Formal Verification through Combinatorial Topology: the CAS-Extended Model*

Christina Peterson
University of Central Florida
Orlando, Florida
clp8199@knights.ucf.edu

Damian Dechev
University of Central Florida
Orlando, Florida
dechev@cs.ucf.edu

ABSTRACT

Wait-freedom guarantees that all processes complete their operations in a finite number of steps regardless of the delay of any process. Combinatorial topology has been proposed in the literature as a formal verification technique to prove the wait-free computability of decision tasks. Wait-freedom is proved through the properties of a static topological structure that expresses all possible combinations of execution paths of the protocol solving the decision task. The practical application of combinatorial topology as a formal verification technique is limited because the existing theory only considers protocols in which the manner of communication between processes is through read-write memory. This research proposes an extension to the existing theory, called the CAS-extended model. The extended theory includes Compare-And-Swap (CAS) and Load-Linked/Store-Conditional (LL/SC) which are atomic primitives used to achieve wait-freedom in state-of-the-art protocols. The CAS-extended model theory can be used to formally verify wait-free algorithms used in practice, such as concurrent data structures. We present new definitions detailing the construction of a protocol complex in the CAS-extended model. As a proof-of-concept, we formally verify a wait-free queue with three processes using the CAS-extended combinatorial topology.

CCS CONCEPTS

• **Computing methodologies** → **Concurrent programming languages; Concurrent algorithms.**

KEYWORDS

formal verification, combinatorial topology, wait-free

ACM Reference Format:

Christina Peterson and Damian Dechev. 2019. Formal Verification through Combinatorial Topology: the CAS-Extended Model. In *The 10th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'19)*, February 17, 2019, Washington, DC, USA. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3303084.3309493>

*This work is supported by the National Science Foundation under NSF award No. 1717515 and NSF award No. 1740095.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PMAM'19, February 17, 2019, Washington, DC, USA

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6290-0/19/02...\$15.00

<https://doi.org/10.1145/3303084.3309493>

1 INTRODUCTION

Multiprocessing holds the potential for increased performance over a single processor through the utilization of multiple concurrent processes to solve a computational problem. A *decision task* in a multiprocessor system is a problem where each process accepts a private input value, communicates with the other processes in parallel, and halts with a private output value [14]. A *protocol* is a program that solves a decision task. Multiprocessor systems are naturally asynchronous since processes may be halted or delayed due to scheduler pre-emption, hardware interrupts, or software interrupts. *Wait-freedom* guarantees that all processes complete their operations in a finite number of steps regardless of the delay of any process [15]. Such unpredictable process delays make wait-freedom a highly desirable progress property for decision tasks in a multiprocessor system.

The methodology for proving wait-free computability of decision tasks through combinatorial topology was initially introduced by Herlihy et al. [12]. The strength of the combinatorial formalism resides in the ability to prove wait-free computability through the properties of the topological structure. The theory considers protocols in which the only manner of communication between processes is through *read-write memory*, a shared-memory model where processes can read and write to a shared array of memory locations. Although this model demonstrates the power of the theory, it is impractical because it does not encompass atomic primitives for process communication that are stronger than read-write memory.

A *concurrent object* (or simply *object* for brevity) is a shared data structure that can be accessed concurrently by all processes [6]. Herlihy [8] introduced the *consensus number*, which is the largest number n for which a concurrent object can wait-free solve n -thread consensus. The consensus problem is a decision task such that for every process that invokes `decide()`, the decision value must be the same across all participating processes and also be some process's input. Atomic registers have a consensus number 1; FIFO queues and read-modify-write registers have a consensus number of 2. Registers providing Compare-And-Swap (CAS) or Load-Link Store-Conditional (LL/SC) have an infinite consensus number. That is, an implementation of `decide()` using CAS or LL/SC can solve consensus for any arbitrary number of threads.

Herlihy proved that it is impossible to construct a wait-free implementation of an object with consensus number n with an object with a consensus number lower than n [8]. This impossibility result demonstrates the necessity for CAS or LL/SC to be supported on modern architectures. As is the case for many decision tasks, a wait-free protocol does not exist without the use of objects with an infinite consensus number. It follows that if wait-free protocols require a stronger computational model than read-write memory,

then the verification techniques for these protocols require a computational model of equivalent strength.

This research proposes an extension to the existing theory on wait-free computability through combinatorial topology to include atomic primitives such as CAS and LL/SC, referred to as the CAS-extended model. The inclusion of CAS and LL/SC in the CAS-extended model allows combinatorial topology to be a practical formal verification technique for state-of-the-art wait-free protocols. We demonstrate the practical application of the CAS-extended model theory to verify wait-freedom for a class of problems that is more complex than a decision task - the concurrent object. As proven by Herlihy et al. [12], a protocol wait-free solves a decision task if a mapping exists between the protocol complex and the output complex. We verify wait-freedom by showing the existence of a mapping between the protocol complex modeling an algorithm and the output complex. The benefit of the proposed CAS-extended model is that it enables an automatic approach for verifying wait-freedom by constructing and analyzing the protocol complex during model checking, which we reserve for future work. The contributions of the paper include:

- (1) An extension to combinatorial topology theory for wait-free computability to include atomic primitives such as CAS, LL/SC, and atomic Read-Modify-Write (RMW) operations. The inclusion of these operations in the model for combinatorial topology theory is necessary for the verification of wait-freedom because the protocol complex is constructed from the input complex, which is dependent on the operations used by the protocol.
- (2) The introduction of the input complex in the CAS-extended model that captures the successful and unsuccessful write for each process's input values. Since atomic reads/writes are guaranteed to succeed, an input complex modeling CAS and LL/SC is not trivially obtained from the input complex described in [12] because CAS and LL/SC may fail to write to memory.
- (3) A detailed description of the construction of the protocol complex in the CAS-extended model. We present a proof-of-concept example demonstrating the analysis of the protocol complex to formally verify a wait-free queue.

2 BACKGROUND

A *view* is a process's observed state of the shared memory. A *layer* is a computational step in which a process writes its view to shared memory and reads the other processes view of the shared memory [9]. The read-write protocol, detailed in Algorithm 1, allows processes to communicate through a series of N layers. The memory is structured as a one-dimensional array of words, where a process can either read or write an individual word in one atomic step. Multiple atomic steps are required to read or write multiple words, which may be interleaved with the atomic steps of other processes. Process i may read any word but may only write to $\text{mem}[i]$. The read-write protocol accepts a single word v_i that is the input for process i . At each layer l , process i atomically writes its input to $\text{mem}[i]$ at line 7 and atomically copies the memory one word at a time into its view at line 9. This layered communication between processes is performed independent of the decision task. Once a

process has finished copying the memory into its view, it applies a decision mapping defined by the decision task to its final view, shown at line 12.

