Combinatorial characterization of asynchronous distributed computability

Thibault Rieutord

université PARIS-SACLAY

TELECOM ParisTech

NNT : 2018SACLT007

Thèse de doctorat

Combinatorial Characterization of Asynchronous Distributed Computability

Thèse de doctorat de l'Université Paris-Saclay
préparée à Télécom ParisTech

Ecole doctorale n°580 Sciences et technologies
de l'information et de la communication (STIC)
Spécialité de doctorat : Mathématiques et Informatique

Thèse présentée et soutenue à Paris, le 25 Octobre 2018, par

THIBAULT RIEUTORD

Composition du Jury :

Carole Delporte-Gallet Professeur, Université Paris-Diderot Institut de Recherche en Informatique Fondamentale	**Président du Jury**
Emmanuel Godard Professeur, Université d'Aix-Marseille Laboratoire d'Informatique Fondamentale de Marseille	**Rapporteur**
Maurice Herlihy Professor, Brown University Computer Science Department	**Rapporteur**
Joffroy Beauquier Professeur, Université Paris-Sud Laboratoire de Recherche en Informatique	**Examinateur**
Maria Potop-Butucaru Professeur, Sorbonne Université Laboratoire d'Informatique de Paris 6	**Examinateur**
Petr Kuznetsov Professeur, Télécom ParisTech Laboratoire Traitement et Communication de l'Information	**Directeur de thèse**

Résumé

Les systèmes informatiques modernes sont des systèmes distribués, allant de multiples processeurs sur une même puce à des systèmes internet de large échelle. Dans cette thèse nous étudions les problèmes de calculabilité et de complexité dans les systèmes distribués *asynchrones communiquant par mémoire partagée.*

Dans la première et majeure partie de cette thèse, nous étudions la capacité des modèles communiquant par mémoire partagée à résoudre des *tâches distribuées.* Notre première contribution est une technique de *simulation distribuée* utilisant la capacité d'accord du système afin de synchroniser les différents processus entre eux. Cette technique de simulation permet de comparer la capacité de différents modèles à résoudre des tâches distribuées. À l'aide de cet outil, nous montrons que pour les modèles d'adversaires en mémoire partagée, la capacité à résoudre un ensemble particulier de tâches d'accord permet de déterminer sa capacité à résoudre n'importe quelle tâche distribuée. Nous utilisons ensuite les outils issus de la topologie combinatoire afin de caractériser la calculabilité des modèles par le biais de *tâches affines:* des complexes simpliciaux extraits d'itérations finies de la sous-division colorée standard. Cette caractérisation s'applique aux modèles dits *sans-attente* avec accès à des objets de "k-test-and-set" ainsi qu'à un large ensemble de modèles d'adversaires en mémoire partagée dits *équitables.* Ces résultats généralisent et améliorent toutes les caractérisations topologiques connues de la capacité à résoudre des tâches pour les modèles communiquant par mémoire partagée.

Dans la seconde partie de la thèse, nous étudions la *complexité spatiale* de l'implémentation d'un *stockage fiable*, c.à.d., assurant qu'une valeur écrite en mémoire est persistante, dans le modèle à base de comparaison où seuls les identifiants peuvent être comparés. Nos résultats montrent l'existence d'un compromis non-trivial entre la complexité spatiale d'une implémentation et les garanties de vivacité qu'elle apporte.

Mots Clés : calculabilité distribuée; topologie combinatoire; mémoire partagée asynchrone; simulations distribuées; tâches affines;

Abstract

Modern computing systems are distributed, ranging from single-chip multi-processors to large-scale internet systems. In this thesis, we study computability and complexity issues raising in *asynchronous crash-prone shared memory* systems.

The major part of this thesis is devoted to characterizing the power of a shared memory model to solve *distributed tasks*. Our first contribution is a refined and extended *agreement-based* simulation technique that allows us to reason about the relative task computability of shared-memory models. Using this simulation technique, we show that the task computability of a shared-memory adversarial model is grasped by its ability to solve specific agreement tasks. We then use the language of combinatorial topology to characterize the task computability of shared-memory models via *affine tasks*: sub-complexes of a finite iteration of the standard chromatic subdivision. Our characterization applies to the wait-free model enhanced with k-test-and-set objects and a to large class of *fair* adversarial models. These results generalize and improve all previously derived topological characterizations of the task computability power of shared memory models.

In the second part of the thesis, we focus on *space complexity* of implementing *stable storage*, i.e., ensuring that written values persists in memory, in the *comparison-based* model using multi-writer registers. Our results exhibit a non-trivial tradeoff between space complexity of stable-storage implementations and the progress guarantees they provide.

Keywords: distributed computability; combinatorial topology; asynchronous shared memory; distributed simulations; affine tasks;

Contents

Chapter 1

Introduction

In the old days, computers were *sequential systems* isolated from the world. The questions of understanding what and how efficiently a sequential system can compute were the major theoretical challenges in the field of computing. In *computability*, probably the most fundamental result was the equivalence between computing systems and Turing machines, high-level abstract models of computation that can be used instead of low-level real computing systems. Within the Turing universal models, the limits of sequential computability were better understood, starting from the original "halting" impossibility [Tur37].

Since then, computing systems evolved and became more elaborate. Nowadays, even the simplest computing device is no longer a sequential machine. It possesses multiple processors which communicate with each other. While each processor and its private memory forms a sequential system equivalent to a Turing machine, the global system is much more sophisticated. Modern computing systems are no longer isolated from each other and are almost always connected through networks, allowing them to interact with each other. This evolution led to the rise of the field of *distributed computing* in which multiple sequential processes evolve in a joint distributed system.

In a distributed setting, the notion of a sequential function itself is replaced by a *distributed task*. In a task, each process possesses a private input value and must output a value at the end of the computation. The validity of an output value depends not only on all inputs but also on the outputs of other processes. Indeed, given an *input vector* associating an input to each (participating) process, the task specifies which *output vectors* are valid. The most well-known example of a task is the *consensus* task [FLP85] where processes must agree on one of the processes task input.

Distributed systems are very diverse. They can differ in communication abstractions (e.g., *message-passing* channels, *shared-memory*), timing assumptions (e.g., *synchrony*, *partial synchrony*, or *asynchrony*) or the type or number of possible failures (e.g., *crash* or *Byzantine*). Perhaps one of the most influential results in distributed computability was the proof that the task of consensus is not solvable in a *crash-prone asynchronous message-passing* system [FLP85], extended later to the shared-memory context [LAA87]. Hence, not only complexity but computability may vary across different distributed systems.

Hence, distributed computability deserves its own computability theory, distinct from Turing computability. One of the main focuses in distributed computing is, therefore, the search for a unified computability theory applicable to all models of distributed computing.

1.1 Distributed Computability in Shared-Memory Models

In this thesis, we focus on *shared-memory* models, inspired by modern multicore machines. The main challenge in this domain is to harness failures and the lack of synchrony.

While a processor rarely crashes *per se*, it is quite common that a *process* (also called a thread) running on it is interrupted, either due to the rotation of resource allocation or due to remote memory accesses. In computing cycles, such an interruption feels like an eternity. Hence, it is reasonable to model interruptions or failures due to programming errors as a *crash* failure: a crashed process behaves according to specification until it prematurely stops taking any more steps.

Moreover, ensuring a precise timing of operations is very costly and is only done in highly critical systems. In general-purpose machines, the duration of a step of computation may vary. This leads to the *asynchronous* model of computation where there are no bounds on the relative process speeds or communications delays. Therefore, understanding the computational power of an asynchronous crash-prone shared-memory system is paramount.

The communication abstractions available to the processes play a crucial role in understanding the system's computational power. At a low-level, computing systems are often provided with *registers* which can be used to store words that can be accessed for reading. But guarantees provided by the registers may be very weak when they are accessed by different processes *concurrently*, i.e., their executions overlap in time. Many efforts have been devoted to show that all shared-memory communication abstractions are equivalent and thus can be used interchangeably [Lam86, AAD+93, BG93b].

It turned out that stronger and more convenient primitives, such as *atomic snapshots objects* [AAD+93] and *immediate snapshots* [BG93b], are *computationally equivalent* to the weakest form of *safe* read-write communication.

In the *atomic-snapshot* (AS) model of communication, processes can perform update operations on individual memory locations and take snapshots of the whole memory *atomically*, that is as if the operation took place instantaneously. The *immediate snapshot* model ensures, additionally, that updates and snapshots performed by the processes are *coupled*: the execution can be represented as a sequence of *batches*, where each batch is a sequence of updates performed by a set of processes immediately followed by a matching set of snapshots operations.

In the following, the asynchronous read-write model with no restrictions on when and where failure might occur will be called *wait-free* [Her91]. Intuitively, in an algorithm designed for the wait-free model, a process is expected to make progress *in its own steps*, without waiting for other processes to move.

After many fundamental tasks were found to be impossible to solve in the wait-free model [BG93a, HS99, SZ00], it became appealing to explore its generalizations. Delporte et al. [DGFGT11] introduced the notion of an *adversary*, a collection $\mathcal{A}$ of process subsets, called *live sets*. Intuitively, adversaries allow for abstracting out models in which processes failures might occur in the non-independent and non-identical manner. A run is in the corresponding (read-write) $\mathcal{A}$-*adversarial model* if the set of processes taking infinitely many (read-write) steps in it is in $\mathcal{A}$. Others generalizations proposed, instead of restricting possible executions, to provide the processes with additionnal communications primitives such as *test-and-set* or *compare-and-swap* objects commonly available in processing units.

In this thesis, we evaluate computability of a shared-memory model via the set of tasks it allows to solve, called the model's *task computability*. Our major tools are *simulations* and *combinatorial topology*.

1.2 Simulations and Iterated Models

Faced with the huge variety of models in distributed computing, we can expect model simulations to be extremely useful. Suppose that we are provided with an algorithm that solves a task T in a model M_1. If there is a way to *simulate* M_1 in a different model M_2, we can obtain an algorithm solving T in M_2. Therefore, task computability of different models can be related via simulations. In particular, if M_1 can be simulated in M_2, then M_2 is at least as powerful as M_1, i.e., it can solve any task solvable in M_1.

One of the most influential simulation algorithms designed for this purpose is the *BG simulation* (for Borowsky and Gafni) [BG93a], recently awarded the Dijkstra Prize in distributed computing. BG simulation was first used to relate *colorless* task solvability (particular tasks in which processes can adopt inputs and outputs from each other) of the $t+1$-process wait-free model and the n-process t-*resilient* model, for any $n \geq t+1$, in which at most t processes may fail by crashing. Among numerous applications of this result, we should mention the fundamental impossibility of k-resilient k-set consensus, a strict generalization of the 1-resilient consensus impossibility [FLP85, LAA87]. BG simulation was latter refined to tackle all type of (colored) tasks [Gaf09] or to dynamically reduce synchrony of a system [FGRR14].

Furthermore, simulations allowed us to relate conventional shared-memory models and their *iterated* counterparts. In an iterated model, the processes proceed in (asynchronous) rounds, where each round is associated with a distinct memory. Iterated models have been introduced as simpler forms of communication model, providing one-shot communication-closed primitives accessed in a elegant and simple recursive structure. A prominent example of an iterated model is the *iterated immediate snapshot* (IIS) model, where each round is associated with a distinct (one-shot) immediate snapshot memory. Via simulations, it was shown that the IIS model is equivalent to the wait-free model in terms of task computability [BG97, GR10]. As we will see, the IIS model allows for an elegant combinatorial representation.

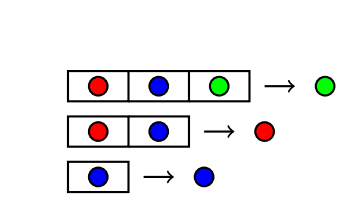
1.3 Distributed Computing through Combinatorial Topology

The major difficulty in understanding distributed computations is the explosion of possible permutations of steps of concurrent processes that could be observed in an execution of a given model. It is therefore natural to use the combinatorial approach. We model sets of such permutations allowed by the model as a combinatorial structure, and the power of the model can be induced from the abstract properties of the structure. Below, we briefly overview the basic elements of this approach, proposed and developed by Herlihy et al. [HS99, HKR14].

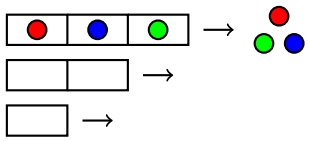
A *simplex* is a generalization of an edge from a graph including arbitrarily many vertices. It can be used to express the state of a distributed system where each vertex represents the state of a given process.

A *simplicial complex* is an inclusion-closed set of simplices. Using simplicial complexes to represent possible states of a distributed system is very convenient as states indistinguishable for a set of processes Q must share the same vertices corresponding to processes from Q. Moreover, the sets of possible input and output vectors of a task also form simplicial complexes, i.e., the input and output simplicial complexes, while the specification of the task defines a *carrier map*.

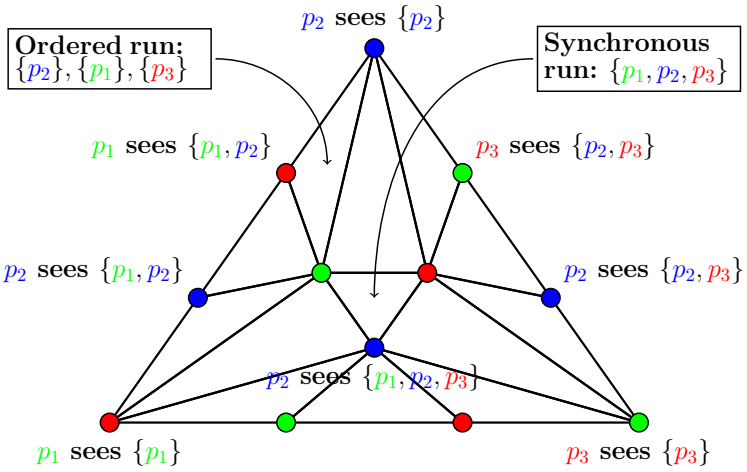
It turns out that the simplicial complex corresponding to all runs of one round of immediate snapshot (two particular IS runs are depicted in Figures 1.1a and 1.1b), in which the processes perform updates to the memory with their input value and takes an immediate snapshot, is



(a) Ordered run: $\{p_2\}, \{p_1\}, \{p_3\}$.



(b) Synchronous run: $\{p_1, p_2, p_3\}$.



(c) Chr(**s**), the standard chromatic subdivision of a 2-simplex, the output complex of the 3-process *IS* task.

Figure 1.1 – Example of IS runs on the left and the set of all IS runs represented as a simplicial complex on the right.

precisely captured by the *standard chromatic subdivision* (denoted Chr) (Figure 1.1c) [BG97, HS99, Koz12]. Moreover, the iterated immediate snapshot (IIS) model, obtained by performing immediate snapshots iteratively, where the current iteration output is used as the input value for the next one, is precisely captured by iterations of Chr.

Topology of wait-freedom. Recall that the wait-free model makes no assumptions on when and where failures might occur. Herlihy and Shavit proposed an elegant characterization of wait-free (read-write) task computability via the existence of a specific *simplicial* map from geometrical structures describing inputs and outputs [HS99].

A task T has a wait-free solution using read-write registers if and only if there exists a simplicial, chromatic map from some *subdivision* of the *input simplicial complex*, describing the inputs of T, to the *output simplicial complex*, describing the outputs of T, which respect the task specification Δ. In particular, we can choose this subdivision to be a number of iterations of the standard chromatic subdivision Chr.

Therefore, the celebrated *Asynchronous Computability Theorem (ACT)* [HS99] can be formulated as:

> A task $T = (\mathcal{I}, \mathcal{O}, \Delta)$, where $\mathcal{I}$ is the input complex, $\mathcal{O}$ is the output complex, and Δ is a carrier map from $\mathcal{I}$ to sub-complexes of $\mathcal{O}$, is wait-free solvable if and only if there exists a natural number ℓ and a carrier and color preserving simplicial map $\phi : \mathrm{Chr}^{\ell}(\mathcal{I}) \to \mathcal{O}$ carried by Δ.

The ACT theorem can thus be interpreted as: the set of wait-free (read-write) solvable task is precisely the set of tasks solvable in the IIS model. The ability of (iteratively) solving the *IS* task thus allows us to solve any task in the wait-free model. Hence, from the task computability perspective, the *IS* task is a finite representation of the wait-free model.

Results derived from topology. This characterization of wait-free solvability was introduced to show the impossibility to solve the k-set consensus task [Cha93]. The k-set consensus task is a generalization of the consensus task in which processes are allowed to return at most k distinct outputs instead of only one. Attempts to prove its impossibility when $k < n$ using traditional FLP-like proof schemes were unsuccessful. But using characterizations of wait-free task solvability through combinatorial models such as the IIS model, the impossibility of k-set consensus was straightforward to show [SZ93, BG93a, HS93]. While demonstrated by three groups independently from each other, all took a similar approach. It consists in reducing a solution of the k-set consensus task to a Sperner coloring, which according to Sperner's Lemma [Spe28], must color an odd number of maximal simplices of a subdivision using all the colors. Hence, there is always an execution for which the output vector contains as many distinct values as in the input vector.

The topological characterization of wait-free solvability has been also used to show other prominent results in distributed computability. Traditional proof schemes also failed to understand the computability of the renaming tasks. In the M-renaming task [ABND$^+$90], processes have arbitrary distinct input values from an arbitrary large domain and must produce different outputs in the range $\{1, \ldots, M\}$. Finding the smallest M for which the task is wait-free solvable turned out to be a very complicated issue. By using the topological characterization of wait-free solvability, it was shown by Casteñada and Rajsbaum [CR10, CR12] that the best renaming possible is for $M = 2n - 1$ when n is a prime power but that $(2n - 2)$-renaming is possible when n is not a prime power.

1.4 Contributions

This thesis primarly focuses on the study of task computability in shared-memory models. This subjects is approached in three complementary ways: (1) improving simulation techniques in order to provide finer reductions between models; (2) measuring task computability of models in order to provide a classification of their relative task computability; and (3) providing a canonical representation of models using combinatorial topology in order to simplify task solvability characterization.

As a complementary result, this thesis also studies complexity issues of implementing stable storage in the comparison-based model. In particular, it exhibits the existence of a trade-off between space complexity and liveness guarantees.

Agreement-Based Distributed Simulation

The first contribution of this thesis concerns the area of *distributed simulations*. Here we build on two classic tools: BG simulation and the universal construction. The *BG-simulation* technique leverages resilience of a model in order to *simulate* steps of other distributed systems with a smaller concurrency level. The *universal construction* by Herlihy [Her91] uses consensus objects to simulate any distributed object in a wait-free manner by simply removing concurrency issues.

These two approaches led to the proposal of an *agreement-based* simulation technique, informally sketched by Gafni and Guerraoui in a technical report [GG09]. The idea consists in using the ability of processes to solve set-consensus tasks to provide a limited concurrency.

This thesis provides the first formalization of an *agreement-based* distributed simulation. Compared to [GG09], we present here a direct and simplified simulation scheme which,

additionnaly, is made *adaptive*. Indeed, in [GG09], the simulation is described by combining multiple constructs such as generalized state machines replication [GG11] and extended BG-simulation [Gaf09]. By refining the base constructs of the simulation such as *commit-adopt* [Gaf98] and *safe-agreement* [BG93a, BGLR01], we are able to construct a much simpler and direct simulation. By also making the simulation adaptive, i.e., automatically adjusting its synchronization power to the global state of the simulating system, the design of simulations is made much simpler by removing the potential need of additionnal external mechanisms.

Measuring and Classifying the Task Computability of Models

A conventional way to capture the power of a model is to determine its task computability, i.e., the set of distributed tasks that can be solved in it. But task solvability has been shown to be an undecidable problem [GK99, HR97]. This is why we focus on trying to measure the *relative power* of models with respect to each other. It has been shown that shared memory models can be classified by their ability to solve set-consensus tasks as long as this concerns a limited class of *colorless* tasks solvability [HR12].

In this thesis we try to provide a classification of shared-memory computability for *all* distributed tasks. For this, we introduce the notion of an *agreement function*. An agreement function stipulates, given a shared-memory model M and sets of processes Q and P, what is the best k-set-consensus task solvable in M between processes from Q when only processes in P might participate. We compute this function for different kinds of models, such as shared-memory adversaries and the wait-free model enhanced with a collection of set-consensus objects. We show that for these models, agreement functions capture their ability of solving tasks, i.e., they are weaker as regards task solvability than any (*long-lived*) model with a smaller or equal agreement function.

This provides some understanding of the classification of models by their relative computability power. Moreover, the study of natural properties of agreement functions such as *fairness*, when they only depends on participation, *locality*, when they only depends on the targeted subset, or *symmetry*, when they only depends on sizes of sets P, Q and $P \cap Q$, provides some insights on the computational power of the corresponding models. In particular, we show that a large class of shared-memory adversaries, including but not restricted to super-set closed and symmetric adversaries, have *fair* agreement functions and that the models defined through collections of (potentially asymmetric) set-consensus objects have *local* agreement functions.

Models as Iterated Affine Tasks

The characterization of the wait-free model of computation through the IIS model and the standard chromatic subdivision [HS99, BG97], awarded the Gödel prize in theoretical computer science, is a core result of the area of distributed computing. Indeed, this characterization using combinatorial topology allowed to grasp the intrinsic properties of wait-free solvability using *compact* finite objects. Given that many fundamental tasks are not solvable in the wait-free way [BG93a, HS99, SZ00], more general models, such as shared-memory adversaries, were considered. But, can the task computability of all shared memory models be characterized using similar combinatorial representations?

In this thesis, we follow the proposal of Gafni et al. in [GKM14] to characterize models using *affine tasks*. An affine task is a pure sub-complex of a finite number of iterations of the

standard chromatic subdivision. We show that the computational power of many classical models can be grasped by iterations of affine tasks. In particular we construct affine tasks corresponding to the wait-free model enhanced with k-test-and-set objects and to the large class of *fair* adversarial models.

This result generalizes all existing topological characterizations of distributed computing models [HS99, GKM14, SHG16], as it applies to *all* previously considered shared-memory models and *all* tasks (and not only *colorless* tasks). We believe that the results can be extended to all "practical" restrictions of the wait-free model and to all enhancements of the wait-free model with distributed objects, which may result in a complete computability theory for distributed shared-memory models.

Complexity of SWMR Implementations

Most shared-memory distributed systems assume the single-writer multi-reader (SWMR) setting, where each process is provided with a unique register that can only be written by the process and read by all. This provides processes with a *stable storage* mechanism, providing the ability of sharing knowledge persistently. But in the *comparison-based* model, where computation is performed by a bounded number n of processes coming from a large scale system composed of a high number (possibly unbounded) of potential participants, the assumption of an SWMR memory is no longer reasonable. The natural question that arise is how many multi-writer multi-reader (MWMR) registers are required to implement a stable storage, or equivalently, an SWMR memory.

In this thesis, we show that the answer depends on the desired progress condition. We propose an SWMR implementation that adapts the number of MWMR registers used to the desired progress condition. For any given k from 1 to n, we present an algorithm that uses $n + k - 1$ registers to implement a *k-lock-free* SWMR memory, i.e., in which at least k processes can concurrently perform operations. This generalizes previous state of the art implementations [DFGR13], providing either lock-freedom with n MWMR registers, or wait-freedom with $2n - 1$ MWMR registers.

More interestingly, using novel covering-based arguments, we show that any 2-*obstruction-free* algorithm requires $n + 1$ base MWMR registers. Previously, only a trivial lower bound of n registers was shown for lock-free implementations [DFGR13]. An interesting implication of our results is that 2-lock-free and 2-obstruction-free SWMR implementations have the same optimal space complexity. Curiously, our results highlight a contrast between complexity and computability, as we know that certain problems, e.g., consensus, can be solved in an obstruction-free way, but not in a lock-free way [HLM03].

1.5 Organization

The thesis begins with an introductory chapter recalling classical definitions about distributed computing. We then proceed with two technical chapters corresponding with our contributions concerning simulations, reductions, and classification of shared-memory models. Afterwards, a second introductory chapter focusing on combinatorial topology constructs used in distributed computing is presented. It allows us to continue with our main contribution concerning combinatorial representation of the task computability of broad classes of shared-memory models. Our complexity results are then presented in the last technical chapter before concluding the thesis.

Chapter 2: Preliminaries. The first chapter is a brief introduction to distributed computing. It first recalls the basic formalism of distributed systems. Then, more details are given about definitions of shared-memory based distributed systems models. We also focus on the crucial notion of *distributed tasks* and their solvability. We terminate with constructs concerning the iterated immediate snapshot model and its equivalence with shared memory.

Chapter 3: Distributed Simulations. The first technical chapter concerns the distributed simulation tools. A detailed review of the classical, resilience-based, *BG-simulation* technique is provided. Building on this simple simulation tool, we then provides additionnal features such as *abort* mechanisms and *conflict detection*. This yield in more complex of powerful simulation tools similar to the *extended* BG-simulation technique introduced by Gafni [Gaf09]. We terminate with the replacement of the resilience-based synchronization by an adaptive agreement-based technique, yielding in a more widely applicable, yet simple to use, simulation.

Chapter 4: Agreement Functions. The second technical chapter revolves around the notion of *agreement functions*, a measure of models computability power. Agreement functions are introduced along with a derived natural classifaction of models. The importance of agreement functions to measure task computability is then justified by their ability to characterize task solvability of classical shared-memory models. In particular, it is shown that all shared memory adversaries and wait-free systems enhanced by a collection of set-consensus objects are characterized by their agreement function. Moreover, we show that many adversaries, including super-set closed and symmetric adversary families, have nice *fair* agreement functions.

Chapter 5: Combinatorial Topology. The second introductory chapter focuses on the notions from combinatorial topology used in distributed computing. Through a detailed description of the celebrated *asynchronous computability theorem* [HS99], tools of combinatorial topology specifically relevant for distributed computing are introduced. The notions presented in this chapter will provide essential concepts used in the following and the concluding chapters.

Chapter 6: Affine Tasks. The third technical chapter contains the main contribution of this thesis. It starts with a formal introduction of *affine tasks* and shows general results about them. As a simple illustration, it is shown how the wait-free model enhanced with *k-test-and-set* objects can be captured by a small affine task. Afterwards, affine tasks for the k-set consensus model are constructed as a step torwards defining affine tasks for the most complex fair adversarial models.

Chapter 7: SWMR Memory Implementation. This last technical chapter is dedicated to our complexity analysis of SWMR memory implementation in the *comparison-based* model. We first motivate the study of trade-offs between space complexity and progress conditions in this context. Then, we present a generalization of existing SWMR algorithms for refined progress conditions. Afterwards, we present our main contribution, a lower bound showing a necessary trade-off between space complexity and the desired progress condition. Lastly, we discuss various partial results as well as open related questions and problems.

Chapter 8: Conclusion and Open Questions. The last chapter summarizes the results of this thesis and discuss questions left open as well as possible future research directions.

1.6 Publications

The main results of this thesis appeared originally in the following papers (in the chronological order):

1. Eli Gafni, Yuan He, Petr Kuznetsov, and Thibault Rieutord. Read-Write Memory and k-Set Consensus as an Affine Task. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, volume 70 of *LIPIcs*, pages 6:1–6:17, 2016.
2. Petr Kuznetsov and Thibault Rieutord. Agreement Functions for Distributed Computing Models. In *Networked Systems (NETYS)*, volume 10299 of *LNCS*, pages 175–190, 2017.
3. Petr Kuznetsov, Thibault Rieutord, and Yuan He. Brief Announcement: Compact Topology of Shared-Memory Adversaries. In *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *LIPIcs*, pages 56:1–56:4, 2017.
4. Damien Imbs, Petr Kuznetsov, and Thibault Rieutord. Progress-Space Tradeoffs in Single-Writer Memory Implementations. In *21st International Conference on Principles of Distributed Systems (OPODIS 2017)*, volume 95 of *LIPIcs*, pages 9:1–9:17, 2017.
5. Petr Kuznetsov, Thibault Rieutord, and Yuan He. An Asynchronous Computability Theorem for Fair Adversaries. In *37th ACM Symposium on Principles of Distributed Computing (PODC 2018)*, pages 387–396, 2018. Co-laureate of the best student paper award.

In parallel with working on the thesis, the author was involved in other projects, which resulted in the following papers:

1. Pierre Fraigniaud, Sergio Rajsbaum, Corentin Travers, Petr Kuznetsov, and Thibault Rieutord. Perfect Failure Detection with Very Few Bits. In *18th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2016)*, pages 154–169, 2016.
2. Denis Jeanneau, Thibault Rieutord, Luciana Arantes, and Pierre Sens. Solving k-Set Agreement Using Failure Detectors in Unknown Dynamic Networks. In *IEEE Trans. Parallel Distrib. Syst.*, volume 28, number 5, pages 1484–1499, 2017.

Chapter 2

Preliminaries

This chapter briefly recalls classical notions and results in distributed computing that will be used in the technical chapters of this thesis.

2.1 Basic Notions

Throughout the thesis, we only consider finite distributed systems. A system is composed of a set of n processes, denoted Π, and we assume by default that the set is known to the processes.

States, configurations and executions. An *algorithm* (we also sometimes say a *protocol*) assigned to each process is a (possibly non-deterministic) automaton that may accept high-level operation requests as an application input. In each state, the process is poised to perform a *step*, e.g., a read or write operations on base registers. Once the step is performed, the process changes its state according to the result of the step operation, possibly non-deterministically and possibly to a state corresponding to another high-level operation.

A *configuration*, or system state, consists of the states of all processes and the content of all shared objects. In the *initial* configurations, all processes and all objects are in their initial states.

We say that a step e by a process p is *applicable* to a configuration C, if e is the pending step of p in C, and we denote Ce the configuration reached from C after p performed e. A sequence of steps $e_1, e_2, \ldots$ is applicable to C, if e_1 is applicable to C, e_2 is applicable to Ce_1, etc. A (possibly infinite) sequence of steps applicable to a configuration C is called an *execution from* C. A configuration C is said to be *reachable* from a configuration C', and denoted $C \in \mathit{Reach}(C')$, if there exists a finite execution ζ applicable to C', such that $C = C'\zeta$. If omitted, the starting configuration is the initial configuration, and is denoted as $C \in \mathit{Reach}$. Note that an execution can also be called a *run* of the system.

A process is called *correct* in a given (infinite) execution if it takes infinitely many steps in that execution. A process is called *faulty* if it takes only finitely many steps in an execution. Let $\mathit{Correct}(\zeta)$, respectively $\mathit{Faulty}(\zeta)$, denote the set of correct, respectively faulty, processes in the execution ζ.

Atomic snapshot memory. The atomic-snapshot (AS) memory is represented as a vector of n shared variables, where each process p_i is associated with the position i. The memory

can be accessed with two operations: *update* and *snapshot*. An *update* operation performed by p_i modifies the value at position i and a *snapshot* returns the vector current state. The atomic-snapshot memory model is known to be computationally equivalent to the more realistic *read-write* shared memory model in which processes communicate by reading and writing to shared individual memory locations [Lam86, AAD+93].

To simplify further systems in which processes have only access to an AS memory, we consider the *full information* protocol. In the full information protocol, a process first shares its initial state using the *update* operation. Then, processes alternate between snapshot operations and update operations containing their complete current state. Thus, when running the full information protocol, an execution can be reduced to the sequence of processes identifiers performing the operations in it. If a process completed its first step in a given run, it is said to be *participating*, and the set of participating processes is called the *participating set*. Note that, in particular, every correct process is participating.

2.2 Shared-Memory Models

In an abstract sense, a (shared-memory) *model* can be seen as a combinations of a set of runs and infinitely many copies of every elements of a set of shared objects. For simplicity, we only considered models in which processes run the full information protocol on an AS memory and may proceed, in between steps, to operations on other available shared objects. To solve a problem in a model means to design an algorithm that satisfies the problem specification in every execution which belongs to the model. Below we review classes of models considered in this thesis and discuss their high-level properties.

Wait-free model. The simplest and least powerful shared-memory model that can be defined is such a way is the *wait-free* model [Her91]. In the wait-free model, processes have only access to an AS memory and no assumptions are made on the possible executions of the model, execept that there is a correct process. Hence, the wait-free model is simply the set of all infinite executions. The wait-free model is trivially the weakest possible shared-memory model as a problem must be solved in any possible infinite execution without using any shared objects besides the AS memory. Hence, any algorithm designed for the wait-free model can be executed correctly in any other shared-memory model.

Adversaries

The main class of models considered in this thesis are the *adversarial* models introduced by Delporte et al. [DGFGT11]. It corresponds to models defined purely on restrictions on the set of runs without access to any objects additionally to the AS memory. Several types of adversarial models exist in the distributed computing literature, such as message adversaries [Gaf98] or crash-failures adversaries considered here. Note that, in this thesis, we only focus on shared-memory crash-failures adversaries.

An adversary $\mathcal{A}$ is defined as a set of possible correct process subsets. Formally, $\mathcal{A}$ is a set of subsets of Π, called *live sets*, $\mathcal{A} \subseteq 2^{\Pi}$. An infinite run is *$\mathcal{A}$-compliant* if the set of processes that are correct in that run belongs to $\mathcal{A}$. An adversarial $\mathcal{A}$-model is then defined as the set of $\mathcal{A}$-compliant runs.

An adversary is *superset-closed* [Kuz12] if each superset of a live set of $\mathcal{A}$ is also an element of $\mathcal{A}$, i.e., if $\forall S \in \mathcal{A}$, $\forall S' \subseteq \Pi$, $S \subseteq S' \implies S' \in \mathcal{A}$. Superset-closed adversaries provide a

non-uniform generalization of the classical *t-resilient* model in which at most t processes may crash, corresponding to the adversary consisting of sets of $n-t$ or more processes.

An adversary $\mathcal{A}$ is a *symmetric* adversary if it does not depend on process identifiers: $\forall S \in \mathcal{A}$, $\forall S' \subseteq \Pi$, $|S'| = |S| \implies S' \in \mathcal{A}$. Symmetric adversaries provide another interesting generalization of the classical t-resilience condition and k-obstruction-free progress conditions [GG09], i.e., executions where there are at most k correct processes. Such sets of executions were previously formalized by Taubenfeld as the symmetric progress conditions [Tau10].

Set consensus power. The notion of *set consensus power* [GK10] was originally proposed to capture the power of adversaries in solving *colorless* tasks [BG93b, BGLR01], i.e., tasks that can be defined by relating *sets* of inputs and outputs, independently of process identifiers.

Definition 2.1. *The* set consensus power *of $\mathcal{A}$, denoted by $setcon(\mathcal{A})$, is defined as follows:*
- *If $\mathcal{A} = \emptyset$, then $setcon(\mathcal{A}) = 0$*
- *Otherwise, $setcon(\mathcal{A}) = \max_{S\in\mathcal{A}} \min_{a\in S} setcon(\mathcal{A}|_{S\setminus\{a\}}) + 1$.* [1]

Thus, for a non-empty adversary $\mathcal{A}$, $setcon(\mathcal{A})$ is determined as $setcon(\mathcal{A}|_{S\setminus\{a\}})+1$ where S is an element of $\mathcal{A}$ and a is a process in S that "max-minimize" $setcon(\mathcal{A}|_{S\setminus\{a\}})$. Note that for $\mathcal{A} \neq \emptyset$, $setcon(\mathcal{A}) \geq 1$. It is shown in [GK10] that $setcon(\mathcal{A})$ is the smallest k such that $\mathcal{A}$ can solve k-set consensus.

It was previously shown in [GK11] that for a superset-closed adversary $\mathcal{A}$, the set consensus power of $\mathcal{A}$ is equal to $csize(\mathcal{A})$, where $csize(\mathcal{A})$ denotes the minimal hitting set size of $\mathcal{A}$, i.e., a minimal subset of Π that intersects with each live set of $\mathcal{A}$. Therefore if $\mathcal{A}$ is superset-closed, then $setcon(\mathcal{A}) = csize(\mathcal{A})$. For a symmetric adversary $\mathcal{A}$, it can be easily derived from the definition of $setcon$ that $setcon(\mathcal{A}) = |\{k \in \{1,\ldots,n\} : \exists S \in \mathcal{A}, |S| = k\}|$.

Models of k-Active-Resilience and k-Concurrency

We now describe two models of particular interest, also defined purely on a restriction on the set of runs of an AS memory, but that cannot be directly defined as adversaries.

k-concurrency. Technically, the model of k-concurrency is not defined as a generic restriction on the set of valid runs but only for terminating executions in which processes eventually complete their executions. A process is called *active* at the end of a finite run R if it participates in R but did not returned at the end of R. Let $active(R)$ denote the set of all processes that are active at the end of R. Since only finite runs are considered, the notion of correct and faulty processes is no longer adequate. Instead we may split processes into *non-terminated* processes, i.e., active or not participating, and *terminated* processes.

A finite run R 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 [GG11] is then the set of (finite) full-information AS runs which are k-concurrent.

k-active-resilience. Similarly to adersarial models, a run R is *k-active-resilient* ($k = 0,\ldots,n-1$) if at most k participating processes take only finitely many steps in R. Note that the difference lies in limiting the number of failures among participating processes, while a

1. $\mathcal{A}|_P$ is the adversary consisting of all live sets of $\mathcal{A}$ that are subsets of P.

process that is not participating is also a faulty process. The *k-active-resilient* model is the set of (infinite) k-active-resilient full-information AS runs [Kuz13].

k-Test-and-Set Model

For an integer $k \geq 1$, a *k-test-and-set* object exports one operation, *apply*(), that may be only accessed once by each proces, takes no parameters, and returns a boolean value. It guarantees that at most k processes will get 1 as output and that not all processes accessing it obtained 0. In the special case of $k = 1$ the object is simply called *test-and-set*.

The *k-test-and-set model* is then simply defined as the wait-free model with, additionnally, access to any number of k-test-and-set objects. Hence processes run a full information protocol on an AS memory without any restrictions on the set of possible runs, but processes may proceed, between any operations on the AS memory, to operations on any number of k-test-and-set objects.

Set consensus collections

A k-set consensus object is a (nondeterministic) object that can be accessed with a single one-shot *propose*(v) operation ($v \in V$) that returns a proposed value and ensures that at most k distinct values are returned. When $k = 1$ then the object is deterministic and is simply called a *consensus* object [FLP85]. More generally, *(ℓ, k)-set-consensus objects* ($k \leq \ell$) are defined similarly but can be accessed by at most ℓ processes. Note that if more than ℓ processes access the object, then the object may return a special value $\perp$.

Let $\mathcal{C}$ be a set of pairs $\{(\ell_0, j_0), (\ell_1, j_1), \ldots, (\ell_m, j_m)\}$, where $\ell_i, j_i \in \mathbb{N}$. Then $\mathcal{C}$ can be used to define the *$\mathcal{C}$-set-consensus* model [DGFGK16] which correspond to the wait-free model with, additionnally, access to any number of (ℓ, j)-set-consensus objects such that $(\ell, j) \in C$.

2.3 Distributed Tasks

In order to study computability in distributed shared-memory models, a notion of *distributed problem* is required. Most distributed problems are either formulated as the implementability of a distributed object or the solvability of a distributed task. In this thesis, we focus on the notion of distributed task which captures (most of) terminating distributed problems.

Definition. The notion of distributed *task* was formally introduced in [HS99]. 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. We require that Δ is a *carrier* map: $\forall \rho, \sigma \in \mathcal{I}$, $\rho \subseteq \sigma$: $\Delta(\rho) \subseteq \Delta(\sigma)$. Informally, a carrier map ensures that if a set that is outputs is valid for some inputs, then a superset of these outputs are valid for a superset of the inputs. Look at Section 5.2 or to [HKR14] for more details about carrier maps. An input value equal to $\perp$ denotes a *non-participating* process, and an output value equal to $\perp$ indicates a process that is *undecided*.

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

Agreement tasks. A classical family of tasks in distributed computing are *agreement tasks*. In an agreement task, processes are given values from some set V and must output values which are a subset of the inputs values. Additionnal requirements on the relations between inputs and outputs are then added for specific agreement tasks.

In the *k-set consensus* task [Cha90], input values are in a set of values V ($|V| \geq k+1$), output values are in V, and for each input vector I and output vector O, $(I, O) \in \Delta$ if the set of non-$\perp$ values in O is a subset of values in I of size at most k. The case of 1-set consensus is called *consensus* [FLP85].

Renaming task. Another classical task is the *renaming* task [ABND$^+$90]. The idea is that in many distributed systems, processes are provided with unique identifiers but from a vast set of possible names. But names are very often used by processes in their communications to identify the messenger for example. Therefore, the size of identifiers may have a significant impact on complexity issues. A natural question is thus whether it is possible to provide processes with a new name from a much smaller namespace while preserving the unicity of identifiers.

Formally, in the task of *M-renaming*, for $M \in \mathbb{N}$, inputs come from a set of comparable values V such that $\mathcal{I}$ is the set of vectors composed of n distinct values from V. Similarly, the set of valid output vectors are sets of n distinct values from $\{1, \ldots, m\}$. The task specification is trivial as any input vector from $\mathcal{I}$ can be mapped to any output vector of $\mathcal{O}$.

A variant of renaming is called *adaptive* f-renaming, where the space of output values must scale with the number of participating processes. When p processes are participating in the task of adaptive f-renaming, then the outputs should be distinct elements from $\{1, \ldots, f(p)\}$.

2.4 IIS Model

Let us introduce the iterated immediate snapshot model formally and recall its proof of equivalence with the more traditional AS wait-free model. Indeed, elements of this equivalence results will be later used in Chapter 6 to derive a combinatorial characterization of other shared-memory models.

Iterated immediate snapshot model. In the iterated immediate snapshot (IIS) model, processes proceed through an infinite sequence of independent memories $M_1, M_2, \ldots$. Each memory M_r is accessed by a process with a single *WriteSnapshot* operation [BG93b]: the operation performed by p_i takes a value v_{ir} and returns a set V_{ir} of submitted values (w.l.o.g, values of different processes are distinct), satisfying the following properties (See Figure 2.1 for IS examples):

— self-inclusion: $v_{ir} \in V_{ir}$;
— containment: $(V_{ir} \subseteq V_{jr}) \vee (V_{jr} \subseteq V_{ir})$;
— immediacy: $v_{ir} \in V_{jr} \Rightarrow V_{ir} \subseteq V_{jr}$.

(a) Ordered run: $\{p_2\}, \{p_1\}, \{p_3\}$.

(b) Synchronous run: $\{p_1, p_2, p_3\}$.

Figure 2.1 – Examples of valid sets of IS outputs.

Then the IIS model runs a full information protocol on this sequence of independent memories. The first memory M_1 is accessed with the initial state as argument to the *WriteSnapshot* operation , and afterwards, each memory with the output of the *WriteSnapshot* operation on the preceding memory. Note that in the IIS model there are no process failures, each process complete infinitely many operations. Instead, the notion of *fast* and *slow* processes may be used. A process is said to be *fast* if it is observed in all processes *WriteSnapshot* operations outputs for infinitely many memories. A process is then simply said to be *slow* if it is not *fast*.

Such models with independent memories are often denominated as *communication-closed* as processes can use only once each object. Each memory object is thus simple to reason about as it may be impacted only with a finite number of operations. Moreover, each instance can be seen as a task and the iterated model as an infinite sequence of identical tasks, each time using the output of a given iteration as input for the next one.

Solving the IS Task

Showing that the IIS model and the wait-free model solves the same set of tasks can be done through simulations. Simulating the IIS model in a wait-free model is very convenient as it consists in iteratively solving the IS task. Indeed, since memories in the IIS model are independent, each object can be seen as a task to be solved. Moreover, in the wait-free model, any task can be solved iteratively as any infinite sub-sequence of a wait-free execution is a wait-free execution. This last aspect will be discussed in more details in Chapters 4 and 6 dealing with iterated solutions in more complex models.

Algorithm's description. A level-based implementation of an immediate snapshot works as follows [BG93b]: (1) processes write their input in memory associated with a level ℓ starting at $\ell = n$; (2) they perform a snapshot of memory; (3) if the snapshot contains ℓ values associated with a level $\ell' \leq \ell$ then the process returns with the snapshot consisting of these ℓ values, otherwise processes re-iterate the procedure using the level $\ell - 1$. The formal description can be found in Algorithm 2.1. Figure 2.1 present a classical representation of both outputs of an IS task and its level-based solution as the memories of decreasing sizes correspond to the update operations of deacreasing levels. Indeed, as we will show, the bound on the number of processes reaching the successive levels are bounded in any possible executions.

