a probability distribution over the set of possible next states of the process and it chooses one from this distribution. (This models the fact that a process may have a private source of randomness, e.g. fair coins, which it uses to determine the steps it takes.) In a *randomized wait-free* algorithm, from any reachable configuration, each process **outputs** a value after taking finitely many steps, with probability 1. Notice that, if the algorithm is an implementation of an object, then this implies that each operation invoked by a process completes after it takes finitely steps, with probability 1.

A *solo-terminating* execution by a process p from a configuration C is a solo execution by p from C in which it outputs a value. An algorithm is *nondeterministic solo-terminating* if, for every process p and from every reachable configuration C of the algorithm, there *exists* a *solo-terminating* execution by p . Observe that, if the algorithm is deterministic, then nondeterministic solo-termination is the same as obstruction-freedom. Randomized wait-free algorithms are also nondeterministic solo-terminating.

2.6 Snapshots of Readable Objects

An object is *readable* if it has an operation that returns the current value of the object, without modifying it. Without loss of generality, we assume that a readable object has a `read` operation. For example, buffer objects, test-and-set objects, and max-registers are all readable.

Given a sequence of readable objects R_1, \dots, R_m , a `collect` operation on R_1, \dots, R_m performs $R_1.\text{read}$, $R_2.\text{read}$, \dots , $R_m.\text{read}$, one at a time, and returns an m -component vector \vec{v} , where \vec{v}_i contains the value read from R_i . There may not be a moment in time during the collect where R_1, \dots, R_m have values $\vec{v}_1, \dots, \vec{v}_m$, respectively.

A *snapshot* object, R , consists of a sequence of m readable objects, R_1, \dots, R_m . If the domain of object R_i is D_i , for $i \in [m]$, then the domain of R is $D_1 \times \dots \times D_m$. For each integer $i \in [m]$, R supports all operations of object R_i . We denote an operation `op` on the i 'th object in R by $R_i.\text{op}$. The object also supports a `scan` operation, which returns an m -component vector \vec{v} containing the current value, \vec{v}_i , of each object, R_i . We denote a `scan` operation on R by `scan`(R_1, \dots, R_m).

A special case of a snapshot object is when the object, R , consists of m registers, R_1, \dots, R_m . In the literature, this is referred to as an *m -component atomic snapshot* object. In this case, to be consistent with the terminology and notation used in the literature, register R_i is referred to as *component i* of R , each $R_i.\text{write}(v)$ operation is denoted by $R.\text{update}(i, v)$, and `scan`(R_1, \dots, R_m) is denoted by $R.\text{scan}$. A *single-writer* atomic snapshot object is an n -component atomic snapshot object where the i 'th component of the object is only **updated** by the i 'th process.

Afek *et al.* [2] showed that an m -component atomic snapshot object can be deterministically implemented in a linearizable, non-blocking manner from m registers. (They also extended their non-blocking implementation to make it wait-free.) The basic idea of their non-blocking implementation is to ensure that no value is written to the same component more than once. This can be accomplished by having each process include its *identifier* (a unique value that is part of the state machine of the process) and a

monotonically increasing *sequence number* (a locally stored counter value) along with the value that it writes. Then, to implement `scan`, a process repeatedly performs collects on the registers until it performs two consecutive collects that return the same vector of values, \vec{v} . Since no value is written to the same component more than once, the registers had values $\vec{v}_1, \dots, \vec{v}_m$, respectively, at all times between the two consecutive collects. Hence, the `scan` is linearizable.

Guerraoui and Ruppert [35] showed that it is possible to remove the process identifiers in Afek *et al.*'s implementation at the cost of the processes performing more collects in the implementation of `scan`. While removing the identifiers makes it possible for a value to be written more than once to a component, the sequence numbers ensure that no value is written more than n times to a component. In this case, to implement `scan`, a process repeatedly performs collects on the registers until it performs $m(n-1)+2$ consecutive collects that all return the same vector of values, \vec{v} . Since no value can be written more than n times to the same component, this ensures that, among the $m(n-1)+2$ consecutive collects returning \vec{v} , there is a pair of consecutive collects such that the registers had values $\vec{v}_1, \dots, \vec{v}_m$ at all times between the two collects. Hence, the `scan` can be linearized between these two collects.

We say a sequence of operations on a object causes the object to *experience an ABA* for a value v in its domain if the value of the object before the sequence is v , the value of the object after some operation in the sequence is $v' \neq v$, and the value of the object at the end of the sequence is v again. We observe that, if a readable object supports operations that causes it to experience at most n ABA's for each value in its domain, then Guerraoui and Ruppert's implementation of `scan` is linearizable. Notice that sequence numbers allow us to bound the number of ABA's experienced by each value.

A max register or test-and-set object never experiences an ABA for any value in its domain. We can append sequence numbers to bound the number of ABA's experienced by any value in the domain of a swap object or a buffer object. Hence, we have the following theorem.

Theorem 2.1. *There is an anonymous, linearizable, non-blocking implementation of a snapshot object consisting of m readable objects, each of which is either a max-register, a test-and-set object, a swap object, or a buffer object.*

Aspnes *et al.* [6] gave a wait-free implementation of a snapshot object consisting of 2 max-register objects, each of which has a bound on the maximum value written to the object. Afek *et al.*'s [2] wait-free implementation of an atomic snapshot object can be generalized to yield wait-free implementations of snapshot objects consisting of swap and buffer objects.

Chapter 3

Nondeterministic Solo-termination vs Obstruction-freedom

In this chapter, we prove the following theorem, which relates the space complexity of nondeterministic solo-terminating algorithms and (deterministic) obstruction-free algorithms in asynchronous shared memory systems containing only deterministic readable objects.

Theorem 3.1. *If there is a nondeterministic solo-terminating algorithm that uses a finite number of deterministic readable objects, then there is a (deterministic) obstruction-free algorithm solving the same task or implementing the same object that uses the same objects. Moreover, if the nondeterministic solo-terminating algorithm is anonymous, then so is the obstruction-free algorithm.*

The contrapositive of this theorem implies that the space complexity of solving a task (or implementing an object) in a nondeterministic solo-terminating manner from deterministic readable objects is the same as the space complexity of deterministically solving the task (or implementing the object) in an obstruction-free manner from the same types of objects.

3.1 Related Work

Ellen, Herlihy, and Shavit [28] introduced *nondeterministic solo-termination* as a progress condition for nondeterministic algorithms, which generalizes both deterministic obstruction-free and randomized wait-free algorithms. In the same paper, they proved space lower bounds for nondeterministic solo-terminating algorithms using only historyless objects, which support operations that can only change the object to have a value that does not depend on its current value. (Registers and swap objects are both examples of historyless objects.) Hence, they obtain the same space lower bounds for randomized wait-free and deterministic obstruction-free algorithms using historyless objects.

Herlihy, Luchango, and Moir [39] studied deterministic solo-terminating implementations. They introduced the term *obstruction-free* to refer to such implementations. They devised obstruction-free implementations of objects, which are much simpler than non-blocking implementations. It has become standard to use the term obstruction-free to refer to deterministic solo-terminating protocols as well.

There are two different types of adversaries that are considered in randomized computation. An *oblivious* adversary must choose the sequence of responses of the input steps of each process and fix a

schedule in advance. It does not get to see the coin flips of processes. On the other hand, an *adaptive* adversary sees the results of all coin flips as they occur and it can choose the responses of input steps and schedule processes based on this information.

Giakkoupis, Helmi, Higham, and Woelfel [34] showed that it is possible to transform an obstruction-free algorithm into a randomized wait-free algorithm, against an oblivious adversary, while maintaining its space complexity. Their result holds for algorithms using base objects that support a *trivial* operation, such as `read`, which does not change the value of the object. They show that, if the obstruction-free algorithm has a known bound, b , on the maximum length of any solo execution, then each process in the resulting randomized wait-free algorithm terminates after taking $O(b(n + b) \log n)$ steps in expectation, and with high probability. The basic idea is that, every b steps, each process flips a coin to determine if it continues performing the algorithm during its next b steps or if it performs b trivial steps. Eventually, each process performs b consecutive steps of the algorithm without interference from the other processes, and terminates.

Ellen, Luchango, Moir, and Shavit [29] showed that it is possible to transform an obstruction-free algorithm designed for an asynchronous shared memory system into a wait-free algorithm when the system provides an upper bound on the amount of elapsed time between consecutive steps of any process. Specifically, they show that, even if this bound is not known to the processes, the resulting transformed algorithm is wait-free. Their transformation does not preserve the space complexity of the obstruction-free algorithm.

3.2 The Transformation

In this section, we describe a general transformation that takes a nondeterministic solo-terminating algorithm, \mathbf{A} and converts it into an obstruction-free algorithm, \mathbf{B} that solves the same task or implements the same object. The transformation works so long as \mathbf{A} is designed for an asynchronous shared memory system containing only readable objects. In particular, the transformation works for all systems that contain only the objects considered in this thesis.

3.2.1 Sketch

The idea of the conversion is simple: After taking a step of \mathbf{A} , a process performs a `collect` on all the objects to determine the next step that it takes. Suppose there is a reachable configuration C of algorithm \mathbf{A} in which the process has its current state (excluding the result of the `collect`) and the contents of the objects match the result of the `collect` it just performed. Since C is reachable and \mathbf{A} is nondeterministic solo-terminating, the process has a solo execution from C in which it `outputs` a value. In this case, the process deterministically picks a shortest such execution α and takes the first step in α , which is another step of \mathbf{A} . Otherwise, it performs another `collect` and tries again.

A `collect` does not affect the contents of the objects. By removing the `reads` performed in `collects`, each execution of \mathbf{B} can be converted to an execution of \mathbf{A} . Hence, \mathbf{B} solves the same task or implements the same object as \mathbf{A} . In any solo execution of a process, the result of a complete `collect` contains the contents of the objects in the current (reachable) configuration, so the process alternately performs `collects` and steps of \mathbf{A} in the solo execution. Since the lengths of the solo executions α that it picks are decreasing, it eventually `outputs` a value. Hence, \mathbf{B} is obstruction-free.

A subtle point is that it may not be possible for a process to compute, in finite time, whether or not there is a reachable configuration C of \mathbf{A} in which the process has the same state and the contents of the objects match the result of its last `collect`. The difficulty is that a process may have infinitely many possible next states from a particular state. Hence, \mathbf{A} may have infinitely many reachable configurations. Likewise, the process may not be able to determine a shortest solo execution α from C in which it `outputs` a value. However, since the transition function for each process may be arbitrary, we can represent \mathbf{B} as a collection of deterministic state machines.

3.2.2 Formal Description

Suppose that algorithm \mathbf{A} uses m deterministic readable objects, R_1, \dots, R_m , with domains D_1, \dots, D_m . For each process p , let $M_p = (S_p, I_p, O_p, \iota_p, \nu_p, \delta_p)$ be the nondeterministic state machine that \mathbf{A} specifies for p . By the Well-Ordering Theorem, it is possible to define a total order on S_p such that, for any non-empty subset S of S_p , there is a smallest element in S , i.e. $\min(S)$ exists. Fix such a total ordering on S_p . To define the deterministic obstruction-free algorithm \mathbf{B} , for each process p , we define a deterministic state machine $M'_p = (S'_p, I'_p, O'_p, \iota'_p, \nu'_p, \delta'_p)$ from M_p as follows.

Let \square be a special symbol that is not the response of any step of \mathbf{A} . Each state in S'_p is a triple (s, a, \vec{v}) , where $s \in S_p$, a is either a response to step $\nu_p(s)$ or \square , and $\vec{v} \in (\cup_{i=0}^{m-1} \prod_{j=1}^i D_j)$ is a (possibly empty) proper prefix of an m -component vector in $D_1 \times \dots \times D_m$. The set of input states is $I'_p = I_p \times \{\square\} \times \{\varepsilon\}$ and the set of output states is $O'_p = O_p \times \{\square\} \times \{\varepsilon\}$, where ε denotes the empty prefix. The initial state, ι'_p , of p is $(\iota_p, \square, \varepsilon)$. Given a state $(s, a, \vec{v}) \in S'_p$, we define $\nu'_p(s, a, \vec{v})$ and $\delta'_p((s, a, \vec{v}), r)$, for each possible response r of $\nu'_p(s, a, \vec{v})$, by considering a number of cases.

Suppose that $a = \square$. In this case, p has not yet performed the step, $\nu_p(s)$, in \mathbf{A} that is associated with state s . If s is not an output state, then we define $\nu'_p(s, \square, \varepsilon) = \nu_p(s)$ and, for each possible response r to $\nu_p(s)$, we define $\delta'_p((s, \square, \varepsilon), r) = \{(s, r, \varepsilon)\}$. Now suppose s is an output state. Then $\nu_p(s)$ is an output step, which always returns `ACK` as a response. If \mathbf{A} is a protocol, then $\delta_p(s, \text{ACK}) = \emptyset$, i.e. p terminates in \mathbf{A} after taking step $\nu_p(s)$. In this case, we define $\delta'_p((s, \square, \varepsilon), \text{ACK}) = \emptyset$, so p terminates in \mathbf{B} as well. If \mathbf{A} is an implementation, then, by definition, $\delta_p(s, \text{ACK}) \neq \emptyset$. In this case, we define $\delta'_p((s, \square, \varepsilon), \text{ACK}) = \{(s', \square, \varepsilon)\}$, where $s' = \min(\delta'_p(s, \text{ACK}))$. Notice that, in all of the cases, we only need to define $\nu'_p(s, \square, \varepsilon)$ and $\delta'_p(s, \square, \varepsilon)$ since p does not start performing `collects` until it performs $\nu_p(s)$ and receives a response. (Recall that \square is not the response of any step.) Hence, p will never reach state (s, \square, \vec{v}) , for any $\vec{v} \neq \varepsilon$, in \mathbf{B} .

Now suppose that $a \neq \square$. In this case, p has performed $\nu_p(s)$, received response a from the step, and is currently performing a `collect`. In particular, \vec{v} is a j -component vector, for some $0 \leq j < m$, which contains the values that p read from R_1, \dots, R_j in its current, partial `collect`. We define $\nu'_p((s, a, \vec{v})) = R_{j+1}.\text{read}$. If $j + 1 < m$, then p has not finished its `collect` and we define $\delta'_p((s, a, \vec{v}), r) = \{(s, a, \vec{v} \cdot r)\}$, for each possible response r of $R_{j+1}.\text{read}$, where $\vec{v} \cdot r$ denotes the concatenation of \vec{v} and r .

If $j + 1 = m$, then $\vec{v} \cdot r$ is the result of a complete `collect`. Consider the set, \mathcal{C} , of all reachable configurations of \mathbf{A} in which p 's state is in $\delta_p(s, a)$ and the contents of R_1, \dots, R_m are equal to $\vec{v} \cdot r$. If $\mathcal{C} = \emptyset$, then we define $\delta'_p((s, a, \vec{v}), r) = \{(s, a, \varepsilon)\}$, so that p performs another `collect`. Now suppose that $\mathcal{C} \neq \emptyset$. Since \mathbf{A} is nondeterministic solo-terminating, for each configuration $C \in \mathcal{C}$, there is a solo execution of p from C in which it `outputs` a value and the length, $\ell(C)$, of a shortest such execution is defined. Let $\ell = \min\{\ell(C) : C \in \mathcal{C}\}$, let $S \subseteq \delta(s, a)$ be the set of states of p in configurations $C \in \mathcal{C}$ where $\ell(C) = \ell$, and let $s' = \min(S)$. Then we define $\delta'_p((s, a, \vec{v}), r) = \{(s', \square, \varepsilon)\}$.

3.3 Proof of Correctness/Linearizability

We now prove that transformed algorithm, \mathbf{B} , in the previous section solves the same task or implements the same object as \mathbf{A} and is obstruction-free.

We first map each state of a process p in \mathbf{B} to either a single state in S_p or a non-empty subset of S_p . Each state $(s, \square, \varepsilon) \in S'_p$ maps to s . In particular, ι'_p maps to ι_p . If $s \in S_p$ is not an output state, then each state $(s, a, \vec{v}) \in S'_p$ where $a \neq \square$, maps to $\delta_p(s, a)$. If s is an output state and \mathbf{A} is a protocol, then $(s, \text{ACK}, \varepsilon)$ maps to s . If s is an output state and \mathbf{A} is an implementation, then $(s, \text{ACK}, \varepsilon)$ maps to $\min(\delta_p(s, \text{ACK}))$.

We rely on the following key observation: The state of every process in \mathbf{B} maps to either one of its states in \mathbf{A} , which it has entered, or a set of its states in \mathbf{A} , each of which it *could* have entered. Moreover, whenever a process transitions to a state in \mathbf{B} mapping to one of its states s' in \mathbf{A} , by definition of δ'_p , p arrived at $(s', \square, \varepsilon)$ from a state (s, a, \vec{v}) , where $s' \in \delta'_p(s, a)$. Hence, when the state of p in \mathbf{B} last mapped to one of its states in \mathbf{B} , it mapped to the state s in \mathbf{A} .

It follows that every reachable configuration D of \mathbf{B} maps to a non-empty set, $\mathcal{C}(D)$, of reachable configurations of \mathbf{A} , one for each possible combination of states that the processes states in \mathbf{B} could be mapped to. Since collects do not change the contents of any shared object, the contents of the shared objects are the same at D and each configuration $C \in \mathcal{C}(D)$. Moreover, if α is an execution of \mathbf{A} from the initial configuration that results in a configuration $C \in \mathcal{C}(D)$ and β is an execution of \mathbf{B} from the initial configuration that results in D , then the steps in α are the same as the steps β , but with the reads performed in collects of β removed.

3.3.1 \mathbf{B} is correct/linearizable

To prove that \mathbf{B} is a correct protocol or linearizable implementation, we consider two cases.

Suppose \mathbf{A} is a protocol for a task T . Consider a reachable configuration D of \mathbf{B} . If some incorrect values have been output in D , then the same incorrect values have been output in some reachable configuration $C \in \mathcal{C}(D)$ of \mathbf{A} , which D maps to. Since \mathbf{A} is a correct protocol for T , it follows that \mathbf{B} is a correct protocol for T .

Now suppose that \mathbf{A} is a linearizable implementation of some object \mathcal{X} . To show that \mathbf{B} is linearizable, consider a reachable configuration D of \mathbf{B} . By the preceding observation, D maps to at least one reachable configuration $C \in \mathcal{C}(D)$ of \mathbf{A} . Let α be the execution of \mathbf{A} from the initial configuration that results in C and let β be the execution of \mathbf{B} from the initial configuration that results in D . Since \mathbf{A} is linearizable, there is a way to linearize the operations in α . Since the steps in α are the same as the steps β , but with the reads performed in collects of β removed, it follows that the same linearization works for β . Therefore, \mathbf{B} is a linearizable implementation of \mathcal{X} .

3.3.2 \mathbf{B} is obstruction-free

Assume, for a contradiction, that there is an infinite solo execution β by some process p from a reachable configuration D_0 of \mathbf{B} , in which the process never outputs a value. Then p performs infinitely many complete collects in β . Without loss of generality, assume that β begins with a complete collect. For $i \geq 1$, let γ_i be the last read (on \mathbf{R}_m) performed in the i 'th complete collect in β and let D_i be the configuration in β immediately after γ_i .

Consider the state $(s, a, \vec{v}) \in S'_p - O'_p$ of p the configuration in β immediately before γ_i . Let r be the response of γ_i . By definition of δ'_p , $a \neq \square$ and \vec{v} is an $(m - 1)$ -component vector containing the responses of the reads from R_j , for $1 \leq j \leq m - 1$, performed prior to γ_i . Since D_{i-1} is a reachable configuration of \mathbf{B} , it maps to a non-empty set of reachable configurations of \mathbf{A} , i.e. $\mathcal{C}(D_{i-1})$. At each configuration $C \in \mathcal{C}(D_{i-1})$, the state of p is in $\delta_p(s, a)$ and the contents of the objects are $\vec{v} \cdot r$. It follows that the set of reachable configurations, \mathcal{C}_i , of \mathbf{A} , in which the state of p is in $\delta_p(s, a)$ and the contents of the objects are equal to $\vec{v} \cdot r$, is non-empty. Let $\ell_i = \min\{\ell(C) : C \in \mathcal{C}_i\}$. By construction, $\delta'_p((s, a, \vec{v}), r) = \{(s', \square, \varepsilon)\}$, where s' is the state of p in a configuration $C \in \mathcal{C}_i$ where $\ell(C) = \ell_i$.

To conclude the proof, we note that $\ell_i > \ell_{i+1}$. This is because each solo-terminating execution of p from D_i , with the first step removed, is also a solo-terminating execution of p from D_{i+1} . This implies that there is an infinite sequence of decreasing natural numbers, which is a contradiction. Therefore, \mathbf{B} is obstruction-free.

It is tempting to think that, because we always select the shortest solo executions, we may be able to bound the solo step complexity of the resulting obstruction-free protocol, \mathbf{B} . For instance, if the original protocol, \mathbf{A} , is randomized wait-free, it would be nice to be able to bound the solo step complexity of \mathbf{B} by the expected step complexity of \mathbf{A} . However, we cannot do so in a straightforward way because it may be possible for a process to reach a configuration that occurs with low probability in \mathbf{A} , from which the process has large solo step complexity.

Chapter 4

Consensus

In this chapter, we study the space complexity of consensus in various asynchronous shared memory systems. In Section 4.1, we describe the existing work in this area. In Section 4.2, we prove that, for any integer $b \geq 1$, any protocol solving consensus among $n \geq 2$ processes in an obstruction-free manner using only b -buffer objects must use at least $\lceil (n-1)/b \rceil$ objects. In particular, when $b = 1$, our lower bound says that $n - 1$ registers are necessary. In Section 4.3, we show that this bound is almost tight by presenting a simple, anonymous protocol using $\lceil n/b \rceil$ objects. In the rest of the chapter, we prove space bounds for asynchronous shared memory systems with other types of base objects. Specifically, in Section 4.4, we prove that an unbounded number of test-and-set objects are necessary to solve binary consensus in an obstruction-free manner among $n \geq 3$ processes. In Section 4.5, we prove that two max-registers are necessary and sufficient to solve binary consensus in an obstruction-free manner among $n \geq 2$ processes. Finally, in section 4.6, we prove that it is possible solve consensus in an obstruction-free manner among $n \geq 2$ processes using $n - 1$ swap objects.

4.1 Related Work

The notion of valency was introduced by Fischer, Lynch, and Paterson [31] to prove the impossibility of deterministically solving wait-free consensus in asynchronous systems where processes may crash. They consider a configuration of a consensus protocol to be *bivalent* if there are two executions from the configuration in which different values are output.

A key observation is that if a configuration is bivalent, then no process has yet **output** a value in the configuration. Hence, no process has terminated and every process has a next step. The main challenge in Fischer, Lynch, and Paterson's result was to show that it is possible to maintain bivalency even as processes take more steps.

Covering arguments were introduced by Burns and Lynch [21] in order to prove a lower bound on the number of registers needed to solve the mutual exclusion problem. Since then, it has become the main technique used in proving space lower bounds.

The basic idea of a covering argument for registers is as follows. Suppose that there is a set of processes P and a set of registers V , such that, for each register $r \in V$, there is some process in P that is about to perform a **write** to r . Then it is possible for the processes in P to perform their **writes** to V all at once. This obliterates the contents of all registers in V . Hence, if there is another process, $q \notin P$,

that needs to communicate some information to the other processes, then it must write to a register not in V .

The lower bound in Section 4.2 combines a covering argument with a valency argument. In particular, we use a covering argument to show that more buffer objects need to be used as more processes participate. We use a valency argument to show that processes need to keep taking steps. We first proved this result for the case of 1-buffers, i.e. registers, in [56]. There, we also proved that n registers are sufficient. The lower bound when $b > 1$ appears in joint work with Ellen, Gelashvili, and Shavit [26]. In the same paper, we also present an obstruction-free protocol that solves consensus among n processes using $\lceil n/b \rceil$ different b -buffers. In Section 4.3, we show that there is an *anonymous* obstruction-free protocol using the same number of b -buffers.

Ellen, Herlihy, and Shavit [28] proved the first space lower bound for consensus. In particular, they proved that any nondeterministic solo-terminating protocol solving consensus using only registers must use $\Omega(\sqrt{n})$ registers. Similarly to our proof, their approach was to use a combination of a covering argument and a valency argument. However, the difference is that, in their proof, once a set of processes performs a block write, they do not take any more steps.

In the same paper, Ellen, Herlihy, and Shavit introduced the approach of studying anonymous protocols, which are easier to analyze. They showed that there is a simple proof of the $\Omega(\sqrt{n})$ bound in this case. To prove the bound, they introduced the idea of a *clone* of a process, which always takes the same steps as that process. As in their general argument, once a set of clones performs a block write, they do not take any more steps. By using the idea of clones much more elaborately, Gelashvili proved that $\Omega(n)$ registers are necessary in the anonymous case [32].

Remarkably, Ellen, Herlihy, and Shavit's space lower bound for consensus holds for systems with historyless objects. In particular, they prove that $\Omega(\sqrt{n})$ swap objects are necessary to solve consensus. However, neither Gelashvili's proof or our proof holds for such systems.

There are many randomized wait-free protocols for consensus [1, 4, 7, 9], all of which use either n or $n + 1$ registers. In particular, Attiya and Censor-Hillel [9] showed that it is possible to solve consensus such that the total number of steps taken by all processes is $O(n^2)$ in expectation. In the same paper, they also proved a matching lower bound.

Our protocol in Section 4.3 is based on the randomized wait-free consensus protocol of Aspnes [4]. Intuitively, in this protocol, the processes are racing around a track trying to complete laps for their preferred values. A process *completes* lap ℓ for input value v by writing (v, ℓ) to its single-writer register. For each value v , if ℓ is the largest lap that has been completed for value v , then v is on lap $\ell + 1$. After reading all the registers, a process always tries to complete the current lap for a leading value, using coin flips to break ties. If a process sees that some value is on a lap that is 2 ahead of any other value, then it outputs that value and terminates.

There are also many anonymous obstruction-free protocols for consensus using a single m -component atomic snapshot object. Specifically, Guerraoui and Ruppert gave a protocol using $m = 8n + 2$ components [35], Bowman gave a protocol using $m = 2n$ components [20], and Bouzid, Reynal, and Sutra gave a protocol using $m = n$ components [19]. These protocols are also based on the randomized wait-free protocol by Aspnes [4]. To obtain a protocol using registers, the m -component atomic snapshot object is replaced with Guerraoui and Ruppert's anonymous, non-blocking implementation of an m -component atomic snapshot object from m registers [35]. This implementation uses extra metadata fields of unbounded size, so the resulting anonymous protocols for consensus use registers of unbounded

size.

Herlihy [37] defined the *consensus number* of a shared object to be the maximum number of processes that can use copies of the object and registers to solve consensus. Consensus numbers have been widely used to compare the computation power of shared objects. The *consensus hierarchy* classifies objects by their consensus numbers. In the same paper, Herlihy proved that registers are at level 1 of hierarchy, while test-and-set objects and swap objects are at level 2 of the hierarchy. Aspnes et al. [5] introduced max-registers and showed that they are at level 1 of the hierarchy.

In [26], we classify objects by the number of instances of the object needed to solve obstruction-free n -valued consensus among n processes (without using registers). The results in Sections 4.5, 4.4, and 4.6 are from this paper. We also introduced b -buffer objects, for $b \geq 1$, and showed that their power increases in this classification as b increases. Mostefaoui, Perrin, and Raynal [49, 50] independently defined the same object, which they called a *b -sliding window register*. They showed that the object has consensus number b .

There is an anonymous protocol that solves binary consensus among $n \geq 3$ processes in an obstruction-free manner using an unbounded number of binary registers (each of which is set to 1 at most once) [35]. The protocol also works if the registers are replaced with test-and-set objects.

4.2 Buffer Lower Bound

Let Π be any deterministic protocol that solves binary consensus protocol among $n \geq 2$ processes, p_0, p_1, \dots, p_{n-1} , in an obstruction-free manner using only b -buffers, for some $b \geq 1$. In this section, we show that Π uses at least $\lceil (n-1)/b \rceil$ b -buffers. In particular, if $b = 1$, then Π uses at least $n-1$ registers. Fix an integer $b \geq 1$. Throughout this section, a b -buffer is called a buffer.

We introduce a stronger definition of bivalency, called *solo bivalency*. The standard definition of bivalency requires the existence of executions (involving any number of processes) in which different values are **output**. We require the existence of *solo* executions in which different values are **output**. More formally, we say that a process *decides* a value $v \in \{0, 1\}$ from a reachable configuration C of Π if the process **outputs** v in its solo-terminating execution from C . A pair of processes is *solo bivalent* from C if the processes decide different values from C .

In the following, we will strive to keep a fixed pair of processes, $\{p_0, p_1\}$, solo bivalent from all the configurations that we consider. As a first step, we prove that there exists a reachable configuration of Π from which $\{p_0, p_1\}$ is solo bivalent. This is a standard proposition that appears in every valency argument, but we include the proof here for completeness. Note that the proposition does not depend on the base objects in the system.

Proposition 4.1. *There exists a reachable configuration I of Π from which $\{p_0, p_1\}$ is solo bivalent.*

Proof. For $v \in \{0, 1\}$, let I_v be the reachable configuration in which every process is assigned input v by the scheduler, i.e. each process has taken its **input** step, which returned v as a response, and no other steps. By the specification of consensus, every process decides v from I_v . Consider a reachable configuration I in which p_0 is assigned input 0, p_1 is assigned input 1, and the rest of the processes are assigned any input in $\{0, 1\}$. Since no process has taken a shared memory step, for each $v \in \{0, 1\}$, I is indistinguishable from I_v to p_v , so it decides v from I . Thus, $\{p_0, p_1\}$ is solo bivalent from I . \square

A standard definition used in covering arguments for registers ($b = 1$) is the following: If a set of processes, P , is poised to write to a set of $|P|$ registers, V , in a configuration, C , then a *block write* by P to V is a sequence of steps consisting of the writes that the processes in P are poised to perform.

To generalize covering arguments to buffers, we generalize the notion of a block write. Suppose that there is a set of processes P and a set of buffers V , such that, for each buffer $r \in V$, there are exactly b processes in P that are about to perform a write to r . Then it is possible for the processes in P to perform their writes to V all at once. This obliterates the contents of all buffers in V . This is formalized in the next definition.

Let C be a configuration. A buffer is *covered* by a process p in C if p is about to perform write on the buffer. A set of processes P *fully covers* a set of buffers V in C if each buffer in V is covered by exactly b processes in P in C and $|P| = b|V|$. In this case, a *block write* by P to V is a sequence of steps, β , consisting of the writes that the processes in P are poised to perform. Moreover, we say that P is *poised* to perform β in configuration C .

Suppose a set of processes, P , is poised to perform a block write, β , to a set of registers, V , in a configuration C . If some process, $q \notin P$, decides a value from C and some other process decides a different value from $C\beta$, then q must write to a register that is not in V during its solo-terminating execution from C . Many of the existing lower bounds on the number of registers needed to solve obstruction-free consensus [28, 32, 55] contain a proof of this fact. The following proposition generalizes this to buffers.

Proposition 4.2. *Suppose C is a reachable configuration of Π in which a set of processes P fully covers a set of buffers V . Let β be a block write by P to V . If a process $q \notin P$ decides v from C and another process $p \neq q$ decides \bar{v} from $C\beta$, then q writes to a buffer not in V in its solo-terminating execution from C .*

Proof. Suppose not. Then every step in q 's solo-terminating execution, ζ , from C is either a read (from any buffer) or a write to a buffer in V . Let τ be the sequence of steps in p 's solo-terminating execution from $C\beta$, in which \bar{v} is decided. Since ζ contains only reads or writes to buffers in V , $C\zeta\beta$ is indistinguishable from $C\beta$ to p . Indeed, any value that q writes to buffers in V during ζ is obliterated by the block write β , while reads are never visible to other processes. It follows that τ is applicable at $C\zeta\beta$ and leaves p in the same state in $C\zeta\beta\tau$ as $C\beta\tau$. In particular, p has output \bar{v} in $C\zeta\beta\tau$. Therefore, $C\zeta\beta\tau$ is a reachable configuration of Π in which both 0 and 1 have been output, which contradicts the correctness of Π . \square

The next lemma combines covering and bivalency in a way that is different from the existing space lower bounds for consensus. The lemma says that, by running only $\{p_0, p_1\}$, we can keep $\{p_0, p_1\}$ solo bivalent after a block write β , even if β writes to many buffers and may convey a lot of information to p_0 and p_1 . As mentioned before, analogously to Fischer, Lynch, and Paterson's impossibility result, this is one of the main difficulties in the argument.

Lemma 4.3. *Suppose C is a reachable configuration of Π in which a set of processes, P , disjoint from $\{p_0, p_1\}$, fully covers a set of buffers, V . If $\{p_0, p_1\}$ is solo bivalent from C , then there is a $\{p_0, p_1\}$ -only execution σ from C such that $\{p_0, p_1\}$ is solo bivalent from $C\sigma\beta$, where β is a block write by P to V .*

Proof. If $\{p_0, p_1\}$ is solo bivalent from $C\beta$, then let σ be the empty execution. Now suppose p_0 and p_1 both decide the same value, say $v \in \{0, 1\}$, from $C\beta$. Since $\{p_0, p_1\}$ is solo bivalent from C , some process $p_i \in \{p_0, p_1\}$ decides \bar{v} from C . Let τ be the sequence of steps in p_i 's solo-terminating execution from

C , in which it outputs \bar{v} . Consider the longest prefix τ' of τ such that p_0 and p_1 both decide v from $C\tau'\beta$. Note that τ' is a proper prefix of τ since p_i outputs \bar{v} in τ . Let δ be the next step in τ after τ' and let $\sigma = \tau'\delta$. There are two cases to consider.

Case 1: δ is a read or δ is a write to a buffer in V . Note $C\tau'\delta\beta$ is indistinguishable from $C\tau'\beta$ to p_{1-i} . Indeed, if δ is a read, then no other process can tell that p_i performed δ . On the other hand, if δ is a write to a buffer in V , then the value written by δ is overwritten by β , so no other process can tell that p_i performed δ . Now, since p_{1-i} decides v from $C\tau'\beta$, which is indistinguishable from $C\tau'\delta\beta$ to p_{1-i} , it decides v from $C\tau'\delta\beta$ as well. By definition of τ' , p_i decides \bar{v} from $C\tau'\delta\beta$. Thus, $\{p_0, p_1\}$ is solo bivalent from $C\tau'\delta\beta = C\sigma\beta$.

Case 2: δ is a write to a buffer not in V . Since β is a block write to V , $C\tau'\beta\delta$ is indistinguishable from $C\tau'\delta\beta$ to every process. Now, since p_i decides v from $C\tau'\beta$ and δ is a step by p_i , it decides v from $C\tau'\beta\delta$. Since $C\tau'\beta\delta$ is indistinguishable from $C\tau'\delta\beta$ to p_i , it decides v from $C\tau'\delta\beta$ as well. By definition of τ' , p_{1-i} decides \bar{v} from $C\tau'\delta\beta$. Thus, $\{p_0, p_1\}$ is solo bivalent from $C\tau'\delta\beta = C\sigma\beta$. \square

Our main technical lemma says that, for each $1 \leq k \leq n-1$, we can find a configuration C such that $\{p_0, p_1\}$ is solo bivalent from C , each process in $\{p_2, \dots, p_k\}$ covers a buffer, and at most b processes in $\{p_2, \dots, p_k\}$ cover the same buffer in C . By the pigeonhole principle, this witnesses the fact that Π uses at least $\lceil (k-1)/b \rceil$ buffers.

Lemma 4.4. *Let $1 \leq k \leq n-1$ and let I be a reachable configuration of Π . If $\{p_0, p_1\}$ is solo bivalent from I , then there is a $\{p_0, p_1, \dots, p_k\}$ -only execution σ from I such that $\{p_0, p_1\}$ is solo bivalent from $I\sigma$, each process in $\{p_2, \dots, p_k\}$ covers a buffer in $I\sigma$, and at most b processes in $\{p_2, \dots, p_k\}$ cover the same buffer in $I\sigma$.*

Proof. By induction on k . The base case is when $k = 1$ and is satisfied by letting σ be the empty execution. Now suppose that, for some $1 \leq k < n-1$, whenever $\{p_0, p_1\}$ is solo bivalent from a reachable configuration C of Π , there is a $\{p_0, p_1, \dots, p_k\}$ -only execution σ from C such that $\{p_0, p_1\}$ is solo bivalent from $C\sigma$, every process in $\{p_2, \dots, p_k\}$ covers a buffer in $C\sigma$, and at most b processes in $\{p_2, \dots, p_k\}$ cover the same buffer in $C\sigma$.

We define a sequence of configurations C_0, C_1, C_2, \dots reachable from I by $\{p_0, p_1, \dots, p_k\}$ -only executions as follows. Let $C_0 = I$. For $i \geq 0$, let σ_i be the execution obtained by applying the induction hypothesis to C_i so that $\{p_0, p_1\}$ is solo bivalent from $C_i\sigma_i$, each process in $\{p_2, \dots, p_k\}$ covers a buffer, and at most b processes in $\{p_2, \dots, p_k\}$ cover the same buffer in $C_i\sigma_i$. Let V_i be the set of buffers that are covered by exactly b processes in $\{p_2, \dots, p_k\}$, let $P_i \subseteq \{p_2, \dots, p_k\}$ be the set of processes that fully cover V_i in $C_i\sigma_i$, and let β_i be a block write by P_i to V_i . By Lemma 4.3, there is a $\{p_0, p_1\}$ -only execution τ_i from $C_i\sigma_i$ such that $\{p_0, p_1\}$ is solo bivalent from $C_i\sigma_i\tau_i\beta_i$. Let $C_{i+1} = C_i\sigma_i\tau_i\beta_i$.