Algorithm 1 Read-Write Protocol [9]

```

1: // A process may either read or write an individual word in one atomic step
2: //  $i \in \{0, \dots, n\}$  is the set of process names
3: shared mem: array  $[0..n]$  of Value ▷ There are  $n + 1$  processes
4: procedure READWRITEPROTOCOL( $v_i$ : Value) : Value
5:   view: set of Value :=  $\{v_i\}$ 
6:   for  $l := 0$  to  $N - 1$  do
7:      $\text{mem}[i] := v_i$  ▷ Process  $i$  may only write to  $\text{mem}[i]$ 
8:     for  $j := 0$  to  $n$  do
9:        $\text{view}[j] := \text{mem}[j]$ 
10:    end for
11:  end for
12:  return  $\delta(\text{view})$ 
13: end procedure

```

Not all decision tasks are wait-free solvable in read-write memory. Herlihy et al. [12] introduced a formalism for characterizing wait-free computability of decision tasks in read-write memory based on properties of a high-dimensional geometric structure associated with the decision task. The asynchronous computability theorem, presented by Herlihy et al. [12], gives the necessary and sufficient conditions for the existence of a wait-free protocol that solves a particular decision task. This wait-free computability characterization led to impossibility results for open problems in distributed computing such as the renaming problem [2], the set agreement problem [5], and the approximate agreement problem.

2.1 Combinatorial Topology Foundation

A high-dimensional geometric structure associated with a decision task is generalized using notions from combinatorial topology [14].

Definition 2.1. A *simplicial complex* is a set composed of vertices, edges, triangles, tetrahedrons, and their higher-dimensional extensions [9].

Definition 2.2. Given a set S and a family $\mathcal{K}$ of finite subsets of S , $\mathcal{K}$ is an *abstract simplicial complex* on S if [9]:

- (1) If $X \in \mathcal{K}$, and $Y \subseteq X$, then $Y \in \mathcal{K}$; and
- (2) $v \in \mathcal{K}$ for all $v \in S$.

The term *vertex* refers to an element of S , and the term *simplex* refers to an element of $\mathcal{K}$. An abstract simplicial complex is a combinatorial view of a simplicial complex [9]. We refer to an abstract simplicial complex as a *complex* for brevity. The set of all vertices of $\mathcal{A}$ is denoted by $V(\mathcal{A})$.

Definition 2.3. A simplex τ is a *face* of simplex σ if $\tau \subseteq \sigma$ [9].

The number of vertices contained in simplex σ is denoted as $|\sigma|$.

Definition 2.4. The *dimension* of simplex $\sigma \in \mathcal{K}$ is $|\sigma| - 1$ [9].

Definition 2.5. The *dimension* of complex $\mathcal{K}$ is highest dimension of any of its simplices [14].

Definition 2.6. A *labeling* of a complex $\mathcal{A}$ is a map carrying each vertex of $\mathcal{A}$ to an element of some domain [9].

Definition 2.7. A *coloring* of an n -dimensional complex $\mathcal{A}$ to a domain Π is a labeling $\mathcal{X} : V(\mathcal{A}) \rightarrow \Pi$ such that $\mathcal{X}$ is injective on the vertices of every simplex $\sigma \in \mathcal{A}$: for distinct $s_0, s_1 \in \sigma$, $\mathcal{X}(s_0) \neq \mathcal{X}(s_1)$.

Definition 2.8. A complex $\mathcal{A}$ together with a coloring $\mathcal{X}$ is called a *chromatic complex*, written $(\mathcal{A}, \mathcal{X})$ [9].

Definition 2.9. The *distinguished value* $\perp$ is a value assigned to a process if it has no input (or output) for a decision task.

Definition 2.10. An $(n+1)$ -process *input vector* $\vec{I}$ (respectively, *output vector* $\vec{O}$) is an $(n+1)$ -component vector, each component of which is either a value of input data type (respectively, output data type), or the distinguished value $\perp$. At least one component of $\vec{I}$ (respectively, $\vec{O}$) must be different from $\perp$ [14].

Definition 2.11. Vector $\vec{U}$ is a *prefix* of $\vec{V}$ if, for $0 \leq i \leq n$, either $\vec{U}[i] = \vec{V}[i]$ or $\vec{U}[i] = \perp$ [14].

Definition 2.12. A set V of vectors is *prefix-closed* if for all $\vec{V} \in V$, every prefix $\vec{U}$ of $\vec{V}$ is in V [14].

Definition 2.13. Let I be a prefix-closed set of input vectors. Let $\vec{I} \in I$ be an input vector. The *input simplex* corresponding to $\vec{I}$, denoted $S(\vec{I})$, is the chromatic complex whose vertices (P_i, v_i) correspond to the entries in $\vec{I}$, for which $\vec{I}[i] = v_i \neq \perp$ [14].

Definition 2.14. Let O be a prefix-closed set of output vectors. Let $\vec{O} \in O$ be an output vector. The *output simplex* corresponding to $\vec{O}$, denoted $S(\vec{O})$, is the chromatic complex whose vertices (P_i, v_i) correspond to the entries in $\vec{O}$, for which $\vec{O}[i] = v_i \neq \perp$ [14].

Definition 2.15. Let I be a prefix-closed set of input vectors. Let O be a prefix-closed set of output vectors. The *input complex* corresponding to I , denoted $\mathcal{I}$, is the collection of input simplices $S(\vec{I})$ corresponding to the input vectors of I . The *output complex* corresponding to O , denoted $\mathcal{O}$, is the collection of output simplices $S(\vec{O})$ corresponding to the output vectors of O [14].

Definition 2.16. For complexes $\mathcal{A}$ and $\mathcal{B}$, a *vertex map* $\mu : V(\mathcal{A}) \rightarrow V(\mathcal{B})$ carries each vertex of $\mathcal{A}$ to a vertex of $\mathcal{B}$ [9].

Definition 2.17. For complexes $\mathcal{A}$ and $\mathcal{B}$, a vertex map μ from $\mathcal{A}$ to $\mathcal{B}$ is a *simplicial map* if for each simplex $\{s_0, \dots, s_n\} \in \mathcal{A}$, $\{\mu(s_0), \dots, \mu(s_n)\}$ is a simplex of $\mathcal{B}$ [9].

Definition 2.18. A complex $\mathcal{K}$ is *path-connected* if there is a path between every two vertices in $\mathcal{K}$. The largest subcomplexes of $\mathcal{K}$ that are path-connected are the *path-connected components* [9].

Definition 2.19. A *carrier map* Φ is a mapping from complex $\mathcal{A}$ to complex $\mathcal{B}$ that takes each simplex $\sigma \in \mathcal{A}$ to a subcomplex $\Phi(\sigma)$ of $\mathcal{B}$ such that for all $\sigma, \tau \in \mathcal{A}$, $\sigma \subseteq \tau$ implies $\Phi(\sigma) \subseteq \Phi(\tau)$ [9].

An n -dimensional complex is *pure* if every simplex is a face of some n -simplex [14].

Definition 2.20. A carrier map Φ that maps $\mathcal{A}$ to $\mathcal{B}$ is *rigid* if for every simplex $\sigma \in \mathcal{A}$ of dimension d , the subcomplex $\Phi(\sigma)$ is pure of dimension d [9].

Definition 2.21. A carrier map Φ that maps $\mathcal{A}$ to $\mathcal{B}$ is *strict* if $\Phi(\sigma \cap \tau) = \Phi(\sigma) \cap \Phi(\tau)$, for all $\sigma, \tau \in \mathcal{A}$ [9].

