Read-Write Memory and k-Set Consensus as an Affine Task

Eli Gafni[1], Yuan He[2], Petr Kuznetsov*[3], and Thibault Rieutord*[4]

1 **UCLA, Los Angeles, CA, USA**
`eli@ucla.edu`
2 **UCLA, Los Angeles, CA, USA**
`yuan.he@ucla.edu`
3 **Télécom ParisTech, Paris, France**
`petr.kuznetsov@telecom-paristech.fr`
4 **Télécom ParisTech, Paris, France**
`thibault.rieutord@telecom-paristech.fr`

Abstract

The wait-free read-write memory model has been characterized as an iterated *Immediate Snapshot* (IS) task. The IS task is *affine* – it can be defined as a (sub)set of simplices of the standard chromatic subdivision. In this paper, we highlight the phenomenon of a "natural" model that can be captured by an iterated affine task and, thus, by a subset of runs of the iterated immediate snapshot model. We show that the read-write memory model in which, additionally, k-set-consensus objects can be used is "natural" by presenting the corresponding simple affine task captured by a subset of 2-round IS runs. As an "unnatural" example, the model using the abstraction of *Weak Symmetry Breaking* (WSB) cannot be captured by a set of IS runs and, thus, cannot be represented as an affine task. Our results imply the first combinatorial characterization of models equipped with abstractions other than read-write memory that applies to generic tasks.

1998 ACM Subject Classification C.2.4 Distributed Systems, F.1.1 Models of Computation

Keywords and phrases iterated affine tasks, k-set consensus, k-concurrency, simplicial complexes, immediate snapshot

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.6

1 Introduction

A principal challenge in distributed computing is to devise protocols that operate correctly in the presence of failures, given that system components (processes) are asynchronous.

The most extensively studied *wait-free* model of computation [21] makes no assumptions about the number of failures that can occur. In particular, in a *wait-free* solution of a distributed *task*, a process participating in the computation should be able to produce an output regardless of the behavior of other processes.

Topology of wait-freedom. Wait-free task solvability in the read-write shared-memory model has been characterized in an elegant way through the existence of a specific continuous map from geometrical structures describing inputs and outputs of the task [22, 24].

* This work has been partially supported by the Franco-German ANR project DISCMAT devoted to connections between mathematics and distributed computing.

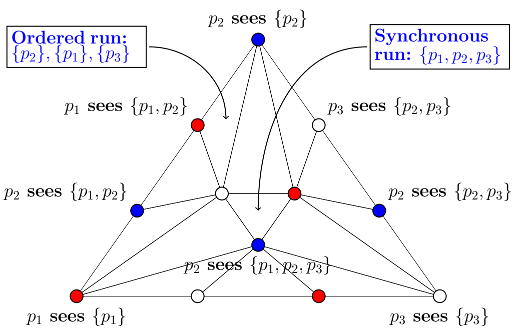

Ordered run: $\{p_2\}, \{p_1\}, \{p_3\}$
p_2 sees $\{p_2\}$
Synchronous run: $\{p_1, p_2, p_3\}$
p_1 sees $\{p_1, p_2\}$
p_3 sees $\{p_2, p_3\}$
p_2 sees $\{p_1, p_2\}$
p_2 sees $\{p_2, p_3\}$
p_2 sees $\{p_1, p_2, p_3\}$
p_1 sees $\{p_1\}$
p_3 sees $\{p_3\}$

Figure 1 Chr **s** in the 2-dimensional case.

A task T is *wait-free solvable* (using reads and writes) if and only if there exists a simplicial, chromatic map from a subdivision of the *input* simplicial complex to the *output* simplicial complex, satisfying the specification of T. In particular, using the iterated *standard chromatic subdivision* [22, 26] (one such iteration of the *standard* 2*-simplex* **s**, denoted by Chr **s**, is depicted in Figure 1), we obtain a combinatorial representation of the wait-free model. *Iterations* of this subdivision capture precisely rounds of the iterated immediate snapshot (IIS) model [5, 24].

This characterization can be interpreted as follows: the wait-free read-write model can be characterized, regarding task solvability, by an *iterated* (one-shot) Immediate Snapshot task, which, in turn, captured by the *chromatic simplex agreement* task [5, 24] on Chr **s**.

Beyond wait-freedom: k-concurrency and k-set consensus. Unfortunately, very few tasks are solvable in the wait-free manner using read and writes [3, 24, 31], so a lot of work has been invested in characterizing task solvability in various *restrictions* of the *wait-free* model.

A straightforward way to define such a restriction is to bound the *concurrency level* of runs [14]. In the k*-concurrency* model, at most k processes can be concurrently *active* (during the interval between the invocation and response of the task). The k-concurrency model is known to be equivalent to the k*-set-consensus* model in which processes, in addition to the read-write shared memory, can access k-set-consensus objects [13].

Iterated tasks for k-set-consensus. In this paper, we show that the k-set-consensus model can be captured by an iterated *affine* task [17]. Informally, an affine task consists in solving chromatic simplex agreement [5, 24] on a *subcomplex* of some iteration of the standard chromatic subdivision. We show that the affine task $\mathcal{R}_k$ capturing the k-set-consensus model consists of all simplices of the second chromatic subdivision, in which at most k processes *contend* with each other (cf. examples of $\mathcal{R}_1$ and $\mathcal{R}_2$ for 3 processes in Figure 3).

We show that $\mathcal{R}_k^*$, the set of IIS runs corresponding to iterations of this subcomplex $\mathcal{R}_k$, solves precisely the same set of tasks as the k-set-consensus model does. Note that our definition of the IIS model does not assume process failures: every process takes infinitely many steps in every run, but, because of the use of iterated memory, a “slow” process may not be seen by “faster” ones from some point [29, 10, 7]. Thus, solving a task in $\mathcal{R}_k^*$ requires *every* process to output.

Techniques and results. Our result is established through the existence of three *simulations*.

The first one simulates, in the k-set-consensus model, a k-concurrent execution of any read-write algorithm in the k-set-consensus model. We derive that the k-set-consensus model solves every task that is solvable k-concurrently.

The second algorithm *solves* $\mathcal{R}_k$ in the k-concurrency model, i.e., solves chromatic simplex agreement on the $\mathcal{R}_k$ complex. By iterating this solution, we can *simulate* the $\mathcal{R}_k^*$ model and, thus, solve any task solvable in $\mathcal{R}_k^*$.

The third algorithm simulates runs of an algorithm using read-write memory and k-set-consensus objects in $\mathcal{R}_k^*$. Compared to the simulations in [23, 18, 16, 7, 17], our algorithm ensures that every process eventually outputs in $\mathcal{R}_k^*$, assuming that the simulated algorithm ensures that every *correct* process eventually outputs.

Thus, a task is solvable using iterations of $\mathcal{R}_k$ *if and only if* it can be solved in the wait-free model using reads, writes, and k-set-consensus objects (or, equivalently, assuming k-concurrency). Therefore, the k-set-consensus model has a bounded representation as an *iterated affine task*: processes iteratively invoke instances of $\mathcal{R}_k$ a bounded number of times until they assemble enough knowledge to produce an output for the task they are solving.