Since Π uses a finite number of buffers, by the pigeonhole principle, there exists $0 \leq i < j$ such that $V_i = V_j$. Let v be the value that p_{k+1} decides in its solo-terminating execution ζ from $C_i\sigma_i\tau_i$. By construction, $\{p_0, p_1\}$ is solo bivalent from $C_i\sigma_i\tau_i\beta_i$. Hence, there is a process $p \in \{p_0, p_1\}$ that decides \bar{v} from $C_i\sigma_i\tau_i\beta_i$. By Proposition 4.2 with $P = P_i$, $q = p_{k+1}$, and p, ζ contains a write to a buffer not in V_i . Let ζ' be the prefix of ζ up to, but not including, the first such write. In $C_i\sigma_i\tau_i\zeta'$, p_{k+1} covers a buffer not in V_i . Since ζ' only contains reads (from any buffer) and writes to buffers in V_i , after the block write β_i by P_i to V_i , $C_i\sigma_i\tau_i\zeta'\beta_i$ is indistinguishable from $C_i\sigma_i\tau_i\beta_i$ to $\{p_0, p_1, \dots, p_{n-1}\} - \{p_{k+1}\}$. Since $\sigma_{i+1}\tau_{i+1}\beta_{i+1} \cdots \sigma_{j-1}\tau_{j-1}\beta_{j-1}\sigma_j$ is a $\{p_0, p_1, \dots, p_k\}$ -only execution, its steps are applicable from $C_i\sigma_i\tau_i\zeta'\beta_i$ and leaves p_0, p_1, \dots, p_k in the same states as in $C_j\sigma_j$.

Let σ be the execution that consists of the steps in $\sigma_0\tau_0\beta_0 \cdots \sigma_i\tau_i$, followed by the steps in ζ' , followed by the steps in $\beta_i\sigma_{i+1}\tau_{i+1}\beta_{i+1} \cdots \sigma_{j-1}\tau_{j-1}\beta_{j-1}\sigma_j$. Then σ is a $\{p_0, p_1, \dots, p_{k+1}\}$ -only execution such that $\{p_0, p_1\}$ is solo bivalent from $I\sigma$, each process in $\{p_2, \dots, p_{k+1}\}$ covers a buffer in $I\sigma$, and at most b processes in $\{p_2, \dots, p_{k+1}\}$ cover the same buffer in $I\sigma$. Note that V_j is the set of buffers that are fully covered by P_j in $I\sigma$ and p_{k+1} covers a buffer not in $V_j = V_i$ in $I\sigma$ since it takes no steps in the suffix of σ after ζ' . \square

We can now prove our main theorem.

Theorem 4.5. *Any protocol that solves consensus among $n \geq 2$ processes in an obstruction-free manner using only b -buffer objects must use at least $\lceil (n-1)/b \rceil$ objects.*

Proof. By Proposition 4.1, there exists an initial configuration of Π from which $\{p_0, p_1\}$ is solo bivalent. Applying Lemma 4.4 with $k = n - 1$ to this initial configuration, there is a reachable configuration C from which $\{p_0, p_1\}$ is solo bivalent, every process in $P = \{p_2, \dots, p_{n-1}\}$ covers a buffer, and every buffer is covered by at most b processes in P in C . Let V be the set of buffers covered by P in C . Then $|V| \geq \lceil (n-2)/b \rceil$.

If $|V| > \lceil (n-1)/b \rceil - 1$, then Π uses at least $\lceil (n-1)/b \rceil$ buffers. Otherwise, $\lceil (n-1)/b \rceil - 1 \geq |V| \geq \lceil (n-2)/b \rceil$, so $|V| = \lceil (n-2)/b \rceil = \lceil (n-1)/b \rceil - 1$. Then $n-2$ is a multiple of b . Since $|P| = n-2$, $|P| = b|V|$ and, therefore, V is fully covered by P in C . Since $\{p_0, p_1\}$ is solo bivalent from C , Proposition 4.2 implies that there is a $\{p_0, p_1\}$ -only execution ξ from C such that some process in $\{p_0, p_1\}$ covers a buffer not in V in configuration $C\xi$. Therefore, Π uses at least $\lceil (n-2)/b \rceil + 1 = \lceil (n-1)/b \rceil$ buffers. \square

Since buffers are readable, by Theorem 3.1, we have the following corollary.

Corollary 4.6. *Any nondeterministic solo-terminating protocol that solves consensus among $n \geq 2$ processes using only b -buffer objects must use at least $\lceil (n-1)/b \rceil$ objects.*

4.3 Buffer Upper Bound

In this section, we show that, for all $b \geq 1$, it is possible to solve m -valued consensus among $n \geq 2$ processes in an obstruction-free manner using $k = \lceil n/b \rceil$ different b -buffer objects. The domain of each buffer object is $(\mathbb{N}^m)^b$. Initially, each buffer object contains $(\vec{0}, \dots, \vec{0})$, where $\vec{0} \in \mathbb{N}^m$ is the all-zero vector. At the end of this section, we will describe how to extend the protocol to handle arbitrary inputs.

To simplify the proof of correctness, we assume that the processes have access to a snapshot object, \mathbf{B} , consisting of the k buffer objects, $\mathbf{B}_1, \dots, \mathbf{B}_k$ (as described in Section 2.6). Notice that a scan returns a vector with k components, one for each buffer object. Each of these components contains a vector of b components, each of which in turn contain a vector of m natural numbers. It is helpful to denote the result of a scan operation by the vector $(\vec{s}(i, j) : (i, j) \in [k] \times [b]) \in (\mathbb{N}^m)^{k \times b}$. We call this the *vector returned by the scan*. For each $(i, j) \in [k] \times [b]$, $\vec{s}(i, j) \in \mathbb{N}^m$ is the vector stored in the j 'th component of buffer \mathbf{B}_i . For $v \in \{0, \dots, m-1\}$, we will denote the v 'th component of $\vec{s}(i, j)$ by $\vec{s}(i, j)_v$.

Our protocol is very similar to the randomized wait-free consensus by Aspnes [4]. The main difference is that processes are anonymous. Hence, a process cannot simply write to one location to complete a lap for a value, as its write may be overwritten by another process. Another difference is that we use

```

1:  $v^* \leftarrow \text{input}(), \vec{\ell} \leftarrow \vec{0}$ 
2: loop
3:    $(\vec{s}(i, j) : (i, j) \in [k] \times [b]) \leftarrow \text{scan}(\mathbf{B}_1, \dots, \mathbf{B}_k)$ 
4:   for  $v = 0, 1, \dots, m-1$  do
5:      $\vec{\ell}_v \leftarrow \max\{\vec{s}(i, j)_v : (i, j) \in [k] \times [b]\}$ 
6:     if  $\vec{\ell}_v > \vec{\ell}_{v^*}$  then
7:        $v^* \leftarrow v$ 
8:     if  $\vec{s}(i, j) = \vec{\ell}$  for all  $(i, j) \in [k] \times [b]$  then
9:       if  $\vec{\ell}_{v^*} \geq \vec{\ell}_v + 2$  for all  $v \neq v^*$  then
10:        output( $v^*$ ) and terminate
11:        $\vec{\ell}_{v^*} \leftarrow \vec{\ell}_{v^*} + 1$ 
12:        $i \leftarrow \min\{i \in [k] : \exists j \in [b] \text{ such that } \vec{s}(i, j) \neq \vec{\ell}\}$ 
13:        $\mathbf{B}_i.\text{write}(\vec{\ell})$ 

```

Algorithm 4.1: An anonymous protocol that solves m -valued consensus among $n \geq 2$ processes in an obstruction-free manner using a snapshot object consisting of the b -buffer objects, $\mathbf{B}_1, \dots, \mathbf{B}_k$, where $k = \lceil n/b \rceil$. The domain of each \mathbf{B}_j is $(\mathbb{N}^m)^b$. Initially, $\mathbf{B}_j = (\vec{0}, \dots, \vec{0})$. Code for each process.

buffer objects instead of registers. Finally, when a process tries to complete the next lap for a leading value, it breaks ties in a deterministic manner, in favour of its preferred value.

We now define some useful terminology. For every **scan** S by a process p that returns a vector $(\vec{s}(i, j) : (i, j) \in [k] \times [b])$, let $\vec{\ell}(S)$ denote the m -component vector whose v 'th component, $\vec{\ell}_v(S)$, is $\max\{\vec{s}(i, j)_v : (i, j) \in [k] \times [b]\}$. We say that $\ell_v(S)$ is the *lap that p thinks value v is on after S* . For every **write** U to some buffer object, let $\vec{\ell}(U)$ denote the argument of the **write** operation. Observe that the definitions of $\vec{\ell}(S)$ and $\vec{\ell}(U)$ do not depend on the process that performed the operation.

At any moment in time, a process is trying to complete a lap for its *preferred* value, $v^* \in \{0, \dots, m-1\}$. Initially, a process prefers its input value. After performing a **scan**, the process chooses its preferred value among the ones that it thinks have the largest number of completed laps. To *complete a lap* for its preferred value, v^* , the process tries to fill the contents of all buffer objects with a vector consisting of the number of laps it thinks each value is on. (If it has performed no **scans**, then it thinks every value is on lap 0.) Once it sees that all components of the result of a **scan** contain the same value, it increments the lap that it thinks (its possibly new preferred value) v^* is on. It then writes this to buffer \mathbf{B}_1 (along with the laps that it thinks the other values are on). Once it thinks that v^* is on a lap that is at least 2 ahead of any other value $v \neq v^*$, the process **outputs** v^* and **terminates**.

More formally, each process has a local variable, $\vec{\ell} \in \mathbb{N}^m$, storing, for each value v , the number of laps $\vec{\ell}_v \geq 0$ that the process thinks v is on. Initially, $\vec{\ell} = \vec{0}$. Each process also has a local variable, v^* , for its preferred value. It initializes v^* to the input that it is assigned. A process begins by performing a **scan**, S , which returns $(\vec{s}(i, j) : (i, j) \in [k] \times [b])$. Then, for each value v , it updates its view, $\vec{\ell}_v$, of v 's current lap to be $\vec{\ell}_v(S)$. It also updates its preferred value, v^* , to v if it sees that $\vec{\ell}_v > \vec{\ell}_{v^*}$. If some buffer object contains a vector other than $\vec{\ell}$, i.e. $\vec{s}(i, j) \neq \vec{\ell}$ for some $(i, j) \in [k] \times [b]$, then the process performs **write**($\vec{\ell}$) on the first such buffer object. Now suppose the buffer objects all contain b copies of $\vec{\ell}$. If the preferred value, v^* , of the process is at least 2 laps ahead of any other value $v \neq v^*$, then the process **outputs** v^* and **terminates**. Otherwise, it considers v^* to have completed lap $\vec{\ell}_{v^*}$ and it increments $\vec{\ell}_{v^*}$. Then it performs **write**($\vec{\ell}$) on \mathbf{B}_1 . If the process doesn't terminate, then it repeats this sequence of steps. Algorithm 4.1 contains the pseudocode for each process.

We now proceed with the formal proof of correctness. Fix an execution of the protocol. We begin

with an easy observation, which follows by inspection of the code.

Observation 4.7. *Let U be a write by a process p and let S be the last scan by p before U .*

(a) *For each value $v \in \{0, \dots, m-1\}$, $\vec{\ell}_v(U) \geq \vec{\ell}_v(S)$.*

(b) *If there is a value $v^* \in \{0, \dots, m-1\}$ such that $\vec{\ell}_{v^*}(U) > \vec{\ell}_{v^*}(S)$, then $\vec{\ell}_{v^*}(U) = \vec{\ell}_{v^*}(S) + 1$, $\vec{\ell}_{v^*}(S) \geq \vec{\ell}_v(S) = \vec{\ell}_v(U)$ for all other values $v \neq v^*$, and S returned a vector whose components all contain $\vec{\ell}(S)$.*

The next lemma follows from Observation 4.7. It says that if a process thinks value v is on lap $\ell > 0$ after scan S , i.e. $\vec{\ell}_v(S) = \ell$, then was an earlier scan, which returned a vector whose components are all the same, and some process thought v was on lap $\ell - 1$ after this scan.

Lemma 4.8. *Let S be any scan and let $v \in \{0, \dots, m-1\}$ be a value. If $\vec{\ell}_v(S) > 0$, then there was a scan, S' , performed prior to S such that S' returned a vector whose components all contain $\vec{\ell}(S')$, where $\vec{\ell}_v(S') = \vec{\ell}_v(S) - 1$ and $\vec{\ell}_{v'}(S') \leq \vec{\ell}_v(S')$, for all values $v' \neq v$.*

Proof. Since each buffer object initially contains $(\vec{0}, \dots, \vec{0})$, and $\vec{\ell}_v(S) > 0$, there was a write U prior to S such that $\vec{\ell}_v(U) = \vec{\ell}_v(S)$. Consider the first such write. Let p be the process that performed U and let S' be the last scan performed by p before U . By Observation 4.7(a), $\vec{\ell}_v(U) \geq \vec{\ell}_v(S')$. If $\vec{\ell}_v(U) = \vec{\ell}_v(S')$, there would have been a write U' prior to S' and, hence, prior to U , with $\vec{\ell}_v(U') = \vec{\ell}_v(S') = \vec{\ell}_v(U) = \vec{\ell}_v(S)$, contradicting the definition of U . Hence $\vec{\ell}_v(U) > \vec{\ell}_v(S')$. By Observation 4.7(b), it follows that $\vec{\ell}_v(U) = \vec{\ell}_v(S') + 1$, $\vec{\ell}_{v'}(S') \leq \vec{\ell}_v(S')$ for all other values $v' \neq v$, and S' returned a vector whose components all contain $\vec{\ell}(S')$. Since $\vec{\ell}_v(U) = \vec{\ell}_v(S)$, it follows that $\vec{\ell}_v(S') = \vec{\ell}_v(U) - 1 = \vec{\ell}_v(S) - 1$. \square

The following lemma is key to the proof of correctness. It considers the situation after a process has performed a scan S that returned a vector whose components all contain $\vec{\ell}(S)$. The lemma says that each process will think that v is on lap at least $\vec{\ell}_v(S)$ after performing any scan after S , for every value v .

Lemma 4.9. *Suppose S is a scan that returned a vector whose components all contain $\vec{\ell}(S)$. If T is a scan performed after S , then, for each value $v \in \{0, \dots, m-1\}$, $\vec{\ell}_v(T) \geq \vec{\ell}_v(S)$.*

Proof. Suppose, for a contradiction, that there is a scan T performed after S such that $\vec{\ell}_v(T) < \vec{\ell}_v(S)$ for some value $v \in \{0, \dots, m-1\}$. Consider the first such scan T . By definition of $\vec{\ell}_v(T)$, T returned a vector $(\vec{t}(i, j) : (i, j) \in [k] \times [b])$ such that $\vec{t}(i, j)_v < \vec{\ell}_v(S)$, for each $(i, j) \in [k] \times [b]$. Since S returned a vector whose components all contain $\vec{\ell}(S)$, it follows that, for each $(i, j) \in [k] \times [b]$, there was some write $U_{i,j}$ performed between S and T such that $\vec{t}(i, j) = \vec{\ell}(U_{i,j})$. Since processes alternately perform scan and write, at most $n-1$ writes among $\{U_{i,j} : (i, j) \in [k] \times [b]\}$ are by processes whose last scan occurred before S . It follows that at least one $U_{i,j}$ was by some process whose last scan, S' , prior to $U_{i,j}$ occurred no earlier than S . Note that, by definition of T , $\vec{\ell}_v(S') \geq \vec{\ell}_v(S)$ when $S' \neq S$. By Observation 4.7(a), $\vec{\ell}_v(U_{i,j}) \geq \vec{\ell}_v(S')$. Therefore, $\vec{t}(i, j)_v = \vec{\ell}_v(U_{i,j}) \geq \vec{\ell}_v(S)$. This is a contradiction. \square

The previous lemma allows us to prove that, once a process thinks that value v^* is on lap ℓ , which is 2 laps ahead of every other value, and every component contains this information, then no process will ever think that another value is on lap at least ℓ , i.e. v^* will always be at least one lap ahead of any other value.

Lemma 4.10. *Suppose S is a scan that returned a vector whose components all contain $\vec{\ell}(S)$ and there is some value $v^* \in \{0, \dots, m-1\}$ such that $\vec{\ell}_{v^*}(S) \geq \vec{\ell}_v(S) + 2$ for all other values $v \neq v^*$. Then, for every scan T and every value $v \neq v^*$, $\vec{\ell}_v(T) < \vec{\ell}_v(S) + 2$.*

Proof. Suppose, for a contradiction, that there is some scan T and some value $v \neq v^*$ such that $\vec{\ell}_v(T) \geq \vec{\ell}_v(S) + 2 > 0$. Consider the first such scan T . By Lemma 4.8, there was a scan, T' , performed prior to T such that T' returned a vector whose components all contained $\vec{\ell}(T')$, where $\vec{\ell}_v(T') = \vec{\ell}_v(T) - 1$ and $\vec{\ell}_{v^*}(T') \leq \vec{\ell}_{v^*}(T)$. By definition of T and T' , $\vec{\ell}_v(T') < \vec{\ell}_v(S) + 2 \leq \vec{\ell}_v(T) = \vec{\ell}_v(T') + 1$, so $\vec{\ell}_v(S) + 2 = \vec{\ell}_v(T) = \vec{\ell}_v(T') + 1$. If S was performed after T' , then, by Lemma 4.9, $\vec{\ell}_v(S) \geq \vec{\ell}_v(T')$, which is not the case. Thus T' was performed after S . Hence, Lemma 4.9 implies that $\vec{\ell}_{v^*}(T') \geq \vec{\ell}_{v^*}(S)$. By assumption, $\vec{\ell}_{v^*}(S) \geq \vec{\ell}_v(S) + 2$. Hence, $\vec{\ell}_{v^*}(T') \geq \vec{\ell}_v(S) + 2 = \vec{\ell}_v(T') + 1$. This contradicts the fact that $\vec{\ell}_{v^*}(T') \leq \vec{\ell}_{v^*}(T')$. \square

We can now prove that the protocol is correct and obstruction-free.

Lemma 4.11. *No two processes output different values.*

Proof. The last step performed by a process before it outputs a value v^* is a scan, S , such that S returned a vector whose components all contain $\vec{\ell}(S)$ and $\vec{\ell}_{v^*}(S) \geq \vec{\ell}_v(S) + 2$ for all other values $v \neq v^*$. Consider the first such scan S by any process. By Lemma 4.9, $\vec{\ell}_{v^*}(T) \geq \vec{\ell}_{v^*}(S)$ for every scan T performed after S . By Lemma 4.10, $\vec{\ell}_v(T) \leq \vec{\ell}_v(S) + 1$ for all $v \neq v^*$. Hence, $\vec{\ell}_{v^*}(T) > \vec{\ell}_v(T)$. It follows that no process ever decides $v \neq v^*$. \square

Lemma 4.12. *If some process outputs x , then some process has input x .*

Proof. Suppose that some process outputs x . Then there was a write, U , such that $\vec{\ell}_x(U) > 0$. Consider the first such write U . Let p be the process that performed U and let S be the last scan performed by p before U . By definition of U , $\vec{\ell}_x(S) = 0$. By Observation 4.7(b), $\vec{\ell}_x(S) \geq \vec{\ell}_v(S)$, for all values $v \neq x$. It follows that $\vec{\ell}(S) = \vec{0}$. By construction, p initialized its preferred value, v^* , to its input and $\vec{\ell} = \vec{0}$ on line 1. Since $\vec{\ell}(S) = \vec{0}$, when p runs lines 4–7 immediately after S , it does not change v^* . Therefore, since p sets $\vec{\ell}_x = 1$, it must be that $v^* = x$ and x is the input of process p . \square

Lemma 4.13. *Every process outputs a value and terminates after performing at most $3kb + 1$ scans in a solo execution.*

Proof. Let p be any process and consider the first scan S performed by p in its solo execution. Let $\vec{\ell} = \vec{\ell}(S)$. Consider the value v^* stored by p at the end of the loop on lines 4–7 immediately after S . Notice that v^* is not changed during the rest of p 's solo execution.

After performing $\text{write}(\vec{\ell})$ at most kb times, p will perform a scan that returns a vector whose components all contain $\vec{\ell}$. If $\vec{\ell}_{v^*} \geq \vec{\ell}_v + 2$ for all $v \neq v^*$, then p outputs v^* immediately after this scan. Otherwise, p performs $\text{write}(\vec{\ell}')$ kb more times, where $\vec{\ell}'_{v^*} = \vec{\ell}_{v^*} + 1$ and $\vec{\ell}'_v = \vec{\ell}_v$, for all values $v \neq v^*$. Then it performs a scan that returns a vector whose components all contain $\vec{\ell}'$. If $\vec{\ell}'_{v^*} \geq \vec{\ell}'_v + 2$ for all $v \neq v^*$, then p outputs v^* immediately after this scan. If not, then p performs $\text{write}(\vec{\ell}'')$ kb more times, where $\vec{\ell}''_{v^*} = \vec{\ell}'_{v^*} + 1 = \vec{\ell}_{v^*} + 2$ and $\vec{\ell}''_v = \vec{\ell}'_v = \vec{\ell}_v$ for all values $v \neq v^*$. Finally, p performs a scan that returns a vector whose components all contain $\vec{\ell}''$ and outputs v^* immediately afterwards. Since p performs at most $3kb$ writes and each write is immediately followed by a scan, this amounts to at most $3kb + 1$ scans, including the first scan, S . \square

The preceding lemmas immediately yield the following theorem.

Theorem 4.14. *For $b \geq 1$, there is an anonymous protocol that solves m -valued consensus among $n \geq 2$ processes in an obstruction-free manner using a snapshot object consisting of $\lceil n/b \rceil$ different b -buffer objects.*

To support arbitrary input values, we note that, even if the domain of the input values is infinite, a process needs to record information about the laps of at most n different values. Hence, we can represent $\vec{\ell}$ as a set of pairs (v, ℓ) , where v is a value and $\ell > 0$ is the lap that the process thinks v is on. If a value v does not appear in $\vec{\ell}$, then the process thinks that v is on lap 0. With this representation, we can iterate through the vectors $\vec{s}(i, j)$ to update $\vec{\ell}$ and v^* instead of iterating over values (on lines 4–7).

Consequently, by Theorem 2.1, we obtain the following corollary.

Corollary 4.15. *For $b \geq 1$, there is an anonymous protocol that solves consensus among $n \geq 2$ processes in an obstruction-free manner using $\lceil n/b \rceil$ different b -buffer objects.*

4.4 Test-and-Set Objects

Let Π be any deterministic protocol that solves binary consensus among $n \geq 3$ processes in an obstruction-free manner using only test-and-set objects. We show that Π uses an unbounded number of test-and-set objects. There is a matching upper bound due to Guerraoui and Ruppert [35].

Fix 3 processes, p_0, p_1, p_2 . By Proposition 4.1, there is a reachable configuration of Π from which $\{p_0, p_1\}$ are solo bivalent.

Ideally, we would like to prove something similar to Lemma 4.3 for Π , e.g., if $\{p_0, p_1\}$ is solo bivalent from a reachable configuration C and p_2 is poised to perform a test-and-set operation β in C , then there is a $\{p_0, p_1\}$ -only execution σ such that $\{p_0, p_1\}$ is solo bivalent from $C\sigma\beta$. Instead, we show that there is a $\{p_0, p_1\}$ -only execution σ such that $\{p_0, p_1\}$ is *bivalent* from $C\sigma\beta$.

Lemma 4.16. *If $\{p_0, p_1\}$ is solo bivalent from a reachable configuration C of Π , then there is a $\{p_0, p_1\}$ -only execution σ from C such that $\{p_0, p_1\}$ is bivalent from $C\sigma\beta$, where β is the next step of p_2 in C .*

Proof. Since $\{p_0, p_1\}$ is solo bivalent from C , p_0 and p_1 decide different values in their solo-terminating executions, γ_0 and γ_1 , from C . Without loss of generality, assume that p_0 decides 0 and p_1 decides 1. If β is a read or a test-and-set applied to a test-and-object that has value 1 in C , then $C\beta$ is indistinguishable from C to $\{p_0, p_1\}$ and we may let σ be the empty execution. Now suppose β is a test-and-set operation applied to a test-and-set object X that has value 0 in C . If $\{p_0, p_1\}$ is bivalent from $C\beta$, then let σ be the empty execution. So, without loss of generality, suppose that $\{p_0, p_1\}$ is 0-univalent from $C\beta$.

Let σ be the longest prefix of γ_1 such that p_0 decides 0 from $C\sigma\beta$. Note that $\sigma \neq \gamma_1$ since 1 is decided in γ_1 . Consider the next step, δ , after σ in γ_1 . If δ is a read or a test-and-set applied to X , then $C\sigma\delta\beta$ is indistinguishable from $C\sigma\beta$ to p_0 . This is impossible, since p_0 decides 0 from $C\sigma\beta$ and decides 1 from $C\sigma\delta\beta$. Thus, δ is a test-and-set applied to a test-and-set object other than X . It follows that $C\sigma\delta\beta = C\sigma\beta\delta$ and, hence, p_0 decides 1 from $C\sigma\beta\delta$. Since p_0 decides 0 from $C\sigma\beta$ and δ is a step by p_1 , it follows that $\{p_0, p_1\}$ is bivalent from $C\sigma\beta$. \square

The next lemma says that if $\{p_0, p_1\}$ is bivalent from a configuration C , then it is possible to reach a configuration C' by a $\{p_0, p_1\}$ -only execution from C such that $\{p_0, p_1\}$ is solo bivalent from C' . Observe

that the proof does not use any properties of the system and, hence, the lemma is applicable to any system.

Lemma 4.17. *Suppose $\{p_0, p_1\}$ is bivalent from a reachable configuration C of Π . Then there is a $\{p_0, p_1\}$ -only execution σ from C such that $\{p_0, p_1\}$ is solo bivalent from $C\sigma$.*

Proof. Suppose, for a contradiction, that, for every $\{p_0, p_1\}$ -only execution σ from C , p_0 and p_1 output the same value in their solo-terminating executions from $C\sigma$. In particular, p_0 and p_1 output the same value, v , in their solo-terminating executions from C . Since $\{p_0, p_1\}$ is bivalent from C , there is a $\{p_0, p_1\}$ -only execution α from C in which \bar{v} is output. Consider the longest prefix α' of α such that some (and, hence, every) process in $\{p_0, p_1\}$ outputs v in its solo-terminating execution from $C\alpha'$. Notice that $\alpha' \neq \alpha$ since \bar{v} is output in α . Consider the next step, δ , after α' in α . Since α is a $\{p_0, p_1\}$ -only execution, δ is by process p_i , for some $i \in \{0, 1\}$. By definition of α' , every process in $\{p_0, p_1\}$ decides \bar{v} in its solo-terminating execution from $C\alpha'\delta$. However, since p_i outputs v in its solo-terminating execution from $C\alpha'$, it decides v in its solo-terminating execution from $C\alpha'\delta$. This is a contradiction. \square

We now combine the preceding lemmas to prove our main technical lemma.

Lemma 4.18. *Let C_0 be a reachable configuration of Π . If $\{p_0, p_1\}$ is solo bivalent from C_0 , then, for every $k \geq 0$, it is possible to reach a configuration C_k by a $\{p_0, p_1, p_2\}$ -only execution from C_0 such that $\{p_0, p_1\}$ is solo bivalent from C_k and at least k test-and-set objects have been set to 1 in C_k .*

Proof. By induction on k . The base case, $k = 0$, holds trivially for C_0 . Given C_k , for some $k \geq 0$, we show how to reach C_{k+1} . Let L be the set of test-and-set objects that have been set to 1 in C_k . By the induction hypothesis, $|L| \geq k$. Let ζ be p_2 's solo-terminating execution from C_k , in which it decides some value $v \in \{0, 1\}$. Let ζ' be the longest prefix of ζ such that $C_k\zeta'$ is indistinguishable from C_k to $\{p_0, p_1\}$. Note that $\zeta' \neq \zeta$ since p_2 outputs v in ζ and $\{p_0, p_1\}$ is solo bivalent from $C_k\zeta'$. Consider the next step, β , after ζ' in ζ . If β is a read or a test-and-set applied to an object in L , then $C_k\zeta'$ and $C_k\zeta'\beta$ are indistinguishable to $\{p_0, p_1\}$, which is impossible. Hence, β is a test-and-set to an object $X \notin L$. Since $C_k\zeta'$ is indistinguishable from C_k to $\{p_0, p_1\}$, $\{p_0, p_1\}$ is solo bivalent from $C_k\zeta'$. By Lemma 4.16, there is a $\{p_0, p_1\}$ -only execution α_1 from $C_k\zeta'$ such that $\{p_0, p_1\}$ is bivalent from $C_k\zeta'\alpha_1\beta$. By Lemma 4.17, there is a $\{p_0, p_1\}$ -only execution α_2 from $C_k\zeta'\alpha_1\beta$ such that $\{p_0, p_1\}$ is solo bivalent from $C_{k+1} = C_k\zeta'\alpha_1\beta\alpha_2$. Note that every object in $L \cup \{X\}$ has been set to 1 in C_{k+1} and $|L \cup \{X\}| \geq k + 1$. \square

By Proposition 4.1, there is a reachable configuration of Π from which $\{p_0, p_1\}$ is solo bivalent. It follows from Lemma 4.18 that Π uses an unbounded number of test-and-set objects. Since Π was arbitrary, by Theorem 3.1, we have proved the following theorem.

Theorem 4.19. *Any nondeterministic solo-terminating protocol solving binary consensus among $n \geq 3$ processes using only test-and-set objects must use an unbounded number of objects.*

4.5 Max-Registers

In this section, we show that two max-registers are necessary for solving binary consensus in an obstruction-free manner. We also show that two max-registers are sufficient for solving m -valued consensus in an obstruction-free manner, with anonymous processes.

Theorem 4.20. *Any nondeterministic solo-terminating protocol solving binary consensus in an obstruction-free manner among $n \geq 2$ processes using only max-registers must use at least 2 max-registers.*

Proof. Suppose there is a nondeterministic solo-terminating protocol solving binary consensus using one max-register. Let C be an initial configuration where processes p and q have inputs 0 and 1, respectively. Consider a solo-terminating execution α of p from C and a solo-terminating execution β of q from C . We show how to interleave these two executions so that the resulting execution is indistinguishable to both processes from their respective solo executions. Hence, both values will be returned, contradicting agreement.

To build the interleaved execution, run both processes until they are first poised to perform `write-max`. Suppose p is poised to perform `write-max(a)` and q is poised to perform `write-max(b)`. If $a \leq b$, let p take steps until it is next poised to perform `write-max(a')`, with $a' > b$, or until the end of α , if it performs no such `write-max` operations. Otherwise, let q take steps until it is next poised to perform `write-max(b')`, with $b' > a$, or until the end of β . Repeat this until one of the processes reaches the end of its execution and then let the other process finish. \square

We now describe a protocol for m -valued consensus using two max-registers, M_1 and M_2 . Consider the lexicographic ordering \prec on the set $S = \mathbb{N} \times \{0, 1, \dots, m-1\}$. Let y be a fixed prime that is larger than m . Note that, for $(r, x), (r', x') \in S$, $(r, x) \prec (r', x')$ if and only if $(x+1)y^r < (x'+1)y^{r'}$. Thus, by identifying $(r, x) \in S$ with $(x+1)y^r$, we may assume that M_1 and M_2 are max-registers defined on S with respect to the lexicographic ordering \prec . Initially, both M_1 and M_2 have value $(0, 0)$. We also assume that the processes have access to a snapshot object, M , consisting of the two max-registers, M_1 and M_2 (as described in Section 2.6).

Each process alternately performs `write-max` on one component and takes a `scan` of both components. It begins by performing `write-max(0, x')` on M_1 , where $x' \in \{0, \dots, m-1\}$ is its input value. If its `scan` returns $[(r+1, x), (r, x)]$, then it `outputs` x and `terminates`. If its `scan` returns $[(r, x), (r, x)]$, then it performs `write-max(r+1, x)` on M_1 . Otherwise, its `scan` returns $[v, v']$ and it performs `write-max(v)` on M_2 . Algorithm 4.2 contains the pseudocode for each process.

```

M1.write-max(0, input())
loop
  [(r, x), (r', x')] ← scan(M1, M2)
  if r = r' + 1 and x = x' then
    output(x) and terminate
  else if r = r' and x = x' then
    M1.write-max(r + 1, x)
  else
    M2.write-max(r, x)

```

Algorithm 4.2: An anonymous protocol that solves m -valued consensus among $n \geq 2$ processes in an obstruction-free manner using a snapshot object consisting of 2 max-registers, M_1 and M_2 . The domain of each M_j is $\mathbb{N} \times \{0, 1, \dots, m-1\}$, under the lexicographic ordering. Code for each process.

As in Section 4.3, our protocol is similar to the randomized wait-free consensus protocol of Aspnes [4]. In that protocol, the crucial information that a process needs to know is a leading value, x , (so that it may adopt the value), the lap that x is on, and whether any other value is at most 1 lap behind. With max-registers, it is easy to determine this information.

In our protocol, a process performs $\text{write-max}(r, x)$ to tell the other processes it thinks value x is on lap r . Since a process performs $\text{write-max}(r, x)$ on M_2 only if it sees (r, x) in M_1 , at all times, $M_1 \succeq M_2$. Hence, M_1 always contains a pair (r, x) such that x is a leading value and r is the lap that x is on. This is why the processes can always adopt the value it sees in M_1 . Since a process performs $\text{write-max}(r+1, x)$ on M_1 only if it sees (r, x) in M_2 , at all times, M_2 contains information about a value that is at most 1 lap behind the value in M_1 .

If a process sees $[(r, x), (r, x)]$ as the result of a `scan`, then it thinks that x has completed lap r and is about to start lap $r+1$. Now suppose a process sees $[(r+1, x), (r, x)]$ as the result of a `scan`. Since another process performs $\text{write-max}(r'+1, x')$ on M_1 only if it sees $[(r', x'), (r', x')]$ as the result of its last `scan`, no other process is poised to perform $\text{write-max}(r'+1, x')$ on M_1 for $(r', x') \succ (r, x)$. It follows that, when any other process next performs a `scan`, x is still the leading value that they see.

We now prove that the protocol is correct and obstruction-free.

Lemma 4.21. *If some process outputs x , then some process has input x .*

Proof. If some process outputs x , then it saw $[(r+1, x), (r, x)]$ in its last `scan`, for some value r . Consider the smallest r' such that some process performed $M_1.\text{write-max}(r', x)$. Since a process performs $M_1.\text{write-max}(r', x)$, for $r' > 0$, only if it sees $(r'-1, x)$ in M_1 and M_2 as a result of a `scan`, it follows that $r' = 0$. Note that a process performs $M_1.\text{write-max}(0, x)$ only if it has input x . \square

Lemma 4.22. *No two processes output different values.*

Proof. To obtain a contradiction, suppose that there is an execution in which some process p outputs value x and another process q outputs value $x' \neq x$. Immediately before its output step, p performed a `scan` S_p where M_1 had value $(r+1, x)$ and M_2 had value (r, x) , for some $r \in \mathbb{N}$. Similarly, immediately before its output step, q performed a `scan` where M_1 had value $(r'+1, x')$ and M_2 had value (r', x') , for some $r' \in \mathbb{N}$. Without loss of generality, we may assume that p 's `scan` occurs before q 's `scan`. In particular, M_2 had value (r, x) before it had value (r', x') . So, from the specification of a max-register, $(r, x) \preceq (r', x')$. Since $x' \neq x$, it follows that $(r, x) \prec (r', x')$.

We show inductively, for $j = r', \dots, 0$, that some process performed a `scan` in which both M_1 and M_2 had value (j, x') . By assumption, q performed a `scan` where M_1 had value $(r'+1, x')$. So, some process performed $\text{write-max}(r'+1, x')$ on M_1 . From the algorithm, this process performed a `scan` where M_1 and M_2 both had value (r', x') . Now suppose that $0 < j \leq r'$ and some process performed a `scan` in which both M_1 and M_2 had value (j, x') . So, some process performed $\text{write-max}(j, x')$ on M_1 . From the algorithm, this process performed a `scan` where M_1 and M_2 both had value $(j-1, x')$.

Consider the smallest value of j such that $(r, x) \prec (j, x')$. Note that $(r, x) \prec (r', x')$, so $j \leq r'$. Hence, some process performed a `scan` S in which both M_1 and M_2 had value (j, x') . Since $(r, x) \prec (j, x')$, S occurred after the `scan` S_p by p , in which M_2 had value (r, x) . But M_1 had value (j, x') in S and M_1 had value $(r+1, x)$ in S_p , so $(r+1, x) \preceq (j, x')$. Since $x \neq x'$, it follows that $(r+1, x) \prec (j, x')$. Hence $j \geq 1$ and $(r, x) \prec (j-1, x')$. This contradicts the choice of j . \square

Lemma 4.23. *Each process terminates after performing at most 3 scans in a solo execution.*

Proof. Consider the first `scan` performed by a process in its solo execution. Suppose the `scan` returns $[(r, x), (r', x')]$. If $r = r'+1$ and $x = x'$, then the process immediately terminates. If $r = r'$ and $x = x'$, then the process performs $M_1.\text{write-max}(r+1, x)$ and terminates immediately after its next `scan`. Otherwise, the process performs $M_2.\text{write-max}(r, x)$ and terminates after 2 more scans. \square

The preceding lemmas immediately imply the following theorem.

Theorem 4.24. *There is an anonymous protocol that solves m -valued consensus among $n \geq 2$ processes in an obstruction-free manner using only two max-registers.*

4.6 Swap Objects

In this section, we show that it is possible to solve m -valued consensus among $n \geq 2$ anonymous processes in an obstruction-free manner using $n-1$ swap objects, S_1, \dots, S_{n-1} . Each swap object stores a vector in \mathbb{N}^m , which is initially $\vec{0}$. The main idea is that, because a **swap** operation returns the previous contents of the object, we can use essentially the same protocol as in Section 4.3 (when $b = 1$) and save one register by having the processes take into account the information they gain when they perform a **swap**. At the end of the section, we will show how to extend the protocol to handle arbitrary inputs. By Section 2.6, we may assume that processes have access to a snapshot object consisting of S_1, \dots, S_{m-1} .