Definition 2.22. Given two chromatic complexes $(\mathcal{A}, \mathcal{X}_{\mathcal{A}})$ and $(\mathcal{B}, \mathcal{X}_{\mathcal{B}})$, carrier map $\Phi : \mathcal{A} \rightarrow \mathcal{B}$ is *chromatic* if Φ is rigid and for all $\sigma \in \mathcal{A}$ we have $\mathcal{X}_{\mathcal{A}}(\sigma) = \mathcal{X}_{\mathcal{B}}(\Phi(\sigma))$, where $\mathcal{X}_{\mathcal{B}}(\Phi(\sigma)) := \{\mathcal{X}_{\mathcal{B}}(v) | v \in V(\Phi(\sigma))\}$ [9].

Definition 2.23. A complex $\sigma(\mathcal{K})$ is a *subdivision* of complex $\mathcal{K}$ if each simplex in $\sigma(\mathcal{K})$ is contained in $\mathcal{K}$ and each simplex of $\mathcal{K}$ is the union of finitely many simplices in $\sigma(\mathcal{K})$ [14].

Definition 2.24. Let $\mathcal{K}$ be a complex and let $\text{Div}(\mathcal{K})$ be a subdivision of $\mathcal{K}$. For τ in $\text{Div}(\mathcal{K})$, the *carrier* of τ , denoted $\text{carrier}(\tau, \mathcal{K})$ is the unique smallest simplex $\sigma \in \mathcal{K}$ such that $\tau \in \text{Div}(\sigma)$ [9].

Definition 2.25. Let $(\mathcal{A}, \mathcal{X})$ be a chromatic complex with dimension n and let $\sigma = \{s_0, \dots, s_n\}$ be an n -simplex of $\mathcal{A}$. Its *standard chromatic subdivision* $\text{Ch } \mathcal{A}$ is the complex of which the vertices have the form (i, σ_i) , where $i \in \{0, \dots, n\}$, σ_i is a nonempty face of σ , and $i \in \mathcal{X}(\sigma_i)$. A $(k+1)$ -tuple $(\sigma_0, \dots, \sigma_k)$ is a simplex of $\text{Ch } \mathcal{A}$ if and only if

- (1) The tuple can be indexed so that $\sigma_0 \subseteq \dots \subseteq \sigma_k$, and
- (2) For $0 \leq i, j \leq n$, if $i \in \mathcal{X}(\sigma_j)$, then $\sigma_i \subseteq \sigma_j$.

To make the subdivision chromatic, we define the coloring $\mathcal{X} : \text{Ch } \mathcal{A}$ to be $\mathcal{X}(i, \sigma) = i$ [9].

Definition 2.26. A *general task* is formally denoted as the triple $(\mathcal{I}, \mathcal{O}, \Delta)$, where $\mathcal{I}$ is a chromatic complex representing the input complex, $\mathcal{O}$ is a chromatic complex representing the output complex, and Δ is a chromatic carrier map from $\mathcal{I}$ to $\mathcal{O}$ [9].

Definition 2.27. A *protocol* is formally denoted as a triple $(\mathcal{I}, \mathcal{P}, \Xi)$, where $\mathcal{I}$ is a chromatic complex representing the input complex, $\mathcal{P}$ is a chromatic complex representing the final configuration, and Ξ is a chromatic strict carrier map $\mathcal{I}$ to $\mathcal{P}$ [9].

The chromatic complex $\mathcal{P}$ is referred to as the *protocol complex*. A protocol $(\mathcal{I}, \mathcal{P}, \Xi)$ solves a task $(\mathcal{I}, \mathcal{O}, \Delta)$ if there exists a chromatic simplicial map δ that maps $\mathcal{P}$ to $\mathcal{O}$ such that $\delta(\Xi(\sigma)) \subseteq \Delta(\sigma)$ for all $\sigma \in \mathcal{I}$ [9].

The asynchronous computability theorem [12] states that a decision task $(\mathcal{I}, \mathcal{O}, \Delta)$ has a wait-free read-write protocol if and only if $\mathcal{I}$ has a chromatic subdivision $\text{Div } \mathcal{I}$ and a color-preserving simplicial map $\mu : \text{Div } \mathcal{I} \rightarrow \mathcal{O}$ carried by Δ . Thus, a task is wait-free solvable if and only if the input complex can be subdivided and each component of the subdivided complex can be mapped to the output complex according to the carrier map of the decision task.

2.2 Binary Consensus in the Read-Write Memory Model

The consensus problem consists of a set of processes where each process starts with a private input value and calls the `decide()` method to determine the output value. The constraint of the `decide` method is that all processes must decide the same value, and the decision value must be at least one of the participating process's input. Binary consensus is a special case of consensus in which there are only two processes whose inputs are either 0 or 1, shown in Figure 1. The four connected 1-simplices in Figure 1a represent the four possible input configurations by the processes: P and Q with input 1 (top), P and Q with input 0 (bottom), P with input 1 and Q with input 0 (left), and P with input 0 and Q with input 1

(right). The two connected 1-simplices in Figure 1b represent the two possible output configurations: Either P and Q decide 1 (top) or P and Q decide 0 (bottom).

Figure 1: Binary Consensus

The read-write memory model only allows processes to either read or write an individual word in one atomic step. As detailed in the read-write protocol of Algorithm 1, each process writes to its designated memory location. In the binary consensus problem, if process P runs solo it will decide its own input since it only observes its local input. Similarly, process Q will decide its own input if it runs solo. If P and Q run together, then one process must change its tentative decision while preventing the other process from changing its decision. Since it is unknown which process wrote to memory first, a decision value cannot be obtained that is guaranteed to be unanimous among the processes.

The fact that binary consensus has no wait-free read-write protocol can be demonstrated through the structure of the input and output complexes. If processes P and Q have input 0, they must decide 0. It follows that the simplicial map must map input simplex $\{P(0), Q(0)\}$ to output simplex $\{P:0, Q:0\}$. If processes P and Q have input 1, they must decide 1. It follows that the simplicial map must map input simplex $\{P(1), Q(1)\}$ to output simplex $\{P:1, Q:1\}$. If processes P and Q have different inputs, they must both decide 0 or both decide 1. It follows that the simplicial map must map input simplices $\{P(0), Q(1)\}$ and $\{P(1), Q(0)\}$ to both output simplex $\{P:0, Q:0\}$ and output simplex $\{P:1, Q:1\}$, which are disconnected components. However, simplicial maps must preserve connectivity. Therefore, according to the asynchronous computability theorem for read-write memory, binary consensus is not wait-free solvable because a simplicial map does not exist that maps the input complex to the output complex.