Our results suggest a separation between “natural” models that have matching affine tasks and, thus, can be captured precisely by subsets of IIS runs and less “natural” ones, like WSB, having a manifold structure that is not affine [19]. We conjecture that such a combinatorial representation can also be found for a large class of restrictions of the wait-freedom, beyond k-concurrency and k-set consensus. This claim is supported by a recent derivation of the t-resilience affine task [32].

Related work. There have been several attempts to extend the topological characterization of [24] to models beyond the wait-free one [23, 16, 17, 28]. However, these results either only concern the special case of *colorless* tasks [23], consider weaker forms of solvability [16], introduce a new kind of *infinite* subdivisions [17], or also use non-iterated infinite subset of IIS runs [28].

In particular, Gafni et al. [17] characterized task solvability in models represented as subsets of IIS runs via *infinite* subdivisions of input complexes. This result assumes a limited notion of task solvability in the iterated model that only guarantees outputs to “fast” processes [11, 29, 7] that are “seen” by every other process infinitely often.

Concerning the reduction to iterated models, Imbs et al [25] showed that the model of iterated x-consensus, i.e., consensus among x processes, is equivalent regarding task solvability to the model of read-write memory and access to x-consensus objects.

In contrast with the earlier work, this paper studies the inherent combinatorial properties of general (colored) tasks and assumes the conventional notion of task solvability. Concurrently with the recently discovered affine task for *t-resilience* [32], our results truly capture the combinatorial structure of a restriction of the wait-free model.

Roadmap. The rest of the paper is organized as follows. Section 2 gives model definitions, briefly overviews the topological representation of iterated shared-memory models. In Section 3, we present the definition of $\mathcal{R}_k$ corresponding to the k-concurrency model. In Section 4, we show that $\mathcal{R}_k$ can be implemented in the k-set-consensus model and that any task solvable in the k-set-consensus model can be solved by iterating $\mathcal{R}_k$. Section 5 discusses related models and open questions.

2 Preliminaries

Let Π be a system composed of n asynchronous processes, $p_1, \ldots, p_n$. We consider two models of communication: (1) *atomic snapshots* [1] and (2) *iterated immediate snapshots* [5, 24].

Atomic snapshots. The atomic-snapshot (AS) memory is represented as a vector of shared variables, where processes are associated to distinct vector positions, and exports two operations: *update* and *snapshot*. An *update* operation performed by p_i replaces the shared variable at position i with a new value and a *snapshot* returns the current state of the vector. The model in which processes only have access to an AS memory is called the AS model.

Iterated immediate snapshots. In the iterated immediate snapshot (IIS) model, processes proceed through an ordered sequence of independent memories $M_1, M_2, \ldots$. Each memory M_r is accessed by a process with a single *immediate snapshot* operation [4]: the operation performed by p_i takes a value v_i and returns a set V_{ir} of values submitted by the processes (w.l.o.g, we assume that values submitted by different processes are distinct), so that the following properties are satisfied: (self-inclusion) $v_i \in V_{ir}$; (containment) $(V_{ir} \subseteq V_{jr}) \vee (V_{jr} \subseteq V_{ir})$; and (immediacy) $v_i \in V_{jr} \Rightarrow V_{ir} \subseteq V_{jr}$.

Protocols and runs. A *protocol* is a distributed automaton that, for each local state of a process, stipulates which operation and which state transition the process is allowed to perform in its next step. We assume here *deterministic* protocols, where only one operation and state transition is allowed in each state. A *run* of a protocol is defined as a possibly infinite sequence of states and operations.

In the IIS communication model, we assume that processes run the *full-information* protocol: the first value each process writes is its *initial state*. For each $r > 1$, the outcome of the immediate snapshot operation on memory M_{r-1} is submitted as the input value for the immediate snapshot operation on memory M_r. After a certain number of such (asynchronous) rounds, a process may gather enough information to *decide*, i.e., to produce an irrevocable non-$\perp$ output value. A *run* of the IIS communication model is thus a sequence V_{ir}, $i \in \mathbb{N}_n$ and $r \in \mathbb{N}$, determining the outcome of the immediate-snapshot operation for every process i and each iterated memory M_r.

Failures and participation. In the AS model (or the defined below AS model in which k-set-consensus objects can additionally be accessed), a process that takes only finitely many steps of the assigned protocol in a given run is called *faulty*, otherwise it is called *correct*. We assume that in its first step, a process writes its initial state in the shared memory using the *update* operation. If a process completed this first step it is said to be *participating*, the set of participating processes in a given run is called the *participating set*. Note that, since every process writes its initial state in its first step, the initial states of participating processes are eventually known to every process that takes sufficiently many steps.

In contrast, the IIS model does not have the notion of a faulty process. Instead, a process may appear "slow" [29, 10, 7], i.e., be late in accessing iterated memories from some point on so that some "faster" processes do not see them.

Tasks. In this paper, we focus on distributed *tasks* [24]. A process invokes a task with an *input* value and the task returns an *output* value, so that the inputs and the outputs across the processes, respect the task specification. Formally, a *task* is defined through a set $\mathcal{I}$ of input vectors (one input value for each process), a set $\mathcal{O}$ of output vectors (one output value for each process), and a total relation $\Delta : \mathcal{I} \mapsto 2^{\mathcal{O}}$ that associates each input vector with a set of possible output vectors. An input $\perp$ denotes a *non-participating* process and an output value $\perp$ denotes an *undecided* process. Check [22] for more details on the definition.

A protocol solves a task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ in a given model, if it ensures that in every infinite run of the model in which processes start with an input vector $I \in \mathcal{I}$, there is a finite prefix R of the run where: (1) all decided values form a vector $O \in \mathcal{O}$ such that $(I, O) \in \Delta$, and (2) every correct process has decided.

k-set-consensus and k-concurrency models. A k-set-consensus object can be accessed with a *propose* operation that takes a *proposed* value as an argument and returns a *decided* value as a response, so that, in each run, decided values constitute a subset of proposed values of size at most k. For $k \in \{1, \ldots, n-1\}$, in the *k-set-consensus model*, processes communicate via the AS memory and k-ste-consensus objects.

Let R be a finite run of the AS model. A process p_i is called *active at the end of R* if p_i participates in R but has not terminated by the end of R. Let *active*(R) denote the set of all processes that are active at the end of R.

A run R of the AS model is *k-concurrent* ($k = 1, \ldots, n$) if at most k processes are *concurrently active* in R, i.e., $\max\{|active(R')|;\ R' \text{ prefix of } R\} \leq k$. The *$k$-concurrency* model is the set of k-concurrent runs of the AS model.

It is known that the k-concurrency model is *equivalent* to the k-set-consensus model [13][1]: any task that can be solved k-concurrently can also be solved in the k-set-consensus model, and vice versa.

Standard chromatic subdivision and IIS. To give a combinatorial representation of the IIS model, we use the language of *simplicial complexes* [33, 22]. In short, a simplicial complex is defined as a set of *vertices* and an inclusion-closed set of vertex subsets, called *simplices*. The dimension of a simplex σ is the number of its vertices minus one. Any subset of these vertices is called a *face* of the simplex. A simplicial complex is *pure* (of dimension n) if each its simplices are contained in a simplex of dimension n.