As in Section 4.3, each process has a local variable, $\vec{\ell} \in \mathbb{N}^m$, storing, for each value $v \in \{0, \dots, m-1\}$, the lap $\vec{\ell}_v \geq 0$ that the process thinks v is on after a **scan**. Initially, $\vec{\ell} = \vec{0}$. Each process also has a local variable, v^* , for its preferred value. It initializes v^* to the input that it is assigned. In addition, each process has a local variable, \vec{e} , containing the vector returned by the last **swap** performed by the process. Initially, $\vec{e} = \vec{0}$.

For each **scan**, S , by a process p , we define $\vec{\ell}(S)$ to be the m -component vector such that, for each value v , $\vec{\ell}_v(S)$ is the v 'th component, $\vec{\ell}_v$, of p 's local variable $\vec{\ell}$ computed on line 5 immediately after S . For every **swap** U to some swap object, let $\vec{\ell}(U)$ denote the argument of the **swap** operation.

A process begins by performing **scan**(S_1, \dots, S_{n-1}), which returns a vector ($\vec{s}(j) : j \in [n-1]$), where $\vec{s}(j) \in \mathbb{N}^m$ contains the vector stored in S_j , for each $j \in [n-1]$. Then, for each value v , it updates its view, $\vec{\ell}_v$, of v 's current lap to be the maximum among $\{\vec{\ell}_v, \vec{e}_v\} \cup \{\vec{s}(j)_v : j \in [n-1]\}$. It also updates its preferred value, v^* , to v if it sees that $\vec{\ell}_v > \vec{\ell}_{v^*}$. If some swap object contains a vector other than $\vec{\ell}$, i.e. $\vec{s}(j) \neq \vec{\ell}$ for some $j \in [n-1]$, then the process performs **swap**($\vec{\ell}$) on the first such swap object. Now suppose the swap objects all contain $\vec{\ell}$. If the preferred value, v^* , of the process is at least 2 laps ahead of any other value $v \neq v^*$, then the process outputs v^* and terminates. Otherwise, it considers v^* to have completed lap $\vec{\ell}_{v^*}$ and it increments $\vec{\ell}_{v^*}$. Then it performs **swap**($\vec{\ell}$) on S_1 and records the result returned in \vec{e} . If the process doesn't terminate, then it repeats this sequence of steps. Algorithm 4.3 contains the pseudocode for each process.

Fix an execution of the protocol. For each **scan**, S , by a process p , we define $\vec{\ell}(S)$ to be the m -component vector such that, for each value v , $\vec{\ell}_v(S)$ is the v 'th component, $\vec{\ell}_v$, of p 's local variable $\vec{\ell}$ computed on line 5 immediately after S . For every **swap** U to some swap object, let $\vec{\ell}(U)$ denote the argument of the **swap** operation.

The proof of correctness of this protocol is nearly identical to that of the protocol in Section 4.3. We begin with an observation that corresponds to Observation 4.7. The last additional part follows by definition of $\ell_v(S)$.

Observation 4.25. *Let U be a swap by a process p and let S be the last scan by p before U .*

(a) *For each value $v \in \{0, \dots, m-1\}$, $\vec{\ell}_v(U) \geq \vec{\ell}_v(S)$.*

```

1:  $v^* \leftarrow \text{input}()$ ,  $\vec{\ell} \leftarrow \vec{0}$ ,  $\vec{e} \leftarrow \vec{0}$ 
2: loop
3:    $(\vec{s}(j) : j \in [n-1]) \leftarrow \text{scan}(S_1, \dots, S_{n-1})$ 
4:   for  $v = 0, 1, \dots, m-1$  do
5:      $\vec{\ell}_v \leftarrow \max(\{\vec{\ell}_v, \vec{e}_v\} \cup \{\vec{s}(j)_v : j \in [n-1]\})$ 
6:     if  $\vec{\ell}_v > \vec{\ell}_{v^*}$  then
7:        $v^* \leftarrow v$ 
8:     if  $\vec{s}(j) = \vec{\ell}$  for all  $j \in [n-1]$  then
9:       if  $\vec{\ell}_{v^*} \geq \vec{\ell}_v + 2$  for all  $v \neq v^*$  then
10:        output( $v$ ) and terminate
11:         $\vec{\ell}_{v^*} \leftarrow \vec{\ell}_{v^*} + 1$ 
12:         $j \leftarrow \min\{j \in [n-1] : \vec{s}(j) \neq \vec{\ell}\}$ 
13:         $\vec{e} \leftarrow S_j.\text{swap}(\vec{\ell})$ 

```

Algorithm 4.3: An anonymous protocol that solves m -valued consensus among $n \geq 2$ processes in an obstruction-free manner using a snapshot object consisting of $n-1$ swap objects, S_1, \dots, S_{n-1} . The domain of each S_j is \mathbb{N}^m . Initially, each $S_j = \vec{0}$. Code for each process.

(b) If there is a value $v^* \in \{0, \dots, m-1\}$ such that $\vec{\ell}_{v^*}(U) > \vec{\ell}_{v^*}(S)$, then $\vec{\ell}_{v^*}(U) = \vec{\ell}_{v^*}(S) + 1$, $\vec{\ell}_{v^*}(S) \geq \vec{\ell}_v(S) = \vec{\ell}_v(U)$ for all other values $v \neq v^*$, and S returned a vector whose components all contain $\vec{\ell}(S)$.

(c) If \vec{e} is the vector returned by U , then, for any scan S' by p after U , $\vec{\ell}_v(S') \geq \max\{\vec{e}_v, \vec{\ell}_v(S)\}$, for each value $v \in \{0, \dots, m-1\}$.

The next lemma corresponds to Lemma 4.8 and it has nearly the same proof.

Lemma 4.26. Let S be any scan and let $v \in \{0, \dots, m-1\}$ be a value. If $\vec{\ell}_v(S) > 0$, then there was a scan, S' , performed prior to S such that S' returned a vector whose components all contain $\vec{\ell}(S')$, where $\vec{\ell}_v(S') = \vec{\ell}_v(S) - 1$ and $\vec{\ell}_{v'}(S') \leq \vec{\ell}_{v'}(S)$, for all values $v' \neq v$.

Proof. Since each buffer object initially contains $(\vec{0}, \dots, \vec{0})$, and $\vec{\ell}_v(S) > 0$, there was a **swap** U prior to S such that $\vec{\ell}_v(U) = \vec{\ell}_v(S)$. Consider the first such **swap**. Let p be the process that performed U and let S' be the last scan performed by p before U . By Observation 4.25(a), $\vec{\ell}_v(U) \geq \vec{\ell}_v(S')$. If $\vec{\ell}_v(U) = \vec{\ell}_v(S')$, there would have been a **swap** U' prior to S' and, hence, prior to U , with $\vec{\ell}_v(U') = \vec{\ell}_v(S') = \vec{\ell}_v(U) = \vec{\ell}_v(S)$, contradicting the definition of U . Hence $\vec{\ell}_v(U) > \vec{\ell}_v(S')$. By Observation 4.25(b), it follows that $\vec{\ell}_v(U) = \vec{\ell}_v(S') + 1$, $\vec{\ell}_{v'}(S') \leq \vec{\ell}_{v'}(S)$ for all other values $v' \neq v$, and S' returned a vector whose components all contain $\vec{\ell}(S')$. Since $\vec{\ell}_v(U) = \vec{\ell}_v(S)$, it follows that $\vec{\ell}_v(S') = \vec{\ell}_v(U) - 1 = \vec{\ell}_v(S) - 1$. \square

The next lemma is key to the proof of correctness. It corresponds to Lemma 4.9, but is more complex. The reason is that we need to argue about the information that processes locally store from their **swaps**.

Lemma 4.27. Suppose S is a scan that returns a vector whose components all contain $\vec{\ell}(S)$. If T is a scan performed after S , then, for each value $v \in \{0, \dots, m-1\}$, $\vec{\ell}_v(T) \geq \vec{\ell}_v(S)$.

Proof. Suppose, for a contradiction, that there is a scan T performed after S such that $\vec{\ell}_v(T) < \vec{\ell}_v(S)$ for some value $v \in \{0, \dots, m-1\}$. Consider the first such scan T . By definition of $\vec{\ell}_v(T)$, T returned a vector $(\vec{t}(j) : j \in [n-1])$ such that, for each component $j \in [n-1]$, $\vec{t}(j)_v < \vec{\ell}_v(S)$. Since S returned a vector whose components all contain $\vec{\ell}(S)$, for each $j \in [n-1]$, there is some **swap** U_j performed between S and T such that $\vec{t}(j) = \vec{\ell}(U_j)$.

Suppose there is some $j \in [n - 1]$ such that U_j is by some process whose last `scan`, S' , prior to U_j occurs no earlier than S . If $S' \neq S$, then S' occurs between S and T and, by definition of T , $\vec{\ell}_v(S') \geq \vec{\ell}_v(S)$. By Observation 4.25(a), $\vec{\ell}_v(U_j) \geq \vec{\ell}_v(S')$. Therefore, $\vec{t}(j)_v = \vec{\ell}_v(U_j) \geq \vec{\ell}_v(S)$. This is a contradiction.

So, for each $j \in [n - 1]$, `swap` U_j is by a process, q_j , whose last `scan` prior to U_j occurs before S . Since processes alternately perform `scan` and `swap`, it follows that each U_j is by a different process and is not by the process, p , that performed S . Thus, each process in $\{q_1, \dots, q_{n-1}\}$ is poised to perform `swap` to a different component immediately after S .

Consider any $j \in [n - 1]$. Let \vec{e} be the vector returned by U_j . We will show that $\vec{e}_v \geq \vec{\ell}_v(S)$. Suppose $\vec{e} \neq \vec{\ell}(S)$. Then \vec{e} is the argument of some `swap` U' performed by process p after S or by some process $q_i \neq q_j$ after it has performed its `swap` U_i . If U' is by p , then $\vec{e}_v \geq \vec{\ell}_v(S)$ by Observation 4.25(a). Now suppose U' is by some process $q_i \neq q_j$. Let S' be the last `scan` by q_i before U' . Since S' occurs between S and T , $\vec{\ell}_v(S') \geq \vec{\ell}_v(S)$, by definition of T . By Observation 4.25(a), $\vec{\ell}_v(U') \geq \vec{\ell}_v(S')$. Hence, $\vec{e}_v = \vec{\ell}_v(U') \geq \vec{\ell}_v(S)$. By Observation 4.25(c), for any `scan` T' performed by q_j after U_j , $\vec{\ell}_v(T') \geq \vec{e}_v \geq \vec{\ell}_v(S)$. Since $\vec{\ell}_v(T) < \vec{\ell}_v(S)$ and T was performed after U_j , it follows that T was not performed by process q_j .

Therefore, T was performed by p . However, by Observation 4.25(c), $\vec{\ell}_v(T) \geq \vec{\ell}_v(S)$, which is a contradiction. \square

The next lemma corresponds to Lemma 4.10. The proof is identical, except we replace Lemmas 4.8 and 4.9 with Lemmas 4.26 and 4.27.

Lemma 4.28. *Suppose S is a scan that returned a vector whose components all contain $\vec{\ell}(S)$ and there is some value $v^* \in \{0, \dots, m - 1\}$ such that $\vec{\ell}_{v^*}(S) \geq \vec{\ell}_v(S) + 2$ for all other values $v \neq v^*$. Then, for every scan T and every value $v \neq v^*$, $\vec{\ell}_v(T) < \vec{\ell}_v(S) + 2$.*

Proof. Suppose, for a contradiction, that there is some `scan` T and some value $v \neq v^*$ such that $\vec{\ell}_v(T) \geq \vec{\ell}_v(S) + 2 > 0$. Consider the first such `scan` T . By Lemma 4.26, there was a `scan`, T' , performed prior to T such that T' returned a vector whose components all contained $\vec{\ell}(T')$, where $\vec{\ell}_v(T') = \vec{\ell}_v(T) - 1$ and $\vec{\ell}_{v^*}(T') \leq \vec{\ell}_v(T')$. By definition of T and T' , $\vec{\ell}_v(T') < \vec{\ell}_v(S) + 2 \leq \vec{\ell}_v(T) = \vec{\ell}_v(T') + 1$, so $\vec{\ell}_v(S) + 2 = \vec{\ell}_v(T) = \vec{\ell}_v(T') + 1$. If S was performed after T' , then, by Lemma 4.27, $\vec{\ell}_v(S) \geq \vec{\ell}_v(T')$, which is not the case. Thus T' was performed after S . Hence, Lemma 4.27 implies that $\vec{\ell}_{v^*}(T') \geq \vec{\ell}_{v^*}(S)$. By assumption, $\vec{\ell}_{v^*}(S) \geq \vec{\ell}_v(S) + 2$. Hence, $\vec{\ell}_{v^*}(T') \geq \vec{\ell}_v(S) + 2 = \vec{\ell}_v(T') + 1$. This contradicts the fact that $\vec{\ell}_{v^*}(T') \leq \vec{\ell}_v(T')$. \square

We can now prove that the protocol is correct and obstruction-free. The proofs are nearly identical to Lemmas 4.11, 4.12, and 4.13, except we replace `write` with `swap` and we replace each reference to Lemmas 4.8, 4.9, and 4.10 with its corresponding counterpart in this section.

Lemma 4.29. *No two processes output different values.*

Proof. The last step performed by a process before it outputs a value v^* is a `scan`, S , such that S returned a vector whose components all contain $\vec{\ell}(S)$ and $\vec{\ell}_{v^*}(S) \geq \vec{\ell}_v(S) + 2$ for all other values $v \neq v^*$. Consider the first such `scan` S by any process. By Lemma 4.27, $\vec{\ell}_{v^*}(T) \geq \vec{\ell}_{v^*}(S)$ for every `scan` T performed after S . By Lemma 4.28, $\vec{\ell}_v(T) \leq \vec{\ell}_v(S) + 1$ for all $v \neq v^*$. Hence, $\vec{\ell}_{v^*}(T) > \vec{\ell}_v(T)$. It follows that no process ever decides $v \neq v^*$. \square

Lemma 4.30. *If some process outputs x , then some process has input x .*

Proof. Suppose that some process outputs x . Then there was a swap, U , such that $\vec{\ell}_x(U) > 0$. Consider the first such swap U . Let p be the process that performed U and let S be the last scan performed by p before U . By definition of U , $\vec{\ell}_x(S) = 0$. By Observation 4.25(b), $\vec{\ell}_x(S) \geq \vec{\ell}_v(S)$, for all values $v \neq x$. It follows that $\vec{\ell}(S) = \vec{0}$. By construction, p initialized its preferred value, v^* , to its input and $\vec{\ell} = \vec{0}$ on line 1. Since $\vec{\ell}(S) = \vec{0}$, when p runs lines 4–7 immediately after S , it does not change v^* . Therefore, since p sets $\vec{\ell}_x = 1$, it must be that $v^* = x$ and x is the input of process p . \square

Lemma 4.31. *Every process outputs a value and terminates after performing at most $3n - 2$ scans in a solo execution.*

Proof. Let p be any process and consider the first scan S performed by p in its solo execution. Let $\vec{\ell} = \vec{\ell}(S)$. Consider the value v^* stored by p at the end of the loop on lines 4–7 immediately after S . Notice that v^* is not changed during the rest of p 's solo execution.

After performing $\text{swap}(\vec{\ell})$ at most $n - 1$ times, p will perform a scan that returns a vector whose components all contain $\vec{\ell}$. If $\vec{\ell}_{v^*} \geq \vec{\ell}_v + 2$ for all $v \neq v^*$, then p outputs v^* immediately after this scan. Otherwise, p performs $\text{swap}(\vec{\ell})$ $n - 1$ more times, where $\vec{\ell}'_{v^*} = \vec{\ell}_{v^*} + 1$ and $\vec{\ell}'_v = \vec{\ell}_v$, for all values $v \neq v^*$. Then it performs a scan that returns a vector whose components all contain $\vec{\ell}'$. If $\vec{\ell}'_{v^*} \geq \vec{\ell}'_v + 2$ for all $v \neq v^*$, then p outputs v^* immediately after this scan. If not, then p performs $\text{swap}(\vec{\ell}')$ $n - 1$ more times, where $\vec{\ell}''_{v^*} = \vec{\ell}'_{v^*} + 1 = \vec{\ell}_{v^*} + 2$ and $\vec{\ell}''_v = \vec{\ell}'_v = \vec{\ell}_v$ for all values $v \neq v^*$. Finally, p performs a scan that returns a vector whose components all contain $\vec{\ell}''$ and outputs v^* immediately afterwards. Since p performs at most $3(n - 1)$ swaps and each swap is immediately followed by a scan, this amounts to at most $3(n - 1) + 1 = 3n - 2$ scans, including the first scan, S . \square

In the same way as in Section 4.3, we may generalize Algorithm 4.3 to work for arbitrary inputs. Hence, by Theorem 2.1, we have proved the following theorem.

Theorem 4.32. *There is an anonymous protocol that solves consensus among $n \geq 2$ processes in an obstruction-free manner using $n - 1$ swap objects.*

Chapter 5

Augmented Snapshot Objects

In this chapter, we describe some extensions to atomic snapshot objects. In Section 5.1, we describe related work. In particular, we describe Dolev and Shavit’s wait-free implementation of a multi-writer register from n single-writer registers. This forms the basis of our first extension, in Section 5.2. There, we describe an atomic snapshot object that supports, in addition to `update` and `scan`, a `block-update` operation, which atomically updates multiple components of the snapshot, for one fixed process. We then extend this implementation, in Section 5.3, so that the `block-update` operation returns a view of the snapshot at a prior point in the execution, satisfying some desirable properties. The final extension, presented in Section 5.4, supports `block-update` operations by any process, which either returns a view or a special yield symbol. We call this final object an *augmented* snapshot object and we use this object extensively in Chapter 6.

5.1 Related Work

Afek *et al.* [2] introduced the atomic snapshot object and showed that it can be implemented in a wait-free manner from registers. In particular, they gave a wait-free implementation of an m -component atomic snapshot object from m registers, each storing a value and a bounded amount of additional metadata. In their implementation, each operation completes after $O(mn)$ steps. Attiya and Rachman [12] showed how to implement a single-writer atomic snapshot object from single-writer registers so that each operation completes after $O(n \log n)$ steps. Inoue and Chen [44] showed how to implement a wait-free m -component atomic snapshot object from registers so that each operation completes after $O(m)$ steps. In contrast to Afek *et al.*’s implementation, the latter two implementations both use registers that store an unbounded amount of additional metadata.

Anderson [3] considered the problem of implementing an m -component atomic snapshot using a single-writer atomic snapshot object. Anderson’s implementation uses bounded-size metadata fields and is fairly complex. It may also be possible to extend Anderson’s implementation to implement the snapshot object in Section 5.2.

Dolev and Shavit [25] gave a simple implementation of a multi-writer register from n single-writer registers that store vector timestamps as metadata. Since multiple single-writer registers with the same writing process can be merged into one single-writer register, this allows one to implement any number of multi-writer registers using n single-writer registers. Our implementation in Section 5.2 is nearly

identical to this construction, except we use a single-writer atomic snapshot object instead of single-writer registers. We show that this allows us to implement extra functionality, such as **block-update** and **scan**.

An *immediate snapshot* object, introduced by Borowsky and Gafni [17], is a well-studied variant of a single-writer atomic snapshot object. The object supports a single operation, **update&scan**(v), which each process may perform to **update** its component in the snapshot to v and then perform a **scan**. Like the **block-update** operation in Section 5.4, the **update&scan** operation is not linearizable. However, the two operations, **update** and **scan**, contained inside the operation can be linearized separately. Moreover, they can be linearized so that, in the linearization, any maximal sequence of **update** operations is immediately followed by a sequence of **scans** by the same processes.

The *m-assignment* operation was introduced by Herlihy [37]. The operation takes m distinct registers, r_1, \dots, r_m , m arbitrary values, v_1, \dots, v_m , and atomically sets each register r_j to v_j . Herlihy proved that, for $m > 1$, the *m-assignment* operation has consensus number $2m - 2$, assuming the system has sufficiently many registers [37]. The **block-update** operation in this chapter may be viewed as a variant of an *m-assignment* operation.

5.2 Implementing block-update for a Single Process

In this section, we describe an easy modification to Dolev and Shavit's wait-free implementation of m registers, R_1, \dots, R_m , shared by n processes, p_1, \dots, p_n , from n single-writer registers, H_1, \dots, H_n , one per process. The main idea is that, if we replace H_1, \dots, H_n with a single-writer atomic snapshot object, H , then a **scan** of H yields a **scan** of R_1, \dots, R_m . The result is a wait-free implementation of an m -component atomic snapshot object, R , using a single-writer atomic snapshot object, H .

More importantly, we show how to extend this implementation of R so that it supports a **block-update** operation by a single process, p_1 . A **block-update** operation allows p_1 to atomically **update multiple** components of R . This forms the basis of our augmented snapshot implementations. To easily disambiguate between operations on R and H , we use **UPDATE**, **SCAN**, and **BLOCK-UPDATE** to denote R .**update**, R .**scan**, and R .**block-update**, respectively.

Suppose that, for $j \in [m]$, the domain of register R_j is D and the initial value of R_j is $\perp \in D$. Then, for $i \in [n]$, H_i stores a sequence of *records*, which represents the sequence of **UPDATES** performed by p_i on R . Each record is a triple (j, v, \vec{t}) , where $j \in [m]$ is a component, $v \in D$ is a value, and $\vec{t} \in \mathbb{N}^n$ is a *timestamp*. We order timestamps *lexicographically*, i.e. for any two timestamps $\vec{s}, \vec{t} \in \mathbb{N}^n$, \vec{s} is larger than \vec{t} if and only if there exists $i \in [n]$ such that $\vec{s}_j = \vec{t}_j$, for $1 \leq j < i$, and $\vec{s}_i > \vec{t}_i$. Initially, each component of H is the empty sequence.

To **UPDATE** component $j \in [m]$ of R to value $v \in D$, process p_i first performs an **H.scan**. Then, using the result, \vec{h} , of this **scan**, it *generates a new timestamp*, \vec{t} , as follows. For each $k \in [n]$, p_i sets \vec{t}_k to the length of \vec{h}_k , i.e. the number of records in \vec{h}_k , which is denoted $|\vec{h}_k|$. Then it increments \vec{t}_i . Finally, it appends the record (j, v, \vec{t}) to its component in H . We call (j, v, \vec{t}) the *record appended by* the **UPDATE** and \vec{t} the *timestamp associated with* the **UPDATE**. Algorithm 5.1 contains the pseudocode.

At all times, the contents of H represents a *view* of R , i.e. the contents of R at some point. To get the view, \vec{v} , represented by the contents, \vec{h} , of H , for each component $j \in [m]$ of R , process p_i computes the value \vec{v}_j as follows. If there is a record in \vec{h} whose first component is j , then \vec{v}_j is the value, $w \in D$, in the *unique* record (j, w, \vec{t}) in \vec{h} with the *largest* timestamp \vec{t} , among all records in \vec{h} whose first component

```

function UPDATE( $j, v$ )
1:  $\vec{h} \leftarrow \text{H.scan}()$ 
2:  $(\vec{t}_1, \dots, \vec{t}_{i-1}, \vec{t}_i, \vec{t}_{i+1}, \dots, \vec{t}_n) \leftarrow (|\vec{h}_1|, \dots, |\vec{h}_{i-1}|, |\vec{h}_i| + 1, |\vec{h}_{i+1}|, \dots, |\vec{h}_n|)$ 
3: H.update( $i, \vec{h}_i \cdot (j, v, \vec{t})$ )
4: return ACK

```

Algorithm 5.1: Implementation of an UPDATE by process p_i , for $i \in [n]$.

is j . (In Lemma 5.9, we show that all records with the same first component have different timestamps.) Otherwise, \vec{v}_j is the initial value, \perp . Algorithm 5.2 contains the pseudocode for this local procedure. In Sections 5.3 and 5.4, this procedure is also used as part of a BLOCK-UPDATE operation.

```

function GET-VIEW( $\vec{h}$ )
1: for  $j \in [m]$  do
2:    $\vec{t} \leftarrow \max(\{\vec{t} \in \mathbb{N}^n : \exists(j, w, \vec{t}) \in \vec{h}\} \cup \{\vec{0}\})$ 
3:    $\vec{v}_j \leftarrow \begin{cases} \text{value } w \text{ of unique record } (j, w, \vec{t}) \in \vec{h} & \text{if } \vec{t} \neq \vec{0} \\ \perp & \text{if } \vec{t} = \vec{0} \end{cases}$ 
4: return  $\vec{v}$ 

```

Algorithm 5.2: The view associated with the result, \vec{h} , of an H.scan. Code for each process.

To perform a SCAN on R , process p_i first performs an H.scan to obtain the current contents, \vec{h} , of H . It then gets the view of R represented by \vec{h} (using the local procedure GET-VIEW(\vec{h})) and returns the view. Algorithm 5.3 contains the pseudocode.

```

procedure SCAN
1:  $\vec{h} \leftarrow \text{H.scan}()$ 
2: return GET-VIEW( $\vec{h}$ )

```

Algorithm 5.3: Implementation of a SCAN. Code for process p_i .

We now discuss how the operations are linearized. Recall that the execution interval of an operation begins at its invoke step and ends at the response step. To show that the operation is linearizable, we must find a point in this interval (the linearization point) at which we can say the operation took effect.

It is tempting to linearize an UPDATE operation, U , by process p_i at its H.update. If $i = 1$, then it can be shown that this is a valid linearization point for the UPDATE. However, we note that it is not necessarily a valid linearization point when $i > 1$. The problem is that another UPDATE operation, U' , to the same component, by a process p_j , with $j < i$, may have begun and completed in interval between the H.scan and H.update of U . Since $j < i$, p_j increments a smaller component than p_i when it generates a new timestamp. As timestamps are ordered lexicographically, it follows that U' has a larger associated timestamp than U . Hence, during the GET-VIEW in any SCAN operation after U , the record appended by U' would be considered to be more recent than the record appended by U . Thus, in this case, U must be linearized before U' , so U can't be linearized at its H.update, which occurs after U' .

Instead, we linearize an UPDATE operation to component j with associated timestamp \vec{t} at the *first* point that H contains a record (j, v, \vec{t}') , where \vec{t}' is at least \vec{t} . Multiple UPDATES linearized at the same point are ordered by their associated timestamps. Observe that, due to how timestamps are generated, each Update is linearized at a point in the interval of its H.scan and H.update. Moreover, this point is immediately after the H.update that appended (j, v, \vec{t}') , which is performed in some UPDATE, possibly

by another process. A SCAN operation is linearized at its H.scan operation. It can be shown that this defines a correct implementation of an m -component atomic snapshot object.

Theorem 5.1. *Algorithms 5.1, 5.2, and 5.3 define a linearizable, wait-free implementation of an m -component atomic snapshot object from a single-writer atomic snapshot object.*

To extend this implementation so that process p_1 can perform a BLOCK-UPDATE to *distinct* components j_1, \dots, j_r of the snapshot with values v_1, \dots, v_r , respectively, we simply allow p_1 to simultaneously append *multiple* update records, $(j_1, v_1, \vec{t}), \dots, (j_r, v_r, \vec{t})$, each with the same timestamp, \vec{t} , which is the timestamp *associated* with the BLOCK-UPDATE. As the components j_1, \dots, j_r are distinct, this maintains the property that all records with the same first component have a different timestamp. Notice p_1 increments \vec{t}_1 by r when it generates \vec{t} .

procedure BLOCK-UPDATE($(j_1, \dots, j_r), (v_1, \dots, v_r)$)
 1: $\vec{h} \leftarrow \text{H.scan}()$
 2: $(\vec{t}_1, \vec{t}_2, \dots, \vec{t}_n) \leftarrow (|\vec{h}_1| + r, |\vec{h}_2|, \dots, |\vec{h}_n|)$
 3: **H.update**($1, \vec{h}_1 \cdot (j_1, v_1, \vec{t}) \cdots (j_r, v_r, \vec{t})$)
 4: **return** ACK

Algorithm 5.4: Implementation of a BLOCK-UPDATE by process p_1 .

A BLOCK-UPDATE is linearized at its H.update, while UPDATES and SCANS are linearized as described above. To see why this works, we view a BLOCK-UPDATE operation as a sequence of UPDATE operations, one for each component involved in the BLOCK-UPDATE. For each such UPDATE, the timestamp associated with the UPDATE is the timestamp associated with the BLOCK-UPDATE. Then all UPDATES (either in a BLOCK-UPDATE or performed individually) are linearized using the definition given earlier. Since p_1 generates the largest timestamps, it can be shown that UPDATES in a BLOCK-UPDATE operation are linearized consecutively. We omit the proof because it is a special case of the proof of Theorem 5.5.

Theorem 5.2. *Algorithms 5.1, 5.2, 5.3, and 5.4 define a linearizable, wait-free implementation of an m -component atomic snapshot object, which additionally supports a block-update operation by a single process, from a single-writer atomic snapshot object.*

5.3 Simple Augmented Snapshot Object

Suppose we wish to augment the implementation of Section 5.2 so that the BLOCK-UPDATE operation by p_1 returns a view of the snapshot object. Ideally, we would like the view to contain the contents of the snapshot immediately before the BLOCK-UPDATE operation. (Since processes in our simulation in Chapter 6 attempt to simulate a covering argument similar to the one Chapter 4, this would be very useful.) Unfortunately, this would make the object too powerful. The reason is that such a BLOCK-UPDATE operation can implement a swap operation. However, swap, together with read or scan, can be used to deterministically solve wait-free consensus between 2 processes [37], which is impossible using registers and atomic snapshot objects [2, 46].

Instead, we require that a BLOCK-UPDATE operation B returns a view of R at a previous point T such that the only operations linearized between T and T' , the linearization point of B , are UPDATES by other processes. In particular, no SCAN operation is linearized between T and T' . We call the resulting object a *simple augmented snapshot* object.

We note a consequence of the definition of a simple augmented snapshot object, which is used in Chapter 6. Recall that a $\text{BLOCK-UPDATE}((j_1, \dots, j_r), (v_1, \dots, v_r))$ operation can be viewed as a sequence of $\text{UPDATE}(j_1, v_1), \dots, \text{UPDATE}(j_r, v_r)$ operations. If there are only finitely many complete BLOCK-UPDATE operations, B_1, \dots, B_ℓ , in an execution, then the sequence of linearized UPDATES and SCANS in the execution may be written as $\alpha_1 \gamma_1 \beta_1 \cdots \alpha_\ell \gamma_\ell \beta_\ell \alpha_{\ell+1}$, where $\alpha_{\ell+1}$ contains all operations linearized after the last complete BLOCK-UPDATE operation and, for each $j \in [\ell]$,

- B_j returns the view at $\alpha_1 \gamma_1 \beta_1 \cdots \alpha_{j-1} \gamma_{j-1} \beta_{j-1} \alpha_j$,
- β_j consists of the UPDATES that are part of B_j , and
- the only operations that γ_j contains are UPDATE operations by processes other than p_1 . In particular, γ_j does not contain any SCAN operations.

An incorrect implementation of BLOCK-UPDATE would be to take Algorithm 5.4 and have p_1 simply return the view represented by the result, \vec{h} , of its H.scan , instead of ACK . To see why this is incorrect, consider the subinterval I of the execution interval of a BLOCK-UPDATE that has the H.scan and H.update as its endpoints. If there are only SCAN operations performed in I , then they all return \vec{h} and we may pick T as the linearization point of the last such SCAN . However, there is a problem if another process $p_i \neq p_1$ performs an UPDATE , followed by a SCAN , in I . In this case, the BLOCK-UPDATE must return the view represented by the contents of H at some point following this Scan , which it does not have access to (as it only knows \vec{h}).

Instead, each process p_i has an unbounded array of single-writer registers $L_i[b]$, for $b \geq 0$. Initially, $L_i[b]$ contains the n -component vector $(\varepsilon, \dots, \varepsilon)$, where ε denotes the empty sequence. We modify the implementation of a SCAN operation so that, during the operation, whenever process p_i performs an H.scan that returns a result, \vec{h} , it writes \vec{h} to $L_i[b]$, where $b = |\vec{h}_1|$, the number of records in H_1 . We call this a *helping scan* by p_i . Algorithm 5.5 contains the pseudocode.

function HELPING-SCAN

1: $\vec{h} \leftarrow \text{H.scan}()$
 2: $L_i[|\vec{h}_1|].\text{write}(\vec{h})$
 3: **return** \vec{h}

Algorithm 5.5: A helping scan by process p_i .

Then p_1 can use this information to perform a $\text{BLOCK-UPDATE}((j_1, \dots, j_r), (v_1, \dots, v_r))$ as follows. First, it performs lines 1–3 of Algorithm 5.4, i.e. p_1 performs an H.scan , obtaining \vec{h} as a result, generates a new timestamp \vec{t} , and then appends records $(j_1, v_1, \vec{t}), \dots, (j_r, v_r, \vec{t})$ to H_1 . After this, p_1 performs a collect on registers $L_2[|\vec{h}_1|], \dots, L_n[|\vec{h}_1|]$ to obtain vectors $\vec{L}_2, \dots, \vec{L}_n$. Since the number of records in each component of H is non-decreasing, p_1 can determine the result of the latest H.scan performed in the interval between its H.scan and H.update by determining the vector, $\vec{\ell}$, among $\{\vec{h}, \vec{L}_1, \dots, \vec{L}_n\}$ with the most records. It then returns the view represented by $\vec{\ell}$. Algorithm 5.6 contains the pseudocode. For uniformity, p_1 performs a helping scan to obtain \vec{h} in the pseudocode. Hence, it has written \vec{h} to $L_1[|\vec{h}_1|]$. Then it collects $L_1[|\vec{h}_1|], \dots, L_n[|\vec{h}_1|]$.

Going back to our counterexample for the incorrect implementation of a BLOCK-UPDATE , we see that, if p_i writes the result, \vec{h} , of its H.scan to $L_i[|\vec{h}_1|]$ prior to the collect on line 4, then p_1 would correctly return a view at a point no earlier than the H.scan , where p_i 's SCAN is linearized. However, if p_i writes \vec{h} to $L_i[|\vec{h}_1|]$ after the collect on line 4, then p_1 may still return an incorrect view. To fix this

```

procedure BLOCK-UPDATE( $(j_1, \dots, j_r), (v_1, \dots, v_r)$ )
1:  $\vec{h} \leftarrow$  HELPING-SCAN()
2:  $(\vec{t}_1, \vec{t}_2, \dots, \vec{t}_n) \leftarrow (|\vec{h}_1| + r, |\vec{h}_2|, \dots, |\vec{h}_n|)$ 
3: H.update( $1, \vec{h}_1 \cdot (j_1, v_1, \vec{t}) \cdots (j_r, v_r, \vec{t})$ )
4:  $(\vec{L}_1, \dots, \vec{L}_n) \leftarrow$  collect( $L_1[|\vec{h}_1|], \dots, L_n[|\vec{h}_1|]$ )
5:  $\vec{\ell} \leftarrow$  vector in  $\{\vec{L}_1, \dots, \vec{L}_n\}$  with the most records
6: return GET-VIEW( $\vec{\ell}$ )

```

Algorithm 5.6: Implementation of a BLOCK-UPDATE by process p_1 .

problem, we modify the implementation of SCAN so that p_i repeatedly performs helping scans until it performs a pair of consecutive helping scans that return the same result. Observe that, in our example, if p_i returns the view represented by \vec{h} , then p_i must have written \vec{h} to $L_i[|\vec{h}_1|]$ before the collect on line 4. Otherwise, its next helping scan would return $\vec{h}' \neq \vec{h}$ since p_1 appended new records before its collect. Algorithm 5.7 contains the pseudocode.

```

procedure SCAN
1:  $\vec{h} \leftarrow$  HELPING-SCAN()
2: loop
3:    $\vec{h}' \leftarrow$  HELPING-SCAN()
4:   if  $\vec{h}' = \vec{h}$  then return GET-VIEW( $\vec{h}$ )
5:    $\vec{h} \leftarrow \vec{h}'$ 

```

Algorithm 5.7: Implementation of SCAN for each process.

UPDATES and BLOCK-UPDATES are linearized as in the previous section. Each SCAN that returns is linearized at its last H.scan. It can be shown that this defines a non-blocking, linearizable implementation of an m -component simple augmented snapshot object. It is possible to remove the single-writer registers $L_i[b]$ by including the information in H. We discuss how to do this later on. We omit the proof because it is a special case of the proof of Theorem 5.5.

Theorem 5.3. *Algorithms 5.1, 5.2, 5.5, 5.6, and 5.7 define a non-blocking, linearizable implementation of an m -component simple augmented snapshot object from a single-writer atomic snapshot object.*

5.4 Augmented Snapshot Object

We now extend our implementation in the previous section so that every process, not only p_1 , can perform BLOCK-UPDATE operations. We call this object an *augmented snapshot* object. Unfortunately, we can no longer guarantee that all BLOCK-UPDATES are linearizable. The reason is that a BLOCK-UPDATE operation that atomically updates 2 components can implement a 2-assignment operation. However, 2-assignment, together with read or scan, can be used to deterministically solve wait-free consensus between 2 processes [37], which is impossible using registers or atomic snapshot objects [2, 46].

Instead, for the rest of this section, we view a BLOCK-UPDATE($(j_1, \dots, j_r), (v_1, \dots, v_r)$) operation as a sequence of UPDATE(j_1, v_1), \dots , UPDATE(j_r, v_r) operations. Since a BLOCK-UPDATE operation to a single component can implement an UPDATE operation to that component, we assume that each UPDATE is part of some BLOCK-UPDATE. We will show that all UPDATES are linearizable. (Analogously, a collect operation cannot always be linearized at a single point, but its individual reads are linearizable.)

In certain circumstances, a BLOCK-UPDATE operation is able to determine that its UPDATES can be linearized consecutively, at a single point, T' , and it can return a view at a previous point T such that the only operations linearized between T and T' are UPDATES that are part of *non-atomic* BLOCK-UPDATE operations by other processes. In this case, it returns some such view. We call such BLOCK-UPDATE operations *atomic*. (This can happen, for example, if it perceives no interference from other BLOCK-UPDATE operations.)