3 RELATED WORK

Herlihy et al. [10] demonstrate that it is impossible for a protocol to solve set consensus if the protocol's complex lacks holes of a small dimension. Additionally, the authors identify the kinds of tasks that can be solved by N processes using (M, j) -consensus objects, for $M \leq N$ where M is the number of processes that can access the object and j is the size of the set of values returned by the object. Although these impossibility results encompass a more powerful computational model than read/write registers, process communication is limited to Test-And-Set variables or (M, j) -consensus objects. Test-And-Set has a consensus number of 2, which is clearly weaker than CAS or LL/SC. The (M, j) -consensus object, however, can solve consensus for N processes when M equals N and j equals 1. Nevertheless, the $(N, 1)$ -consensus object is more constraining than CAS or LL/SC because the consensus protocol chooses the

input of the first process to write to memory. The success of CAS and LL/SC is dependent on the order in which the reads and writes interleave, which is not captured by the $(N, 1)$ -consensus object.

Borowsky et al. [4] propose the iterated immediate snapshot model of computation. This model of computation is used to characterize models more complex than the read-write shared memory model. The authors prove that the iterated immediate snapshot model is computationally equivalent to the read-write shared memory model. The results presented by Borowsky and Gafni provide a computational model that is much closer to the shared memory model provided by modern architectures than the read-write shared memory model. However, the method of communication between processes limits each process to read or write once per iteration. The communication does not account for atomic primitives such as CAS or LL/SC.

Herlihy et al. [11] present results concerning the decidability of tasks in a range of different models and resilience levels. Synchronization primitives stronger than atomic read and atomic write, such as Test-And-Set, Fetch-And-Add, CAS, and LL/SC, are modeled using a parameterized family of (m, k) -set agreement objects, where m is the number of processes that share an object and k is the number of distinct values chosen by the object. As these results model atomic primitives using (m, k) -set agreement objects, the power of the model is limited to the m -process consensus object. CAS and LL/SC are more powerful than the m -process consensus object because they allow multiple writes to the same memory location given that the CAS or LL/SC operations succeed. All processes that access the m -process consensus object, however, will decide the input of the first process to write to memory and all subsequent writes to memory will fail.

4 THE CAS-EXTENDED MODEL

The model is an extension to the asynchronous shared memory read-write model. Memory updates may be made with atomic primitives including LL/SC, CAS, atomic write, and RMW operations such as Fetch-And-Add. Fetch-And-Add atomically increments a memory location by a specified value and returns the value prior to the update. Since the read and write of a memory location by Fetch-And-Add is performed in one atomic step, the write by Fetch-And-Add is modeled the same as an atomic write in the CAS-extended model. CAS accepts as parameters a memory location, an expected value, and an update value. If the data referenced by the memory location is equivalent to the expected value, then the data referenced by the memory location is changed to the update value with a return value of true; otherwise, no change is made and the return value is false. LL accepts as a parameter a memory location. LL reads the memory location and returns the data referenced by the memory location. The subsequent SC call accepts as parameters the memory location previously passed to LL and an update value. If no updates have occurred to the memory location since the invocation of LL, then the data referenced by the memory location is changed to the update value with a return value of true; otherwise, no change is made and false is returned. Atomic write, LL/SC, and CAS are formally defined as follows:

Atomic Write: Let T be a set of elements of arbitrary data type. Let M be a shared memory array whose elements are in set T .

Let $P_i \in \{0, \dots, n\}$ be the name of the participating process. Let $v_{arbitrary} \in T$ be an arbitrary value in set T . Atomic write is a pair $((x_i, v_i), \Delta_{ATOMIC})$, where $(x_i \in \{0, \dots, size(M)\}, v_i \in T)$ is the input pair to atomic write by process P_i , the output to P_i is void, and $\Delta_{ATOMIC} : (M[x_i] \mapsto v_{arbitrary}) \xrightarrow{\Delta} (M[x_i] \mapsto v_i)$ is an atomic state transition that defines the write of v_i to $M[x_i]$ in one indivisible step.

Load-Link/Store-Conditional (LL/SC): Let T be a set of elements of arbitrary data type. Let M be a shared memory array whose elements are in set T . Let $P_i \in \{0, \dots, n\}$ be the name of the participating process. Load-link is a triple $(x_i, v_i, \Delta_{ATOMIC})$, where $x_i \in \{0, \dots, size(M)\}$ is the input to LL by process P_i , $v_i \in T$ is the output of LL to process P_i , and $\Delta_{ATOMIC} : (v_i \mapsto NULL) \xrightarrow{\Delta} (v_i \mapsto M[x_i])$ is an atomic state transition that defines the atomic read of $M[x_i]$ stored in v_i . Store-conditional is a triple $((x_i, v'_i), z_i, \Delta_{ATOMIC})$, where $(x_i \in \{0, \dots, size(M)\}, v'_i \in T)$ is the input pair to SC by process P_i , $z_i \in \{true, false\}$ is the output of SC to process P_i , and Δ_{ATOMIC} is an atomic state transition that defines the atomic write of v_i to $M[x_i]$ such that if $M[x_i]$ is not updated by SC for the entire duration of time between the invocation of LL by P_i and the invocation of SC by P_i , then $\Delta_{ATOMIC} : ((M[x_i] \mapsto v_i) \xrightarrow{\Delta} (M[x_i] \mapsto v'_i)) \wedge ((z_i \mapsto false) \xrightarrow{\Delta} (z_i \mapsto true))$. If $M[x_i]$ is updated by SC at any point of time between the invocation of LL by P_i and the invocation of SC by P_i , then no state transition occurs.

Compare-And-Swap (CAS): Let T be a set of elements of arbitrary data type. Let M be a shared memory array whose elements are in set T . Let $P_i \in \{0, \dots, n\}$ be the name of the participating process. Compare-And-Swap is a triple $((x_i, v_i, v'_i), z_i, \Delta_{ATOMIC})$, where $(x_i \in \{0, \dots, size(M)\}, v_i \in T, v'_i \in T)$ is the input triple to CAS by process P_i , $z_i \in \{true, false\}$ is the output of CAS to process P_i , and Δ_{ATOMIC} is an atomic state transition that defines the atomic write of $M[x_i]$ such that if $M[x_i] = v_i$, then $\Delta_{ATOMIC} : ((M[x_i] \mapsto v_i) \xrightarrow{\Delta} (M[x_i] \mapsto v'_i)) \wedge ((z_i \mapsto false) \xrightarrow{\Delta} (z_i \mapsto true))$. If $M[x_i] \neq v_i$, then no state transition occurs.