Algorithm's proof. In Chapter 6, we will present a variation on this algorithm. Since large part of its proof will be inhereted from this algorithm's proof of correctness, let us recall in detail its proof. We first show that the number of processes which may reach the successive levels is decreasing before showing that this property can be used to show its correctness.

Algorithm 2.1: Level-based immediate snapshot implementation for p_i.

1 **Shared Objects**: $MEM[1 \dots n] \in Val \times \mathbb{N})$, **initially** $(\perp, \perp)$;
2 **Init**: $level = n + 1$, $value = InputValue$, $snap = \emptyset$;

3 **Do**
4 $\quad level = level - 1$;
5 $\quad MEM[i].Update(value, level)$;
6 $\quad snap = MEM.Snapshot()$;
7 **While** $|\{(v, \ell) \in snap, \ell \leq level\}| \neq level$;
8 **Return** $\{(v, \ell) \in snap, \ell \leq level\}$;

Lemma 2.1. *In an execution of Algorithm 2.1, at most ℓ processes can reach the $(n - \ell)^{th}$ iteration of the while loop.*

Proof. Let us show this result by induction on the number of iterations k. If $k = 1$, then at most n processes can reach the first iteration of the while loop as there is n processes in the system. Assume that the lemma holds for iteration k, then at most $n - k$ processes may reach iteration k of the while loop. If there is strictly less than k iterations or if a process crashes during iteration k, then the lemma property holds for iteration $k + 1$. Assume that exactly $n - k + 1$ processes reach iteration k and complete this iteration of the while loop and consider the process p which takes last its iteration snapshot. As all $n - k + 1$ processes must have previously updated their register with a value which is associated with level $n - k + 1$ and which may have been replaced only with the same input value associated with a smaller level value, and as at most $n - k + 1$ have updated their register with an associated value smaller or equal to $n - k + 1$ (i.e., have access the k^{th} iteration of the while loop), then p obtains a snapshot containing exactly $n - k + 1$ values associated with a level smaller or equal to $n - k + 1$ and thus it exits the while loop. Therefore the lemma property holds for level $k + 1$ which completes the induction proof. □

Theorem 2.1. *Algorithm 2.1 implements an immediate snapshot operation.*

Proof. According to Lemma 2.1, every correct process eventually exit the while loop and thus returns a set of input values. The immediate snapshot properties are as follows:

— *self-containment :* A process returns a set of input values including its own. Indeed, a process returning at level ℓ (i.e., with a variable *level* currently set to ℓ), returns all input values associated with a level smaller or equal to ℓ, thus its own, as its last update on its register includes its input value associated with the level ℓ.

— *containment :* The set of values V and V', returned by two processes p and p' respectively, are such that either $V \subseteq V'$ or $V' \subseteq V$. Consider that p returns at level ℓ and p' returns at level ℓ', w.l.o.g., with $\ell' \leq \ell$. Assume that there is an input value from a process p'' returned by p' but not by p. p'' must have reached level ℓ' and thus has reached level ℓ, but as its input value is not included by p, p' must have updated its register in the iteration corresponding to level ℓ after p proceeded to its snapshot for the same level. But p must have observed ℓ processes which already accessed the iteration corresponding to level ℓ to exit the loop, as it does not include p'' this implies that strictly more than ℓ processes accessed to this iteration, a contradiction with the property of Lemma 2.1.

— *Immediacy :* Given two processes p and p' returning the set of values V and V' respectively, if V includes the input value from p' then $V' \subseteq V$. To see that the set returned by Algorithm 2.1 verifies this property, assume that this is not the case. By the inclusion property we have $V \subsetneq V'$. Thus, with ℓ and ℓ' the levels at which p and p' returned respectively, we have $\ell = |V| < |V'| = \ell'$. This implies that p' only updated its register with an associated level greater or equal to ℓ', and so p' input value cannot have been returned by p as only values associated with a level smaller or equal to ℓ are returned — a contradiction.

□

Simulating AS Memory Operations in the IIS Model

Simulating the wait-free model in the IIS model is more complicated than solving the IS task. Indeed, we must ensure that persistent operations are simulated on top of a communication-closed abstraction. Therefore, a *slow* process cannot terminate its simulated update operations as long as fast processes might still perform snapshot operations which should return the last completed update operations of all processes. But since there are no failures in the IIS model, to solve a task, all processes must produce valid outputs.

The solution to this issues consists on ensuring that some non-terminated process make progress with its simulation, i.e., completing an unbounded number of AS memory operations. Hence, when executing a task solution, this process will eventually terminate as correct processes must return with a task output. Thus, as long as there are non-terminated processes, one non-terminated process will obtain a task output and terminate its simulation, allowing slower processes also to terminate until all processes get a task output.

Algorithm's description. We present, in Algorithm 2.2, the simulation proposed in [GR10] which simplifies the original simulation proposed in [BG97]. Another simulation was proposed in [BGK14] which ensures that all fast processes make progress with the simulation. But as any execution is a run of the wait-free model, guaranteeing that at least one non-terminated process makes progress is sufficient.

The algorithm is relatively simple. Processes maintain in a local array of n variables, *SIM_MEM*, containing the value of the most recent known simulated write operation for each process. Processes also keep an array of n integers *SIM_counters*, associating write operations in *SIM_MEM* with an operation counter. At each round of the IIS model, processes share as input to the immediate snapshot all known pending or completed update operations along with the matching operations counter. The immediate snapshot outputs are then used to update *SIM_MEM* and *SIM_counters* with the operations associated with the highest counters.

Afterwards, if the sum of all operations counters is equal to the IIS round number, the update operation is completed along with the following simulated snapshot operation by using the values from the *SIM_MEM* array. If the process is still *undecided*, then it simulates a new update operation. Otherwise, the process stops increasing its write operation counter and just use a special value $\perp$ instead of its pending update operation value.

Sketch of simulation's correctness. This algorithm will not be modified, consequently, only a rough sketch of its correctness is given. Refer to [GR10] for more details. [2]

2. Note that there are some issues in the proofs and algorithm from [GR10], but they only concern stronger assumptions that are not used in the version of the algorithm presented here.

Algorithm 2.2: Simulation of the AS memory in the IIS model for p_i.

```
1  Shared Objects: infinite sequence of IS memories M_1, M_2, ...;
2  Init: SIM_MEM[1,...,n] initialized to ⊥, SIM_counters[1,...,n] initialized to 0;

3  IS_round = 1, SIM_MEM[i] = Initial_State_i, SIM_counters[i] = 1;

4  Do
5      IS_View ← M_round.WriteSnapshot(SIM_MEM[i], SIM_counters[i]);
6      forall (V, C) ∈ IS_View do
7          for j = 1,...,n do
8              if C[j] > SIM_counters[j] then
9                  SIM_MEM[j] ← V[j], SIM_counters[j] ← C[j];

10     sum = 0, for j = 1,...,n do sum = sum + SIM_counters[j];
11     if (Undecided) ∧ (IS_round = sum) then
12         SimulateSnapshot(SIM_MEM);
13         if Decision(SIM_MEM) ≠ ⊥ then
14             SIM_MEM[i] = ⊥, SimulateDecision(Decision(SIM_MEM));
15         else SIM_MEM[i] ← SIM_MEM, SIM_counters[i] ← SIM_counters[i] + 1 ;
16     IS_round + +;
17 While true;
```

The proof of correctness needs to tackle several aspects. The primary auxiliary proof concerns showing that the sum of operations counters made at line 10 is always smaller or equal to the round counter for undecided processes. It comes directly from a sum increasing with rounds, and strictly increasing in rouns where the sum is equal to the iteration counter. As operation counters increase only with new operations completed by undecided processes, the increase of rounds ensure the liveness of the simulation, i.e., if there is an undecided process then at least one undecided process will eventually complete a new operation.

Showing the safety of the simulation is more intricate as it consists in providing a valid order for the simulated operations. It is done by using the order of the simulated snapshot operations and placing simulated update operations just before the first simulated snapshot operation observing it. Then, showing the consistency of simulated operations boils down to showing that the containment and self-inclusion properties are well inherited from the immediate snapshot properties.

Chapter 3

Distributed Simulations

Comparing the computational power of models is a serious challenge. For example how an n-process system with access to test-and-set objects compares with an adversarial model which ensures an odd number of correct processes? In this chapter, we discuss *simulations* as means of determining relative computability in the universe of models.

3.1 Relating Models via Agreement-Based Simulations

Distributed simulations were introduced to provide a generic way for a model to emulate executions of another model. A classic simulation technique is the *universal construction* by Herlihy in [Her91]. The idea consists in using the ability of a system to solve consensus to reach agreement on all simulated operations one by one. But most distributed systems cannot solve the consensus task. Hence, managing to produce valid simulated executions while submitted to concurrency is an important feature for widely applicable simulation techniques.

In this thesis, we primarily focus on simulation techniques inspired by *BG simulation*, the classical simulation tool invented by Borowsky and Gafni [BG93a, BGLR01]. Our goal is to design a novel agreement-based simulation using insights from Gafni and Guerraoui [GG09].

We proceed incrementally. We start with a simple BG-like simulation. We then add additional features, one at a time, each time yielding a more refined and powerful simulation. This approach also helps to compare the simulations and to understand the advantages each feature provides. The final simulation, mixing a flexible agreement-based synchronization and a simple round-by-round structure, is widely applicable and yet very simple to use.

We start, in Section 3.2, by recalling the principles of BG simulation. In particular, we focus on the *safe-agreement* protocol which allows synchronizing simulators to avoid potential conflicts in simulated operations. It also enables us to define formally the basic structures and properties followed by the distributed simulations presented in this chapter.

Next, in Section 3.3, we improve the simulation by providing the ability to abort pending simulation steps, which results in a simulation similar to *Extended BG simulation* introduced by Gafni [Gaf09]. We continue, in Section 3.4, by refining the use of this abort mechanism and proposing a simpler variant of Extended BG simulation. This new simulation has a straightforward round-by-round structure which can be used to easily prove fine properties of the simulated executions. We proceed to the last improvement in Section 3.5, by replacing the safe-agreement protocol with a more flexible *agreement-based* synchronization. This last simulation will be instrumental in comparing models task computability in the next chapter.

3.2 Safe Agreement and BG Simulation

We use here a different approach than the original BG-simulation technique [BG93a, BGLR01]. We first provide an explicit simulation of atomic update and snapshot operations that is expected to build a valid AS (atomic snapshot) run, under the condition that the simulation calls respect certain restrictions.

Then we introduce a non-blocking version of the *safe-agreement* protocol that can be used to ensure agreement on simulated steps. Using these two constructs, we provide the simulation scheme that meets the specified requirement and briefly discuss how it can be applied.

We assume in this chapter that an n-process (of n *simulators*) model intends to simulate an m-process model (of m *simulated processes*). Moreover, we assume that simulated processes run the *full-information protocol*, i.e., they start with an *update* operation using their initial state as a parameter and then alternate between *snapshot* and *update* operations, each time using the output of the preceding *snapshot* operation as a parameter for the *update*.

Shared Memory Simulation

The first issue in simulating another system is that multiple simulators should be able to perform memory operations on behalf of the same simulated process. But, by default, processes have access to a single-writer atomic snapshot memory. Hence, we first need to simulate a multi-writer atomic snapshot memory for the simulated processes. The issue is somewhat standard, but let us formally recall how to do so, as we will later extend it with additional features.

Implementation description. We use an array of $n \times m$ registers $SIM\_MEM[n][m]$. Each couple of simulator s_i and simulated process b_j is associated with the register $SIM\_MEM[i][j]$. Each register contains a couple (c, v) where c is an operation counter and v is a process state.

Simulator s_i can directly simulate the c^{th} update operation of value v from b_j by updating $SIM\_MEM[i][j]$ with (c, v) (Line 3). To simulate a snapshot, simulators can take a snapshot of SIM_MEM and return, for all simulated process b_j, the written value associated with the greatest counter in $SIM\_MEM[i][j]$, for all $i \in \{1, \ldots, n\}$ (Line 6). Note that $\max_{lex}$ corresponds to the maximal value according to the *lexicographical* order, hence by comparing first the operation counter before comparing, in case of equality, the written state. These two operations, dealing with initializations issues, are presented in Algorithm 3.1.

Algorithm 3.1: Shared memory simulation for simulator s_i.

```
1 Shared objects: SIM_MEM : Snapshot object of n × m registers initialized to ⊥;

2 simUpdate(j, c, v):
3 |   SIM_MEM[i][j].update(c, v);

4 simSnaphot():
5 |   Let res be an m-dimensional array; S ← SIM_MEM.snapshot();
6 |   forall j ∈ {1, ..., m} do res[j] ← max_lex{snap[k][j] | ∀k ∈ {1, ..., n}};
7 |   Return res;
```

Simulation requirements. Simulators must fulfill two requirements to ensure that operations simulated using Algorithm 3.1 produce consistent transitions of the simulated atomic snapshot memory:

1. *Unicity:* For all $c \geq 0$ and simulated processes b_j, no two simulators may invoke $simUpdate(j, c, v)$ with different values v.
2. *Consistency:* Let a simulator s_i invokes $simUpdate(j, c, v)$. If $c = 0$, then v must be a *valid* initial state of simulated process b_j. If $c > 0$, then v must contain the output of a preceding $simSnaphot()$ operation containing the value of a preceding $simUpdate(j, c-1, \_)$.

Proof of correctness. Let us show that if simulators respect these two requirements, then Algorithm 3.1 indeed simulate an m-process atomic snapshot memory. Formally:

Theorem 3.1. *Any run in which simulators execute operations in Algorithm 3.1 respecting the* consistency *and* unicity *properties simulates an AS run on simulated processes.*

Proof. Consider a run of simulated operations $simSnaphot()$ and $simUpdate()$ satisfying the *consistency* and *unicity* properties. We proceed by induction on the number of operations in such a run. The claims of the theorem trivially hold for an empty run.

Now assume that the claim of the theorem is satisfied for a given sequence of simulated operations and that a simulator completes a new operation. Since only update operations modify the state of the memory, consider that this new operation is an update operation. Moreover, since an update operation contains a single atomic operation, we do not need to consider concurrent operations. Two cases are possible:

1) If the update operation is the first simulated update operation with a given set of parameters $(j, c, \_)$, then the new state of the simulated memory is consistent with an update operation (c, v) from b_j. Indeed, by the *consistency* property, this operation write a snapshot consisting in a valid snapshot operation following the $(c-1)^{th}$ update operation by b_j or writes a valid initial state of b_j if no preceding operation of b_j was simulated.

2) Now consider that the simulated operation follows another simulated operation for the same update operation by b_j. As the *unicity* property is satisfied, the other operation wrote the same couple (c, v) to the simulated memory. If a simulated snapshot was returning (c, v) before the update, then it still returns (c, v) afterward. If it was returning a couple (c', v') for b_j, then c' must be strictly greater than c as an update operation can only replace a couple with a smaller operation counter according to the *consistency* property. Hence, in both cases, the result of a simulated snapshot is not modified. This new update operation can, therefore, be mapped to the same operation from b_j as the preceding identical simulated update. □

Safe Agreement

Ensuring that simulators satisfy the (afore defined) consistency property is trivial, as it only depends on a correct ordering of the simulator own simulated operations.

But ensuring that simulators respect the *unicity* property is not trivial. A naive approach would consist in trying to ensure that no two simulators perform simulations of a given update operation concurrently. But this would require to provide *mutual exclusion* [Dij65]. Unfortunately, mutual exclusion cannot be solved wait-free, even in the *one-shot* case. Indeed, this would reduce to the task of test-and-set which is not solvable in most models [Her91].

The solution proposed by Borowsky and Gafni in [BG93a], called *safe agreement*, consists in providing a consensus abstraction which may block in case of a process failure. Intuitively, it can be used to agree on a simulation step safely with progress guaranteed in the absence of failures. The advantage compared to mutual exclusion is that a process failure can block only one safe-agreement protocol, and hence, at most one simulated process.

Definition. The abstraction exports one $propose(v)$ operation which returns a value v'. Unlike the original safe-agreement abstraction [BG93a], we assume that the operation can return a special value $\perp$.

The following properties are satisfied:

— **Validity:** Every non-$\perp$ returned value has been previously proposed.

— **Agreement:** All non-$\perp$ returned values are identical.

— **Termination:** Every correct process eventually returns.

— **Non-Triviality:** If all participating processes return, at least one returns a non-$\perp$ value.

The advantage of allowing processes to return $\perp$ instead of blocking (like in [BG93a]) is that we do not require processes to run multiple safe-agreement instances in parallel. With this definition, we can use safe-agreement protocols sequentially.

Bakery-Style Safe Agreement Implementation

We propose here a novel Safe-Agreement implementation inspired by Lamport's Bakery Algorithm [Lam74]. We observe that the safe-agreement abstraction is close to mutual exclusion, except that multiple processes sharing the same output value are allowed to proceed to the critical section at the same time.

Description of the algorithm. The Bakery Algorithm consists of two stages, first selecting a ticket number and then waiting for its turn to enter the critical section. Our safe-agreement implementation, presented in Algorithm 3.2, is structured similarly to the Bakery Algorithm.

The first stage is used to determine the proposal that can be used for deciding. It uses a *doorway* to restrict the selection to the proposals from the first processes to participate in the protocol. If process i does not observe any proposal in the memory, then it writes its own input to its private register $RES[i]$, and otherwise, $RES[i]$ is left equal to $\perp$. Moreover, processes use a *guard* mechanism to let other processes know they are currently trying to share their proposal. The mechanism consists in setting a boolean register $FLAG[i]$ to true exclusively during this first stage. This stage ensures that at least one of the first processes to participate shares a proposal, but that no late process does.

The second stage consists in trying to find the proposal which might be shared by the process with the smallest identifier. As soon as a process with a raised guard is observed, processes return the particular value $\perp$ as a process may be concurrently sharing its proposal. If there are no raised guard, then the decision value can be determined and returned. It is sufficient for solving safe-agreement as, informally, the last process to exit the first stage cannot see a raised guard, and hence, will not return $\perp$.

Proof of Algorithm 3.2. Let us show that all properties of safe-agreement are satisfied:

— **Termination:** Immediate, as once p_i completes the first stage (Line 6) at least one $RES[j] \neq \perp$.

Algorithm 3.2: Safe-agreement protocol for process i.

```
1 Shared objects: FLAG[1,...,n]: Atomic binary registers initialized to false;
2 RES[1...n]: Atomic registers initialized to ⊥;

3 propose(val):
4     FLAG[i] ← true;
5     if ∀j ∈ {1,...,n} : RES[j] = ⊥ then RES[i] ← val;
6     FLAG[i] ← false;

7     for j = 1,...,n do
8         if FLAG[j] then return ⊥;
9         if RES[j] ≠ ⊥ then return RES[j];
```

— **Non-Triviality:** Assume that all participating processes return and consider the last process, p_i, to lower its flag (Line 6). Afterwards, for each process p_j, we have $FLAG[j] = false$, whether p_j participates or not. Therefore, p_i cannot successfully pass the test at line 8 and so can only return at line 9 with a non-$\perp$ value.

— **Validity:** Processes either returns $\perp$ at line 8 or else a non-$\perp$ value read from a register from RES at line 9, and hence, the *validity* property is satisfied.

— **Agreement:** Let p_k be the process with the smallest identifier to write a (non-$\perp$) value v in RES. We claim that every process that returns a non-$\perp$ value must return v. By p_k definition, no process can return a non-$\perp$ value read in $RES[j]$ such that $j < k$. Suppose, by contradiction, that a process p_i returns a non-$\perp$ value different from v. By the assumption and the algorithm, p_i must have read $\perp$ in $RES[k]$ in line 9. Before that, p_i read *false* in $FLAG[k]$. This can only happen if $RES[k]$ is read by p_i before p_k set $FLAG[k]$ to *true*. Thus, p_i completed line 6 before p_k started line 4. But when p_i completed line 6, it made sure that at least one position in RES is non-$\perp$.
Hence, in executing lines 4-6, p_k must have found a non-$\perp$ value in some RES and therefore cannot write its input to $RES[k]$—a contradiction.

BG simulation

BG simulation uses instances of safe agreement to ensure that simulators performing the same update operation agree on the written state. If a safe-agreement instance returns $\perp$, the simulator becomes *blocked* on this operation. Until another process completes simulating this operation, the simulator can only try to simulate operations for other processes. Obtaining a non-$\perp$ output from a safe-agreement protocol ensures the *unicity* property, hence, if it happens, processes can proceed with simulating the corresponding update operation.

Simulating a Task Solution. As we are interested in *task computability*, we assume that the simulated system runs a protocol solving a task. The simulation is then used to solve a matching task on the simulators. In particular, the first update operation of a simulated process must contain a valid task input computed from the simulators' inputs. Likewise, the simulators' outputs should be determined based on the outputs obtained by simulated processes. The exact relation between the task on the simulators and the task on the simulated processes is application-specific. Hence, in the simulation framework, we specify several

application-specific external predicates.

Task inputs will be dealt with by the primary application function **SelectOperation** selecting which operation to try to perform. But for outputs, the application must additionally define the **Undecided**(S) predicate which must return the set of simulators not yet able to compute a task output given a snapshot of the simulated memory S. Similarly, **Decision**(i, S) should return the corresponding valid output when $i \notin$ **Undecided**(S).

Note that when dealing with *colorless* tasks, as soon as a simulator knows a valid input for one simulated process, it knows one for all simulated processes. Likewise, if a simulator returns, then all simulators can return. Hence, **Undecided**(S) may only be equal to $\emptyset$ or Π.

Simulation protocol. Algorithm 3.3 describes the implementation of the simulation. Simulators first share their input states in the atomic snapshot object *INPUT* (Line 4). Then, simulators proceed to a simulation loop repeated until the simulation advanced sufficiently to provide a task output for the simulator. As soon as a task output can be computed, simulators exit the simulation loop and return a valid task output (Line 14).

The simulation loop works as follows. A simulator first simulates a snapshot operation, using the function of Algorithm 3.1, and takes a snapshot of *INPUT* (Line 6). Then, the primary external function **SelectOperation**, described later in more detail, is used to select which update operation to attempt performing next (Line 7). If the returned operation *op* is undefined ($op = \perp$), then the simulator goes back to the beginning of the simulation loop directly. Otherwise, *op* should be a triplet (j, c, v) providing the simulated process, j, the update operation counter, c, and the simulated state to write, v. A safe-agreement protocol specific to this operation (i.e., indexed by j and c), is then executed using v as argument of the *propose* operation (Line 10). If the value returned by the safe-agreement protocol is distinct from $\perp$, the simulator performs the targeted update operation with the returned value (Line 11). If the safe-agreement protocol returns $\perp$, then the simulator marks this update operation as blocked by raising the value of the local variable $blocked[j]$ to the update operation counter c (Line 12). In both cases, the simulator completes the loop and, if it is still unable to compute a task output, then it starts a new loop iteration.

Algorithm 3.3: BG simulation for simulator s_i.

```
1  Shared objects: INPUT: Snapshot object of n registers initialized to ⊥;
2  Local variables: blocked : m-dimensional array of integers initialized to −1;

3  simulate(input):
4      INPUT[i].update(input);
5      do
6          S ← simSnaphot(); I ← Inputs.snapshot();
7          op ← SelectOperation(S, I, blocked);
8          if op ≠ ⊥ then
9              (j, c, v) ← op;
10             v ← safeAgreement[j][c].propose(v);
11             if v ≠ ⊥ then simUpdate(j, c, v);
12             else blocked[j] ← c;
13     while i ∈ Undecided(S);
14     return Decision(i, S);
```

Properties of SelectOperation. The primary function that has to be defined by the simulation application is **SelectOperation**$(S, I, \mathit{blocked})$, used on Line 7. The first parameter, S, is a snapshot of the simulated system. The second one, I, is the set of simulators task inputs shared so far. The last one, *blocked*, corresponds to the set of operations to simulate with a blocked safe-agreement protocol for this particular simulator.

This function should fulfill three types of requirements. First, it should select non-blocked operations respecting the *consistency* property.

— *Op-consistency*: If **SelectOperation**$(S, I, \mathit{blocked})$ does not return $\perp$, then it should return a triplet (j, c, v) such that: if $c = 0$, then v is a valid task input for b_j; and if $c > 0$, then v should be equal to S with $S[j] = (c - 1, \_)$.

— *Availability*: If **SelectOperation**$(S, I, \mathit{blocked})$ returns (j, c, v), then $\mathit{blocked}[j] < c$.

We assume that the two properties above are satisfied by default.

Secondly, the function should pilot the simulation to produce only valid runs of the simulated model. Indeed, without any restrictions, the simulation will produce any valid m-process AS run. But if we intend to simulate only specific AS runs, the selection should be made accordingly. Since it is application-specific, proving the validity of the produced runs requires knowing the function and the model, as we will see later.

Finally, the function should avoid returning $\perp$ too often to guarantee progress with the simulation. One way to ensure some progress is to never return $\perp$ when the number of blocked simulators is too small. For example, we may impose the following condition:

— *k-selection*: If there are at most k simulated processes in *blocked*, then the function **SelectOperation**$(S, I, \mathit{blocked})$ does not return $\perp$.

Correctness. Let us show some basic properties of the simulation.

Theorem 3.2. *Assuming that the function* **SelectOperation** *satisfies* op-consistency *and* availability*, then the simulated operations satisfy* consistency *and* unicity*.*

Proof. It is not difficult to check that the *op-consistency* property directly implies the *consistency* property of the simulated operation. Indeed, consider any simulated snapshot operation, it is either accessed directly with the value returned by the **SelectOperation** function or with the value returned to another process. Indeed, it may only be modified by adopting the non-$\perp$ value returned by a safe-agreement object which therefore is a value proposed by some simulator **SelectOperation** function. But the *op-consistency* ensures that the *consistency* property is consequently verified as the first argument of the **SelectOperation** function is the output of a simulated snapshot operation.

The *unicity* property is a direct result of the agreement property of the safe-agreement operations. But to use it, we first need to show that processes indeed call this one-shot abstraction only once. This results directly from the *availability* and *op-consistency* properties. The *availability* property ensures that a safe-agreement protocol which returned $\perp$ is never reaccessed. The *op-consistency* ensures that a simulated update operation is never proposed again in a new simulation loop. Thus, a safe-agreement protocol is not accessed twice if it returned a non-$\perp$ output. Hence, all non-$\perp$ outputs of safe-agreement protocols are identical, and therefore all simulated update operations having the same associated process and operation counter are identical. □

Let us now show that in a t-resilient system, the $(t + 1)$-*selection* property is sufficient to prove that the simulation is non-blocking:

Theorem 3.3. *If the* **SelectOperation** *function satisfies the* $(t+1)$-selection*, at most* t *simulators may fail, and a correct simulator never returns, then infinitely many selected operations are simulated.*

Proof. Assume that all the theorem conditions are matched and assume by contradiction that only finitely many simulated operations are completed. Simulators attempt only once to simulate a given update operation. Hence, the correct simulator which did not return eventually always obtains $\perp$ from the **SelectOperation** function. Thus, there are at least $t+1$ blocked simulation steps according to the $(t+1)$-*selection* property. But recall that not all processes participating in a safe-agreement instance may obtain $\perp$ (*non-triviality* property). Moreover, a simulator receiving a non-$\perp$ value proceed with simulating the corresponding update operation, which once complete, results in an increase of the operation counter in the simulated snapshot. Therefore, no simulator completes these t blocked operations, as otherwise they would no longer be blocked. Hence, a simulator failed during each of these safe-agreement protocols or before applying the corresponding simulated update operation. Therefore, there are at least $t+1$ failed simulators — a contradiction. □

Application Example

The original application of BG simulation consisted in showing that the task of t-set consensus is not solvable in an n-process t-resilient model with $n > t$. This is a special case of a slightly stronger result:

Theorem 3.4. *For all* $n, m, t \in \mathbb{N}$ *such that* $n > t$ *and* $m > t$*: any colorless task solvable in the* m*-process* t*-resilient model is solvable in the* n*-process* t*-resilient model.*

Proof. Consider the following simulation. The **SelectOperation**$(S, I, \mathit{blocked})$ returns an operation for a non-blocked process b_j among those which have one of the $t+1$ smallest operation counter in S, if possible, and $\perp$ otherwise. This operation either selects a valid input state if $S[j] = \perp$ or returns an operation (j, c, S) with $(c-1, \_) = S[j]$. Note that the simulators and the simulated processes solve the same colorless task, hence, any input shared in I is a valid input for b_j. Therefore, **SelectOperation** satisfies the *op-consistency* and the *availability* properties. Thus, according to Theorem 3.2, the simulation produces valid simulated AS memory operations.

Moreover, **SelectOperation** only returns $\perp$ if there are at least $t+1$ blocked processes, so it respects the $(t+1)$-*selection* property. Thus, according to Theorem 3.3, the simulation completes new selected operations until all correct simulators return.

Now assume, by contradiction, that a correct simulator never returns. Thus, there are infinitely many selected operations which complete. Hence, at least one of the $t+1$ processes which have infinitely often one of the $t+1$ smallest operation counters completes infinitely many operations. Therefore, at least $m-t$ simulated processes have infinitely many simulated operations. Thus, we simulate a correct m-process t-resilient execution and so processes with infinitely many simulated operations eventually obtain a task output. As soon as it happens, all simulators may return with the task output of any simulated process — a contradiction.

Thus, we have constructed a simulation which can be used to solve any colorless task solvable in an n-process t-resilient model $(n > t)$ when provided with a solution to the task in an m-process t-resilient model $(m > t)$. □

3.3 Abortable BG Simulation

The main issue with BG simulation is that a simulator crashing while simulating a process step may block the simulation of this process forever. If the task is not colorless, the simulators may have different termination conditions. Hence, it may happen that a failed simulator which, based on the current state of the simulation, can already terminate keeps blocking a safe-agreement protocol. But correct simulators may require the output of this blocked simulated process to compute their outputs.

Avoiding this possible deadlock is the motivation behind the *Extended* BG-simulation proposed by Gafni [Gaf09]. The idea is to provide an *abort* mechanism which takes all processes out of the computation and thus let only correct, non-terminated, processes come back to the simulation. In the original extended BG-simulation, simulators can abort individual simulation steps. For convenience, we present here a different approach where the abort mechanism is *simulation-wide*.

An alternative solution was proposed by Imbs and Raynal [IR09]. Their solution consists, intuitively, in improving the properties of the safe-agreement protocols to equip the simulators with relative priorities. Roughly, the user defines for each safe-agreement protocol a priority order among simulators such that a simulator cannot be blocked in a safe-agreement protocol by another simulator with a lower priority. Implementing this extra property is relatively easy, which produces elegant solutions. But we believe that this solution is not as flexible as the abort mechanism. Indeed, the priorities are fixed and cannot adapt easily to the actual simulation execution. Moreover, as we will show in the following section, the abort mechanism can be automated, and hence, its use becomes invisible from the simulation application perspective.

Commit-Abort

The original abort mechanism uses the *commit-adopt* protocol [Gaf98]. A commit adopt is another form of weak consensus protocol that is wait-free solvable. Processes propose a value and must return with a proposed value and either a *commit* or an *adopt* flag. When all the participants propose the same value, then commit should be returned. Moreover, when a process obtains a commit flag with a value v, all processes should commit or adopt v.

But if our goal is to merely provide an abort mechanism, we can significantly simplify the *commit-adopt* abstraction. Indeed, simulators want either to commit a unique proposal or only to abort the operation. We call the resulting variant of *commit-adopt*—*commit-abort*. This abstraction will prove itself to be much easier to integrate into our simulations.

Definition. A *commit-abort* object can be accessed by two operations. Processes using the *propose* operation must all give the same value as arguments and obtain a boolean as output. Processes using the *abort* operation do not provide any argument and obtain either a *proposed* value (i.e., used as an argument in a *propose* call) or a special value $\perp$. The following properties are satisfied:

— *Agreement*: If a *propose* operation returns *true*, then no *abort* operation returns $\perp$.
— *Non-Triviality*: If no *abort* operation is invoked, then no *propose* operation can return *false*.
— *Termination*: Every operation invoked by a correct process eventually returns.

Commit-Abort implementation. Algorithm 3.4 presents a simple commit-abort implementation. Processes use a multi-value register *PROP* and a boolean register *CLOSED*. A *propose*(v) operation writes v to *PROP* and returns the reverse of the value in *CLOSED*. An *abort* operation writes *true* in *CLOSED* and returns the content of *PROP*.

Algorithm 3.4: Commit-abort object: algorithm for process i.

```
1 Shared objects: PROP : Atomic multi-valued MWMR register initialized to ⊥;
2 CLOSED : Atomic binary MWMR register initialized to false;

3 commit(v):
4 |   PROP ← v;
5 |   return ¬CLOSED;

6 abort():
7 |   CLOSED ← true;
8 |   return PROP;
```

Theorem 3.5. *Algorithm 3.4 implements a commit-abort object.*

Proof. A *propose* operation returns a boolean value to all correct processes. An *abort* operation returns the content of the register *Prop* to all correct processes, thus either its uninitialized value $\perp$ or the argument of a *propose* operation written to it. The Termination property is immediate.

In the absence of *abort* operations, every *propose* operation returns *true*, the initial value of *CLOSED*–the Non-Triviality property is satisfied.

Let us show that the *Agreement* property is also verified. Assume that a process obtains *true* from a propose operation. This process hence read *false* in *CLOSED*, and hence no abort operation wrote to it yet. Therefore, all abort operations will return the non-$\perp$ value written to *PROP* by processes executing *propose* operations. □

Abortable Memory Simulation

Instead of using many *one-shot one-value* commit-aborts, we are going to enrich the shared memory simulation in Algorithm 3.1 with an *abort* mechanism. Intuitively, the abort mechanism should guarantee that all *committed* operations are at least adopted. A committed operation is a simulated update operation which passed its check of an abort call, but it may not have yet applied the operation to the simulated memory.

Implementation. Algorithm 3.5 depicts the *abortable* memory simulation. Compared to Algorithm 3.1, here we assume that two types of modifications may be applied to the simulated memory: a value can be *adopted* or *committed*.

A committed value can be seen as a completed operation as before in *SIM_MEM*. Adopted values are stored in an atomic snapshot *ADOPT_MEM*. Note that several rounds of adoption may be executed successively for the same operation, hence, to keep track on the order of these rounds, the simulator equips the adopted operations with the *abort counter*.

The simulated snapshot operation is unchanged, as it only takes into account committed operations. To obtain the set of adopted operations, we provide a variant of the simulated

snapshot operation called *adopted*(); the only difference is that here the maximal returned value is computed lexicographically based on the operation counter, the abort counter, and finally the states which are shared by update operations.

Algorithm 3.5: Abortable memory simulation for simulator s_i.

```
1  Shared objects: SIM_MEM: Snapshot object of n × m registers initialized to ⊥;
2  ADOPT_MEM: Snapshot object of n × m registers initialized to ⊥;
3  EPOCH : Snapshot object of n registers initialized to 0;

4  simUpdate(j, c, e, v):
5  |  ADOPT_MEM[i][j].update(c, e, v);
6  |  if e = epoch() then SIM_MEM[i][j].update(c, v);

7  simSnaphot():
8  |  let res be an m-dimensional array; S ← SIM_MEM.snapsot();
9  |  forall j ∈ {1, ..., m} do res[j] ← max_lex{S[k][j] | ∀k ∈ {1, ..., n}};
10 |  return res;

11 adopted():
12 |  let res be an m-dimensional array; A = ADOPT_MEM.snapsot();
13 |  forall j ∈ {1, ..., m} do res[j] ← max_lex{A[k][j] | ∀k ∈ {1, ..., n}};
14 |  return res;

15 abort(e):
16 |  EPOCH[i].update(e + 1);

17 epoch():
18 |  return max(EPOCH.snapshot());
```

The simulated update operation is slightly more complex. Indeed, a simulator cannot just apply an operation as before. The operation must be first simply proposed for adoption by updating *ADOPT_MEM*. Then the simulator must check if another simulator performed a conflicted abort operation by checking an abort counter. Note that the abort counter is simulated using a snapshot object *EPOCH*, the abort counter is then the maximal value found in it. If the abort counter is identical to the one tagged with the operation, then the operation is up to date and can be committed by writing it to *SIM_MEM*.

The abort operation takes an abort counter to be invalidated. Simulators do it by just writing an incremented value in their *EPOCH* register. We also provide an *epoch*() function which can be used to obtain the current value of the abort counter.

Simulation requirements. The safety of the simulation relies on similar assumptions as for the non-abortable shared-memory simulation. The *unicity* property should be updated as follows:

— *Abortable-unicity*: For all $c \geq 0$, $e \geq 0$ and simulated processes b_j, no two simulators may invoke $simUpdate(j, c, e, v)$ with different values v.

Note that we only add the abort counter to the list of parameters required to be matched to check for the unicity of the written state. But restricting the scope of the *unicity* property

to a subset of corresponding update operations requires to add another property about the correct adoption of operations. We do it by updating the *consistency* property as follows:

— *Abortable-consistency*: Let a simulator s_i invokes $simUpdate(j, c, e, v)$.

1. If $c = 0$, then v must be a *valid* initial state of simulated process b_j;
2. If $c > 0$, then v must contain the output of a preceding $simSnaphot()$ operation containing the value of a preceding $simUpdate(j, c-1, \_)$;
3. An *adopted* operation, preceding the *simUpdate* operation but following an *epoch* operation returning e, returned A such that $A[j] = (c, \_, v)$, $A[j] = \perp$ or $A[j] = (c', \_, \_)$ with $c' < c$.
4. The parameter e must come from an *epoch* operation executed after the last *simUpdate* operation.

Intuitively, the *abortable-consistency* requires additionally that simulators, once they selected their abort counter, check if an operation is available for adoption before proposing a new one. Note that we also need to add a third property about the validity of abort operations:

— *Abort-validity*: An *abort* operation must use the output of an *epoch* operation as argument.

Proof of correctness. Let us show that if simulators respect these three requirements, then Algorithm 3.5 indeed simulates an m-process atomic snapshot memory. Formally:

Theorem 3.6. *Any run in which simulators execute operations in Algorithm 3.5 respecting the* abortable-consistency*,* abortable-unicity *and* abort-validity *properties simulates an AS run on simulated processes.*

Proof. To show the correctness of the abortable simulation, we can simply show that the *abortable-consistency*, *abortable-unicity* and *abort-validity* imply the *consistency* and *unicity* properties on the values written to SIM_MEM. The *consistency* property is directly implied by the first two items of the *abortable-consistency* property.

Hence, consider two update operations (c, v) and (c, v') applied to $SIM\_MEM[j]$ by respectively $simUpdate(j, c, e, v)$ and $simUpdate(j, c, e', v')$ operations and let us show that $v = v'$. If $e = e'$ then $c = c'$ by the *abortable-unicity* property. Hence, w.l.o.g., assume that $e < e'$. As the $simUpdate(j, c, e, v)$ updated $SIM\_MEM[j]$, then $ADOPT\_MEM[j]$ was updated with (c, e, v) before obtaining e from an $epoch()$ operation. But values returned by consecutive $epoch()$ operations must return increasing values, by a simple application of the *abort-validity* property. Hence, the third condition of the *abortable-consistency* for $simUpdate(j, c, e', v')$ must be satisfied for the result of an $adopted()$ operation executed after $ADOPT\_MEM[j]$ was updated with (c, e, v). Note that the couples (c, e) presents in items from arrays returned by *adopted* operations can only increase. Hence, the $adopted()$ operation could not have returned A such that $A[j] = \perp$ nor $A[j] = (c', \_, \_)$ with $c' < c$. Hence $A[i] = (c, \_, \_)$.

Let us show that $A[j] = (c, \_, v)$. Assume by contradiction that it is not the case, hence that some $simUpdate(j, c, e'', v'')$ operation with $e'' > e$ performed an update operation on $ADOPT\_MEM[j]$ inbetween. Without loss of generality, consider the first one to do so. This *simUpdate* operation must respect all constraints that we have shown to be respected by our $simUpdate(j, c, e', v')$ operation. But since it is the first to modify $ADOPT\_MEM[j]$, its *adopted* operation must have returned A' such that $A'[j] = (c, \_, v)$. Hence v'' must be equal to v — a contradiction. Hence, $A[j] = (c, \_, v)$ and thus $v' = v$. □

Extended BG simulation

We can now present a variant of the extended BG-simulation [Gaf09]. Roughly, the idea is merely to replace the simulated memory in Algorithm 3.3 with the abortable simulated memory. Moreover, in each round, the simulation checks whether the application wants to proceed to the abort mechanism. This straightforward extension is presented in Algorithm 3.6.

Description of the Algorithm. The algorithm is a straightforward generalization of the standard BG-simulation from Algorithm 3.3. This time, at the beginning of each simulation round, processes not only take a snapshot of the simulated memory but also take a snapshot of currently adopted values as well as the abort counter. Note that the abort counter is the first to be updated. Then, before selecting a process to try to simulate, simulators check if an abort call should be made based on the current status of the simulation. The simulation application should ensure that finitely many abort calls are made per simulator for a given simulation status. If an abort call is made, then simulators skip the rest of the simulation round.

Otherwise, simulators proceed with the selection of which process to simulate. The only difference is that the external **SelectOperation** now has an extra parameter A which is the result of the *adopted* operation. Moreover, the **SelectOperation** should now satisfy the refined version of *abortable-consistency* property. Another difference lies in the fact that the safe-agreement protocol is now additionaly indexed with abort counters. As before, if the safe-agreement protocol is successful, then the update operation is attempted to be simulated to the memory before moving to the next simulation round. Note that now, the update operation may be unsuccessful due to a call to the abort mechanism. If the safe-agreement protocol is not successful, then the local variable *blocked* is updated. Note that the *blocked* array now store couples of an operation counter and an abort counter.

Algorithm 3.6: Abortable BG simulation for simulator s_i.

```
1  Shared objects: INPUT: Snapshot object of n registers initialized to ⊥;
2  Local variable: blocked : A m-sized array of couples of integers initialized to
     (0, −1);

3  simulate(input):
4      INPUT[i].update(input);
5      do
6          e ← epoch(); S ← simSnaphot(); A ← adopted();
7          if SelectAbort(S, I, blocked, e) then abort(e);
8          else
9              op ← SelectOperation(S, A, I, e, blocked);
10             if op ≠ ⊥ then
11                 (j, c, v) ← op;
12                 v = safeAgreement[j][c][e].propose(v);
13                 if v ≠ ⊥ then simUpdate(j, c, e, v);
14                 else blocked[j] = (e, c);
15     while i ∈ Undecided(S);
16     return Decision(i, S);
```

Properties of SelectOperation and SelectAbort. As before, some of the external functions used in the in the simulation must satisfy some properties to ensure that the valid operations are simulated. Concerning **SelectOperation**, it consists in ensuring *abortable-consistency* by merely adapting it to the function structure:

— *Op-abortable-consistency*: If **SelectOperation**$(S, A, I, e, blocked)$ does not return $\perp$, then it should return a triplet (j, c, v) such that:

1. If $c = 0$, then v is a valid task input for b_j.
2. If $c > 0$ and $A[j] = (c, e', v')$, then v should be equal to v'.
3. Otherwise, v should be equal to S with $S[j] = (c - 1, \_)$.

— *Abortable-availability*: If **SelectOperation**$(S, A, I, e, blocked)$ returns (j, c, v), then $blocked[j] < (c, e)$.

On the other hand, the function **SelectAbort** does not need to satisfy any condition to ensure that simulated operations are valid. But it should meet some requirements to ensure that some progress can be made:

— *Abort-parsimony*: If **SelectAbort**$(S, I, blocked, e)$ returns true, then the preceding **SelectAbort** operation to return true had a different simulated snapshot operation output as argument.

Note that the *abort-parsimony* property is sufficient to ensure that the abort operations do not prevent liveness of the simulation, but it is not necessarily required. Liveness also requires additional properties about the limited use of $\perp$ as output for the **SelectOperation** property. An example of such a limitation is the *k-selection* property introduced in Section 3.2. But showing the liveness of the simulation or that it simulates valid runs of the target models are application dependent.