If a BLOCK-UPDATE operation is unable to determine that its UPDATES can be linearized consecutively or if it cannot return a view satisfying this property, then it returns a special “yield” symbol, ∇ . In Theorem 5.5, we show that if a BLOCK-UPDATE operation returns ∇ , then there was an update performed by a process with a smaller identifier in the execution interval of the BLOCK-UPDATE.

We note a consequence of the definition of an augmented snapshot object, which is used in Chapter 6.

Observation 5.4. *If there are only finitely many atomic BLOCK-UPDATE operations, B_1, \dots, B_ℓ , in an execution, then the sequence of UPDATES and SCANS linearized in the execution may be written as $\alpha_1\gamma_1\beta_1 \cdots \alpha_\ell\gamma_\ell\beta_\ell\alpha_{\ell+1}$, where $\alpha_{\ell+1}$ contains all operations linearized after the last atomic BLOCK-UPDATE operation and, for each $j \in [\ell]$,*

- B_j returns the view at $\alpha_1\gamma_1\beta_1 \cdots \alpha_{j-1}\gamma_{j-1}\beta_{j-1}\alpha_j$,
- β_j consists of the UPDATES that are part of B_j , and
- the only operations that γ_j contains are UPDATE operations that are part of non-atomic BLOCK-UPDATE operations by processes other than the process that performed B_j . In particular, γ_j does not contain any SCAN operations.

Recall that, in the previous section, each process p_i helped p_1 using an unbounded array L_i of single-writer registers. Specifically, after p_i performs a H.scan, p_i writes the result, \vec{h} , of the H.scan to $L_i[\vec{h}_1]$. To support BLOCK-UPDATE operations by every process, when a process performs a helping scan, it should help every other process. To help process p_j , process p_i stores an unbounded array $L_{i,j}[b]$ of single-writer registers, for $b \geq 0$. Initially, $L_{i,j}[b]$ contains the n -component vector $(\varepsilon, \dots, \varepsilon)$, where ε denotes the empty sequence. After p_i performs H.scan, it writes the result, \vec{h} , of the H.scan to $L_{i,j}[\vec{h}_j]$ for all $j \in [n]$. (It writes to $L_{i,i}[\vec{h}_i]$ to simplify the code.) Algorithm 5.8 contains the pseudocode.

function HELPING-SCAN
1: $\vec{h} \leftarrow \text{H.scan}()$
2: **for** $j \in [n]$ **do** $L_{i,j}[\vec{h}_j].\text{write}(\vec{h})$
3: **return** \vec{h}

Algorithm 5.8: A helping scan by process p_i .

Let B be a BLOCK-UPDATE($(j_1, \dots, j_r), (v_1, \dots, v_r)$) operation performed by process p_i . To perform B , p_i first performs a helping scan, H . (As in the implementation of a BLOCK-UPDATE by p_1 in Section 5.3, H can be a normal H.scan, but it makes the pseudocode on lines 7–8 simpler to have it be a helping scan.) Then, based on the result, \vec{h} , of H , p_i generates a new timestamp, \vec{t} . It then performs an H.update, X , which appends records $(j_1, v_1, \vec{t}), \dots, (j_r, v_r, \vec{t})$ to H_i . After this, p_i performs another helping scan, G , and obtains a result, \vec{g} . This helping scan serves two purposes.

First, G allows p_i to check whether there was an H.update by another process that may prevent the UPDATES in p_i 's BLOCK-UPDATE from being linearized consecutively. Specifically, if $|\vec{g}_j| \neq |\vec{h}_j|$ for

some $j < i$, then p_i knows that there was at least one concurrent `H.update` by process p_j that appended some records between H and G . Since p_j has a smaller identifier, these records may have a larger timestamp than \vec{t} . Hence, as discussed in Section 5.2, it may be necessary to linearize some `UPDATES` in p_i 's `BLOCK-UPDATE` before X , since they are superseded by p_j 's `UPDATES`. However, there may be other `UPDATES` in p_i 's `BLOCK-UPDATE` that are not superseded by any `UPDATES` prior to X . If another process performs a `SCAN` immediately before X , then it will not see these other `UPDATES`. On the other hand, if another process performs a `SCAN` immediately after X , then it will see them. Hence, these `UPDATES` must be linearized immediately after X . Therefore, the `UPDATES` in B may not be linearized consecutively and p_i returns ∇ to indicate this.

The second purpose of G is a bit more complicated to describe. Consider an atomic `BLOCK-UPDATE` B' that is concurrent with B . Recall that B' must return a view at a point T' such that no atomic `BLOCK-UPDATES` are linearized between T' and the linearization point of B' . The reason G is a helping scan instead of an `H.scan` is so that p_i can help B' determine T' . In particular, G makes B' aware of the `UPDATES` in B , which were recorded in H by X .

However, as in the discussion in Section 5.3 for the implementation of a `SCAN`, p_i 's help may have been missed by a process performing an atomic `BLOCK-UPDATE` that is concurrent with B . So, p_i performs another helping scan, G' , and obtains a result, \vec{g}' , to see whether its help may have been missed. If $|\vec{g}'_j| \neq |\vec{g}_j|$ for some $j < i$, then B also returns ∇ . The reason p_i only checks $|\vec{g}'_j|$ for $j < i$ is because concurrent `BLOCK-UPDATES` by processes with larger identifiers would return ∇ due the `H.update`, X , by p_i . So, p_i does not need to make sure that these processes see its help.

Finally, if $|\vec{g}'_j| = |\vec{h}_j|$ and $|\vec{g}'_j| = |\vec{g}_j|$ for all $j < i$, then p_i knows that the first point where H contains a timestamp greater than or equal to \vec{t} is immediately after its `H.update`. Moreover, it knows that the result, \vec{g} , of its helping scan G will be seen by any concurrent atomic `BLOCK-UPDATES`. In particular, p_i determines that the `UPDATES` that are part of B can be consecutively linearized immediately after X . Then p_i collects $L_{1,i}[\vec{h}_i], \dots, L_{n,i}[\vec{h}_i]$ in order to see the relevant helping scans (by itself and other processes). It returns the view represented by the result of the latest helping scan, which has the most records (as in the `BLOCK-UPDATE` of the previous section).

Algorithm 5.9 contains the pseudocode for a `BLOCK-UPDATE` by p_i . Observe that, for p_1 , the condition on line 6 is never true, so lines 4–6 are unnecessary.

```

function BLOCK-UPDATE( $(j_1, \dots, j_r), (v_1, \dots, v_r)$ )
1:  $\vec{h} \leftarrow$  HELPING-SCAN()
2:  $(\vec{t}_1, \dots, \vec{t}_{i-1}, \vec{t}_i, \vec{t}_{i+1}, \dots, \vec{t}_n) \leftarrow (|\vec{h}_1|, \dots, |\vec{h}_{i-1}|, |\vec{h}_i| + r, |\vec{h}_{i+1}|, \dots, |\vec{h}_n|)$ 
3: H.update( $i, \vec{h}_i \cdot (j_1, v_1, \vec{t}) \cdots (j_r, v_r, \vec{t})$ )
4:  $\vec{g} \leftarrow$  HELPING-SCAN()
5:  $\vec{g}' \leftarrow$  HELPING-SCAN()
6: if  $|\vec{g}'_j| \neq |\vec{h}_j|$  or  $|\vec{g}'_j| \neq |\vec{g}_j|$  for some  $j < i$  then return  $\nabla$ 
7:  $(\vec{L}_1, \dots, \vec{L}_n) \leftarrow$  collect( $L_{1,i}[\vec{h}_i], \dots, L_{n,i}[\vec{h}_i]$ )
8:  $\vec{\ell} \leftarrow$  vector in  $\{\vec{L}_1, \dots, \vec{L}_n\}$  with the most records
9: return GET-VIEW( $\vec{\ell}$ )

```

Algorithm 5.9: Implementation of `BLOCK-UPDATE` for process p_i .

We linearize the operations as follows. Each `UPDATE` to component $j \in [m]$ in a `BLOCK-UPDATE` with associated timestamp \vec{t} is linearized at the first point in the execution where H contains a record (j, v, \vec{t}') such that \vec{t}' is at least \vec{t} . If multiple `UPDATES` are linearized at the same point, then they are

ordered by their associated timestamps, from smallest to largest, and then in increasing order of the components they update. Recall that the timestamp associated with an UPDATE is the timestamp associated with the BLOCK-UPDATE that it is a part of. Each SCAN that returns is linearized at its last H.scan.

5.5 Proof of Correctness and Termination

The following theorem summarizes the properties that an augmented snapshot object should satisfy and says that our implementation does satisfy these properties.

Theorem 5.5. *Algorithms 5.2, 5.5, 5.7, and 5.9 define a non-blocking implementation of an m -component augmented snapshot object from a single-writer atomic snapshot object. Each SCAN operation and each UPDATE that is part of a BLOCK-UPDATE operation is linearizable. The UPDATES in an atomic BLOCK-UPDATES are linearized consecutively. If a BLOCK-UPDATE by process p_i is not atomic, then there was a step by another process p_j , with $j < i$, performed in the execution interval of the BLOCK-UPDATE. Moreover, this step by p_j was part of another BLOCK-UPDATE.*

We begin by making three observations. The first follows from the fact that each process only appends records to its component H. The second follows from this fact and the fact that H.scan is an atomic operation. The third follows from the way timestamps are generated on line 2 of Algorithm 5.9.

Observation 5.6. *The number of records in each component of H is non-decreasing. The number of records in each component of $L_{i,j}[b]$, for $i, j \in [n]$ and $b \geq 0$, is also non-decreasing.*

Observation 5.7. *Let H and H' be H.scans that return \vec{h} and \vec{h}' , respectively. If H occurred before H' , then $|\vec{h}_j| \leq |\vec{h}'_j|$ for all $j \in [n]$. Conversely, if $|\vec{h}_j| < |\vec{h}'_j|$ for some $j \in [n]$, then H occurred before H' .*

Observation 5.8. *Consider an H.update by process p_i that appends records with timestamp \vec{t} . Then, immediately after this H.update, the number of records in H_i is precisely \vec{t}_i and, for each $j \neq i$, the number of records in H_j is at least \vec{t}_j . Consequently, each timestamp generated by a process is larger than any timestamp contained in the result of the H.scan the process used to generate the timestamp.*

Now we prove that each BLOCK-UPDATE operation has a different associated timestamp. Since no BLOCK-UPDATE contains multiple UPDATES to the same component, this implies that each UPDATE has a different associated timestamp.

Lemma 5.9. *Each BLOCK-UPDATE operation has a different associated timestamp.*

Proof. Consider any two BLOCK-UPDATE operations B and B' with associated timestamps \vec{t} and \vec{t}' , respectively. Let \vec{h} and \vec{h}' the results of the H.scans H and H' that were performed to generate \vec{t} and \vec{t}' , respectively. If B is performed before B' and by the same process, then \vec{t} is contained in \vec{h}' . Thus, by Observation 5.8, \vec{t} is larger than \vec{t}' . Now suppose B and B' are performed by different processes, p_i and p_j , respectively. From the way timestamps are generated, $\vec{t}_i > |\vec{h}_i|$, $\vec{t}_j = |\vec{h}_j|$, $\vec{t}'_j > |\vec{h}'_j|$, and $\vec{t}'_i = |\vec{h}'_i|$. If $\vec{t} = \vec{t}'$, then $\vec{t}_i = |\vec{h}'_i|$ and $\vec{t}'_j = |\vec{h}_j|$. It follows that $|\vec{h}'_i| > |\vec{h}_i|$ and $|\vec{h}_j| > |\vec{h}'_j|$. However, by Observation 5.7, this is impossible. Therefore, $\vec{t} \neq \vec{t}'$. \square

Next, we show an operation that is completed before another operation begins is linearized before the latter operation.

Lemma 5.10. *Each operation is linearized at a point in the execution interval of the operation. Moreover, each UPDATE that is part of a BLOCK-UPDATE operation is linearized at or before the point immediately following the H.update in the BLOCK-UPDATE (on line 3 of Algorithm 5.9).*

Proof. A SCAN is linearized at its last H.scan, which is in its execution interval. Now consider a BLOCK-UPDATE operation, B . Let H be the first H.scan in B , and let X be the first H.update in B . We show that all UPDATES in B are linearized after H and no later than the point immediately after X . Let \vec{t} be the timestamp associated with B and let \vec{h} be the result of H that is used to generate \vec{t} . Consider any UPDATE U , to some component $j \in [m]$, that is part of B . By definition, U is linearized at the first point that H contains a record with component j and timestamp \vec{t}' that is at least \vec{t} . By Observation 5.8, \vec{t} is larger than any timestamp contained in \vec{h} . Thus, U is linearized after H . Since X appends a record with component j and timestamp \vec{t} , the claim follows. \square

The following observation holds because the condition on line 6 of Algorithm 5.9 is not satisfied for an atomic BLOCK-UPDATE.

Observation 5.11. *Let B be a BLOCK-UPDATE operation performed by p_i . Let H be the first H.scan in B (on line 1) and G' be the last H.scan in B (on line 5). Then B is atomic if and only if, between H and G' , there are no H.updates by any process p_j with $j < i$.*

Using this observation, we can show that an atomic BLOCK-UPDATE operation is linearized immediately following its first H.update.

Lemma 5.12. *All UPDATES that are part of an atomic BLOCK-UPDATE are linearized consecutively, immediately after the H.update in B , in order of the components they update.*

Proof. Consider any atomic BLOCK-UPDATE B with associated timestamp \vec{t} that is performed by a process p_i . Let H be the first H.scan in B (performed on line 1 during HELPING-SCAN), let \vec{h} be the result of H (which p_i uses to generate \vec{t}), and let X be the H.update in B (on line 3 of Algorithm 5.9).

We claim that, prior to X , H contains only records with timestamps that are less than \vec{t} . Suppose, for a contradiction, that there was an H.update prior to X that appended a record with a timestamp that is at least \vec{t} . Let X' be the first such H.update and let \vec{t}' be the timestamp of each record appended by X' . By Lemma 5.9, each BLOCK-UPDATE has a different associated timestamp. Hence, \vec{t}' is larger than \vec{t} . It follows, by Observation 5.8, that \vec{t}' is not contained in \vec{h} and, hence, X' occurs after H . Since X' occurs between H and X , X' was performed by a process $p_j \neq p_i$. Let \vec{h}' be the result of the H.scan, H' , that p_j performed to generate \vec{t}' . Since B is atomic, Observation 5.11 implies that, between H and X , there are no H.updates by any process p_k with $k < i$. It follows that $j > i$ and $|\vec{h}'_k| \leq |\vec{h}_k|$, for $k < i$. Moreover, since H' occurs before X' and X' occurs before X , by Observation 5.8, $|\vec{h}'_i| < |\vec{h}_i|$. Thus, $\vec{t}'_k = |\vec{h}'_k| \leq |\vec{h}_k| = \vec{t}_k$, for $k < i$, and $\vec{t}'_i = |\vec{h}'_i| < \vec{t}_i$. However, this implies that \vec{t} is larger than \vec{t}' , which is a contradiction.

Thus, any H.update that appends a record with timestamp larger than \vec{t} must occur after X . All UPDATES that are part of B have the same timestamp, \vec{t} . By Lemma 5.9, no other BLOCK-UPDATE has this timestamp. Hence, all UPDATES that are part of B are linearized immediately after X . Recall that UPDATES linearized at the same point are ordered first by their timestamps and then by the components they update. Thus, there will not be any UPDATES that are part of other BLOCK-UPDATES interleaved with the UPDATES that are part of B . Therefore, all UPDATES that are part of B are linearized consecutively, in increasing order of the components that they update. \square

We now show that the result of each `H.scan` represents view that is consistent with the `UPDATE` operations linearized before the `H.scan`. This implies that each `SCAN` operation returns a view that is consistent with its linearization point, which is at its last `H.scan`.

Lemma 5.13. *Let H be an `H.scan` that returns \vec{h} and let \vec{v} be the view represented by \vec{h} , i.e. $\vec{v} = \text{GET-VIEW}(\vec{h})$. Then, for each $j \in [m]$, v_j is the written value by the last `UPDATE` to component j linearized before H , or \perp , if no such `UPDATE` exists.*

Proof. Consider any component $j \in [m]$. By Lemma 5.10, each `UPDATE` to component j that is part of a `BLOCK-UPDATE` is linearized at or before the point immediately following the `H.update` in the `BLOCK-UPDATE`, which appends a record with component j . If no `UPDATES` to component j are linearized before H , then no `H.update` appended a record with component j prior to H . Thus, \vec{h} does not contain any record with component j . By the definition of `GET-VIEW` (Algorithm 5.2), this implies that $v_j = \perp$.

Now suppose that there was an `UPDATE` to component j linearized prior to H . Consider the last such `UPDATE` U . Let \vec{t} be the timestamp associated with U and let \vec{t}' be the largest timestamp of any record with component j in \vec{h} . Recall that U is linearized at the first point that `H` contains a record with component j and timestamp at least \vec{t} . Since U is linearized before H , it follows that \vec{t}' is at least \vec{t} . If \vec{t}' is larger than \vec{t} , then there is another `UPDATE` to component j with associated timestamp \vec{t}' . Its `H.update` was performed before H and, hence, is linearized prior to H . Since it has a larger timestamp than U , this `UPDATE` is linearized after U . This is impossible because U is the last `UPDATE` linearized before H . Thus, $\vec{t} = \vec{t}'$. By Lemma 5.9, there is exactly one record in \vec{h} with component j and timestamp \vec{t} . This is the record representing the `UPDATE` U . By definition of `GET-VIEW`(\vec{h}), the j 'th component of the vector, \vec{v} , returned by H is value of this record, which is the value written by U . \square

Consider the view returned by an atomic `BLOCK-UPDATE`, B , by a process p_i . Recall that, to compute the view, p_i performs `collect` on $L_{1,i}[\vec{h}_i], \dots, L_{n,i}[\vec{h}_i]$, where \vec{h} is the result of its first `H.scan` in B . Let $\vec{L}_1, \dots, \vec{L}_n$ be the vectors that it collects. We define $\vec{\ell}(B)$ to be the vector in $\{\vec{L}_1, \dots, \vec{L}_n\}$ with the most records, computed on line 8 of `BLOCK-UPDATE` (Algorithm 5.9). Notice that, by Observation 5.7, if the results of two `H.scans` contain the same number of records, they contain the same number of records in each component. By line 3 of `BLOCK-UPDATE`, records are only appended to each component of `H`. Thus, if the results of two `H.scans` contain the same number of records, they are the same.

Lemma 5.14. *Let B be an atomic `BLOCK-UPDATE` operation, let H be the first `H.scan` in B , let X be the `H.update` in B , and let L be the last `H.scan` that returns $\vec{\ell}(B)$. Then L occurs at or after H and before X .*

Proof. Let p_i be the process that performed B , let \vec{h} be the result of H , and let $b = |\vec{h}_i|$. Let $\vec{L}_1, \dots, \vec{L}_n$ be the vectors that p_i collects from $L_{1,i}[b], \dots, L_{n,i}[b]$ on line 7 of `BLOCK-UPDATE` (Algorithm 5.9). Each $L_{j,i}[b]$ is initialized to $(\varepsilon, \dots, \varepsilon)$. By line 2 of `HELPING-SCAN` (Algorithm 5.8), when a process writes to $L_{j,i}[b]$, it writes the result, \vec{h}' , of an `H.scan` such that $|\vec{h}'_i| = b$. In particular, H was performed in a `HELPING-SCAN` and the result, \vec{h} , of H was written to $L_{i,i}[b]$. Since X appends new records to component i of `H`, the number of records in component i of `H` is greater than b after X . It follows that each \vec{L}_j , which was read from $L_{j,i}[b]$, is either $(\varepsilon, \dots, \varepsilon)$ or the result of an `H.scan` performed before X . Notice that H is the last `H.scan` performed by p_i before X , so $\vec{L}_i = \vec{h}$.

By definition, $\vec{\ell}(B)$ is the vector in $\{\vec{L}_1, \dots, \vec{L}_n\}$ with the most records. Hence, $\vec{\ell}(B)$ contains at least as many records as \vec{h} . If $\vec{\ell}(B)$ contains more records than \vec{h} , then Observation 5.7 implies that L occurs

after H . Otherwise, $\vec{\ell}(B)$ contains the same number of records as \vec{h} and, hence, $\vec{\ell}(B) = \vec{h}$. Since L is the last H.scan that returns $\vec{\ell}(B)$ and H returns \vec{h} , L occurs at or after H .

If $\vec{\ell}(B) \neq (\varepsilon, \dots, \varepsilon)$, then it is the result, \vec{L}_j , of an H.scan performed before X . Otherwise, every \vec{L}_j is $(\varepsilon, \dots, \varepsilon)$ and $\vec{\ell}(B) = (\varepsilon, \dots, \varepsilon)$ is the result, $\vec{h} = \vec{L}_i$, of an H.scan performed before X . In both cases, the number of records in component i of $\vec{\ell}(B)$ is less than the number of records in component i of \mathbf{H} after X . It follows that L occurs before X . \square

We now prove that each atomic BLOCK-UPDATE returns a view satisfying the specifications of the augmented snapshot object.

Lemma 5.15. *Let B be an atomic BLOCK-UPDATE operation by process p_i , let L be the last scan that returns $\vec{\ell}(B)$, and let X be the H.update in B . Then the only operations linearized between L and X are UPDATES that are part of non-atomic BLOCK-UPDATES by other processes $p_j \neq p_i$.*

Proof. We begin by showing that there are no SCAN operations linearized between L and X . Suppose, for a contradiction, that a SCAN operation S is linearized between L and X . Let F be the last H.scan in S . Then S is linearized at F . It follows that F occurs between L and X . Let H be the first H.scan in B . By Lemma 5.14, H occurs at or before L and, hence, before F . Since p_i performs no H.scans between H and X , F is performed by another process $p_j \neq p_i$.

Let \vec{f} be the result of F and let \vec{h} be the result of H . Since F is performed after H and before X , $|\vec{f}_i| = |\vec{h}_i|$. Since F is the last H.scan by p_j performed before S returns, the H.scan by p_j prior to F also returned \vec{f} , by line 4 of SCAN (Algorithm 5.7). Hence, by line 2 of HELPING-SCAN (Algorithm 5.8), p_j wrote \vec{f} to $\mathsf{L}_{j,i}[\vec{h}_i]$ prior to F . After performing X , p_i reads $\mathsf{L}_{j,i}[\vec{h}_i]$ on line 7. Since X occurs after F , by Observation 5.6, p_i sees a vector with at least as many records as \vec{f} in each component. Thus, $|\vec{\ell}(B)_j| \geq |\vec{f}_j|$ for each component j . Since L occurs before F , $|\vec{\ell}(B)_j| \leq |\vec{f}_j|$ for each component j , by Observation 5.7. It follows that $\vec{\ell}(B) = \vec{f}$. Since F occurs after L , this contradicts the fact that L is the last H.scan that returns $\vec{\ell}(B)$.

We now show that there are no UPDATES that are part of atomic BLOCK-UPDATE operations linearized between L and X . Suppose, for a contradiction, that some UPDATE in another atomic BLOCK-UPDATE B' is linearized between L and X . By Lemma 5.12, the UPDATES in B' are linearized consecutively, immediately after the H.update, X' , in B' . It follows that X' occurs between L and X . Let p_i and p_j be the processes that perform B and B' , respectively. Notice that $p_i \neq p_j$ since B and B' are concurrent BLOCK-UPDATES. By Lemma 5.14, L occurs at or after the first H.scan in B . Since X' occurs between L and X , Observation 5.11 implies that $j > i$.

Let \vec{h} , \vec{g} , and \vec{g}' , be the results of the three H.scans H , G , and G' in B' (on lines 1, 4, and 5 of BLOCK-UPDATE). Since B' is atomic, the test on line 6 of BLOCK-UPDATE is false. In particular, since $i < j$, $|\vec{h}_i| = |\vec{g}_i| = |\vec{g}'_i|$. By definition, H occurs before X' , which occurs before X . Since X appends new records to component i , it follows that both G and G' occur before X . The $\mathsf{L}_{j,i}[\vec{g}_i]$.write(\vec{g}) in the HELPING-SCAN containing G occurs before G' . Hence, it occurs before X .

From the beginning of B until X , p_i performs no H.updates. Hence, the i 'th component of \mathbf{H} does not change. Since G occurs between the beginning of B and X , it follows that the first H.scan in B returns a vector with the same i 'th component as \vec{g} . Let \vec{L}_j be the vector that p_i reads from $\mathsf{L}_{j,i}[\vec{g}_i]$ in its collect on line 7 of BLOCK-UPDATE (Algorithm 5.9), which occurs after X . Since \vec{g} was written to $\mathsf{L}_{j,i}[\vec{g}_i]$ prior to X , by Observation 5.6, \vec{L}_j contains at least as many records as \vec{g} . By definition, $\vec{\ell}(B)$ contains at least as many records as \vec{L}_j and, hence, \vec{g} .

Recall that L is the last H.scan to return $\vec{\ell}(B)$. If \vec{g} contains fewer records than $\vec{\ell}(B)$, then Observation 5.7 implies that G occurs before L . Otherwise, $\vec{\ell}(B)$ contains the same number of records as \vec{g} and, hence, $\vec{\ell}(B) = \vec{g}$. In this case, by definition of L , G occurs at or before L . In either case, G occurs at or before L , which occurs before X' . This contradicts the fact that G occurs after X' in B' .

Finally, consider an UPDATE that is part of a non-atomic BLOCK-UPDATE by p_i . By Lemma 5.10, it is linearized at some point in the execution interval of this BLOCK-UPDATE. Since this BLOCK-UPDATE and B are different operations by the same process, they have disjoint execution intervals. Hence, the UPDATE is not linearized between L and X . \square

We observe that the set of arrays $L_{i,j}$, for $j \in [n]$, can be represented using an additional field, L , of H_i . To perform $L_{i,j}[b].write(\vec{h})$, p_i appends (j, b, \vec{h}) to the L field of H_i . To perform $L_{j,i}[b].read$, p_i performs an H.scan and then checks if there is a triple (i, b, \vec{h}) in the L field of H_j . If (i, b, \vec{h}) is the last such triple, then p_i returns \vec{h} as its result. If no such triple exists, then p_j returns the initial value, $(\varepsilon, \dots, \varepsilon)$, as its result. Observe that, given this representation, it is possible for p_i to perform the sequence of writes to $L_{i,j}$, for $j \in [n]$, on line 2 of HELPING-SCAN (Algorithm 5.8), by performing a single H.update. Similarly, it can perform the collect of $L_{j,i}$, for $j \in [n]$, on line 7 of BLOCK-UPDATE (Algorithm 5.9), by performing a single H.scan. It is also necessary to throw away the contents of the L fields from the result of H.scans performed on line 1 of HELPING-SCAN. Therefore, the single-writer registers are unnecessary for the implementation.

Finally, it remains to prove that the implementation is non-blocking. In particular, each BLOCK-UPDATE is wait-free and a SCAN terminates unless infinitely many BLOCK-UPDATE operations are completed in the execution interval of the SCAN. We also analyze the step complexity of the operations.

Lemma 5.16. *Each BLOCK-UPDATE operation consists of 8 steps. During the execution interval of a SCAN operation, if there are k H.update operations performed on line 3 of BLOCK-UPDATE, then the SCAN consists of at most $2k + 4$ steps.*

Proof. Each BLOCK-UPDATE operation by a process p_i consists of 3 helping scans, an H.update, and a collect of $L_{1,i}, \dots, L_{n,i}$. A helping scan by p_i consists of an H.scan and one write to $L_{i,j}$, for each $j \in [n]$. These writes may be simultaneously performed by a single H.update to the L field of H_i . Hence, each helping scan consists of 2 steps. Similarly, the collect of $L_{1,i}, \dots, L_{n,i}$ may be performed with a single H.scan. Therefore, each BLOCK-UPDATE operation consists of 8 steps.

Each SCAN operation by p_i begins with a helping scan by p_i . In each iteration of the loop in Algorithm 5.7, p_i performs exactly one helping scan. An iteration of the loop returns unless there was an H.update by another process performed on line 3 of BLOCK-UPDATE in between the last helping scan by p_i prior to iteration and the helping scan by p_i in the iteration. Hence, p_i performs at most $k + 1$ iterations. Since each helping scan by p_i consists of an H.scan and an H.update, the SCAN consists of most $2k + 4$ steps. \square

Chapter 6

Revisionist Simulations

In this chapter, we introduce *revisionist simulations*, which is a technique for proving space lower bounds using a simulation argument. We show how revisionist simulations can be used to obtain space lower bounds for colourless tasks, which includes consensus, set agreement, and approximate agreement. In Section 6.1, we describe related work. In Section 6.2, we give an overview of the main theorem proved in the chapter as well as some notation. In Section 6.3, we describe a simple case of the simulation, where each simulator is essentially simulating a single process. In Section 6.4, we describe the general simulation, where some simulators may simulate multiple processes at once. In Section 6.5, we describe an intermediate execution, which allows us to more easily reason about the steps taken by the simulators. In Sections 6.6 and 6.7, we prove the correctness and wait-freedom of the simulation. Finally, in Section 6.8, we show how the simulation can be used to obtain space lower bounds for colourless tasks.

6.1 Related Work

There are only a few space bounds for x -obstruction-free k -set agreement using registers. Delporte-Gallet, Fauconnier, Kuznetsov, and Ruppert proved that it is impossible to solve obstruction-free k -set agreement using a single register [23]. In the same paper, they proved that, for $1 \leq x \leq k < n$, $\Omega(\sqrt{\frac{n}{k}x})$ registers are needed to solve k -set agreement in an x -obstruction-free manner among anonymous processes, when the inputs of the processes are arbitrary natural numbers. The bound is proved using a covering argument that exploits the use of clones (as in [28]) and the fact that the set of possible inputs is infinite. Finally, they gave an x -obstruction-free protocol for k -set agreement using $n - k + 2x$ registers. Bouzid, Reynal, and Sutra gave an anonymous x -obstruction-free protocol for k -set agreement using $n - k + x$ registers [19], which is the best known upper bound.

Schenk proved that $\Theta(\log(\frac{1}{\epsilon}))$ binary registers are necessary and sufficient to solve ϵ -approximate agreement, for $\epsilon > 0$ [52]. The lower bound is obtained by an information theoretic argument. Attiya, Lynch, and Shavit proved that n single-writer registers are sufficient to solve ϵ -approximate agreement, for $\epsilon > 0$ [10].

Schenk also proved that, in systems containing only binary registers, the step complexity of ϵ -approximate agreement is $O(\log(\frac{1}{\epsilon}))$ [52]. Herlihy proved that $\Omega(\log(\frac{1}{\epsilon}))$ steps are necessary to solve ϵ -approximate agreement in a wait-free manner using registers [36].

Borowsky and Gafni introduced the BG simulation [16, 15]. The BG simulation enables a small

number of processes to simulate a protocol designed for a larger number of processes. In particular, given a protocol solving k -set agreement among $n > f \geq k$ processes using only registers, which tolerates up to f crashes, the BG simulation enables $f+1$ processes to simulate the protocol in a wait-free manner, using only registers. Since it is impossible for $f+1$ processes to solve k -set agreement in a wait-free manner using only registers [18, 41, 51], no such protocol can exist. Borowsky, Gafni, Lynch, and Rajsbaum generalized the BG simulation to protocols for colourless tasks [15].

Yanagisawa [54] showed that, for systems containing only registers, a colourless task can be solved in a wait-free manner if and only if it can be solved anonymously in a wait-free manner. Delporte-Gallet, Fauconnier, Rajsbaum and Yanagisawa [24] showed that the same result holds when at most $f < n$ processes may crash. The main component of both papers is an anonymous implementation of a *weak set* object using only registers. A weak set object maintains a set of values. It supports an $\text{add}(v)$ operation, which adds a value to the set, and a get operation, which returns the set of values added so far. Delporte-Gallet *et al.*'s implementation is simple and non-blocking, while Yanagisawa's is wait-free, but more complex. Yanagisawa used weak set objects to describe a full-information anonymous protocol and showed that, for colourless tasks, anonymous full-information protocols have the same topological structure as non-anonymous full-information protocols. Hence, they solve precisely the same set of colourless tasks. Delporte-Gallet *et al.* used weak set objects to give an anonymous version of the BG simulation. In particular, they showed that a protocol solving a colourless task in a wait-free manner among n processes, at most $f < n$ of which may crash, can be simulated anonymously.

6.2 Overview

In the rest of this chapter, we focus on proving the following result.

Theorem 6.1. *Let T be a colourless task and let Π be an obstruction-free protocol that solves T among $n' \geq n \geq 2$ processes using an m -component atomic snapshot object, where $m \geq 2$.*

1. *If $m \leq \lfloor n'/n \rfloor$, then there exists a deterministic, wait-free protocol that solves T among n processes using a single-writer atomic snapshot object. Moreover, in this protocol, every process terminates after taking at most 2^{nm^2} steps.*
2. *If Π is x -obstruction-free for some $x \in [n-1]$ and $m \leq \lfloor (n'-x)/(n-x) \rfloor$, then there exists a deterministic, wait-free protocol that solves T among n processes using a single-writer atomic snapshot object.*

Notice that case 1 provides an upper bound on the number steps any process needs to take before terminating, but case 2 does not. However, case 2 is applicable for more values of m .

For the rest of this chapter, let Π be an obstruction-free protocol that solves a task among $n' \geq n$ processes using an m -component atomic snapshot object, \mathbf{S} . In both cases, we define a protocol (*the simulation*) for n processes, $\mathbf{p}_1, \dots, \mathbf{p}_n$, that simulates Π using an m -component augmented snapshot object, \mathbf{R} . (More precisely, we use the non-blocking implementation of \mathbf{R} from a single-writer atomic snapshot object in Chapter 5.) Each *simulator* \mathbf{p}_i is responsible for simulating a set of processes, P_i , of Π and locally stores a state for each of these processes. A simulator typically, but not always, updates the state of one of its processes immediately after it simulates some steps of that process. The processes in $P_1 \cup \dots \cup P_n$ are called *simulated* processes. Crucially, each simulated process is simulated by exactly

one simulator, i.e. for all $i \neq j$, P_i and P_j are disjoint. For each $i \in [n]$, we fix an ordering on P_i and, for each $r \in [|P_i|]$, we denote the r 'th process in P_i as $p_{i,r}$.

If $m \leq \lfloor n'/n \rfloor$, then each simulator simulates m processes. Observe that, since $m \leq \lfloor n'/n \rfloor$, the total number of simulated processes is $nm \leq n'$. If Π is x -obstruction-free for some $x \in [n-1]$ and $m \leq \lfloor (n'-x)/(n-x) \rfloor$, then each of the first $n-x$ simulators simulates m processes, while the remaining x simulators simulate one process each. Observe that, since $m \leq \lfloor (n'-x)/(n-x) \rfloor$, the total number of simulated processes is $(n-x)m + x \leq n'$. In either case, each simulator that simulates m (≥ 2) processes is called a *covering* simulator. The remaining simulators are called *direct* simulators.

For clarity, we distinguish between two asynchronous shared memory systems, the *real* system, in which the simulation is taking place, and the *simulated* system, in which the protocol Π is running. We will use bold fonts for processes, executions, and steps in the real system and use normal fonts for processes, executions, and steps in the simulated system. We use BLOCK-UPDATE, UPDATE, and SCAN to denote the operations on \mathbf{R} and we use update and scan to denote operations on \mathbf{S} .

Finally, without loss of generality, we assume that each simulated process $p_{i,r}$ in Π alternately performs scan and update operations on the snapshot object, \mathbf{S} , starting with a scan, until it performs a scan that allows it to output a value and terminate. To justify this assumption, notice that, between two consecutive update operations, $p_{i,r}$ can perform a scan and ignore its result. Moreover, if $p_{i,r}$ is supposed to perform multiple consecutive scans, it can, instead, perform one scan and use its result as the result of the others. This is because it is possible for all these scans to occur consecutively in an execution, in which case, they *would* all get the same result.

6.3 Direct Simulators and a Simple Case of the Simulation

In this section, we consider a simple case of the simulation, when the atomic snapshot object used in Π , \mathbf{S} , contains $m = 2$ components and there are $n = 2$ simulators, \mathbf{p}_1 and \mathbf{p}_2 . Notice that, since $n = 2$, $x = 1$ in case 2 of Theorem 6.1. Hence, the only difference in the simulations for the two cases of the theorem is that \mathbf{p}_2 is a covering simulator in case 1 and \mathbf{p}_2 is a direct simulator in case 2. In both cases, \mathbf{p}_1 is a covering simulator.

In general, the simulation guarantees that, for each *real* execution, α , of the real system, there exists a *corresponding simulated* execution, α , of the simulated system, such that, for each simulator \mathbf{p}_i , the state of each simulated process $p_{i,r} \in P_i$ stored by \mathbf{p}_i at the end of α is the same as the state of $p_{i,r}$ at the end of α . Moreover, if α contains no **output** steps, then the contents of \mathbf{R} at the end of α are the same as the contents of \mathbf{S} at the end of α . To ensure this holds when α is the empty execution, \mathbf{R} stores the initial contents of \mathbf{S} and each simulator \mathbf{p}_i stores the initial state of each process in P_i .