Algorithm 2 CAS-Extended Protocol

```

1: // A process may either read, write, or perform CAS or LL/SC on an individual word
   in one atomic step
2: //  $i \in \{0, \dots, n\}$  is the set of process names
3: shared mem: array  $[0..n]$  of Value
4: procedure CASEXTENDED( $v_i$ : Value, loc: int) : Value
5:   exp: Value, result: bool
6:   exp := mem[loc]                                ▶ LL(loc)
7:   for  $j := 0$  to  $n$  do
8:     view[j] := mem[j]
9:   end for
10:  result := CAS(loc, exp,  $v_i$ )                      ▶ SC(loc,  $v_i$ )
11:  if result then
12:    view[loc] :=  $v_i$ 
13:    return  $\delta_T$  (view)
14:  else
15:    return  $\delta_F$  (view)
16:  end if
17: end procedure

```

The read-write, snapshot [1], layered snapshot [4], immediate snapshot [3], and layered immediate snapshot [4] are equivalent models of computation for asynchronous shared memory [9]. We extend the read-write model of computation for the CAS-extended

protocol because the read-write model is a close representation of the asynchronous shared memory model provided by modern architectures. The protocol for updates made by CAS or LL/SC is detailed in Algorithm 2. If a CAS or SC call returns true, it modifies its view to reflect the update at line 12 and applies the decision mapping function δ_T at line 13 for a successful update. If a CAS or SC call returns false, it applies the decision mapping function δ_F at line 15 for an unsuccessful update. A process may either read, write, or perform CAS or LL/SC on an individual word in one atomic step. The order in which the CAS and LL/SC calls by the processes interleave affect the success or failure of a process to update the memory location. We eliminate the N layers of communication incorporated in the read-write model since the return value of the CAS or SC provides sufficient information for a process to apply its decision mapping function.

4.1 CAS-Extended Protocol Complex

The topological structure of the protocol complex in the CAS-extended model must account for each memory location operated on through a CAS, LL/SC, or atomic write by each participating process because our goal is to verify that a concurrent object is wait-free, rather than an individual decision task. An *execution path* is an interleaving of atomic operations by the processes on a memory location. For each memory location that serves as a shared variable among processes, all possible execution paths by the processes to update the respective memory location are represented by the protocol complex. The order in which the CAS, LL/SC, and atomic writes interleave affects the outcome of the CAS and LL/SC calls. The protocol complex must therefore account for the various interleavings of the atomic operations.

In read-write memory, the protocol complex encompasses a chromatic subdivided image of the input complex [14]. This is not the case in the CAS-extended model. The input complex alone does not sufficiently represent the possible outcomes of the CAS and LL/SC calls. For each input to CAS or LL/SC, the outcome is either a successful update in which true is returned, or an unsuccessful update in which false is returned. Therefore, the input complex must be augmented with additional vertices representing the case of an unsuccessful update for an input value to CAS or LL/SC.

Definition 4.1. Let I be a prefix-closed set of input vectors to all memory locations, where each input vector is the input to an individual memory location. Let $\vec{I} \in I$ be an input vector and let $S(\vec{I})$ be the input simplex. Let $\vec{I}_{CAS} \in I$ be an input vector comprising the elements of $\vec{I}$ such that the inputs are passed to CAS or LL/SC, where $S(\vec{I}_{CAS})$ is the input simplex corresponding to $\vec{I}_{CAS}$ and $S(\vec{I}_{CAS}) \subseteq S(\vec{I})$. Let $\vec{I}'$ be the concatenation of input vector $\vec{I}$ and $\vec{I}_{CAS}$, where $\vec{I}' = \vec{I} \cdot \vec{I}_{CAS}$. The *write-attempt simplex* corresponding to $\vec{I}'$, denoted $S(\vec{I}')$, is the chromatic complex whose vertices $(P_i, (v_i, ret_i))$ correspond to the participating entries in $\vec{I}'$, for which $\vec{I}'[i] = (v_i, ret_i)$, where ret_i is a boolean type that indicates if the input v_i represents a successful update or an unsuccessful update.

Definition 4.2. Let I be a prefix-closed set of input vectors to all memory locations, where each input vector is the input to an individual memory location. Let $\vec{I} \in I$ be an input vector. Let $\vec{I}_{CAS} \in$

I be an input vector comprising the elements of $\vec{I}$ such that the inputs are passed to CAS or LL/SC. Let I' be a prefix-closed set of input vectors such that for each $\vec{I}' \in I'$, $\vec{I}' = \vec{I} \cdot \vec{I}_{CAS}$, and each input vector is the input to an individual memory location. The *write-attempt complex* corresponding to I' , denoted $\mathcal{I}'$, is the collection of write-attempt simplices $\mathcal{S}(\vec{I}')$ corresponding to the input vectors of I' . Therefore, $\mathcal{I}'$ comprises input simplices modeling the attempted inputs to all memory locations.

Given N processes, each $(N - 1)$ -dimensional face of an input simplex represents an input configuration by the N processes. Each process may have multiple inputs to a memory location. Since the N processes are asynchronous, each input by a process has a chance of interleaving with all possible inputs from the other N processes. Therefore, each possible combination of inputs by the N processes must be modeled as a face in the input simplex. A face representing an input configuration only models a single input by each process to the memory location. The particular input order relative to the other inputs by a single process does not affect the possible input configurations by the N processes since the input configuration only models a single input by each process.

Herlihy et al. [13] prove that the full information protocol complex of input complex $\mathcal{I}^n$ in the immediate snapshot model has the structure of the iterative standard chromatic subdivision applied to $\mathcal{I}^n$, where the number of iterations is one. The difference between the immediate snapshot model and the read-write model, shown in Algorithm 1, is that after an atomic write, the immediate snapshot model reads the entire memory instantaneously rather than performing an atomic read one memory location at a time. A reduction of the immediate snapshot model to the read-write model is non-trivial. Borowsky et al. [3] present a reduction of the immediate snapshot model to the snapshot model. Gafni et al. [7] present a reduction of the snapshot model to the layered snapshot model. Afek et al. [1] present a reduction of the layered snapshot model to the read-write model. Following these reductions, the immediate snapshot model and read-write model are equivalent models of computation. Therefore, the existence of a protocol in the immediate snapshot model implies the existence of a protocol in the read-write model.

Some executions that are possible in the read-write memory model are not necessarily possible in the CAS-extended model. Process i in the read-write memory model writes its input to $mem[i]$ and updates its view by reading the memory. It is possible for a process to observe multiple writes in the same view. In the CAS-extended model, processes read prior to writing when calling CAS or LL/SC. Additionally, the write will fail if some other process wrote to memory between the read and write. Therefore, given input simplex $\mathcal{I}^n$, the simplices $\tau \in \text{Ch } \mathcal{I}^n$ that correspond to a concurrent execution in which multiple processes write by CAS or LL/SC in the same view are not possible in the CAS-extended model.

LEMMA 4.3. *Let $(\mathcal{I}', \mathcal{P}', \Xi)$ be the protocol for $\mathcal{I}'$ in the CAS-extended model, then $\mathcal{P}' \subseteq \text{Ch } \mathcal{I}'$.*

PROOF. We show by a direct proof that $\mathcal{P}' \subseteq \text{Ch } \mathcal{I}'$. The proof that the full information protocol complex of input complex $\mathcal{I}^n$ in the immediate snapshot model has the structure of the iterative

standard chromatic subdivision (with the number of iterations set to one) applied to $\mathcal{I}^n$ is provided in [13]. $\text{Ch } \mathcal{I}'$ therefore contains all immediate snapshot executions for input $\mathcal{I}'$. The CAS-extended model is different from the immediate snapshot model in that a process invoking CAS or LL/SC reads the memory and updates the memory only if no other process has written to memory between the read and the write. Therefore, a process's view in the CAS-extended model will reflect at most one write occurring at the instant the view was read. Since the immediate snapshot model allows process i to write to $mem[i]$ at any instant within the N rounds, $\text{Ch } \mathcal{I}'$ contains simplices corresponding to a concurrent execution in which each process's view reflects only one write occurring at the instant the view was read. It follows that $\mathcal{P}' \subseteq \text{Ch } \mathcal{I}'$. $\square$

We now present the main result of the paper.