A simplicial complex is *chromatic* if it is equipped with a *coloring function* – a non-collapsing simplicial map χ from its vertices to the *standard* $(n-1)$*-simplex* $\mathbf{s}$ of n vertices, in one-to-one correspondence with n *colors* $1, 2, \ldots, n$. All simplicial complexes we consider here are pure and chromatic. Please refer to the extended version of this paper [15] or [22] for more details on the formalism.

For a chromatic complex C, we let $\operatorname{Chr} C$ be the subdivision of C obtained by replacing each simplex in C with its *standard chromatic subdivision* [26]. The vertices of $\operatorname{Chr} C$ are pairs (v, σ), where p is a vertex of C and σ is a simplex of C containing v. Vertices (v_1, σ_1), …, (v_m, σ_m) form a simplex if all v_i are distinct and all σ_i satisfy the properties of immediate snapshots. Subdivision $\operatorname{Chr}^1 \mathbf{s}$ for the 2-dimensional simplex $\mathbf{s}$ is given in Figure 1. Each vertex represents a local state of one of the three processes p_1, p_2 and p_3 (red for p_1, blue for p_2 and white for p_3) after it takes a single immediate snapshot. Each triangle (2-simplex) represents a possible state of the system. A corner vertex corresponds to a local state in which the corresponding process only sees itself (it took its snapshot before the other two processes moved). An interior vertex corresponds to a state in which the process sees all three processes. The vertices on the 1-dimensional faces capture the snapshots of size 2.

If we *iterate* this subdivision m times, each time applying the same subdivision to each of the simplices, we obtain the m^{th} chromatic subdivision, $\operatorname{Chr}^m C$. It turns out that $\operatorname{Chr}^m \mathbf{s}$ precisely captures the m-round (full-information) IIS model, denoted IS^m [24]. Each run of IS^m corresponds to a simplex in $\operatorname{Chr}^m \mathbf{s}$. Every vertex v of $\operatorname{Chr}^m \mathbf{s}$ is thus defined

[1] In fact, this paper contains a self-contained proof of this equivalence result.

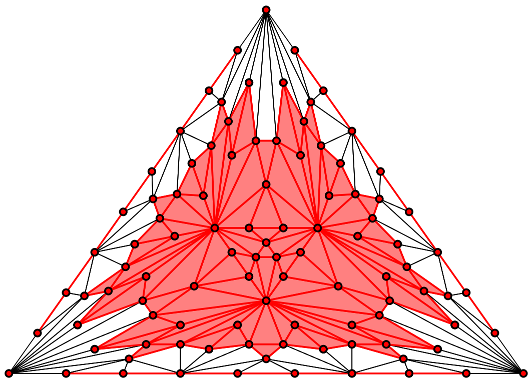


Figure 2 Contention sets (simplices in red) in a 3-process system.

as $(p, IS^1(p,\sigma), \dots, IS^m(p,\sigma))$, where each $IS^i(p,\sigma)$ is interpreted as the set of processes appearing in the i^{th} IS iteration obtained by p in the corresponding IS^m run. The *carrier* of vertex v is then defined as the set of all processes seen by p in this run, possibly through the views of other processes: it is the smallest face of $\mathbf{s}$ that contains v in its geometric realization. The carrier of a simplex is the maximal carrier of its vertices (related by inclusion).

Affine Tasks. As we show in this paper, the k-set-consensus model (and, thus, the k-concurrency model) can be captured by an iterated *affine* task [17]. Affine tasks can be seen as a generalization of simplex agreement tasks [5, 24], where the output complex is no longer a subdivision but a subset of some iteration of the standard chromatic subdivision. More formally, let L be a pure subcomplex of $\mathrm{Chr}^l\,\mathbf{s}$ for some $l \in \mathbb{N}$ of the dimension of $\mathbf{s}$. The affine task associated to L is then simply defined as $(\mathbf{s}, L, \Delta)$, where, for every face $\mathbf{t} \subseteq \mathbf{s}$, $\Delta(\mathbf{t}) = L \cap \mathrm{Chr}^l\,\mathbf{t}$. With a slight abuse of notations, the subcomplex L is used to denote the affine task associated to L.

By running m iterations of this task, we obtain L^m, a subcomplex of $\mathrm{Chr}^{lm}\,\mathbf{s}$, corresponding to a subset of IS^{lm} runs (each iteration includes l IS rounds). We denote by L^* the set of infinite runs of the IIS model where every prefix restricted to a multiple m of l IS rounds belongs to the subset of IS^{lm} runs associated to L^m.

3 The complex of k-set consensus

We now define $\mathcal{R}_k$, a subcomplex of $\mathrm{Chr}^2\,\mathbf{s}$, that precisely captures the ability of k-set consensus (and read-write memory) to solve tasks. The definition of $\mathcal{R}_k$ is expressed via a restriction on the simplices of $\mathrm{Chr}^2\,\mathbf{s}$ that bounds the size of *contention sets*. Informally, a contention set of a simplex $\sigma \in \mathrm{Chr}^2\,\mathbf{s}$ (or, equivalently, of an IS^2 run) is a set of vertices (or, processes) that "see each other". When a process p_i starts its IS^2 execution after another process p_j terminates, p_i must observe p_j's input, but not vice versa. Thus, a set of processes that see each others' inputs must have been concurrently active at some point.

Topologically speaking, a contention set of a simplex $\sigma \in \mathrm{Chr}^2\,\mathbf{s}$ is a set of vertices in σ sharing the same carrier, i.e., a minimal face $\mathbf{t} \subseteq \mathbf{s}$ that contains their vertices. Thus, for a given simplex $\sigma \in \mathrm{Chr}^2\,\mathbf{s}$, the set of contention sets is defined as follows:

▶ **Definition 1** (Contention sets). $\mathit{Cont}(\sigma) = \{\tau \subseteq \sigma, \forall v \in \tau, \mathit{carrier}(v) = \mathit{carrier}(\tau)\}$.

Contention sets for simplices of $\mathrm{Chr}^2\,\mathbf{s}$ in a 3-process system are depicted in Figure 2: for each simplex $\sigma \in \mathrm{Chr}^2\,\mathbf{s}$, every face of σ that constitutes a red simplex is a contention

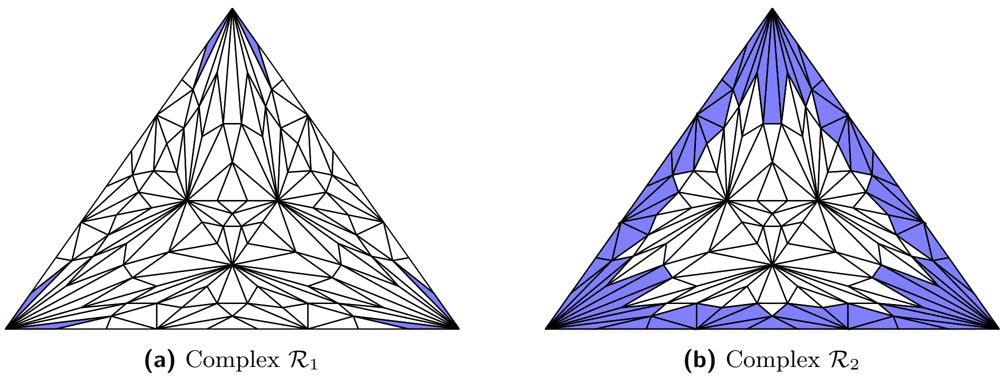


Figure 3 $\mathcal{R}_1$ and $\mathcal{R}_2$ (in blue) for 3 processes.

set of σ. In an interior simplex, every set of vertices are contention sets. Every "total order" simplex (shown in blue in Figure 3a), matching a run in which processes proceed, one by one, in the same order in both IS^1 and IS^2, has only three singleton as contention sets. All other simplices include a contention set of two processes. In the associated IIS run, processes are said to be *contending* if they are associated to vertices forming a contention set.