Correctness. Let us now show some basic properties of the simulation.

Theorem 3.7. *Assuming that function* **SelectOperation** *satisfies* op-abortable-consistency *and* abortable-availability*, then the simulated operations satisfy the* abortable-consistency*, the* abortable-unicity *and the* abort-validity *properties.*

Proof. Showing the validity of *abortable-unicity* follows the same argument as for Theorem 3.2, deduced using the *abortable-availability* property and the *agreement* property of safe-agreement protocols. Moreover, checking the *abort-validity* property is trivial.

Hence, let us show that the *op-abortable-consistency* property implies the *abortable-consistency* property for the simulated operation. Thus, consider any simulated snapshot operation with arguments (j, c, e, v). Using the *validity* property of safe-agreement protocols, we can easily show that (j, c, v) was returned by a **SelectOperation** with e as an argument. Hence, the first two conditions of *abortable-consistency* are provided by the first two matching conditions of the *op-abortable-consistency* property. The third condition follows from the third condition of *op-abortable-consistency* and the fact the abort counter e is computed from the *epoch* function before the snapshot A given as an argument to **SelectOperation** is computed using the *adopted* function. The last condition of *abortable-consistency* is ensured directly by the simulation structure as e is computed from the *epoch* function at each round. □

Let us now show that in a t-resilient system, the $(t + 1)$-*selection* property is still sufficient to show that the simulation is non-blocking when the *abort-parsimony* property is satisfied:

Theorem 3.8. *If the* **SelectOperation** *function satisfies* $(t+1)$-selection*, the* **SelectAbort** *function satisfies* abort-parsimony*, at most* t *simulators may fail, and a correct simulator never returns, then infinitely many selected operations are simulated.*

Proof. Assume that all the theorem conditions are matched and assume by contradiction that only finitely many simulated operations are completed. Hence, after some time, the state of the simulated memory does not evolve anymore. The *abort-parsimony* implies that no more **SelectAbort** return true and hence that no calls to the *abort*() function are made. The simulation behaves then as the simulation from 3.2 and Theorem 3.3 implies that a correct process should perform infinitely many *simUpdate* operations. But since the abort counter no longer increases, all these *simUpdate* operations successfully apply to the simulated memory. □

Application Example

This abortable BG simulation is very similar to the extended BG simulation. Hence, it is of no surprise that it can also be used to reduce the task solvability of the n-process t-resilient model to the solvability of tasks in the $(t+1)$-process wait-free model. Given a (colored) task in the n-process model, the matching task in the $(t+1)$-process model is as follows. A process knows the inputs of $n-t$ processes from the n-process tasks such that for process i, the smallest process identifier associated with a task input is i. Let us consider the simulation by the $(t+1)$-process wait-free model of the n-process t-resilient model which can be used to show that a solution to $\mathcal{T}$ implies a solution to $\mathcal{T}'$.

The simulation is similar to the simulation for colorless task presented in Section 3.2. Processes try to simulate one of the $t+1$ non-blocked processes with the least advanced simulation. As for the colorless case, $n-t$ simulated processes will make progress as long as no simulator terminates. Hence, eventually, at least $n-t$ simulated processes will obtain task outputs. When it happens, at least one simulator may terminate. But now there are only $t+1-k$ active simulators, with k the number of terminated simulators. At this point, processes can select to simulate only the $t+1-k$ least advanced processes.

The issue that existed in BG simulation is that the process with a task output may still be active if it failed, hence, not allowing to reduce the number of targeted processes. But now, processes can make a call to the abort mechanism to make this simulator inactive. If it resumes the simulation, it will directly terminate. But since processes do not know easily when a simulated process is blocked by a failed simulator provided with a task output, we make simulators preemptively proceed to an abort each time they see an increase in the number of simulators provided with a task output. Hence, after this abort call, processes can select to simulate only the $t+1-k$ least advanced processes, with k the number of simulators provided with a task output. As long as there is a correct non-terminated simulator, $n-t+k$ simulated processes make progress. When all these processes obtained a task output, at least $k+1$ simulators are provided with a task output. Therefore, as long as there is a correct non-terminated simulator, the number of simulators provided with a task output increases. Hence, all correct simulators eventually terminate.

Note that this simulation is quite simple to define and showing that the calls to the abort mechanism do not prevent the liveness of the simulation is trivial. But with other simulation applications it could prove to be more complex. This is why we are going to show how the abort mechanism can be removed from the application side at no cost.

3.4 Round-Based Simulation

In order to simplify the simulation design of the extended BG-simulation, the ability to make abort calls can be removed. Indeed, calls can be made as often as desired as long as it does not prevent the liveness of the simulation. In particular, no calls to the abort mechanism should eventually be made when no new simulated update operations are successfully completed.

What can be done is to use the abort mechanism systematically each time a new update operation is successfully completed. But we can go even further. Indeed, as long as processes do not agree on the set of completed update operations, they can proceed to abort calls. Indeed, either some new update operations are regularly completed or else, all processes should eventually agree on the set of completed update operations and no longer make calls to the abort mechanism.

This allows us to construct a round-based simulation where all simulators participating to the same round agree on the set of completed update operations, or equivalently, on the result of the simulated snapshot operation. But some additional mechanism is required to ensure that all simulators entering the same round agree on their simulated snapshot. This can be done by using *conflict detectors*.

Conflict Detectors

Another protocol that is used in our simulation is a conflict detector [AE14]. It was introduced as a simplified version of Commit-Adopt. Indeed, a Commit-Adopt protocol can be shown to be decomposable into two complementary protocols: a conflict-detector and a commit-abort. The conflict-detector is tasked with simply answering if distinct values are proposed or not, but does not have to deal with the adoption mechanism of a commit-adopt.

Formally, in a conflict-detector, processes access a single operation $check(val)$ which takes a value from some set as input and returns a boolean such that:

— If all processes propose the same value, then *false* must be returned.
— All processes obtaining *true* must share the same proposal.
— All correct processes eventually return.

Multiple implementations of a conflict detector exist and efficient implementations can be found in [AE14].

We use this mechanism to ensure that simulators executing a round of the simulation all agree on the current state of the simulation. This is very convenient as it allows us to design a much simple operation selection procedure. Indeed, simulators are able to know if the operation they select can only complete during the round with operations selected based on the same simulation state.

Using this conflict detector, simulators may fail to agree on the simulation state. But this may happen only finitely many times if no new update operations are completed. If a conflict is observed, simulators can thus simply apply an *abort* operation and move directly to the next simulation round.

Round-Based Extended BG simulation

The abortable shared-memory simulation presented in Algorithm 3.5 can be used for this alternative simulation scheme with no modifications. The global structure of the simulation is

not changed much either. Simulators first share their private input before entering a simulation loop until they are provided with a task output.

The distinction mostly comes from this loop structure. Processes compute the current *epoch* before taking a snapshot of the simulated memory and of adopted operations. But then, before continuing further, simulators proceed to a conflict dectection by using the snapshot and the set of registers containing the simultors input values. If a conflict is observed, processes directly launch an abort operation and proceed with a new simulation round. If simulators do not observe any conflict, then they proceed to trying to simulate processes steps almost as before. The main difference is that if processes fail to complete an update operation and if the abort counter did not increase, then they go back to trying to simulate a process step without going back to the begining of the simulation loop and hence they do not actualize their estimation of the simulation state.

Algorithm 3.7: Round-based simulation for simulator s_i.

```
1  Shared objects: INPUT: Snapshot object of n registers initialized to ⊥;
2  Local variable: blocked: m-sized array of couples of integers initialized to (0, −1);

3  simulate(input):
4      INPUT[i].update(input);
5      do
6          e ← epoch(); S ← simSnaphot(); A ← adopted();
7          if (¬conflictDetector[e].apply(S, I)) ∧ (i ∈ Undecided(S)) then
8              do
9                  op ← SelectOperation(S, A, I, e, blocked);
10                 if op ≠ ⊥ then
11                     (j, c, v) ← op;
12                     v ← safeAgreement[j][c][e].propose(v);
13                     if v ≠ ⊥ then
14                         simUpdate(j, c, e, v);
15                         abort(e);
16                     else blocked[j] ← (e, c);
17             while e = epoch();
18         else abort(e);
19     while i ∈ Undecided(S);
20     return Decision(i, S);
```

Correctness of the simulation. Showing that simulated operations form a valid AS run is almost identical to the Abortable simulation. As for the previous simulation, the *abortable-unicity* follows directly from the fact that the states to be written by *simUpdate* operations with parameters j, c and e are all identical thanks to the safe-agreement protocol. The *abortable-consistency* is enforced by requiring that the **SelectOperation** function also satisfies the *Op-abortable-consistency* and the *Abortable-availability* properties. Therefore, this round-based simulation simulates steps of a valid AS run. To show that infinitely many calls to the *simUpdate* operation indeed produces infinitely many simulated operations is not complicated either. Indeed, if no new operations are completed, then the value of simulated

snapshots do not evolve. Moreover, eventually the *INPUT* array is constant and calls to the conflict detector may only return *false*. Hence, the *abort* can only be used after *simUpdate* operations are performed. Consider the first simulator to proceed to an *abort* operation with e as argument. Hence, at this point the abort counter never exeeded e. Thus the *simUpdate* operation performed just before by this simulator was associated with the abort counter e and hence successfully updated the simulated memory.

Showing properties about the liveness of the simulation also depends on the *selectOperation* definition. Let us show that the simulation is lock-free if the following property is verified:

— *active-selection*: *selectOperation*$(S, A, I, e, blocked)$ may not return $\perp$ if there are k or less blocked simulated processes, with k the number of possible simulators failures among **Participating**$(I) \cap$ **Undecided**(S).

Indeed, assume that this property is satisfied but, by contradiction, that eventually no new *simUpdate* operation is successfully completed. As we have previously shown, if infinitely many *simUpdate* operations are completed, infinitely many of them are successful. Hence, correct simulators must eventually always obtain $\perp$ from the **SelectOperation**. This implies that there are at least $k+1$ blocked simulated processes, with k the number of possible failures among **Participating**$(I) \cap$ **Undecided**(S). Therefore, there are at least $k+1$ failed simulators blocking simulation steps. These simulators are participating, and so, one of them cannot belong to **Undecided**(S). But this simulator must have failed the test on line 3.7 at least for the current simulation round — a contradiction.

Simulation Application

Let us show that the definition of the simulation example from Section 3.3 can be simplified a bit with this new simulation framework. Indeed, we can now define the **selectOperation** function to simply return either $\perp$ if the $k+1$ processes with the smallest snapshot outputs are blocked, where k is equal to the number of possible failures in **Participating**$(I) \cap$ **Undecided**(S), or else return a valid operation for the least advanced non-blocked process. The proof of correctness can then directly show that the simulated run induces that the number of simulators with task outputs must increase without taking into account any abort mechanism or call.

3.5 Agreement-Based Simulation

All the simulations presented so far are based on safe-agreement protocols. While safe agreement is a very nice synchronization tool, it can only take advantage of the resilience of the system to lower concurrency. In this last simulation, we are going to replace it by an agreement-based synchronization. It was originaly proposed in a technical paper by Gafni and Guerraoui [GG09]. They have shown in [GG11] how k-set consensus operations can be used to concurrently simulate k state machines. State machines are abstractions which maintain a local state and can apply operations to it sequentially. In [GG09], they propose to enhance these state machines in order to accept memory operations too. This way, a k-process system is simulated. They then propose to use it to execute a BG simulation as a second layer of simulation. Hence, designing simulation applications is rather complex as both layers should be taken into account .

Instead of combining the two simulation techniques, in our simulation, we directly integrate the agreement-based synchronization technique to replace the safe-agreement protocol.

The state machine replication can be seen as three main steps. The first step consists in reducing the distinct set of proposed operations through a set consensus operation. Then, in a second step, the set of operations selected are written in a snapshot object. This allows, by taking a snasphot, to order the set of proposed operations with an associated rank such that sets associated with the same rank agree on the selected set. These two steps form a solution to the problem called k-simulataneous consensus [AGR$^+$06]. The last step consists in applying these selected operations using commit-adopts protocols (used as commit-aborts). Indeed, this last step is required as processes are only aware of their selected operations but not the one selected by other processes.

In our simulation presented in Section 3.4, we already provides a round-based step validation with systematic abort of incomplete operations. Hence, we can integrate the synchronization based on simultaneous agreement directly without using the third step. Still, the simultaneous consensus select proposed operations associated with a rank from 1 to k. But in our simulation we wis to simulate operations for a simulated system composed of more than k processes. We therefore improve the protocol to consider operations associated from a larger set of possible operations. This new protocol is called the *dispatcher*.

Dispatcher

The simulated update operations require that simulators do not propose distinct new states to write for the same update operation in a given round of the simulation. The idea is to use a construct similar to the one introduced in [AGR$^+$06] to solve the problem of simultaneous consensus. Simulators are expected to provide a large enough set of proposed operations, so that they can be "dispatched" to operations corresponding to distinct simulated processes if they cannot agree on the state to write for the simulated update operation.

We call this protocol a dispatcher. The dispatcher is accessed by processes with at most k distinct sets of at least k proposed operations. Processes first write their set of operations to an atomic snapshot object $MEM[1, \ldots, n]$. Then they take a snapshot of MEM and select a value from it according to the number l of distinct lists of operations which are observed. The selected operation is the one associated with the largest simulated process identifier which is the l^{th} smallest of its list of proposals. This ensures (1) that if processes see a different number of distinct sets of operations, then they select proposals associated with distinct simulated processes, and (2), that if processes see as many distinct sets of operations, then they return the same operation.

The detailed implementation of the dispatcher is presented in Algorithm 3.8. Note that operations are selected using the lexicographical order, that is, comparing first the simulated process identifier, then the operation counter, before, as a last resort, comparing the state to be written. Comparing states to be written is not well defined but any total order may be used as we only require that all simulators select the same state to write if there are multiple choices for the same simulated operation.

Lemma 3.1. *If processes submit at most k distinct set of operations, each set composed of at least k operations for distinct simulated processes, then the dispatcher returns an operation such that all processes obtaining operations for the same process have the same operation.*

Proof. Consider two processes p and p' observing a different number of distinct sets of operations, ℓ and ℓ' respectively, and, w.l.o.g, such that $\ell < \ell'$. Consider the set of operations from which p selected its operation. As it selected the ℓ^{th} smallest operation, the ℓ'^{th} smallest

Algorithm 3.8: Dispatcher for process p_i.

```
1 Shared objects: MEM, Snapshot object of n registers initialized to ∅;

2 dispatcher(L):
3 |   MEM[i].update(L);
4 |   S ← MEM.snapshot();
5 |   return max_lex{v|∃L ∈ S : |{v' ∈ L|v' ≤ v}| = |{L ∈ S|L ≠ ∅}|};
```

operation is associated with a simulated process with a greater identifier. But the ℓ'^{th} smallest operation is among the possible choices of operations for p' according to snapshots inclusion property. As p' selects the greatest among the ℓ'^{th} smallest operations, it returns one associated with a distinct simulated process than the operation selected by p.

If they observe the same number of distinct sets of operations, then, according to the snapshot inclusion property, they observe the same sets of operations. Therefore, the deterministical operation selection returns the same operation for both processes. □

Agreement Based Simulation

The structure of the simulation is very similar to the round-based simulation from Section 3.4.

On the global structure, there are no differences. Simulators first share their private input before executing rounds of simulation until they are able to produce a task output. In a simulation round, simulators first update the current state of the simulation: the abort counter e; the simulated snasphot S; the set of operations to adopt A; and, the set of known inputs I. Then, they perform a conflict detection on I and S for the round which is indexed by e. As before, if a conflict is detected, the round is aborted and simulators start a new round.

Differences appears if no conflicts are observed. Afterwards, simulators now select a list of operations to simulate using the external function **SelectOperationList**. Processes then reduce the number of distinct sets of operations to simulate using an externally defined set consensus protocol. Note that this set consensus can assume that only processes which were observed as participating and were still undecided may participate. Hence, varying levels of agreement may reached depending on the simulation state. Note also that, if the participation change, we interupt this protocol. This allow to define protocols that terminates if the participation does not have to change. If the participation did not change, then this reduced number of distinct sets of operations is then submitted to the dispatcher which returns a single operation. As simulators obtaining an operation for the same simulated process all share the same operation, the operation can be submitted to a *simUpdate* operation. As before, processes execute an abort operation when *simUpdate* is completed.

Note that we assume that the **SelectOperationList** function returns a sufficiently large set of operations so that the set consensus protocol can reduce the number of distinct sets sufficiently to ensure that the dispatcher successfully returns. The only case where this condition may not be provided is when the set of participating simulators evolved. Hence, before applying an operation, simulators checks that it is not the case. Otherwise all processes are provided with operations to simulate, and $\perp$ may not be returned as it was the case when using safe-agreement protocols.

Algorithm 3.9: Agreement-based simulation for simulator s_i.

```
1  Shared objects: INPUT: Snapshot object of n registers initialized to ⊥;

2  simulate(input):
3      INPUT[i].update(input);
4      do
5          e ← epoch(); S ← simSnaphot(); A ← adopted(); I ← INPUT.snapshot();
6          if (¬conflictDetector[e](S, I)) ∧ (i ∈ Undecided(S)) then
7              opList ← SelectOperationList(S, A, I);
8              try
9                  opList ← SetCons(Undecided(S), Participating(I))[e](opList);
10             until I ≠ INPUT.snapshot();
11             if I = INPUT.snapshot() then
12                 (j, c, v) ← dispatcher[e](opList);
13                 simUpdate(j, c, e, v);
14         abort(e);
15     while i ∈ Undecided(S);
16     return Decision(i, S);
```

Properties of SetCons. The **SetCons** external function is an agreement-based synchronization mechanism. As any agreement task, it takes inputs from some domain, here a set of operations. It must eventually return a proposed input, that is, a set of operations that was previously proposed. The number of distincts returned outputs is therefore smaller than the number of distinct proposed inputs. But, the less distinct outputs may be return, the higher the synchronization is, allowing for a wider range of possibilities for the **SelectOperationList** function and hence for the simulation.

Formally, the **SetCons** operation must only satisfy the following properties:

— *Validity*: Outputs values should be a subset of the input values.

— *Uninterrompted-termination*: Processes must eventually return if the simulation participation does not change.

Besides these very generic requirements of an *agreement task*, the number of distinct outputs that may be returned under specific constraints is primordial to define tha properties that must be followed by the **SelectOperationList**. Given set of processes (U, P) with $U \subseteq P \subseteq \Pi$, we define the synchronization power of the function **SetCons**(U, P), denoted as $sync$(**SetCons**(U, P)) as the maximal number of dictinct outputs that may be returned by processes in U in a run in which only processes in P participated [1]. It can be noted that the **SetCons** must always return proposed inputs, even if processes not in P might participate, but we do not care about the number of distinct outputs that are return in such a setting for $sync$(**SetCons**(U, P)) and hence for the simulation. Indeed, the result **SetCons**(U, P) is discarded by the simulation as soon as the observed participation evolved. Note also that, intuitively, the **SetCons** external function depends only on the simulators model of computation and not on type of simulated runs we wish to perform.

1. More details on this aspect will be given in Chapter 4, we only introduce this *sync* function to formally define the requirements that the **SelectOperationList** must satisfy in our simulation.

Properties of SelectOperationList. The main external function that must be defined for a specific simulation is the **SelectOperationList** function. The main difference with the previously described **SelectOperation** function used for other simulation schemes is that now a set of operations must be returned and not only a single operation. Hence, it is not surprising that we require that returned operations must follow a similar *op-abortable-consistency* property that was required of operations returned by **SelectOperation** in previous simulations. Some small variations on the definition follows from a list being returned:

— *OpList-abortable-consistency*: All items returned by **SelectOperationList**(S, A, I) should triplets (j, c, v) such that:

1. If $c = 0$, then v is a valid task input for b_j, deterministically chosen from I.
2. If $c > 0$ then $S[j] = (c-1, \_)$, and, either $A[j] = (c, e', v)$ or $A[j] \neq (c, \_, \_) \wedge (v = S)$.

Note that, indirectly, the property ensures that a single operation is provided per simulated process. Indeed, given two returned triplets (j, c, v) and (j, c', v'), then the choices of c and v depend only on the valaues of S, A and I.

We no longer have a notion of *blocked* simulation step, hence, we no longer need to satisfy the *abortable-consistency* property. On the other hand, as synchronization is enforce by an agreement-based mechanism, we no longer allow simulators to wait to provide proposals. Moreover, we require simulators to provide sufficiently many proposals, in accordance with the synchronization provided by the **SetCons** function. More formally, we require that the **SelectOperationList** function satisfy the following *setCons-selection* property, that is:

— *SetCons-selection*: the number of operations returned by **SelectOperationList**(S, A, I) is greater than or equal to $sync(\mathbf{SetCons}(\mathbf{Undecided}(S), \mathbf{Participating}(I)))$.

As we will see, this property will have an impact on both the safety and the liveness of the simulation. In particular, it will allow us to show that, without any other assumption, that if a correct simulator never returns, then infinitely many selected operations are simulated.

Correctness of the simulation. Let us now show that simulated operations form a valid AS run in which only selected operations are simulated, and that if a correct simulator never returns, then infinitely many operations are simulated.

Theorem 3.9. *Assuming that function SetCons solves an agreement task with synchronization power equal to* $sync(\mathbf{SetCons}(U, P))$ *and that function* **SelectOperationList** *satisfies opList-abortable-consistency and SetCons-selection properties, then the simulated operations satisfy the abortable-consistency, abortable-unicity and abortable-validity properties.*

Proof. To check that the *abortable-unicity* property is satisfied, consider two executions of *simUpdate* operations with arguments (j, c, e, v) and (j, c, e, v') respectively. A simulator may execute only one update operation for a given epoch e. Hence, these operations are executed by distinct simulators. Moreover, both simulators did not observe any conflict and hence agree on the value of S and I for the round associated to e. According to the *SetCons-selection* property, all such simulators return with lists of proposals containing at least $sync(\mathbf{SetCons}(\mathbf{Undecided}(S), \mathbf{Participating}(I)))$ distinct proposals. As I did not change, all simulators accessing the dispatcher obtained at most $\mathbf{SetCons}(\mathbf{Undecided}(S), \mathbf{Participating}(I))$ distinct list of proposals. Hence, the dispatcher ensures that all simulators obtains proposed simulated steps and that processes obtaining steps for the same simulated process obtains the same simulated step (Lemma 3.1), hence, we have $v = v'$.

Verifying that the *abortable-validity* property is satisfied is trivial as the variable e used for *abort* operations can only be set to the output of an *epoch* operation. Moreover, an *epoch* operation is performed at each round, hence between any *simUpdate* operations, implying the last item of the *abortable-consistency* property.

As concerns the first two items *abortable-consistency* property, it follows mainly from the *opList-abortable-consistency* property. Consider any *simUpdate* operation with argument (j, c, e, v). As shown for the *abortable-unicity* property, (j, c, e, v) must be a proposed simulation step, hence verifying the items of the *opList-abortable-consistency* property. Hence, the first item of *opList-abortable-consistency* implies directly the first item of *abortable-consistency*. For the second item of *abortable-consistency*, it directly follows from the second item of *opList-abortable-consistency* and the observation that e is updated using the *epoch* function before the last update of A with the *adopted* function. □

Let us now show that, we have already strong enough assumptions to ensure the liveness of any simulation satisfying our desired properties, that is:

Theorem 3.10. *Assuming that function SetCons solves an agreement task with synchronization power equal to sync*(**SetCons**(U, P))*,that function* **SelectOperationList** *satisfies opList-abortable-consistency and SetCons-selection properties, and that a correct simulator never returns, then infinitely many selected operations are simulated.*

Proof. Let us assume that all these conditions are verified and let us assume by contradiction that eventually no new operations are simulated. Hence, all correct simulators will eventually always obtain the same values for S and I. Therefore, they will eventually never observe a conflict and undecided simulators will select operation lists. According to the *SetCons-selection* and Lemma 3.1, as shown for the preceding lemma, a proposed operatation will be proposed to the *simUpdate* operation.

Therefore, no *simUpdate* operation may be successful, or equivalently, an *abort* must be executed with the same or a greater epoch before the *simUpdate* operation returned. But as eventually all simulators perform *simUpdate* in every round, the first process to call an *abort* for the epoch must have completed its mathcing *simUpdate* operation before — A contradiction. □

Application Example

We will use this simulation technique on elaborate examples in Chapter 4 to show task computability equivalences between classes of models. Hence, let us only treat here the simple example given for preceding abortable simulations. That is that the task solvability of the n-process t-resilient systems reduces to the solvability of tasks in the $(t + 1)$-process wait-free model. Since we will use this simulation technique extensively, let us treat this example with as many details as possible.

From n-processes t-resilient to $t + 1$-processes wait-free

Let us first start by providing the agreement protocol, before providing a solution through the **SelectOperationList** function.

Definition of *SetCons*. We first need to provide a set-consenus algorithm allowing to reduce the number of proposal sufficiently. Solving the $(t+1)$-set-consensus task in the n-process t-resilient system is a classical and simple algorithm. We need to generalize this solution slightly to improve its level of agreement when the partition is low.

The usual solution consists in selecting a fixed set of $t+1$ processes and returning their proposal. Resilience allows here to ensure that one is seen. The same solution can be applied when the number of participating process is slower. We only need to select the set of $t+1$ processes based on the participating processes. For this, we simply select the inputs from the $|Part| - n + t + 1$ participating processes with the smaller identifier, with $|Part|$ the number of participating processes. Note that the algorithm may block forever, but it is easy to see that it correspond to runs that are not valid for a t-resilient system.

The solution is formalized in Algorithm 3.10. Processes first identify the set of participating processes (Line 3). Then, the set of $|Part| - n + t + 1$ participating processes with the smallest identifiers are selected as leaders. The leaders only share their inputs to a shared memory and return it (Lines 5–7). Other processes wait to observe the input from a leader to select it and return it (Lines 8–11).

Algorithm 3.10: Adaptive set-consensus for t-resilient systems (for process p_i).

```
1  Shared objects: Proposals: Snapshot object of n registers initialized to ⊥;

2  setConsensus(U, I)(opList):
3      Part = {j ∈ {1, ..., i}, I[j] ≠ ⊥};
4      Leaders = {j ∈ Part : |{k ∈ {1, ..., j}, I[k] ≠ ⊥}| ≤ |Part| − n + t + 1};
5      if i ∈ Leaders then
6          Proposals[i] = opList;
7          return opList;
8      else
9          wait until {j ∈ Leaders : Proposals[i] ≠ ⊥} ≠ ∅;
10         let ℓ = min{j ∈ Leaders : Proposals[i] ≠ ⊥};
11         return Proposals[ℓ];
```

Correctness of *SetCons*. Let us show that the validity and uninterrompted-termination properties are satisfied first. Then, we will show that $sync(\mathbf{SetCons}(U, P)) = \max(0, \min(|P| - n + t + 1, |U \cap I|))$.

- **Validity:** Leaders return their own inputs, while others return the one of the inputs shared in the shared memory by the leaders. Hence, all processes return only proposed input values.
- **Termination:** In a t-resilient system, at least $n - t + 1$ processes should be correct. Hence, if the simulation participation does not evolve, all correct processes should be running the set-consensus protocol. Therefore, at most $n - |Part| + t$ processes may fail. This implies that one of the selected leaders is correct and will share its input in the memory. Thus no correct process will be blocked waiting for a leader input on Line 9. Thus the termination property is satisfied.
- $sync(\mathbf{SetCons}(U, P)) = \max(0, \min(|P| - n + t + 1, |U \cap I|))$: This is a direct implication of the leader selection process. Indeed, only leaders inputs may be returned, so the

number of distinct returned values is smaller than the number of selected leaders. But $|P|$ is clearly equal to the size of *Part* computed in the algorithm, hence there are at most $n-|P|+t+1$ selected outputs. Leaders are also participating and non-terminated simulators, hence there are at most $|U \cap I|$ of them too. We use 0 to denote a case where no process returns as no leader is selected.

Definition of SelectOperationList. The main aspect of the simulation consists on defining **SelectOperationList** as it implies which processes might be simulated. This selection is very simple in our case as all processes are valid choices, hence, any valid simulation step is valid. The difficulty lies in the initialization step as a valid simulated input requires $n-t$ simulators inputs. We skip a formal definition for **SelectOperationList** as it only consists on selecting valid simulation steps available. Let us just show that there are enough available choices to satisfy the SetCons-selection property.

Correctness. Let j be the number of participating simulators and let k the number of terminated simulated processes. The number of simulated with valid inputs is equal to $\max(0, |P| - n + t + 1)$. If $k = 0$, then all simulators are non-terminated, therefore there are $\max(0, |P| - n + t + 1)$ available simulation steps. Enough to satisfy the SetCons-selection property. If $k \neq 0$, then $k + n - t - 1$ simulators terminated too. Hence, the number of available simulation steps is equal to the number of available simulators, that is $|U \cap I|$. Hence, the SetCons-selection property is also satisfied.

From $t+1$-processes wait-free to n-processes t-resilient

In this direction, no non-trivial consensus is possible. All processes simply return their own input. Therefore, we have $sync(\mathbf{SetCons}(U, P)) = |U \cap I|$. However, we have more choices of simulated processes steps. Given k participating simulators, we have at least $n - t + k - 1$ simulated processes inputs. The idea of the simulation is to simulate the processes with the least simulated steps. But we still need to select sufficiently many.

Definition of SelectOperationList. The definition of **SelectOperationList** consists in selecting as few operations to perform as possible to maintain liveness, that is $|U \cap I|$ of them. Moreover, we need to ensure that sufficiently many simulated processes makes progress, hence we select to simulate the least advanced simulated processes. Therefore, we select the $|U \cap I|$ simulated processes with the least number of completed simulation steps (possibly already terminated). All we need to show is thus that there are always sufficiently many such choice and that the resulting simulation provides a t-resilient run.

Correctness. First, let us observe that if $k \geq 1$ simulators participate, then there are at least $n-t-1+k$ simulated processes input available. Hence, we always have at least as many available simulated processes than there are active simulators.

Let us now show that we have a valid t-resilient simulated run. For this, let $j < k$ be the number of non-terminated simulators and assume that a non-terminated simulator is correct. This implies that there are at least $n - t + k - j$ simulated processes taking infinitely many steps. Therefore, the simulated run is t-resilient and thus at least $n - t + k - j$ simulated process obtain an input value. This implies that at least $j + 1$ simulated process terminate — A contradiction. Thus all correct simulators eventually terminate.

Chapter 4

Agreement Functions

The question of whether a task is wait-free solvable is undecidable when $n \geq 3$ [GK99,HR97]. Thus, it may seem impossible to compare the task solvability power of models, as determining a necessary and sufficient condition for task solvability is in general impossible. Still, comparing the task computability of models is possible, and even for some models for which task solvability is undecidable. One way consists in using distributed simulations to reduce the task solvability of one model to another. But trying to produce simulations for each possible couple of models would be tedious. Moreover, if we are not able to simulate a model M' in some model M, this does not imply that M' can solve tasks which are not solvable in M. Hence, additional considerations are required. In this chapter, we investigate the possibility of measuring the relative task computability of models through their ability to solve specific agreement tasks.

We start by introducing, in Section 4.1, the notion of *agreement function*. An agreement function stipulates, for a given model M, which are the best k-set consensus tasks solvable in M among any set of processes and under any limited participation. In Section 4.2, we examine the properties of agreement functions. We also introduce and compare natural classes of well-structured agreement functions. Afterwards, we study the agreement functions of multiple models by computing them, determining their class and showing that they grasp the relative task computability of all considered models. In Section 4.3, we focus on families of models defined through the notion of *active-resilience*, i.e., when the number of possible failures depends on the participation. In Section 4.4, we continue with adversarial models and analyse how adversary families relate to agreement function classes. Lastly, in Section 4.5, we discuss the partial model hierarchy, induced by models relative task computability, which is obtained by the characterization of models by their agreement functions.

4.1 Definition of Agreement Functions

We first introduce the notion of *agreement power* of a model M corresponding to the smallest k such that the task of k-set consensus is solvable in M. This notion is then generalized to the *agreement function* of a model M by considering variations on the k-set consensus tasks. A first variation weakens the k-set consensus tasks by requiring the k-agreement property to be satisfied only when the participation is under a given threshold. Another variation looks at the solvability of the k-set consensus task when we restrict the competition to a given set of processes. Agreement functions solely stipulate, for all variations of the k-set-consensus tasks, what are the k for which the task is solvable.

Agreement Power

A model which can solve *consensus* is universal, i.e., it can resolve any "completable" task (the task specification associates any set of inputs to some set of outputs) or it can implement any deterministic object [Her91]. But what about models in which consensus is not solvable? Unfortunately, these models are still quite heterogeneous as they may have distinct sets of solvable tasks.

Definition. It was proposed to generalize the task of consensus by using the set of *k-set consensus* tasks, for $k \in \mathbb{N}^+$ [Cha93]. Recall that in the k-set consensus task processes must return at most k distinct input values. The task of k-set consensus is hierarchical in the sense that a solution of the task of k-set consensus is also a solution of k'-set consensus for any $k' > k$. Indeed, sets of outputs of size at most k are sets of outputs of size at most k' if $k' > k$. Hence, given a model M, there is a natural number k_M such that k-set consensus is solvable if and only if $k \geq k_M$. We call this the *agreement power* of M, denoted as α_M. Note that if M is a model in which the set of processes is of size n, then $\alpha_M \leq n$. Indeed, the task of n-set consensus is trivially solvable in an n-process system as processes can directly output their input without any inter-process communication to solve it.

Characterization of colorless tasks solvability. In colorless tasks, processes can adopt the inputs and outputs of other processes. Thus, for colorless tasks, simulating an algorithm solving a task is much more straightforward. Indeed, the simulation can terminate as soon as one simulated process outputs. Moreover, the simulation can initiate steps of any process as soon as it learns a single task input. Using these restrictions, Herlihy and Rajsbaum showed in [HR12] that all shared memory models with the same agreement power share the same ability to solve colorless tasks.

Unfortunately, this is not true for colored tasks. For example, the test-and-set model, the wait-free model enhanced with test-and-set objects, can solve the task of perfect renaming [AM97]. But perfect renaming cannot be solved in the t-resilient model as soon as $t > 0$ [ABND$^+$90]. Yet, the agreement power of both is equal to 2 when $n = 3$ and $t = 1$.

Limited Agreement Scope

The agreement power of a model provides only limited information about its relative computational power. For example, in a 3-process system, the 1-resilient, 1-test-and-set, and 2-set consensus models all share the same agreement power, 2, while solving different sets of tasks. When comparing the test-and-set model with the others, one can observe that the test-and-set model is the only one that can solve consensus between two processes. Thus, looking at the ability of subsets to solve agreement tasks may prove itself useful in comparing their relative computational power. Not all models are symmetric as the test-and-set model, thus looking at all possible subsets of competitors and not just at their size may be relevant.

Task of k-set consensus among Q. Looking at the ability of models to solve set-consensus among a subset can be formalized using a set of agreement tasks which we call k-set-consensus *among* Q. In the task of k-set-consensus among Q, inputs come from a set of values V ($|V| > k$), outputs are in V, and for each input vector I and output vector O, $(I, O) \in \Delta$ if: (1) the set of non-$\perp$ values in O is a subset of values in I; and (2) the set of non-$\perp$ values of

processes from Q in O is a subset of values of processes from Q in I which size is at most k. The only particular property of this task is the second condition on elements of Δ, which requires that processes in Q return inputs from processes in Q and at most k distinct values.

Agreement power among Q. As it is the case for k-set consensus, a solution to k-set consensus among Q is trivially a solution to the $(k+1)$-set consensus among Q. Thus the notion of agreement power naturally generalizes to the notion of agreement power *among* Q:

Definition 4.1. *[α_M^Q] The* agreement power of a model M among *a subset $Q \subseteq \Pi$, denoted α_M^Q, is the smallest $k \in \{1, \ldots, n\}$ such that the task of k-set consensus among Q is solvable.*

Agreement Power Under Limited Participation

Looking at the ability to solve k-set consensus among subsets of processes allows differentiating the test-and-set model from the 1-resilient and 2-set consensus models. But the 1-resilient model and the 2-set-consensus models are still not discriminated, their agreement power among any subset is the same. The 1-resilient and 2-set consensus models mostly differ when some processes are not yet participating. If a single process took steps, in the 1-resilient model we can wait for another process, while in the 2-set consensus model we must already try to produce an output. To grasp this distinction, we are looking at the ability to solve k-set consensus according to the level of participation.

Task of k-set consensus under P. As for the variation of scope, we consider here a subset of the system P and modify the k-set consensus task to obtain the task of k-set consensus *under* P. In this task, inputs come from a set of values V ($|V| > k$), outputs are in V, and for each input vector I and output vector O, $(I, O) \in \Delta$ if: (1) the set of non-$\perp$ values in O is a subset of values in I; and (2) the set of non-$\perp$ values in O is of size at most k when the set of processes with non-$\perp$ values in I is a subset of P.

Agreement power under P. A solution to the task of k-set consensus under P is also a solution to the task of $(k+1)$-set consensus under P. Thus, the notion of agreement power also naturally generalizes to the notion of *agreement power under* P. But additionally, it is possible that the set of correct processes cannot be a subset of a given participation P. In this case, the task of k-set consensus under P is solvable for any value of k, in particular for $k = 0$ as processes can wait for an increase in participation before returning a task output. Formally:

Definition 4.2. *[$\alpha_M|_P$] The* agreement power of a model M under *a subset $P \subseteq \Pi$, denoted $\alpha_M|_P$, is the smallest $k \in \{0, \ldots, n\}$ such that the task of k-set consensus among Q is solvable.*

Equivalent formulation. Another way to consider this measure would consist in restraining the set of runs of M to include only the executions in which the participation is a subset of P. Let $M|_P$ be this restricted model issued from M. We can notice that, with their similar notations, $\alpha_M|_P$ is equal to $\alpha_{M|_P}$. Indeed, in M, processes can execute a solution to the k-set consensus task in $M|_P$ while checking the participation periodically: if the participation exceeds P, then processes can return their input; otherwise, the task solution must produce an output that can be returned by the processes. Similarly, a solution to the task of k-set consensus under P executed in $M|_P$ solves the task of k-set consensus. Note that when the

set of correct processes in M cannot be a subset of P, then $M|_P$ possesses no infinite run and thus any task is solvable, in particular, the 0-set consensus task.

The task-based definition allows us to consider always the same model and different tasks solvability. It is particularly convenient when used for comparing the relative task computability of models.

Agreement Functions

The agreement power under various participation levels can be used to differentiate models such as the 3-process 1-resilient and 2-set consensus models. But it does not discriminate the 3-process test-and-set model and the symmetric adversarial model in which live sets are of size 1 or 3. Variations on the agreement scope and participation are thus complementary.

Task of k-set consensus among Q and under P. We can combine the restrictions on scope and participation in a single task. Given a set of processes $P, Q \subseteq \Pi$ and a natural integer k, the task of k-set-consensus *among* Q and *under* P can be defined as follows: Inputs come from a set of values V ($|V| > k$), outputs are in V, and for each input vector I and each output vector O, $(I, O) \in \Delta$ if (1) the set of non-$\perp$ values in O is a subset of values in I and (2) the set of non-$\perp$ values of processes from Q in O, is a subset of non-$\perp$ values from processes from Q in O of size at most k when the set of processes with non-$\perp$ values in I is a subset of P. Naturally, when $P = \Pi$, the task boils down to the task of k-set consensus among Q. Moreover, when $P \subseteq Q$, the task boils down to the task of k-set consensus under P.

Agreement function. This generalized family of agreement task is then used to define the notion of agreement power *among* Q and *under* P:

Definition 4.3. *[$\alpha^Q_M|_P$] The* agreement power of a model M under *a subset $P \subseteq \Pi$ and* among *a subset $Q \subseteq P$, denoted $\alpha^Q_M|_P$, is the smallest $k \in \mathbb{N}$ such that the task of k-set consensus among Q and under P is solvable.*

This generalized agreement power is used to define what we call the *agreement function* of a model. Given a model M, the agreement function of M is a map which returns for all possible subsets P of Π and all possible subsets Q of Π the agreement power of M *among* Q and *under* P:

Definition 4.4. *[$\alpha^{\mathcal{F}}_M$] The agreement function of a model M is a map $\alpha^{\mathcal{F}}_M : 2^\Pi \times 2^\Pi \to \{0, \ldots, n\}$ which maps each couple of sets of processes $Q, P \subseteq \Pi$ to the agreement power of M among Q and under P, i.e., $\forall Q, P \subseteq \Pi, \alpha^{\mathcal{F}}_M(Q, P) = \alpha^Q_M|_P$.*

Agreement Functions and Relative Task Computability. Agreement functions can be seen as a predicate on our variations of the k-set consensus tasks, stating whether they are solvable or not. Therefore, if a model M solves all tasks solvable by a model M' then $alpha^{\mathcal{F}}_M \leq alpha^{\mathcal{F}}_{M'}$, i.e., $\forall Q, P \subseteq \Pi, \alpha^{\mathcal{F}}_M(Q, P) \leq \alpha^{\mathcal{F}}_{M'}(Q, P)$. Moreover, if two models M and M' have incomparable agreement functions, i.e., $alpha^{\mathcal{F}}_M \not\leq alpha^{\mathcal{F}}_{M'}$ and $alpha^{\mathcal{F}}_{M'} \not\leq alpha^{\mathcal{F}}_M$, then their task computability are also incomparable.

Unfortunately, having $alpha^{\mathcal{F}}_M \leq alpha^{\mathcal{F}}_{M'}$ does not necessarily imply that M solves all the tasks solvable by M'. Finding a counter-example or trying to show that agreement functions precisely capture the relative task computability of models is an intriguing question that we are not able to answer at this point.

4.2 Properties and Classification of Agreement Functions

The notion of agreement function is quite elaborate as it considers many variations of k-set consensus tasks in its definition. We are first going to show that many possible functions do not correspond to any model. Hence, we can make some additional restrictions on the profile of agreement functions. After looking at the inherent constraints, we are going to look at natural restrictions which do not apply to all models but can be used to define classes of agreement functions. Relations between these classes are then studied, leading to a classification of agreement functions, and hence, of models with such agreement functions.

Inherent Restrictions of Agreement Functions Profiles

The ability to solve some level of k-set consensus task among Q and under P is very often related to the ability to solve k'-set consensus among Q' and under P' when Q, Q' and P, P' are similar to each other. In this section, we provide some constraints about the possible agreement powers that a model may have.

On the agreement power among related subsets. The question here is whether a solution to the task of k-set consensus among Q implies restrictions on the solvability of k'-set consensus tasks among Q'. A trivial limitation comes from the fact that we can let all processes in Q' decide their input to solve $|Q'|$-set consensus among Q'. We can improve this solution if there are more than k processes in $Q \cap Q'$ by letting all processes in $Q \cap Q'$ execute the solution to k-set consensus among Q and let the rest decide as before on their input. Thus we obtain the following constraint:

$$\forall Q, Q', P \subseteq \Pi, \alpha_M^{Q'} \leq \max(|Q'|, \alpha_M^Q + |Q' \setminus Q|).$$

This solution can be further generalized to any set of sets S. Indeed, processes in each subset S_i can use a solution to k-set consensus among S_i while processes in none of the sets return their input. It results in the following constraint:

$$\forall Q \subseteq \Pi, \forall S \in 2^\Pi : \alpha_M^Q \leq \left(\Sigma_{Q_i \in S} \alpha_M^{Q_i}\right) - |Q \setminus (\cup_{Q_i \in S} Q_i)|.$$

Defining lower bounds on the possible agreement power among Q is difficult. For example, consider the wake-up adversarial model in which any set of processes may participate as long as it includes some process p_1. For any set Q which does not include p_1, one can show that the agreement power among Q is equal to $|Q|$ as for them it can be reduced to the wait-free model. But for any set Q which includes p_1, the agreement power among Q is equal to 1 since processes can all select to output p_1 input value. In this example, the agreement power among Q when $p_1 \in Q$ is not related to smaller sets such as $Q \setminus \{q\}$.