LEMMA 4.4. *Let $(\mathcal{I}', \mathcal{P}', \Xi)$ be the protocol for $\mathcal{I}'$ in the CAS-extended model. For any simplex $\tau \in \text{Ch } \mathcal{I}'$, let $\text{views}(\tau)$ be the set of unique views that reflect the instantaneous snapshot after an atomic write in the immediate snapshot model by each of the processes associated with the vertices of τ . Let $\text{observed}(\tau)$ be the set of ids of processes whose inputs appear in $\text{views}(\tau)$. Let $\text{size}(S)$ be the number of unique elements in set S . Let $\phi(\mathcal{I}')$ be the complex containing all simplices $\tau \in \text{Ch } \mathcal{I}'$ such that $\text{size}(\text{views}(\tau)) < \text{size}(\text{observed}(\tau))$. Let $\bar{N}$ be the set of vertices in $\mathcal{I}'$ representing the input for a failed CAS or LL/SC. $\mathcal{P}' = \text{Ch } \mathcal{I}' - \phi(\mathcal{I}') - \bar{N}$.*

PROOF. We show by a proof by cases that $\mathcal{P}' = \text{Ch } \mathcal{I}' - \phi(\mathcal{I}') - \bar{N}$. We demonstrated that $\mathcal{P}' \subseteq \text{Ch } \mathcal{I}'$ in the proof for Lemma 4.3. There are two execution cases of $\text{Ch } \mathcal{I}'$ in the immediate snapshot model that are not possible in the CAS-extended model. The first impossible case consists of executions in which multiple processes have written their input to memory in the same view. The second impossible case consists of executions in which a vertex representing a process with a failed CAS or LL/SC reflects a view in which that process was the first to write to a memory location.

Case 1: The size of $\text{views}(\tau)$ will be less than the size of $\text{observed}(\tau)$ when multiple processes write their input to memory in the same view. Since $\phi(\mathcal{I}')$ contains all simplices $\tau \in \text{Ch } \mathcal{I}'$ such that $\text{size}(\text{views}(\tau)) < \text{size}(\text{observed}(\tau))$, this implies that $\phi(\mathcal{I}')$ contains simplices in which multiple processes have the same view. The complex $\phi(\mathcal{I}')$ is non-empty since multiple processes may write to memory in the same round in the immediate snapshot model, resulting in processes having the same view. Updates made by CAS or LL/SC in the CAS-extended model require processes to read before writing, and only one process out of the processes that actively read the memory will successfully write to memory. Each update made by an atomic write is performed in a serialized fashion such that only one write may be performed to a memory location at any atomic step. Since $\phi(\mathcal{I}')$ is the complex containing all simplices in which multiple processes writing by CAS, LL/SC, or atomic write have the same view, $\text{Ch } \mathcal{I}' - \phi(\mathcal{I}')$ is the complex in which every process's view reflects only one write performed at an atomic step.

Case 2: $\mathcal{I}'$ contains vertices $\bar{N}$ representing the input for a failed CAS or LL/SC. Since it is impossible for a process with a failed CAS or LL/SC to reflect a view in which it was the first to write to

memory, all simplices connected to vertex $v \in \bar{N}$ must be removed from $\text{Ch } I'$. $\text{Ch } I - \bar{N}$ removes all 1-simplices incident to every vertex $v \in \bar{N}$.

Since $\text{Ch } I' - \phi(I') - \bar{N}$ represents the complex without the impossible execution cases in the CAS-extended model, $\mathcal{P}' = \text{Ch } I' - \phi(I') - \bar{N}$. $\square$

In the CAS-extended model, the protocol complex $\mathcal{P}$ encompasses a chromatic subdivided image of the write-attempt complex I' . Since an execution in which an input to CAS or LL/SC results in a successful update is mutually exclusive with an execution in which this input to CAS or LL/SC results in an unsuccessful update, the vertex representing the successful update and the vertex representing the unsuccessful update are not path-connected in $\mathcal{P}$ and therefore reside in different path-connected components of $\mathcal{P}$.

4.1.1 Two-Process Protocol Complex Example. We demonstrate an example for a two-process one-input protocol complex in which both processes attempt to write their input to memory using LL/SC. This example is also applicable to processes that attempt to write their input to memory using CAS. The read by LL corresponds to the read performed to obtain the expected value parameter passed to CAS. Figure 2 shows the input complex for the two-process system where process P has input p and process Q has input q . Figure 2a shows the original input complex I . We apply Lemma 4.4 to construct the write-attempt complex. Figure 2b shows the write-attempt complex I' representing the case where the input is successfully written to memory and the case where the input is unsuccessfully written to memory.

Figure 2: Two-Process One-Input Input Complex for Updates with LL/SC

Figure 3 shows the construction of the protocol complex from the write-attempt complex I' by applying Lemma 4.4. We first interpret the standard chromatic subdivision $\text{Ch } I'$ in read-write memory and then explain how to derive the protocol complex from $\text{Ch } I'$ in the CAS-extended model. In the read-write memory model, the standard chromatic subdivision captures all possible ways in which the processes communicate through the shared memory. Figure 3a shows the standard chromatic subdivision $\text{Ch } I'$ modeling the execution paths in read-write memory. Each of the four corner vertices represents an execution path in which the process associated with the vertex reads the memory prior to the other process writing to memory. Every pair of corner vertices is subdivided into three 1-simplex segments. Each 1-simplex represents an execution path between two processes. Recall that if $\bar{I}[i] = \perp$, then P_i does not participate in the execution. For example, the three 1-simplex segments at the top of Figure 3a can be interpreted in the following

way. The segment $(p \perp)$ to (pq) represents the execution in which P writes and reads before Q writes. This interpretation is obtained because process P 's view includes only its own write while process Q 's view includes the write by both P and Q . The segment (pq) to $(q \perp)$ represents the execution in which P and Q write and each reads after the other writes. This is due to the inclusion of the write by process P and the write by process Q in both process P 's view and process Q 's view. The segment (pq) to $(\perp q)$ represents the execution in which Q writes and reads before P writes. This interpretation is obtained because process Q 's view includes only its own write while process P 's view includes the write by both P and Q .

Figure 3: Two-Process One-Input Protocol Complex for Updates with LL/SC

All possible execution paths of P and Q calling LL/SC on a memory location are contained within the subdivision $\text{Ch } I'$ of Figure 3a. There are four possible ways in which the LL/SC calls by two processes with one input interleave. These interleavings are (1) P writes before Q reads ($LLpSCpLLqSCq$), (2) Q writes before P reads ($LLqSCqLLpSCp$), (3) P writes after Q reads and before Q writes ($LLpLLqSCpSCq$), and (4) Q writes after P reads and before P writes ($LLqLLpSCqSCp$). Reads are commutative, so the order in which consecutive reads occur does not matter. Interleavings (1) and (2) will result in a successful update by both P and Q because no write occurs between the read and write performed by each process. Interleaving (3) will result in a successful update by P and an unsuccessful update by Q because P writes between the instant that Q reads and the instant the Q writes. Interleaving (4) will result in a successful update by Q and an unsuccessful update by P because Q writes between the instant that P reads and the instant the P writes. The mapping of these four interleavings to $\text{Ch } I'$ is shown in the protocol complex of Figure 3b.