Now $\mathcal{R}_k$ is defined as the set of all simplices in $\mathrm{Chr}^2\,\mathbf{s}$, in which the contention sets have cardinalities of at most k:

▶ **Definition 2** (Complex $\mathcal{R}_k$). $\mathcal{R}_k = \{\sigma \in \mathrm{Chr}^2\,\mathbf{s}, \forall \tau \in \mathit{Cont}(\sigma), |\tau| \leq k\}$.

It is immediate to see that the set of simplices in $\mathcal{R}_k$ constitutes a simplicial complex: every face τ of $\sigma \in \mathcal{R}_k$ is also in $\mathcal{R}_k$.

Examples of $\mathcal{R}_1$ and $\mathcal{R}_2$ for a 3-process system are shown in Figures 3a and 3b, respectively. Obviously, for the unrestricted 3-set consensus case, $\mathcal{R}_3 = \mathrm{Chr}^2\,\mathbf{s}$. Note that $\mathcal{R}_1$ only contains six "total order" simplices, while $\mathcal{R}_2$ consists of all simplices of $\mathrm{Chr}^2\,\mathbf{s}$ that touch the boundary.

4 From k-set consensus to $\mathcal{R}_k^*$ and back

In this section, we show that any task solvable in the k-set consensus model can be solved in $\mathcal{R}_k^*$, and vice versa. The main result is established via *simulations*: a run of an algorithm solving a task in one model is simulated in the other.

4.1 From k-set consensus to k-concurrency

We first show that the k-concurrency model is equivalent, regarding task solvability, to the k-set-consensus model. This result has been stated in a technical report [13], but no explicit proof appears in the literature, and we fill the gap below.

Simulating a k-process shared memory system. We employ *generalized state machines* (proposed in [14] and extended in [30]) that allow us to simulate an *inputless* (i.e., simulated processes have a default initial state) k-process read-write memory system in the k-set-consensus model. To ensure consistency of simulated operations, we use *commit-adopt* objects [11] that can be implemented using AS. A commit-adopt object exports one operation *propose*(v) that takes a parameter in an arbitrary range and returns a couple (flag, v'), where *flag* can be either *commit* or *adopt* and where v' is a previously proposed value. Moreover, if

Algorithm 1: k processes shared memory system simulation: process p_i.

```
SharedObjects: $KSC[1\dots]$ $k-$simultaneous consensus objects;
$CA[1\dots][1\dots k]$ : commit − adopt objects;
$MEM[1\dots n][1\dots k]$ init $(-1,\perp)$ : single writer shared memory array;
Init: $r_i \leftarrow 0$; foreach $m \in \{1,\dots,k\}$ do $(WC_i[m], View_i[m]) \leftarrow (0,\emptyset)$;

Repeat forever
    $r_i \leftarrow r_i + 1$;
    $(Index_i, Value_i) \leftarrow KSC[r_i].propose(WC, View_i)$;
    $(Flag_i[Index_i], val_i) \leftarrow CA[r_i][Index_i].propose(Value_i)$;
    if $val_i = (c,*)$ with $c \geq WC_i[Index_i]$ then $(WC_i[Index_i], View_i[Index_i]) \leftarrow val_i$;
    foreach $m \in \{1,\dots,k\} \setminus Index_i$ do
        $(Flag_i[m], val_i) \leftarrow CA[r_i][m].propose(View_i[m])$;
        if $val_i = (c,*)$ with $c \geq WC_i[m]$ then $(WC_i[m], View_i[m]) \leftarrow val_i$;
    foreach $m \in \{1,\dots,k\}$ do
        if $Flag_i[m] = Commit$ then
            $MEM[i][m].Update(WC_i[m], WriteVal(WC_i[m], View_i[m]))$;
            $WC_i[m] \leftarrow WC_i[m] + 1$;
            $View_i[m] = CurWrites(MEM.Snapshot())$;
End repeat;

With CurWrites $(MEM_{val})$=
    foreach $m \in \{1,\dots,k\}$ do $curWC[m] = -1$, $curWrite[m] = \perp$;
    foreach $(m,l) \in \{1,\dots,k\} \times \{1,\dots,n\}$ do
        if $MEM_{val}[l][m].WC > curWC[m]$ then
            $curWC[m] = MEM_{val}[l][m].WC$, $curWrite[m] = MEM_{val}[l][m].Value$;
    return $curWrite$;
```

a process returns a *commit* flag, then every process must return the same value. Further, if no two processes propose different values, then all returned flags must be *commit.*

Liveness of the simulation relies on calls to k-*simultaneous consensus* objects [2]. To access a k-simultaneous consensus object, a process proposes a vector of k inputs, one for each of the *consensus instances*, $1, 2, \dots, k$, and the object returns a couple(i, v), where *index* i belongs to $\{1, \dots, k\}$ and v is a value proposed by some process at index i. It ensures that no two processes obtain different values with the same index. Moreover, if $\ell \leq k$ distinct input vectors are proposed then only values at indices $1, \dots, \ell$ can be output. The k-simultaneous consensus object is equivalent to k-set-consensus in the read-write shared-memory system [2].

Our simulation is described in Algorithm 1. We use three shared abstractions: an infinite array of k-simultaneous consensus objects KSC, one object per round, an infinite array of arrays of k indexed commit-adopt objects CA, i.e., k objects per round, and a single-writer multi-reader memory MEM with k slots.

In every round, the simulator starts by accessing the round k-simultaneous-consensus object with its local estimation of the simulated system state (line 7). Then the simulator access the commit-adopt object associated with the index, and using the state estimation as proposed value, output earlier by the k-simultaneous-consensus object (line 8). After that the simulator go through the $k-1$ remaining commit-adopt objects of the round, using its current estimation of the simulated processes state as proposals (lines 8–11). It is guaranteed that at least one process commits, in particular, process p_j that is the first to return from its first commit-adopt invocation in this round (on a commit-object C), because any other process with a different proposal must access a different commit-adopt object first and, thus, it must invoke C after p_j returns. To ensure that a unique written value is selected, simulators replace their current proposal values with the values returned by the commit-adopt objects (lines 8–11). Note that the processes do not select values corresponding to an older round of simulation, to ensure that processes do not alternate committing and adopting the same value indefinitely.