On related participation levels. A trivial solution to the task of $|P|$-set consensus under P consists in letting all processes from P return their task input. Moreover, given a solution to k-set consensus under P, we can derive a solution to $(k + |P' \setminus P|)$-set consensus under P'. Indeed, processes in P can apply the solution to k-set consensus under P until it obtains a task output or until it observes a process from $P' \setminus P$. Any output provided by the solution to k-set consensus under P and any input value from a process in $P' \setminus P$ may be returned to solve $(k + |P' \setminus P|)$-set consensus under P'.

This time a trivial lower bound can also be derived. Indeed, a solution to the task of k-set consensus under P is a solution under P' when $P' \subseteq P$. Thus $\forall P \subseteq P' \subseteq \Pi : \alpha_M|_P \geq \alpha_M|_{P'}$. The following property summarizes all these restrictions:

$$\forall P, P' \subseteq \Pi, \alpha_M|_{P \cap P'} \leq \alpha_M|_P \leq \max(|P|, \alpha_M|_{P'} + |P' \setminus P|).$$

In particular for a non-empty set of processes P and any process p from P, we obtain that $\alpha_M|_{P\setminus\{p\}} \leq \alpha_M|_P \leq \min(|P|, \alpha_M|_{P\setminus\{p\}}+1)$. Thus we have either $\forall p \in P, \alpha_M|_{P\setminus\{p\}} = \alpha_M|_P - 1$ or $\alpha_M|_P = \max\{\alpha_M|_{P\setminus\{p\}}, p \in P\}$.

General restrictions. Many possible functions of the type $2^\Pi \times 2^\Pi \to \{0, \ldots, n\}$ are not the agreement function of any existing model. We can first observe that the task of k-set consensus among Q and under P is equal to the task of k-set consensus among $Q \cap P$ and under P.

Moreover, we can transpose the restrictions inherited from the agreement power among subsets or under limited participation. In doing so, we can notice that when extending a solution for P and Q to P' and Q', processes either return using the solution for P and Q or the input value from a process from $P' \setminus P$ or $Q' \setminus Q$. Hence, we obtain the following property:

$$\forall P, P', Q, Q' \subseteq \Pi, \alpha^Q_M|_{P\cap P'} \leq \alpha^Q_M|_P = \alpha^{Q\cap P}_M|_P \leq \min(|Q \cap P|, \alpha^{Q'}_M|_{P'} + |(Q \setminus Q') \cup (P \setminus P')|).$$

Note that as before, we can replace Q' by a set of process sets S by replacing $\alpha^{Q'}_M|_{P'}$ by the sum of all agreement powers among elements of S and under P' and by replacing $Q \setminus Q'$ by the set of processes from Q which are not in any element of S.

These restrictions are not necessarily exhaustive. Hence, any agreement function satisfying these conditions does not necessarily correspond to a model. Still, we believe that agreement functions satisfying these conditions are likely to match some shared memory model.

Classes of Agreement Functions

We can find additional restrictions which are satisfied only for a limited number of models. It allows us to define sub-classes of agreement functions and indirectly of models.

Symmetry. Agreement functions consider the agreement power of models among any subset and under any participation. But in many models, the behavior of processes does not depend on their identifier. In such models, the agreement power among Q and under P should not rely on the composition of P and Q but only on their size $|P|$ and $|Q|$. If this is the case, then we say that the agreement function is *symmetric*:

Property 4.1. *[Symmetry] We say that the agreement function of a model M, $\alpha^{\mathcal{F}}_M$, is* symmetric *if, for all $P, P', Q, Q' \subseteq \Pi$ such that $|P| = |P'|$, $|Q| = |Q'|$ and $|P \cap Q| = |P' \cap Q'|$ we have $\alpha^{\mathcal{F}}_M(Q, P) = \alpha^{\mathcal{F}}_M(Q', P')$.*

Locality. An agreement function is *local* if it depends exclusively on the scope of the tasks. Of course, the agreement power among Q and under P may not always be equal to the agreement power among Q if P does not include all processes in Q. Thus, the natural definition consists in requiring the agreement power among Q and under P to be equal to the agreement power among $Q \cap P$:

Definition 4.5. *[Locality] We say that the agreement function of a model M, $\alpha_M^{\mathcal{F}}$, is* local *if, for all $P, Q \subseteq \Pi$ we have $\alpha_M^{\mathcal{F}}(Q, P) = \alpha_M^{Q \cap P}$.*

Fairness. The natural counterpart to locality is agreement functions depending exclusively on the participation. We say that such agreement functions are *fair*. The agreement power among Q and under P is always smaller or equal to $|Q \cap P|$. Therefore, for a fair agreement function, the agreement power among Q and under P should be equal to the agreement power under P when greater than $|Q \cap P|$:

Definition 4.6. *[Fairness] We say that the agreement function of a model M, $\alpha_M^{\mathcal{F}}$, is* fair *if, for all $P, Q \subseteq \Pi$ we have $\alpha_M^{\mathcal{F}}(Q, P) = \min(|P \cap Q|, \alpha_M|_P)$.*

Regularity. As we discussed previously, the agreement power among Q is not necessarily smaller than the agreement power among Q' when Q' is a superset of Q. Such a property seems natural for many models, and thus we call an agreement function with such a property a *regular* agreement function:

Definition 4.7. *[Regularity] We say that the agreement function of a model M, $\alpha_M^{\mathcal{F}}$, is regular if, for all $P, Q, Q' \subseteq \Pi$ we have $\alpha_M^{\mathcal{F}}(Q, P) \geq \alpha_M^{\mathcal{F}}(Q' \cap Q, P)$.*

Plainness. We started initially by considering only the agreement power of a model without looking at the variation on scope or participation. If an agreement function unaffected by such variations, we say that it is *plain*. Thus except when $Q \cap P$ contains less than α_M processes, the agreement power among Q and under P should be equal to α_M:

Definition 4.8. *[Plainness] We say that the agreement function of a model M, $\alpha_M^{\mathcal{F}}$, is* plain *if, for all $P, Q \subseteq \Pi$ we have $\alpha_M^{\mathcal{F}}(Q, P) = \min(|Q \cap P|, \alpha_M)$.*

This class is obviously quite small. The agreement power of a model is necessarily between 1 and n. Therefore, we say that a model is k-plain if its agreement function is plain and its agreement power is equal to k.

Comparison of Agreement Functions Classes

Let us now compare these classes of agreement functions. Let us start by showing that plain agreement functions correspond exactly to those which are both fair and local.

Property 4.2. *An agreement function is* plain *if and only if it is both* local *and* fair*.*

Proof. Consider a model M with an agreement function which is both fair and local. Thus for all $P, Q \subseteq \Pi$, we have $\alpha_M^{\mathcal{F}}(Q, P) = \alpha_M^{Q \cap P} = \min(|P \cap Q|, \alpha_M|_P)$. Assume that $\alpha_M|_P \geq |P \cap Q|$, in this case we have $\alpha_M^{Q \cap P} = \alpha_M|_P$. Thus $\alpha_M^{\mathcal{F}}(Q \cap P, \Pi) = \alpha_M^{\mathcal{F}}(P, P)$. By taking $P = \Pi$, we obtain that $\alpha_M^{\mathcal{F}}(Q, \Pi) = \alpha_M^{\mathcal{F}}(\Pi, \Pi)$, the agreement power of M. Hence $\alpha_M^{\mathcal{F}}(Q, P) = \min(|P \cap Q|, \alpha_M)$. Therefore an agreement function which is both fair and local is plain.

The reverse direction is even more trivial as if $\forall Q, P \subseteq \Pi : \alpha_M^{\mathcal{F}}(Q, P) = \min(|Q \cap P|, \alpha_M)$, then $\alpha_M^{Q \cap P} = \min(|(Q \cap P) \cap \Pi|, \alpha_M)$ and hence is equal to $\alpha_M^{\mathcal{F}}(Q, P)$. Likewise, we have $\alpha_M|_P = \min(|P|, \alpha_M)$ hence $\alpha_M^{\mathcal{F}}(Q, P) = \min(|Q \cap P|, \alpha_M) = \min(|Q \cap P|, \alpha_M|_P)$. □

A simple observation can be made that, moreover, a plain agreement function is by definition symmetric:

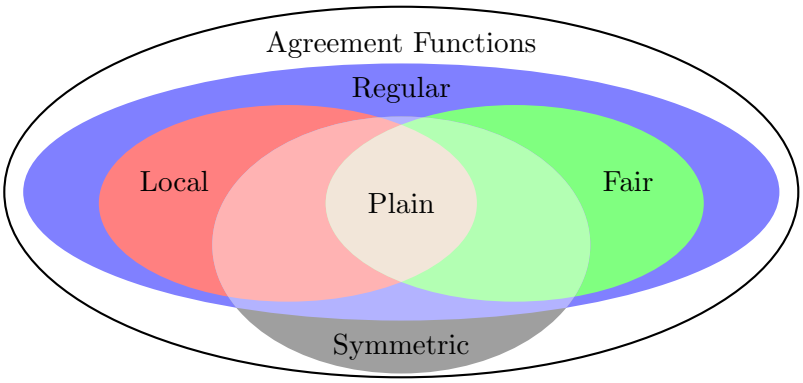


Figure 4.1 – Relations between agreement function properties.

Property 4.3. *A* plain *agrement function is* symmetric.

Let us now show that an agreement function that is either local or fair is also regular.

Property 4.4. *A* fair *agreement function is* regular.

Proof. Consider a model M with a fair agreement function. By its definition we have that $\alpha_M^{\mathcal{F}}(Q' \cap Q, P) = \min(|P \cap Q \cap Q'|, \alpha_M|_P)$ and $\alpha_M^{\mathcal{F}}(Q, P) = \min(|P \cap Q|, \alpha_M|_P)$. But since we have $|P \cap Q \cap Q'| \leq |P \cap Q|$, we obtain that $\alpha_M^{\mathcal{F}}(Q, P) \geq \alpha_M^{\mathcal{F}}(Q' \cap Q, P)$. □

Property 4.5. *A* local *agreement function is* regular.

Proof. Consider a model M with a local agreement function $\alpha_M^{\mathcal{F}}$. By definition we have that $\forall P, Q \subseteq \Pi$, $\alpha_M^{\mathcal{F}}(Q, P) = \alpha_M^{Q \cap P}$. In particular $\alpha_M^{\mathcal{F}}(Q, Q) = \alpha_M^{Q}$, but it is also equal to $\alpha_M^{\mathcal{F}}(Q, Q) = \alpha_M|_Q$ and hence we obtain that $\alpha_M^{\mathcal{F}}(Q, P) = \alpha_M|_{Q \cap P}$. But $\forall Q, Q' \subseteq \Pi$, $\alpha_M|_Q \geq \alpha_M|_{Q \cap Q'}$. Hence, $\alpha_M^{\mathcal{F}}(Q, P) = \alpha_M|_{Q \cap P} \geq \alpha_M|_{P \cap Q \cap Q'} = \alpha_M^{\mathcal{F}}(P \cap Q \cap Q', P \cap Q \cap Q')$ which is equal to $\alpha_M^{P \cap Q \cap P}$ since M is local, and hence, is equal to $\alpha_M^{\mathcal{F}}(Q' \cap Q, P)$. □

These relations between classes of agreement functions are depicted in Figure 4.1. Counter-examples required to show the completeness of Figure 4.1 will be summarized in Section 4.5.

4.3 Fairness through Active Resilience

In this section we are going to investigate the relation between fair agreement functions and active-resilient models. We show that any fair agreement function corresponding to a model is the agreement function of an active resilient model. We call this active resilient model, generalizing the k-active-resilient model, the *α-model*. Moreover, we show that any suffix-closed model or any wait-free model enhanced with shared objects which has a smaller or equal agreement function can solve any task solvable in the α-model.

Let us first formally define the α-model:

Definition 4.9 (α-model)**.** *Given a fair agreement function corresponding to a model, α, the* α-model *is the set of runs in which, the participating set P satisfies: (1) $\alpha(P, P) \geq 1$; and, (2) at most $\alpha(P, P) - 1$ participating processes are faulty, i.e., take only finitely many steps.*

In an α-model, the number of possible failures adapts to the participation of the run. But distinctively from the k-active-resilient models, the participation cannot be arbitrary, i.e., it must be such that $\alpha(P, P) \geq 1$. Still, the k-active-resilient models are specific α-models corresponding to plain agreement functions. Namely, the $(k+1)$-active-resilient model corresponds to the α-model associated with the k-plain agreement function.

Agreement Function of the α-Model

The α-model is defined by using an agreement function α. Let us now check that for any $P, Q \subseteq \Pi$, the α-model does indeed solve the $\alpha(Q, P)$-set-consensus task among Q and under P.

Theorem 4.1. *For all $P, Q \subseteq \Pi$, the task of $\alpha(Q, P)$-set-consensus task among Q and under P is solvable in the α-model.*

Proof. Let us first note that as α is fair, we have that $\alpha(Q, P) = \min(|P \cap Q|, \alpha(P, P))$. If $\alpha(Q, P) = |P \cap Q|$, then the task is trivially solvable by letting processes return their own input. Therefore, let us consider any subsets P and Q of Π such that $\alpha(Q, P) = \alpha(P, P)$. First assume that $\alpha(P, P) = 0$, hence, we can let processes wait to observe a participation that is not a subset of P and then return their own input as a run is valid only if the participation as an associated agreement power greater than 1.

Let us now assume that $\alpha(P, P) \geq 1$. Processes in $\Pi \setminus Q$ can simply return their own input values. For processes in Q, they submit successively their input value to $\alpha(P, P)$ safe-agreement instances. If a process obtains a non-$\perp$ output, then it write it to the shared memory and returns it. If a process obtains $\perp$ from an instance, it moves to the next, and if all were accessed then it waits to see a shared output and returns it or to observe a participation not including P and return its own input.

It is trivial to see that at most $\alpha(P, P)$ distinct outputs can be returned as safe-agreement ensures that a single non-$\perp$ value can be returned. Assume now that no correct process obtains a non-$\perp$ output from any of the $\alpha(P, P)$ safe-agreement instances and that the participation remains a subset of P, i.e., the only condition preventing termination. It implies that in all safe-agreement instance, a proccess accessing it crashed before sharing its non-$\perp$ output. Thus that there are at least $\alpha(P, P)$ crashed process — A contradiction with a participation included in P. □

Task Computability

The following result is instrumental in our characterizations of *fair* models:

Theorem 4.2. *For any task T solvable in an α-model, T is solvable in any read-write shared memory model which solves, for all $P, Q \in \Pi$, the task of $\alpha(Q, P)$-set-consensus among Q and under P.*

Proof. Let us use the solutions to the tasks of $\alpha(Q, P)$-set-consensus among Q and under P in an agreement-based simulation in which the selection function selects to simulate the $\alpha(Q, P)$ least advanced processes. By construction such a simulation ensures that enough processes are selected for simulation. The participation in the simulated run must be such that its associated agreement power is not equal to 0. Moreover, there are at most $\alpha(Q, P)$ participating processes in the simulated run that takes only finitely many steps, hence the

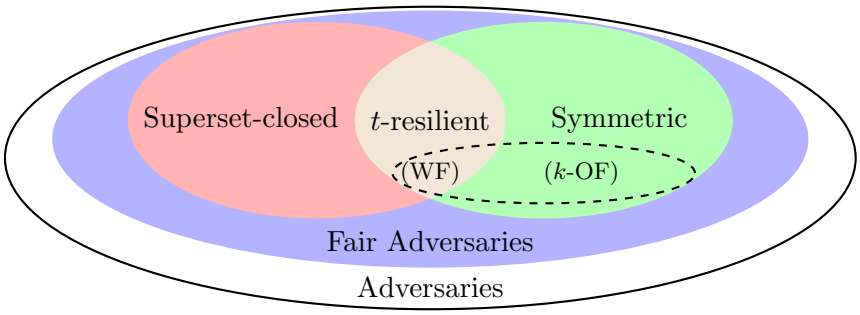


Figure 4.2 – Relations between agreement function properties.

simulated run is a valid run of the α-model. Therefore this simulation can be used to solve any task solvable in the α-model. □

Using Theorem 4.2, we can derive that:

Corollary 4.1. *Let M be any model and T be any task that is solvable in the $\alpha_M^{\mathcal{F}}$-model. Then M solves T.*

4.4 Agreement Functions for Adversaries

Let $\mathcal{A}|_{P,Q} = \{S \in \mathcal{A}, S \subseteq P \wedge S \cap Q \neq \emptyset\}$, then the agreement of an adversary can be computed as follows:

Theorem 4.3. *The agreement function of adversary $\mathcal{A}$ is $\alpha_{\mathcal{A}}(Q,P) = setcon(\mathcal{A}|_{P,Q})$.*

Proof. An algorithm A_P that solves $\alpha_{\mathcal{A}}(Q,P)$-set-consensus among Q and under P, is a straightforward generalization of the result of [GK10]. It is shown in [GK10] that $setcon(\mathcal{A})$-set consensus is the best set-consensus solvable in $\mathcal{A}$. But if we restrict the runs to assume that the processes in $\Pi \setminus P$ do not take a single step, then the set of possible live sets reduces to $\mathcal{A}|_P$. Moreover, if we consider that only processes in Q must output, then the set of possible live sets reduces to $\mathcal{A}|_{P,Q}$ □

It is immediate from Theorem 4.3 that $\mathcal{A} \subseteq \mathcal{A}'$ implies $setcon(\mathcal{A}) \leq setcon(\mathcal{A}')$.

***Fair* adversaries** We propose a class of adversaries which encompasses both classical classes of super-set closed and symmetric adversaries. Informally, an adversary is *fair* if its set consensus power does not change if only a subset of the processes are participating in an agreement protocol.

More precisely, consider $\mathcal{A}$-compliant runs with participating set P and assume that processes in $Q \subseteq P$ want to reach agreement *among themselves*: only these processes propose inputs and are expected to produce outputs. We can only guarantee outputs to processes in Q when the set of correct processes include some process in Q, i.e., when the current live set intersect with Q. Thus, the best level of set consensus reachable by Q is defined the set consensus power of adversary $\mathcal{A}|_{P,Q} = \{S \in \mathcal{A}|_P, S \cap Q \neq \emptyset\}$, unless $|Q| < setcon(\mathcal{A}|_P)$.

Definition 4.10. *[Fair adversary] An adversary $\mathcal{A}$ is fair if and only if:*

$$\forall P \subseteq \Pi, \forall Q \subseteq P, setcon(\mathcal{A}|_{P,Q}) = min(|Q|, setcon(\mathcal{A}|_P)).$$

Property 4.6.

$$setcon(\mathcal{A}|_{P,Q}) \leq \min(|Q|, setcon(\mathcal{A}|_P))$$

Proof. For any $P \subseteq \Pi$ and $Q \subseteq P$, $\mathcal{A}|_{P,Q} = \{S \in \mathcal{A}|_P, S \cap Q \neq \emptyset\}$ is a subset of $\mathcal{A}|_P$ and, thus, $setcon(\mathcal{A}|_{P,Q}) \leq setcon(\mathcal{A}|_P)$. Moreover, $setcon(\mathcal{A}|_{P,Q}) \leq |Q|$, as $|Q|$-set consensus can be solved in $\{S \in \mathcal{A}|_P, S \cap Q \neq \emptyset\}$ as follows: every process waits until some process in Q writes its input and decides on it. □

Theorem 4.4. *Any superset-closed adversary is fair.*

Proof. Suppose that there exists a superset-closed adversary $\mathcal{A}$ that is not *fair*, i.e., by Property 4.6, $\exists P \subseteq \Pi, \exists Q \subseteq P, setcon(\{S \in \mathcal{A}|_P, S \cap Q \neq \emptyset\}) < min(|Q|, setcon(\mathcal{A}|_P))$. Clearly $\mathcal{A}|_P$ and $\mathcal{A}|_{P,Q}$ are also superset-closed and, thus, $setcon(\mathcal{A}|_P) = csize(\mathcal{A}|_P)$ and $setcon(\mathcal{A}|_{P,Q}) = csize(\mathcal{A}|_{P,Q})$.

Since $setcon(\mathcal{A}|_{P,Q}) < |Q|$, a minimal hitting set H' of $\mathcal{A}|_{P,Q}$ is such that $|H'| < |Q|$, and therefore there exists a process $q \in Q$, $q \notin H'$. Also, since $setcon(\mathcal{A}|_{P,Q}) < setcon(\mathcal{A}|_P)$, H' is not a hitting set of $\mathcal{A}|_P$. Thus, there exists $S \in \mathcal{A}|_P$ such that $S \cap H' = \emptyset$. Hence, $(S \cup \{q\}) \cap H' = \emptyset$. Since $\mathcal{A}|_P$ is superset closed, we have $S \cup \{q\} \in \mathcal{A}|_P$ and, since $q \in Q$, $S \cup \{q\} \in \mathcal{A}|_{P,Q}$. But $(S \cup \{q\}) \cap H' = \emptyset$—a contradiction with H' being a hitting set of $\mathcal{A}|_{P,Q}$. □

Theorem 4.5. *Any symmetric adversary is fair.*

Proof. The set consensus power of a generic adversary $\mathcal{A}$ is defined recursively through finding $S \in \mathcal{A}$ and $p \in S$ which max-minimize the set consensus power of $\mathcal{A}|_{S\setminus\{p\}}$. Let us recall that if $\mathcal{A} \subseteq \mathcal{A}'$ then $setcon(\mathcal{A}) \leq setcon(\mathcal{A}')$. Therefore, S can always be selected to be *locally maximal*, i.e., such that there is no live set in $S' \in \mathcal{A}$ with $S \subsetneq S'$.

Suppose by contradiction that $\mathcal{A}$ is symmetric but not *fair*, i.e., by Property 4.6, for some $P \subseteq \Pi$ and $Q \subseteq P$, $setcon(\mathcal{A}|_{P,Q}) < min(|Q|, setcon(\mathcal{A}|_P))$. We show that if the property holds for P and Q such that $\mathcal{A}|_{P,Q} \neq \emptyset$ then it also holds for some $P' \subsetneq P$ and $Q' \subseteq Q$.

First, we observe that $|Q| > 1$, otherwise $setcon(\mathcal{A}|_{P,Q}) = 0$ and, thus, we have $\mathcal{A}|_{P,Q} = \emptyset$.

Since $\mathcal{A}$ is symmetric, $\mathcal{A}|_P$ is also symmetric. Thus, for every $S \in \mathcal{A}|_P$ and $p \in S$ such that $setcon(\mathcal{A}|_P) = 1 + setcon(\mathcal{A}|_{S\setminus\{p\}})$, any S' such that $|S'| = |S|$ and for any $p' \in S'$, we also have $setcon(\mathcal{A}|_P) = 1 + setcon(\mathcal{A}|_{S'\setminus\{p'\}})$. Since we can always choose S to be a maximal set, we derive that the equality holds for every maximal set S in $\mathcal{A}|_P$ and every $p \in S$.

Let us recall that, by the definition of *setcon*, there exists $L \in \mathcal{A}|_{P,Q}$ and $a \in L$ such that $setcon(\mathcal{A}|_{P,Q}) = 1 + setcon((\mathcal{A}|_{P,Q})|_{L\setminus\{a\}}) = setcon(\mathcal{A}|_{L,Q})$. Since $\mathcal{A}|_P$ is symmetric, for all L', $|L'| = |L|$ and $L \cap Q \subseteq L' \cap Q$, we have $setcon(\mathcal{A}|_{L',Q}) \geq setcon(\mathcal{A}|_{L,Q})$. Indeed, modulo a permutation of process identifiers, $\mathcal{A}|_{L',Q}$ contains all the live sets of $\mathcal{A}|_{L,Q}$ plus live sets in $\mathcal{A}|_{L'}$ that overlap with $(L' \cap Q) \setminus (L \cap Q)$. Since $setcon(\mathcal{A}|_{L,Q}) = setcon(\mathcal{A}|_{P,Q})$ and $L' \in \mathcal{A}|_{P,Q}$, we have $setcon(\mathcal{A}|_{L',Q}) = setcon(\mathcal{A}|_{L,Q})$. Therefore, for any $a \in L'$, $setcon(\mathcal{A}|_{L'\setminus\{a\},Q}) < setcon(\mathcal{A}|_{L'\setminus\{a\}})$.

In particular, for L' with $L' \cap Q \in \{L', Q\}$, $setcon(\mathcal{A}|_{L',Q}) = setcon(\mathcal{A}|_{L,Q})$. Note that $L' \not\subseteq Q$, otherwise, $\mathcal{A}|_{L',Q} = \mathcal{A}|_{L'}$ and, thus, $setcon(\mathcal{A}|_{L',Q}) = setcon(\mathcal{A}|_{L'}) = setcon(\mathcal{A}|_P)$, contradicting our assumption.

Thus, let us assume that $Q \subsetneq L'$. Note that $Q' = Q \setminus \{a\} \subsetneq L' \setminus \{a\}$, and since $|Q| \geq 2$, $Q' \neq \emptyset$, we have $setcon(\mathcal{A}|_{P',Q'}) < setcon(\mathcal{A}|_{P'})$ for $P' = L' \setminus \{a\}$ and $Q' \subseteq P'$, $Q' \neq \emptyset$. Furthermore, since $setcon(\mathcal{A}|_{P,Q}) < |Q|$, we have $setcon(\mathcal{A}|_{P',Q'}) < |Q'|$.

By applying this argument inductively, we end up with a live set P and $Q \subseteq P$ such that $setcon(\mathcal{A}|_P) \geq 1$, $Q \neq \emptyset$ and $setcon(\mathcal{A}|_{P,Q}) = 0$. By the definition of *setcon*, $\mathcal{A}|_P \neq \emptyset$ and $\mathcal{A}|_{P,Q} = \emptyset$. But $\mathcal{A}|_P$ is symmetric and $Q \neq \emptyset$, so for every $S \in \mathcal{A}|_P$, there exists $S' \in \mathcal{A}|_P$ such that $|S| = |S'|$ and $S' \cap Q \neq \emptyset$, i.e., $\mathcal{A}|_{P,Q} \neq \emptyset$—a contradiction. □

Note that not all adversaries are fair. Let us consider, for example, the adversary $\mathcal{A} = \{\{p_1\}, \{p_2, p_3\}, \{p_1, p_2, p_3\}\}$ is not fair. On the other hand, not all fair adversaries are either super-set closed or symmetric. For example, the adversary $\mathcal{A} = 2^{\{p_1,p_2,p_3\}} \setminus \{p_1, p_2\}$ is fair but is neither symmetric not super-set closed. Understanding what makes an adversary fair is an interesting challenge.

Task computability in adversarial models

In this part, we show that the task computability of a adversarial $\mathcal{A}$-model is fully grasped by its associated agreement function $\alpha^{\mathcal{F}}_{\mathcal{A}}$.

For this, we use the agreement-based simulation again. The goal is to show that the task computability of an adversary is grasped by its agreement function. This is done by simulation an adversary using its ability to solve k-set-consensus tasks among some Q and under some P. The simulation is not too much complicated, it consists on simulating a live-set that contains a non-terminated process if any. The structure of the adversary and its link with the *setcon* function is the main tool behind the simulation.

In the simulation, up to $\alpha^{\mathcal{F}}_{\mathcal{A}}(Q, P)$ simulators execute the given algorithm solving T, where Q is the number of non-terminated processes and P is the participating set of the current run. Note that the simulated processes correspond to the simulators in this simulation. We adapt the currently simulated live set to include processes not yet provided with a task output, and ensure that the chosen live set is simulated sufficiently long until some active processes are provided with outputs of T. The simulation terminates as soon as all correct processes are provided with outputs.

The simulation is defined by the **selectOperationList** function that must select processes to simulate. It consists of a recursive selection of first a live set and then a process in it.

The selection goes as follows. The first live set to be selected is a live set S_1 with non-terminated processes with the highest associated agreement power. Then, the least advanced process p_1 is selected to be simulated for the first choice. This pair (S_1, p_1) then drives the choice for other selected processes to simulate. It is done recursively in the same way by selecting a live set with non-terminated processes in $\mathcal{A}|_{S_1 \setminus \{p_1\}}$ the set of live sets that are subsets of S_1 but which does not include p_1. Hence, we can select a live set S_2 with the highest associated agreeement power and its least advance process in the simulation p_2. We repeat this process iteratively until the set of live sets in the restriction becomes empty.

Correctness of the simulation. The fact that the **selectOperationList** returns sufficiently many processes to simulate directly follows from the definition of the *setcon* function. Hence, the simulation ensures that infinitely many selected processes makes progress in the simulation as long as there is a non-terminated simulator. Let us show that this corresponds to a live set including a non-terminated simulated process.

For this, consider the simulation after all simulated process terminated in the given run. This implies that the number of selected live sets becomes stable and all include a non-terminated process. Condider the highest level for which the simulation makes progress. As the selection of live sets is deterinistic, the previous level selected the same process each time (as eventually the selected process never makes progress). This implies that the selected live set for which the selected process makes progress is the stable. The least advanced process in this live set is selected, hence, all processes in this live set must proceed to infinitely many steps. Higher levels do not make any progress and lower levels simulates processes in this live set, hence, the set of live processes is exactly equal to this live set. This live set include a non-terminated process by construction and therefore we reach a contradiction as all process in this live set must terminate.

4.5 Shared-Memory Models and Agreement Functions

Adversaries are regular.

Property 4.7. *The agreement function of an adversary is regular.*

Proof. Consider sets of processes P and Q, and assume that the participating set is $P \cup Q$. Now consider the following protocol: (1) processes in P solve an $setcon(\mathcal{A}|_P)$ set-consensus algorithm by assuming that processes in $Q \setminus (P \cap Q)$ have failed; (2) processes in $Q \setminus (P \cap Q)$ return directly their proposal; (3) processes also return any value decided by some process. A process terminate as a process in Q is correct and some process returns according to (2), or else every process in Q are faulty and thus (1) terminate. Moreover, it is easy to see that at most $setcon(\mathcal{A}|_P)$ distinct values can be returned by (1) and at most $|Q \setminus (P \cap Q)|$ values can be returned by (2). Thus we can derive that $\alpha_{\mathcal{A}}(P \cup Q) \leq \alpha_{\mathcal{A}}(P) + |Q \setminus (P \cap Q)|$. □

Regular vs. other classes. Symmetric models on the other hand are not necessarily regular. Indeed, consider the 3-process model in which the first process to participate proceeds to at least 2 update and snapshot operations before any other process takes steps. Thus, all processes can identify this first process. In this model, it is easy to see that as long as the participation P is distinct from Π, then processes can solve the 1-set consensus task among Q and P, and cannot solve better if $P \cap Q \neq \emptyset$. The agreement power among a single process under Π is 1, such as among Π, as processes can return the proposal from the first process to participate. But if Q is of size 2, then only 2-set consensus among Q and under Π can be solved. Indeed, the process in $\Pi \setminus Q$ might be the first to participate and may crash before the other two execute an arbitrary 2-process wait-free run. But the 2-process wait-free model cannot solve 1-set consensus. One can check that this model is indeed symmetric but is not regular, since for a participation equal to Π, the argeement power among 3 processes is smaller than the agreement power among 2.

Hence these relations can be summarized as follows:

Property 4.8. *A model which is fair, local is a regular model. But not all symmetric models are regular.*

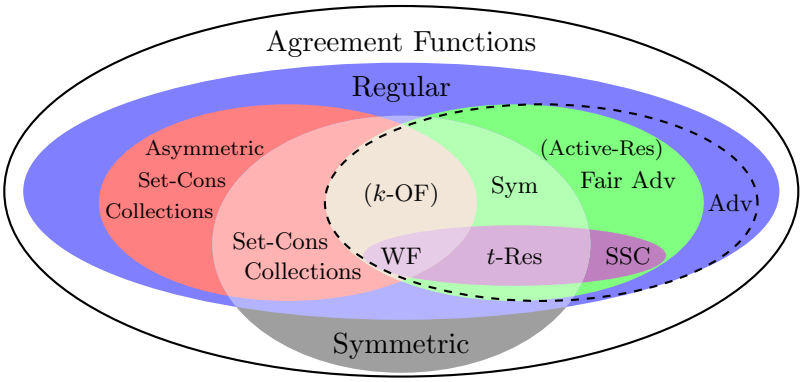


Figure 4.3 – Integration of adversary families, active resilience and collection of set-consensus objects inside a representation of agreement function classes.

Chapter 5

Combinatorial Topology

The use of combinatorial topology in distributed computing was first introduced in [BG93a, HS99, SZ00] to show the impossibility of k-set consensus in a $k+1$-process wait-free model. Since then, the use of topological methods in distributing computing has evolved to the point where a book detailing the links between the two fields was published [HKR14]. The reader should refer to this book for more details about distributed computing results and models derived from combinatorial topology. In this chapter, we give a brief overview of the notions from combinatorial topology that are used in this thesis and in some related work.

5.1 Simplicial Complexes

A configuration of a distributed system is a collection of processes and object local sates. But in general, no process is aware of the global state of the system and processes can only obtain outdated information about other processes states. Hence, a configuration can be represented as a set of processes local states where processes are only aware of their own state. But a process may have the same local state in distinct global configurations. The set of possible configurations of the system is what is called a simplicial complex in combinatorial topology.

Let us now recall several notions from combinatorial topology. For more detailed coverage of the topic, please refer to [Spa66, HKR14].

Simplicial complex. A *simplicial complex* is a set V, together with an *inclusion-closed* collection $\mathcal{K}$ of finite non-empty subsets of V such that:

1. For any $v \in V$, the one-element set $\{v\}$ is in $\mathcal{K}$;
2. If $\sigma \in \mathcal{K}$ and $\sigma' \subseteq \sigma$, then $\sigma' \in \mathcal{K}$.

The elements of V are called *vertices*, and the elements of $\mathcal{K}$ are called *simplices*. We usually drop V from the notation and refer to the simplicial complex as $\mathcal{K}$ directly. Indeed, we can extract from $\mathcal{K}$ the set of vertices composing it. We denote as $Vert(\mathcal{K})$ the set of vertices of $\mathcal{K}$. A simplicial complex $\mathcal{K}$ is *finite* if the collection $\mathcal{K}$ is finite. For simplicity, we will assume that our complexes are finite.

The *dimension* of a simplex σ, denoted $\dim(\sigma)$, is its cardinality minus one, i.e., $\#(\sigma) - 1$ (the use of $|.|$ will be avoided since it is traditionally used for the geometrical representation of a simplex). Any subset of a simplex σ is also a simplex and is called a *face* of σ. We denote

as $faces(\sigma)$ the set containing all faces of σ. Given a complex $\mathcal{K}$ and a simplex $\sigma \in \mathcal{K}$, σ is a *facet* of $\mathcal{K}$, denoted $facet(\sigma, \mathcal{K})$, if σ is not the face of any strictly larger simplex in $\mathcal{K}$. Let $facets(K) = \{\sigma \in \mathcal{K}, facet(\sigma, K)\}$. The dimension of a complex is equal to the maximal dimension of the simplices composing it.

A *sub-complex* of $\mathcal{K}$ is a subset of $\mathcal{K}$ that is also a simplicial complex. A simplicial complex $\mathcal{K}$ is called *pure* of dimension n if $\mathcal{K}$ has no simplices of dimension $> n$, and every k-dimensional simplex of $\mathcal{K}$ (for $k < n$) is a face of an n-dimensional simplex of $\mathcal{K}$. Hence, equivalently, a simplicial complex $\mathcal{K}$ is pure of dimension n if all its facets are of dimension n.

Chromatic complexes. We now turn to the chromatic complexes used in distributed computing. Fix $n \geq 0$. The *standard n-simplex* $\mathbf{s}^n$ has $n+1$ vertices, in one-to-one correspondence with $n+1$ *colors* $0, 1, \ldots, n$. A face $\mathbf{t}$ of $\mathbf{s}$ is specified by a collection of vertices from $\{0, \ldots, n\}$. We view $\mathbf{s}^n$ as a complex, with its simplices being all possible faces $\mathbf{t}$.

A *chromatic complex* is a simplicial complex $\mathcal{K}$ together with a non-collapsing simplicial map $\chi : \mathcal{K} \to \mathcal{C}$, $\mathcal{C}$ being a set of colors. See the following section for formal definitions about simplicial maps, but informally, it corresponds to associating colors to any vertex of a *chromatic complex* such that all vertices of the same simplex have distinct associated colors. Note that therefore, $\mathcal{K}$ can have dimension at most $\#(\mathcal{C}) - 1$. We usually drop χ from the notation and consider that vertices are couples (v, c) where v is the vertex and c its associated color. We write $\chi(\mathcal{K})$ for the union of $\chi(v)$ over all vertices $v \in Vert(\mathcal{K})$. Note that if $\mathcal{K}' \subseteq \mathcal{K}$ is a sub-complex of a chromatic complex, it inherits a chromatic structure by restriction. In particular, the standard n-simplex $\mathbf{s}^n$ is a chromatic complex, with χ being the identity map.

In our setting, colors correspond by default to processes identifiers. In this case, the set of colors of a complex is equal to $\chi(\mathbf{s}^n)$. Since most of the time the size of the system is fixed and known from the context, we use $\mathbf{s}$ to denote the standard $(\#(\Pi) - 1)$-simplex. Note that, when colors correspond to processes identifiers, we use the map χ to obtain both the color of a vertex and the process corresponding to the identifier.

5.2 Basic Operations

Let us now recall some additional notions which apply to simplicial complexes.

Maps. Let $\mathcal{K}$ and $\mathcal{L}$ be simplicial complexes. A simplicial map $f : \mathcal{K} \to \mathcal{L}$ is a function from $\mathcal{K}$ to $\mathcal{L}$ such that for any face θ of a simplex $\sigma \in \mathcal{K}$, then $f(\theta)$ is a face of $f(\sigma)$ in $\mathcal{L}$. A simplicial map is said to be non-collapsing if for any strict face θ of a simplex $\sigma \in \mathcal{K}$, then $f(\theta)$ is a strict face of $f(\sigma)$. Hence, the image of an m-dimensional simplex through a non-collapsing map is also an m-dimensional simplex. Let $\mathcal{K}$ and $\mathcal{L}$ be chromatic complexes. A simplicial map $f : \mathcal{K} \to \mathcal{L}$ is *color-preserving*, also called a *chromatic map*, if for all vertices $v \in Vert(\mathcal{K})$, we have $\chi(v) = \chi(f(v))$. Note that a color-preserving map is automatically non-collapsing.

A vertex map $g : \mathcal{K} \to \mathcal{L}$ is a function from $Vert(\mathcal{K})$ to $Vert(\mathcal{L})$. Any simplicial map induces a vertex map by restricting it to the vertices. For a vertex map to produce a simplicial map, the union of the image of vertices from a simplex in $\mathcal{K}$ must form a simplex in $\mathcal{L}$. When it is the case, the image of a simplex is just the union of the images of its vertices.

A carrier map $\Psi : \mathcal{K} \to 2^{\mathcal{L}}$ sends simplices to sub-complexes such that a face θ of a simplex $\sigma \in \mathcal{K}$ is sent to a complex $\Psi(\theta)$ which is a sub-complex of $\Psi(\sigma)$. A simplicial map ϕ is carried by the carrier map Ψ if $\phi(\sigma) \in \Psi(\sigma)$ for every simplex σ in its domain.

Standard Construction. The *k-skeleton* of a complex $\mathcal{K}$ is the set of simplices of $\mathcal{K}$ of dimension smaller than or equal to k. Note that the skeleton is also a simplicial complex of dimension k or of the dimension of $\mathcal{K}$. Moreover, if $\mathcal{K}$ is a pure simplicial complex, then its skeleton is also pure.

The *closure* of a set of simplices S, denoted $Cl(S)$, is the complex formed by all faces of simplices in S, i.e., $\bigcup_{\sigma \in S} faces(\sigma)$. Given a complex $\mathcal{K}$, the *open star* of $S \subseteq \mathcal{K}$ in $\mathcal{K}$, denoted $St^o(S, \mathcal{K})$, is the set of all simplices in $\mathcal{K}$ having a simplex from S as a face, i.e., $\{\sigma \in \mathcal{K} | faces(\sigma) \cap S \neq \emptyset\}$. Note that, in general, the open star is not a complex. The *closed star* of $S \subseteq \mathcal{K}$ in $\mathcal{K}$, denoted $St^c(S, \mathcal{K})$, is simply the closure of the open star of S in $\mathcal{K}$, i.e., $Cl(St^o(S, \mathcal{K}))$. The *star* is used sometimes to refer to the *open star* and sometimes to refer to the *closed star*. Hence, to avoid confusion, we are going to avoid using the notion of a *star* without additional precisions. The *link* of $S \subseteq \mathcal{K}$ in $\mathcal{K}$, denoted $Lk(S, \mathcal{K})$, is the difference between the closed star of S and the open star of the closure of S, i.e., $St^c(S, \mathcal{K}) \setminus (St^o(Cl(S), \mathcal{K}))$.

Given a pure complex $\mathcal{K}$, we also define a new construct that we call the *pure complement* of $S \subseteq \mathcal{K}$ in $\mathcal{K}$, $Pc(S, \mathcal{K})$. It is the maximal pure sub-complex of $\mathcal{K}$ of the same dimension as $\mathcal{K}$ which does not intersect with S, i.e., $Pc(S, \mathcal{K}) = Cl(\{\sigma \in facets(\mathcal{K}) | faces(\sigma) \cap S = \emptyset\})$.

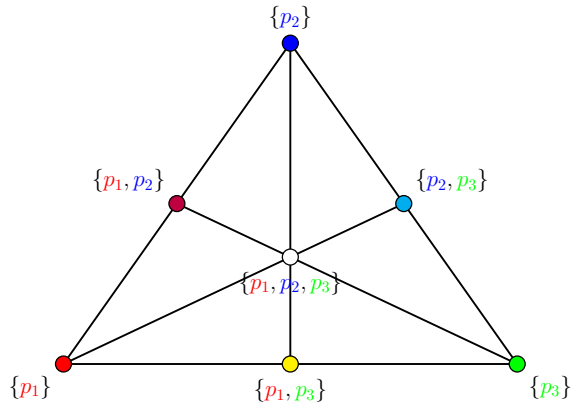
Continous representation. We can associate a simplicial complex $\mathcal{K}$ with a topological space $|\mathcal{K}|$, called its geometrical realization. The geometrical realization is defined incrementally: first, each vertex of $\mathcal{K}$ is associated with points in $[0, 1]^{\dim(\mathcal{K})}$ such that vertices from the same simplex are associated with affinely independent positions; then the geometric realization of a simplex is equal to the convex-hull of its vertices.

In continuous spaces, a *homeomorphism* is a bijective continuous function with a continuous inverse between topological spaces. Two spaces are said to be homeomorphic if there exists a homeomorphism from one space to the other. Note that a k-dimensional simplex is therefore associated with a k-dimensional closed space homeomorphic to the k-disk. The k-disk corresponds to the set of points of a k-dimensional space with a distance to a center point smaller or equal to 1. The k-sphere is the set of points of a k-dimensional space at a distance of exactly 1 to a center point.

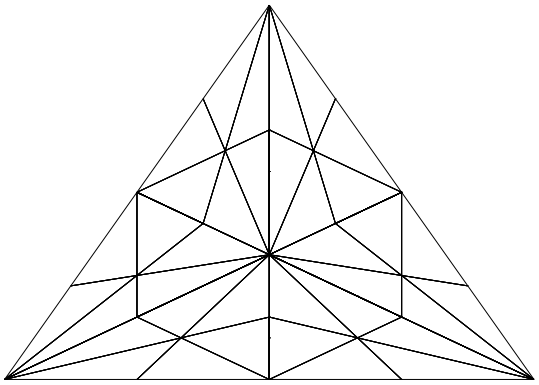
Note that given a simplicial map $f : \mathcal{K} \to \mathcal{L}$, we can extend it linearly to obtain a continuous map $|f| : |\mathcal{K}| \to |\mathcal{L}|$: a point $p \in |\mathcal{K}|$ is a linear combination of the vertices from $\mathcal{K}$ and its image in $|f|$ is the same linear combination of the images of the vertices.