The subdivision $\text{Ch } I'$ of Figure 3a also contains some execution paths that are not possible. Since each of the four corner vertices represents an execution path in which the associated process reads the memory prior to the other process writing to memory, the write by the process represented by the corner vertex must succeed. Therefore, the execution paths represented by the 1-simplices connected to vertices $Q(q) : F$ and $P(p) : F$ are not possible. The interior 1-simplices (pq) to (pq) are also not possible, since these

segments represent the execution in which P and Q each read after the other writes. CAS and LL/SC require the read to precede the write, in which case the write is only performed under the circumstance that no other process has written between the instant that the read and the write occur. Therefore, only one write can occur at an atomic step. Eliminating all 1-simplices representing impossible execution paths yields the final protocol complex of Figure 3b.

4.2 Binary Consensus in the CAS-Extended Model

In the wait-free protocol for binary consensus using CAS, process i proposes its input in an array at position i . Memory location r is initialized to -1. Process i calls CAS on memory location r , passing -1 as the expected value parameter and its proposed input as the update value parameter. If the CAS return value is true, it chooses its own input as the decision value; otherwise, it chooses the other process's proposed value [15]. The protocol complex and output complex for binary consensus in the CAS-extended model is shown in Figure 4.

Figure 4: Binary Consensus with CAS

Each of the four disjoint complexes, shown in Figure 4a, represents the possible execution paths to arrive at the final state of memory location r , shown in Figure 4b. For example, the top left complex shows the execution paths that will result in process P writing 1 to memory location r . The center vertex of this complex represents the execution in which process P runs solo. The edge connecting vertex $P(1)$ ($CAS_P:T$) to vertex $Q(1)$ ($CAS_Q:F$) represents the execution in which process P writes 1 and process Q unsuccessfully attempts to write 1. The edge connecting vertex $P(1)$ ($CAS_P:T$) to vertex $Q(0)$ ($CAS_Q:F$) represents the execution in which process P writes 1 and process Q unsuccessfully attempts to write 0. The CAS return value is false because the value of memory location r is no longer equivalent to the expected value of -1.

In order for a protocol complex to be wait-free, there must exist a simplicial map from $\mathcal{P}$ to $\mathcal{O}$. A simplicial map must preserve connectivity. The simplicial map for the binary consensus protocol maps the top components of the protocol complex to the top component of the output complex and the bottom components of the protocol complex to the bottom component of the output complex.

There is no particular order associated with the components of the protocol complex. However, each component does encompass an ordering among the processes that participate in the execution path.

5 WAIT-FREE QUEUE ALGORITHM

The wait-free queue implementation [16] that this work verifies supports multiple concurrent enqueue operations and only one concurrent dequeue operation. The strategy for this wait-free queue implementation is for each of n processes to maintain a local queue of elements. The front element of each local queue is inserted into a global queue maintained as a binary tree of size n . The data structure as a whole is an array of size $2n$, where elements $[0, n-1]$ are reserved for the binary tree and elements $[n, 2n-1]$ are reserved for pointers to the local queues. This can be visualized as one large binary tree where the bottom tier of the tree is the access point into each of n local queues, as shown in Figure 5.

Figure 5: Queue Structure

Figure 6: Overall System Input/Output Structure

6 VERIFICATION OF THE WAIT-FREE QUEUE

A wait-free concurrent object can be formally verified through combinatorial topology by constructing a protocol complex $\mathcal{P}$ by Lemma 4.4 from a specified input set $\mathcal{I}$ and output set $\mathcal{O}$ and defining a simplicial map from $\mathcal{P}$ to $\mathcal{O}$. The protocol complex must account for all final states of the memory locations operated on by multiple processes, where each simplex represents an execution path to arrive at the final state.

For the purposes of demonstrating the application of combinatorial topology to the wait-free queue, we will consider a three-process system where two processes (p and q) perform an enqueue operation and one process (r) performs a dequeue operation, shown in Figure 6. Since the number of processes greatly increases the possible interleavings of the atomic operations, performing the proof for an arbitrary number of processes is a non-trivial task.

The order in which the operations are executed by the threads is uncertain, so the output complex must represent all possible outcomes based on the given input complex. Figure 6b shows three connected 2-simplex complexes. In all three complexes, the output for p (white) and q (black) is the empty set because their return type is void. However, the output for r (gray) may be either A (if p linearizes first), B (if q linearizes first), or the empty set (if r linearizes first). The correctness of this input/output pair is determined according to the linearizability correctness condition, which is the property where the real-time behavior of method calls is equivalent to a legal sequential history, and each method call takes effect at some point between its invocation and response [15].

Figure 7: Propagate at Node 2

Figure 8: Propagate/Dequeue at Node 1

Once the processes calling `enqueue()` have enqueued their element into their local queue, they call `propagate()`. The `propagate()` function sifts an element up to its proper place in the binary tree based on its time stamp. The process calling `dequeue()` will only call `propagate()` if it has an element to dequeue. This implies that if there exists an element to dequeue, then `propagate()` must have linearized for at least one of the processes. Therefore, there will be a maximum of two processes attempting an LL/SC operation on a node at any given time.

The input, output, write-attempt complex I' , and protocol complex for operations to Node 2 of the binary tree in Figure 5 are shown in Figure 7. During the `propagate()` call, a process will choose the child with the minimum time stamp to atomically write

to the parent node. If two processes are simultaneously attempting an LL/SC operation, they will have read the same child nodes and will both choose the same node to atomically write to the parent node. The disjoint input and output complexes represent how both processes will either both choose element A or both choose element B based on the minimum time stamp read. No matter which process succeeds, the correct node will be atomically written to the parent node. The protocol complex shows all possible execution paths of the LL/SC calls by the processes.

The input, output, write-attempt complex I' , and protocol complex for operations to Node 1 of the binary tree in Figure 5 are shown in Figure 8. The write-attempt complex shows how processes p and q may attempt a simultaneous LL/SC on Node 1 while

process r may attempt a read on Node 1. If process r is attempting a write on Node 1, then it must have dequeued a non-null value, implying that one of the other processes must have already linearized. This relationship is demonstrated by the write by process r being connected to either process p or process q , but not both.

The top component of the protocol complex shows the execution paths in which the read by process r , and the writes by process p and process q interleave. The connection between the three processes, represented by a triangle, indicates that all three processes participated in the execution path. The mapping of execution paths to 2-simplices in the protocol complex of Figure 8 is shown in Table 1. The bottom component of the protocol complex shows the execution paths in which the write by process r occurs first. If process r is performing a write, then process p or process q must have already linearized, resulting in a maximum of two processes participating in the execution path.