We now describe a direct simulator's algorithm, which is the same as in the general case of the simulation. At a high level, a direct simulator simply takes steps on behalf on its simulated process, without doing anything extra. More precisely, a direct simulator, \mathbf{p}_i , *directly* simulates its single process, $p_{i,1} \in P_i$, as follows. Recall that the first step of any process in a protocol is an **input** step, which it takes in order to be assigned an input. To *simulate* the input step of $p_{i,1}$, \mathbf{p}_i takes an **input** step and updates $p_{i,1}$'s state so that it is assigned the same input. Recall that each simulated process alternately performs scan and update, starting and ending with at least one scan, until it **outputs** a value. Hence, $p_{i,1}$ is now poised to perform a scan. To *simulate* a scan of \mathbf{S} , \mathbf{p}_i performs a SCAN of \mathbf{R} and updates the state of $p_{i,1}$ to reflect that it has taken its scan step, which returns the result of \mathbf{p}_i 's SCAN. If $p_{i,1}$

is poised to **output** some value y , then \mathbf{p}_i *simulates* this step by taking an **output**(y) step and updating the state of $p_{i,1}$ to reflect that it has taken its **output** step. Then \mathbf{p}_i terminates. Otherwise, $p_{i,1}$ is poised to perform an **update**(j, v) on \mathbf{S} , for some $j \in [m]$ and value v . To *simulate* this **update**, \mathbf{p}_i performs a BLOCK-UPDATE($(j), (v)$) on \mathbf{R} and updates the state of $p_{i,1}$ to reflect that it has taken its **update** step. (More precisely, the UPDATE(j, v) in the BLOCK-UPDATE *simulates* the **update**(j, v) by $p_{i,1}$.) At this point, $p_{i,1}$ is now poised to perform another **scan** and \mathbf{p}_i repeats the same sequence of steps. The pseudocode appears in Algorithm 6.1.

```

1: simulate  $p_{i,1}$ 's input step by taking an input step
   and update its state (so it has the same input as  $\mathbf{p}_i$ )
2: loop
3:   simulate  $p_{i,1}$ 's next step, a scan, using SCAN and update its state
4:   if  $p_{i,1}$  is poised to output some value  $y$  then
5:     simulate  $p_{i,1}$ 's next step by taking an output( $y$ ) step, update its state, and terminate
6:   simulate  $p_{i,1}$ 's next step, an update( $j, v$ ), using BLOCK-UPDATE( $(j), (v)$ ) and update its state

```

Algorithm 6.1: Pseudocode for a direct simulator \mathbf{p}_i .

We note the following properties of the direct simulator's algorithm, which follows from our assumption on how processes take steps in Π and the fact that there is a one-to-one correspondence between the steps and operations performed by a direct simulator and its simulated process.

Observation 6.2. *After taking an **input** step, each direct simulator, \mathbf{p}_i , alternately performs SCAN and UPDATE, starting with at least one SCAN, until it performs a SCAN that causes it to **output** a value and terminate. The **input** and **output** steps of \mathbf{p}_i simulate input and output steps of $p_{i,1}$, respectively. Each UPDATE (part of a BLOCK-UPDATE to a single component) and SCAN operation performed by \mathbf{p}_i simulates an update and scan of $p_{i,1}$, respectively.*

If only direct simulators take steps in a real execution, α , then it is simple to describe the steps of the corresponding simulated execution, α . Recall that SCANS and UPDATES that are part of BLOCK-UPDATES are linearizable. Let σ be the sequence of operations (on \mathbf{R}) linearized in α and let σ be the sequence of operations (on \mathbf{S}) obtained by replacing each operation in σ (by a direct simulator) with the step (of its simulated process) that it simulates. Then α begins with the **input** steps simulated by the direct simulators, followed by σ , followed by the **output** steps simulated by the direct simulators (if any).

We now describe the algorithm of a covering simulator in our simple case. At a high level, a covering simulator, \mathbf{p}_i , attempts to simulate its processes, $p_{i,1}$ and $p_{i,2}$, so that they are poised to perform **updates** to different components of \mathbf{S} , i.e. it attempts to construct a *block update* by $\{p_{i,1}, p_{i,2}\}$. To do so, \mathbf{p}_i begins by taking an **input** step to *simulate* the **input** steps of $p_{i,1}$ and $p_{i,2}$ and it updates their states so that they are assigned the same input. Then, \mathbf{p}_i directly simulates the **scan** and **update** steps of $p_{i,1}$, exactly like a direct simulator, until $p_{i,1}$ is poised to **update** some component to which \mathbf{p}_i has previously performed an *atomic* BLOCK-UPDATE (or \mathbf{p}_i simulates an **output** step by $p_{i,1}$, in which case it **outputs** the same value and terminates).

Suppose that $p_{i,1}$ is poised to perform an **update** to a component to which \mathbf{p}_i has previously performed an atomic BLOCK-UPDATE, \mathbf{B} . Then, instead of simulating $p_{i,1}$'s **update**, \mathbf{p}_i *revises the past* of $p_{i,2}$ using \mathbf{B} as follows. First, \mathbf{p}_i considers any *reachable* configuration, C , of Π in which the contents of \mathbf{S} are the same as the view returned by \mathbf{B} and $p_{i,2}$ has the state that \mathbf{p}_i currently stores (i.e. its initial state). Observe that such a configuration exists since \mathbf{B} returns the contents of \mathbf{R} at some point in the real

execution, which is the same as the contents of S at the corresponding point in the simulated execution. Then p_i *locally* computes the longest prefix, ζ , of $p_{i,2}$'s solo execution from C that contains only **scans** and **updates** to the component updated by B . Finally, p_i updates the state of $p_{i,2}$ to its state at the end of ζ and we say that p_i has *locally simulated* the additional steps, ζ , of $p_{i,2}$. We emphasize that p_i does not take any shared memory steps during its revision of $p_{i,2}$'s state. This is one of the key ideas of the simulation.

As defined in the preliminaries, after a process takes a step in the real execution, it immediately transitions to a new state, which corresponds to performing all of its local computation that immediately follows the step. However, it will be useful to distinguish p_i 's revision of the $p_{i,2}$'s past from its other local computation. To do so, we introduce an additional type of step, **revise**(B), which p_i may take in a real execution to revise the past of $p_{i,2}$ using an atomic BLOCK-UPDATE, B , to a single component. Like an **input** or **output** step, a **revise** step does not modify the contents of R and only affects the state of p_i . In particular, a **revise**(B) step by p_i only affects the state it stores for $p_{i,2}$.

After p_i revises the past of $p_{i,2}$ using B , i.e. after it takes a **revise**(B) step, $p_{i,2}$ is poised to either **output** a value or **update** a component that is not updated by B . In the first case, p_i *simulates* the **output** step of $p_{i,2}$ by taking an **output** step with the same value and updating its state. Then p_i terminates. In the second case, we say that p_i has *constructed* a block update by $\{p_{i,1}, p_{i,2}\}$.

Finally, suppose that p_i has constructed a block update, β , by $\{p_{i,1}, p_{i,2}\}$. Let C be any reachable configuration of Π in which $p_{i,1}$ and $p_{i,2}$ have the states stored by p_i . (The proof of Lemma 6.6 shows that such a configuration exists.) Let ξ be the solo execution of $p_{i,1}$ from $C\beta$, in which it **outputs** a value, y . Then p_i *simulates* $\beta\xi$ by taking an **output**(y) step and updating the states of $p_{i,1}$ and $p_{i,2}$ to reflect that the steps $\beta\xi$ have occurred. Then p_i terminates. Observe that, since β overwrites all components of S , the steps $\beta\xi$ can occur at any point in the future. In fact, they will occur at the end of the final simulated execution.

We now sketch why the simulation is wait-free and correct. In our simple case, a covering simulator directly simulates its first process until (it terminates or) it takes a **revise** step, after which it **outputs** a value and terminates. Since the implementation of R is non-blocking, as the simulators take more steps, they simulate more and more steps of their simulated processes. Recall that R has only $m = 2$ components. Hence, eventually either both simulators terminate or some covering simulator, p_i , will simulate $p_{i,1}$ so that it is poised to **update** some component to which p_i has previously performed an atomic BLOCK-UPDATE. Then p_i takes a **revise** step, followed by an **output** step. After the **output** step, only the other simulator is simulating steps. Since Π is obstruction-free, this simulator must eventually **output** a value and terminate as well. It follows that the simulation is wait-free.

Now consider a real execution, α , in which both simulators have terminated. Let σ be the sequence of operations linearized in α . There are only finitely many atomic BLOCK-UPDATES B_1, \dots, B_ℓ in α . Hence, by Observation 5.4, we may write σ as $\alpha_1\gamma_1\beta_1 \cdots \alpha_\ell\gamma_\ell\beta_\ell\alpha_{\ell+1}$, where $\alpha_{\ell+1}$ contains all operations linearized after the last atomic BLOCK-UPDATE, B_ℓ , and, for each $j \in [\ell]$,

- B_j returns the contents of R immediately following $\alpha_1\gamma_1\beta_1 \cdots \alpha_{j-1}\gamma_{j-1}\beta_{j-1}\alpha_j$,
- β_j contains the UPDATE in B_j , and
- the only operations that γ_j contains are UPDATE operations in non-atomic BLOCK-UPDATE operations by the simulator that did not perform B_j . In particular, γ_j does not contain any SCAN operations.

Let $\sigma = \alpha_1\gamma_1\beta_1 \cdots \alpha_\ell\gamma_\ell\beta_\ell\alpha_{\ell+1}$ be the sequence of steps obtained by replacing each SCAN and UPDATE operation in σ with the step (of a simulated process) that it simulates, so that $\alpha_j, \gamma_j, \beta_j$ consists of the steps simulated by $\alpha_j, \gamma_j, \beta_j$, respectively. Then σ almost defines an execution of Π ; it is only missing the steps that are simulated using **input**, **output**, and **revise** steps.

First, we place all **input** steps simulated by **input** steps at the beginning of α_1 . Now consider the first **revise** step, δ , in α . Let p_i be the covering simulator that performed δ , which revised the past of $p_{i,2}$ using some atomic BLOCK-UPDATE B , and let ζ be the steps of $p_{i,2}$ locally simulated by p_i in δ . By definition, ζ only contains scans and updates to the component updated by B . Since B is an atomic BLOCK-UPDATE, $B = B_j$ for some $j \in [\ell]$, and B returns the contents of R immediately following $\alpha_1\gamma_1\beta_1 \cdots \alpha_{j-1}\gamma_{j-1}\beta_{j-1}\alpha_j$, which are the same as contents of S immediately following $\alpha_1\gamma_1\beta_1 \cdots \alpha_{j-1}\gamma_{j-1}\beta_{j-1}\alpha_j$. Hence, ζ is a valid solo execution of $p_{i,2}$ immediately following $\alpha_1\gamma_1\beta_1 \cdots \alpha_{j-1}\gamma_{j-1}\beta_{j-1}\alpha_j$. We insert ζ immediately after α_j in σ , i.e. the block $\alpha_j\gamma_j\beta_j$ in σ now becomes $\alpha_j\zeta\gamma_j\beta_j$. Since $\gamma_j\beta_j$ only consists of updates and β_j is an update to the component updated by B , ζ is hidden from all other simulated processes. It follows that the rest of the steps in σ are applicable. We handle the second **revise** step in α (if any) in a similar way. Note that this **revise** step uses an atomic BLOCK-UPDATE $B' \neq B$. Hence, when we insert the steps locally simulated in **revise**(B') into the simulated execution, they are also hidden from all other simulated processes. In general, a **revise** step has the effect of inserting hidden steps in the simulated execution. Finally, for each **output** step, we insert the step(s) simulated by the step at the end of $\alpha_{\ell+1}$. Let α be the simulated execution after we have inserted all hidden steps simulated by **revise** steps as well as the steps simulated by **input** and **output** steps. It can be shown that the real execution, α , corresponds to α . Since each **output** value in α is the output value of some simulated process in α , the simulation is correct. (This is where we use the fact that the task is colourless.)

6.4 Covering Simulators

In general, a covering simulator, p_i , attempts to simulate its $m \geq 2$ processes in P_i so that they are all about to update a different component of S , i.e. p_i is attempting to construct a *block update* by P_i . Each covering simulator, p_i , begins by taking an **input** step to *simulate* the input step of each process in P_i and updates their states so that they are assigned the same input. Then p_i attempts to construct a block update by P_i . We define a recursive procedure, $\text{CONSTRUCT}(r)$, which describes how p_i attempts to construct a block update by $\{p_{i,1}, \dots, p_{i,r}\}$, for each $r \in [m]$.

As a base case, we describe $\text{CONSTRUCT}(1)$, in which p_i *attempts to construct a block update* by $\{p_{i,1}\}$. We will ensure that $p_{i,1}$ is poised to perform a **scan** whenever p_i attempts to construct a block update by $\{p_{i,1}\}$. To *simulate* this **scan** of S by $p_{i,1}$, p_i performs a SCAN of R and updates the state of $p_{i,1}$ to reflect that it has taken its **scan** step, which returns the result of p_i 's SCAN. If $p_{i,1}$ is now poised to **output** a value, then p_i *simulates* the **output** step of $p_{i,1}$ by taking an **output** step with the same value and updates $p_{i,1}$'s state. Then p_i terminates. Otherwise, $p_{i,1}$ is now poised to perform an **update** on S (since processes in Π alternately perform **scan** and **update**). In this case, we say that p_i *has constructed* a block update by $\{p_{i,1}\}$.

For $r > 1$, each time p_i calls $\text{CONSTRUCT}(r)$, to *attempt to construct a block update* by $\{p_{i,1}, \dots, p_{i,r}\}$, it keeps track of a set, A , of *atomic* BLOCK-UPDATES to $r-1$ components that it has performed during its attempt. It initializes A to be the empty set. Then p_i recursively attempts to construct a block update

```

function CONSTRUCT( $r$ )
  ▷ Assumes  $p_{i,1}$  is poised to perform a scan
  ▷ Returns a block update by  $p_{i,1}, \dots, p_{i,r}$ , represented as a pair  $((j_1, \dots, j_r), (v_1, \dots, v_r))$ ,
  where  $p_{i,g}$  is poised to perform update( $j_g, v_g$ ), for all  $g \in [r]$ .
1: if  $r = 1$  then ▷ base case
2:   simulate  $p_{i,1}$ 's next step (a scan) using SCAN and update its state
3:   if  $p_{i,1}$  is poised to perform output( $y$ ) for some value  $y$  then
4:     simulate  $p_{i,1}$ 's output( $y$ ) step with output( $y$ ), update its state, and terminate
5:   else ▷  $p_{i,1}$  is poised to perform some update( $j, v$ )
6:     return  $((j), (v))$ 
7: else ▷  $1 < r \leq m$ 
8:    $A \leftarrow \emptyset$  ▷  $A$  contains atomic BLOCK-UPDATES
9:   loop
10:     $((j_1, \dots, j_{r-1}), (v_1, \dots, v_{r-1})) \leftarrow \text{CONSTRUCT}(r - 1)$ 
11:    if there exists BLOCK-UPDATE  $B \in A$  such that  $B$  updates  $\{j_1, \dots, j_r\}$  then
12:      revise( $B$ ) ▷ revise past of  $p_{i,r}$  using  $B$ 
13:      if  $p_{i,r}$  is poised to perform output( $y$ ) for some value  $y$  then
14:        simulate  $p_{i,r}$ 's output( $y$ ) step with output( $y$ ), update its state, and terminate
15:      else ▷  $p_{i,r}$  is poised to perform some update( $j_r, v_r$ )
16:        return  $((j_1, \dots, j_r), (v_1, \dots, v_r))$ 
17:      else
18:        simulate  $p_{i,1}, \dots, p_{i,r-1}$ 's next steps using
          BLOCK-UPDATE( $((j_1, \dots, j_{r-1}), (v_1, \dots, v_{r-1}))$ ) and update their states
19:        if this BLOCK-UPDATE,  $B$ , is atomic then
20:           $A \leftarrow A \cup \{B\}$ 

```

Algorithm 6.2: Pseudocode for covering simulator, p_i , to construct a block update by $\{p_{i,1}, \dots, p_{i,r}\}$.

by $\{p_{i,1}, \dots, p_{i,r-1}\}$. If p_i terminates in this recursive attempt, then some process in $\{p_{i,1}, \dots, p_{i,r-1}\}$ has output a value and p_i has **output** the same value. Otherwise, at the end of this recursive attempt, $p_{i,1}, \dots, p_{i,r-1}$ are poised to perform **update**(j_1, v_1), \dots , **update**(j_{r-1}, v_{r-1}), where j_1, \dots, j_{r-1} are distinct components and v_1, \dots, v_{r-1} are some values.

If A does not contain a BLOCK-UPDATE to $\{j_1, \dots, j_{r-1}\}$, then p_i simulates the next steps of $p_{i,1}, \dots, p_{i,r-1}$ by performing a BLOCK-UPDATE($((j_1, \dots, j_{r-1}), (v_1, \dots, v_{r-1}))$) on \mathbf{R} and updates their states to reflect that their updates have occurred. In particular, for each $g \in [r - 1]$, the UPDATE(j_g, v_g) in the BLOCK-UPDATE *simulates* the **update**(j_g, v_g) of $p_{i,g}$. Recall that a BLOCK-UPDATE by p_i , for $i > 1$, may not always be atomic, i.e. it may return ∇ . However, the implementation of \mathbf{R} guarantees that the individual UPDATES that are part of a BLOCK-UPDATE are linearizable. Hence, p_i is guaranteed that the next steps by $p_{1,1}, \dots, p_{1,r-1}$ have all been simulated, though not necessarily consecutively. If the BLOCK-UPDATE, B , does not return ∇ , then p_i has simulated a block update by $\{p_{i,1}, \dots, p_{i,r-1}\}$ and it adds B to A . Regardless of what B returns, p_i then recursively attempts to construct another block update by $\{p_{i,1}, \dots, p_{i,r-1}\}$.

Now suppose A contains a BLOCK-UPDATE, B , to $\{j_1, \dots, j_{r-1}\}$. Then p_i *revises the past of* $p_{i,r}$ using B as follows. First, p_i considers any *reachable* configuration, C , of Π in which the contents of S are the same as the view returned by B and $p_{i,r}$ has the state that p_i currently stores. (The proof of Lemma 6.6 shows that such a configuration exists.) Then p_i *locally* computes the longest prefix, ζ , of $p_{i,r}$'s solo execution from C that contains only **scans** and **updates** to the components updated by B . Finally, p_i updates the state of $p_{i,r}$ to its state at the end of ζ and we say that p_i has *locally simulated*

the additional steps, ζ , of $p_{i,r}$.

As in Section 6.3, we let **revise**(\mathbf{B}) be the step that \mathbf{p}_i takes in a real execution to revise the past of $p_{i,r}$ using an atomic BLOCK-UPDATE, \mathbf{B} , to $r - 1$ components. Note that a **revise**(\mathbf{B}) step by \mathbf{p}_i is performed locally and only affects the state it stores for $p_{i,r}$.

After \mathbf{p}_i revises the past of $p_{i,r}$ using \mathbf{B} , i.e. after it takes a **revise**(\mathbf{B}) step, $p_{i,r}$ is poised to either output a value or update a component not in $\{j_1, \dots, j_{r-1}\}$. In the first case, \mathbf{p}_i *simulates* the output step of $p_{i,r}$, by taking an **output** step with the same value and updates the state of $p_{i,r}$. Then \mathbf{p}_i terminates. In the second case, we say that \mathbf{p}_i *has constructed* a block update by $\{p_{i,1}, \dots, p_{i,r}\}$. Algorithm 6.2 contains the pseudocode.

To conclude the description of \mathbf{p}_i 's simulation procedure, suppose that \mathbf{p}_i returns from **CONSTRUCT**(m), where it has constructed a block update, β , by P_i . Let C be any reachable configuration of Π in which each process in P_i has the state stored by \mathbf{p}_i . (The proof of Lemma 6.6 shows that such a configuration exists.) Let ξ be the solo execution of $p_{i,1}$ from $C\beta$ in which it **outputs** a value, y . Then \mathbf{p}_i *simulates* $\beta\xi$ by taking an **output**(y) step, updates the states of P_i to reflect that the steps $\beta\xi$ have occurred, and terminates. Algorithm 6.3 contains the pseudocode for the algorithm of a covering simulator.

- 1: simulate $p_{i,1}, \dots, p_{i,m}$'s input steps with **input**
and update their states so they are assigned same input
- 2: $\beta \leftarrow \mathbf{CONSTRUCT}(m)$ \triangleright note \mathbf{p}_i may terminate during the call
- 3: let ξ be $p_{i,1}$'s terminating solo execution after β in which it outputs a value y
- 4: simulate $\beta\xi$ with **output**(y), update P_i 's states to reflect $\beta\xi$, and **terminate**

Algorithm 6.3: Pseudocode for a covering simulator, \mathbf{p}_i .

We now prove important properties of the covering simulator's algorithm.

Lemma 6.3. *Consider a call to **CONSTRUCT**(r), for some $r \in [m]$, by a covering simulator, \mathbf{p}_i . Suppose $p_{i,1}$ is poised to perform **scan** when it is in the state stored by \mathbf{p}_i immediately before the call.*

1. *During the call, \mathbf{p}_i alternately performs **SCAN** and **BLOCK-UPDATE**, starting with at least one **SCAN**. Each **SCAN** simulates a **scan** by $p_{i,1}$ and each **BLOCK-UPDATE** simulates updates by $p_{i,1}, \dots, p_{i,s}$ to different components, for some $s \in [r - 1]$.*
2. *During the call, \mathbf{p}_i does not take any **revise** steps to revise the past of $p_{i,s}$, for $s > r$. If \mathbf{p}_i takes a **revise** step to revise the past of $p_{i,r}$ during the call, then it does so after the last recursive call to **CONSTRUCT**($r - 1$) in the call returns. After taking a **revise** step, \mathbf{p}_i either **outputs** a value (and terminates) or it returns.*
3. *If \mathbf{p}_i **outputs** a value and terminates during the call, then its **output** step simulates an **output** step by $p_{i,s}$ with the same value, for some $s \in [r]$. Otherwise, $p_{i,1}, \dots, p_{i,r}$ are poised to perform updates to different components in their states stored by \mathbf{p}_i when it returns from the call. In both cases, the last operation \mathbf{p}_i performed in the call was a **SCAN**.*

Proof. By induction on r . The base case is when $r = 1$. Observe that, in **CONSTRUCT**(1), \mathbf{p}_i performs a single **SCAN**, which simulates the **scan** that $p_{i,1}$ is poised to perform. It does not revise the past of any process. Thus parts 1 and 2 of the claim hold. Part 3 follows from lines 3–6.

Let $r > 1$ and suppose the claim holds for $r - 1$. Consider any call, \mathbf{Q} , to **CONSTRUCT**(r). We show that the following properties hold during each iteration of the loop in \mathbf{Q} (on lines 9–20).

- During the iteration, p_i alternately performs SCAN and BLOCK-UPDATE, starting with at least one SCAN. Each SCAN simulates a scan by $p_{i,1}$ and each BLOCK-UPDATE simulates updates by $p_{i,1}, \dots, p_{i,s}$ to different components, for some $s \in [r-1]$. If p_i finishes the iteration, but does not **output** a value (and terminate) or return from Q, then the last operation that p_i performs in the iteration is a BLOCK-UPDATE and $p_{i,1}$ is poised to perform scan at the start of the next iteration.
- During the iteration, p_i does not take any **revise** steps to revise the past of $p_{i,s}$, for $s > r$. If p_i takes a **revise** step that revises the past of $p_{i,r}$ during the iteration, then p_i either **outputs** a value (and terminates) or returns from Q in the iteration and the **revise** step is the last step it takes before doing so.
- If p_i **outputs** a value (and terminates) in the iteration, then its **output** step simulates an output step by $p_{i,s}$ with the same value, for some $s \in [r]$. If p_i returns from Q in the iteration, then $p_{i,1}, \dots, p_{i,r}$ are poised to perform updates to different components in their states stored by p_i when it returns from Q. In both cases, the last operation p_i that performs is a SCAN.

Since $p_{i,1}$ is poised to perform scan immediately before Q, it is poised to perform scan at the start of the first iteration. Now suppose that $p_{i,1}$ is poised to perform scan at the start of some iteration. Then the induction hypothesis applies to p_i 's recursive call, Q' , to CONSTRUCT($r-1$) (on line 10) in the iteration. By the induction hypothesis, during Q' , p_i alternately performs SCAN and BLOCK-UPDATE, starting with at least one SCAN. Each SCAN simulates a scan by $p_{i,1}$ and each BLOCK-UPDATE simulates updates by $p_{i,1}, \dots, p_{i,s}$ to different components, for some $s \in [r-2]$. Moreover, during Q' , p_i does not take any **revise** steps to revise the past of $p_{i,s}$, for $s > r-1$. Finally, if p_i **outputs** a value and terminates during Q' , then its **output** step simulates an output step by $p_{i,s}$ with the same value, for some $s \in [r-1]$. Otherwise, $p_{i,1}, \dots, p_{i,r-1}$ are poised to perform updates to different components in their states stored by p_i when it returns from Q' . In either case, the last operation p_i performed in Q' was a SCAN. If p_i **outputs** a value (and terminates) in Q' , then the desired properties hold for the iteration. Now suppose p_i returns from Q' . We consider two cases.

Suppose p_i takes a **revise** step (on line 12) to revise the past of $p_{i,r}$ immediately after Q' returns. Then p_i either **outputs** a value (and terminates on line 14) or returns from Q immediately after this step (on line 16). In the second case, by line 11 and the definition of a **revise** step, $p_{i,1}, \dots, p_{i,r}$ are poised to perform updates to different components (immediately after line 12). In either case, the last operation p_i performed was in Q' , which was a SCAN. Hence, the desired properties hold for the iteration.

Now suppose p_i does not take a **revise** step to revise the past of $p_{i,r}$ immediately after Q' returns. Then the next operation that p_i performs in Q is a BLOCK-UPDATE (on line 18), to simulate the updates (to different components) that $p_{i,1}, \dots, p_{i,r-1}$ were poised to perform as a result of Q' . The iteration ends after this BLOCK-UPDATE and $p_{i,1}$ is poised to perform scan at the start of the next iteration. Hence, the desired properties hold for the iteration.

To conclude the proof, notice that the first, second, and third properties of each iteration implies parts 1, 2, and 3 of the claim for Q. \square

The following corollary describes the main properties of the covering simulator's procedure. It follows from the preceding lemma because the first process, $p_{i,1}$, of a covering simulator, p_i , is poised to perform scan immediately before p_i 's call to CONSTRUCT(m). Notice that, in contrast to the properties of a direct simulator (in Observation 6.2), a step by p_i may simulate a sequence of steps by its processes, a BLOCK-UPDATE operation by p_i may update multiple components, and p_i may take **revise** steps.

Corollary 6.4. *After taking an **input** step, each covering simulator, p_i , alternately performs SCAN (followed by at most $m - 1$ **revise** steps) and BLOCK-UPDATE until it performs a SCAN that causes it to **output** a value and terminate. The **input** step of p_i simulates the **input** steps of each process in P_i . The **output** step of p_i either simulates the **output** step of a single process in P_i or a block update by P_i , followed by the solo-terminating execution of $p_{i,1}$. Each SCAN operation performed by p_i simulates a scan of $p_{i,1}$. The UPDATES that are part of a BLOCK-UPDATE to r components simulate updates by $p_{i,1}, \dots, p_{i,r}$ to different components.*

6.5 The Intermediate Execution of a Real Execution

Recall that \mathbf{R} is implemented from a single-writer atomic snapshot object. Moreover, the implementation is *not* linearizable, so we cannot simply assume operations on \mathbf{R} are atomic. Hence, technically, each shared memory step in a real execution is an operation on the underlying single-writer snapshot object used to implement \mathbf{R} , which is a rather low-level view.

In this section, we define the *intermediate execution*, σ , of a real execution, which abstracts away the details of the implementation of \mathbf{R} and focuses on the details relevant to the simulation, i.e. the operations on \mathbf{R} that the simulators perform, the **input**, **output**, and **revise** steps that the simulators take (if any), and the states of their simulated processes after each such step or operation.

Each *step* in σ is performed by a simulator. A *shared memory step* in σ is either a SCAN or UPDATE operation on \mathbf{R} . The remaining steps are either **input**, **output**, or **revise** steps. The sequence of steps in σ is obtained from the real execution as follows. Recall that each SCAN and UPDATE that is part of a BLOCK-UPDATE can be linearized at some point in the real execution. Let $\delta_1, \delta_2, \dots$ be the sequence of (SCAN and UPDATE) operations on \mathbf{R} linearized in the real execution. Then the sequence of steps in the intermediate execution consists of $\delta_1, \delta_2, \dots$, together with the **input**, **output**, and **revise** steps in the real execution. The operations on \mathbf{R} in σ occur in order of their linearization points in the real execution. The **input**, **output**, and **revise** steps in σ occur in the same order as in the real execution. An **input**, **output**, or **revise** step in σ occurs before a shared memory step δ_j in σ if and only if it occurs before the linearization point of δ_j in the real execution.

Each *configuration* in σ consists of the contents of \mathbf{R} as well as the state of each simulated process. We inductively define each configuration in σ as follows. At the *initial* configuration of σ , \mathbf{R} contains the initial contents of \mathbf{S} and each simulated process is in its initial state. Now suppose \mathbf{C} is a configuration in σ that we have defined and consider the configuration, \mathbf{C}' , after \mathbf{C} in σ . Let δ be the step between \mathbf{C} and \mathbf{C}' in σ and let p_i be the simulator that performed δ . We consider two cases.

Case 1: δ is an **input**, **output**, or **revise** step. Then the contents of \mathbf{R} are the same at \mathbf{C} and \mathbf{C}' . Suppose that δ simulates a sequence of steps, δ , of a set of processes $P \subseteq P_i$. Then the state of each process in P at \mathbf{C}' is the state of that process stored by p_i immediately after δ in the real execution, to reflect that it has taken its additional steps in δ . The states of the other simulated processes are the same at \mathbf{C} and \mathbf{C}' .

Case 2: δ is a SCAN or UPDATE operation on \mathbf{R} . By Observation 6.2 and Corollary 6.4, δ simulates a step, δ , of a single process $p_{i,r} \in P_i$. The states of the other simulated processes are the same at \mathbf{C} and \mathbf{C}' . If δ is an UPDATE(j, v), then the contents of \mathbf{R} are the same at \mathbf{C}' and \mathbf{C} , except component j of \mathbf{R} has value v . Let \mathbf{B} be the BLOCK-UPDATE that δ is part of. Then the state of $p_{i,r}$ at \mathbf{C}' is the state stored by p_i immediately after \mathbf{B} returns in the real execution, to reflect that its update, δ , occurred.

Now suppose δ is a SCAN. Then $r = 1$ and the contents of \mathbf{R} are the same at \mathbf{C}' and \mathbf{C} . The state of $p_{i,1}$ at \mathbf{C}' is the state stored by \mathbf{p}_i immediately after δ returns in the real execution, to reflect that its scan, δ , occurred and returned the same view as the SCAN.

Observe that the intermediate execution, σ , is neither an execution of the real system nor an execution of the simulated system. This is because the operations in σ are performed by the simulators, while the configurations in σ contain the states of the simulated processes. Also, in σ , the state of a process $p_{i,r}$ is updated immediately after its update step is simulated by an UPDATE operation. In a real execution, its simulator only updates its state after the BLOCK-UPDATE containing the UPDATE completes.

By Algorithms 6.1, 6.2, and 6.3, a simulator only simulates a step by a process if that process was poised to take that step in the state stored by the simulator.

Observation 6.5. *Suppose δ is a step or operation in the intermediate execution, σ , by a simulator, \mathbf{p}_i , that simulates a sequence of steps, δ , of a set of processes $P \subseteq P_i$. Then each process in P is poised to take its first step in δ at the configuration in σ immediately before δ .*

By Observation 5.4, if there are only finitely many atomic BLOCK-UPDATE operations, $\mathbf{B}_1, \dots, \mathbf{B}_\ell$, in the real execution, then the sequence of UPDATES and SCANS linearized in the real execution may be written as $\alpha_1\gamma_1\beta_1 \cdots \alpha_\ell\gamma_\ell\beta_\ell\alpha_{\ell+1}$, where $\alpha_{\ell+1}$ contains all operations linearized after the last atomic BLOCK-UPDATE operation and, for each $j \in [\ell]$,

- \mathbf{B}_j returns the contents of \mathbf{R} immediately following $\alpha_1\gamma_1\beta_1 \cdots \alpha_{j-1}\gamma_{j-1}\beta_{j-1}\alpha_j$,
- β_j consists of the UPDATES that are part of \mathbf{B}_j , and
- the only operations that γ_j contains are UPDATE operations that are part of non-atomic BLOCK-UPDATE operations by simulators other than the simulator that performed \mathbf{B}_j . In particular, γ_j does not contain any SCAN operations.

By definition, the sequence of UPDATES and SCANS linearized in the real execution is precisely the sequence of operations in σ . We call $\alpha_1\gamma_1\beta_1 \cdots \alpha_\ell\gamma_\ell\beta_\ell\alpha_{\ell+1}$ the *block decomposition* of the operations in σ and, for $j \in [\ell]$, we call $\alpha_j\gamma_j\beta_j$ the *j 'th block* in the decomposition.

6.6 Correctness of the Simulation

In this section, we state and prove the main properties of our simulation and use them to prove that our simulation solves the same colourless task as the protocol Π . Roughly, our invariants say that, for each intermediate execution (of a real execution), there is a corresponding (simulated) execution of Π such that the state of each process $p_{i,g} \in P_i$ at the end of the simulated execution is the same as the state of $p_{i,g}$ at the end of the intermediate execution. By definition of the intermediate execution, this is the state of $p_{i,g}$ stored by \mathbf{p}_i at the end of the real execution, provided \mathbf{p}_i has no pending operation. The actual properties are more complicated because we need to know the exact structure of the simulated execution in order to describe where the hidden steps of simulated processes are inserted.

Lemma 6.6. *Let σ be the intermediate execution of a real execution from the initial configuration of the real system, let $\mathbf{B}_1, \dots, \mathbf{B}_\ell$ be the sequence of atomic BLOCK-UPDATES in σ , and let $\alpha_1\gamma_1\beta_1 \cdots \alpha_\ell\gamma_\ell\beta_\ell\alpha_{\ell+1}$ be the block decomposition of the operations in σ . Then there is a simulated execution, σ , from the initial configuration of the simulated system, which satisfies the following properties.*

1. The steps of σ may be written as $\iota \cdot \alpha_1 \zeta_1 \gamma_1 \beta_1 \cdots \alpha_\ell \zeta_\ell \gamma_\ell \beta_\ell \alpha_{\ell+1} \cdot \omega$ such that the following holds.
 - (a) ι consists of the sequence of **input** steps simulated by the **input** steps in σ .
 - (b) For each $t \in [\ell]$, α_t , γ_t , and β_t are obtained by replacing each **SCAN** and **UPDATE** operation in α_t , γ_t , and β_t , respectively, with the step that it simulates. If there is a **revise**(B_t) step in σ , then ζ_t consists of the steps locally simulated in this **revise** step; otherwise, ζ_t is empty.
 - (c) $\alpha_{\ell+1}$ is obtained by replacing each **SCAN** and **UPDATE** operation in $\alpha_{\ell+1}$ with the step that it simulates.
 - (d) ω is obtained by concatenating the sequence of steps simulated by each **output** step in σ .
2. For each $i \in [n]$, each process in P_i has the same state at the end of σ and σ' .
3. For each $t \in [\ell + 1]$, if $\hat{\alpha}_t$ is a prefix of α_t and $\hat{\alpha}_t$ is obtained by replacing each **SCAN** and **UPDATE** operation in $\hat{\alpha}_t$ with the step that it simulates, then the contents of \mathbf{R} in σ immediately after the last operation in $\alpha_1 \gamma_1 \beta_1 \cdots \alpha_{t-1} \gamma_{t-1} \beta_{t-1} \hat{\alpha}_t$ are the same as the contents of \mathbf{S} in σ immediately after the last step in $\iota \cdot \alpha_1 \zeta_1 \gamma_1 \beta_1 \cdots \alpha_{t-1} \zeta_{t-1} \gamma_{t-1} \beta_{t-1} \hat{\alpha}_t$.

Proof. By induction on the length of σ . The base case is when σ only contains the initial configuration. In this case, let σ be the simulated execution that only contains the initial configuration of the simulated system. Since there are no steps in σ , property 1 is vacuously true. In the initial configuration of an intermediate execution, each simulated process has its initial state and the contents of \mathbf{R} are the initial contents of \mathbf{S} . Hence, properties 2 and 3 hold. Therefore, σ witnesses the claim for σ .

Now suppose σ witnesses the claim for σ . Let $\alpha_1 \gamma_1 \beta_1 \cdots \alpha_\ell \gamma_\ell \beta_\ell \alpha_{\ell+1}$ be the block decomposition of the operations in σ . By part 1 of the induction hypothesis, the steps of σ may be written as $\iota \cdot \alpha_1 \zeta_1 \gamma_1 \beta_1 \cdots \alpha_\ell \zeta_\ell \gamma_\ell \beta_\ell \alpha_{\ell+1} \cdot \omega$ so that property 1 holds.

Consider $\sigma' = \sigma \delta'$, where δ' is a step or operation performed by a simulator, p_i . Let δ' be the sequence of steps simulated by δ' . By Observation 6.5, each process that takes a step in δ' is poised to take their first step in δ' at the configuration in σ' immediately before δ' , i.e. at the end of σ . Hence, by property 2 of the induction hypothesis, each such process is poised to its first step in δ' at the end of σ . To define a simulated execution, σ' , that witnesses the claim for σ' , we consider a number of cases. In each case, we will obtain σ' by inserting the additional simulated steps, δ' , somewhere in σ .

Case 1: δ' is an **input step.** By the properties of the simulators (Observation 6.2 and Corollary 6.4), δ' consists of the **input** step of each process in P_i , which it is poised to perform at the end of σ . Since **input** steps do not affect the contents of memory and σ does not contain any steps by P_i , we may insert these steps at the beginning of σ . Let $\iota' = \iota \cdot \delta'$ and let σ' be the simulated execution whose steps are $\iota' \cdot \alpha_1 \zeta_1 \gamma_1 \beta_1 \cdots \alpha_\ell \zeta_\ell \gamma_\ell \beta_\ell \alpha_{\ell+1} \cdot \omega$. Then the states of the processes in P_i at the end of σ' are the same as their states at the end of σ . The states of the other processes are unchanged. Since σ contains the same operations as σ' , the block decomposition of the operations in σ' is the same as that of σ . Therefore, σ' witnesses the claim for σ' .