Connectivity. Using the continuous representation of simplicial complexes, the notion of path connectivity generalizes to higher dimensions. A simplicial complex $\mathcal{K}$ is *simply connected* or 0-connected if for any couple of points $p, p' \in |\mathcal{K}|$ there exists a path in $|\mathcal{K}|$ from p to p'. Two points correspond to a 0-sphere and a path between the two points correspond to a 1-disk. Hence, a simplicial complex $\mathcal{K}$ is said to be *m-connected* if any continuous function from the $(m-1)$-sphere to $|\mathcal{K}|$ can be extended to a continuous map from the m-disk to $|\mathcal{K}|$.

A related notion which is particularly useful for colored simplices is the notion of *link-connectivity*. A complex $\mathcal{K}$ is *link-connected* if the link of any simplex $\sigma \in \mathcal{K}$, $Lk(\sigma, \mathcal{K})$, is $(\dim(\mathcal{K}) - 2 - \dim(\sigma))$-connected. A link-connected complex is necessarily pure. Intuitively, a pure complex is link-connected if it is locally "thick".



(a) Single iteration: $Bar(\mathbf{s})$.



(b) Second iteration: $Bar^2(\mathbf{s})$.

Figure 5.1 – First and second iteration of the 2-dimensionnal barycentric subdivision of $\mathbf{s}$.

5.3 Subdivisions

An important notion in combinatorial topology and more specifically for its application to distributed computing is the notion of *subdivision*. A subdivision of a simplicial complex $\mathcal{K}$, is a simplicial complex $Sub(\mathcal{K})$ such that: (1) the geometrical realization of any simplex from $Sub(\mathcal{K})$ is included in the geometrical realization of a simplex from $\mathcal{K}$; and (2) the geometrical realization of a simplex in $\mathcal{K}$ is the union of geometric realizations of simplices of $Sub(\mathcal{K})$. Note that the subdivision of a subdivision of $\mathcal{K}$ is by definition of subdivision of $\mathcal{K}$.

The *diameter* of a simplex σ of dimension k is the smallest diameter of a k-disk embedding $|\sigma|$. The *mesh* of a complex is the maximal diameter of any of its simplices. A subdivision $Sub(\mathcal{K})$ is said to be *mesh-shrinking* if the mesh of $Sub(\mathcal{K})$ is strictly smaller than the mesh of $\mathcal{K}$.

Barycentric subdivision. Every complex $\mathcal{K}$ has a *barycentric subdivision*, denoted $Bar(\mathcal{K})$. Its vertices are the barycenters of simplices of $\mathcal{K}$. For each set of simplices of $\mathcal{K}$ related by inclusion, the set of associated barycenters forms a simplex. The barycentric subdivision is very convenient as its definition is straightforward while being mesh-shrinking. Moreover, it can be applied iteratively on any complex. By applying k times the barycentric subdivision on a complex $\mathcal{K}$, we obtain the k^{th} barycentric subdivision, $Bar^k(\mathcal{K})$. Hence, we obtain a subdivision with an arbitrarily small mesh by just applying the barycentric subdivision sufficiently many times. But the complexes constructed by the barycentric subdivision are not chromatic. The first and second iterations of the 2-dimensional barycentric subdivisions are displayed in Figure 5.1.

Standard chromatic subdivision. Every chromatic complex $\mathcal{K}$ has a *standard chromatic subdivision* $\mathrm{Chr}\,\mathcal{K}$. Let us first define $\mathrm{Chr}\,\mathbf{s}$ for the standard simplex $\mathbf{s}$. The vertices of $\mathrm{Chr}\,\mathbf{s}$ are pairs $(i, \mathbf{t})$, where $i \in \{0, \ldots, n\}$ and $\mathbf{t}$ is a face of $\mathbf{s}$ containing i. We let $\chi(i, \mathbf{t}) = i$. Further, $\mathrm{Chr}\,s$ is characterized by its n-simplices; they are the $(n+1)$-tuples $((0, \mathbf{t}_0), \ldots, (n, \mathbf{t}_n))$ such that:

(a) For all $\mathbf{t}_i$ and $\mathbf{t}_j$, one is a face of the other;

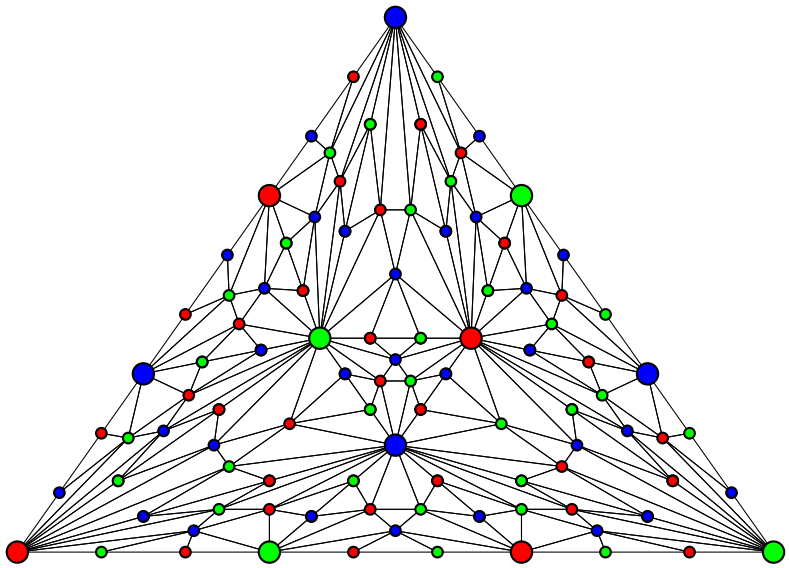


Figure 5.2 – Second iteration of the 2-dimensionnal standard chromatic subdivision of $\mathbf{s}$.

(b) If $j \in \mathbf{t}_i$, then $\mathbf{t}_j \subseteq \mathbf{t}_i$.

The geometric realization of $\mathbf{s}$ can be taken to be the set $\{\mathbf{x} = (x_0, \ldots, x_n) \in [0,1]^{n+1} \mid \sum x_i = 1\}$, where the vertex i corresponding to the point $\mathbf{x}^i$ with the i coordinate 1 and all other coordinates 0. Then, we can identify a vertex $(i, \mathbf{t})$ of $\operatorname{Chr}\mathbf{s}$ with the point

$$\frac{1}{2k-1}\mathbf{x}_i + \frac{2}{2k-1}\Big(\sum_{\{j \in \mathbf{t} \mid j \neq i\}} \mathbf{x}_j\Big) \in |\mathbf{s}| \subset \mathcal{R}^{n+1},$$

where k is the cardinality of $\mathbf{t}$. The standard chromatic subdivision, $\operatorname{Chr}\mathbf{s}$, is illustrated for a 3-process system in Figure 1.1c from Chapter 1.

Next, given a chromatic complex $\mathcal{K}$, we let $\operatorname{Chr}\mathcal{K}$ be the subdivision of $\mathcal{K}$ obtained by replacing each simplex in $\mathcal{K}$ with its chromatic subdivision. Thus, the vertices of $\operatorname{Chr}\mathcal{K}$ are pairs (p, σ), where p is a vertex of $\mathcal{K}$ and σ is a simplex of $\mathcal{K}$ containing p. If we iterate this process m times, we obtain the m^{th} chromatic subdivision, $\operatorname{Chr}^m\mathcal{K}$.

It has been shown formally by Kozlov in [Koz12] that Chr is indeed a *subdivision*. In particular, the geometric realization of $\operatorname{Chr}\mathbf{s}$, $|\operatorname{Chr}\mathbf{s}|$, is homeomorphic to $|\mathbf{s}|$, the geometric realization of $\mathbf{s}$ (i.e., the convex hull of its vertices).

If we *iterate* this subdivision m times, each time applying Chr to all simplices, we obtain the m^{th} chromatic subdivision, Chr^m. The second iteration of the 2-dimensional standard chromatic subdivision is depicted in Figure 5.2. $\operatorname{Chr}^m\mathbf{s}$ precisely captures the m-round IIS model, IS^m [BG97, HS99].

Carriers. Given a complex $\mathcal{K}$ and a subdivision of it, $Sub(\mathcal{K})$, the carrier of a simplex $\sigma \in Sub(\mathcal{K})$ in $\mathcal{K}$, $carrier(\sigma, \mathcal{K})$, is the smallest simplex $\rho \in \mathcal{K}$ such that the geometric realization of σ, $|\sigma|$, is contained in $|\rho|$: $|\sigma| \subseteq |\rho|$. The carrier of a vertex $(p, \sigma) \in \operatorname{Chr}\mathbf{s}$ is σ. In the matching IS task, the carrier corresponds to the snapshot returned by p, i.e., the set of processes *seen* by p. The carrier of a simplex $\rho \in \operatorname{Chr}\mathcal{K}$ is just the union (or, due to inclusion,

the maximum) of the carriers of vertices in ρ. Given a simplex $\sigma \in \mathrm{Chr}^2\,\mathbf{s}$, $carrier(\sigma, \mathbf{s})$ is equal to $carrier(carrier(\sigma, \mathrm{Chr}\,\mathbf{s}), \mathbf{s})$. $carrier(\sigma, \mathrm{Chr}\,\mathbf{s})$ corresponds to the set of all snapshots seen by processes in $\chi(\sigma)$. Hence, $carrier(\sigma, \mathbf{s})$ corresponds to the union of all these snapshots. Intuitively, it results in the set of all processes *seen* by processes in $\chi(\sigma)$ through the two successive immediate snapshots instances.

Simplicial approximation and convergence algorithm. A fundamental result in algebraic topology is that continuous mappings can be approximated by simplicial maps over a sufficiently fined-grained subdivision of the topological space. It provides a relation between simplicial complexes and their geometrical realization that can be very useful. Indeed, while some issues might be more tractable in the discrete simplicial setting, some are easier to tackle in the continuous environment. Formally, the simplicial approximation states the following:

Theorem 5.1. *[Simplicial approximation] Given two simplicial complexes $\mathcal{L}$ and $\mathcal{K}$ and a continuous (carrier preserving) function F from $|\mathcal{L}|$ to $|\mathcal{K}|$, then for any mesh-shrinking subdivision Sub, there exists an $\ell \in \mathbb{N}$ such that there exists a simplicial (carrier preserving) map from $Sub^{\ell}(\mathcal{L})$ to $\mathcal{K}$* approximating *F.*

Note that f is approximating F if the continuous extension of f, $|f|$, maps any point $x \in |\mathcal{L}|$ to a point of $|\mathcal{K}|$ with the same carrier as $F(x)$ in $\mathcal{K}$.

Using the simplicial approximation theorem, one can construct a continuous map with some desired properties between geometrical realization and then derive a simplicial map approximating it while preserving many of the specific requirements. The other direction is also possible as one can easily produce a continuous map matching a simplicial map by merely taking its continuous extension $|f|$.

While the simplicial approximation can be useful in the distributed computing setting, it presents one major drawback limiting its use: it does not provide color-preserving maps when used in colored simplicial complexes. In general, providing color-preserving approximations is unfortunately impossible. But given some additional constraints on the simplicial complexes, it has been shown that the simplicial approximation can be extended to color-preserving maps. A distributed algorithm solving this issue was sketched in [BG97] and was recently formalized and proven by Saraph et al. [SHG18]. It requires that the image of the continuous function $F(|\mathcal{L}|)$ is included in the geometric realization of a pure link-connected sub-complex of $\mathcal{K}$ of the same dimension as $\mathcal{L}$. While this restriction is rather substantial, it allows to extend the application of the simplicial approximation to relevant cases in distributed computing such as done in [SHG16].

5.4 Characterization of the Wait-Free Model

The characterization of wait-free task solvability proposed in [HS99] reduces the search of a task solution to the existence of a simplicial map from a subdivision of the input complex to the output complex which satisfies the task specification. But subdividing the input complex can be seen as the ability to solve a task itself, called *simplex agreement tasks*.

Hence, one direction of the proof of the ACT consists in showing that simplex agreement tasks are solvable while the other direction consists in showing that any protocol can be reduced to a subdivision of the initial configuration.

Simplex agreement task. In the *simplex agreement task*, processes start on vertices of some complex $\mathcal{K}$ forming a simplex $\sigma \in \mathcal{K}$, and they must output vertices of some subdivision of $\mathcal{K}$, $Sub(\mathcal{K})$, so that outputs constitute a simplex ρ of $Sub(\mathcal{K})$ respecting carrier inclusion, i.e., $carrier(\rho, \mathcal{K}) \subseteq \sigma$.

Such tasks are primordial for the proof of the asynchronous computability theorem (ACT) [HS99]. Indeed, given a simplicial map from a subdivision of the task input complex solving the task, processes must manage first to solve the simplex agreement task on the given subdivision to be able to apply the task solution provided by the simplicial map. In the original version of the ACT, a map could be given for an arbitrary subdivision. But using the equivalence with the IIS model, it was shown that we could improve the result by considering only iterations of the standard chromatic subdivision [BG97].

Asychronous computability theorem. Restricting the original Asynchronous Computability Theorem (ACT) by considering only iterations of the standard chromatic subdivision yields in this following formulation of the ACT:

Theorem 5.2. *[Asynchronous Computability Theorem (ACT)] A task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is solvable in the wait-free model if and only if there exists a natural number ℓ and a carrier and color preserving simplicial map $\phi : \mathrm{Chr}^{\ell}(\mathcal{I}) \to \mathcal{O}$ carried by Δ (informally, respecting the task specification Δ).*

Note that ACT implies that solutions to a task are *bounded* in the sense that given a task, there must exist a number of iterations after which a solution to the task always exists, whatever the corresponding execution is. Note that this is not true in general as the length of the prefix for which a solution exists might be unbounded and depend on the specific execution. This restriction comes from the *compactness* of the IIS model, that is, intuitively, that runs belong to the model if all its finite prefixes satisfy a given property. Then the bounded solvability of tasks comes from the combination of the *compactness* of a model with *Köenig's Lemma*. The notion of *compactness* and its implication with bounded solvability will be discussed in more details in Chapter 6.

Continuous ACT. It was observed by Herlihy and Shavit in [HS99], that when only specific tasks, called *colorless* tasks, are considered for the characterization, ACT can be provided with a continuous analogue. In a colorless task, processes can adopt input and outputs of other processes while satisfying the task specification. Hence, a simplicial map providing a task solution does not have to be color-preserving. Thus, the protocol complex, i.e., the subdivision, does not have to be chromatic either. It can for example be replaced by the barycentric subdivision. In turn, this allow to use the simplicial approximation theorem to replace the existence of a carrier-preserving simplicial map by the existence of a continuous carrier-preserving function, resulting in the following formulation of ACT:

Theorem 5.3. *[Continuous ACT] A colorless task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is wait-free solvable if and only if there exists continuous map F from $|\mathcal{I}|$ to $|\mathcal{O}|$ carried by $|\Delta|$.*

Note that the generalization of the simplicial approximation theorem implies that the continuous version of the ACT theorem can be generalized to task with a link-connected output complex.

Sperner's lemma and the impossibility of $(n-1)$-set agreement. The topological characterization of wait-free task solvability was introduced first to show the impossibility of solving the $(n-1)$-set agreement task. It was observed that a solution to an agreement task with the standard simplex as input complex defines a *Sperner's coloring*, that is, a map which associates each vertex to a color from its carrier in $\mathbf{s}$. Moreover, in the task of k-set consensus, a solution must ensure that in any configuration of the system, at most k value may be returned, or equivalently, that at most k colors may be used for any simplex of the protocol complex.

But *Sperner's Lemma* states that for any sperner coloring over a subdivision of an m-simplex, there is an odd number of m-simplices colored with $m+1$ distinct colors. But using ACT, a solution to an agreement task implies that there exists a Sperner coloring, and thus, that there is always some execution for which there are as many distinct outputs as there are processes. Therefore, the task of $(n-1)$-set consensus is not solvable in the wait-free model.

Chapter 6

Affine Tasks

In this chapter, we provide combinatorial characterization for many shared memory models. It is the most significant contribution of the thesis, and most of the results presented here were published in [GHKR16, KRH18]. In Section 6.1, we introduce and study affine tasks, objects from combinatorial topology used to characterize the task computability of shared-memory models. In Section 6.2, a first example is given of small affine tasks used to characterize k-test-and-set models. We continue in Section 6.3, with a characterization of k-obstruction-free models. In Section 6.4, we generalize the characterization of k-obstruction-free models to all *fair* adversarial models, or equivalently, to the active-resilient α-models.

6.1 Preliminaries

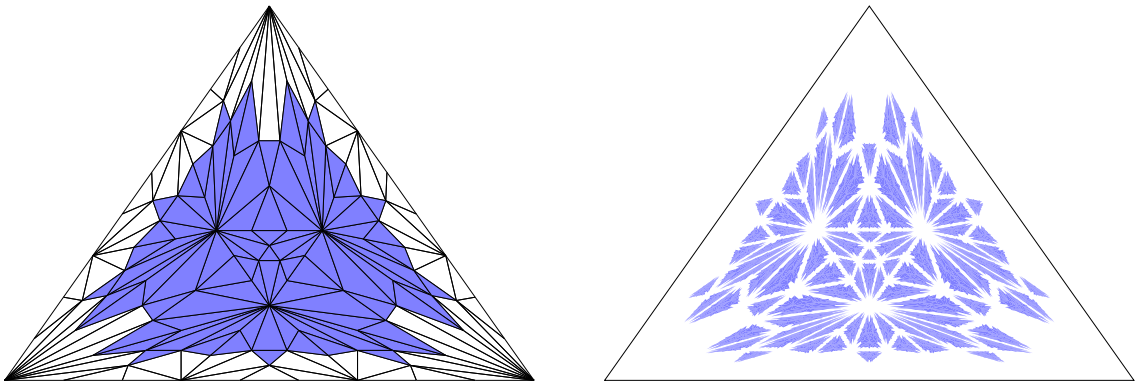
Following the proposal by Gafni et al. [GKM14], we use *affine tasks* to provide model representations using combinatorial topology. We also define the notion of an *affine model*. Using the *compactness* of affine models, we offer a simple characterization of task solvability in affine models.

Affine Tasks and Models

We first introduce affine tasks, a generalization of simplex agreement tasks on sub-complexes of some number of iterations of the standard chromatic subdivision. The notion of an affine model derived from a given affine task is then formally introduced.

Affine tasks. Let us first recall the definition of a simplex agreement task, given in Chapter 5. A simplex agreement task is defined by a (possibly chromatic) subdivision Sub. In the simplex agreement task corresponding to Sub, the input complex is the standard n-simplex $\mathbf{s}^n$, and the output complex is the corresponding subdivision of $\mathbf{s}^n$, i.e., $Sub(\mathbf{s}^n)$. The task specification is then the carrier map associated with the subdivision, that is, a face $\mathbf{t} \in \mathbf{s}^n$ must be mapped on simplices in the subdivision of the face, i.e., $Sub(\mathbf{t})$. When the subdivision is chromatic, the map must additionally be color-preserving.

An *affine task* is a generalization of the simplex agreement task, where the output complex is a pure non-empty sub-complex of some finite number of iterations of the standard chromatic subdivision, $\mathrm{Chr}^\ell \mathbf{s}$. Formally, let $\mathcal{L}$ be a pure non-empty sub-complex of $\mathrm{Chr}^\ell \mathbf{s}^n$ of dimension n for some $\ell \in \mathbb{N}$. The affine task associated to $\mathcal{L}$ is then defined as $(\mathbf{s}^n, \mathcal{L}, \Delta)$, where, for every



(a) Single iteration of the affine task.

(b) Two iterations of the affine task.

Figure 6.1 – A 3-processes affine task including the "interior" simplices of $\mathrm{Chr}^2\,\mathbf{s}^2$.

face $\sigma \subseteq \mathbf{s}^n$, $\Delta(\sigma) = \mathcal{L} \cap \mathrm{Chr}^\ell(\sigma)$. Hence, processes start on vertices of their color in $\mathbf{s}^n$ and must eventually output vertices of $\mathcal{L}$ of their color such that the set of outputs forms a simplex in $\mathcal{L}$ with a carrier equal to the set of observed processes. Note that $\mathcal{L} \cap \mathrm{Chr}^\ell(\mathbf{t})$ can be empty, in which case processes are not allowed to output views containing only the inputs from processes in $\chi(\mathbf{t})$. Intuitively, this is used to guarantee that the model can provide sufficiently large participation to be able to solve the affine task.

Note that, since an affine task is characterized by its output complex, with a slight abuse of notation, we use $\mathcal{L}$ for both the affine task $(\mathbf{s}, \mathcal{L}, \Delta)$ and its output complex.

An example of an affine task, defined as a sub-complex of $\mathrm{Chr}^2\,\mathbf{s}^2$, is depicted in Figure 6.1a for a 3-processes system. Facets of the output complex are displayed in blue on top of the edges of the two iterations of the standard chromatic subdivision.

Affine model. It can be noted that an affine task can also be seen as an operator on any pure simplicial complexes of the same dimension. Indeed, given a pure simplicial complex $\mathcal{K}$, we can construct the simplicial complex $\mathcal{L}(\mathcal{K})$ where each facet of $\mathcal{K}$ is replaced by an occurrence of $\mathcal{L}$. In particular, since the operation maintains purity, we can iterate this operation by recursively replacing simplices by an occurrence of the affine task.

By running m iterations of this operation on $\mathbf{s}$, we obtain $\mathcal{L}^m(\mathbf{s})$, a sub-complex of $\mathrm{Chr}^{\ell m}\,\mathbf{s}$. The second iteration of the 3-processes affine task of Figure 6.1a is given in Figure 6.1b. Since the affine task is a subset of the second iteration of Chr, two iterations thus form an affine task which is a subset of $\mathrm{Chr}^4\,\mathbf{s}^2$. The affine model $\mathcal{L}^*$ corresponding to the affine task $\mathcal{L}$ is obtained by iterating infinitely often the affine task. The m^{th} iteration of an affine task corresponds to a subset of $IS^{\ell m}$ runs (as each of the m iterations includes ℓ *IS* rounds). Hence the affine model $\mathcal{L}^*$ corresponds to the set of infinite runs of the IIS model where every prefix restricted to a multiple of ℓ *IS* rounds belongs to the subset of $IS^{\ell m}$ runs associated with $\mathcal{L}^m$.

Note that, by construction, affine models are compact. Indeed, they are defined through a "safety" property on the set of IIS runs: if all prefixes of an IIS run satisfy the model conditions then the infinite run belongs to the model.

Task Solvability in Affine Models

In an affine model $\mathcal{L}^*$ associated with the affine task $\mathcal{L}$, there are no possible failures. Hence, when solving a task, all processes must eventually produce an output. By default, a task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is thus solvable in an affine model $\mathcal{L}^*$ if there exists a protocol such that: in every run of $\mathcal{L}^*$ in which processes start with an input vector $I \in \mathcal{I}$, there is a finite prefix of the run in which decided values form a vector $O \in \mathcal{O}$ with only non-$\perp$ values and with $(I, O) \in \Delta$.

Bounded termination. The compactness of affine models can be used to show a stronger result: that any protocol solving a task must provide outputs to all processes after a bounded number of iterations. The primary tool to prove this result is König's Lemma [Kön26] stating that infinite trees must either have infinite branching or infinite paths.

Property 6.1. *[Bounded termination] A task $T = (\mathcal{I}, \mathcal{O}, \Delta)$* is solvable in $\mathcal{L}^*$ *if there exists a protocol and an integer $\ell \in \mathbb{N}$ for which in every prefix of size ℓ of a run of L^*: if processes started with an input vector $I \in \mathcal{I}$, then decided values form a vector $O \in \mathcal{O}$ with only non-$\perp$ values such that $(I, O) \in \Delta$.*

Proof. Showing that there must exist a bound on the length of prefixes to satisfy the condition is a simple application of König's Lemma stating that an infinite tree with finite branching must contain an infinite path. An affine model can be seen as an infinite tree where the root is the empty prefix, its children are the prefixes of length 1 of valid runs of the model (simplices of the affine task), their children are the extensions of length 2 of their prefix which are the prefixes of valid runs of the model (simplices of the affine task applied to the simplex of the parent), and we can continue constructing this tree inductively to represent runs of $\mathcal{L}^*$.

When we consider a protocol solving a task, every run must possess a terminated prefix. Hence we can prune the tree at all minimal terminated prefix. If the pruned tree is finite, then there exists a bound on the length of prefixes required for the protocol to terminate. Hence, assume that the pruned tree remains infinite. It implies that it includes an infinite never terminating path. Thus, there is an infinite run which does not belongs to $\mathcal{L}^*$ while its prefixes are prefix of valid runs of the model — a contradiction with the compactness of $\mathcal{L}^*$. □

Combinatorial characterization. The existence of a bound on the length of the executions required to be able to produce a task output can be used to provide a more elegant formulation of task solvability in an affine model $\mathcal{L}^*$. Indeed, let ℓ be the maximal length of an execution in $\mathcal{L}^*$ required for the protocol to terminate. Even if processes could terminate earlier in some executions, nothing prevents us from making processes always execute ℓ rounds and only then return an output. The state of processes after these ℓ rounds correspond to vertices of $\mathcal{L}^\ell(I)$ forming a simplex. Given a state, the protocol must select an output to return, hence, a protocol is equivalent with a map from $\mathcal{L}^\ell(\mathcal{I})$ to $\mathcal{O}$. Moreover, the protocol solves the task if and only if simplices of $\mathcal{L}^\ell(I)$ are mapped to simplices of $\Delta(I)$. Hence the existence of a protocol solving the task equates with the existence of a simplicial (color-preserving) map carried by Δ. It leads to the following formulation of task solvability in affine models:

Property 6.2. *[Affine solvability] A task $T = (\mathcal{I}, \mathcal{O}, \Delta)$* is solvable in $\mathcal{L}^*$ *if and only if there exists $\ell \in \mathbb{N}$ and a carrier and color preserving simplicial map $\phi : \mathcal{L}^\ell(\mathcal{I}) \to \mathcal{O}$ carried by Δ.*

Shared Memory Simulation in affine models

Most of the shared-memory models considered in this thesis can be seen as a combination of a shared memory and other operations such as agreement protocols or access to distributed objects. In this, simulating such models shares a common issue that is combining a shared memory simulation with other terminating operations. Simulating a long-lived shared memory in an affine model is not very complicated in itself. Indeed, solutions proposed for the IIS model can be applied to any affine model as it is a restriction on the set of IIS runs. The issue is to deal with interference that other operations may create with the shared memory simulation.

All processes must terminate. One of the main complexity of the simulation comes from the combination of the failure-freedom and the iterative structure of the affine models. A process obtaining small outputs in all iterations, often denominated as a "fast" process, may never observe the values shared by "slow" processes with larger views. But as there are no process failures, eventually, all processes must obtain a task output. It requires that fast processes make progress with the simulation without waiting for slower processes. Slow processes must thus wait for faster processes to terminate their simulation before being able to make progress themselves.

To resolve this issue, a terminated fast process must let slower processes know that it will no longer participate to the simulation and thus that its presence may just be ignored, e.g., it does not need to be able to read the content of completed write operations. This can be done by making processes which obtained a task output in the simulation use a particular value $\perp$ as input for all further iterations of the affine model. Slower processes are then aware that processes using $\perp$ do not interfere anymore and no longer need to witness their modifications of the simulated system state. But this particular value $\perp$ can only be used by terminated processes. A process with no pending memory operation, waiting for some other operation to terminate, cannot use $\perp$ at this stage.

Interference on liveness. A shared memory simulation working in an iterated model can only provide lock-free progress, ensuring that some non-terminated process makes progress. But processes may be trying to simulate other types of operations. If the shared memory simulation provides progress to processes without a pending memory operation, then the shared memory simulation may be blocked. In this setting, either the shared memory simulation should be improved to guarantee still some progress or the other operations should provide higher liveness guarantees.

Shared memory is a long-lived abstraction, and all memory operations are dependent on each other. Moreover, processes can write their complete current state and hence cannot ignore preceding operations result. Thus a memory operation cannot be initiated early before other operations terminated. But a fast process cannot let processes that it does not observe complete new write operations that it may never be able to read. Thus, when fast processes do not have a pending memory operation, progress cannot be ensured. Hence, all simulations must at least provide progress to fast processes when all of them participate and possibly even when a single fast process is performing an operation.

Dealing with waiting processes. A fast process without a memory operation must still be able to prevent slower processes from making progress with their write operations. It can

be easily integrated into a given simulation algorithm or dealt with externally. For example in the solution presented in [GR10], processes can increment their write counter at each round in which they do not have a pending memory operation. This simple trick prevents slower processes from making progress as long as some "waiting" process is observed. Intuitively, this does not prevent progress when all fast processes have a pending operation.

For simplicity, we are going to use an even simpler approach that can be used with any shared memory simulation scheme. A process waiting for another operation can merely re-write its last written value as long as necessary. This *dummy* write operation does not affect safety as the simulated write operation replaces a given value by itself. The liveness is not impacted when all processes really have a pending memory operation as even if some dummy write operations may still be ongoing, as soon a process terminates one, it replaces it with the true pending memory operation. Hence, eventually, a true operation will be completed as infinitely many operations are completed in a long-lived simulation.

Shared memory simulation properties. The shared memory simulation we use is not much modified from the ones used for the characterization of wait-freedom using the standard chromatic subdivision. Let us call this simulation the *composable* shared memory simulation. The only difference is that processes keep repeating their previously completed write operations when another type of operation is being simulated. Since the simulation algorithms are not modified, the safety of the simulation is guaranteed.

Theorem 6.1. *All operations completed by the* composable *shared memory simulation satisfy atomic snaphsot properties.*

The issue that may arise concerns liveness of the simulation. In itself, the simulation will keep the liveness property of the underlying simulation. But completed operations may be repetitions of the same operation over and over. For the liveness property, we rely on the specific simulation proposed in [GR10]. As long as there is at least one non-terminated process, eventually a non-terminated process with the smallest view in some iteration will complete its memory operation. Ensuring that processes do not proceed to dummy write operations forever is delegated to other operations, but it may use the property that a process performing write operations infinitely often has the smallest IIS view infinitely often.

Theorem 6.2. *As long as there is a non-terminated process, the* composable *shared memory simulation eventually provides progress to a non-terminated process with the smallest IIS view in some round.*

6.2 Affine Tasks for k-Test-and-Set

Before looking at large classes of models and showing how they can be characterized using affine tasks, let us first look at a simple example of the *k-test-and-set models*. Recall that in the n-processes k-test-and-set model, defined in Chapter 2, processes have access to a shared memory and to infinitely many k-test-and-set objects. Processes can access objects in any order and there are no restrictions on the set of possible runs. Recall also that a k-test-and-set object or task is accessed by a single input-less operation *apply*() which must eventually return 0 or 1 to all correct processes and such that at least 1 and at most k out of the participating processes return 1.

The first step consists in defining an affine task that aims to characterize the k-test-and-set model. Then we need to show that the corresponding affine model is weaker than the k-test-and-set model, by showing how to solve the affine task itself. We also need to show that the affine model is stronger and hence equivalent to the k-test-and-set model. This last part is slightly more complicated as it consists in simulating the k-test-and-set model running an arbitrary algorithm solving a task and which must provide outputs to all processes.

Definition of $\mathcal{R}_{k-T\&S}$

The goal is to design an affine task as simple as possible. As we will see, we can do so by using an affine task which is a subset of one-round the chromatic subdivision.

Test-and-set is well known for its ability to be used to solve perfect renaming [AM97]. Moreover, it provides adaptive solutions where the order of processes ranks is a linearization of their access to the renaming task. Hence a process obtaining the name j can see all values shared previously by processes which receive a smaller name i with $i < j$. In terms of immediate snapshots, this can be seen as requiring that processes are totally ordered with a distinct rank j such that all inputs from processes with smaller ranks are observed. It implies that the process of rank j sees precisely j inputs, its own and those of processes with a strictly smaller rank. These particular immediate snapshot executions are often denominated as *total order* executions as no two process possess the same output.

Generalizing this constraint to k-test-and-set objects is quite natural. It can be seen as providing processes with a preorder in which a rank may be given to at least 1 and at most k processes. Similarly, a process can observe the values previously shared by any process obtaining a smaller or equal rank. In an immediate snapshot, this is equivalent with having at most k processes sharing the same output. It leads to a definition of k-*ordered* executions or simplices: among any set of $k+1$ processes at least two have different ranks. Note that the set of total order and 1-ordered executions are indeed identical.

The definition of the affine task proposed to characterize the k-test-and-set model follows this restriction that executions should be k-ordered. We call this n-processes affine task composed of k-ordered simplices $\mathcal{R}_{k-T\&S}$. Formally, $\mathcal{R}_{k-T\&S}$ is the set of simplices of the standard chromatic subdivision such that at most k of its vertices share the same carrier:

Definition 6.1.

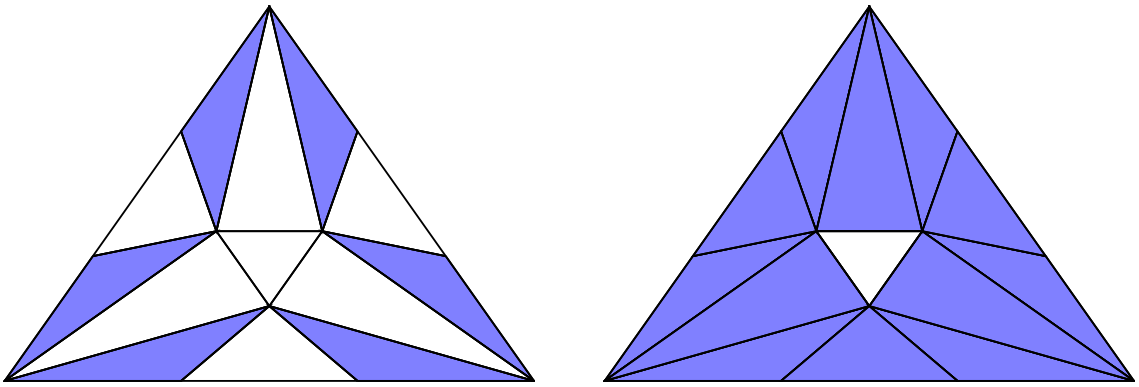
$$\mathcal{R}_{k-T\&S} = \{\sigma \in \mathrm{Chr}(\mathbf{s}) | \forall \sigma' \subseteq \sigma : (\forall v, v' \in \sigma', \mathit{carrier}(v) = \mathit{carrier}(v')) \implies |\sigma'| \leq k\}.$$

The definition of $\mathcal{R}_{k-T\&S}$ clearly defines a subset of simplices of the standard chromatic subdivision. But this is not enough to be an affine task. Indeed, it must be checked that $\mathcal{R}_{k-T\&S}$ is a pure sub-complex of the same dimension as $\mathbf{s}$:

Property 6.3. *$\mathcal{R}_{k-T\&S}$ is an affine task.*

Proof. The fact that $\mathcal{R}_{k-T\&S}$ is a sub-complex of $\mathrm{Chr}\,\mathbf{s}$ is trivial. Indeed, the definition is inclusion-closed. Consider any simplex $\sigma \in \mathcal{R}_{k-T\&S}$ and any subset of it $\sigma' \subseteq \sigma$. Any face of σ' is a face of σ and hence satisfies the condition of having at most k vertices sharing the same carrier.

Showing that $\mathcal{R}_{k-T\&S}$ is pure and of the same dimension as $\mathbf{s}$ is less obvious. For this, we need to show that any simplex $\sigma \in \mathcal{R}_{k-T\&S}$ is a face of a simplex $\sigma' \in \mathcal{R}_{k-T\&S}$ of dimension



(a) $\mathcal{R}_{1-T\&S}$ (b) $\mathcal{R}_{2-T\&S}$

Figure 6.2 – 3-processes affine tasks $\mathcal{R}_{1-T\&S}$ and $\mathcal{R}_{2-T\&S}$ with their facets displayed in blue.

equal to $dim(\mathbf{s})$. Note that by transitivity, it is sufficient to show that any simplex of $\mathcal{R}_{k-T\&S}$ of a strictly smaller dimension is a face of a strictly larger simplex of $\mathcal{R}_{k-T\&S}$.

Consider a simplex $\sigma \in \mathcal{R}_{k-T\&S}$ and any color c from Π such that $c \notin \chi(\sigma)$ and let $v \in \mathbf{s}$ be the vertex of $\mathbf{s}$ of color c. Two cases may happen, either c is a color of the carrier of all vertices in σ, or there is a vertex from σ with the largest carrier $\mathbf{t}$ such that $c \notin \chi(\mathbf{t})$. In the former case, we can add the vertex $(c, \{v\})$ to σ. In the latter case, we can add the vertex $(c, \{v\} \cup \mathbf{t})$ to σ. It is easy to check that the new simplex still verifies the immediacy, self-inclusion and containment properties. Moreover, the new vertex has by construction a carrier shared by no vertex in σ, and hence the new simplex belongs to $\mathcal{R}_{k-T\&S}$. Indeed all vertices $v \in \sigma$ such that $\chi(v) \in \chi(\mathbf{t})$ must have a carrier which is a face of $\mathbf{t}$ due to the immediacy property. Hence, as long as there are missing colors, we can find a larger simplex including σ as a face which still belongs to $\mathcal{R}_{k-T\&S}$. □

The affine tasks corresponding to 3-processes 1-test-and-set and 2-test-and-set models are depicted in Figure 6.2. The facets of the affine tasks are displayed in blue, and thus, any boundaries of blue simplices also belong to the affine task.

Solving $\mathcal{R}_{k-T\&S}$ with k-Test-and-Set

Solving $\mathcal{R}_{k-T\&S}$ using access to shared memory and access to k-test-and-set objects is not very difficult. It was previously resolved in [MRT06, GRT07] under the name of the *k-participating-set* object. The idea consists in using a level-based immediate snapshot implementation (as the one presented in Algorithm 2.1 from Section 5) with the addition of k-test-and-set objects to prevent more than k processes from outputting on the same level.

The modification proposed in [MRT06, GRT07] to solve $\mathcal{R}_{k-T\&S}$, consists in modifying the level-based immediate snapshot exit condition as follows: if the snapshot contains ℓ values associated with a level $\ell' \leq \ell$ then the process accesses the k-test-and-set object with identifier ℓ; If it obtains true, then it returns with the snapshot consisting of these ℓ values. Otherwise, the process resumes its loop as if it saw less than ℓ values with smaller or equal levels. The formal description can be found in Algorithm 6.1.

Since the immediate snapshot algorithm is modified, we can no longer rely on the property which was proved in Lemma 2.1. Here is an adapted version:

Algorithm 6.1: Solving $\mathcal{R}_{k-T\&S}$ for process p_i.

Shared Objects: $MEM[1 \dots n] \in Val \times \mathbb{N}$, **initially** $(\perp, \perp)$;
Init: $level = n + 1$, $exit = true$, $value = InputValue$, $snap = \emptyset$;

Do
  Do
    $level = level - 1$;
    $MEM[i].Update(value, level)$;
    $snap = MEM.Snapshot()$;
  While $|\{(v, \ell) \in snap, \ell \leq level\}| \neq level$;
  $exit = k\text{-}Test\&Set[level]$;
While $\neg exit$;
Return $\{(v, \ell) \in snap, \ell \leq level\}$;

Lemma 6.1. *In an execution of Algorithm 6.1, at most* $(n - \ell + 1)$ *processes can reach the* ℓ^{th} *iteration of the inner while loop.*

Proof. Let us show this result by induction on the number of iterations m of the inner while loop. If $m = 1$, the claim is trivially verified as there are only n processes. Let us assume that the lemma holds for iteration m, then at most $n - m + 1$ processes may reach iteration m of the while loop. If there is strictly less than $n - m + 1$ processes or if a process crashes during iteration m, then the lemma property holds for iteration $m + 1$.

So let us now assume furthermore that exactly $n - m + 1$ processes reach iteration m and complete it without crashing, and consider the process p which takes the last snapshot during this iteration. So p must see the value shared by all $n - m + 1$ processes which participate to the round. These processes may have updated their value in further rounds, but only with a smaller associated level. Hence, p sees $n - km1$ processes with an associated level smaller than or equal to $n - m + 1$. Thus p exits the inner while loop.

If p obtains *true* as output to the k-test-and-set operation, then it exits the second while loop and returns from the algorithm. Otherwise, if p obtains *false*, another process must have accessed the k-test-and-set and either crashed or obtained *true*. Therefore p or another process does not access the next round of the while loop as it crashes or exits at iteration m. Hence, at most $n - m$ processes access iteration $m + 1$. □

Theorem 6.3. *Algorithm 6.1 solves* $\mathcal{R}_{k-T\&S}$.

Proof. First, let us show that Algorithm 6.1 solves the immediate snapshot task. It can be noticed that the proof of Theorem 2.1 still applies to the modified version of Algorithm 6.1, only Lemma 2.1 proof is no longer valid, but Lemma 6.1 may be used as a replacement.

As Algorithm 6.1 solves the immediate snapshot task, all left to be shown is that at most k processes output at the same level, i.e., are provided with the same output set and thus are associated with vertices sharing the same carrier. It is trivial to see that this is the case. Indeed, consider a set of processes Q all outputting on the same level ℓ, they all accessed the same k-test-and-set object (associated with level ℓ) and all obtained *true* as output. But at most k processes may obtain *true* as output from the same k-test-and-set object, hence, Q is of a size of at most k. □

Simulating Test-and-Set and Shared Memory in $\mathcal{R}^*_{1-T\&S}$

Simulating k-test-and-set and read-write shared memory in $\mathcal{R}^*_{k-T\&S}$ is a bit more complicated than solving $\mathcal{R}_{k-T\&S}$ in the k-test-and-set model. We first focus on the test-and-set model and $\mathcal{R}^*_{1-T\&S}$. Only after that, we will show how to proceed for the general case.

Using the composable shared memory. For the shared memory operations we use the composable shared memory simulation presented in Section 6.1. The shared memory simulation can be combined with other operations, and the global simulation provides liveness as long as other operations ensure progress to any non-terminated process obtaining the smallest view infinitely often.

Solving a test-and-set operation in one iteration of $\mathcal{R}_{1-T\&S}$ is very simple: a process seeing only itself returns 1 while others return 0. But in detail it is slightly more complicated as processes may access distinct test-and-set operations, access them in various orders, and initiate them at different iterations.

Fast termination. Consider a non-terminated process trying to simulate a test-and-set operation. If this process sees infinitely often only itself as non-terminated processes, it should eventually terminate. Moreover, it must return 1, at least when it does not see any other process trying to compete for the same operation. But processes with the smallest view may change arbitrarily between rounds. Hence if after some iteration a process gathered enough information to return 1, it must be that other non-terminated processes already know that 0 has to be returned. Otherwise, if a process may still return 1 and sees only itself in following iterations, then it will also return 1, violating the test-and-set unicity property.

Hence slow processes must fail preemptively when they see a fast process participating in a test-and-set operation. Indeed, if they start later on their participation, they must still remember that some other process may have won in preceding rounds. The issue is that as soon as three processes are seen, a process cannot know if an observed process is the fastest or only the second one. Hence it is difficult to know if it should wait more or preemptively fail the test-and-set operation accessed by some process it sees.

Divide and conquer. To simplify the simulation of test-and-set operations and resolve our issue of identifying the fastest process, we do not solve n-processes test-and-set directly, but instead, we provide 2-processes test-and-set operations. Since 2-processes test-and-set can be used to solve n-processes test-and-set by merely using a combination of 2-processes test-and-set operations: a process accesses 2-processes test-and-set operations iteratively competing with each other process one by one, if it wins a contest it continues but if it loses one it stops competing to remaining operations. A process must succeed all its 2-processes test-and-set operations to win its n-processes test-and-set operation. Indeed, assume that it is not the case, then each process failed to another process. This forms a loop of failures. But as a process stops participating as soon as they fail, they cannot win against a process with a greater identifier than the one they lost to. Hence the loop must be composed of decreasing identifiers which is absurd. Seeing that at most one process returns 1 is trivial as any two processes compete with each other. Note that even if processes names are not known initially, processes can proceed first to a $2n-1$ wait-free renaming before competing with the $2n-2$ possible other names, as if there is no competitor then a process can only win.