Table 1: Mapping of Execution Path to Protocol Complex of Figure 8

2-Simplex of Figure 8	Execution Path
Simplex 1	$READ_r LL_p SC_p LL_q SC_q$
Simplex 2	$READ_r LL_q SC_q LL_p SC_p$
Simplex 3	$LL_p SC_p READ_r LL_q SC_q$
Simplex 4	$LL_p SC_p LL_q SC_q READ_r$
Simplex 5	$LL_q SC_q READ_r LL_p SC_p$
Simplex 6	$LL_q SC_q LL_p SC_p READ_r$
Simplex 7	$READ_r LL_p LL_q SC_p SC_q$
Simplex 8	$LL_p LL_q SC_p READ_r SC_q$
Simplex 9	$LL_p LL_q SC_p SC_q READ_r$
Simplex 10	$READ_r LL_q LL_p SC_q SC_p$
Simplex 11	$LL_q LL_p SC_q READ_r SC_p$
Simplex 12	$LL_q LL_p SC_q SC_p READ_r$

In order to show that protocol $(\mathcal{I}, \mathcal{P}, \Xi)$ wait-free solves a task $(\mathcal{I}, \mathcal{O}, \Delta)$, we must define a chromatic simplicial map δ that maps $\mathcal{P}$ to $\mathcal{O}$ such that $\delta(\Xi(\sigma)) \subseteq \Delta(\sigma)$ for all $\sigma \in \mathcal{I}$. The simplicial mapping is shown in Table 2. Some of the complexes from $\mathcal{P}$ map to two different complexes from $\mathcal{O}$, where the output of process r may be A or B. This occurs because any process will attempt to atomically write to Node 1 the element with the minimum time stamp. This atomic write operation is independent of which process initially enqueued the element in its local queue.

7 CONCLUSION

We have presented an extended theory on combinatorial topology which includes the CAS, LL/SC, and atomic RMW operations. CAS and LL/SC provide an additional communication channel between processes, making the CAS-extended model a stronger model of computation than read-write memory. We demonstrated the application of the CAS-extended model by formally verifying a wait-free queue implementation. The wait-free property of the queue was verified by constructing the protocol complexes modeling the algorithm and defining a simplicial mapping between the protocol complex and output complex. For future work, we plan to incorporate the CAS-extended model into model checking tools to enable a practical and efficient methodology for verifying wait-free progress. The automation provided through model checkers will make the application of the CAS-extended model theory feasible for complex data structures subject to operations by any number of processes.

Table 2: Simplicial Mapping from $\mathcal{P}$ to $\mathcal{O}$

Simplex σ	Simplicial Mapping $\delta(\sigma)$
$READ_r$	$\emptyset_r$
$READ_r LL_p SC_p$	$\emptyset_r \emptyset_p$
$READ_r LL_q SC_q$	$\emptyset_r \emptyset_q$
$READ_r LL_p SC_p LL_q SC_q$	$\emptyset_r \emptyset_p \emptyset_q$
$LL_p SC_p$	$\emptyset_p$
$LL_p LL_q SC_p SC_q$	$\emptyset_p \emptyset_q$
$LL_p LL_q SC_p SC_q READ_r$	$\emptyset_p \emptyset_q A_r, \emptyset_p \emptyset_q B_r$
$LL_p SC_p READ_r$	$\emptyset_p A_r, \emptyset_p B_r$
$LL_p SC_p LL_q SC_q$	$\emptyset_p \emptyset_q$
$LL_p SC_p LL_q SC_q READ_r$	$\emptyset_p \emptyset_q A_r, \emptyset_p \emptyset_q B_r$
$LL_p SC_p LL_r SC_r$	$\emptyset_p \emptyset_q B_r$
$LL_p LL_r SC_p SC_r$	$\emptyset_p \emptyset_q B_r$
$LL_q SC_q$	$\emptyset_q$
$LL_q SC_q READ_r$	$\emptyset_q A_r, \emptyset_q B_r$
$LL_q LL_p SC_q SC_p$	$\emptyset_q \emptyset_p$
$LL_q LL_p SC_q SC_p READ_r$	$\emptyset_q \emptyset_p A_r, \emptyset_q \emptyset_p B_r$
$LL_q SC_q LL_p SC_p$	$\emptyset_q \emptyset_p$
$LL_q SC_q LL_p SC_p READ_r$	$\emptyset_q \emptyset_p A_r, \emptyset_q \emptyset_p B_r$
$LL_q SC_q LL_r SC_r$	$\emptyset_q \emptyset_q A_r$
$LL_q SC_q LL_r SC_r$	$\emptyset_q \emptyset_q A_r$
$LL_r SC_r$	$\emptyset_r$
$LL_r LL_p SC_r SC_p$	$\emptyset_p \emptyset_q B_r$
$LL_r LL_q SC_r SC_q$	$\emptyset_p \emptyset_q A_r$
$LL_r SC_r LL_q SC_q$	$\emptyset_p \emptyset_q A_r$
$LL_r SC_r LL_p SC_p$	$\emptyset_p \emptyset_q B_r$

REFERENCES

- [1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic snapshots of shared memory. *Journal of the ACM (JACM)* 40, 4 (1993), 873–890.
- [2] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. 1990. Renaming in an asynchronous environment. *Journal of the ACM (JACM)* 37, 3 (1990), 524–548.
- [3] Elizabeth Borowsky and Eli Gafni. 1993. Immediate atomic snapshots and fast renaming. In *Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*. ACM, 41–51.
- [4] Elizabeth Borowsky and Eli Gafni. 1997. A simple algorithmically reasoned characterization of wait-free computation. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*. ACM, 189–198.
- [5] Soma Chaudhuri. 1990. Agreement is harder than consensus: Set consensus problems in totally asynchronous systems. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*. ACM, 311–324.
- [6] Soma Chaudhuri, Maurice Herlihy, and Mark R Tuttle. 1999. Wait-free implementations in message-passing systems. *Theoretical Computer Science* 220, 1 (1999), 211–245.
- [7] Eli Gafni and Sergio Rajsbaum. 2010. Distributed programming with tasks. In *International Conference On Principles Of Distributed Systems*. Springer, 205–218.
- [8] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 1 (1991), 124–149.
- [9] Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. 2013. *Distributed computing through combinatorial topology*. Newnes.
- [10] Maurice Herlihy and Sergio Rajsbaum. 1994. Set consensus using arbitrary objects (preliminary version). In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*. ACM, 324–333.
- [11] Maurice Herlihy and Sergio Rajsbaum. 1997. The decidability of distributed decision tasks. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, 589–598.
- [12] Maurice Herlihy and Nir Shavit. 1993. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. ACM, 111–120.
- [13] Maurice Herlihy and Nir Shavit. 1994. A simple constructive computability theorem for wait-free computation. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*. ACM, 243–252.
- [14] Maurice Herlihy and Nir Shavit. 1999. The topological structure of asynchronous computability. *Journal of the ACM (JACM)* 46, 6 (1999), 858–923.
- [15] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier.
- [16] Prasad Jayanti and Srđjan Petrovic. 2005. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science*. Springer, 408–419.