Case 2: δ' is an **output step.** By the properties of the simulators (Observation 6.2 and Corollary 6.4), δ' either consists of an **output** step by a single process $p_{i,r} \in P_i$ or it consists of a block update, β , by P_i , followed by the solo execution, ξ , of $p_{i,1}$. In the first case, since $p_{i,r}$ is poised to perform δ' at the end of σ , we may insert δ' at the end of σ . In the second case, since P_i is poised to perform β at the end of σ and β overwrites the contents of \mathbf{S} , we may insert $\delta' = \beta \xi$ at the end of σ . In either case, let $\omega' = \omega \cdot \delta'$ and let σ' be the simulated execution whose steps are $\iota \cdot \alpha_1 \zeta_1 \gamma_1 \beta_1 \cdots \alpha_\ell \zeta_\ell \gamma_\ell \beta_\ell \alpha_{\ell+1} \cdot \omega'$. Then the states

of the processes in P_i at the end of σ' are the same as their states at the end of σ . The states of the other processes are unchanged. Since σ contains the same operations as σ' , the block decomposition of the operations in σ' is the same as that of σ . Therefore, σ' witnesses the claim for σ .

Case 3: δ' is an UPDATE or SCAN operation. By the properties of the simulators (Observation 6.2 and Corollary 6.4), δ' is either an update or scan of a single process, $p_{i,r} \in P_i$, and $p_{i,r}$ is poised to perform δ' at the end of σ . Note that, since p_i has not terminated in σ , ω does not contain any steps by $p_{i,r}$. Thus, $p_{i,r}$ is poised to perform δ' at the end of $\alpha_{\ell+1}$. We will insert δ' immediately following $\alpha_{\ell+1}$. By property 3 of the induction hypothesis, the contents of \mathbf{R} at the end of $\alpha_{\ell+1}$ are the same as the contents of \mathbf{S} at the end of $\alpha_{\ell+1}$. If δ' is a SCAN, then this implies that δ' and δ' return the same output. Otherwise, δ' and δ' update the same component with the same value. Define σ' to be the execution obtained from σ by adding the additional step, δ , to the end of $\alpha_{\ell+1}$. Then the state of $p_{i,r}$ at the end of σ' is the same as the state of $p_{i,r}$ at the end of σ . The states of all other processes are unchanged. Hence, σ' is a possible execution of Π and property 2 holds for σ' . To show that properties 1 and 3 hold, we consider the block decomposition of σ' .

Suppose σ' and σ contains the same number of blocks in their block decompositions. Then the block decomposition of σ' is the same as that of σ , except with δ' appended to the end of $\alpha_{\ell+1}$. Observe that we may write the steps of σ' as $\iota \cdot \alpha_1 \zeta_1 \gamma_1 \beta_1 \cdots \alpha_\ell \zeta_\ell \gamma_\ell \beta_\ell \alpha'_{\ell+1} \cdot \omega$, where $\alpha'_{\ell+1} = \alpha_{\ell+1} \delta'$. The definition of $\alpha'_{\ell+1}$ and property 1c of the induction hypothesis applied to σ imply that property 1c holds for σ' . Properties 1a, 1b, and 1d are unaffected because they only refer to the parts of σ and σ that are unchanged. As discussed above, the contents of \mathbf{R} after $\hat{\alpha}_t$ in σ' and the contents of \mathbf{S} after $\hat{\alpha}_t$ in σ' are the same when $t = \ell + 1$ and $\hat{\alpha}_t = \alpha'_{\ell+1}$. Property 3 of the induction hypothesis applied to σ implies that property 3 holds for the remaining choices of $\hat{\alpha}_t$.

Now suppose σ' and σ do not contain the same number of blocks in their block decompositions. Then δ' is the last UPDATE that is part of an atomic BLOCK-UPDATE \mathbf{B} by p_i in σ' and there are $\ell + 1$ atomic BLOCK-UPDATES in σ' , namely, $\mathbf{B}_1, \dots, \mathbf{B}_\ell$ and $\mathbf{B}_{\ell+1} = \mathbf{B}$. By definition of a block decomposition, the first ℓ blocks in the block decomposition of σ' are the same as first ℓ blocks in the block decomposition of σ . However, σ' contains an additional block, $\hat{\alpha}_{\ell+1} \gamma_{\ell+1} \beta_{\ell+1}$, in its decomposition where

- $\hat{\alpha}_{\ell+1} \gamma_{\ell+1}$ is a prefix of $\alpha_{\ell+1}$,
- $\mathbf{B}_{\ell+1}$ returns the contents of \mathbf{R} at $\alpha_1 \gamma_1 \beta_1 \cdots \alpha_\ell \gamma_\ell \beta_\ell \hat{\alpha}_{\ell+1}$,
- $\gamma_{\ell+1}$ only contains UPDATE operations that are part of non-atomic BLOCK-UPDATE operations by simulators other than p_i (the simulator that performed $\mathbf{B}_{\ell+1}$), and
- $\beta_{\ell+1}$ consists of the UPDATES that are part of $\mathbf{B}_{\ell+1}$.

In particular, $\beta_{\ell+1}$ consists of the UPDATES in $\alpha_{\ell+1}$ after $\hat{\alpha}_{\ell+1} \gamma_{\ell+1}$, followed by δ' .

Let $\hat{\alpha}_{\ell+1}$, $\gamma_{\ell+1}$, and $\beta_{\ell+1}$ be obtained by replacing each UPDATE and SCAN operation in $\hat{\alpha}_{\ell+1}$, $\gamma_{\ell+1}$, and $\beta_{\ell+1}$ with the step that it simulates, and let both $\zeta_{\ell+1}$ and $\alpha_{\ell+2}$ be empty. Then the steps of σ' may be written as $\iota \cdot \alpha_1 \zeta_1 \gamma_1 \beta_1 \cdots \alpha_\ell \zeta_\ell \gamma_\ell \beta_\ell \hat{\alpha}_{\ell+1} \zeta_{\ell+1} \gamma_{\ell+1} \beta_{\ell+1} \alpha_{\ell+2} \cdot \omega$. By definition of $\hat{\alpha}_{\ell+1}$, $\gamma_{\ell+1}$, $\beta_{\ell+1}$, $\zeta_{\ell+1}$, and $\alpha_{\ell+2}$ and the fact that property 1 holds for σ , property 1 holds for σ' . Since property 3 holds for σ , property 3 holds for σ' for $t \in [\ell]$. Since every prefix of $\hat{\alpha}_{\ell+1}$ is a prefix of $\alpha_{\ell+1}$, property 3 holds for σ' if $t = \ell + 1$. The definition of δ' implies that property 3 holds if $t = \ell + 2$.

Therefore, σ' witnesses the claim for σ .

Case 4: δ' is a **revise**(\mathbf{B}_t) step, for some $t \in [\ell]$. Suppose that \mathbf{B}_t is a BLOCK-UPDATE (by \mathbf{p}_i) to r components. By definition, δ' consists of the steps of $p_{i,r+1}$ locally simulated by \mathbf{p}_i during δ' . Note that \mathbf{B}_t and δ' were performed during a call, \mathbf{Q} , to CONSTRUCT($r + 1$).

We claim that the operations in $\gamma_t\beta_t\alpha_{t+1}\gamma_{t+1}\beta_{t+1}\cdots\alpha_\ell\gamma_\ell\beta_\ell\alpha_{\ell+1}$ are all linearized during the execution interval of \mathbf{Q} . Since \mathbf{B}_t is an atomic BLOCK-UPDATE that was performed in \mathbf{Q} , the UPDATES, β_t , that comprise \mathbf{B}_t also occur in \mathbf{Q} . By Lemma 6.3.1, \mathbf{Q} begins with a SCAN. Since γ_t only contains UPDATES, this SCAN occurs before the first operation in γ_t . By Lemma 6.3.2, δ' is the last step taken by \mathbf{p}_i before it **outputs** a value (and terminates) or returns from \mathbf{Q} . Since δ' occurs after σ , it occurs after the last operation in $\alpha_{\ell+1}$.

By Lemma 6.3.1, \mathbf{p}_i does not perform any BLOCK-UPDATES to $r + 1$ or more components during \mathbf{Q} . It follows that $\gamma_t\beta_t\alpha_{t+1}\gamma_{t+1}\beta_{t+1}\cdots\alpha_\ell\gamma_\ell\beta_\ell\alpha_{\ell+1}$ does not contain any UPDATES that are part of a BLOCK-UPDATE by \mathbf{p}_i to $r + 1$ or more components. By Corollary 6.4, it follows that $\gamma_t\beta_t\alpha_{t+1}\gamma_{t+1}\beta_{t+1}\cdots\alpha_\ell\gamma_\ell\beta_\ell\alpha_{\ell+1}$ does not contain any operations that simulate a step by $p_{i,r+1}$. By property 1b of the induction hypothesis applied to σ , it follows that $\gamma_t\beta_t\alpha_{t+1}\gamma_{t+1}\beta_{t+1}\cdots\alpha_\ell\gamma_\ell\beta_\ell\alpha_{\ell+1}$ does not contain any steps by $p_{i,r+1}$.

By Lemma 6.3.2, during \mathbf{Q} , **revise**(\mathbf{B}_t) is the only **revise** step that \mathbf{p}_i takes to revise the past of $p_{i,r+1}$. Since \mathbf{B}_t was performed during \mathbf{Q} , any BLOCK-UPDATE in $\{\mathbf{B}_{t+1}, \dots, \mathbf{B}_\ell\}$ performed by \mathbf{p}_i was also performed during \mathbf{Q} . Hence, σ , which occurs before δ' , does not contain any **revise** steps by \mathbf{p}_i that revise the past of $p_{i,r+1}$ using an (atomic) BLOCK-UPDATE in $\{\mathbf{B}_t, \dots, \mathbf{B}_\ell\}$. By property 1b of the induction hypothesis applied to σ , it follows that that $\zeta_t, \dots, \zeta_\ell$ do not contain any steps by $p_{i,r+1}$ and ζ_t is empty.

Since \mathbf{p}_i takes δ' , none of its processes have taken an **output** step. Thus, by property 1d of σ , ω does not contain any steps by $p_{i,r+1}$, so $\gamma_t\beta_t\alpha_{t+1}\zeta_{t+1}\gamma_{t+1}\beta_{t+1}\cdots\alpha_\ell\zeta_\ell\gamma_\ell\beta_\ell\alpha_{\ell+1}\cdot\omega$ does not contain any steps by $p_{i,r+1}$. Hence, we may insert the steps in δ' immediately before γ_t in σ by setting $\zeta'_t = \delta'$. Let σ' be the execution obtained by σ by replacing ζ_t with ζ'_t . Then σ' witnesses the claim for σ' . \square

The preceding lemma shows that, once every operation completes in a real execution, the states of the processes stored by the simulators are the same as the states of the simulated processes at the end of the corresponding simulated execution. This allows us to show that every value output in the simulation is correct.

Lemma 6.7. *The simulation solves the same task as the protocol.*

Proof. Let T be the task that the protocol, Π , solves. Consider any real execution of the simulation from the initial configuration. Let σ be its intermediate execution. By Lemma 6.6, there is a possible execution σ of Π that satisfies the invariants for σ . Let I be the set of inputs to the simulators in the real execution and let O be the set of outputs in σ . By construction, each simulated process is assigned the input of its simulator, i.e. the set of inputs of the simulated processes in σ is I as well. Since Π is assumed to be a correct protocol solving task T and σ is a possible execution of Π , O is correct for I . Observe that, for each $i \in [n]$, exactly one process in P_i has **output** a value in σ and this is the value **output** by \mathbf{p}_i . Thus, the set of outputs of the simulators in the real execution is exactly O . It follows that the simulation solves the task T . \square

6.7 Wait-freedom of the simulation

We now prove that the simulation is wait-free. In particular, we will show that covering simulators are wait-free and we will prove a bound on the maximum number of steps a covering simulator needs to take before terminating. We will then show that, from any configuration, the direct simulators will all eventually terminate, provided the covering simulators don't take any more steps.

Note that there are at most $\binom{m}{r-1}$ different sets of $r-1$ components. This bounds the number of atomic BLOCK-UPDATES to $r-1$ components that a covering simulator performs before it constructs a block update to $r-1$ components to which it has previously performed an atomic BLOCK-UPDATE. After it does so, the simulator either terminates or returns from the call.

Observation 6.8. *For $1 < r \leq m$, during any call to $\text{CONSTRUCT}(r)$, at most $\binom{m}{r-1}$ atomic BLOCK-UPDATES to $r-1$ components are performed.*

The following recursively defined function, $a(r)$, is an upper bound on the number of BLOCK-UPDATES performed in a call to $\text{CONSTRUCT}(r)$, provided each BLOCK-UPDATE in the call is atomic.

$$a(r) = \begin{cases} 0 & \text{if } r = 1 \\ \left(\binom{m}{r-1} + 1\right) a(r-1) + \binom{m}{r-1} & \text{if } 1 < r \leq m. \end{cases}$$

It can be verified that $a(r) \leq \left(\binom{m}{m/2} + 1\right)^{r-1} - 1 \leq 2^{m(r-1)}$.

Lemma 6.9. *For $r \in [m]$, if every BLOCK-UPDATE performed by a covering simulator, \mathbf{p}_i , during a call to $\text{CONSTRUCT}(r)$ is atomic, then \mathbf{p}_i performs at most $a(r)$ BLOCK-UPDATES during the call.*

Proof. By induction on r . The base case is $r = 1$. The claim holds in this case since \mathbf{p}_i performs no BLOCK-UPDATES in $\text{CONSTRUCT}(1)$. Now let $r > 1$ and suppose the claim holds for $r-1$. Consider a call to $\text{CONSTRUCT}(r)$. By Observation 6.8, \mathbf{p}_i performs at most $\binom{m}{r-1}$ (atomic) BLOCK-UPDATES in the call, each to exactly $r-1$ components. Each of these BLOCK-UPDATES is immediately preceded by a recursive call by \mathbf{p}_i to $\text{CONSTRUCT}(r-1)$. Furthermore, after applying the last BLOCK-UPDATE, \mathbf{p}_i recursively calls $\text{CONSTRUCT}(r-1)$ once more. By the induction hypothesis, in each of the recursive calls to $\text{CONSTRUCT}(r-1)$, \mathbf{p}_i performs at most $a(r-1)$ BLOCK-UPDATES. It follows that \mathbf{p}_i performs at most $\left(\binom{m}{r-1} + 1\right)a(r-1) + \binom{m}{r-1} = a(r)$ BLOCK-UPDATES in $\text{CONSTRUCT}(r)$. \square

Recall that BLOCK-UPDATES by \mathbf{p}_i , for $i > 1$, are not always atomic. The following recursively defined function, $b(i)$, is an upper bound on the number of BLOCK-UPDATES performed by the i 'th covering simulator, \mathbf{p}_i , in any real execution. Observe that the bound increases as i increases.

$$b(i) = \begin{cases} a(m) & \text{if } i = 1 \\ \left(a(m-1) + 1\right) \sum_{j=1}^{i-1} b(j) + a(m) & \text{if } 1 < i \leq n. \end{cases}$$

It can be verified that $b(i) = a(m)(a(m-1) + 1)^{i-1} \leq a(m)^i \leq 2^{im(m-1)}$ for all $i \in [n]$.

Lemma 6.10. *\mathbf{p}_i performs at most $b(i)$ BLOCK-UPDATES in any real execution.*

Proof. By induction on i . The base case is $i = 1$. Since \mathbf{p}_1 has the smallest identifier, all of its BLOCK-UPDATES are atomic. Hence, by Lemma 6.9, \mathbf{p}_1 performs at most $a(m) = b(1)$ BLOCK-UPDATES.

Now let $i > 1$ and suppose that the claim holds for $i - 1$. In this case, we also need to count the BLOCK-UPDATES performed by \mathbf{p}_i that are not atomic.

Recall that, in any real execution, \mathbf{p}_i performs at most one call to $\text{CONSTRUCT}(m)$ and it does not perform any BLOCK-UPDATES to m components. The non-atomic BLOCK-UPDATES performed by \mathbf{p}_i and the BLOCK-UPDATES that \mathbf{p}_i performed to construct them are all useless towards its goal of constructing a block update to m components. Since $a(r) > a(r - 1)$ for all $1 < r \leq m$, \mathbf{p}_i performs the maximum number of BLOCK-UPDATES when only BLOCK-UPDATES performed by \mathbf{p}_i to $m - 1$ components are non-atomic.

We now count the number of BLOCK-UPDATES performed by \mathbf{p}_i to $m - 1$ components. By Observation 6.8, \mathbf{p}_i performs at most $\binom{m}{m-1} = m$ atomic BLOCK-UPDATES to $m - 1$ components. By Observation 5.11, if a BLOCK-UPDATE, \mathbf{B} , performed by \mathbf{p}_i is not atomic, then some covering simulator \mathbf{p}_j with $j < i$ performed an H.update on line 3 of BLOCK-UPDATE (Algorithm 5.9) in the execution interval of \mathbf{B} . By the induction hypothesis, $\mathbf{p}_1, \dots, \mathbf{p}_{i-1}$ collectively perform at most $\sum_{j=1}^{i-1} b(j)$ BLOCK-UPDATES during the execution. In the worst case, each BLOCK-UPDATE performed by these simulators causes a different BLOCK-UPDATE to $m - 1$ components performed by \mathbf{p}_i to be non-atomic.

Since all BLOCK-UPDATES performed by \mathbf{p}_i to at most $m - 2$ components are atomic, by Lemma 6.9, it performs at most $a(m - 1)$ BLOCK-UPDATES in each of its calls to $\text{CONSTRUCT}(m - 1)$. Each BLOCK-UPDATE to $m - 1$ components is immediately preceded by a call to $\text{CONSTRUCT}(m - 1)$. Furthermore, after \mathbf{p}_i performs the last BLOCK-UPDATE, \mathbf{p}_i calls $\text{CONSTRUCT}(m - 1)$ once more.

Therefore, in total, \mathbf{p}_i performs at most $[m + \sum_{j=1}^{i-1} b(j)]a(m-1) + a(m-1) = (a(m-1)+1) \sum_{j=1}^{i-1} b(j) + a(m) = b(i)$ BLOCK-UPDATES. \square

Lemma 6.11. *Each covering simulator \mathbf{p}_i performs at most $2b(i) + 1$ operations. If there are only covering simulators, then every covering simulator terminates after taking at most 2^{nm^2} shared memory steps (i.e. scans and updates of the underlying single-writer snapshot object used to implement \mathbf{R}).*

Proof. By Observation 6.2 and Corollary 6.4, each simulator alternately performs SCAN and BLOCK-UPDATE, starting with at least one SCAN, until it performs a SCAN that causes it to **output** a value (and terminate). By Lemma 6.10, covering simulator \mathbf{p}_i performs at most $b(i)$ BLOCK-UPDATES and, hence, at most $b(i) + 1$ SCANS. It follows that \mathbf{p}_i performs at most $2b(i) + 1$ operations in total.

By Lemma 5.16, each BLOCK-UPDATE operation consists of 8 steps and each SCAN operation, S , consists of at most $2k(S) + 3$ steps, where $k(S)$ is the number of H.updates performed on line 3 of BLOCK-UPDATE (Algorithm 5.9) by other simulators during the execution interval of S . Notice that $\sum[k(S) : S \text{ is a SCAN by } \mathbf{p}_i]$ is bounded above by the number of BLOCK-UPDATE operations performed by the other simulators.

If there are only covering simulators, then $\sum[k(S) : S \text{ is a SCAN by } \mathbf{p}_i] \leq \sum_{j \neq i} b(j) \leq (n - 1)b(n)$. Thus, \mathbf{p}_i takes at most $\sum[2k(S) + 3 : S \text{ is a SCAN by } \mathbf{p}_i] \leq 2(n - 1)b(n) + 3(b(i) + 1)$ steps in its SCAN operations. In each of its at most $b(n)$ BLOCK-UPDATE operations, \mathbf{p}_i takes 8 steps. Therefore, \mathbf{p}_i takes at most $8b(n) + 2(n - 1)b(n) + 3(b(i) + 1) \leq (2n + 9)b(n) + 3$ steps in total. It can be verified that $(2n + 9)b(n) + 3 \leq (2n + 9)2^{nm(m-1)} + 3 \leq 2^{nm^2}$ whenever $n \geq 2$ and $m \geq 2$. \square

Lemma 6.12. *The simulation is wait-free.*

Proof. Lemma 6.11 takes care of the case when there are only covering simulators. Now suppose there are $x > 0$ direct simulators. Recall that, in the simulation, x direct simulators are only used in the case when Π is x -obstruction-free.

Suppose, for a contradiction, that there is a real execution where some process p_i performs infinitely many BLOCK-UPDATE operations on \mathbf{R} . Let σ be its intermediate execution. By Lemma 6.11, covering simulators perform only finitely many UPDATE operations in σ , so p_i is a direct simulator. Since σ is infinite, there is an infinite suffix σ'' of σ in which only direct simulators perform UPDATE operations. Let σ' be the prefix of σ prior to σ'' and let σ' be a simulated execution of Π that satisfies Lemma 6.6 for σ' . Let σ'' be the execution obtained by replacing each step and operation in σ'' with the step that it simulates.

By part 2 of Lemma 6.6 for σ' , the state of each simulated process at the end of σ' is the same as the state of the process at the end of σ' . Moreover, by part 3 of Lemma 6.6 for σ' , the contents of \mathbf{R} and \mathbf{S} are the same at the end of σ' and σ' , respectively. Thus, σ'' is applicable at the end of σ' . Since only direct simulators perform UPDATES in σ'' , σ'' is indistinguishable to the processes simulated by these direct simulators from an execution σ''' in which the processes simulated by covering simulators take no steps. Since σ''' is an infinite execution involving at most x processes, this contradicts the fact that Π is x -obstruction-free. Thus, every simulator performs only finitely many BLOCK-UPDATE operations on \mathbf{R} in the real execution.

Our implementation of \mathbf{R} is non-blocking. So, if processes perform infinitely many accesses to the underlying single-writer snapshot object in the implementation, then infinitely many operations on \mathbf{R} will complete. Since each simulator alternately performs SCAN and BLOCK-UPDATE and every simulator performs only finitely many BLOCK-UPDATE operations, there is no infinite execution. This means that the simulation is wait-free. \square

6.8 Applications of the Simulation

The main application of Theorem 6.1 is proving space lower bounds for colourless tasks. In particular, the lower bounds come from the contrapositive of the theorem.

Corollary 6.13. *Let T be a colourless task and let Π be an obstruction-free protocol that solves T among $n' \geq n \geq 2$ processes using an m -component atomic snapshot object, where $m \geq 2$.*

1. *Suppose that, in any deterministic, wait-free protocol that solves T among n processes using a single-writer atomic snapshot object, there is some process that takes at least L steps before terminating. Then $m \geq \min\{\lfloor n'/n \rfloor + 1, \sqrt{(\log_2 L)/n}\}$.*
2. *Suppose that there is no deterministic, wait-free protocol that solves T among n processes using a single-writer atomic snapshot object. If Π is x -obstruction-free, for some $x \in [n - 1]$, then $m \geq \lfloor (n' - x)/(n - x) \rfloor + 1$.*

Proof. Consider part 1 of the corollary. By Theorem 6.1.1, if $m < \min\{\lfloor n'/n \rfloor + 1, \sqrt{(\log_2 L)/n}\}$, then there exists a deterministic, wait-free protocol that solves T among n processes using a single-writer atomic snapshot object such that every process terminates after taking at most $2^{mn^2} < 2^{n(\sqrt{(\log_2 L)/n})^2} = L$ steps, which contradicts the assumption in part 1.

Now consider part 2 of the corollary. By Theorem 6.1.2, if $m < \lfloor (n' - x)/(n - x) \rfloor + 1$, then there exists a deterministic, wait-free protocol that solves T among n processes using a single-writer atomic snapshot object, which contradicts the assumption in part 2. \square

By using known impossibility results, we also obtain the following corollaries.

Corollary 6.14. *For $0 < \epsilon < 1$, any obstruction-free protocol for solving ϵ -approximate agreement among $n \geq 2$ processes uses at least $\min\{\lfloor \frac{n}{2} \rfloor + 1, \sqrt{\log_2 \log_3(\frac{1}{\epsilon})} - 2\}$ registers.*

Proof. It is known that any protocol for solving ϵ -approximate agreement between 2 processes using a single-writer atomic snapshot object has an execution in which some process takes at least $L = \frac{1}{2} \log_3(\frac{1}{\epsilon})$ steps [43]. Applying the first part of Corollary 6.13 with $n = 2$ gives the desired bound. \square

Corollary 6.15. *For $1 \leq x \leq k$, any x -obstruction-free protocol for solving k -set agreement among $n \geq k + 1$ processes uses at least $\lfloor \frac{n-x}{k+1-x} \rfloor + 1$ registers. In particular, any obstruction-free protocol for solving consensus among $n \geq 2$ processes uses at least n registers.*

Proof. It is known that it is impossible to solve k -set agreement among $k + 1$ processes in a wait-free manner using a single-writer atomic snapshot object [16, 41, 51, 8, 11]. Applying the first part of Corollary 6.13 with $n = k + 1$ gives the desired bound. \square

In our simulations, it appears that we are performing a covering argument similar to the one used in the proof of Theorem 4.5 Section 4.2. It is natural to wonder if it is possible to directly use a covering argument and obtain the same bounds. The difficulty is ensuring that processes do not terminate as we attempt to construct block updates to more components. On the other hand, in the context of our wait-free simulation, it is beneficial when a simulated process terminates.

Chapter 7

Why Extension-based Proofs Fail

In this chapter, we define a class of *extension-based* proofs and show that they cannot establish the impossibility of a deterministic, wait-free protocol for k -set agreement among $n > k > 1$ processes using registers. In Section 7.1, we describe some existing proofs for the impossibility of consensus and k -set agreement. In Section 7.2, we describe the non-uniform iterated immediate snapshot model, the topological view of a protocol, as well as some topological tools that we use. In Section 7.3, we describe a restricted version of extension-based proofs. In Section 7.4, we show that restricted extension-based proofs cannot establish the impossibility of a deterministic, wait-free protocol for k -set agreement. Finally, in Section 7.5, we define extension-based proofs and show that they also cannot establish the impossibility result.

7.1 Related Work

Fischer, Lynch, and Paterson proved that it is impossible to deterministically solve consensus, or 1-set agreement, in a wait-free manner among $n \geq 2$ processes [31]. Their approach is to inductively build an execution where processes have not yet reached agreement and, hence, need to take more steps. Specifically, they show that, from any bivalent configuration, there is a step of some process that results in another bivalent configuration. This allows them to explicitly construct an infinite execution in which no process has output a value. In Section 7.3, we show how to transform their proof into a restricted extension-based proof.

Consensus has been studied in many types of systems. Moses and Rajsbaum [48] gave a framework for studying the solvability of consensus, unifying many existing results. To do so, they introduced *layered* executions, a simple class of executions, and showed that it suffices to consider such executions in order to derive many of the existing results for consensus. Proofs using layered executions can be transformed into extension-based proofs.

The first proofs of the impossibility of deterministically solving k -set agreement in a wait-free manner among $n > k > 1$ processes all use methods from combinatorial topology [18, 41, 51]. The main idea is that, if there exists such a protocol, then the set of all possible final configurations (in which each process has output a value) of the protocol may be modeled by a simplicial complex and, moreover, this complex is a subdivision of the input complex. Then Sperner's Lemma may be used to show that there is a final configuration in which $k + 1$ different values have been output, which is a contradiction.

Section 7.5 shows that it is impossible to transform these proofs into extension-based proofs.

There are combinatorial proofs of the impossibility result for k -set agreement [8, 11, 13]. Like the topological proofs, these proofs also consider the set of final configurations of a supposedly wait-free k -set agreement protocol. However, they directly prove that there is a final configuration in which $k + 1$ different values have been output by relating different final configurations to one another using indistinguishability and employing arguments similar to proofs of Sperner’s Lemma.

There are proofs of the impossibility of consensus that show the protocol complex remains path-connected as the number of processes increases. This type of argument has been successfully generalized to prove the impossibility of k -set agreement. In particular, it can be shown that, for every wait-free protocol in the iterated immediate snapshot model, the protocol complex is highly connected. This allows one to apply a version of Sperner’s lemma. Chapter 10 of Herlihy, Kozlov, and Rajsbaum [38] contains more details.

The proofs of the impossibility of wait-free k -set agreement, for $k > 1$, all restrict attention to special executions or assume the existence of immediate snapshot objects. The configurations that result from such executions are much easier to analyze than arbitrary configurations in executions of protocols that use registers. Borowsky and Gafni’s proof [18] explicitly considers a system in which processes communicate using a sequence of one-shot immediate snapshot objects. Each process in a wait-free protocol must output a value after accessing the same number of immediate snapshot objects. They called this the *iterated immediate snapshot* (IIS) model and showed that it is computationally equivalent to the asynchronous shared memory model with registers. To do so, they showed how to implement an immediate snapshot in a wait-free manner using registers, which shows that the IIS model may be simulated in the registers model. They also showed how to simulate a protocol that uses registers in the IIS model. The main benefit of the IIS model is that the set of final configurations of any wait-free protocol in the IIS model may be represented by a simplicial complex that is obtained by performing a finite number of relatively simple standard chromatic subdivisions of the input complex.

Hoest and Shavit [43] introduced the NIIS model as a generalization of the IIS model in order to study the step complexity of tasks in the topological framework. They introduced the nonuniform chromatic subdivision operation, which generalizes the standard chromatic subdivision operation. They showed that the final configurations of any wait-free protocol in the NIIS model may be represented by a simplicial complex that is obtained by performing a finite number of nonuniform chromatic subdivisions of the input complex. As an application, they gave a topological proof that the step complexity of ϵ -approximate agreement in the NIIS model is $\Omega(\log(\frac{1}{\epsilon}))$.

7.2 Preliminaries

7.2.1 NIIS Model

We consider the *non-uniform iterated immediate snapshot* (NIIS) model with $n \geq 2$ processes, introduced by Hoest and Shavit [43]. For deterministic, wait-free computation, it is known that the NIIS model is equivalent to the standard asynchronous shared memory model, in which processes can only communicate by reading from and writing to shared registers. Specifically, any task that has a deterministic, wait-free solution in one of these models has a deterministic, wait-free solution in the other [43].

In the NIIS model, n processes, p_0, \dots, p_{n-1} , communicate using an infinite sequence, S_1, S_2, \dots , of

shared *single-writer atomic snapshot* objects. Each snapshot object has n components. The initial value of each component is \perp . A snapshot object supports two operations, `update(v)` and `scan()`. An `update(v)` operation by process p_i performed on a snapshot object updates component i of the object to have value v , where v is an element of an arbitrarily large set that does not contain \perp . A `scan()` operation returns a vector containing the current value of each component of the object.

Each process p_i accesses each snapshot object at most twice, starting with S_1 . Initially, p_i 's *state* consists of its identifier, i , and its input. The first time p_i accesses each snapshot object, it performs an `update` to set the i 'th component of the object to its current state. At its next step, it performs a `scan` of the same object. Its new state, s_i , consists of i and the result of the `scan`. Note that process p_i remembers its entire history, because its previous state is the i 'th component of the result of the `scan`. Next, p_i consults a map, Δ , to determine whether it should output a value. If $\Delta(s_i) \neq \perp$, then p_i outputs $\Delta(s_i)$ and terminates. If $\Delta(s_i) = \perp$, then, at its next step, p_i accesses the next snapshot object. A protocol in the NIIS model is completely specified by the map Δ .

A *configuration* consists of the contents of each shared object and the state of each process. A process is *active* in a configuration if it has not terminated. A configuration is *terminal* if it has no active processes. An *initial* configuration is specified by the input of each process. In an initial configuration, each component of each immediate snapshot object contains \perp .

From any configuration C , a *scheduler* decides the order in which the processes take steps. It repeatedly selects a set of processes that are all poised to perform `updates` on the same snapshot object. Each of the processes in the set, in order of their identifiers, performs its `update`. Then, each of these processes performs its next `scan`, in the same order. Note that the scheduler never selects processes that have terminated. The sequence of subsets of processes selected by the scheduler is called a *schedule from C* . Given a finite schedule α from C , we use $C\alpha$ to denote the resulting configuration. For example, if p_0 and p_1 are poised to access the same snapshot object in C , then $(\{p_0, p_1\})$ is a schedule from C . If p_1 is active in $C' = C(\{p_0, p_1\})$, then $(\{p_1\})$ is a schedule from C' and $(\{p_0, p_1\}, \{p_1\})$ is a schedule from C . After this schedule, p_1 has updated and scanned one more snapshot object than p_0 . Each finite schedule from an initial configuration results in a *reachable* configuration. A protocol is *wait-free* if there is no infinite schedule from any initial configuration.

Since each process remembers its entire history and only process p_i can update the i 'th component of each snapshot object, the contents of the snapshot objects in a configuration are fully determined by the states of the processes in that configuration.

Observation 7.1. *A reachable configuration is fully specified by the set of states of all processes in the configuration (including the processes that have terminated).*

Two configurations C and C' are *indistinguishable* to a set of processes P if every process in P has the same state in C and C' . Two finite schedules α and β from C are *indistinguishable* to a set of processes P if the resulting configurations $C\alpha$ and $C\beta$ are indistinguishable to P .

Observation 7.2. *Suppose C and C' are two reachable configurations that are indistinguishable to P , every active process in P is poised to `update` S_t in C , each snapshot object S_r has the same contents in C and C' for all $r \geq t$, and α is a finite schedule from C containing only processes in P . Then α is a schedule from C' and the configurations $C\alpha$ and $C'\alpha$ are indistinguishable to P .*

Suppose C is a reachable configuration in which all active processes are poised to `update` the same snapshot object. For any set of processes P , a *P -only 1-round schedule* from C is an ordered partition

of the processes in P that are active in C . If none of the processes in P are active in C , then the empty schedule is the only 1-round schedule. A *full* 1-round schedule from C is a P -only 1-round schedule where P is the set of all processes. Observe that, in the NIIS model, if α is a P -only 1-round schedule from C and β is any full 1-round schedule from C such that $\beta = \alpha\alpha'$, then α and β are indistinguishable to the processes in P .

For each $t > 1$, a *P -only t -round schedule* from C (as defined above) is a schedule $\alpha_1 \cdots \alpha_t$ such that, for each $1 \leq i \leq t$, α_i is a P -only 1-round schedule from $C\alpha_1 \cdots \alpha_{i-1}$. Notice that some processes in P may have terminated during $\alpha_1 \cdots \alpha_{i-1}$. These processes are not included in α_i . A *full t -round schedule* from C is a P -only t -round schedule from C where P is the set of all processes. Observe that, if $\alpha_1 \cdots \alpha_t$ is a P -only t -round schedule from C , $\beta_1 \cdots \beta_t$ is a full t -round schedule from C , and α_i is a prefix of β_i for all $1 \leq i \leq t$, then the configurations $C\alpha_1 \cdots \alpha_t$ and $C\beta_1 \cdots \beta_t$ are indistinguishable to the processes in P .

7.2.2 Topological Representation of a Protocol

In the *topological* view of a protocol in the NIIS model (specified by a map Δ), every reachable configuration, C , of the protocol is represented by a simplex, σ , which is the set of all subsets of states of processes in C . In particular, each vertex of σ is a set containing the state one process in C . Since the state of a process includes its identifier, no two processes have the same state in C and, hence, σ has n vertices. By Observation 7.1, the set of vertices of σ fully specifies C .

For each $t \geq 0$, let \mathbb{S}^t denote the simplicial complex consisting of all simplices representing configurations reachable from initial configurations by full t -round schedules. In particular, the *input complex* of the protocol, \mathbb{S}^0 , represents all possible initial configurations.

Hoest and Shavit [43] introduced the *non-uniform chromatic subdivision* operation, χ , and proved that $\mathbb{S}^{t+1} = \chi(\mathbb{S}^t, \Delta)$, i.e. \mathbb{S}^{t+1} is the non-uniform chromatic subdivision of \mathbb{S}^t . In general, the *non-uniform chromatic subdivision operation* χ maps every subcomplex \mathbb{A} of \mathbb{S}^t to a subcomplex $\chi(\mathbb{A}, \Delta)$ of \mathbb{S}^{t+1} . It has the property that $\chi(\mathbb{A}, \Delta)$ is the union of $\chi(\sigma, \Delta)$ over all simplices $\sigma \subseteq \mathbb{A}$. The formal definition of this operation is fairly technical and appears in Section 7.2.3.

Suppose σ is an n -vertex simplex in \mathbb{S}^t , which represents a reachable configuration C . A special case of Hoest and Shavit's result is that every n -vertex simplex in $\chi(\sigma, \Delta) \subseteq \mathbb{S}^{t+1}$ represents a configuration reachable from C by a full 1-round schedule. More generally, if P is a subset of the processes and $\tau \subseteq \sigma$ is a simplex that consists of vertices representing the states of these processes in configuration C , then each simplex in $\chi(\tau, \Delta)$ consists of vertices representing the states of these processes in a configuration reachable from C by a P -only 1-round schedule. If τ is also a subset of an n -vertex simplex σ' that represents another configuration C' , then, by Observation 7.2, for each P -only 1-round schedule α , the states of the processes in P are the same in $C\alpha$ and $C'\alpha$.

There is a natural geometric interpretation of an (abstract) simplicial complex and subdivision. A *geometric simplex* σ is the set of convex combinations of a finite number of affinely independent points (each of which is a *vertex* of σ) in some Euclidean space [38]. A *face* of σ is the set of convex combinations of a subset of the affinely independent points. A *geometric simplicial complex* \mathcal{K} is a finite collection of geometric simplices such that each face of $\sigma \in \mathcal{K}$ is a simplex in \mathcal{K} and, for any two simplices $\sigma, \tau \in \mathcal{K}$, $\sigma \cap \tau \in \mathcal{K}$. The *geometric realization* of \mathcal{K} is the union of the simplices in \mathcal{K} (in Euclidean space). A geometric simplicial complex \mathcal{B} is a *subdivision* of \mathcal{A} if their geometric realizations are the same and each simplex in \mathcal{A} is the union of finitely many simplices in \mathcal{B} . One of the main contributions of Hoest and

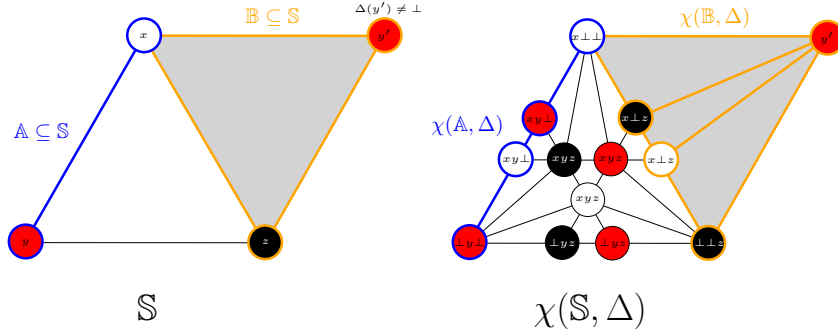


Figure 7.1: A non-uniform chromatic subdivision.