Solving 2-processes test-and-set operations is much more straightforward. Indeed, a process knows the only possible competitor. Hence, in a given iteration of $\mathcal{R}_{1-T\&S}$, it knows whether it is the faster one of the two (if it does not see the other process value) or else that the other process is the fastest. This simple property can be used to solve test-and-set operation and ensure that fast processes terminate directly without waiting for other processes to participate.

Simulation description. The simulation consists in solving 2-processes test-and-set operations and simulating shared memory operations. The 2-processes test-and-set operations are performed by sharing through the affine task inputs, the operation calls to the 2-processes test-and-set objects. As soon as a process sees a competitor sharing its call to a 2-test-and-set object, it remembers it failed this operation and must return 0 without taking any step in simulating it. If the competitor never saw its competitor accessing the operation, then it shares its call to the operation in iterations of the affine task until either: (1) it does not see its competitor in some iteration and returns 1; or (2) it recognizes that its competitor terminated its global simulation and also returns 1; or (3) it sees the competitor accessing the operation too and in this case returns 0. Seeing that the simulation is safe and live is straightforward:

Theorem 6.4. *The n-processes test-and-set model and the affine model $\mathcal{R}^*_{1-T\&S}$ are equivalent for task solvability.*

Proof. The affine model can be simulated by using iterations of the solution to $\mathcal{R}_{1-T\&S}$ from Algorithm 6.1, hence, any task solvable in $\mathcal{R}^*_{1-T\&S}$ is solvable in the test-and-set model.

For the other direction of the proof, let us show that our simulation of shared memory and 2-processes test-and-set operation is *safe* and *non-blocking*. In a 2-processes test-and-set operation, processes may only return 1 or 0. Moreover, a process which observes the other processes participating before returning can only return 0, hence at most one process may return 1 as the containment property ensure that at least one process see the other one. But as processes never share the same view, one of the two processes must have a strictly smaller view and not see the other competitor and hence returns 1. Therefore test-and-set and shared memory operations (see Theorem 6.1) are thus safe.

The liveness of the combined simulation requires that non-terminated processes obtaining the smallest view infinitely often must eventually terminate its test-and-set operations (see Theorem 6.2). But a process terminates its test-and-set operation as soon as it does not observe its competitor in some iteration, hence, in particular, the process with the smallest view among non-terminated processes. Thus the combined simulation provides progress to a non-terminated process. But as processes execute an algorithm solving a task, they terminate after a finite number of simulated operations. Hence, all processes eventually terminate.

Therefore a task is solvable in the n-processes test-and-set model if and only if it is solvable in the affine model $\mathcal{R}^*_{1-T\&S}$. □

Simulating k-Test-and-Set and Shared Memory in $\mathcal{R}^*_{k-T\&S}$

Simulating k-test-and-set operations in $\mathcal{R}^*_{k-T\&S}$ is more complicated. Solving a k-test-and-set operation when all processes participate in the same iteration is simple: processes seeing k or less non-terminated processes return 1 while others return 0. But as processes may access objects in any iteration and any order, even by focusing on k-test-and-set operations among $k+1$ processes, letting slow processes wait to see faster processes participating is no longer sufficient. Indeed, when k processes participate, they may all see all $k+1$ potential

participants. But a process cannot know if other processes are also waiting or saw fewer processes and returned 1, and so, whether it should return 0 or wait. Instead, we are going to solve the equivalent operation of k-set consensus among $k+1$ processes.

From k-set consensus among $k+1$ processes to k-test-and-set. Using k-set consensus among $k+1$ processes to k-test-and-set operations among $k+1$ processes is rather simple. Processes can access a k-set consensus operation with their identifier. Then they write their output to the shared-memory and take a snapshot. If a process sees that some process obtained its identifier as output, it returns 1, and otherwise, it returns 0. As at most k distinct identifiers may be returned by the k-set consensus, at most k processes may return 1. Moreover, the first identifier written to the memory will be observed by all processes and hence the one with this identifier which will return 1.

Solving n-processes k-test-and-set can then be obtained from $k+1$-processes k-test-and-set operations. Indeed, as for test-and-set, we can make processes compete against all possible sets of $k+1$ processes in increasing order until they lose to one and return 0 and win to all and return 1. A process must obtain 1 as processes access the sequence in the same order and stop participating as soon as they fail to one operation. Moreover, at most k processes may return 1 as in any set of $k+1$ processes, one must fail the k-test-and-set operation among them. Hence, a solution to k-set consensus among $k+1$ processes is sufficient to solve k-test-and-set operations.

Active participation. The advantage of k-set consensus operations compared to k-test-and-set operations is that processes can participate as soon as they see some process participating. Indeed, since it a colorless task, processes can adopt inputs from any other process. Hence, to solve k-set consensus operations among $k+1$ processes, processes maintain a decision estimate for all k-set consensus operations and share them in all iterations of $\mathcal{R}^*_{k-T\&S}$. When a process initiates a new operation for which it has no decision estimate yet, it simply adds a decision estimate corresponding to its input value. Moreover, when a process sees a process participating in a new operation, it adopts its decision estimate.

Now, at the end of each iteration of $\mathcal{R}_{k-T\&S}$, processes look at the decision estimate for all operations. If a process sees all $k+1$ potential participants of an operation, then it replaces its decision estimate by the decision estimate of the process with the next identifier (going back to the first to form a loop when there are none higher). For a process to terminate, it must see that all potential participants share a decision estimate during the same round. When this happens, a process returns its potentially updated decision estimate as its k-set consensus output.

Correctness of the simulatation of k-set consensus among $k+1$ processes. Let us first show that simulated operations respect the specification of k-set consensus among $k+1$ processes before showing that sufficient progress is also guaranteed to simulated operations.

Lemma 6.2. *The simulation safely solves k-set consensus among $k+1$ in $\mathcal{R}^*_{k-T\&S}$.*

Proof. Processes return their decision estimate which is initially set to their input (processes identifiers) or adopted from other processes decision estimates. Hence validity is satisfied.

Now consider the first iteration of the affine task after which the first process returns with an output. In this iteration, all processes with the smallest view shared a decision estimate.

Hence, all processes adopted a decision estimate at the end of the round. If at the end of the round there are less than k distinct decision estimates, then the agreement property will be ensured as the number of distinct decision estimates in later rounds is a subset of this one.

To see that there are at most k distinct decision estimates at the end of this first iteration in which a process decides, consider the processes which see the $k+1$ potential participants. These processes adopt the decision estimate of the next process (relatively to identifier ranks). But in $\mathcal{R}_{k-T\&S}$, at most k vertices may share the same carrier. Hence a process seeing all participants must adopt the decision estimate of a process not seeing all of them. But this process does not change its decision estimate. Thus, two processes share the same decision estimate. The number of distinct decision estimate is, therefore, smaller than or equal to k and hence at most k distinct outputs may be returned. □

Lemma 6.3. *The simulation of k-set consensus among $k+1$ in $\mathcal{R}^*_{k-T\&S}$ provides progress to processes having infinitely often the smallest view among non-terminated processes.*

Proof. Processes participate in an operation as soon as they see another process participating. In particular if a process with the smallest view participates in some iteration, all processes participate in the next iteration. But if all processes observed in some iteration are participating, then a process returns at the end of the round. Hence, a process with a k-set consensus operation terminates at most one round after obtaining the smallest view among non-terminated processes. □

Equivalence between k-test-and-set model and $\mathcal{R}^*_{k-T\&S}$. Therefore, we can conclude with the equivalence of the two classes of models considered. Indeed, a simulation of a shared memory model with access to k-set consensus operations among $k+1$ processes can be used to simulate the k-test-and-set model. Thus, as Algorithm 6.1 can be used to simulate the affine model $\mathcal{R}^*_{k-T\&S}$ in the k-test-and-set model, we obtain the following equivalence:

Theorem 6.5. *A task is solvable in the n-processes k-test-and-set model if and only if it is solvable in the affine model $\mathcal{R}^*_{k-T\&S}$.*

This can lead to the following generalization of the asynchronous computability theorem for the k-test-and-set models:

Theorem 6.6. *A task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is solvable in the k-test-and-set model if and only if there exists $\ell \in \mathbb{N}$ and a simplicial map $\delta : (\mathcal{R}_{k-T\&S})^\ell(\mathcal{I}) \to \mathcal{O}$ carried by Δ.*

6.3 Affine Tasks for k-Obstruction-Free Adversaries

In this section, we show howaffine tasks can capture the class of k-plain models. Recall that k-plain models correspond to the well-behaved family of k-obstruction-free, $(k+1)$-active-resilient, k-concurrent, or equivalently, k-set consensus models. We first show why we need to define an affine task using the second iteration of the standard chromatic subdivision. Then, we formally introduce the candidate affine task $\mathcal{R}_k$. Lastly, we show its task computability equivalence, by solving the affine task in the $(k+1)$-active-resilient model and by simulating the k-set consensus model in the affine model. This result was published in [GHKR16] using a different formulation, here we use a definition that more easily extends to fair models.

One Iteration is not Enough

According to agreement functions classes, plain models appear to be the simplest class of models. However, it is impossible to characterized their task computability using 1-level affine tasks. A similar impossibility result was shown for the t-resilient models in [DFRR16], but proving this result for k-plain models is much simpler.

Strongest solvable 1-level affine task is $\mathcal{R}_{k-T\&S}$. To show the impossibility of characterizing k-plain models using 1-level affine tasks, we are going to show that all 1-level affine tasks solvable in the k-obstruction-free model include $\mathcal{R}_{k-T\&S}$.

Consider an $(n-1)$-dimensional simplex $\sigma \in \mathcal{R}_{k-T\&S}$. Now, consider the $m \leq k$ processes with the smallest view in σ. Indeed, at most k processes can have the same view and hence the smallest one in particular. In the k-obstruction-free model, nothing prevents only these m processes to take steps and any m-processes execution is valid, i.e., they essentially run wait-free. Hence, they cannot solve a better simplex agreement task than the m-processes wait-free model. Therefore, nothing can prevent all processes from outputting the same view where they see each other. At this point the processes are terminated. We can then take the next batch of processes with the second smallest view, there is always at most k of them, and let them run wait-free hence preventing us to avoid allowing them to see each other. This scheme can then be continued until all processes are provided with outputs and form the set of outputs σ. Hence any first level affine task solvable in the k-obstruction-free model is necessarily a superset of $\mathcal{R}_{k-T\&S}$.

$\mathcal{R}^*_{k-T\&S}$ is weaker than the k-set consensus model. The n-processes k-test-and-set model is strictly weaker than the k-obstruction-free model when $k \leq n-2$. Hence for all these models, there is no 1-level affine task grasping their task solvability power.

We can easily show this impossibility explicitly for the case of consensus and $\mathcal{R}_{1-T\&S}$. First, note that the task presents a path between each couple of corners (vertices of $\mathbf{s}$) when $n \geq 3$. Indeed, a process seeing three process outputs cannot distinguish the two executions in which the order of the first two processes is reversed. Hence, it has the same state in executions in which either one of these two processes was first and thusassociated with their "corner" vertex in $\mathcal{R}_{1-T\&S}$. Hence, all corners are connected through 1-dimensional paths. When iterated, each edge of the path can be replaced by the 1-dimensional path between the two corners of the corresponding colors. Therefore, after any number of iterations, a 1-dimensional path remains between all corners. But in the consensus task, processes seeing only themselves must output their input value. Therefore, according to the 1-dimensional case of Sperner's Lemma, there exists an edge on these 1-dimensional path connecting corners that must have its associated vertices colored by the two corners colors. Hence, no algorithm can solve consensus in $\mathcal{R}_{1-T\&S}$ when $n \geq 3$.

Definition of $\mathcal{R}_k$

Since no 1-level affine task exists in general for plain models, let us define an affine task $\mathcal{R}_k$ as a pure subcomplex of $\mathrm{Chr}^2\,\mathbf{s}$. We will later show that $\mathcal{R}^*_k$ precisely captures the task computability of the n-processes k-obstruction-free adversarial model, or equivalently, any k-plain model.

Original solution. In our original solution [GHKR16], the definition of $\mathcal{R}_k$ was inspired by the k-concurrency model where at most k processes might be concurrently active. It leads to a definition of $\mathcal{R}_k$ almost identical to the definition of $\mathcal{R}_{k-T\&S}$, but in $\mathrm{Chr}^2\,\mathbf{s}$. Indeed, if two processes are not executed concurrently, then the first to run cannot see the other process. Hence, they see different sets of processes, or equivalently, end up with vertices with distinct carriers. This is why we first proposed to use the following definition for $\mathcal{R}_k$:

Definition 6.2. *[Definition of [GHKR16]]*

$$\mathcal{R}_k = \{\sigma \in \mathrm{Chr}^2(\mathbf{s}) | \forall \sigma' \subseteq \sigma : (\forall v, v' \in \sigma', \mathit{carrier}(v) = \mathit{carrier}(v')) \implies |\sigma'| \leq k\}.$$

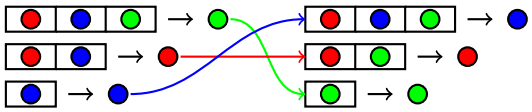
The only difference with $\mathcal{R}_{k-T\&S}$ is that Chr is replaced by Chr^2. While this definition was shown to be valid and is very appealing, it does generalize well to a broader class of models such as fair models. The general intuition that may emerge from the original definition [GHKR16] is that among any subsets, processes can identify at most k "leaders" which are the processes observed with the smallest IS^1 view. It can intuitively be used to solve k-set consensus operations which can itself be used to simulate the k-set consensus model.

More flexible definition. Our issue with this definition comes from the fact that restrictions are enforced before any process is participating. If all processes are participating in the k-obstruction free model, we must prevent all but k of them from making any progress with the resolution of the affine task until some process terminates. It implies that the "concurrency level" cannot fluctuate with the system participation. Instead, we desire a more independent property which makes as little restriction as possible while still allowing us to solve the desired set-consensus tasks.

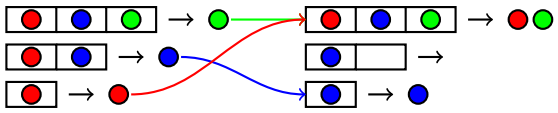
For this, we replace the "leader" selection by a view or carrier selection. We want processes to be able to select at most k distinct carriers. For this, we do not need to make any restriction on the set of first IS outputs processes that may obtain. We can restrict only their possible outputs in the second iteration of Chr, based on the set of IS1 view they acquired. Intuitively if $k+1$ processes obtained different IS^1 output, then they must be able to select at most k of them. But no restrictions can be made on an execution where these $k+1$ processes do not execute concurrently, and hence obtain distinct views during the two IS with relative sizes reflecting the relative execution order. In such execution, a slow process must select a smaller IS^1 view obtained by another process. But picking the smallest IS^1 view observed does not decrease the number of chosen carriers selected when processes have different views in the two iterations of Chr, but with a reversed inclusion ordering in each. This worst execution case leads us to the definition of *contention simplices* that must be avoided in our affine tasks.

Contention simplices. For a vertex $v \in \mathrm{Chr}^2\,\mathbf{s}$, let $\mathit{View}^1(v)$ and $\mathit{View}^2(v)$ be the sets of processes seen by the process $\chi(v)$ in, respectively, the first and the second *IS* (we call these View^1 and View^2). Formally, $\mathit{View}^2(v) = \mathit{carrier}(v, \mathrm{Chr}\,\mathbf{s})$ and $\mathit{View}^1(v) = \mathit{carrier}(v', \mathbf{s})$ with $v' \in \mathit{carrier}(v, \mathrm{Chr}\,\mathbf{s})$ such that $\chi(v) = \chi(v')$.

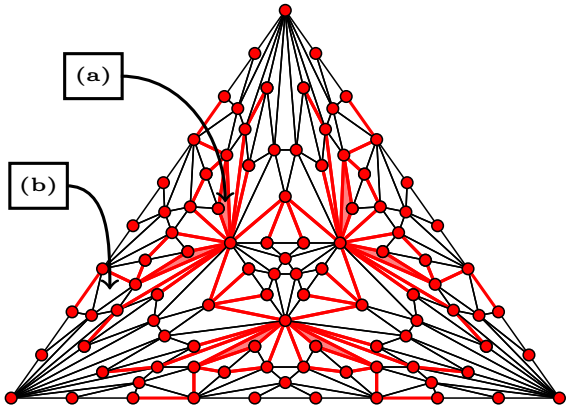
We formalize the intuitive description of contention simplices as follows: In a simplex $\delta \in \mathrm{Chr}^2\,\mathbf{s}$, we say that vertices v and v' are *contending* if their View^1 and View^2 are ordered in the opposite way: $\mathit{View}^1(v)$ is a proper subset of $\mathit{View}^1(v')$ and $\mathit{View}^2(v')$ is a proper subset of $\mathit{View}^2(v)$, or vice versa. If every two vertices of δ are contending, then we say that δ is a 2-*contention* simplex. Let Cont_2 be the set of 2-contention simplices, formally:



(a) Two reversed IS ordered runs: $\{p_2\}, \{p_1\}, \{p_3\}$ and $\{p_3\}, \{p_1\}, \{p_2\}$. Any set of processes is contending due to inverted execution orders.



(b) Two ordered runs mixed orders: $\{p_1, p_2, p_3\}$ and $\{p_2\}, \{p_3, p_1\}$. The only couple of contending processes is $\{p_1, p_2\}$.



(c) The 2-contention complex shown in red.

Figure 6.3 – Representation, in a 3-processes system, of all 2-contention simplices in $\text{Chr}^2\,\mathbf{s}$ and some detailed IS runs.

Definition 6.3. *[$Cont_2$]* $\sigma \in \text{Chr}^2\,\mathbf{s} : \forall v, v' \in \sigma, v \neq v'$:

$$((View^1(v) \subsetneq View^1(v')) \wedge (View^2(v') \subsetneq View^2(v))) \vee$$
$$((View^1(v') \subsetneq View^1(v)) \wedge (View^2(v) \subsetneq View^2(v'))).$$

$Cont_2$ is inclusion-closed: any face of a 2-contention simplex is also in $Cont_2$. Thus, $Cont_2$ is a complex: the 2-*contention complex* (depicted for a 3-processes system in Figure 6.3c). Particular executions of two IS rounds are also represented in Figures 6.3a and 6.3b. In these executions, one can see that a couple of processes is contending if the execution "order" is strictly reversed in the two IS runs.

Defining $\mathcal{R}_k$. The definition of $\mathcal{R}_k$ follows quite directly from the notion of contention simplices. The affine task should be the maximal pure sub-complex of $\text{Chr}^2(\mathbf{s})$ which does not possess any face which is a contention simplex of a too large dimension. It corresponds to the *pure complement* operator, defined in Chapter 5, applied to the set of contention simplices of dimension greater than or equal to k:

Definition 6.4. *[Affine task $\mathcal{R}_k$]*

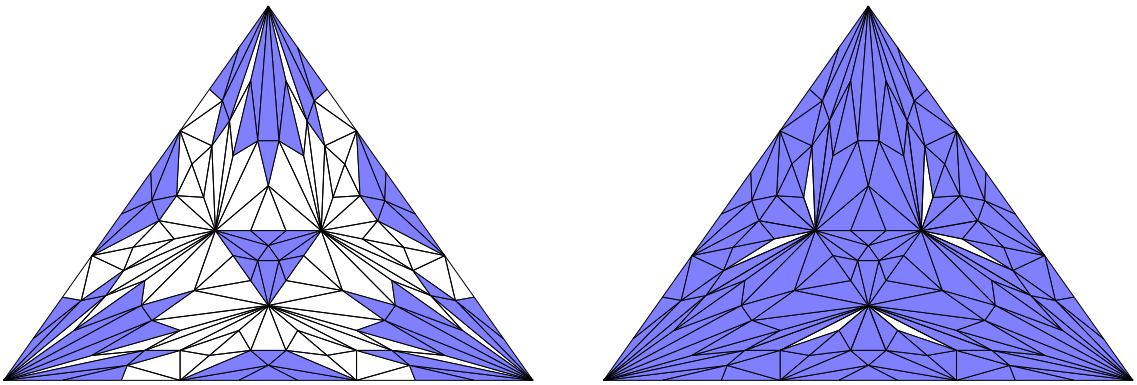
$$\mathcal{R}_k = Pc(\{\sigma \in Cont_2 | dim(\sigma) \geq k\}, \text{Chr}^2\,\mathbf{s}).$$

The definition of the pure complement ensures by construction that we obtain a pure sub-complex of $\text{Chr}^2\,\mathbf{s}$ if at least one $n-1$ dimensional simplex is valid. It can be observed that the two rounds of simultaneous executions has contending faces of dimension 0, hence that it belongs to all affine tasks $\mathcal{R}_k$, for any $k \geq 1$.

Examples of $\mathcal{R}_1$ and $\mathcal{R}_2$ for a 3-processes system are shown in Figures 6.4a and 6.4b, respectively. Obviously, for the unrestricted 3-set consensus case, $\mathcal{R}_3 = \text{Chr}^2\,\mathbf{s}$.

Solving $\mathcal{R}_k$ in the $(k-1)$-Active-Resilient Model

Since $(k+1)$-active-resilient, k-obstruction-free, k-concurrent and k-set-consensus models were shown to be all equivalent in Chapter 4, we can use any of them to solve $\mathcal{R}_k$, i.e., to



(a) $\mathcal{R}_1$ (b) $\mathcal{R}_2$

Figure 6.4 – Facets of $\mathcal{R}_1$ and $\mathcal{R}_2$ in blue on top of the edges, in black, of $\text{Chr}^2\,\mathbf{s}$.

solve the chromatic simplex agreement task on the complex $\mathcal{R}_k$. For convenience, we choose to solve it in the $(k-1)$-active-resilient model. We present here a simple algorithm which uses any immediate snapshot algorithm as a black box.

Algorithm's description. In our solution of $\mathcal{R}_k$, presented in Algorithm 6.2, every process accesses two immediate snapshot objects: *FirstIS*, to which it proposes its initial state, and *SecondIS*, to which it proposes the outcome of *FirstIS*. Hence, the outcomes of *SecondIS* form a simplex in $\text{Chr}^2\,\mathbf{s}$. To ensure that this simplex is in $\mathcal{R}_k$, after finishing *FirstIS*, processes wait for their turns to proceed to *SecondIS*.

In this *waiting phase* (Line 4), processes simply check a specific condition on the IS outcomes that they share with each other in registers $IS1[1,\ldots,n]$ and $IS2[1,\ldots,n]$. Each process p_i periodically checks whether the number of non-terminated processes ($IS2[j] \neq \emptyset$) which may have a smaller *FirstIS* view ($j \in IS1[i]$ and $IS2[j] \neq IS1[i]$) is smaller than k.

Algorithm 6.2: Solution to $\mathcal{R}_k$ in the n-processes $(k+1)$-active-resilient model for process p_i.

```
1 Shared: Registers IS1[1],...,IS1[n] ∈ 2^Π, initially ∅; IS2[1],...,IS2[n] ∈ 2^(2^Π),
    initially ∅; Immediate Snapshot Objects FirstIS, SecondIS;

2 Solve_R_k():
3 |   IS1[i] ← FirstIS(InitialState);
4 |   Wait until |{p_j ∈ IS1[i] : IS2[j] = ∅ ∧ IS1[j] ≠ IS1[i]}| < k;
5 |   IS2[i] ← SecondIS(IS1[i]);
6 |   Return(IS2[i]);
7 End Solve_R_k;
```

Intuitively, the waiting phase is used to ensure that processes which obtained smaller views in the first immediate snapshot are prioritized to proceed with their second immediate snapshot. A process may proceed to its *SecondIS* as soon as it knows that it has one of the $k-1$ smallest $View^1$ among non-terminated processes. But processes cannot directly identify such processes as they may not have shared yet their $View^1$ in $IS1$. Still, any such process must

be included in the process own View^1 along with processes sharing the same View^1. Hence, if processes with the same View^1 are withdrawn we obtain an over-estimation of how many non-terminated processes have a strictly smaller View^1. As soon as this count returns less than k processes, processes are allowed to exit the wait-phase and proceed to their *secondIS*.

Proof of correctness. Algorithm 6.2 is quite simple and so is the proof of its correctness in solving $\mathcal{R}_k$. To do so, let us prove its safety:

Lemma 6.4. *The set of values returned by Algorithm 6.2 forms a simplex from $\mathcal{R}_k$.*

Proof. First, since two immediate snapshot algorithms are used in a sequence and by using the output of the first as input of the second, the set of value returned forms a simplex in $\mathrm{Chr}^2\,\mathbf{s}$. Therefore, all we need to show is that this simplex belongs to $\mathcal{R}_k$ and hence that they do not include a contending face of dimension greater than or equal to k. Since contending simplices form a simplicial complex, we simply need to check the faces of dimension equal to k. Indeed, if there is a contending face of higher dimension, its sub-faces are also contending.

Therefore, let us consider the execution of any $k+1$ processes in Algorithm 6.2 and assume that the set of outputs of these $k+1$ processes forms a contending simplex. Since they form a contending simplex, their output vertices can be ordered in a sequence $v_0 \ldots, v_k$ such that:

$$\left(\mathit{View}^1(v_0) \subsetneq \cdots \subsetneq \mathit{View}^1(v_k)\right) \wedge \left(\mathit{View}^2(v_0) \supsetneq \cdots \supsetneq \mathit{View}^2(v_k)\right).$$

In particular, process $\chi(v_k)$ saw all other processes in its View^1 and all the k other processes considered have distinct View^1, thus they are always in the computation by $\chi(v_k)$ of $\{p_j \in \mathit{IS1}[\chi(v_k)] : \mathit{IS1}[j] \neq \mathit{IS1}[\chi(v_k)]\}$. Moreover, since $\chi(v_k)$ has the smallest View^2, it cannot have initiated its *SecondIS* after any of the k other processes terminated their execution of *SecondIS* and set their *IS2* register to a value distinct from $\perp$. Hence at the time $\chi(v_k)$ exited the wait condition, $\{p_j \in \mathit{IS1}[\chi(v_k)] : \mathit{IS1}[j] \neq \mathit{IS1}[\chi(v_k)] \wedge \mathit{IS2}[j] \neq \perp\}$ contained all the k other considered processes. But the size of this set must be strictly smaller than k to satisfy the exit condition — A contradiction. □

Let us now show that the solution presented Algorithm 6.2 is also live:

Lemma 6.5. *All correct process eventually terminate their execution of Algorithm 6.2 in the n-processes $(k+1)$-active-resilient model.*

Proof. Assume by contradiction that some correct process never completes its execution of Algorithm 6.2. Since correct processes eventually obtain outputs from immediate snapshot algorithms, a correct process can only be blocked by the wait condition of Line 4. Hence, all correct process obtained a *FirstIS* output. Let p be the correct process which never terminates with the smallest View^1. Since p has the smallest View^1 among non-terminated correct processes, it implies that all non-terminated processes observed in the View^1 of p with a distinct *IS1* (smaller or never updated) are faulty processes. As p counts at least $k+1$ such processes, there are at least $k+1$ faulty processes in the execution. But there can be at most k processes failures in the $(k+1)$-active-resilient model — A contradiction. □

Algorithm 6.2 can be used to solve any number of iterations of $\mathcal{R}_k$, and hence to solve any task solvable in $\mathcal{R}_k^*$. Hence we obtain that:

Theorem 6.7. *A task T solvable in $\mathcal{R}_k^*$ is solvable in any n-processes k-plain model.*

Simulating the k-Set Consensus Model in $\mathcal{R}_k$

Now we show how to simulate in $\mathcal{R}_k^*$ any algorithm that uses read-write memory and k-set consensus operations among subset of processes. To do so, let us first show how to simulate k-set consensus operations before showing that its works well with the composable shared-memory presented in Section 6.1.

k-set consensus simulation description. Solving k-set consensus operations in $\mathcal{R}_k^*$ is very similar to the solution of k-set consensus operations among set of $k+1$ processes presented for the affine model $\mathcal{R}_{k-T\&S}^*$. Indeed, we want all processes observed in some iteration to have a decision estimate in order to terminate. As moreover we want processes to eventually terminate as soon as they obtain the smallest view among non-terminated processes in some round, we also make processes adopt decision estimate of other processes as soon as a process is observed participating in an operation. When a process initiates a k-set consensus operation and has not updated its decision estimate yet, it simply sets its decision estimate for the operation to its input value.

The main difference is that we want any subset to solve k-set consensus operations and not only just sets of $k+1$ processes. To do so, we are going to simply change how processes adopt a new decision estimate at the end of each round. As sketched with the definition of $\mathcal{R}_k$, the idea consists in selecting a deterministically selected proposal from the smallest observed $View^1$. But not all $View^1$ may contain a decision estimate as they may not contain any process competing for the k-set consensus operation or all of them may not have a decision estimate yet. Thus, we simply select a deterministically chosen estimate from the smallest observed $View^1$ wich contains a decision estimate. After updating its decision estimate, a process may return a decision as soon as all observed processes which are concerned with the k-set consensus operation shared a decision estimate in the last iteration.

Safety of the k-set consensus simulation. The validity property is trivial as processes return their decision estimate which can only be adopted from another process decision estimate or directly from the process input value.

The agreement property is not very difficult either. Indeed, consider the first iteration after which a process p returns. As p returned, all competitors it observed shared a decision estimate. Hence, in particular, all processes which obtained a $View^1$ smaller than p and thus in all the smallest $View^1$ which are observed by any competitor. Indeed, competitors without a decision estimate see a smallest $View^1$ smaller than or identical to the one observed by p. Therefore, at the end of the round, all competitors adopt a deterministically chosen estimate from the smallest observed $View^1$ containing a competitor. If we assume that there are at most k distinct such smallest observed $View^1$, it implies that all processes terminate the round with at most k distinct decision estimates. But, in later rounds, processes can only adopt one of these decision estimates, and hence can only return one of these at most k distinct values.

The fact that at most k distinct smallest $View^1$ containing a competitor may be observed follows from the restriction on the size of contention simplices. Let us first observe that among all processes which observed a given $View^1$ as the smallest containing a competitor, one of these process obtained this $View^1$. Indeed, if the process which has the smallest $View^2$ among those with this $View^1$ observed a strictly smaller $View^1$ containing a competitor, then all process observing this $View^1$ also observed this strictly smaller $View^1$. Therefore, for all the smallest $View^1$ containing a competitor observed, we can select a process with this $View^1$ for

which its own $View^1$ is the smallest observed containing a competitor. Let $p_1, \ldots, p_m$ be these processes ordered by the size of their $View^1$. For any $i < j$, p_i has a strictly larger $View^2$ than p_j at it saw a $View^1$ not seen by p_i. Hence, for any couple of processes in this sequence, one has a strictly smaller $View^1$ and a strictly larger $View^2$ than the other: they are contending. Hence, the processes $p_1, \ldots, p_m$ form a contention simplex of dimension $m-1$. As it is a simplex of $\mathcal{R}_k$ we have $m-1 < k$, and hence at most k distinct $View^1$ are observed.

Liveness of the simultion. The composable shared memory and the k-set consensus simulations are both safe. To show the liveness of the combined simulation we need to show that if a process obtains the smallest IS output infinitely often among non-terminated processes, then it eventually terminates its pending k-set consensus operations.

This is straighforward to show. Indeed, processes can return from a k-set consensus operation as long as it sees all non-terminated competitors sharing a decision estimate in some round. But as soon as a decision estimate is observed, all non-terminated competitors adopt a decision estimate if they did not have one already. In particular, if a process shares its decision while having the smallest IS output among non-terminated process, then it will be observed by all competitors and all competitors will have a decision estimate in the next affine task iteration. Therefore the combined simulation guarantees progress to a non-terminated process.

Providing progress to a non-terminated process is sufficient to make all processes terminate when simulating a terminating algorithm such as a solution to a task. Hence we obtain the following result:

Theorem 6.8. *Any task solvable in the n-processes k-set-consensus model is solvable in $\mathcal{R}_k^*$.*

Using the equivalence of all families of plain models shown in Chapter 4, we can derive the following equivalence:

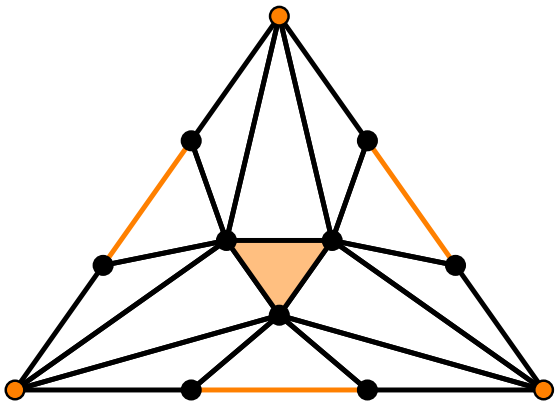
Corollary 6.1. *The n-processes k-concurrency, k-set-consensus, $(k+1)$-active-resilient and k-obstruction-free models are all equivalent to $\mathcal{R}_k^*$ in terms of task solvability.*

This equivalence result can be used to derive a generalization of the *asynchronous computability theorem* from [HS99] in its discrete formulation:

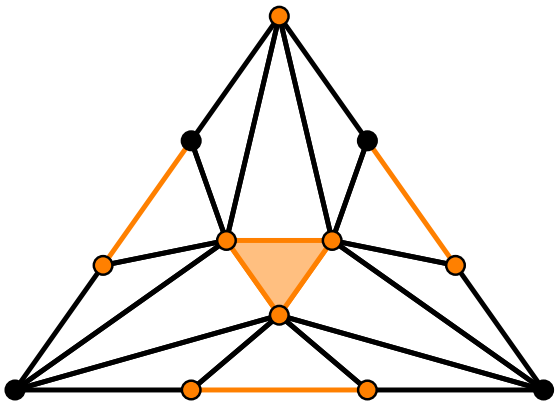
Theorem 6.9. *Plain Asynchronous Computability Theorem [PACT]:*
A task $(\mathcal{I}, \mathcal{O}, \Delta)$ is solvable in an n-processes k-plain model if and only if there exists $\ell \in \mathbb{N}$ and a color-preserving, carrier-preserving, simplicial map ϕ: $(\mathcal{R}_k)^\ell \to \mathcal{O}$.

6.4 Affine Tasks for Fair Adversaires

Let us now show how the affine tasks defined for plain models can be extended to all models in the class of fair models which corresponds to the class of fair adversarial models, or equivalently, of α-model. While much more complex than plain models, the affine task proposed for fair models is still defined using only the second iteration of the standard chromatic subdivision. This model and the equivalence result was published in [KRH18] and as a brief announcement in [KRH17].



(a) Critical simplices for the α-model with $\alpha(P) = min(|P|, 1)$ (1-obstruction-freedom)



(b) Critical simplices for the adversary defined as $\{p_2\}$, $\{p_1, p_3\}$ plus all supersets.

Figure 6.5 – Critical simplices are displayed in orange (with p_2 the top vertex, p_1 the bottom left vertex and p_3 the bottom right vertex).

6.4.1 Definition of $\mathcal{R}_\mathcal{A}$

Given a fair adversary $\mathcal{A}$ and its associated fair agreement function α, we define the affine task $\mathcal{R}_\mathcal{A}$, a sub-complex of the second degree of the standard chromatic subdivision $\text{Chr}^2\,\mathbf{s}$. In Sections 6.4.2 and 6.4.3, we show that $\mathcal{R}^*_\mathcal{A}$, i.e., the affine model of IIS runs obtained by iterating $\mathcal{R}_\mathcal{A}$, is equivalent to the α-model regarding task solvability.

Agreement vs. participation. Solving a desired level of agreement is no longer sufficient. The agreement function of an adversary may define different levels of agreement for different participating sets. In iterated affine tasks, participation is captured by views of the processes: $carrier(v, \mathbf{s})$ is the participating set witnessed by process $\chi(v)$.

The naive approach would consists in varying the restriction on the size of contention simplices according to the carrier size. Such a restriction would indeed provide an affine task which is strong enough to solve the desired level of agreement, but it would be impossible to solve. Indeed, contention assumes that processes with the smallest $View^1$ go first. But when the agreement power is equal to 0, processes must be ensured to obtain larger views and hence to let processes with larger $View^1$ go first. But letting processes with large $View^1$ go first inherently creates contention.

The idea of the solution consists in switching between resilience and concurrency requirements. Indeed, as long as the agreement power is steady over the participation, we rely on restrictions made by limiting contention. But when the agreement power increases due to an increase of participation, we identify a "witness" of this new agreement power and require it to go first and be seen by other processes. This corresponds to changing the selection of the smallest $View^1$ by looking first on $View^1$ "witnessing" a new agreement level and otherwise, by default, selecting the smallest $View^1$. These "witnesses" of participation is what we call *critical simplices*.

Critical simplices. The goal here is to identify for each increase of participation a new $View^1$ witnessing it. An easy requirement is that this $View^1$ should correspond to a participation level associated with the new level of agreement power. But two issues must be solved: (1)

the provided $View^1$ may be irregular and there could be none for a given agreement power; and (2) distinct $View^1$ may share the same level of agreement power and the smallest one may be different depending on the executions.

The idea is to select $View^1$ which are minimal in the given execution for some level of agreement power. To do so, the value of $View^1$ is not sufficient on its own. But if we know that multiple processes all share the same $View^1$, we can deduce that all other processes with a strictly smaller view must have a $View^1$ corresponding to a lower level of agreement power. This solves the second issue, but indirectly also the first one. Indeed, if no $View^1$ exists for an agreement level, it implies that the smallest view for the next level is provided to sufficiently many processes to be able to deduce that no process with a smaller $View^1$ may obtain a $View^1$ corresponding to the "missing" level, hence this $View^1$ is a witness of both agreement levels.

A *critical set* or *critical simplex* is set of processes sharing the same $View^1$ which is sufficiently large to ensure that their $View^1$ is the smallest one for some level of agreement power. Formally, a simplex $\sigma \in \mathrm{Chr}\,\mathbf{s}$ is a *critical simplex* if: (1) all its vertices share the same carrier; and (2) the set consensus power associated to $carrier(\sigma, \mathbf{s})$ is strictly greater than the set consensus power of $\chi(carrier(\sigma, \mathbf{s})) \setminus \chi(\sigma)$.

Definition 6.5. $\forall \sigma \in \mathrm{Chr}\,\mathbf{s}$, $Critical_\alpha(\sigma) \equiv$

$$(\forall v \in \sigma : carrier(v, \mathbf{s}) = carrier(\sigma, \mathbf{s})) \wedge (\alpha(\chi(carrier(\sigma, \mathbf{s})) \setminus \chi(\sigma)) < \alpha(\chi(carrier(\sigma, \mathbf{s}))))\,.$$

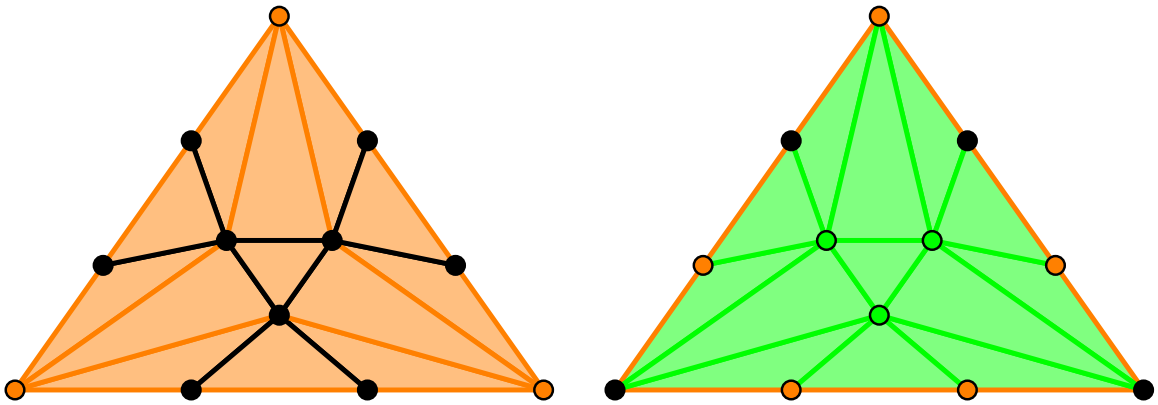
Examples of critical simplices for two 3-processes fair models are depicted in Figure 6.5. The critical simplices are displayed in orange. As it can be observed, the set of critical simplices is not inclusion-closed, hence it does not define a simplicial complex.

Given a simplex $\sigma \in \mathrm{Chr}\,\mathbf{s}$, we denote as $\mathcal{CS}_\alpha(\sigma)$ the set of critical simplices in σ, that is $\mathcal{CS}_\alpha(\sigma) = \{\sigma' \subseteq \sigma : Critical_\alpha(\sigma')\}$. Moreover, identifying the set of processes which compose some critical simplex will be useful. Thus, let $\mathcal{CSM}_\alpha(\sigma)$ (critical simplices members) be the set of vertices of some $\sigma \in \mathrm{Chr}\,\mathbf{s}$ which belongs to some critical simplex in σ, formally $\mathcal{CSM}_\alpha(\sigma) = \{\sigma' \in Cl(\mathcal{CS}_\alpha(\sigma)) : \dim(\sigma') = 0\}$. Note that critical simplices members can be seen also as a sub-complex of $\mathrm{Chr}\,\mathbf{s}$. Intuitively, processes with the smallest $View^2$ should belong to this set. Similarly we also define the notion of the critical simplex view, $\mathcal{CSV}_\alpha(\sigma)$, which corresponds to the set of processes observed by a critical simplex in its $View^1$. It can be simply obtained by taking the carrier in $\mathbf{s}$ of a critical simplex, that is $\mathcal{CSV}_\alpha(\sigma) = carrier(\mathcal{CSM}_\alpha(\sigma), \mathbf{s})$.

Concurrency level. Critical simplices provide a mechanism to select particular $View^1$. This can be used to solve agreement protocols with the desired k-set consensus for an observed participation. But unfortunately this works only when the set of processes trying to solve a set-consensus operation was observed by the critical simplices, i.e., when processes belong to $\mathcal{CSV}_\alpha(\sigma)$. When this is not the case, processes should be able to solve set-consensus operations by themselves. This is where the limition on the size of contention simplices will come in. But this limitation should still be made according to the observed participation. This is done according to the agreement power associated with the observed critical simplices. We define this restriction using the following notion of *concurrency level*:

Definition 6.6. *[Concurrency map]* $\forall \sigma \in \mathrm{Chr}\,\mathbf{s}$:

$$Conc_\alpha(\sigma) = \max(0 \cup \{\alpha(\chi(carrier(\tau, \mathbf{s}))), \tau \in \mathcal{CS}_\alpha(\sigma)\}).$$



(a) For the α-model with $\alpha(P) = min(|P|, 1)$ (1-obstruction-freedom)

(b) For the adversary defined as $\{p_2\}$, $\{p_1, p_3\}$ plus all supersets.

Figure 6.6 – Simplices in black, orange and green are mapped to concurrency levels of 0, 1 and 2 respectively (with p_2 the top vertex, p_1 the bottom left vertex and p_3 the bottom right vertex).

Note that we add 0 to the set of agreement powers in case this set is empty. The concurrency map is displayed in Figure 6.6 for examples of 3-processes models. Each simplex of $\text{Chr}\,\mathbf{s}$ is associated with a concurrency level. One can observe that the set of simplices with a concurrency level equal to k corresponds to the simplices in the *star* of the critical simplices associated with an agreement power equal to k and which are not in the *star* of a critical simplex associated with a greater agreement power.

Affine task $\mathcal{R}_\mathcal{A}$. The affine task for a fair adversary $\mathcal{R}_\mathcal{A} \subseteq \text{Chr}^2\,\mathbf{s}$ is defined as follows:

Definition 6.7. *[$\mathcal{R}_\mathcal{A}$] $\mathcal{R}_\mathcal{A} = Cl(\{\sigma \in facets(\text{Chr}^2\,\mathbf{s}) : \forall \theta \subseteq (\sigma), P(\theta, \sigma)\}$ with P such that (with $\tau = carrier(\theta, \text{Chr}\,\mathbf{s})$ and $\rho = carrier(\sigma, \text{Chr}\,\mathbf{s})$):*

$$P(\theta, \sigma) \equiv \theta \in Cont_2 \wedge (\chi(\theta) \cap \chi(\mathcal{CSM}_\alpha(\rho)) \cap \chi(\mathcal{CSV}_\alpha(\tau))) = \emptyset \implies dim(\theta) < Conc_\alpha(\tau).$$

Intuitively, a simplex $\sigma \in \text{Chr}^2\,\mathbf{s}$ is in $\mathcal{R}_\mathcal{A}$ if and only if any of its "non-critical" subsets that cannot "rely" on the critical simplices in achieving α-adaptive set consensus has a sufficiently low contention level to solve α-adaptive set consensus on its own.