In the simulation, the simulators propose snapshot results for the simulated processes. Once a proposed snapshot has been committed, a simulator stores in the shared memory the value that the simulated process must write in its next step (based on its simulated algorithm), equipped with the corresponding *write counter* (line 15). The write counter is then incremented and a new snapshot proposal is computed (line 17). To compute a simulated snapshot, for each process, the most recent value available in the memory MEM is selected by comparing the write counters WC (auxiliary function *CurWrites* at lines 19–24).

▶ **Lemma 3.** *Algorithm 1 provides a non-blocking simulation of an inputless k-process AS memory system in the k-set consensus model. Moreover, if there are $\ell < k$ active processes, then one of the first ℓ simulated processes is guaranteed to make progress.*

The proof of Lemma 3 can be found in the companion technical report [15]. The proof is constructed by showing that: (1) No two different written values are computed for the same simulated process and the same write counter; (2) In every round of the simulation, at least one simulator commits a new simulated operation; (3) Every committed simulated snapshot operation can be linearized at the moment when the actual snapshot operation which served for its computation took place; and (4) Every simulated write operation can be linearized to the linearization time of the first actual write performed by a simulator with the corresponding value.

Using the extended BG-simulation to simulate a k-concurrent execution. We have shown that a k-process inputless AS memory system can be simulated in the k-set-consensus model. In its turn, the simulated system can be used to simulate a k-concurrent execution by running an extended *BG-simulation* protocol [3, 6].

The BG-simulation technique allows $k+1$ processes $s_1, \ldots, s_{k+1}$, called *BG-simulators*, to simulate, in a wait-free manner, a *k-resilient* run of any protocol $\mathcal{A}$ on m processes $p_1, \ldots, p_m$ $(m > k)$. The simulation guarantees that each simulated step of every process p_j is either agreed upon by all simulators, or is *blocked* because of a slow or faulty simulator (and one less simulator participates further in the simulation for each step which is not agreed on).

In the original BG simulation, a faulty simulator may indefinitely block an arbitrary simulated process. The *extended BG-simulation* [12] additionally exports an *abort* operation that, when applied to a blocked simulated process, re-initializes it, so that the process can move forward until an output for it is computed, or another simulator makes it block again. The abort mechanism must be used carefully in order to keep liveness properties, by, for example, ensuring that the slow or crashed simulator will not participate again in the simulation. Refer [3, 6, 12] for a more formal description of the BG-simulation technique.

As previously done in [9], the main idea of the simulation is to run a *depth-first* BG-simulation (i.e., simulate the more advanced available code) instead of a classical *bread-first* BG-simulation (i.e., simulate the least advanced available code). Indeed, the BG-simulation consists in executing pieces of shared-memory codes in any valid order (i.e., respecting simulated threads dependencies). But to prevent starvation of the BG-simulators, one cannot wait to simulate the next piece of code a blocked thread and thus must move to another thread. But by selecting the more advanced available thread, then, as we will verify, this provides a k-concurrent simulation when executed by k BG-simulators.

Three aspects requires clarification. First, the threads we want to execute k-concurrently require input values, initially available only to the corresponding process. This is resolved by simply having processes write their initial state to the shared memory before participating in the BG-simulation executed on the k-process inputless system provided by Algorithm 1.

Algorithm 2: Process code for the k-concurrent simulation for p_i.

```
Thread[i] ← Input;
while Thread[i]!=Complete do
    Execute 1 round of Algorithm 1(Algorithm 3);
return Thread[i].output();
```

Algorithm 3: Code for BG-simulator number m.

```
Repeat forever
    Let $A = \{i \in \{1,\dots,n\}, Thread[i]! = NotInitialized \wedge Thread[i]! = Complete\}$;
    if $|A| > m$ then
        Let $Blocked = \{i \in A, Thread[i] = Blocked\}$;
        if $A = Blocked$ then
            forall $i \in A$ do  $Thread[i].Abort()$ ;
        Let $j = \mathbf{argmax}_{i \in A \setminus Blocked} Thread[i].Progress()$;
        $Execute(Thread[i].NextStep())$;
End repeat;
```

This way there is at least as many *opened* threads (i.e., a thread associated with a process with an input value visible to all simulators) as possible active BG-simulators.

Secondly, as threads are used to execute an algorithm solving a task, they may terminate, thus reducing the number of threads available. This is why processes execute Algorithm 1 one round at a time: At each round each process checks if its own thread is complete, and if so it stops and it returns with its task output. This ensures that the number of processes active in Algorithm 1 execution follows, with some latency, the number of threads available.

Lastly, when there is $\ell < k$ available threads, the last $(k - \ell)$ BG-simulators stop participating in the simulation to adapt the number of active BG-simulators to the number of available threads. Moreover, a BG-simulator uses the abort mechanism, on all active threads, if it is one of the first ℓ BG-simulators and if every active thread is currently blocked.

These algorithms are available in Algorithm 2 for the process code, and in Algorithm 3 for the BG-simulator code. The threads correspond to process simulations. A thread is said to be *NotInitialized* if the corresponding process did not provide yet its input value (Alg. 2, l. 1). Otherwise it can be either *Blocked*, *Available*, or *Complete*. If it is initialized, *Progress()* returns the number of simulation steps that has been performed, possibly 0. Note that in the case where multiple threads have the same progress when selecting the most advanced one (Alg. 3, l. 7), any one of them can be selected.

▶ **Lemma 4.** *All tasks solvable in the k-concurrency model can be solved in the k-set-consensus model.*

The proof of Lemma 4 is quite tedious, and therefore relegated to the associated technical report [15], even if the construction of the algorithms and the main aspect of their correctness are quite simple and natural.

The main argument relies on the fact that the number of processes participating in Algorithm 1 follows the number of active threads. Thus, when the number of active threads ℓ is smaller than k, we have the property that Algorithm 1 provides progress to one out of the first ℓ simulated processes (see Lemma 3). Therefore, one out of the ℓ first BG-simulators takes infinitely many steps. But as only the first ℓ BG-simulators remains active, the ℓ active threads cannot remain blocked by slow BG-simulators. The technical difficulties relies in showing that the use of the abort mechanism does not cause trouble as well as the delays to reach stability after a change in the number of active threads. On another hand, the safety of the simulation is quite easy to show, as at most $k-1$ active threads may be blocked when selecting a thread to simulate, if k already took steps and did not terminate then one of these k will be selected to continue further its simulation.

4.2 From k-concurrency to $\mathcal{R}_k$.

We now show that k-concurrency can *solve* $\mathcal{R}_k$, i.e., it can solve the affine task on the subcomplex $\mathcal{R}_k$. In fact, running any algorithm implementing immediate snapshot in the k-concurrency model would do the job.

▶ **Lemma 5.** *The two-round immediate snapshot algorithm solves the affine tasks $\mathcal{R}_k$ in the k-concurrency model.*

Proof. Consider two iterated rounds of any IS algorithm (e.g., [4]) in the k-concurrency model. The set of IS^2 outputs of this algorithm, forms a valid simplex σ in $\mathrm{Chr}^2\,\mathbf{s}$ [5].