Shavit [43] is showing that the non-uniform chromatic subdivision of a simplicial complex is a subdivision of the simplicial complex in the geometric setting.

Figure 7.1 contains an example of the non-uniform chromatic subdivision of a (geometric) simplicial complex \mathbb{S} with 3 processes, p_0 , p_1 , and p_2 , in Euclidean space. In the configuration represented by the left triangle of \mathbb{S} , p_0 , p_1 , and p_2 have states x , y , and z , respectively, none of which have terminated. In the configuration represented by the right triangle, p_0 and p_2 have the same state, but p_1 has state y' , in which it terminates and outputs $\Delta(y')$. We also illustrate two subcomplexes \mathbb{A} and \mathbb{B} of \mathbb{S} and their subdivisions. Note that all vertices in \mathbb{S} and $\chi(\mathbb{S}, \Delta)$ representing states of the same process have the same colour. For readability, process identifiers are omitted from the states in $\chi(\mathbb{S}, \Delta)$.

7.2.3 Non-uniform Chromatic Subdivision

In this section, we define the non-uniform chromatic subdivision of a simplex and then extend the definition to a simplicial complex. Hoest and Shavit [43] discuss these definitions in more detail.

Let $t \geq 0$ and let $\sigma \subseteq \mathbb{S}^t$ be an n -vertex simplex that represents a reachable configuration C . For each vertex $v \in \sigma$, let $state(v)$ be the state represented by v . For each simplex $\tau \subseteq \sigma$, let P_τ denote the set of processes whose states appear in τ and let $Id(\tau)$ be the set of identifiers of the processes in P_τ . Let $\vec{\tau}$ be the n -component vector such that $\vec{\tau}_i$ is $state(v)$, if there is a vertex $v \in \tau$ that represents the state of process p_i , and \perp otherwise. Then $\vec{\tau}$ is the result of each process in P_τ 's last scan in the configuration CP_τ .

For any vertex v , we let $\Delta(v)$ denote $\Delta(state(v))$. We say that vertex v is *active* if $\Delta(v) = \perp$ and it has *terminated* if $\Delta(v) \neq \perp$. The non-uniform chromatic subdivision $\chi(\sigma, \Delta)$ of σ can only be defined when $\Delta(v)$ is defined for every vertex $v \in \sigma$. The non-uniform chromatic subdivision $\chi(\mathbb{A}, \Delta) \subseteq \mathbb{S}^{t+1}$ of \mathbb{A} is the union of $\chi(\sigma, \Delta)$ for all simplices $\sigma \subseteq \mathbb{A}$. To define $\chi(\sigma, \Delta)$, we consider two cases.

Case 1: *Every vertex $v \in \sigma$ is active.* Then $\chi(\sigma, \Delta)$ is called the *standard chromatic subdivision* of σ . The vertices of $\chi(\sigma, \Delta)$ are of the form $\{(i, \vec{\tau}_i)\}$, where $i \in Id(\sigma)$ is an identifier and $\tau_i \subseteq \sigma$ is a simplex such that $i \in Id(\tau_i)$. Suppose that $I \subseteq Id(\sigma)$ is a set of identifiers and, for each $i \in I$, $\tau_i \subseteq \sigma$ is a simplex such that $i \in Id(\tau_i)$. Then $\{(i, \vec{\tau}_i) : i \in I\} \in \chi(\sigma, \Delta)$ if and only if there is an ordering \preceq on I such that $i \preceq j$ implies that $\tau_i \subseteq \tau_j$ and, for each $i, j \in I$, if $i \in Id(\tau_j)$, then $\tau_i \subseteq \tau_j$.

Case 2: *Some vertex $v \in \sigma$ has terminated.* Let T be the set of terminated vertices in σ and let $\tau \subseteq \sigma$ be the simplex consisting of all subsets of σ that only contain active vertices. The *non-uniform chromatic subdivision* of σ is the abstract simplicial complex $\chi(\sigma, \Delta)$ whose vertices are the vertices in T and the vertices in the standard chromatic subdivision $\chi(\tau, \Delta)$ of τ . Each set in $\chi(\tau, \Delta)$ is a set in

$\chi(\sigma, \Delta)$. In addition, if $T' \subseteq T$ and $\tau' \in \chi(\tau, \Delta)$, then $T' \cup \tau' \in \chi(\sigma, \Delta)$.

We note the following property for terminated vertices:

Proposition 7.3. *Suppose vertex v has terminated in \mathbb{S}^t . Let \mathbb{A} be the subcomplex of \mathbb{S}^t consisting of all subsets in \mathbb{S}^t that only contain vertices adjacent to v in \mathbb{S}^t . Then v is adjacent to each vertex in $\chi(\mathbb{A}, \Delta)$ in \mathbb{S}^{t+1} .*

7.2.4 Distances between Subcomplexes

Let \mathbb{S} be a simplicial complex and let \mathbb{A} and \mathbb{B} be non-empty subcomplexes of \mathbb{S} . A *path* between \mathbb{A} and \mathbb{B} in \mathbb{S} of *length* ℓ is a sequence of vertices v_0, v_1, \dots, v_ℓ such that $v_0 \in \mathbb{A}$, $v_\ell \in \mathbb{B}$, and, for $0 \leq j < \ell$, $\{v_j, v_{j+1}\}$ is an edge in \mathbb{S} . Notice that a vertex may appear more than once in a path and, if \mathbb{A} and \mathbb{B} both consist of a single vertex, then we have the standard definition of a (non-simple) path in an undirected graph. \mathbb{S} is *connected* if for any two vertices $u, v \in \mathbb{S}$, there is path between u and v in \mathbb{S} . If \mathbb{S} is connected, then the *distance* between \mathbb{A} and \mathbb{B} in \mathbb{S} , denoted $\text{dist}_{\mathbb{S}}(\mathbb{A}, \mathbb{B})$, is the minimum $\ell \geq 0$ such that there is a path between \mathbb{A} and \mathbb{B} in \mathbb{S} of length ℓ .

The following proposition describes some basic properties of the non-uniform chromatic subdivision operation, which all follow from the fact that the non-uniform chromatic subdivision of a simplicial complex is a subdivision of the simplicial complex in the geometric setting. Since the input complex, \mathbb{S}^0 , is connected, the proposition implies \mathbb{S}^t is connected for all $t \geq 1$. Hence, the distance between subcomplexes in \mathbb{S}^t is well-defined.

Proposition 7.4. *The non-uniform chromatic subdivision operation has the following properties:*

1. *If \mathbb{S}^0 is connected, then, for all $t \geq 1$, \mathbb{S}^t is connected.*
2. *\mathbb{A} and \mathbb{B} are disjoint subcomplexes of \mathbb{S}^t if and only if $\chi(\mathbb{A}, \Delta)$ and $\chi(\mathbb{B}, \Delta)$ are disjoint subcomplexes of \mathbb{S}^{t+1} .*
3. *If every path between subcomplexes \mathbb{A} and \mathbb{B} in \mathbb{S}^t passes through a subcomplex \mathbb{C} , then every path between $\chi(\mathbb{A}, \Delta)$ and $\chi(\mathbb{B}, \Delta)$ in \mathbb{S}^{t+1} passes through $\chi(\mathbb{C}, \Delta)$.*
4. *If $\mathbb{C} \subseteq \mathbb{S}^t$ is a subcomplex containing only active vertices and \mathbb{C}_1 and \mathbb{C}_2 are disjoint nonempty subcomplexes of \mathbb{C} , then $\text{dist}_{\mathbb{S}^{t+1}}(\chi(\mathbb{C}_1, \Delta), \chi(\mathbb{C}_2, \Delta)) \geq 2$.*

We now prove one of the main technical tools used in this paper. It allows us to relate the distance between two subcomplexes in \mathbb{S}^t to the distance between the nonuniform chromatic subdivisions of these subcomplexes in \mathbb{S}^{t+1} .

Lemma 7.5. *Let \mathbb{A} and \mathbb{B} be nonempty subcomplexes of \mathbb{S}^t , let $\mathbb{A}' = \chi(\mathbb{A}, \Delta)$, and let $\mathbb{B}' = \chi(\mathbb{B}, \Delta)$. Then $\text{dist}_{\mathbb{S}^{t+1}}(\mathbb{A}', \mathbb{B}') \geq \text{dist}_{\mathbb{S}^t}(\mathbb{A}, \mathbb{B})$. If every path between \mathbb{A} and \mathbb{B} in \mathbb{S}^t contains at least one edge between active vertices, then $\text{dist}_{\mathbb{S}^{t+1}}(\mathbb{A}', \mathbb{B}') \geq \text{dist}_{\mathbb{S}^t}(\mathbb{A}, \mathbb{B}) + 1$.*

Proof. We will prove the first claim by induction on $\text{dist}_{\mathbb{S}^t}(\mathbb{A}, \mathbb{B})$. The base case is when $\text{dist}_{\mathbb{S}^t}(\mathbb{A}, \mathbb{B}) = 0$. Since distances are always non-negative, $\text{dist}_{\mathbb{S}^{t+1}}(\mathbb{A}', \mathbb{B}') \geq 0 = \text{dist}_{\mathbb{S}^t}(\mathbb{A}, \mathbb{B})$.

So, suppose $d = \text{dist}_{\mathbb{S}^t}(\mathbb{A}, \mathbb{B}) > 0$ and the first claim holds for all non-empty subcomplexes $\hat{\mathbb{A}}$ and $\hat{\mathbb{B}}$ of \mathbb{S}^t where $\text{dist}_{\mathbb{S}^t}(\hat{\mathbb{A}}, \hat{\mathbb{B}}) < d$. Let \mathbb{C} be the subcomplex of \mathbb{S}^t consisting of all subsets in \mathbb{S}^t that only contain vertices v for which $\text{dist}_{\mathbb{S}^t}(v, \mathbb{A}) = 1$ and let $\mathbb{C}' = \chi(\mathbb{C}, \Delta)$. Since $\text{dist}_{\mathbb{S}^t}(\mathbb{A}, \mathbb{B}) > 0$, \mathbb{C} is non-empty and

$\text{dist}_{\mathbb{S}^t}(\mathbb{C}, \mathbb{B}) = d - 1$. Moreover, since \mathbb{A} and \mathbb{C} are disjoint, by Proposition 7.4(2), \mathbb{A}' and \mathbb{C}' are disjoint, i.e. $\text{dist}_{\mathbb{S}^{t+1}}(\mathbb{A}', \mathbb{C}') \geq 1$. By the induction hypothesis applied to \mathbb{C} and \mathbb{B} , $\text{dist}_{\mathbb{S}^{t+1}}(\mathbb{C}', \mathbb{B}') \geq \text{dist}_{\mathbb{S}^t}(\mathbb{C}, \mathbb{B}) = d - 1$. Since every path between \mathbb{A} and \mathbb{B} in \mathbb{S}^t passes through \mathbb{C} , by Proposition 7.4(3), every path between \mathbb{A}' and \mathbb{B}' in \mathbb{S}^{t+1} passes through \mathbb{C}' . Therefore, $\text{dist}_{\mathbb{S}^{t+1}}(\mathbb{A}', \mathbb{B}') \geq \text{dist}_{\mathbb{S}^{t+1}}(\mathbb{A}', \mathbb{C}') + \text{dist}_{\mathbb{S}^{t+1}}(\mathbb{C}', \mathbb{B}') \geq d$ and the first claim holds for \mathbb{A} and \mathbb{B} .

Now suppose that every path between \mathbb{A} and \mathbb{B} in \mathbb{S}^t contains at least one edge between active vertices. Let $E \neq \emptyset$ be a smallest set of edges between active vertices in \mathbb{S}^t such that every path between \mathbb{A} and \mathbb{B} contains at least one edge in E . Viewing \mathbb{S}^t as a graph, the removal of E from \mathbb{S}^t results in some number of connected components. Let $\hat{\mathbb{A}}$ be the set of vertices in the connected components that contain at least one vertex in \mathbb{A} and let $\hat{\mathbb{B}}$ be the set of remaining vertices in \mathbb{S}^t . Let $\hat{\mathbb{A}}$ and $\hat{\mathbb{B}}$ be the subcomplexes of \mathbb{S}^t consisting of all subsets in \mathbb{S}^t that only contain vertices in $\hat{\mathbb{A}}$ and $\hat{\mathbb{B}}$, respectively. Observe that \mathbb{A} is a subcomplex of $\hat{\mathbb{A}}$ and \mathbb{B} is a subcomplex of $\hat{\mathbb{B}}$. Let $\hat{E} \subseteq E$ be the set of edges between $\hat{\mathbb{A}}$ and $\hat{\mathbb{B}}$. Note that $\hat{\mathbb{A}}$ and $\hat{\mathbb{B}}$ partition the vertices in \mathbb{S}^t . Hence, every path between $\hat{\mathbb{A}}$ and $\hat{\mathbb{B}}$ contains an edge in \hat{E} . In particular, every path between $\mathbb{A} \subseteq \hat{\mathbb{A}}$ and $\mathbb{B} \subseteq \hat{\mathbb{B}}$ contains an edge in \hat{E} . By the minimality of E , $E = \hat{E}$.

Let \mathbb{C} be the subcomplex of \mathbb{S}^t consisting of all subsets in \mathbb{S}^t that only contain vertices contained in some edge in E . By definition of E , every vertex in \mathbb{C} is active. Moreover, every path between $\hat{\mathbb{A}}$ and $\hat{\mathbb{B}} \cap \mathbb{C}$ passes through $\hat{\mathbb{A}} \cap \mathbb{C}$ and every path between $\hat{\mathbb{A}} \cap \mathbb{C}$ and $\hat{\mathbb{B}}$ passes through $\hat{\mathbb{B}} \cap \mathbb{C}$.

Let $\hat{\mathbb{A}}' = \chi(\hat{\mathbb{A}}, \Delta)$ and let $\hat{\mathbb{B}}' = \chi(\hat{\mathbb{B}}, \Delta)$. Then, by Proposition 7.4(3), every path between $\hat{\mathbb{A}}'$ and $\chi(\hat{\mathbb{B}} \cap \mathbb{C}, \Delta)$ passes through $\chi(\hat{\mathbb{A}} \cap \mathbb{C}, \Delta)$ and every path between $\chi(\hat{\mathbb{A}} \cap \mathbb{C}, \Delta)$ and $\hat{\mathbb{B}}'$ passes through $\chi(\hat{\mathbb{B}} \cap \mathbb{C}, \Delta)$. It follows that every path between $\mathbb{A}' \subseteq \hat{\mathbb{A}}'$ and $\mathbb{B}' \subseteq \hat{\mathbb{B}}'$ consists of a path between \mathbb{A}' and $\chi(\hat{\mathbb{A}} \cap \mathbb{C}, \Delta)$, a path between $\chi(\hat{\mathbb{A}} \cap \mathbb{C}, \Delta)$ and $\chi(\hat{\mathbb{B}} \cap \mathbb{C}, \Delta)$, and a path between $\chi(\hat{\mathbb{B}} \cap \mathbb{C}, \Delta)$ and \mathbb{B}' . Thus

$$\begin{aligned} \text{dist}_{\mathbb{S}^{t+1}}(\mathbb{A}', \mathbb{B}') &\geq \text{dist}_{\mathbb{S}^{t+1}}(\mathbb{A}', \chi(\hat{\mathbb{A}} \cap \mathbb{C}, \Delta)) \\ &\quad + \text{dist}_{\mathbb{S}^{t+1}}(\chi(\hat{\mathbb{A}} \cap \mathbb{C}, \Delta), \chi(\hat{\mathbb{B}} \cap \mathbb{C}, \Delta)) \\ &\quad + \text{dist}_{\mathbb{S}^{t+1}}(\chi(\hat{\mathbb{B}} \cap \mathbb{C}, \Delta), \mathbb{B}'). \end{aligned}$$

By the first claim,

$$\begin{aligned} \text{dist}_{\mathbb{S}^{t+1}}(\mathbb{A}', \chi(\hat{\mathbb{A}} \cap \mathbb{C}, \Delta)) &\geq \text{dist}_{\mathbb{S}^t}(\mathbb{A}, \hat{\mathbb{A}} \cap \mathbb{C}) \text{ and} \\ \text{dist}_{\mathbb{S}^{t+1}}(\mathbb{B}', \chi(\hat{\mathbb{B}} \cap \mathbb{C}, \Delta)) &\geq \text{dist}_{\mathbb{S}^t}(\mathbb{B}, \hat{\mathbb{B}} \cap \mathbb{C}). \end{aligned}$$

Now consider $\text{dist}_{\mathbb{S}^{t+1}}(\chi(\hat{\mathbb{A}} \cap \mathbb{C}, \Delta), \chi(\hat{\mathbb{B}} \cap \mathbb{C}, \Delta))$. Since every vertex in \mathbb{C} is active and $\hat{\mathbb{A}} \cap \mathbb{C}$ and $\hat{\mathbb{B}} \cap \mathbb{C}$ are disjoint non-empty subcomplexes of \mathbb{C} , Proposition 7.4(4) implies that

$$\text{dist}_{\mathbb{S}^{t+1}}(\chi(\hat{\mathbb{A}} \cap \mathbb{C}, \Delta), \chi(\hat{\mathbb{B}} \cap \mathbb{C}, \Delta)) \geq 2.$$

By definition of \mathbb{C} , $\text{dist}_{\mathbb{S}^t}(\hat{\mathbb{A}} \cap \mathbb{C}, \hat{\mathbb{B}} \cap \mathbb{C}) = 1$. Hence,

$$\text{dist}_{\mathbb{S}^t}(\mathbb{A}, \hat{\mathbb{A}} \cap \mathbb{C}) + \text{dist}_{\mathbb{S}^t}(\mathbb{B}, \hat{\mathbb{B}} \cap \mathbb{C}) = \text{dist}_{\mathbb{S}^t}(\mathbb{A}, \mathbb{B}) - 1.$$

Finally, combining these equations, it follows that $\text{dist}_{\mathbb{S}^{t+1}}(\mathbb{A}', \mathbb{B}') \geq \text{dist}_{\mathbb{S}^t}(\mathbb{A}, \mathbb{B}) + 1$. \square

7.3 Restricted Extension-Based Proofs

In this section, we introduce and formally define the class of *restricted extension-based proofs*, which are used to prove impossibility results. To prove that a task has no wait-free solution, given any protocol that supposedly solves the task, such a proof picks an initial configuration and constructs a schedule from the configuration, which witnesses the fact that the protocol is incorrect. Specifically, either the schedule is infinite or one of the specifications of the task is violated in the configuration resulting from the schedule. To learn information about the protocol, the proof queries the protocol about the states of processes in various reachable configurations. In the NIIS model, the only information a proof learns about the state of a process in a reachable configuration is whether that process has output a value and, if so, the value that it output. The rest of the information about its state is the same for all NIIS protocols. In other words, if the protocol is specified by a map Δ from process states to outputs or \perp , then the proof is querying the map Δ at various process states.

More formally, a *restricted extension-based* proof in the NIIS model is an interaction between a prover and any protocol defined by a map Δ from process states to outputs or \perp . The prover starts with no knowledge about the protocol (except its initial configurations) and makes the protocol reveal information about the states of processes in various configurations by asking *queries*. Each query allows the prover to *reach* some configuration of the protocol. The interaction proceeds in phases, beginning with phase 1.

In each phase $\varphi \geq 1$, the prover starts with a finite schedule, $\alpha(\varphi)$, and a set, $\mathcal{A}(\varphi)$, of configurations that are reached by performing $\alpha(\varphi)$ from initial configurations. These initial configurations only differ from one another in the input values of processes that do not appear in the schedule $\alpha(\varphi)$. The prover also maintains a set, $\mathcal{A}'(\varphi)$, containing the configurations it reaches by non-empty schedules from configurations in $\mathcal{A}(\varphi)$ during phase φ . This set is empty at the start of phase φ . At the start of phase 1, $\alpha(1)$ is the empty schedule and $\mathcal{A}(1)$ is the set of all initial configurations of the protocol.

The prover *queries* the protocol by specifying a configuration $C \in \mathcal{A}(\varphi) \cup \mathcal{A}'(\varphi)$ and a set of processes P that are poised to update the same snapshot object in C . For each process $p_i \in P$, let s_i denote the state of p_i in the configuration C' resulting from scheduling P from C . The protocol replies to this query with $\Delta(i, s_i)$, for each $p_i \in P$. Notice that, by the definition of the NIIS model, this is enough for the prover to know the state of every process and the contents of every component of every snapshot object in configuration C' . Then, the prover adds C' to $\mathcal{A}'(\varphi)$, and we say that the prover has *reached* C' .

If the prover reaches a configuration in which the outputs of the processes do not satisfy the specifications of the task, it has demonstrated that the protocol is incorrect. In this case, the prover *wins* (and the interaction ends).

A *chain of queries* is a (finite or infinite) sequence of queries such that, if (C_i, P_i) and (C_{i+1}, P_{i+1}) are consecutive queries in the chain, then C_{i+1} is the configuration resulting from scheduling P_i from C_i . If the prover constructs an infinite chain of queries, it has demonstrated that the protocol is not wait-free. In this case, the prover also *wins* (and the interaction ends). In particular, the prover wins against the trivial protocol in which no process ever outputs a value, by constructing any infinite chain of queries. After constructing finitely many chains of queries in phase φ without winning, the prover must end the phase by committing to an extension of the schedule $\alpha(\varphi)$.

It is important that the prover is allowed to construct chains of queries instead of just single queries. Whenever a chain of queries results in a terminal configuration, rather than going on forever, the prover learns useful information about the protocol, i.e. the outputs of all processes in that configuration. If the

prover cannot construct a chain of queries, then it might be forced to commit to an extension without learning any useful information about the protocol. This is because it is possible to add any finite number of rounds at the beginning of a protocol that are ignored by the processes.

Suppose the prover chooses configuration $C' \in \mathcal{A}'(\varphi)$ at the end of phase φ . Let α' be a (nonempty) schedule such that C' is reached by performing α' starting from some configuration $C \in \mathcal{A}(\varphi)$. Let $\alpha(\varphi + 1)$ denote the schedule $\alpha(\varphi)\alpha'$. Since $C \in \mathcal{A}(\varphi)$, there is an initial configuration I such that C is reached by performing $\alpha(\varphi)$ starting from I . Thus C' is reached by performing $\alpha(\varphi + 1)$ starting from I . Finally, let $\mathcal{A}(\varphi + 1)$ be the set of all configurations that are reached by performing $\alpha(\varphi + 1)$ from the initial configurations that only differ from I by the states of processes that do not appear in this schedule. Then the prover begins phase $\varphi + 1$.

If every process has terminated in every configuration in $\mathcal{A}(\varphi)$ and the outputs satisfy the specification of the task, then $\mathcal{A}'(\varphi) = \emptyset$, the prover *loses*, and the interaction ends.

If the number of phases in the interaction is infinite, the prover has constructed an infinite schedule in which some processes remain active and, hence, the protocol is not wait-free. This is the third way that the prover can *win*.

To prove that a task is impossible using an extension-based proof, there must exist a prover that wins against every protocol.

An example. We express the proof of the impossibility of deterministically solving wait-free binary consensus among two or more processes as a restricted extension-based proof.

Theorem 7.6. *Deterministic wait-free consensus among $n \geq 2$ processes is impossible in the NIIS model.*

Proof. Let C_0 denote the initial configuration in which p_0 has input 0 and p_1 has input 1. Then, by validity, the solo-execution by p_0 must decide $a_0 = 0$ and the solo-execution by p_1 must decide $1 - a_0 = 1$. The prover performs the query chain corresponding to the solo execution by p_0 from C_0 . The prover wins if this does not terminate or p_0 does not output 0. Similarly, the prover performs the query chain corresponding to the solo execution by p_1 from C_0 and wins if this does not terminate or p_1 does not output 1.

The prover will either construct an infinite query chain in some phase, reach a configuration in which both 0 and 1 have been output, or inductively construct an infinite sequence of configurations C_1, C_2, \dots and a corresponding sequence of bits a_1, a_2, \dots such that, for all $i \geq 1$, C_i is reached from C_{i-1} by scheduling one set of processes (either $\{p_0\}$, $\{p_1\}$, or $\{p_0, p_1\}$), the solo-execution by p_0 from C_i outputs a_i , and the solo-execution by p_1 from C_i outputs $1 - a_i$. Let $i \geq 1$ and suppose the claim is true for $i - 1$.

If process p_0 has terminated (and output value a_{i-1}) in configuration C_{i-1} , then the solo execution by p_1 from C_{i-1} , which outputs $1 - a_{i-1}$, results in a configuration in which both 0 and 1 have been output. Similarly, if p_1 has terminated in configuration C_{i-1} , then the prover has reached a configuration in which both 0 and 1 have been output. So, suppose that neither p_0 nor p_1 has terminated in C_{i-1} .

From C_{i-1} , the prover first performs the query chain corresponding to the schedule $\{p_0\}, \{p_1\}, \{p_1\}, \dots$ where p_0 is scheduled once and then p_1 is scheduled until it outputs a value b_i . If that never happens, then the prover wins. If $b_i = 1 - a_{i-1}$, then the prover ends phase i , chooses $C_i = C_{i-1}\{p_0\}$, and sets $a_i = a_{i-1}$. Note that the solo execution by p_0 from C_i outputs $a_i = a_{i-1}$ and the solo execution by p_1 from C_i outputs $1 - a_i = 1 - a_{i-1}$.

Otherwise, $b_i = a_{i-1}$. In this case, the prover performs the query chain from C_{i-1} corresponding to the schedule $\{p_1\}, \{p_0\}, \{p_0\}, \dots$, where p_1 is scheduled once and then p_0 is scheduled until it outputs a value d_i . If that never happens, then the prover wins. If $d_i = a_{i-1}$, then the prover ends the round, chooses $C_i = C_{i-1}\{p_1\}$, and sets $a_i = a_{i-1}$. Note that the solo execution by p_0 from C_i outputs $a_i = a_{i-1}$ and the solo execution by p_1 from C_i outputs $1 - a_i = 1 - a_{i-1}$.

Otherwise, $d_i = 1 - a_{i-1}$. Then the prover performs the query $\{p_0, p_1\}$. Note that the configurations $C_{i-1}\{p_0, p_1\}$ and $C_{i-1}\{p_0\}\{p_1\}$ are indistinguishable to p_1 , i.e. p_1 has the same state in both configurations. Thus, it outputs b_i in its solo execution from $C_{i-1}\{p_0, p_1\}$. Likewise, the configurations $C_{i-1}\{p_0, p_1\}$ and $C_{i-1}\{p_1\}\{p_0\}$ are indistinguishable to p_0 , so it outputs d_i in its solo execution from $C_{i-1}\{p_0, p_1\}$. Finally, the prover ends the phase by choosing $C_i = C_{i-1}\{p_0, p_1\}$, and sets $a_i = d_i$. Note that the solo execution by p_0 from C_i outputs a_i and the solo execution by p_1 from C_i outputs $b_i = a_{i-1} = 1 - d_i = 1 - a_i$.

Thus, in all cases, the claim is true for i . Hence, by induction, the claim is true for all $i \geq 0$. \square

7.4 Why Restricted Extension-based Proofs Fail

In this section, we prove that no restricted extension-based proof can show the impossibility of deterministically solving k -set agreement in a wait-free manner in the NIIS model, for $n > k \geq 2$ processes. Observe that any protocol for $n > k + 1$ processes is also a protocol for $k + 1$ processes, since the remaining processes could crash before taking any steps. Therefore, it suffices to consider $n = k + 1$.

To show our result, we define an adversary that is able to win against every restricted extension-based prover. The adversary maintains a *partial* specification of Δ (the protocol it is adaptively constructing) and an integer $t \geq 0$. The integer t represents the number of non-uniform chromatic subdivisions of the input complex, \mathbb{S}^0 , that it has performed. Once the adversary has defined Δ for each vertex in \mathbb{S}^t , then it may perform a non-uniform chromatic subdivision of \mathbb{S}^t (or *subdivide* \mathbb{S}^t) and construct $\mathbb{S}^{t+1} = \chi(\mathbb{S}^t, \Delta)$.

For each $0 \leq r \leq t$ and each input value $a \in \{0, 1, \dots, k\}$, we define \mathbb{N}_a^r to be the subcomplex of \mathbb{S}^r consisting of all subsets in \mathbb{S}^r that only contain vertices representing states of processes which have not seen a (in any scan) and \mathbb{T}_a^r to be the subcomplex of \mathbb{S}^r consisting of all subsets in \mathbb{S}^r that only contain vertices representing states of processes which have output a and terminated, i.e. vertices for which Δ is a . If a vertex v corresponds to the state of a process that has seen input a , then we say v *contains* a . Notice that \mathbb{N}_a^r does not change when the adversary updates Δ , while \mathbb{T}_a^r could possibly change. From these definitions, it follows that non-uniform chromatic subdivisions of these subcomplexes have simple descriptions.

Proposition 7.7. *For all inputs a and $0 \leq r < t$, \mathbb{N}_a^r is non-empty and $\chi(\mathbb{N}_a^r, \Delta) = \mathbb{N}_a^{r+1}$. If \mathbb{T}_a^r is non-empty, then $\chi(\mathbb{T}_a^r, \Delta) = \mathbb{T}_a^r$.*

One difficulty is ensuring that the processes do output incorrect values. To do so, the adversary terminates a process with output value a at a particular state only if the vertex v in \mathbb{S}^t corresponding to its state contains a , i.e. the process has seen a , and v is sufficiently far from any vertex that has terminated with a different value. When the adversary performs a subdivision, the distance increases between terminated vertices that output different values. Hence, the adversary is able to terminate more vertices. This ensures that every chain of queries is finite. Finally, it can be shown that, if the adversary manages to ensure these conditions throughout the first phase, then the prover is doomed to

lose. This is because the prover must commit to a non-empty schedule at the end of the first phase, which corresponds to some subcomplex \mathbb{Q} of \mathbb{S}^r , for some $1 \leq r \leq t$. Since the outputs are far apart in the current subdivision $\chi^{t-r}(\mathbb{Q}, \Delta)$ of \mathbb{Q} , the adversary can terminate all active vertices in $\chi^{t-r}(\mathbb{Q}, \Delta)$. Hence, after a finite number of phases, the prover commits to an extension that results in a configuration (corresponding to a simplex in $\chi^{t-r}(\mathbb{Q}, \Delta)$) in which all processes have terminated.

Adversarial Strategy in Phase 1. We define an adversarial strategy so that, before and after each query made by the prover in phase 1, the adversary is able to maintain the following invariants:

1. For each $0 \leq r < t$ and each vertex $v \in \mathbb{S}^r$, $\Delta(v)$ is defined. If v is a vertex in \mathbb{S}^t , then either $\Delta(v)$ is undefined or $\Delta(v) \neq \perp$. If s is the state of a process in a configuration that was reached by the prover, then $\{s\}$ is a vertex in \mathbb{S}^r , for some $0 \leq r \leq t$, and $\Delta(s)$ is defined.
2. For any input a , if \mathbb{T}_a^t is non-empty, then $\text{dist}_{\mathbb{S}^t}(\mathbb{T}_a^t, \mathbb{N}_a^t) \geq 2$.
3. For any two inputs $a \neq b$, if \mathbb{T}_a^t and \mathbb{T}_b^t are non-empty, then $\text{dist}_{\mathbb{S}^t}(\mathbb{T}_a^t, \mathbb{T}_b^t) \geq 3$.

We note that there is nothing special about the values 2 and 3. They are simply the smallest values such that the invariant can be maintained and every chain of queries is finite.

The following lemma is a consequence of the invariants. In particular, it says that, after a subdivision, the distance between vertices that output different values increases and the distance between vertices that output a and vertices that do not contain a increases.

Lemma 7.8. *For any two inputs $a \neq b$, if \mathbb{T}_a^t is non-empty, then any path between \mathbb{T}_a^t and $\mathbb{T}_b^t \cup \mathbb{N}_a^t$ in \mathbb{S}^t contains at least one edge between active vertices. Moreover, if the adversary defines $\Delta(v) = \perp$ for each vertex $v \in \mathbb{S}^t$ where $\Delta(v)$ is undefined, subdivides \mathbb{S}^t to construct \mathbb{S}^{t+1} , and \mathbb{T}_b^t is non-empty, then $\text{dist}_{\mathbb{S}^{t+1}}(\mathbb{T}_a^{t+1}, \mathbb{T}_b^{t+1}) \geq \text{dist}_{\mathbb{S}^t}(\mathbb{T}_a^t, \mathbb{T}_b^t) + 1$ and $\text{dist}_{\mathbb{S}^{t+1}}(\mathbb{T}_a^{t+1}, \mathbb{N}_a^{t+1}) \geq \text{dist}_{\mathbb{S}^t}(\mathbb{T}_a^t, \mathbb{N}_a^t) + 1$.*

Proof. Consider any path v_0, v_1, \dots, v_ℓ between \mathbb{T}_a^t and $\mathbb{T}_b^t \cup \mathbb{N}_a^t$ in \mathbb{S}^{t_0} . Let v_j be the last vertex in \mathbb{T}_a^t . Since the invariants hold after each query and $v_\ell \in \mathbb{T}_b^t \cup \mathbb{N}_a^t$, invariants (3) and (2) imply that the distance between v_j and v_ℓ is at least 2. Hence, $\ell \geq j + 2$. Since v_j is the last vertex in \mathbb{T}_a^t , v_{j+1} and v_{j+2} are not in \mathbb{T}_a^t . Moreover, by invariant (3), v_{j+1} and v_{j+2} are not in \mathbb{T}_c^t for any input $c \neq a$. Hence, $\{v_{j+1}, v_{j+2}\}$ is an edge between active vertices.

Suppose the adversary defines $\Delta(v) = \perp$ for each vertex $v \in \mathbb{S}^t$ where $\Delta(v)$ is undefined and subdivides \mathbb{S}^t to construct \mathbb{S}^{t+1} . Then, by Proposition 7.7, for each input a , $\mathbb{T}_a^t = \chi(\mathbb{T}_a^t, \Delta)$ and $\mathbb{N}_a^{t+1} = \chi(\mathbb{N}_a^t, \Delta)$. Since the adversary does not terminate any new vertices, $\mathbb{T}_a^{t+1} = \mathbb{T}_a^t$. Therefore, by the second part of Lemma 7.5, $\text{dist}_{\mathbb{S}^{t+1}}(\mathbb{T}_a^{t+1}, \mathbb{T}_b^{t+1}) \geq \text{dist}_{\mathbb{S}^t}(\mathbb{T}_a^t, \mathbb{T}_b^t) + 1$ and $\text{dist}_{\mathbb{S}^{t+1}}(\mathbb{T}_a^{t+1}, \mathbb{N}_a^{t+1}) \geq \text{dist}_{\mathbb{S}^t}(\mathbb{T}_a^t, \mathbb{N}_a^t) + 1$. \square

We now describe the adversarial strategy and show that the invariants are maintained. Initially, the adversary sets $\Delta(v) = \perp$ for each vertex $v \in \mathbb{S}^0$. It then subdivides \mathbb{S}^0 to construct \mathbb{S}^1 and sets $t = 1$. No vertices in \mathbb{S}^0 have terminated, so \mathbb{T}_a^0 is empty for each input a . Before the first query, the prover has only reached initial configurations. Hence, the invariants are satisfied.

Now suppose the invariants are satisfied immediately prior to a query (C, P) by the prover, where C is a configuration previously reached by the prover and P is a set of active processes in C poised to access the same snapshot object. Let D be the configuration resulting from scheduling P from C . Since each process in P is poised to access the same snapshot object, by invariant (1), there exists $0 \leq r \leq t$

such that the state of each process in P in configuration C corresponds to a vertex in \mathbb{S}^r . Since each process in P is active in C , $\Delta(v) = \perp$ for each such vertex v . Hence, by invariant (1), $r < t$. If $r < t - 1$, then invariant (1) implies that Δ is defined for each vertex corresponding to the state of a process in D . Hence, the adversary does not need to do anything.

So, suppose that $r = t - 1$. Let σ denote the simplex in \mathbb{S}^t whose vertices represent the states of processes in P in D . For each vertex $v \in \sigma$, if $\Delta(v)$ is undefined, the adversary defines $\Delta(v)$ as follows. If there exists an input a such that the distance between v and \mathbb{N}_a^t in \mathbb{S}^t is at least 2 and the distance between v and \mathbb{T}_b^t in \mathbb{S}^t is at least 3, for all inputs $b \neq a$, then the adversary sets $\Delta(v) = a$. Otherwise, the adversary sets $\Delta(v) = \perp$. This ensures that invariants (2) and (3) continue to hold.

If $\Delta(v) \neq \perp$ for every vertex $v \in \sigma$, then invariant (1) holds. Otherwise, the adversary defines $\Delta(v) = \perp$ for each vertex $v \in \mathbb{S}^t$ where $\Delta(v)$ is undefined, subdivides \mathbb{S}^t to construct \mathbb{S}^{t+1} , and increments t . Then invariant (1) holds. By Lemma 7.8, invariants (2) and (3) continue to hold.

Therefore, the invariants hold after the prover's query.

The prover does not win in phase 1. Suppose that the invariants all hold before and after each query made by the prover in phase 1. By invariant (3), at most one value is output in any configuration reached by the prover. Moreover, by invariant (2), if a process outputs a value y , then y is a value that it has seen in one of its scans. Hence, y is the input of some process. So, the prover cannot win in phase 1 by showing that the protocol violates agreement or validity. It remains to show that the prover cannot win by constructing an infinite chain of queries in phase 1.