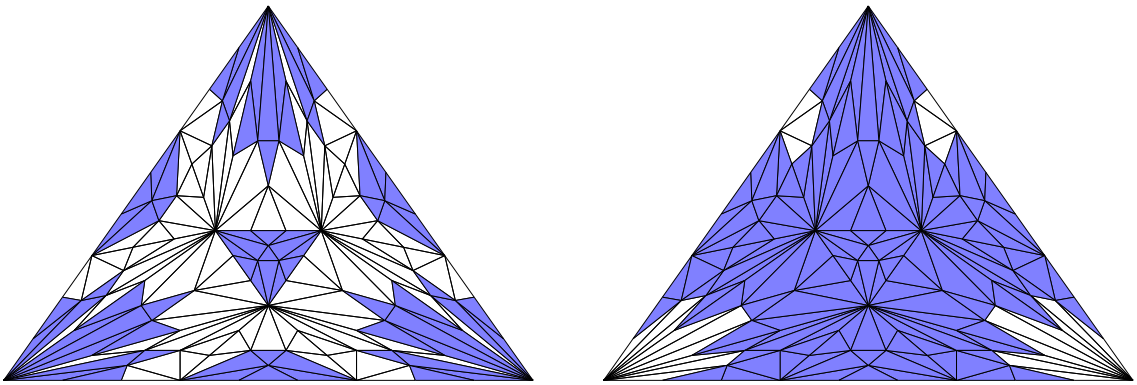
Examples of affine tasks for 3-processes α-models are depicted in Figure 6.7.

6.4.2 Solving $\mathcal{R}_\mathcal{A}$ in the α-Model

To show that any task T solvable in $\mathcal{R}_\mathcal{A}^*$ is solvable in a fair $\mathcal{A}$-model, we present an algorithm solving $\mathcal{R}_\mathcal{A}$ in the α-model. By iterating this task, we obtain $\mathcal{R}_\mathcal{A}^*$ and can solve T.

Algorithm Description

In our solution of $\mathcal{R}_\mathcal{A}$, presented in Algorithm 6.3, every process accesses two immediate snapshot objects: *FirstIS* to which it proposes its initial state, and *SecondIS* to which it proposes the outcome of *FirstIS*. Recall that outcomes of *SecondIS* form a simplex in



(a) Affine task for the α-model with $\alpha(P) = min(|P|, 1)$. (1-obstruction-freedom)

(b) Affine task for the adversary defined as $\{p_2\}$, $\{p_1, p_3\}$ plus all supersets.

Figure 6.7 – Some examples of affine tasks $\mathcal{R}_\mathcal{A}$ in blue (with p_2 the top vertex, p_1 the bottom left vertex and p_3 the bottom right vertex).

Chr^2 **s** [Koz12]. To ensure that simplices are in $\mathcal{R}_\mathcal{A}$, after finishing *FirstIS*, processes wait for their turns to proceed to *SecondIS*.

In this *waiting phase* (Lines 5–9), processes check a specific condition on the *IS* outcomes that they share with each others in registers $IS1[1, \ldots, n]$ and $IS2[1, \ldots, n]$. Each process p_i periodically checks whether either (1) it belongs to a critical simplex by using the formula at Line 7, or (2) if the number, computed at Line 8, of non-terminated processes ($IS2[j] = \emptyset$) which may have a smaller *FirstIS* output ($j \in IS1[i]$ and $IS1[j] \neq IS1[i]$) is smaller that some "level of concurrency". This level of concurrency is computed at Line 9 as the maximum between (1) the agreement power associated with the $View^1$ of the process itself ($\alpha(IS1[i])$) or (2) with the concurrency levels shared using the *Conc* registers by "terminated" critical simplices, i.e., a critical simplex with all its processes provided with *secondIS* outputs (Line 12).

Algorithm 6.3: Resolution of $R_\mathcal{A}$ in the α-model for process p_i.

```
1  Immediate Snapshot Objects: FirstIS, SecondIS;
2  Shared Registers: Conc[1], ..., Conc[n] ∈ {0, ..., n}, initially 0;
3  IS1[1], ..., IS1[n] ∈ 2^Π, initially ∅ and IS2[1], ..., IS2[n] ∈ 2^(2^Π), initially ∅;

4  R_A(input_i):
5      IS1[i] ← FirstIS(input_i);
6      wait until crit ∨ (rank < conc) with
7      crit = (α(IS1[i]) > α(IS1[i] \ {p_j ∈ Π : IS1[j] = IS1[i]}))
8      and rank = |{p_j ∈ IS1[i] : IS2[j] = ∅ ∧ IS1[j] ≠ IS1[i]}|
9      and conc = max(α(IS1[i]), max_{j∈{1,...,n}}(Conc[j]));

10     IS2[i] ← SecondIS(IS1[i]);
11     if (α(IS1[i]) > α(IS1[i] \ {p_j ∈ Π : (IS1[j] = IS1[i]) ∧ (IS2[j] ≠ ∅)})) then
12         Conc[i] ← α(IS1[i]);
13     return(IS2[i]);
14 End R_A;
```

Intuitively, the waiting phase is used to ensure that *critical processes*, i.e., members of critical simplices, are prioritized to proceed with *SecondIS* over non-critical ones. A process may proceed to its *SecondIS* as soon as it knows that it belongs to some *critical* simplex ($crit = true$). A non-critical process is allowed to exit its waiting phase only when the number of potentially contending processes is smaller than the computed concurrency level ($rank < conc$). The proof relies mostly on showing that there are enough critical simplices to prevent non-critical processes from being blocked in the waiting phase.

Proof Sketch

In order to show that Algorithm 6.3 solves $\mathcal{R}_\mathcal{A}$ in the α-model corresponding to the fair adversary $\mathcal{A}$, we need to show that (1) every correct process eventually outputs and that (2) the set of ouputs belongs to a simplex in $\mathcal{R}_\mathcal{A}$. Note that as processes execute two consecutive immediate snapshot protocols, all outputs belong to some simplex in $\mathrm{Chr}^2\,\mathbf{s}$. Let us consider a run of the α model in which the participation is P, hence with $\alpha(P) > 0$.

To show that outputs belong not only to $\mathrm{Chr}^2\,\mathbf{s}$ but to $\mathcal{R}_\mathcal{A}$ and that all correct processes terminate, we mostly rely on the distribution of critical simplices. We are interested in showing that the number of processes failures, required to prevent critical simplices from either appearing in *IS1* or completing their *IS2* computation, scales with the agreement power of the participation. Moreover, we want to show that the less processes fail in such a way, the higher the maximal agreement power associated with a terminated critical simplices.

A process failure may prevent multiple critical simplices to terminate. Indeed, a process may be included in multiple critical simplices, and thus, its failure would prevent multiple critical simplices from terminating. This is why we are interested not in the distribution of critical processes or critical simplices, but instead, in the minimal hitting set size for the set of critical simplices. Let us recall that an hitting set of a set of sets $\mathcal{Q}$, is a set intersecting with all sets from $\mathcal{Q}$, and that *csize* denotes the minimal hitting set size. More precisely, we want to know the minimal hitting set size of (1) any subset of the participation and (2) of the set of critical simplices associated with an agreement power greater than or equal to some level l, i.e., $\{\theta \in \mathcal{CS}_\alpha(\sigma), \alpha(\chi(carrier(\theta, \mathbf{s}))) \geq l\}$.

Distribution of Critical Simplices

Let us first look at the case in which no participating process fails before updating its *IS1* output to the memory. In this case, the set of *IS1* views forms a simplex $\sigma \in \mathrm{Chr}\,\mathbf{s}$ such that $\chi(\sigma) = \chi(carrier(\sigma, \mathbf{s}))$: The observed processes include all participating processes (inclusion property) but no others. In this setting we can show that the minimal hitting set size of the set of critical simplices associated with an agreement power greater than or equal to some level l, is greater than or equal to the agreement power of the participation minus $l - 1$, i.e., $\alpha(\chi(\sigma)) - l + 1$:

Lemma 6.6. *[Distribution of critical simplices]:* $\forall \sigma \in \mathrm{Chr}\,\mathbf{s}, \forall l \in \mathbb{N}$:

$$\chi(\sigma) = \chi(carrier(\sigma, \mathbf{s})) \implies \alpha(\chi(\sigma)) - l + 1 \leq csize(\{\theta \in \mathcal{CS}_\alpha(\sigma), \alpha(\chi(carrier(\theta, \mathbf{s}))) \geq l\}).$$

Proof. Let us fix some integer $l > 0$. To show Lemma 6.6, we proceed by an induction on σ using the lexicographical order on $(\alpha(\chi(\sigma)), |\chi(\sigma)|)$. For any simplex σ, such that $\alpha(\chi(\sigma)) < l$, the result is trivial as for any (possibly empty) set $\mathcal{Q}$, we have $csize(\mathcal{Q}) \geq 0$. Now consider

a simplex $\sigma \in \mathrm{Chr}\,\mathbf{s}$ such that $\chi(\sigma) = \chi(\mathit{carrier}(\sigma))$ and $\alpha(\chi(\sigma)) = k \geq l$. Let us assume by induction that for all $\sigma' \in \mathrm{Chr}\,\mathbf{s}$, if $(\alpha(\chi(\sigma')), |\chi(\sigma')|) <_{lex} (\alpha(\chi(\sigma')), |\chi(\sigma)|)$, then we have:

$$\chi(\sigma) = \chi(\mathit{carrier}(\sigma', \mathbf{s})) \implies \alpha(\chi(\sigma')) - l + 1 \leq \mathit{csize}(\{\theta \in \mathcal{CS}_\alpha(\sigma'), \alpha(\chi(\mathit{carrier}(\theta, \mathbf{s}))) \geq l\}).$$

Now consider the face τ of σ consisting of all vertices of σ with the same carrier as σ, i.e., $\tau = \{v \in \sigma, \mathit{carrier}(v, \mathbf{s}) = \mathit{carrier}(\sigma, \mathbf{s})\}$. Let β be the complement of τ, i.e., $\beta = \sigma \setminus \tau$. Note that $\tau \neq \emptyset$, due to the containment property, and that, $\chi(\mathit{carrier}(\beta, \mathbf{s})) = \chi(\mathit{carrier}(\sigma, \mathbf{s})) \setminus \chi(\tau)$, due to the immediacy property. Therefore, we obtain that $\chi(\mathit{carrier}(\beta, \mathbf{s})) = \chi(\sigma) \setminus \chi(\tau)$, and so that $\chi(\mathit{carrier}(\beta, \mathbf{s})) = \chi(\beta)$. As $(\alpha(\chi(\beta)), |\chi(\beta)|) <_{lex} (\alpha(\chi(\sigma)), |\chi(\sigma)|)$, we obtain that:

$$\alpha(\chi(\beta)) - l + 1 \leq \mathit{csize}(\{\theta \in \mathcal{CS}_\alpha(\beta), \alpha(\chi(\mathit{carrier}(\theta, \mathbf{s}))) \geq l\}). \tag{6.1}$$

Two cases may arise:

1. If $\alpha(\chi(\beta)) = \alpha(\chi(\sigma))$, then, as $\beta \subseteq \sigma$ we get that $\mathcal{CS}_\alpha(\beta) \subseteq \mathcal{CS}_\alpha(\sigma)$, hence, we can derive from Equation 6.1 that:

   $$\alpha(\chi(\sigma)) - l + 1 \leq \mathit{csize}(\{\theta \in \mathcal{CS}_\alpha(\sigma), \alpha(\chi(\mathit{carrier}(\theta, \mathbf{s}))) \geq l\}).$$

2. If $\alpha(\chi(\beta)) < \alpha(\chi(\sigma))$, then let $m = \alpha(\chi(\sigma)) - \alpha(\chi(\beta)) > 0$ and let us consider any subset τ' of τ such that $|\tau'| > |\tau| - m$. By construction we have $\mathit{carrier}(\tau', \mathbf{s}) = \mathit{carrier}(\sigma, \mathbf{s})$ and by assumption we have $\chi(\mathit{carrier}(\sigma, \mathbf{s})) = \chi(\sigma)$, and thus, we obtain that $\chi(\mathit{carrier}(\tau', \mathbf{s})) = \chi(\sigma)$. Let us recall that $\forall v \in \tau : \mathit{carrier}(v, \mathbf{s}) = \mathit{carrier}(\tau, \mathbf{s})$, and therefore $\mathit{Critical}_\alpha(\tau')$ if and only if $\alpha(\chi(\sigma) \setminus \chi(\tau')) < \alpha(\chi(\sigma))$.
   Given a fair adversary, for any $Q \subseteq P$, we have $\alpha(P) \geq \alpha(P \setminus Q) \geq \alpha(P) - |Q|$. Note that this property was shown to be true for any fair model in Chapter 4. Note that this implies that $|\chi(\tau)| \geq m$. By applying the formula for $P = \chi(\sigma) \setminus \chi(\tau')$ and for $Q = \chi(\tau) \setminus \chi(\tau')$ we get that:

   $$\alpha(\chi(\sigma) \setminus \chi(\tau')) \geq \alpha(\chi(\sigma) \setminus \chi(\tau)) \geq \alpha(\chi(\sigma) \setminus \chi(\tau')) - |\chi(\tau) \setminus \chi(\tau')|.$$

   But by construction $\chi(\sigma) \setminus \chi(\tau) = \chi(\beta)$ and $|\chi(\tau) \setminus \chi(\tau')| < m$, thus we obtain that:

   $$\alpha(\chi(\sigma) \setminus \chi(\tau)) \geq \alpha(\chi(\sigma) \setminus \chi(\tau')) - |\chi(\tau) \setminus \chi(\tau')| \implies \alpha(\chi(\sigma) \setminus \chi(\tau')) < \alpha(\chi(\beta)) + m.$$

   As $m = \alpha(\chi(\sigma)) - \alpha(\chi(\beta))$, we obtain that $\alpha(\chi(\sigma) \setminus \chi(\tau')) < \alpha(\chi(\sigma))$, and hence, that $\mathit{Critical}_\alpha(\tau')$. Since by construction $\beta = \sigma \setminus \tau$, we have the following inequality: $\mathit{csize}(\mathcal{CS}_\alpha(\sigma)) \geq \mathit{csize}(\mathcal{CS}_\alpha(\tau)) + \mathit{csize}(\mathcal{CS}_\alpha(\beta))$. Moreover, as $\alpha(\chi(\sigma)) \geq l$, we obtain:

   $$\begin{aligned}\mathit{csize}(\{\theta \in \mathcal{CS}_\alpha(\sigma), \alpha(\chi(\mathit{carrier}(\theta, \mathbf{s}))) \geq l\}) \geq \\ \mathit{csize}(\{\theta \in \mathcal{CS}_\alpha(\beta), \alpha(\chi(\mathit{carrier}(\theta, \mathbf{s}))) \geq l\}) + \mathit{csize}(\mathcal{CS}_\alpha(\tau))\end{aligned} \tag{6.2}$$

   But as any subset τ' of τ, such that $|\tau'| > |\tau| - m$, is critical, we have:

   $$\mathit{csize}(\mathcal{CS}_\alpha(\tau)) \geq \mathit{csize}(\{\tau' \subseteq \tau, |\chi(\tau')| > |\chi(\tau)| - m\}).$$

   Moreover, since $|\chi(\tau)| \geq m$, we have $\mathit{csize}(\{\tau' \subseteq \tau, |\chi(\tau')| > |\chi(\tau)| - m\}) = m$, and hence, that $\mathit{csize}(\mathcal{CS}_\alpha(\tau)) \geq m$. With $m = \alpha(\chi(\sigma)) - \alpha(\chi(\beta))$ and Equations 6.1 and 6.2, we obtain:

   $$\begin{aligned}\alpha(\chi(\sigma)) - l + 1 &= (\alpha(\chi(\beta)) - l + 1) + m \\ &\leq \mathit{csize}(\{\theta \in \mathcal{CS}_\alpha(\beta), \alpha(\chi(\mathit{carrier}(\theta, \mathbf{s}))) \geq l\}) + \mathit{csize}(\mathcal{CS}_\alpha(\tau)) \\ &\leq \mathit{csize}(\{\theta \in \mathcal{CS}_\alpha(\sigma), \alpha(\chi(\mathit{carrier}(\theta, \mathbf{s}))) \geq l\})\end{aligned}$$

□

The result of Lemma 6.6 can be used to generalize it for cases in which not all participating processes shared their *IS1* outputs to the memory. If so, the minimal hitting set size decreases proportionally with the number of missing outputs:

Corollary 6.2. *For any $\sigma \in \operatorname{Chr} \mathbf{s}$, we have:*

$$\alpha(\chi(\mathit{carrier}(\sigma, \mathbf{s}))) - l - |\chi(\mathit{carrier}(\sigma, \mathbf{s})) \setminus \chi(\sigma)| + 1 \leq$$

$$\mathit{csize}(\{\theta \in \mathcal{CS}_\alpha(\sigma), \alpha(\chi(\mathit{carrier}(\theta, \mathbf{s}))) \geq l\}).$$

Proof. Consider some $\sigma \in \operatorname{Chr} \mathbf{s}$. By construction, σ is a sub-simplex of some simplex σ' such that $\chi(\mathit{carrier}(\sigma, \mathbf{s})) = \chi(\mathit{carrier}(\sigma', \mathbf{s})) = \chi(\sigma')$. Hence, we can apply Lemma 6.6 on σ' and obtain that:

$$\alpha(\chi(\mathit{carrier}(\sigma, \mathbf{s}))) - l + 1 \leq \mathit{csize}(\{\theta \in \mathcal{CS}_\alpha(\sigma'), \alpha(\chi(\mathit{carrier}(\theta, \mathbf{s}))) \geq l\}). \tag{6.3}$$

But $\mathcal{CS}_\alpha(\sigma) \subseteq \mathcal{CS}_\alpha(\sigma')$ and thus given H a minimal hitting set of $\mathcal{CS}_\alpha(\sigma')$, $H \cup (\chi(\sigma) \setminus \chi(\sigma'))$ is an hitting set of $\mathcal{CS}_\alpha(\sigma')$. Therefore $\mathit{csize}(\{\theta \in \mathcal{CS}_\alpha(\sigma), \alpha(\chi(\mathit{carrier}(\theta, \mathbf{s}))) \geq l\})$ is greater than or equal to $\mathit{csize}(\{\theta \in \mathcal{CS}_\alpha(\sigma'), \alpha(\chi(\mathit{carrier}(\theta, \mathbf{s}))) \geq l\}) + |\chi(\sigma) \setminus \chi(\sigma')|$, and thus, is greater than or equal to $\mathit{csize}(\{\theta \in \mathcal{CS}_\alpha(\sigma'), \alpha(\chi(\mathit{carrier}(\theta, \mathbf{s}))) \geq l\}) + |\chi(\mathit{carrier}(\sigma, \mathbf{s})) \setminus \chi(\sigma)|$. Using this in Equation 6.3 gives us the property of Corollary 6.2. □

Algorithm Liveness

Corollary 6.2 is a generalization of Lemma 6.6 to account for a partial set of first immediate snapshot outputs. This can be used to show the liveness of the algorithm:

Lemma 6.7. *Algorithm 6.3 provides outputs to all correct processes in any α-model.*

Proof. Let P be the participating set and let us assume that there is a correct process which never terminates. Let p be the correct processes which does not terminate with the smallest *IS1* view, let $v \in \operatorname{Chr} \mathbf{s}$ be the vertex corresponding to its *IS1* view, and let $\sigma \in \operatorname{Chr} \mathbf{s}$ be the simplex corresponding to the set of IS^1outputs when *IS1* has been updated for the last time.

Due to the immediacy property, processes in $\chi(\mathit{carrier}(v, \mathbf{s}))$ must be associated with a vertex v' such that $\mathit{carrier}(v', \mathbf{s}) \subseteq \mathit{carrier}(v, \mathbf{s})$, and therefore, with $\alpha(\chi(\mathit{carrier}(v', \mathbf{s}))) \leq \alpha(\chi(\mathit{carrier}(v, \mathbf{s})))$. Hence, in any completion of σ to a simplex $\sigma' \in \operatorname{Chr} \mathbf{s}$ to include the processes which are in $\chi(\mathit{carrier}(v, \mathbf{s}))$ but not in $\chi(\sigma)$, the set of critical simplices associated with an agreement power strictly greater than $\alpha(\chi(\mathit{carrier}(v, \mathbf{s})))$ does not change. Thus applying Corollary 6.2 to any such completion σ' of σ, we obtain that, for any $l > \alpha(\chi(\mathit{carrier}(v, \mathbf{s})))$:

$$\alpha(\chi(\mathit{carrier}(\sigma, s))) - l - |\chi(\mathit{carrier}(\sigma, \mathbf{s})) \setminus (\chi(\sigma) \cup \chi(\mathit{carrier}(v, \mathbf{s})))| + 1 \leq$$

$$\mathit{csize}(\{\theta \in \mathcal{CS}_\alpha(\sigma), \alpha(\chi(\mathit{carrier}(\theta, \mathbf{s}))) \geq l\}).$$

Moreover, any process in $P \setminus \chi(\mathit{carrier}(\sigma, s))$ must have failed. Thus, in $\chi(\mathit{carrier}(\sigma, s))$ at most $\alpha(P) - 1 - (|P \setminus \chi(\mathit{carrier}(\sigma, s))|)$ processes may fail. Let us recall from the proof of Lemma 6.6, that for the agreement function of any fair adversary, and for any $Q \subseteq P$, we have

$\alpha(P) \geq \alpha(P \setminus Q) \geq \alpha(P) - |Q|$. Thus we can derive, by using $Q = P \setminus \chi(\mathit{carrier}(\sigma, s))$, that at most $\alpha(\chi(\mathit{carrier}(\sigma, s))) - 1$ processes in $\chi(\mathit{carrier}(\sigma, s))$ may fail.

Let $m_1 = |\chi(\mathit{carrier}(\sigma, \mathbf{s})) \setminus (\chi(\sigma) \cup \chi(\mathit{carrier}(v, \mathbf{s})))|$, be the number of processes from $\chi(\mathit{carrier}(\sigma, \mathbf{s}))$ which (1) fail before updating their *IS1* to the memory and (2) are not included in the *IS1* view of p. Let m_2 be the number of critical processes, associated with an agreement power strictly greater than $\alpha(\chi(\mathit{carrier}(v, \mathbf{s})))$, which fail after updating their *IS1* but before updating their *IS2*.

Let us now assume that $\alpha(\chi(\mathit{carrier}(\sigma, s))) - \alpha(\chi(\mathit{carrier}(v, s))) > m_1 + m_2$, then by selecting $l = \alpha(\chi(\mathit{carrier}(\sigma, s))) - m_2 - m_1$, we have $l > \alpha(\chi(\mathit{carrier}(v, s))))$, and hence, we obtain that:

$$\mathit{csize}(\{\theta \in \mathcal{CS}_\alpha(\sigma), \alpha(\chi(\mathit{carrier}(\theta, \mathbf{s}))) \geq \alpha(\chi(\mathit{carrier}(\sigma, s))) - m_2 - m_1\}) \geq m_2 + 1.$$

If no critical simplex in $\{\theta \in \mathcal{CS}_\alpha(\sigma), \alpha(\chi(\mathit{carrier}(\theta, \mathbf{s}))) \geq \alpha(\chi(\mathit{carrier}(\sigma, s))) - m_2 - m_1\}$ terminates, one process from each of these critical simplices failed after updating its *IS1* but before updating its *IS2*, thus an hitting set failed. As only m_2 such processes may fail and as an hitting set must be greater than $m_2 + 1$, a critical simplex associated with an agreement power greater than or equal to $\alpha(\chi(\mathit{carrier}(\sigma, s))) - m_2 - m_1$ terminates its *IS2*. Therefore eventually some process updates its *Conc* register (on line 12) to at least $\alpha(\chi(\mathit{carrier}(\sigma, s))) - m_2 - m_1$.

Now let us look back at p. It fails to terminate and thus never succeeds to pass the test on line 6. Therefore we have that the number of processes seen by p which do not terminate and do not have the same *IS1* view as p are strictly more than the value of $\max(\alpha(\mathit{IS1}[i]), \max_{j \in \{1,\ldots,n\}}(\mathit{Conc}[j]))$, with $\mathit{IS1}[i]$ equal to $\chi(\mathit{carrier}(v, s))$. As p is the correct process with the smallest *IS1* view which does not terminate, it implies that there are strictly more than $\max(\alpha(\chi(\mathit{carrier}(v, s))), \max_{j \in \{1,\ldots,n\}}(\mathit{Conc}[j]))$ failed processes with an *IS1* view strictly smaller than p. These failed processes are neither accounted in m_1 nor in m_2. Therefore, as at most $\alpha(\chi(\mathit{carrier}(\sigma, s))) - 1$ processes in $\chi(\mathit{carrier}(\sigma, s))$ may fail, there are at most $\alpha(\chi(\mathit{carrier}(\sigma, s))) - 1 - m_1 - m_2$ such processes which may fail. Thus $\alpha(\chi(\mathit{carrier}(\sigma, s))) - m_1 - m_2 - 1 \geq \max(\alpha(\chi(\mathit{carrier}(v, s))), \max_{j \in \{1,\ldots,n\}}(\mathit{Conc}[j]))$.

Two cases may arise:

— If $\alpha(\chi(\mathit{carrier}(\sigma, s))) - \alpha(\chi(\mathit{carrier}(v, s))) > m_1 + m_2$, then some process sets its *Conc* register to a value greater than or equal to $\alpha(\chi(\mathit{carrier}(\sigma, s))) - m_2 - m_1$ — A contradiction.

— Otherwise, $\alpha(\chi(\mathit{carrier}(\sigma, s))) - m_1 - m_2 - 1 \geq \alpha(\chi(\mathit{carrier}(v, s)))$ and so, we obtain a contradiction with the fact that $\alpha(\chi(\mathit{carrier}(\sigma, s))) - \alpha(\chi(\mathit{carrier}(v, s))) \leq m_1 + m_2$.

□

Algorithm Safety

Showing the safety of Algorithm 6.3 bears some similarities with the liveness proof. In particular, it relies on the same Lemma 6.6 on the distribution of critical simplices.

Lemma 6.8. *The set of outputs provided by Algorithm 6.3 forms a valid simplex in $\mathcal{R}_\mathcal{A}$.*

Proof. Consider any execution of Algorithm 6.3. Except for the wait-phase, processes execute two rounds of an immediate snapshot protocol. Therefore the set of outputs forms a simplex in $\sigma \in \mathrm{Chr}^2 \mathbf{s}$. Without loss of generality, we can assume that no process fails and thus that $\mathit{dim}(\sigma) = n - 1$. Indeed, if $\sigma \notin \mathcal{R}_\mathcal{A}$, then if failed processes were just slow and resumed

their execution and terminate, it would produce $\sigma' \notin \mathcal{R}_{\mathcal{A}}$. Let us assume by contradiction that $\sigma \notin \mathcal{R}_{\mathcal{A}}$, this implies that there exists $\theta \subseteq \sigma$ such that (for $\tau = \mathit{carrier}(\theta, \mathrm{Chr}\,\mathbf{s})$ and $\rho = \mathit{carrier}(\sigma, \mathrm{Chr}\,\mathbf{s})$):

$$(\theta \in \mathit{Cont}_2) \wedge ((\chi(\theta) \cap (\chi(\mathcal{CSM}_\alpha(\rho)) \cup \chi(\mathcal{CSV}_\alpha(\tau))) = \emptyset) \wedge (\mathit{dim}(\theta) \geq \mathit{Conc}_\alpha(\tau)).$$

As $\theta \in \mathit{Cont}_2$, we can order the processes associated with vertices from θ according to their IS^2 view (or $\mathit{carrier}(v, \mathrm{Chr}\,\mathbf{s})$). Let $q_1, \ldots, q_k$ be this ordered set of processes. As q_1 has the smallest IS^2 view, and as $\theta \in \mathit{Cont}_2$, q_1 also has the largest IS^1 view.

Consider the state of the execution at the time where q_1 successfully passes the test on Line 6. To pass this test, q_1 witnessed $\mathit{IS1}$, Conc and $\mathit{IS2}$ such that (with $q_1 = p_i$):

$$(\alpha(\mathit{IS1}[i]) > \alpha(\mathit{IS1}[i] \setminus \{p_j \in \Pi : \mathit{IS1}[j] = \mathit{IS1}[i]\})) \vee$$

$$\left(|\{p_j \in \mathit{IS1}[i] : \mathit{IS2}[j] = \emptyset \wedge \mathit{IS1}[j] \neq \mathit{IS1}[i]\}| < \mathit{max}\left(\alpha(\mathit{IS1}[i]), \mathit{max}_{j \in \{1,\ldots,n\}}(\mathit{Conc}[j])\right)\right)$$

If $(\alpha(\mathit{IS1}[i]) > \alpha(\mathit{IS1}[i] \setminus \{p_j \in \Pi : \mathit{IS1}[j] = \mathit{IS1}[i]\}))$, then it implies that q_1 belongs to a critical simplex. Indeed, it would belong to a set of processes sharing the same IS^1 view and such that, removing this set of processes from their IS^1 view would result in a set with a strictly smaller agreement power. But this would contradict $\chi(\theta) \cap \chi(\mathcal{CSM}_\alpha(\mathit{carrier}(\sigma, \mathrm{Chr}\,\mathbf{s}))) \neq \emptyset$ as it would include q_1. Therefore we have:

$$|\{p_j \in \mathit{IS1}[i] : \mathit{IS2}[j] = \emptyset \wedge \mathit{IS1}[j] \neq \mathit{IS1}[i]\}| < \mathit{max}\left(\alpha(\mathit{IS1}[i]), \mathit{max}_{j \in \{1,\ldots,n\}}(\mathit{Conc}[j])\right)$$

Two cases may arise:

— $\mathit{max}\left(\alpha(\mathit{IS1}[i]), \mathit{max}_{j \in \{1,\ldots,n\}}(\mathit{Conc}[j])\right) \neq \alpha(\mathit{IS1}[i])$: In this case, a register in Conc was set on Line 12 to a value greater than $\alpha(\mathit{IS1}[i])$. It implies that a critical simplex associated with an agreement level strictly greater than $|\{p_j \in \mathit{IS1}[i] : \mathit{IS2}[j] = \emptyset \wedge \mathit{IS1}[j] \neq \mathit{IS1}[i]\}|$ terminated its computation and thus is included in $\mathit{carrier}(\theta, \mathrm{Chr}\,\mathbf{s})$. But we can observe that $(\chi(\theta) \setminus \{q_1\}) \subseteq \{p_j \in \mathit{IS1}[i] : \mathit{IS2}[j] = \emptyset \wedge \mathit{IS1}[j] \neq \mathit{IS1}[i]\}$, and hence, that $\dim(\theta) < \mathit{Conc}_\alpha(\tau)$ — a contradiction with $\sigma \notin \mathcal{R}_{\mathcal{A}}$.

— $\mathit{max}\left(\alpha(\mathit{IS1}[i]), \mathit{max}_{j \in \{1,\ldots,n\}}(\mathit{Conc}[j])\right) = \alpha(\mathit{IS1}[i])$: Let c be the highest agreement power associated with a terminated critical simplex (with $c = 0$ if there is no terminated critical simplex). Therefore we have $\mathit{Conc}_\alpha(\tau) \geq c$. Let $\lambda \in \mathrm{Chr}\,\mathbf{s}$ be the simplex corresponding to the set of IS^1 views of processes in $\mathit{IS1}[i]$ which shared their IS^1 view at the time q_1 passed the test on Line 6. Consider the simplex $\lambda' \in \mathrm{Chr}\,\mathbf{s}$ corresponding to the completion of λ with the vertices corresponding to IS^1 view of the processes in $\chi(\theta)$ which may be missing from λ. Note that, since q_1 has the largest IS^1 view among processes from $\chi(\theta)$, $\chi(\mathit{carrier}(\lambda', \mathbf{s})) = \chi(\mathit{carrier}(\lambda, \mathbf{s})) = \mathit{IS1}[i]$. Moreover, since $\chi(\lambda) = \{p_j \in \mathit{IS1}[i] : \mathit{IS1}[j] \neq \perp\}$, we obtain that $\chi(\lambda') = \{p_j \in \mathit{IS1}[i] : \mathit{IS1}[j] \neq \perp\} \cup \chi(\theta)$. According to Corollary 6.2 applied to λ' with $l = c + 1$, we obtain that:

$$\alpha(\mathit{IS1}[i]) - c - |(\chi(\mathit{carrier}(\lambda', c)) \setminus \chi(\lambda'))| \leq \mathit{csize}(\{\phi \in \mathcal{CS}_\alpha(\lambda') : \alpha(\chi(\phi)) \geq c + 1\}).$$

Note that, since there is no terminated critical simplex with an agreement power greater than or equal to $c+1$, it implies that one process of each critical simplex identified in λ' did not terminate its IS^2, hence a minimal hitting set. Let S_c be this minimal hitting of size equal to $\mathit{csize}(\{\phi \in \mathcal{CS}_\alpha(\lambda') : \alpha(\chi(\phi)) \geq c + 1\}$. Note that S_c does not include any process in $\chi(\theta)$. Indeed, given a critical simplex with the same IS^1 view as q_i, adding q_i

to the critical simplex would produce a critical simplex, but by assumption processes in $\chi(\theta)$ do not belong to any critical simplex. We also have that S_c does not intersect $S_\emptyset = \{p_j \in \mathit{IS1}[i] : \mathit{IS1}[j] = \bot\}$. Hence, $|S_c| + |S_\emptyset \setminus \chi(\theta)| + |\chi(\theta)| = |S_c \cup S_\emptyset \cup \chi(\theta)|$. Therefore, as $|(\chi(\mathit{carrier}(\lambda', c)) \setminus \chi(\lambda'))| = |S_\emptyset \setminus \chi(\theta)|$ we obtain that $\alpha(\mathit{IS1}[i]) - c \leq |S_c \cup S_\emptyset \cup \chi(\theta)| - |\chi(\theta)|$.
Let us now check that $S_c \cup S_\emptyset \cup \chi(\theta) \subseteq \{q_1\} \cup S_T$, with $S_T = \{p_j \in \mathit{IS1}[i] : \mathit{IS2}[j] = \emptyset \wedge \mathit{IS1}[j] \neq \mathit{IS1}[i]\}$. All are clearly included in $\mathit{IS1}[i]$ by construction. For processes in $S_\emptyset$, since they have their register in *IS1* equal to $\bot$, it is also the case for their register in *IS2*. For processes in $\chi(\theta)$, they have a strictly smaller IS^1 view by assumption. For the IS^2 view, they will have a strictly larger view than q_1. But since q_1 did not start its second immediate snapshot protocol, processes in $\chi(\theta)$ could not have terminated it. For processes in S_c, they do not share the same IS^1 view as any process in $\chi(\theta)$ since they are members of critical simplices, in particular, they thus have a distinct IS^1 view from q_1. By assumption, they did not terminate their second immediate snapshot protocol, and hence also have their *IS2* register still equal to $\bot$.
Therefore, we have $S_c \cup S_\emptyset \cup \chi(\theta) \subseteq \{q_1\} \cup S_T$, and hence, $|S_c \cup S_\emptyset \cup \chi(\theta)| \leq 1 + |S_T|$. But since we also have $|S_T| < \alpha(\mathit{IS1}[i])$ and $\alpha(\mathit{IS1}[i]) - c \leq |S_c \cup S_\emptyset \cup \chi(\theta)| - |\chi(\theta)|$, we obtain that:

$$|S_T| < |S_c \cup S_\emptyset \cup \chi(\theta)| - |\chi(\theta)| + c \leq |S_T| + 1 - |\chi(\theta)| + c.$$

Thus $|\chi(\theta)| \leq c$. But recall that $\mathit{Conc}_\alpha(\tau) \geq c$, and so, $|\chi(\theta)| \leq \mathit{Conc}_\alpha(\tau)$ — a contradiction with $\sigma \notin \mathcal{R}_\mathcal{A}$.

□

Using Lemmata 6.7 and 6.8, we can directly derive the correctness of Algorithm 6.3:

Theorem 6.10. *Algorithm 6.3 solves task $\mathcal{R}_\mathcal{A}$ in the α-model corresponding to the fair adversary $\mathcal{A}$.*

As for other solutions of affine task, we can iterate this solution in order to simulate a run of $\mathcal{R}^*_\mathcal{A}$. Using this simulation we can therefore solve any task which is solvable in $\mathcal{R}^*_\mathcal{A}$:

Theorem 6.11. *Any task solvable in $\mathcal{R}^*_\mathcal{A}$ is solvable in the $\mathcal{A}$-model.*

6.4.3 From $\mathcal{R}^*_\mathcal{A}$ to the Fair Adversarial $\mathcal{A}$-Model

In this section, we show that any task solvable in the fair adversarial $\mathcal{A}$-model can be solved in $\mathcal{R}^*_\mathcal{A}$. This reduction is much more intricated than in the other direction. Indeed, to show that a model is as strong as an affine task based model, it only suffices to show that any number of iterations of the affine task can be solved. In the general case, it is necessary to show that any task solvable in the target model can be solved and thus that we can emulate an algorithm solving any given task.

Simulation description. To simplify the simulation complexity, we are going to show that we can simulate an execution of a shared memory model in which the participation P is such that $\alpha(P) > 0$ and in which α-adaptive set consensus can be solved. Using the equivalence between the α-adaptive set consensus model, the α-model and a fair adversary with agreement

function α shown in Chapter 4, we are able to deduce from it that any task solvable in a fair adversarial model can be solved in $\mathcal{R}^*_{\mathcal{A}}$.

As for other affine models, we are going to use the composable shared memory presented in Section 6.1 in combination with other agreement operations. The main difference comes from some extra memory operation that processes perform after each agreement operation. Indeed, in an α-adaptive set consensus operation, the level of agreement reached must correspond to the current participation. Hence, in order to ensure that participation is high enough, we make processes proceed to a simulated write of all processes inputs they observed during the execution. This operation ensures that all these processes are participating in the simulated run. Indeed, a process is participating as soon as it shared its input state: A write operation that may be performed for other processes as long as their input states are known. Processes can return the value returned by our agreement protocol only after this simulated write operation is completed.

Concerning the α-adaptive set consensus operation itself, the simulation is quite similar to the one proposed for plain models. Indeed, we make processes adopt decision estimate as soon as a process is observed participating. Moreover, we want to ensure that the set of competitors in the agreement protocol can be arbitrary. As before, we make processes adopt decision estimates from selected processes after each iteration and let them validate their decision estimate as soon as all processes observed in some iteration shared a decision estimate. The only differnce concerns the selection of which decision estimate to adopt at each round. This selection and the property associated with it are detailled in the next part. Afterwards, we will show how the property of this selection can be used to show the correctness of the global simulation.

α-adaptive leader election in $\mathcal{R}_{\mathcal{A}}$: the μ_Q map

Let us consider some α-adaptive set consensus and let Q be the set of processes which (1) may participate in the agreement protocol, and, (2) did not terminate yet the main simulation. Using the structure of $\mathcal{R}_{\mathcal{A}}$, we construct a map μ_Q which returns to each vertex $v \in \mathcal{R}_{\mathcal{A}}$, corresponding to a process from Q (i.e., with $\chi(v) \in Q$), a leader selected among Q for the given iteration of $\mathcal{R}_{\mathcal{A}}$. The map μ_Q is constructed in two stages. The first stage consists in selecting an *IS1* view which includes a process from Q. Two cases may happen depending on whether the process observes in $\mathcal{R}_{\mathcal{A}}$ a critical simplex associated with an *IS1* view including a process from Q or not:

If the process observes such a critical simplex (i.e., $\chi(\mathcal{CSV}_\alpha(\mathit{carrier}(v, \operatorname{Chr}\mathbf{s}))) \cap Q \neq \emptyset$), it then simply returns the smallest *IS1* view of a critical simplex which includes a process from Q, using the map δ_Q:

$$\delta_Q = \chi(\min(\{\mathit{carrier}(\sigma', \mathbf{s}) : (\sigma' \in CS_\alpha(\mathit{carrier}(v, \operatorname{Chr}\mathbf{s})) : \chi(\mathit{carrier}(\sigma', \mathbf{s})) \cap Q \neq \emptyset)\}).$$

Otherwise (if $\chi(\mathcal{CSV}_\alpha(\mathit{carrier}(v, \operatorname{Chr}\mathbf{s}))) \cap Q = \emptyset$), the process returns the smallest observed *IS1* view which includes a process from Q, using the map γ_Q:

$$\gamma_Q = \chi(\min(\{\mathit{carrier}(v', \mathbf{s}) : (v' \in \mathit{carrier}(v, \operatorname{Chr}\mathbf{s})) \wedge (\mathit{dim}(v') = 0) \wedge (\mathit{carrier}(v', \mathbf{s}) \cap Q \neq \emptyset)\}).$$

The second stage then simply consists in selecting, from the selected *IS1* view, the process from Q associated with the smallest identifier, let $\min_Q(V) = \min\{p \in V \cap Q\}$ be this map. The map μ_Q is therefore defined as follows:

$$\mu_Q(v) = \textbf{ if } (\chi(\mathcal{CSV}_\alpha(\mathit{carrier}(v, \operatorname{Chr}\mathbf{s}))) \cap Q \neq \emptyset) \textbf{ then } \min_Q \circ \delta_Q \textbf{ else } \min_Q \circ \gamma_Q.$$

Let us first show that, for any vertex $v \in \mathcal{R}_\mathcal{A}$ corresponding to a process in Q, the map μ_Q returns a process from Q observed in $\mathcal{R}_\mathcal{A}$ (i.e., a process in $\chi(carrier(v, \mathbf{s}))$) :

Property 6.4. *[Validity of μ_Q]* $\forall v \in \mathcal{R}_\mathcal{A}, dim(v) = 0, \chi(v) \in Q$:

$$\mu_Q(v) \in \chi(carrier(v, \mathbf{s})) \wedge \mu_Q(v) \in Q.$$

Proof. Let us fix some vertex $v \in \mathcal{R}_\mathcal{A}$ such that $\chi(v) \in Q$.

Let us assume that $\chi(\mathcal{CSV}_\alpha(carrier(v, \mathrm{Chr}\,\mathbf{s}))) \cap Q \neq \emptyset$, and hence, $\mu_Q(v) = \min_Q \circ \delta_Q(v)$. Let us recall that, given $\sigma \in \mathrm{Chr}\,\mathbf{s}$, $\mathcal{CSV}_\alpha(\sigma)$ is equal to $carrier(\cup_{\sigma' \in \mathcal{CS}_\alpha(\sigma)} \sigma', \mathbf{s})$. But due to carriers inclusion, the carrier of a simplex is equal to the carrier of one of its vertices, and so, of any sub-simplex which includes this vertex. Thus, as $\chi(\mathcal{CSV}_\alpha(carrier(v, \mathrm{Chr}\,\mathbf{s}))) \cap Q \neq \emptyset$, we have:

$$\exists \sigma' \in CS_\alpha(carrier(v, \mathrm{Chr}\,\mathbf{s})) : \chi(carrier(\sigma', \mathbf{s})) \cap Q \neq \emptyset.$$

This implies that δ_Q has a valid choice for v and can return the minimal one, and so that:

$$\exists \sigma' \in CS_\alpha(carrier(v, \mathrm{Chr}\,\mathbf{s})) : (\delta_Q(v) = \chi(carrier(\sigma', \mathbf{s}))) \wedge (\chi(carrier(\sigma', \mathbf{s})) \cap Q \neq \emptyset).$$

Since $CS_\alpha(carrier(v, \mathrm{Chr}\,\mathbf{s})) \subseteq \{\sigma \in \mathrm{Chr}\,\mathbf{s}; \sigma \subseteq carrier(v, \mathrm{Chr}\,\mathbf{s})\}$, and as $\mu_Q(v) = \min_Q \circ \delta_Q(v)$, we obtain that:

$$\exists \sigma' \subseteq carrier(v, \mathrm{Chr}\,\mathbf{s}) : (\mu_Q(v) = \min_Q \circ \chi(carrier(\sigma', \mathbf{s}))) \wedge (\mu_Q(v) \in Q).$$

As for any simplex $\sigma \in \mathrm{Chr}^2\,\mathbf{s}$, we have $carrier(carrier(v, \mathrm{Chr}\,\mathbf{s}), \mathbf{s}) = carrier(v, \mathbf{s})$, thus Property 6.4 is verified if $\chi(\mathcal{CSV}_\alpha(carrier(v, \mathrm{Chr}\,\mathbf{s}))) \cap Q \neq \emptyset$.