Let σ be any such simplex. Let S be the contention set of σ containing two vertices v and v', and let p and q be the processes corresponding to the vertices v and v'. By contradiction, suppose that p and q were not concurrently active in the corresponding IS^2 run. Without loss of generality, suppose that p's computation was terminated before the activation of q, so p cannot be aware of q's input. Thus, p cannot see q, which implies that v and v' have different carriers, which contradicts the definition of a contention set.

Therefore, all vertices in a contention set are associated to processes that were *active* at the same time. Thus, k-concurrent runs produce simplices in which contention sets are of size at most k and, thus, belong to $\mathcal{R}_k$. Since the simplices of $\mathcal{R}_k$ output by the simulation correspond to the participating processes only, we indeed get an algorithm solving chromatic simplex agreement on complex $\mathcal{R}_k$. ◀

Now it is easy to show that the k-set consensus model is, regarding task solvability, at least as strong as the $\mathcal{R}_k^*$ model:

▶ **Theorem 6.** *Any task solvable by $\mathcal{R}_k^*$ can be solved in the k-set-consensus model.*

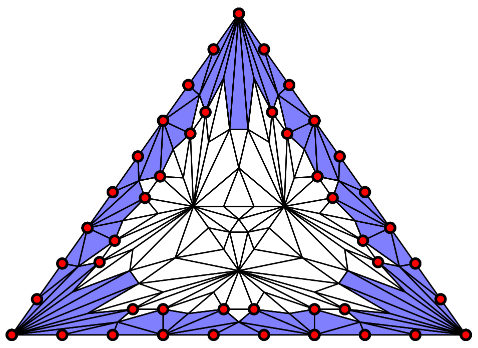
Proof. By Lemma 5, the affine task on $\mathcal{R}_k$ is solvable in the k-concurrency model. Moreover, according to Lemma 4, any task solvable in the k-concurrency model can be solved in the k-set consensus model. Therefore, by iterating a solution to the affine task $\mathcal{R}_k$, a run of $\mathcal{R}_k^*$ can be simulated in the k-set-consensus model and used to solve any task solvable in $\mathcal{R}_k^*$. ◀

4.3 From $\mathcal{R}_k^*$ to k-set consensus

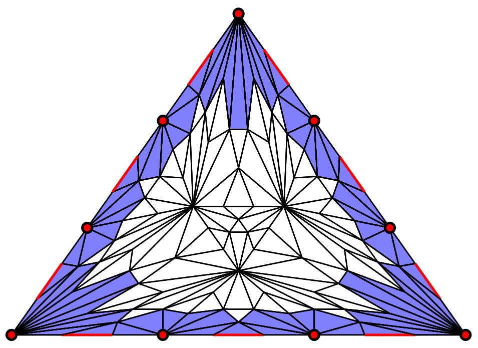
Now we show how to simulate in $\mathcal{R}_k^*$ any algorithm that uses the AS memory and k-set-consensus objects.

k-set consensus simulation design. A *non-blocking* simulation of the AS memory in $\mathcal{R}_k^*$ is straightforward, since the set of $\mathcal{R}_k^*$ runs is a subset of $(\mathrm{Chr}(\mathbf{s}))^*$ runs, and there exist several algorithms simulating the AS memory in $(\mathrm{Chr}(\mathbf{s}))^*$, e.g., [18].

Solving k-set consensus is not very complicated either in $\mathcal{R}_k^*$: every iteration of $\mathcal{R}_k$ provides a set of at most k *leaders*, i.e., processes with an IS^1output containing at most k elements, where at least one such *leader* is visible to every process, i.e., it can be identified as a *leader* and its input is visible to all. The set of leaders of $\mathcal{R}_2$ are shown in figure 4a in red, it is easy to observe that every simplex in $\mathcal{R}_2$ has at most two leaders, and that one is visible to every process (every process with a carrier of size at most 2 is a leader). This



(a) Leaders.



(b) Processes with the smallest IS^2 output.

Figure 4 $\mathcal{R}_2$ for 3 processes: (a) leaders – vertices in red, and (b) processes with the smallest IS^2 output – simplices in red.

property gives a very simple k-set-consensus algorithm: every process decides on the value proposed by one of these k leaders. We will later show how this property can be derived from the restriction of $\mathcal{R}_k$ on the size of *contention sets*.

The real difficulty of the simulation consists in combining the shared-memory and k-set-consensus simulations, as in the simulated protocol, some processes may be accessing k-set-consensus objects while other processes are performing AS operations. Liveness of our k-set-consensus algorithm relies on the *participation* of visible leaders, i.e., on the fact that the leaders propose values for this instance of k-set-consensus. In this sense, our k-set-consensus algorithm may block if some leader is performing an AS operation or is involved in a different instance of k-set-consensus. Similarly if a "fast" process is involved in a k-set-consensus, then it can prevent every "slow" process to complete any AS memory operation as a write may be validated only after having been observed by every active process.

The solution we propose consists in (1) synchronizing the two simulations in order to ensure that, eventually, at least one process will complete its pending operation, and (2) ensuring that the processes collaborate by participating in every simulated operation. In our solution, every process tries to propagate every observed proposed value (for a write operation), and every process tries to reach a decision in every k-set-consensus object accessed by some process. For that, we make the processes participate in both simulation protocols (read-write and k-set-consensus) in every round of $\mathcal{R}_k^*$, until they decide.

Even though the simulated algorithm executes only one operation at a time and requires the output of the previous operation to compute the input for the following one, we enrich the simulated process with *dummy* operations that do not alter the simulation result. Then eventually some *undecided* process is guaranteed to complete both pending operations, where at most one of them is a *dummy* one. This scheme provides a *non-blocking* simulation of any algorithm using the AS memory and k-set-consensus objects.

We use the following observation. The shared memory simulation from [18] provides progress to the processes with the smallest snapshot output (i.e., with the smallest set of observed processes values). Our k-set-consensus algorithm provides progress to the leaders with the smallest $\mathcal{R}_k$ output, i.e., the processes with the smallest associated carrier. We synchronize the liveness properties of the two simulations by running the AS simulation only on every second round of the two rounds of restricted immediate snapshots associated to $\mathcal{R}_k$, denoted IS^2. For example, Figure 4b depicts the 2-dimensional of $\mathcal{R}_2$, where the sets of processes with the smallest IS^2 outputs are represented as red simplices.