Lemma 7.9. *Every chain of queries in phase 1 is finite.*

Proof. Assume, for a contradiction, that there is an infinite chain of queries, (C_j, P_j) , for $j \geq 0$. Let P be the set of processes that are scheduled infinitely often. Then, there exists $j_0 \geq 0$ such that, for all $j \geq j_0$, $P_j \subseteq P$. Let $t_0 \geq 1$ be the value of t held by the adversary immediately prior to query (C_{j_0}, P_{j_0}) . By invariant (1), the state of each process in C_{j_0} corresponds to a vertex $v \in \mathbb{S}^r$, for some $0 \leq r \leq t_0$. Hence, no process has accessed S_{t_0+1} in C_{j_0} and, during this chain of queries, only processes in P access S_t for $t > t_0$. Since the processes in P eventually access S_{r+1} for all $r \geq t_0$, and no process in P ever terminates, the adversary eventually defines $\Delta(v) = \perp$ for each vertex $v \in \mathbb{S}^r$ where $\Delta(v)$ is undefined and subdivides \mathbb{S}^r to construct \mathbb{S}^{r+1} , for all $r \geq t_0$.

Consider the first $j_1 \geq j_0$ such that each process in P_{j_1} is poised to access S_{t_0+2} in C_{j_1} , i.e. P_{j_1} is the first set of processes to access S_{t_0+2} in the chain of queries. By definition of \mathbb{S}^{t_0+2} , the set of states of the processes in P_{j_1} in C_{j_1} correspond to a simplex σ_1 in \mathbb{S}^{t_0+2} . Since the adversary does not terminate any new processes, $\mathbb{T}_a^t = \mathbb{T}_a^{t_0}$, for any input a and any $t \geq t_0$. Thus, by applying Lemma 7.8 twice, for any inputs $a \neq b$, whenever $\mathbb{T}_a^{t_0}$ and $\mathbb{T}_b^{t_0}$ are non-empty, $\text{dist}_{\mathbb{S}^{t_0+2}}(\mathbb{T}_a^{t_0+2}, \mathbb{T}_b^{t_0+2}) \geq 5$ and $\text{dist}_{\mathbb{S}^{t_0+2}}(\mathbb{T}_a^{t_0+2}, \mathbb{N}_a^{t_0+2}) \geq 4$.

If there is a vertex $v \in \sigma_1$ that has distance at most 2 to some vertex in $\mathbb{T}_a^{t_0+2}$ in \mathbb{S}^{t_0+2} , for some input a , then the distance from v to $\mathbb{N}_a^{t_0+1}$ is at least 2 and the distance from v to non-empty $\mathbb{T}_b^{t_0+1}$ in \mathbb{S}^{t_0+1} is at least 3, for all $b \neq a$. Hence the adversary defines $\Delta(v) \neq \perp$ after query (C_{j_1}, P_{j_1}) , i.e. some process in $P_{j_1} \subseteq P$ terminates. This is a contradiction.

So, each vertex in σ_1 has distance at least 3 to $\mathbb{T}_a^{t_0+2}$, for all inputs a where $\mathbb{T}_a^{t_0+2}$ is non-empty. Consider the first $j_2 > j_1$ such that each process in P_{j_2} is poised to access S_{t_0+3} in C_{j_2} . Let P' be the set of processes that have accessed S_{t_0+2} in C_{j_2} . Since each process in $P_{j_1} \cup P_{j_2}$ has already accessed S_{t_0+2} , $P_{j_1} \cup P_{j_2} \subseteq P'$. Hence, the states of P' in C_{j_2} forms a simplex σ_2 in \mathbb{S}^{t_0+2} and $\sigma_1 \subseteq \sigma_2$. Since

every vertex in σ_2 has distance at most 1 from every vertex in σ_1 , it follows that each vertex in σ_2 has distance at least 2 to $\mathbb{T}_a^{t_0+2}$, for all inputs a where $\mathbb{T}_a^{t_0+2}$ is non-empty.

Let a be the input of any process in P_{j_1} . Since P_{j_1} is the first set of processes to access S_{t_0+2} and each process in P' has accessed S_{t_0+2} , each vertex in σ_2 contains a and $\text{dist}_{S_{t_0+2}}(\sigma_2, \mathbb{N}_a^{t_0+2}) \geq 1$. Since the distance between σ_2 and any terminated vertex in S^{t_0+2} is at least 2, any path from a vertex $v \in \sigma_2$ to a vertex in $\mathbb{N}_a^{t_0+1} \cup \bigcup_{b \neq a} \mathbb{T}_b^{t_0+1}$ must contain at least one edge between active vertices (specifically, between v and one of its neighbours). By Lemma 7.5 and Proposition 7.7, it follows that $\text{dist}_{S^{t_0+3}}(\chi(\sigma_2, \Delta), \mathbb{N}_a^{t_0+3}) \geq 2$ and $\text{dist}_{S^{t_0+3}}(\chi(\sigma_2, \Delta), \mathbb{T}_b^{t_0+3}) \geq 3$, for any input $b \neq a$ where $\mathbb{T}_b^{t_0+2} = \mathbb{T}_b^{t_0+3}$ is non-empty. The state of each process in P_{j_2} in C_{j_2+1} corresponds to a vertex v in $\chi(\sigma_2, \Delta)$. Hence, the adversary defines $\Delta(v) \neq \perp$ for at least one such vertex v , i.e. some process in $P_{j_2} \subseteq P$ terminates. This is a contradiction. \square

The prover loses. Since the prover does not win in phase 1, eventually phase 1 must end. At the end of phase 1, the prover must choose a configuration $C \in \mathcal{A}'(1)$. This determines the set of configurations, $\mathcal{A}(2)$, that the prover can initially query in phase 2. The adversary will update Δ one final time. Afterwards, it can answer all future queries by the prover. The prover will eventually be forced to choose a terminal configuration at the end of some future phase and, consequently, will lose in the next phase.

C is a configuration reached by a non-empty schedule $\alpha(2)$ from an initial configuration $C_0 \in \mathcal{A}(1)$. Let P be the first set of processes in $\alpha(2)$ and let a be the input of some process $p \in P$ at C_0 . By invariant (1), the state of each process in P at C_0 is represented by a vertex in \mathbb{S}^1 . Since C is a reachable configuration, there is a simplex $\sigma \subseteq \mathbb{S}^1$ with these vertices.

Let \mathbb{Q} be the subcomplex of \mathbb{S}^1 consisting of all subsets in \mathbb{S}^1 that only contain vertices which are at distance at most 1 from every vertex in σ . Then \mathbb{Q} contains every simplex corresponding to a configuration reachable by a full 1-round schedule α' from some initial configuration C'_0 such that $C'_0 \alpha'$ and $C_0 P$ are indistinguishable to P . In particular, P is the first set of processes in α' and p has input a in C'_0 . It follows that the state of every process contains a in $C'_0 \alpha'$. Hence, the distance between \mathbb{Q} and \mathbb{N}_a^1 in \mathbb{S}^1 is at least 1.

Consider the value of t held by the adversary at the end of phase 1. If $t > 1$, then, applying Lemma 7.5 and Proposition 7.7 ($t - 1$) times, it follows that $\text{dist}_{S^t}(\chi^{t-1}(\mathbb{Q}, \Delta), \mathbb{N}_a^t) \geq 1$, where $\chi^{t-1}(\mathbb{Q}, \Delta) \subseteq S^t$ denotes the simplicial complex obtained by performing $t - 1$ non-uniform chromatic subdivisions of \mathbb{Q} .

By invariant (1), for each vertex $v \in S^t$, $\Delta(v)$ is either undefined or $\Delta(v) \neq \perp$. By invariant (3), each n -vertex simplex in S^t represents a configuration in which all processes that have terminated have output the same value. For each vertex $v \in \chi^{t-1}(\mathbb{Q}, \Delta)$ where $\Delta(v)$ is undefined, the adversary sets $\Delta(v) = a$. This is a valid output for v since $\text{dist}_{S^t}(\chi^{t-1}(\mathbb{Q}, \Delta), \mathbb{N}_a^t) \geq 1$. Then each vertex in $\chi^{t-1}(\mathbb{Q}, \Delta)$ has terminated. Moreover, each n -vertex simplex in S^t represents a reachable configuration in which the processes have output at most $2 \leq k$ different values.

In phases $\varphi \geq 2$, the prover can only query configurations reachable from some configuration in $\mathcal{A}(2)$. By definition, $\mathcal{A}(2)$ is the set of all configurations that are reached by performing $\alpha(2)$ from initial configurations C'_0 that only differ from C_0 in the inputs of processes that do not appear in $\alpha(2)$. It follows that, for any process q and any extension α' of $\alpha(2)$ from $C' \in \mathcal{A}(2)$, q appears at most t times in $\alpha(2)\alpha'$ before its state is represented by a vertex in $\chi^{t-1}(\mathbb{Q}, \Delta)$. Note that, by construction, every vertex in $\chi^{t-1}(\mathbb{Q}, \Delta)$ has terminated. Thus, eventually, the prover chooses a configuration at the

end of some phase in which every process has terminated. The prover loses in the next phase.

7.5 Extension-Based Proofs

In this section, we extend the definition of a restricted extension-based proof to include output queries, explain how the adversarial protocol can respond to these queries, and extend the proof in Section 7.4. Roughly speaking, output queries allow the prover to perform a valency argument.

An *output query* in phase φ is specified by a configuration $C \in \mathcal{A}(\varphi) \cup \mathcal{A}'(\varphi)$, a set of active processes P in C that are poised to access the same snapshot object, and a value $y \in \{0, 1, \dots, k\}$. If there is a schedule from C involving only processes in P , i.e. a *P-only* schedule, that results in a configuration in which some process in P outputs y , then the protocol returns some such schedule. Otherwise, the protocol returns NONE. In each phase, the prover is allowed to make finitely many output queries in addition to finitely many chains of queries.

For example, if P is the set of all processes, then the sequence of output queries $(C, P, 0)$, $(C, P, 1)$, \dots , (C, P, k) enables the prover to determine which values can be output by the processes when they are scheduled starting from C . In particular, the prover can determine if it is only possible to output one value starting from C , i.e. if C is *univalent*.

Responding to output queries in phase 1. Suppose that the invariants hold prior to an output query (C, P, y) in phase 1. We show that the adversary can answer the output query so that it never conflicts with the result of any future query made in phase 1, while still maintaining the invariants.

By definition, each process in P is poised to access the same snapshot object S_r in C , for some $r \geq 0$. By invariant (1), $r \leq t$, where t is the value held by the adversary immediately prior to the query. Let \mathbb{V} be the subcomplex of \mathbb{S}^t consisting of all subsets in \mathbb{S}^t that only contain vertices representing the state of a process in P in a configuration C' reachable from C by a *P-only* $(t-r)$ -round schedule from C . In particular, \mathbb{V} contains the possible states of each process in P after it has been selected t times (or it has terminated) in some configuration C' reachable from C by a *P-only* schedule.

If some vertex $v \in \mathbb{V}$ has terminated with output y , then the adversary returns a *P-only* $(t-r)$ -round schedule from C that leads to a configuration C' such that v represents the state of a process in C' . If every vertex in \mathbb{V} has terminated and none have output y , then the adversary returns NONE. Since the adversary never changes $\Delta(v)$ once it has been set, these responses do not conflict with the result of any future query. In both cases, the invariants continue to hold.

Now suppose that no vertex in \mathbb{V} has terminated with output y and \mathbb{V} contains at least one vertex that has not terminated. Let $\mathbb{U} \neq \emptyset$ be the subcomplex of \mathbb{V} consisting of all subsets in \mathbb{V} that only contain vertices that have not terminated (i.e. Δ is currently undefined for these vertices). For each simplex $\sigma \subseteq \mathbb{U}$, let \mathbb{A}_σ be the subcomplex of \mathbb{S}^t consisting of all subsets in \mathbb{S}^t that only contain vertices at distance at most 1 to each vertex in σ . We consider a number of cases.

Case 1: *There is a simplex $\sigma \subseteq \mathbb{U}$ such that $\sigma \not\subseteq \mathbb{N}_y^t$ and no vertex in \mathbb{A}_σ has terminated.* Then the adversary subdivides \mathbb{S}^t twice, i.e. it defines $\Delta(v) = \perp$ for each $v \in \mathbb{S}^t$ where $\Delta(v)$ is undefined, subdivides \mathbb{S}^t to construct \mathbb{S}^{t+1} , defines $\Delta(v) = \perp$ for each $v \in \mathbb{S}^{t+1}$ where $\Delta(v)$ is undefined, and subdivides \mathbb{S}^{t+1} to construct \mathbb{S}^{t+2} .

Consider the simplex $\rho \subseteq \chi(\sigma, \Delta)$ consisting of all subsets of $\{(i, \vec{\sigma}) : i \in \text{Id}(\sigma)\}$. Since $\sigma \not\subseteq \mathbb{N}_y^t$, some vertex in σ contains y , so each vertex in ρ contains y . Hence, $\text{dist}_{\mathbb{S}^{t+1}}(\rho, \mathbb{N}_y^{t+1}) \geq 1$. Since each

vertex in \mathbb{A}_σ is active, $\chi(\mathbb{A}_\sigma, \Delta)$ is the standard chromatic subdivision of \mathbb{A}_σ . It follows that every path between ρ and $\mathbb{T}_a^{t+1} \cup \mathbb{N}_y^{t+1}$ in \mathbb{S}^{t+1} , for inputs $a \neq y$, contains at least one edge between a vertex in ρ and one of its neighbours in $\chi(\mathbb{A}_\sigma, \Delta)$, which is an edge between active vertices. Since no vertex in \mathbb{T}_a^{t+1} is active, $\text{dist}_{\mathbb{S}^{t+1}}(\rho, \mathbb{T}_a^{t+1}) \geq 2$, for all inputs a . By Lemma 7.5 and Proposition 7.7, it follows that $\text{dist}_{\mathbb{S}^{t+2}}(\chi(\rho, \Delta), \mathbb{N}_y^{t+2}) \geq 2$ and $\text{dist}_{\mathbb{S}^{t+2}}(\chi(\rho, \Delta), \mathbb{T}_a^{t+2}) \geq 3$, for all inputs a .

The adversary defines $\Delta(v) = y$, for one vertex $v \in \chi(\rho, \Delta)$, returns a P -only $(t+2-r)$ -round schedule from C that leads to a configuration C' such that v represents the state of a process in C' , and sets t to $t+2$. By Lemma 7.8, invariants (2) and (3) are not violated by the subdivisions and increments of t . Since $\Delta(v) \neq \perp$, invariant (1) continues to hold and, since $\text{dist}_{\mathbb{S}^t}(v, \mathbb{N}_y^t) \geq 2$ and $\text{dist}_{\mathbb{S}^t}(v, \mathbb{T}_a^t) \geq 3$, for all inputs $a \neq y$, invariants (2) and (3) continue to hold.

Case 2: *There is a simplex $\sigma \subseteq \mathbb{U}$ such that $\sigma \not\subseteq \mathbb{N}_y^t$ and there is a vertex $w \in \mathbb{A}_\sigma$ such that w has terminated with output y .* Then the adversary subdivides \mathbb{S}^t once, i.e. it defines $\Delta(v) = \perp$ for each vertex $v \in \mathbb{S}^t$ where $\Delta(v)$ is undefined and subdivides \mathbb{S}^t to construct \mathbb{S}^{t+1} . Note that $w \in \mathbb{T}_y^t = \mathbb{T}_y^{t+1}$, so \mathbb{T}_y^t is non-empty. By invariant (2) and Lemma 7.8, $\text{dist}_{\mathbb{S}^{t+1}}(\mathbb{T}_y^{t+1}, \mathbb{N}_y^{t+1}) \geq 3$. By invariant (3) and Lemma 7.8, $\text{dist}_{\mathbb{S}^{t+1}}(\mathbb{T}_y^{t+1}, \mathbb{T}_a^{t+1}) \geq 4$ for all inputs $a \neq y$ such that \mathbb{T}_a^t is non-empty. Since w has terminated and $w \in \mathbb{A}_\sigma$, by Proposition 7.3, w is adjacent to every vertex in $\chi(\sigma, \Delta) \subseteq \mathbb{S}^{t+1}$. It follows that $\text{dist}_{\mathbb{S}^{t+1}}(\chi(\sigma, \Delta), \mathbb{N}_y^{t+1}) \geq 2$ and $\text{dist}_{\mathbb{S}^{t+1}}(\chi(\sigma, \Delta), \mathbb{T}_a^{t+1}) \geq 3$, for all inputs $a \neq y$ such that $\mathbb{T}_a^{t+1} = \mathbb{T}_a^t$ is non-empty.

The adversary defines $\Delta(v) = y$, for a vertex $v \in \chi(\sigma, \Delta)$, returns a P -only $(t+1-r)$ -round schedule from C that leads to a configuration C' such that v represents the state of a process in C' , and increments t . By Lemma 7.8, invariants (2) and (3) are not violated by the subdivision and increment of t . Since $\Delta(v) \neq \perp$, invariant (1) continues to hold and, since $\text{dist}_{\mathbb{S}^t}(v, \mathbb{N}_y^t) \geq 2$ and $\text{dist}_{\mathbb{S}^t}(v, \mathbb{T}_a^t) \geq 3$, for all inputs $a \neq y$ such that \mathbb{T}_a^t is non-empty, invariants (2) and (3) continue to hold.

Case 3. *For every simplex $\sigma \subseteq \mathbb{U}$, either $\sigma \subseteq \mathbb{N}_y^t$ or some vertex $w \in \mathbb{A}_\sigma$ has terminated with an output $a \neq y$.* In this case, the adversary returns NONE. If $\sigma \subseteq \mathbb{N}_y^t$, then no vertex in σ contains y and, by Proposition 7.7, no vertex in any subdivision of σ contains y . Hence, by invariant (2), the adversary never terminates any vertex in σ (or a future subdivision of σ) with output y as the result of a future query (in any phase). If some vertex $w \in \mathbb{A}_\sigma$ has terminated with output $a \neq y$, then, since w is adjacent to each vertex in σ , w is adjacent to each vertex in any subdivision of σ (Proposition 7.3). Since invariant (3) holds, the adversary never terminates such vertices with any output other than a as the result of a future query in phase 1. In both cases, the invariants continue to hold.

The prover still loses. Since the adversary is able to maintain the invariants before and after each query, as in Section 7.4, the prover does not win in phase 1 and must eventually choose a configuration $C \in \mathcal{A}'(1)$ at the end of phase 1. The adversary will update Δ one final time. Afterwards, it can answer all future queries by the prover. The prover will eventually be forced to choose a terminal configuration at the end of some future phase and, consequently, will lose in the next phase.

C is a configuration reached by a non-empty schedule $\alpha(2)$ from an initial configuration $C_0 \in \mathcal{A}(1)$. Let P be the first set of processes in $\alpha(2)$ and let a be the input of some process $p \in P$ at C_0 . By invariant (1), the state of each process in P at C_0P is represented by a vertex in \mathbb{S}^1 . Since C is a reachable configuration, there is a simplex $\sigma \subseteq \mathbb{S}^1$ with these vertices.

Let \mathbb{Q} be the subcomplex of \mathbb{S}^1 consisting of all subsets in \mathbb{S}^1 that only contain vertices which are at distance at most 1 from every vertex in σ . Then \mathbb{Q} contains every simplex corresponding to a

configuration reachable by a full 1-round schedule α' from some initial configuration C'_0 such that $C'_0\alpha'$ and C_0P are indistinguishable to P . In particular, P is the first set of processes in α' and p has input a in C'_0 . It follows that the state of every process contains a in $C'_0\alpha'$. Hence, the distance between \mathbb{Q} and \mathbb{N}_a^1 in \mathbb{S}^1 is at least 1.

Consider the value of t held by the adversary at the end of phase 1. If $t > 1$, then, applying Lemma 7.5 and Proposition 7.7 ($t - 1$) times, it follows that $\text{dist}_{\mathbb{S}^t}(\chi^{t-1}(\mathbb{Q}, \Delta), \mathbb{N}_a^t) \geq 1$, where $\chi^{t-1}(\mathbb{Q}, \Delta) \subseteq \mathbb{S}^t$ denotes the simplicial complex obtained by performing $t - 1$ non-uniform chromatic subdivisions of \mathbb{Q} .

To ensure that the prover loses, we have to modify the adversary's strategy at the end of phase 1 in Section 7.4 slightly. In particular, when the adversary defines $\Delta(v)$ for each vertex in $\chi^{t-1}(\mathbb{Q}, \Delta)$, it cannot simply set $\Delta(v) = a$, where a is an input value contained in each vertex of $\chi^{t-1}(\mathbb{Q}, \Delta)$. The problem is that the adversary may have answered NONE to an output query (C, P, a) where not all vertices have terminated. This can only occur in Case 3. Instead, the adversary does the following.

1. It subdivides \mathbb{S}^t twice to construct \mathbb{S}^{t+2} and sets t to $t + 2$.
2. Let T be the set of all terminated vertices in \mathbb{S}^t . For each terminated vertex $w \in T$ and each vertex $v \in \chi^{t-1}(\mathbb{Q}, \Delta) \subseteq \mathbb{S}^t$ that is adjacent to w , if $\Delta(v)$ is undefined, then it sets $\Delta(v) = \Delta(w)$.
3. For each vertex $v \in \chi^{t-1}(\mathbb{Q}, \Delta)$, if $\Delta(v)$ is undefined, then it sets $\Delta(v) = a$.

Immediately before step (1), invariants (3) and (2) imply that $\text{dist}_{\mathbb{S}^t}(\mathbb{T}_b^t, \mathbb{T}_d^t) \geq 3$ and $\text{dist}_{\mathbb{S}^t}(\mathbb{T}_b^t, \mathbb{N}_b^t) \geq 2$ for any inputs $b \neq d$ such that \mathbb{T}_b^t and \mathbb{T}_d^t are non-empty. By Lemma 7.8, immediately after step (1), $\text{dist}_{\mathbb{S}^t}(\mathbb{T}_b^t, \mathbb{T}_d^t) \geq 5$ and $\text{dist}_{\mathbb{S}^t}(\mathbb{T}_b^t, \mathbb{N}_b^t) \geq 4$ for any inputs $b \neq d$ such that \mathbb{T}_b^t and \mathbb{T}_d^t are non-empty. Consequently, immediately after step (1), if a vertex $v \in \mathbb{S}^t$ is adjacent to a vertex $w \in \mathbb{T}_b^t$, then it is not adjacent to any vertex in \mathbb{T}_d^t , for $d \neq b$.

Since invariant (1) holds immediately before step (1) and the adversary does not set $\Delta(v) = \perp$ for any vertex $v \in \mathbb{S}^t$ during step (2), invariant (1) continues to hold immediately after step (2).

Consider any vertex $u \in \chi^{t-1}(\mathbb{Q}, \Delta)$ such that the adversary sets $\Delta(u) = b$ in step (2). Then u is adjacent to a vertex $w \in \mathbb{T}_b^t \cap T$. Since $\text{dist}_{\mathbb{S}^t}(w, \mathbb{N}_b^t) \geq 4$, it follows that $\text{dist}_{\mathbb{S}^t}(u, \mathbb{N}_b^t) \geq 3$ and b is a valid output for u . Thus invariant (2) holds immediately after step (2).

Now consider any vertex $v \in \mathbb{T}_d^t$, where $d \neq b$. If $v \notin T$, then v is adjacent to a vertex $w' \in \mathbb{T}_d^t \cap T$. Since $\text{dist}_{\mathbb{S}^t}(w, w') \geq 5$, it follows that $\text{dist}_{\mathbb{S}^t}(u, v) \geq 3$. If $v \in T$, then $\text{dist}_{\mathbb{S}^t}(w, v) \geq 5$ and, hence, $\text{dist}_{\mathbb{S}^t}(u, v) \geq 4$. Thus invariant (3) holds immediately after step (2).

Next, we show that the changes to Δ do not violate the response to any output query made in phase 1.

Lemma 7.10. *No output queries made in phase 1 are ever violated.*

Proof. Let (C, P, y) be an output query made in phase 1. By definition, each process in P is poised to access the same snapshot object S_r in C , for some $r \geq 1$. By invariant (1), $r \leq t$, where t is the value held by the adversary immediately before the output query. Let \mathbb{V} be the subcomplex of \mathbb{S}^t consisting of all subsets in \mathbb{S}^t that only contain vertices which represent the state of a process in P in a configuration C' reachable from C by a P -only $(t-r)$ -round schedule from C . Let \mathbb{U} be the subcomplex of \mathbb{V} consisting of all subsets in \mathbb{V} that only contain vertices that have not terminated.

If the adversary answers with a P -only schedule α , then there is a vertex $v \in \mathbb{S}^{t'}$ that represents the state of a process in $C\alpha$ such that $\Delta(v) = y$, where $t' \geq t$ is the value held by the adversary immediately

after the output query. Since the adversary never changes $\Delta(v)$ once it has been defined, the adversary will never violate its response to the output query.

If every vertex in \mathbb{V} has terminated and none have output y , then the adversary answers NONE. In this case, it will never violate its response to the output query, because it never changes $\Delta(v)$ for any vertex v once it has been defined.

The remaining case is when the adversary answers NONE and for every simplex σ in \mathbb{U} , either σ is contained in \mathbb{N}_y^t or there is a vertex in \mathbb{A}_σ that has terminated with an output $b \neq y$, where \mathbb{A}_σ is the subcomplex of \mathbb{S}^t consisting of all subsets in \mathbb{S}^t that only contain vertices at distance at most 1 from every vertex in σ . Consider any simplex $\sigma \subseteq \mathbb{U}$ and its subdivision σ' in the complex constructed by the adversary at the end of phase 1. We show that the adversary never terminates any vertex in σ' with output y . Hence it does not violate its response to the output query (C, P, y) .

Case 1. $\sigma \subseteq \mathbb{N}_y^t$. Then no vertex in σ contains y and, hence, no vertex $v \in \sigma'$ contains y . If the adversary set $\Delta(v) \neq \perp$ prior to step (3) at the end of phase 1, then, by invariant (2), $\Delta(v) \neq y$. If the adversary sets $\Delta(v) = a$ in step (3), then a is contained in v (by construction) and, hence, $\Delta(v) \neq y$. Note the adversary never modifies Δ after step (3).

Case 2. $\sigma \not\subseteq \mathbb{N}_y^t$. Then \mathbb{A}_σ contains a vertex w that has terminated with $\Delta(w) \neq y$. Since w has terminated and is adjacent to every vertex in σ , it follows by Proposition 7.3 that w is adjacent to every vertex $v \in \sigma'$. If the adversary set $\Delta(v) \neq \perp$ prior to step (3) at the end of phase 1, then, by invariant (3), $\Delta(v) = \Delta(w) \neq y$. Notice that, since v is adjacent to $w \in \mathbb{T}_y^t$, if $\Delta(v)$ is not set in step (2), then it is not set in step (3). \square

Since no output query is violated and the invariants hold prior to step (3), the rest of the argument in Section 7.4 is unchanged. In particular, the prover must commit to a terminal configuration in $\chi^{t-1}(\mathbb{Q}, \Delta)$ at the end of some phase and, hence, loses in the next phase.

Thus, we have shown that there is no extension-based proof of the impossibility of deterministically solving k -set agreement in a wait-free manner in the NIIS model for $n > k \geq 2$ processes. Since there is a deterministic, wait-free protocol for a task in the NIIS model if and only if there is a deterministic, wait-free protocol for the task using only registers, we have proved our main result.

Theorem 7.11. *No extension-based proof can show the impossibility of a deterministic, wait-free protocol for k -set agreement among $n > k \geq 2$ processes using only registers.*

Chapter 8

Conclusions and Future Work

In Chapter 3, we proved that a space lower bound for obstruction-free protocols implies a space lower bound for protocols that satisfy nondeterministic solo-termination (including randomized wait-free protocols) by converting any nondeterministic solo-terminating protocol to an obstruction-free protocol that uses the same number of registers. This allows researchers to focus on deriving space lower bounds for obstruction-free protocols.

Our construction in Chapter 3 provides no bound on the solo step complexity of the resulting obstruction-free protocol in terms of the expected step complexity of the randomized wait-free protocol. It would be interesting to improve the construction to bound the solo step complexity of the resulting protocols. It would also be interesting to extend the result to protocols using objects that do not support `read` (such as queues and stacks).

We have given two lower bounds on the number of registers used by any obstruction-free protocol solving consensus. The proof in Chapter 4 is simple, uses traditional techniques, and obtains a lower bound of $n - 1$ registers. The proof in Chapter 6 is significantly more complex but obtains a lower bound of n registers, which is tight. Interestingly, neither of our proofs seem to generalize to systems with swap objects. However, the $\Omega(\sqrt{n})$ lower bound of Ellen, Herlihy and Shavit [28] is applicable to these systems. In Chapter 4, we proved that $n - 1$ swap objects suffice to solve obstruction-free consensus. We conjecture that $\Omega(n)$ swap objects are needed to solve obstruction-free consensus.

We conjecture that the space complexity of obstruction-free k -set agreement is $n - k + 1$ registers, matching the upper bounds of [19, 55]. In Chapter 6, we have made significant progress by proving the first non-constant lower bound, $\lceil \frac{n}{k} \rceil$, for non-anonymous processes. Our lower bound is asymptotically tight when k is constant. It is conceivable that revisionist simulations can be further extended to obtain a tight lower bound for general $k > 1$. Moreover, it would be interesting to extend this technique to work for protocols using objects more powerful than registers.

Our simulation theorem (Theorem 6.1) in Chapter 6 only holds for colourless tasks. It would be interesting to prove a similar theorem for general tasks. One approach would be to study the topological structure of protocols in asynchronous systems containing only registers, which is not very well-understood. We believe that doing so would lead to a characterization of the tasks that may be solvable in limited space, similar to Herlihy and Shavit's characterization of tasks that may be solved in a wait-free manner [41].

In Chapter 7, we developed a framework that allows us to show the limitations of certain arguments

for proving impossibility results in asynchronous shared memory systems. Specifically, we defined a class of extension-based proofs, which is formalized as an interaction between a prover and a protocol. We only allowed the prover to make certain types of queries. It would be interesting to consider different or more powerful queries.

We showed that extension-based proofs cannot establish the impossibility of a wait-free solution to k -set agreement, for $n > k > 1$ processes. It is known that the $(2n - 2)$ -renaming problem has no deterministic wait-free solution among $n \geq 3$ processes using registers [22]. We believe there are no extension-based proofs of this result. More generally, it is an open problem to characterize the class of tasks for which there are no extension-based proofs.

We considered extension-based proofs in the NIIS model. However, the definition of an extension-based proof may be generalized to other models, such as the asynchronous shared memory model. We believe that our proof in Chapter 7 may be extended to show that, in systems containing only registers, certain complexity lower bounds cannot be obtained using extension-based proofs. In particular, we conjecture that covering arguments cannot be used to prove a lower bound on the number of registers needed for randomized wait-free solutions to k -set agreement among $n > k \geq 2$ processes that depends on the number of processes.

Bibliography

- [1] Karl Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 291–302, 1988.
- [2] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, September 1993.
- [3] James H. Anderson. Multi-writer composite registers. *Distrib. Comput.*, 7(4):175–195, May 1994.
- [4] James Aspnes. Time- and space-efficient randomized consensus. *J. Algorithms*, 14(3):414–431, May 1993.
- [5] James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2:1–2:24, March 2012.
- [6] James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Limited-use atomic snapshots with polylogarithmic step complexity. *J. ACM*, 62(1):3:1–3:22, March 2015.
- [7] James Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *J. Algorithms*, 11(3):441–461, September 1990.
- [8] Hagit Attiya and Armando Castañeda. A non-topological proof for the impossibility of k -set agreement. *Theor. Comput. Sci.*, 512:41–48, November 2013.
- [9] Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. *J. ACM*, 55(5):20:1–20:26, November 2008.
- [10] Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *J. ACM*, 41(4):725–763, July 1994.
- [11] Hagit Attiya and Ami Paz. Counting-based impossibility proofs for renaming and set agreement. In *Proceedings of the 26th International Conference on Distributed Computing*, DISC '12, pages 356–370, 2012.
- [12] Hagit Attiya and Ophir Rachman. Atomic snapshots in $O(n \log n)$ operations. *SIAM J. Comput.*, 27(2):319–340, April 1998.
- [13] Hagit Attiya and Sergio Rajsbaum. The combinatorial structure of wait-free solvable tasks. *SIAM J. Comput.*, 31(4):1286–1313, April 2002.
- [14] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, Inc., 2004.

- [15] E. Borowsky, E. Gafni, N. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. *Distrib. Comput.*, 14(3):127–146, October 2001.
- [16] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t -resilient asynchronous computations. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, STOC '93, pages 91–100, 1993.
- [17] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, PODC '93, pages 41–51, 1993.
- [18] Elizabeth Borowsky and Eli Gafni. A simple algorithmically reasoned characterization of wait-free computation (extended abstract). In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, PODC '97, pages 189–198, 1997.
- [19] Zohir Bouzid, Michel Raynal, and Pierre Sutra. Anonymous obstruction-free (n, k) -set agreement with $n-k+1$ atomic read/write registers. *Distrib. Comput.*, 31(2):99–117, April 2018.
- [20] Jack R. Bowman. Obstruction-free snapshot, obstruction-free consensus, and fetch-and-add modulo k . Technical Report TR2011-681, Computer Science, Dartmouth College, June 2011.
- [21] James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Inf. Comput.*, 107(2):171–184, December 1993.
- [22] Armando Castañeda and Sergio Rajsbaum. New combinatorial topology bounds for renaming: The upper bound. *J. ACM*, 59(1):3:1–3:49, March 2012.
- [23] Carole Delporte-Gallet, Hugues Fauconnier, Petr Kuznetsov, and Eric Ruppert. On the space complexity of set agreement? In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 271–280, 2015.
- [24] Carole Delporte-Gallet, Hugues Fauconnier, Sergio Rajsbaum, and Nayuta Yanagisawa. A characterization of t -resilient colorless task anonymous solvability. In *International Colloquium on Structural Information and Communication Complexity*, SIROCCO '18, pages 178–192. Springer, 2018.
- [25] Danny Dolev and Nir Shavit. Bounded concurrent time-stamping. *SIAM J. Comput.*, 26(2):418–455, April 1997.
- [26] Faith Ellen, Rati Gelashvili, Nir Shavit, and Leqi Zhu. A complexity-based hierarchy for multi-processor synchronization: [extended abstract]. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 289–298, 2016.
- [27] Faith Ellen, Rati Gelashvili, and Leqi Zhu. Revisionist simulations: A new approach to proving space lower bounds. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 61–70, 2018.
- [28] Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, September 1998.

- [29] Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-free algorithms can be practically wait-free. In *Proceedings of the 19th International Conference on Distributed Computing, DISC '05*, pages 78–92, 2005.
- [30] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, August 1991.
- [31] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.
- [32] Rati Gelashvili. On the optimal space complexity of consensus for anonymous processes. In *Proceedings of the 29th International Symposium on Distributed Computing, DISC '15*, pages 452–466, 2015.
- [33] Rati Gelashvili. *On the complexity of synchronization*. PhD thesis, Massachusetts Institute of Technology, 2017.
- [34] George Giakkoupis, Maryam Helmi, Lisa Higham, and Philipp Woelfel. An $O(\sqrt{n})$ space bound for obstruction-free leader election. In *Proceedings of the 27th International Symposium on Distributed Computing, DISC '13*, pages 46–60, 2013.
- [35] Rachid Guerraoui and Eric Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distrib. Comput.*, 20(3):165–177, October 2007.
- [36] Maurice Herlihy. Impossibility results for asynchronous pram (extended abstract). In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '91*, pages 327–336, 1991.
- [37] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.
- [38] Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann Publishers Inc., 2013.
- [39] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems, ICDCS '03*, pages 522–529, 2003.
- [40] Maurice Herlihy and Sergio Rajsbaum. A classification of wait-free loop agreement tasks. *Theor. Comput. Sci.*, 291(1):55–77, January 2003.
- [41] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, November 1999.
- [42] Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
- [43] Gunnar Hoest and Nir Shavit. Toward a topological characterization of asynchronous complexity. *SIAM J. Comput.*, 36(2):457–497, August 2006.

- [44] Michiko Inoue and Wei Chen. Linear-time snapshot using multi-writer multi-reader registers. In *Proceedings of the 8th International Workshop on Distributed Algorithms*, WDAG '94, pages 130–140, 1994.
- [45] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
- [46] Michael Loui and Hosame Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In F. P. Preparata, editor, *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, 1987.
- [47] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.
- [48] Yoram Moses and Sergio Rajsbaum. A layered analysis of consensus. *SIAM J. Comput.*, 31(4):989–1021, April 2002.
- [49] Achour Mostefaoui, Matthieu Perrin, and Michel Raynal. A simple object that spans the whole consensus hierarchy. *Parallel Processing Letters*, 28(2):1850006, 2018.
- [50] Matthieu Perrin. *Spécification des objets partagés dans les systèmes répartis sans-attente*. PhD thesis, Université de Nantes (Unam), 2016.
- [51] Michael Saks and Fotios Zaharoglou. Wait-free k -set agreement is impossible: The topology of public knowledge. *SIAM J. Comput.*, 29(5):1449–1483, March 2000.
- [52] E. Schenk. Faster approximate agreement with multi-writer registers. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, pages 714–723, 1995.
- [53] Gadi Taubenfeld. Contention-sensitive data structures and algorithms. In *Proceedings of the 23rd International Conference on Distributed Computing*, DISC '09, pages 157–171, 2009.
- [54] Nayuta Yanagisawa. Wait-free solvability of colorless tasks in anonymous shared-memory model. *Theor. Comp. Sys.*, 63(2):219–236, February 2019.
- [55] Leqi Zhu. Brief announcement: Tight space bounds for memoryless anonymous consensus. In *Proceedings of the 29th International Symposium on Distributed Computing*, DISC '15, page 665, 2015.
- [56] Leqi Zhu. A tight space bound for consensus. In *Proceedings of the 48th Annual ACM Symposium on Theory of Computing*, STOC '16, pages 345–350, 2016.