Algorithm 4: k-set consensus simulation in $\mathcal{R}_k^*$: process i.

```
1  Init: $r_i \leftarrow 0$; $State_i \leftarrow undecided$; $ConsId_i \leftarrow \perp$; $ConsProp_i \leftarrow \perp$;
2  $WriteVal_i[i] \leftarrow$ FirstWrite_i(); $WriteCount_i[i] \leftarrow 1$;
3  foreach $m \in \{1, \ldots, n\} \setminus \{i\}$ do $(WriteCount_i[m], WriteVal_i[m]) \leftarrow (0, \perp)$;
4  $ConsHistory_i \leftarrow \emptyset$ : List of adopted agreement proposals;

5  Repeat forever
6      $r_i \leftarrow r_i + 1$; $Leaders_i \leftarrow true$;
7      $IS^2{}_{output} = \mathcal{R}_k[r_i](State_i, (WriteCount_i, WriteVal_i), ConsHistory_i)$;

8      foreach $(j, View_j) \in IS^2{}_{output}$ do
9          Let $(State_j, (WriteCount_j, WriteVal_j), ConsHistory_j) \leftarrow$ RKInput$(j)$;
10         foreach $m \in \{1, \ldots, n\}$ do
11             if $WriteCount_j[m] > WriteCount_i[m]$ then
12                 $WriteCount_i[m] = WriteCount_j[m]$, $WriteVal_i[m] = WriteVal_j[m]$;
13         if $|$Undecided$(View_j)| \leq k$ then
14             if $\nexists (ConsId_i, *) \in ConsHistory_j$ then $Leaders_i \leftarrow false$;
15             foreach $(A_{id}, A_{val}) \in ConsHistory_j$ do
16                 ReplaceOrAdd $(A_{id}, *)$ in $ConsHistory_i$ with $(A_{id}, A_{val})$;

17     if $(\Sigma_{m \in \{1,\ldots,n\}} WritesCount_i[m]) = r_i$ then
18         if PendingWriteSnapshotOperation() then
19             TerminateWriteOperation$(WriteVal_i)$;
20         if $Leaders_i \wedge ConsId_i \neq \perp$ then
21             $ConsProp_i \leftarrow A_{val}$ where $(ConsId_i, A_{val}) \in ConsHistory_i$;
22             TerminateAgreementOperation$(A_{val})$; $ConsId_i \leftarrow \perp$;
23         if Terminated() then $State \leftarrow decided$;
24         else
25             $WriteCount_i[i] \leftarrow WriteCount_i[i] + 1$;
26             if NextAgreementOperation() = Available then
27                 $(ConsId_i, ConsProp_i) \leftarrow$ NextAgreement_i();
28                 if $(\nexists (A_{id}, A_{val}) \in ConsHistory_i$ with $A_{id} = ConsId_i)$ then
29                     Add $(ConsId_i, ConsProp_i)$ in $ConsHistory_i$;
30             if NextWriteSnapshotOperation() = Available then
31                 $WriteVal_i[i] \leftarrow$ NextWrite_i();
32 End repeat;
```

This way at least the leader with the smallest $\mathcal{R}_k$ output will make progress in both simulations. Indeed, the definition of $\mathcal{R}_k$ implies that the set of processes with the smallest $\mathcal{R}_k$ outputs includes a leader, and a process with the smallest IS^2 output also has the smallest $\mathcal{R}_k$ output. Figure 4 gives an example of an intersection between the set of processes with the smallest IS^2 output and the set of leaders: here every process with the smallest IS^2 output has a carrier of size at most 2 and every such process is a leader.

k-set consensus simulation algorithm. Algorithm 4 provides a simulation, in $\mathcal{R}_k^*$, of any protocol designed for the k-set-consensus model. The algorithm is based on the shared memory simulation from [18], applied on IS^2outputs of every iteration of $\mathcal{R}_k$, combined with a parallel execution of instances of our k-set-consensus algorithm. The simulation works in rounds that can be decomposed into three stages: communicating through $\mathcal{R}_k$, updating local information, and validating progress.

The first stage consists in accessing the new $\mathcal{R}_k$ iteration associated with the round, using information on the ongoing operations as an input (see line 7). For memory operations, an input to $\mathcal{R}_k$ consists of an array containing the most recent known *update* operation for every process, $WriteVal_i$, and the timestamp associated with the written value, $WriteCount_i$; $ConsHistory_i$ is a list of all adopted proposals for all accessed agreement operations. Finally, a value *State*, set to *decided* or *undecided*, is also put in $\mathcal{R}_k$'s input, to indicate whether the process has completed its simulation.

The second stage consists in updating the local information according to the output obtained from $\mathcal{R}_k$ (lines 8–16). The input value of each process observed in the second immediate snapshot of $\mathcal{R}_k$ is extracted (line 9). These selected inputs are examined in order to replace the local write values $WriteCount_i$ with the most recent ones, i.e., associated with the largest write counters (lines 10–12). The $ConsHistory$ variable of every leader, i.e., a process with an IS^1 output containing at most k *undecided* process inputs (using the variable *State*), is scanned in order to adopt all its decision estimates (lines 13–16). Moreover, the boolean value $Leaders_i$ is used to check if every observed leader transmitted a decision estimate for the pending agreement operation, $ConsId_i$.

The third stage consists in checking whether pending operations can safely be terminated (lines 17–22), and if so, whether the process has completed its simulation (line 23) or if new operations can be initiated (line 24–31).

Informally, it is safe for a process to decide in line 20, as there are at most k *Leaders* per round, one of which (1) is visible to every process and (2) provides a decision estimate for the pending agreement. Thus, every process adopts the decision estimate from a *leader* of the round, reducing the set of possible distinct decisions to k.

A pending memory operation terminates when the round number r_i equals the sum of the currently observed write counters (test at line 17), as in the original algorithm [18]. Indeed, the equality implies that the writes in the estimated snapshot have been observed by every process (line 19). Last, if a process did not terminate, it increments its write counter and, if there is a new operation available, the process selects the operation (see lines 25–31).

If there is a new agreement operation, then the input proposal and the object identifier are selected (line 27) and they are used for the current decision estimate in $ConsHistory_i$ (line 29), unless a value has already been adopted (line 28). If there is a new write operation then the current write value is simply changed (line 31), a *dummy* write thus consists in re-writing the same value.

▶ **Lemma 7.** *In $\mathcal{R}_k^*$, Algorithm 4 provides a non-blocking simulation of any algorithm designed for the k-set-consensus model.*

The proof of Lemma 7 is delegated to the companion technical report [15]. The main aspects of the proof are taken from the base algorithm from [18], while the liveness of the agreement operations relies on the restriction provided by $\mathcal{R}_k^*$ and the size of *contention sets*.

Lemma 7 implies the following result:

▶ **Theorem 8.** *Any task solvable in the k-set-consensus model can be solved in $\mathcal{R}_k^*$.*

Proof. To solve in $\mathcal{R}_k^*$ a task solvable in the k-set-consensus model, we can simply use Algorithm 4, simulating any given algorithm solving the task in the k-set-consensus model.

The non-blocking simulation provided by Algorithm 4 ensures, at each point, that at least one live process eventually terminates. As there are only finitely many processes, every live process eventually terminates. ◀

Lemma 4, Theorem 6, and Theorem 8 imply the following equivalence result:

▶ **Corollary 9.** *The k-concurrency model, the k-set-consensus model, and $\mathcal{R}_k^*$ are equivalent regarding task solvability.*

This equivalence result can be used to derived a generalization of the *asynchronous computability theorem* from [24] in its discrete formulation from what it means for a task to be solvable in $\mathcal{R}_k^*$:

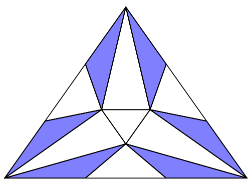


Figure 5 Fully ordered sub-Chr **s**.

▶ **Theorem 10.** *Discrete k-Concurrent ACT: A task $(\mathcal{I}, \mathcal{O}, \Delta)$ is solvable in the k-concurrency or k-set-consensus model if and only if there exists a color-preserving, carrier-preserving, simplicial map $\phi\colon \mathcal{R}_k^N \to \mathcal{O}$ for some natural number N.*

5 Concluding remarks: on minimality of $\mathrm{Chr}^2\,\mathbf{s}$ for k-set consensus

This paper shows that the models of k-set consensus and k-concurrency are captured by the same affine task $\mathcal{R}_k$, defined as a subcomplex of $\mathrm{Chr}^2\,\mathbf{s}$. One may wonder if there exists a simpler equivalent affine task, defined as a subcomplex of $\mathrm{Chr}\,\mathbf{s}$, the 1-degree of the standard chromatic subdivision. Just like for the t-resilient affine task [8, 32], this is in general not possible. Consider the case of $k = 1$ (consensus) in a 3-process system. We can immediately see that the corresponding subcomplex of $\mathrm{Chr}\,\mathbf{s}$ must contain all “ordered” simplexes depicted in Figure 5. Indeed, we must account for a wait-free 1-concurrent IS^1 run in which, say, p_1 runs first until it completes (and it must outputs its corner vertex in $\mathrm{Chr}\,\mathbf{s}$), then p_2 runs alone until it outputs its vertex in the interior of the face (p_1, p_2) and, finally, p_3 must output its interior vertex.

The derived complex is connected. Moreover, any number of its iterations still results in a connected complex. The simple connectivity argument implies that consensus cannot be solved in this iterated model and, thus, the complex cannot capture 1-concurrency.

Interestingly, the complex in Figure 5 precisely captures the model in which, instead of consensus, weaker *test-and-set* (TS) objects are used: (1) using TS, one easily make sure that at most one process terminates at an IS level, and (2) in IS runs defined by this subcomplex, any pair of processes can solve consensus using this complex and, thus, a TS object can be implemented. It is not difficult to generalize this observation to k-*TS* objects [27]: the corresponding complex consists of all simplices of $\mathrm{Chr}\,\mathbf{s}$, contention sets of which are of size at most k. The equivalence (requiring a simple generalization for the backward direction) can be found in [27, 20].

Overall, this raises an intriguing question whether every object, when used in the read-write system, can be captured via a subcomplex of $\mathrm{Chr}^m\,\mathbf{s}$ for some $m \in \mathbb{N}$.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.

2 Yehuda Afek, Eli Gafni, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. The k-simultaneous consensus problem. *Distributed Computing*, 22(3):185–195, 2010.

3 Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100, 1993.

4 Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, pages 41–51, 1993.

5 Elizabeth Borowsky and Eli Gafni. A simple algorithmically reasoned characterization of wait-free computation (extended abstract). In *PODC*, pages 189–198, 1997.

6 Elizabeth Borowsky, Eli Gafni, Nancy A. Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.

7 Zohir Bouzid, Eli Gafni, and Petr Kuznetsov. Strong equivalence relations for iterated models. In *OPODIS*, pages 139–154, 2014.

8 Carole Delporte, Hugues Fauconnier, Sergio Rajsbaum, and Michel Raynal. t-resilient immediate snapshot is impossible. In *SIROCCO*, pages 177–191, 2016.

9 Pierre Fraigniaud, Eli Gafni, Sergio Rajsbaum, and Matthieu Roy. Automatically adjusting concurrency to the level of synchrony. In *DISC*, pages 1–15, 2014.

10 Eli Gafni. On the wait-free power of iterated-immediate-snapshots. Unpublished manuscript, `http://www.cs.ucla.edu/~eli/eli/wfiis.ps`, 1998.

11 Eli Gafni. Round-by-round fault detectors (extended abstract): Unifying synchrony and asynchrony. In *PODC*, 1998.

12 Eli Gafni. The extended BG-simulation and the characterization of t-resiliency. In *STOC*, pages 85–92, 2009.

13 Eli Gafni and Rachid Guerraoui. Simulating few by many: Limited concurrency = set consensus. Unpublished manuscript, `http://web.cs.ucla.edu/~eli/eli/kconc.pdf`, 2009.

14 Eli Gafni and Rachid Guerraoui. Generalized universality. In *CONCUR*, pages 17–27, 2011.

15 Eli Gafni, Yuan He, Petr Kuznetsov, and Thibault Rieutord. Read-write memory and k-set consensus as an affine task. *arXiv preprint arXiv:1610.01423*, 2016.

16 Eli Gafni and Petr Kuznetsov. Relating l-resilience and wait-freedom via hitting sets. In *ICDCN*, pages 191–202, 2011.

17 Eli Gafni, Petr Kuznetsov, and Ciprian Manolescu. A generalized asynchronous computability theorem. In *PODC*, pages 222–231, 2014.

18 Eli Gafni and Sergio Rajsbaum. Distributed programming with tasks. In *OPODIS*, pages 205–218, 2010.

19 Eli Gafni, Sergio Rajsbaum, and Maurice Herlihy. Subconsensus tasks: Renaming is weaker than set agreement. In *DISC*, pages 329–338, 2006.

20 Eli Gafni, Michel Raynal, and Corentin Travers. Test & set, adaptive renaming and set agreement: A guided visit to asynchronous computability. In *SRDS*, pages 93–102, 2007.

21 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

22 Maurice Herlihy, Dmitry N. Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology.* Morgan Kaufmann, 2014.

23 Maurice Herlihy and Sergio Rajsbaum. The topology of shared-memory adversaries. In *PODC*, pages 105–113, 2010.

24 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(2):858–923, 1999.

25 Damien Imbs, Sergio Rajsbaum, and Adrián Valle. Untangling partial agreement: Iterated x-consensus simulations. In *SSS*, pages 139–155, 2015.

26 Dmitry N. Kozlov. Chromatic subdivision of a simplicial complex. *Homology, Homotopy and Applications*, 14(2):197–209, 2012.

27 Achour Mostéfaoui, Michel Raynal, and Corentin Travers. Exploring gafni's reduction land: From $Omega^k$ to wait-free adaptive (2p-[p/k])-renaming via k-set agreement. In *DISC*, pages 1–15, 2006.

28 Sergio Rajsbaum, Michel Raynal, and Corentin Travers. The iterated restricted immediate snapshot model. In *COCOON*, pages 487–497, 2008.

29 Michel Raynal and Julien Stainer. Increasing the power of the iterated immediate snapshot model with failure detectors. In *SIROCCO*, pages 231–242, 2012.

30 Michel Raynal, Julien Stainer, and Gadi Taubenfeld. Distributed universality. In *OPODIS*, pages 469–484, 2014.

31 Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. on Computing*, 29:1449–1483, 2000.

32 Vikram Saraph, Maurice Herlihy, and Eli Gafni. Asynchronous computability theorems for t-resilient systems. In *DISC*, pages 428–441, 2016.

33 Edwin H. Spanier. *Algebraic topology.* McGraw-Hill Book Co., New York, 1966.