Now let us assume that $\chi(\mathcal{CSV}_\alpha(carrier(v, \mathrm{Chr}\,\mathbf{s}))) \cap Q = \emptyset$. Due to the self-inclusion property, $\exists v' \in carrier(v, \mathrm{Chr}\,\mathbf{s}))$ such that $\chi(v') = \chi(v)$. The self-inclusion property again implies that $\exists v'' \in carrier(v', \mathbf{s})$ such that $\chi(v'') = \chi(v') = \chi(v)$. Hence, as $\chi(v) \in Q$, $\exists v' \in carrier(v, \mathrm{Chr}\,\mathbf{s})$ such that $\chi(carrier(v', \mathbf{s})) \cap Q \neq \emptyset$. Thus γ_Q has a valid choice for v and can return the minimal one. As before, by the transitivity of carriers inclusion, the set returned by γ_Q, and so the process returned by μ_Q, is a subset of $\chi(carrier(v, \mathbf{s}))$ which intersects with Q. □

Now that we have checked that μ_Q is well defined, let us show that μ_Q returns a number of distinct leaders (processes) limited by the agreement power associated with processes views in $\mathcal{R}_\mathcal{A}$:

Property 6.5. *[Agreement of μ_Q]* $\forall Q \subseteq \Pi, (\forall \sigma \in \mathcal{R}_\mathcal{A} : dim(\sigma) = n-1), (\forall \theta \subseteq \sigma : \chi(\theta) \subseteq Q)$:

$$|\{\mu_Q(v) : v \in \theta\}| \leq \alpha(\chi(carrier(\theta, \mathbf{s}))).$$

Let us first check the following observation stating that for any simplex $\sigma \in \mathrm{Chr}\,\mathbf{s}$, if two critical simplices in σ are associated with the same agreement power, then they share the same *IS1* view:

Lemma 6.9. $\forall \sigma \in \mathrm{Chr}\,\mathbf{s}$, $\forall \theta_1, \theta_2 \in \mathcal{CS}_\alpha(\sigma)$*:*

$$\alpha(\chi(carrier(\theta_1, \mathbf{s}))) = \alpha(\chi(carrier(\theta_2, \mathbf{s}))) \implies carrier(\theta_1, \mathbf{s}) = carrier(\theta_2, \mathbf{s}).$$

Proof. Let us consider some simplex $\sigma \in \mathrm{Chr}\,\mathbf{s}$ and some critical simplices $\theta_1, \theta_2 \in \mathcal{CS}_\alpha(\sigma)$ such that $\alpha(\chi(\mathit{carrier}(\theta_1, \mathbf{s}))) = \alpha(\chi(\mathit{carrier}(\theta_2, \mathbf{s})))$. The inclusion property implies, w.l.o.g., $\mathit{carrier}(\theta_1, \mathbf{s}) \subseteq \mathit{carrier}(\theta_2, \mathbf{s})$. The immediacy property implies either that $\mathit{carrier}(\theta_1, \mathbf{s}) = \mathit{carrier}(\theta_2, \mathbf{s})$ (and thus Lemma 6.9 is verified) or else that $\chi(\theta_2) \cap \chi(\mathit{carrier}(\theta_1, \mathbf{s})) = \emptyset$.

Let us now assume that $\chi(\theta_2) \cap \chi(\mathit{carrier}(\theta_1, \mathbf{s})) = \emptyset$. Together with $\mathit{carrier}(\theta_1, \mathbf{s}) \subseteq \mathit{carrier}(\theta_2, \mathbf{s})$, it implies that $\mathit{carrier}(\theta_1, \mathbf{s}) \subseteq \mathit{carrier}(\theta_2, \mathbf{s}) \setminus \theta_2$. Since agreement functions are regular (i.e., the agreement power can only grow with a participation increase), we obtain that $\alpha(\chi(\mathit{carrier}(\theta_1, \mathbf{s})) \leq \alpha(\chi(\mathit{carrier}(\theta_2, \mathbf{s}) \setminus \theta_2)$. But as θ_2 is a critical simplex $\alpha(\chi(\mathit{carrier}(\theta_2, \mathbf{s}) \setminus \theta_2)) < \alpha(\chi(\mathit{carrier}(\theta_2, \mathbf{s})))$, and we obtain a contradiction:

$$\alpha(\chi(\mathit{carrier}(\theta_1, \mathbf{s})) \leq \alpha(\chi(\mathit{carrier}(\theta_2, \mathbf{s}) \setminus \theta_2) < \alpha(\chi(\mathit{carrier}(\theta_2, \mathbf{s}))) = \alpha(\chi(\mathit{carrier}(\theta_1, \mathbf{s}))).$$

□

Let us now prove Property 6.5:

Proof. Let σ be a maximal simplex of $\mathcal{R}_\mathcal{A}$, i.e., $\mathit{dim}(\sigma) = n - 1$, and let $\theta \subseteq \sigma$ such that $\chi(\theta) \subseteq Q$.

Note that for both γ_Q and δ_Q, processes returns the *IS1* view of a vertex of $\mathit{carrier}(\theta, \mathrm{Chr}\,\mathbf{s})$. Assume that γ_Q and δ_Q return, for vertices in θ, $k \geq 0$ distinct *IS1* views which are not the *IS1* views of some critical simplex in $\mathit{carrier}(\sigma, \mathrm{Chr}\,\mathbf{s})$. As δ_Q only returns *IS1* views associated with a critical simplex, they have been returned by γ_Q. Let β be the subset of θ including all vertices for which γ_Q returns such *IS1* views. As they are returned by γ_Q, we have $\mathcal{CSV}_\alpha(\mathit{carrier}(\beta, \mathrm{Chr}\,\mathbf{s})) \cap Q = \emptyset$.

Consider any two processes p_1 and p_2 which obtained two distinct such *IS1* views, V_1 and V_2 respectively (w.l.o.g., let $V_1 \subsetneq V_2$). As γ_Q returns the minimal *IS1* view intersecting with Q, a vertex from β sees V_2 but not V_1, and thus, p_2 has a smaller *IS2* view than p_1. Therefore p_1 and p_2 satisfy the condition to be part of a contention simplex, and so, any k processes carrying these k distinct returned *IS1* views form a contention simplex. Let τ be this contention simplex in σ.

As a vertex in β saw all these k distinct *IS1* views, therefore we have $\mathit{carrier}(\tau, \mathrm{Chr}\,\mathbf{s}) \subseteq \mathit{carrier}(\beta, \mathrm{Chr}\,\mathbf{s})$. But since $\mathcal{CSV}_\alpha(\mathit{carrier}(\beta, \mathrm{Chr}\,\mathbf{s})) \cap Q = \emptyset$ is satisfied, we obtain that $\mathcal{CSV}_\alpha(\mathit{carrier}(\tau, \mathrm{Chr}\,\mathbf{s})) \cap Q = \emptyset$. By assumption, these k processes are not critical simplices members ($\chi(\tau) \cap \mathcal{CSM}_\alpha(\sigma) = \emptyset$). Therefore, the definition of $\mathcal{R}_\mathcal{A}$ implies that we have $\mathit{Conc}_\alpha(\mathit{carrier}(\tau, \mathrm{Chr}\,\mathbf{s})) \geq k$, and hence, we obtain that $\mathit{Conc}_\alpha(\mathit{carrier}(\beta, \mathrm{Chr}\,\mathbf{s})) \geq k$.

Having $\mathit{Conc}_\alpha(\mathit{carrier}(\beta, \mathrm{Chr}\,\mathbf{s})) \geq k$, it implies that we have $\exists \sigma_c \in \mathcal{CS}_\alpha(\mathit{carrier}(\beta, \mathrm{Chr}\,\mathbf{s}))$ such that $\alpha(\chi(\mathit{carrier}(\sigma_c, \mathbf{s}))) \geq k$. But, since $\chi(\mathit{carrier}(\sigma_c, \mathbf{s})) \subseteq \mathcal{CSV}_\alpha(\mathit{carrier}(\beta, \mathrm{Chr}\,\mathbf{s}))$, we have $\chi(\mathit{carrier}(\sigma_c, \mathbf{s})) \cap Q = \emptyset$. As the inclusion property implies that any *IS1* view must be strictly larger to intersect with Q, and as there are at most one *IS1* view associated with a critical simplex by agreement level (Lemma 6.9), all *IS1* views corresponding to some critical simplex in $\mathit{carrier}(\sigma, \mathrm{Chr}\,\mathbf{s})$ are associated with an agreement power strictly greater than k.

Let $l \geq 0$ be the number of distinct *IS1* views corresponding to some critical simplex in $\mathit{carrier}(\sigma, \mathrm{Chr}\,\mathbf{s})$ which are returned by δ_Q or γ_Q for vertices in θ. Lemma 6.9 implies that they must be associated with l distinct agreement powers. As they must also be associated with agreement powers strictly greater than k, one of the returned *IS1* views is associated with an agreement power greater than or equal to $k + l$. Therefore, we have $\alpha(\chi(\mathit{carrier}(\theta, \mathbf{s}))) \geq k + l$. As the number of distinct *IS1* views returned by δ_Q or γ_Q is equal to $k + l$, and as the

deterministic selection made by $\min_Q$ could only reduce the number of distinct returned values, we finally obtain that $|\{\mu_Q(v) : v \in \theta\}| \leq \alpha(\chi(carrier(\theta, \mathbf{s})))$. □

Last, let us also observe that knowing which processes terminated the main simulation is not required to compute μ_Q, i.e., that the knowledge of which processes belong to Q among the processes observed in the current iteration of $\mathcal{R}_\mathcal{A}$ is sufficient:

Property 6.6. *[Robustness of μ_Q]* $\forall v \in \mathcal{R}_\mathcal{A}, dim(v) = 0, \forall Q \subseteq \Pi : \mu_Q(v) = \mu_{carrier(v,\mathbf{s})\cap Q}(v)$.

Proof. This is a direct corollary of the definition of δ_Q and γ_Q, that for a given vertex $v \in \mathcal{R}_\mathcal{A}$, to compute $\mu_Q(v)$, the knowledge of $Q \cap (carrier(v, \mathbf{s}))$ is sufficient. Indeed, Q is only used to compute intersections with either $\mathcal{CSV}_\alpha(\chi(carrier(v, \mathrm{Chr}\,\mathbf{s})))$, a subset of $carrier(v, \mathbf{s})$, or with $carrier(v', \mathbf{s})$ for a vertex $v' \in carrier(v, \mathrm{Chr}\,\mathbf{s})$, also a subset of $carrier(v, \mathbf{s})$. □

Correctness of the simulation

Let us first show that all simulated operations are safe. Since the composable shared memory simulation is safe, we only need to show that simulated α-adaptive set consensus operations satisfy the validity property (decision values are proposal values) and the α-agreement property (if k distinct values have been returned, then the current participation P is such that $\alpha(P) \geq k$).

Lemma 6.10. *The shared memory and α-adaptive set consensus simulation in $\mathcal{R}^*_\mathcal{A}$ is safe.*

Proof. For α-adaptive set consensus operations, Property 6.6 ensures that $\mu_{carrier(v,\mathbf{s})\cap Q}(v)$ can be used as if it was $\mu_Q(v)$ and thus that processes can indeed use μ_Q to elect a leader in any iteration of $\mathcal{R}_\mathcal{A}$. Moreover, Property 6.4 ensures that a decision estimate is either the process proposal value or is adopted from another process with a proposal value and thus that the validity property of α-adaptive set consensus is verified.

At the earliest iteration R of $\mathcal{R}^*_\mathcal{A}$ at which a process commits a decision estimate for an α-adaptive set consensus, since a committing process only observed processes from Q with decision estimates, all processes in Q adopt a decision estimate. Moreover, Property 6.5 states that among any k processes adopting k distinct decision estimates at this iteration R, one must have observed a set of processes associated with an agreement level greater or equal to k.

Before completing an α-adaptive set consensus operation, processes make sure that all processes they observed are participating in the simulated run (by simulating for them a write operation of their initial states). Therefore, at the time a k^{th} distinct value is returned for some α-adaptive set consensus, the participation in the simulated run is associated with an agreement power greater than or equal to k, hence, the α-agreement property is verified. □

As we have shown that the simulation is safe, let us also show that it is live, i.e., that it provides outputs to all processes. For this, we only need to show that a process obtaining the smallest IS view among non-terminated processes eventually completes its α-adaptive set consensus operation.

Lemma 6.11. *In the shared memory and α-adaptive set consensus simulation in $\mathcal{R}^*_\mathcal{A}$, all processes eventually terminate.*

Proof. as soon as they observe a process with a decision estimate, processes adopt a decision estimate for any α-adaptive set consensus operation they may participate to. Hence, if a

process has a pending α-adaptive set consensus operation and has the smallest IS view among non-terminated processes, all competitors will have a decision in the next iteration of the affine task. But since a process commits a decision estimate when all non-terminated processes which participate in an operation share a decision estimate during some iteration, a process obtaining the smallest IS view among non-terminated processes eventually commits its operation in the following affine task iteration and hence resume its shared-memory simulation. □

By Lemmata 6.10 and 6.11, the simulation that we provide can be used to solve in $\mathcal{R}^*_{\mathcal{A}}$ any task solvable in a shared memory model with access to α-adaptive set consensus. But it was shown in Chapter 4 that the α-adaptive set consensus model, the α-model and a corresponging fair adversarial model are all equivalent in term of task solvability. Hence, since we have shown in Theorem 6.11 that all task solvable in $\mathcal{R}^*_{\mathcal{A}}$ are solvable in the $\mathcal{A}$-model, we obtain the following equivalence result:

Theorem 6.12. *A task is solvable in the adversarial $\mathcal{A}$-model if and only if it is solvable in $\mathcal{R}^*_{\mathcal{A}}$.*

We thus obtain the following generalization of the ACT [HS99]:

Theorem 6.13. *[Fair Asynchronous Computability Theorem [FACT]] A task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is solvable in the fair adversarial $\mathcal{A}$-model if and only if there exists a natural number ℓ and a simplicial map $\phi : \mathcal{R}^\ell_{\mathcal{A}}(\mathcal{I}) \to \mathcal{O}$ carried by Δ.*

Chapter 7

Stable Storage in Comparison-Based Models

This chapter deals with complexity issues in implementing a *stable storage* in the comparison-based model. Most of the results of this chapter were published in [IKR17]. We first discuss, in Section 7.1, the model and motivate the problem of stable storage implementation. Characteristics of the problem are formalized in Section 7.2. Then, in Section 7.3, we generalize existing implementations to provide alternative liveness guarantees, with variable space complexity costs. Afterwards in Section 7.4, we show our main result, a lower bound of $n + 1$ registers required for a 2-obstruction-free stable storage implementation. This shows the existence of a trade-off between progress conditions and space complexity. Lastly, in Section 7.5, we discuss and show coarse results concerning multiple related problems. Some concluding remarks are provided in Section 7.6.

7.1 Motivation

We consider a distributed computing model in which at most n *participating* processes communicate via reading and writing to a shared memory. The participating processes come from a possibly unbounded set of *potential* participants: each process has a unique identifier (IP address, RFID, MAC address, etc.) which, without loss of generality, are integer values.

Given that processes do not have an *a priori* knowledge of the participating set, it is natural to assume that they can only *compare* their identifiers to establish their relative order, otherwise they essentially run the same algorithm [Ram30]. This model is therefore called *comparison-based* [ABND$^+$90]. In the comparison-based model with bounded shared memory, we cannot assume that the processes are provided with a prior assignment of processes to distinct registers. The only suitable assumption, as is the case for anonymous systems [Yan16], is that processes have access to *multi-writer multi-reader* registers (MWMR).

In this chapter, we study the *space complexity* of comparison-based implementations of an *abstract* single-writer multi-reader (SWMR) memory. The abstract SWMR memory allows each participating process to *write* to a private abstract memory location and to *read* from the abstract memory locations of participating processes. The SWMR abstraction can be further used to build higher-level abstractions, such as renaming [ABND$^+$90] and atomic snapshot [AAD$^+$93].

To implement an SWMR memory, we need to ensure that every write performed by a

participating process on its abstract SWMR register is *persistent*: every future abstract read must see the written value, as long as it has not been replaced by a more recent *persistent* write. To achieve persistence in a MWMR system, the emulated abstract write may have to update *multiple* base MWMR registers in order to ensure that its value is not overwritten by other processes. A natural question arises: *How many base MWMR registers do we need?*

In this chapter, we show that the answer depends on the desired progress condition. It is immediate that n registers are required for a *lock-free* implementation, i.e., we want to ensure that at least *one* correct process makes progress. Indeed, any algorithm using $n-1$ or less registers can be brought into the situation where *every* base register is *covered*, i.e., a process is about to execute a write operation on it [BL93]. If we let the remaining process p_i complete a new abstract write operation, the other $n-1$ processes may destroy the written value by making a *block write* on the covered registers (each covering process performs its pending write operation). Thus, the value written by p_i is "lost": no future read would find it. It has been recently shown that n base registers are not only necessary, but also sufficient for a lock-free implementation [DGFGR15].

A *wait-free* SWMR memory implementation that guarantees progress to *every* correct process can be achieved with $2n-1$ registers [DGFGR15]. The two extremes, lock-freedom and wait-freedom, suggest an intriguing question: is there a dependency between the amount of progress the implementation provides and its space complexity: if processes are guaranteed more progress, do they need more base registers?

Contributions

In this chapter, we give an evidence of such a dependency. Using novel covering-based arguments, we show that any 2-*obstruction-free* algorithm requires $n+1$ base MWMR registers. Recall that k-obstruction-freedom requires that every correct process makes progress under the condition that at most k processes are correct [Tau09]. The stronger property of *k-lock-freedom* [BG15] additionally guarantees that if more than k processes are correct, then at least k out of them make progress.

We also provide, for any $k = 1, \ldots, n$, a k-lock-free SWMR memory implementation that uses only $n+k-1$ base registers. Our lower bound and the algorithm suggest the following:

Conjecture 7.1. *It is impossible to implement a k-obstruction-free SWMR memory in the n-process comparison-based model using $n+k-2$ MWMR registers.*

An interesting implication of our results is that k-lock-free and k-obstruction-free SWMR implementations, for $k = 1$, $k = 2$ and $k = n$ (in this latter case they coincide with wait-freedom), have the same optimal space complexity. Hence, we expect that, for all $k = 1, \ldots, n$, k-obstruction-free and k-lock-free (and all progress conditions in between [BG15]) require the same number $n+k-1$ of base MWMR registers. Curiously, our results highlight a contrast between complexity and computability, as we know that certain problems, e.g., consensus, can be solved in an obstruction-free way, but not in a lock-free way [HLM03].

We also provide a discussion of variants of the SWMR implementation problem. The stronger variant of SWMR allocation, in which processes are associated with a single register is such that, once allocated, it is prohibited from being written by any other process. If SWMR allocation is possible, then SWMR implementation can be easily resolved with emulated write operations consisting in a single write operation of the allocated register. A wait-free SWMR implementation is thus trivially constructed. We also consider how the problem is

affected when we remove assumptions on the system such as the bound on the number of participants n or identifiers availability, i.e., an anonymous system. In particular we show that if both assumptions are removed, then the space complexity no longer depends on the number of participating processes but instead on the number of distinct written values.

Related Work

Jayanti, Tan and Toueg [JTT96] gave linear lower bounds on the space complexity of implementing a large class of *perturbable* objects (such as CAS and counters). For atomic-snapshot algorithms, Fatourou, Ellen and Ruppert [FFR06] showed that there is a tradeoff between the time and space complexities, both in the anonymous and non-anonymous cases. Zhu [Zhu16] showed that $n-1$ MWMR registers are required for obstruction-free consensus.

Delporte et al. [DFKR15] studied the space complexity of anonymous k-set agreement using MWMR registers, and showed a dependency between space complexity and progress conditions. In particular, they provide a lower bound of $n-k+m$ MWMR registers to solve anonymous *repeated* k-set agreement in the m-obstruction-free way, for $k < m$. Delporte et al. [DFGR13] showed that obstruction-free k-set agreement can be solved in the n-process comparison-based model using $2(n-k)+1$ registers. This upper bound was later improved to $n-k+m$ for the progress condition of m-obstruction-freedom ($m \leq k$) by Bouzid, Raynal and Sutra [BRS15]. In particular, their algorithm uses less than n registers when $m < k$.

To our knowledge, the only lower bound on the space-complexity of implementing an SWMR memory has been given by Delporte et al. [DGFGR15] who showed that lock-free comparison-based implementations require n registers.

Delporte et al. [DGFGR15] proposed two SWMR memory implementations: a lock-free one, using n registers, and a wait-free one, using $2n-1$ registers. These algorithms are used in [DGFGL13] to implement a *uniform* SWMR memory, i.e., assuming no prior knowledge on the number of participating processes.

7.2 Model

We consider the asynchronous shared-memory model, in which a bounded number $n > 1$ of asynchronous crash-prone processes communicate by applying read and write operations to a bounded number m of *base* atomic multi-writer multi-reader atomic registers. An atomic register i can be accessed with two memory operations: *write*(i, v) that replaces the content of the register with value v, and *read*(i) that returns its content. The processes are provided with unique identifiers from an unbounded name space. Without loss of generality, we assume that the name space is the set of positive integers.

Comparison-Based Algorithms

We assume that the processes are allowed to use their identifiers only to compare them with the identifiers of other processes: the outputs of the algorithm only depend on the inputs, the relative order of the identifiers of the participating processes, and the schedule of their steps. Formally, we say that an algorithm is *comparison-based*, if, for each possible execution α, by replacing the identifiers of participating processes with new ones preserving their relative order, we obtain a valid execution of the algorithm. Notice that the assumption does not

preclude using the identifiers in communication primitives, it only ensures that decisions taken in the algorithm's run are taken only based on the identifiers relative order.

In this model, m MWMR registers can be used to implement a wait-free m-component *multi-writer atomic-snapshot* memory [AAD$^+$93]. The memory exports operations *Update*(i, v) (updating position i of the memory with value v) and *Snapshot*() (atomically returning the contents of the memory). In the comparison-based atomic-snapshot implementation, easily derived from the original one [AAD$^+$93], *Update*(i, v) writes only once, to register i, and *Snapshot*() is read-only. For convenience, in our upper-bound algorithm we are going to use atomic snapshots instead of read-write registers.

SWMR Memory

A single-writer multi-reader (SWMR) memory exports two operations: *Write*() that takes a value as a parameter and *Collect*() that returns a *multi-set* of values. It is guaranteed that, in every execution, there exists a reading map π that associates each complete Collect operation C, returning a multi-set $V = \{v_1, \ldots, v_s\}$, with a set of s Write operations $\{w_1, \ldots, w_s\}$ performed, respectively, by distinct processes $p_1, \ldots, p_s$ such that:

— The set $\{p_1, \ldots, p_s\}$ contains all processes that completed at least one write operation before the invocation of C;

— For each $i = 1, \ldots, s$, w_i is either the last write operation of process p_i preceding the invocation of C or a write operation of p_i concurrent with C.

Note that our definition does not guarantee atomicity of SWMR operations. Moreover, we do not require that processes are allocated with a unique MWMR register that can be used as a single writer register (this additionnal restriction will be discussed in Section 7.5). Instead, we simply require that processes are able to emulate the use of single writer registers through implementing the SWMR memory.

Intuitively, a collect operation can be seen as a sequence of reads on *regular* registers [Lam86], each associated with a distinct participating process. Such a collect object can be easily transformed into a *single-writer* atomic snapshot abstraction [AAD$^+$93].

Progress Conditions

In this paper we focus on two families of progress conditions, both generalizing the *wait-free* progress condition, namely *k-lock-freedom* and *k-obstruction-freedom*. These two progress conditions are particular cases of a more general definition of (ℓ, k)-freedom which was proposed in [BG15].

An execution α satisfies the property of *k-lock-freedom* [BG15] (for $k \in \{1, \ldots, n\}$) if at least $\min(k, \mathit{Correct}(\alpha))$ correct processes *make progress* in it, i.e., complete infinitely many high-level operations (in our case, Writes and Collects). The special case of n-lock-freedom is called *wait-freedom*.

The property of k-obstruction-freedom [HLM03, Tau09] requires that every correct process makes progress, under the condition that there are at most k correct processes. (If more than k processes are correct, no progress is guaranteed.)

In particular, k-lock-freedom is a stronger requirement than k-obstruction-freedom (strictly stronger for $1 \leq k < n$). Indeed, both require that every correct process makes progress when there are at most k correct processes, but k-lock-freedom additionally requires that some progress is made even if there are more than k correct processes.

7.3 Upper Bound: k-Lock-Free SWMR Memory with $n+k-1$ Registers

Consider a *full-information* algorithm in which every process alternates atomic snapshots and updates, where each update performed by a process incorporates the result of its preceding snapshot. Every value written to a register will *persist* (i.e., will be present in the result of every subsequent snapshot), unless there is another process poised to write to that register. The pigeonhole principle implies that k processes can cover at most k distinct registers at the same time. Thus, if, at a given point of a run, a value is present in n registers, then the value will persist. This observation implies a simple n-register *lock-free* SWMR implementation in which a high-level Write operation alternates snapshots and updates of all registers, one by one in the round-robin fashion, until the written high-level value is present in all n registers. A high-level Collect operation can simply return the set of the most recent values (defined using monotonically growing sequence numbers) returned by a snapshot operation.

The *wait-free* SWMR memory implementation in [DGFGR15] using $2n - 1$ registers follows the n-register lock-free algorithm but, roughly, for each participating process, replaces register n with register $n - 1 + pos$, where pos is the rank of the process among the currently observed participants. This way, there is a time after which every participating process has a dedicated register to write, and each value it writes persists. In particular, every written value will be seen by all processes and will eventually be propagated to the $n - 1$ first registers.

To implement a k-lock-free SWMR memory using $n + k - 1$ registers, a process should determine, in a dynamic fashion, to which out of the last $k - 1$ registers to write. In our algorithm, by default, a Write operation only uses the first n registers, but if a process observes that some of its previous writes have been overwritten by other processes which did not observe its value, it uses extra registers to propagate its value. The number of these extra registers depends on how many other processes have been observed making progress.

Overview of the Algorithm

Our k-lock-free SWMR implementation, which uses $n + k - 1$ base MWMR registers, is presented in Algorithm 7.1.

In a Write operation, the process adds the operation to be performed to its local view (line 5). The process then attempts to add its local view, together with the outcome of a snapshot, to each of the first *WritePosMax*, initially n, registers (lines 8–15). At each loop, *WritePosMax* is set to the smaller value between the number of processes observed as concurrently active and the number of available registers (line 14). The writing process continues to do so until its Write operation value is present in at least n registers (line 15).

In this algorithm, the $k - 1$ extra registers are used according to the liveness observed by blocked processses. In order to be allowed to use the last register, a process must fail to complete its write while observing at least $k - 1$ other processes completing their own. This ensures that when a process accesses this last register, a k^{th} process is able to be observed by processes completing operations and thus will be helped to eventually complete its write.

The Collect operation is rather straightforward. It simply takes a snapshot of the memory and, for each participating process observed in the memory, it returns its most recent value (selected using associated sequence numbers, line 22).

Algorithm 7.1: k-lock-free SWMR implementation using $n+k-1$ MWMR registers.

```
1  View : list of triples of type (ValueType, IdType, ℕ), initially set to ∅;
2  opCounter ∈ ℕ, initially set to 0;

3  Write(v):
4      ActiveProcs = {id};
5      View = View ∪ (v, id, opCounter);
6      WritePos = 0;
7      WritePosMax = n;
8      do
9          Snap = MEM.snapshot();
10         ActiveProcs = ActiveProcs ∪ {pid : ∃(_, pid, c) ∈ Snap, ∀(_, pid, c') ∈ View, c >
               c'};
11         View = View ∪ Snap;
12         Update(MEM[WritePos], View);
13         WritePos = WritePos + 1 (mod WritePosMax);
14         WritePosMax = min(n + |ActiveProcs| − 1, n + k − 1);
15     while |{m ∈ {1, ..., n + k − 1}, (v, id, opCounter) ∈ Snap[m]}| < n;
16     opCounter = opCounter + 1;
17 End Write;

18 Collect():
19     Reads = MEM.snapshot();
20     V = ∅;
21     forall pid such that (_, pid, _) ∈ Reads do
22         V = V ∪ {v} with v such that
               (v, pid, max{c ∈ ℕ, (_, pid, c) ∈ Reads}) ∈ Reads;
23     Return V;
24 End Collect;
```

Algorithm Safety

At a high level, the safety of Algorithm 7.1 relies on the following property of register content stability:

Lemma 7.1. *Let, at some point of a run of the algorithm, value (v, id, c) be present in some register r and such that no process is poised to execute an update on r (i.e., no process is between taking the snapshot of MEM (line 9) and the update of r (line 12)), then at all subsequent times $(v, id, c) \in r$, i.e., the value is present in the set of values stored in r.*

Proof. Suppose that at time τ, a register r contains (v, id, c) and no process is poised to execute an update on r. Suppose, by contradiction, that r does not contain it at some time $\tau' > \tau$. Let τ_{min}, $\tau_{min} > \tau$, be the smallest time such that (v, id, c) is not in r. Therefore, a write must have been performed on r, by some process q, at time τ_{min} with a view which does not contain (v, id, c). Such a write can only be performed at line 12, with a view including the last snapshot of MEM performed by q at line 9. Process q must have performed this snapshot at some time $\tau_S < \tau$ as (v, id, c) is present in r between times τ and τ_{min} and as $\tau_S < \tau_{min}$. Thus q is poised to write on r at time τ — a contradiction. □

The persistence of the values in a specific uncovered register (Lemma 7.1) can be used to show the persistence of the value of a completed Write operation in *MEM*:

Lemma 7.2. *If process p returns from a Write operation $(v, id(p), c)$ at time τ, then for any time $\tau' \geq \tau$ there is a register containing $(v, id(p), c)$.*

Proof. Before returning from its Write operation, p takes a snapshot of *MEM* at some time τ_S, $\tau_S < \tau$ (line 9), which returns a view of the memory in which at least n registers contain the triplet $(v, id(p), c)$. As p is taking a snapshot at time τ_S, at most $n-1$ processes can be poised to perform an update on some register at time τ_S. As a process can be poised to perform an update on at most one register, there can be at most $n-1$ distinct registers covered at time τ_S. Therefore, at time τ_S, there is at least one uncovered register containing $(v, id(p), c)$, let us call it r. By Lemma 7.1, $(v, id(p), c)$ will be present in r at any time $\tau' > \tau_S$, and thus, any time $\tau' > \tau$. □

With Lemma 7.2, we can derive the safety of our SWMR memory implementation (Section 7.2):

Theorem 7.1. *Algorithm 7.1 safely implements an SWMR memory.*

Proof. It can be easily observed that a triplet (v, id, c) corresponds to a unique Write operation of a value v, performed by the process with identifier id. Therefore, a Collect operation returns a set of values proposed by Write operations from distinct processes, and thus the map π is well-defined.

By Lemma 7.2, the value (v, id, c) corresponding to a Write operation completed at time τ is present in some register r for any time $\tau' > \tau$. Thus, the set of values resulting from any snapshot operation performed after time τ contains (v, id, c). Hence, for any complete Collect operation C, $\pi(C)$ contains a value for every process which completed a Write operation before C was invoked. Also, as each value returned by a Collect is the value observed associated to the greatest sequence number for a given process, it comes from the last completed Write or from a concurrent one. □

Algorithm Liveness

We will show, by induction on k, that Algorithm 7.1 satisfies k-lock-freedom. We first show, as in [DGFGR15], that Write operations of Algorithm 7.1 are 1-lock-free:

Lemma 7.3. *Write operations in Algorithm 7.1 satisfy* 1*-lock-freedom.*

Proof. Suppose, by contradiction, that Write operations do not satisfy 1-lock-freedom. Eventually, all n first registers are infinitely often updated only by correct processes, unsuccessfully trying to complete a Write operation. Thus, eventually each of the n first registers contains the value from one of these incomplete Write operations. As there are at most $n-1$ covered registers when a snapshot is taken, one of these value is eventually permanently present in some register (Lemma 7.1). So, this value is eventually added to the local view of all correct processes, and hence, to every update of the registers. Thus, the correct process with this Write value eventually passes the test on line 15 and completes its operation — a contradiction. □

Higher progress conditions are obtained through a helping mechanism: a process making progress ensures that the processes it observes with a pending operation also make progress.

Lemma 7.4. *If a process q performing infinitely many operations sees $(v, id(p), c)$, and if p is correct, then p eventually completes its c^{th} Write operation.*

Proof. By Lemma 7.2, if process q returns from a Write operation with value $(v, id(q), c')$ at time τ, then for any time $\tau' \geq \tau$ there is a register containing $(v, id(q), c')$. But note that $(v, id(q), c')$ is written to a register only associated with q's local view. Thus, as q completes an infinite number of Write operations, each local view of q will eventually be forever present in some register, in particular $(v, id(p), c)$. Thus $(v, id(p), c)$ is eventually observed in every snapshot taken by correct processes, and, therefore, included in their local view. This implies that it will eventually be present in every register written infinitely often, in particular in the first n registers. As p is correct, it eventually sees $(v, id(p), c)$ in n registers for the test at line 15 and, thus, completes its corresponding c^{th} Write operation. □

By a simple inductive proof on k, we can show that k-lock-freedom is satisfied:

Lemma 7.5. *Write operations in Algorithm 7.1 satisfy k-lock-freedom.*

Proof. We proceed by induction on k, starting with the base case of $k = 1$ (Lemma 7.3). Suppose that Write operations satisfy ℓ-lock-freedom for some $\ell < k$. Consider, by ways of contradiction, a run in which at least $\ell + 1$ processes are correct, but only ℓ of them make progress (if such a run doesn't exist, the algorithm satisfies $(\ell + 1)$-lock-freedom). In this run, at least one correct process is eventually blocked in performing a Write operation. According to Lemma 7.4, the ℓ processes performing infinitely many Write operations eventually do not observe new values written by other processes. Therefore, by the algorithm, these processes eventually never write to the last $k - \ell > 0$ registers.

A correct process that never completes a Write operation will execute the while loop (lines 8–15) infinitely many times, and thus, will infinitely often take a snapshot and update its local view (line 9). In particular, it will eventually observe a new Write operation performed by each of the ℓ processes completing infinitely many Write operations. It will then eventually include at least $\ell + 1$ processes in its set of active processes (i.e., the ℓ processes performing infinitely many Write operations and itself). It will therefore eventually write to the $(n + \ell)^{th}$

register infinitely often. In the considered run, this register is written infinitely often only by correct processes which do not complete new Write operations. The value from at least one of such processes will then be observed by the ℓ processes making progress. By Lemma 7.4, this process will eventually complete its Write operation — a contradiction. □

Collect operations in Algorithm 7.1 clearly satisfy wait-freedom as there are no loops and MWMR snapshot operations are wait-free. Thus Lemma 7.5 and the wait-freedom of Collect operations imply that:

Theorem 7.2. *Algorithm 7.1 is a k-lock-free implementation of an SWMR memory for n processes using $n+k-1$ MWMR registers.*

7.4 Lower Bound: 2-Obstruction-Free SWMR Memory Requires $n+1$ Registers

The algorithm in Section 7.3 gives an upper bound of $n+k-1$ on the number of MWMR registers required to implement an SWMR memory satisfying the k-lock-free progress condition in the comparison-based model. In this section, we present a lower bound on the number of MWMR registers required in order to provide a 2-obstruction-free, and hence also a 2-lock-free, SWMR memory implementation.

7.4.1 Proof Overview

Our proof relies on the concepts of *covering* and *indistinguishability*.

A register is *covered* at a given point of a run if there is at least one process poised to write to it (we say that the process covers the register). Hence, a covered register cannot be used to ensure persistence of written data: by awakening the covering process, the adversarial scheduler can overwrite it. This property alone can be used to show that n registers are required for an obstruction-free (and hence also for a 1-lock-free) SWMR memory implementation [BL93], but not to obtain a lower bound of *more* than n shared resources as there is always one which remains uncovered.

Indistinguishability captures bounds on the knowledge that a process has of the rest of the system. Two system states are *indistinguishable* for a process if it has the same local state in both states and if the shared memory includes the same content. Thus, in an SWMR memory implementation, a Write operation can safely terminate only if, in all indistinguishable states, its value is present in a register that is not covered (by a process unaware of that value).

In our proof, we work with a composed notion of covering and indistinguishability. The idea is to show that there is a large set of reachable system states, indistinguishable to a given process p, in which different *sets of registers* are covered. Intuitively, if a set of registers is covered in one of these indistinguishable states, p must necessarily write to a register outside of this set in order to complete a new Write operation. Hence, if such indistinguishable states exist for *all* register strict subsets, then p must write its value to *all* registers. To perform infinitely many high-level Write operations, p must then write infinitely often to all available registers. But then any other process p' taking steps can be masked by the execution of p (i.e., any write p' makes to a MWMR register can be scheduled to be immediately overwritten by p). This way we establish that no 2-obstruction free implementation exists, as it requires that at least two processes must be able to make progress concurrently.

7.4.2 The Notion of Confusion

Assume, by contradiction, that there exists a 2-obstruction-free SWMR implementation using only n registers. To establish a contradiction, we consider a set of runs by a fixed set Π of n processes in which every process performs infinitely many Write operations with monotonically increasing arguments. Let $\mathcal{R}$ denote the set of n available registers.

Indistinguishability. A configuration C is said to be *indistinguishable* from a configuration C' for a set of processes P, if the content of all registers and the states of all processes in P are identical in C and C'. Given a set of configurations $\mathcal{D}$, let $I(\mathcal{D}, P)$ denote that any two configurations from $\mathcal{D}$ are indistinguishable for P.

We say that an execution is P-only, for a set of processes P, if it consists only of steps by processes in P. We say that a set of processes P is *hidden* in an execution α if all writes in α performed by processes in P are overwritten by some processes not in P, without any read performed by processes not from P in between. Given a sequence of steps α and a set of processes P, let $\alpha|_P$ be the sub-sequence of α containing only the steps from processes in P. Let us denote as $\mathcal{D}\alpha$ the set of all configurations reached by applying α to all configurations in $\mathcal{D}$ (note that α must be applicable to all configurations in $\mathcal{D}$).

Observation 7.1. *If a P-only execution α is applicable to a configuration C from a set of configurations $\mathcal{D}$ indistinguishable for P, i.e., $C \in \mathcal{D}$ and $I(\mathcal{D}, P)$, then α is applicable to any configuration $C' \in \mathcal{D}$, and it maintains the indistinguishability of configurations for P, i.e., $I(\mathcal{D}\alpha, P)$.*

A similar observation can be made concerning hidden executions:

Observation 7.2. *Given an execution α applicable to C, with C from a set of configurations $\mathcal{D}$ indistinguishable for P. If processes in $\Pi \setminus P$ are hidden in α, then $\alpha|_P$ is applicable to any $C' \in \mathcal{D}$, and $I((\mathcal{D}\alpha|_P) \cup \{C\alpha\}, P)$.*

Coverings and confusion. We say that a set of processes P *covers* a set of registers R in some configuration C, if for each register $r \in R$, there is a process $p \in P$ such that the next step of p in C is a write on r (the predicate is denoted $\mathit{Cover}(R, P, C)$).

Our lower bound result relies on a concept that we call *confusion*. We say that a set of processes P are *confused* on a set of registers S in a set of reachable configurations $\mathcal{D}$, denoted $\mathit{Confused}(P, S, \mathcal{D})$, if and only if:

1. $I(\mathcal{D}, P)$.
2. $|S| + |P| = n + 1$.
3. For any process $p \in \Pi \setminus P$, there exists two registers $r_p, r'_p \in S$ such that, for any configuration $D \in \mathcal{D}$, there exists $D' \in \mathcal{D}$, such that p covers r_p in D and r'_p in D', or vice versa, and D and D' are indistinguishable to all other processes:

$$\forall p \in \Pi \setminus P, \exists r_p, r'_p \in S, \forall D \in \mathcal{D}, \exists r \in \{r_p, r'_p\} :$$

$$\mathit{Cover}(\{r\}, \{p\}, D) \wedge (\exists D' \in \mathcal{D}, I(\{D, D'\}, \Pi \setminus \{p\}) \wedge \mathit{Cover}(\{r_p, r'_p\} \setminus \{r\}, \{p\}, D')).$$

4. For any strict subset R of S, there exists $D \in \mathcal{D}$ such that R is covered by $\Pi \setminus P$ in D:

$$\forall R \subsetneq S, \exists D \in \mathcal{D} : \mathit{Cover}(R, \Pi \setminus P, D).$$

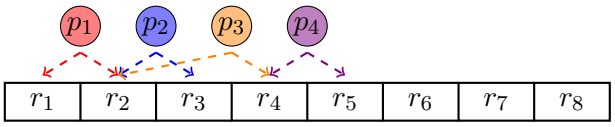


Figure 7.1 – A view of 8 registers and processes with couples of independantly covered registers by processes $p_1, \ldots, p_4$.

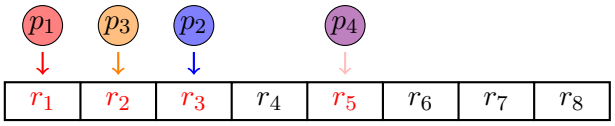


Figure 7.2 – A possible covering of r_1, r_2, r_3 and r_5 induced by the couples of independantly covered registers from Figure 7.1.

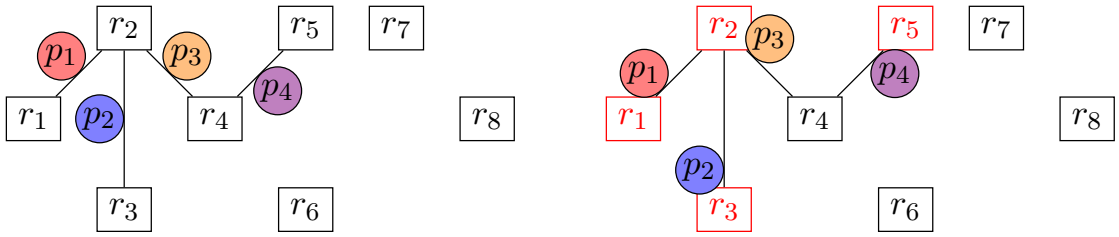
Intuitively, processes in P are confused on S in $\mathcal{D}$, if $\mathcal{D}$ is a set of indistinguishable configurations for P, such that any *strict* subset of S is covered by $\Pi \setminus P$ in some configuration of $\mathcal{D}$ (Conditions 1 and 4). We require that as much processes are confused as possible (Condition 2). Additionally, the property must hold for a set of configurations $\mathcal{D}$ in which processes not in P may cover only one out of 2 given registers, and may cover them independently of other processes states in $\mathcal{D}$ (Condition 3).

Figure 7.1 shows a set of 8 registers r_1,...,r_8, and processes associated with a couple of registers they might cover independantly of others. The set of indistinguishable configurations $\mathcal{D}$ for $\{p_5, p_6, p_7, p_8\}$ are defined via composition of states for p_1, p_2, p_3 and p_4 in which they, respectively, cover registers in $\{r_1, r_2\}$, $\{r_2, r_3\}$, $\{r_2, r_4\}$ and $\{r_4, r_5\}$. Considering the 2^4 possible combinations of resulting configurations in $\mathcal{D}$, one can check that the example shows a confusion of $\{p_5, \ldots, p_8\}$ on $\{r_1, \ldots, r_5\}$. Indeed, by construction, Conditions 1, 2 and 3 are satisfied and for any strict subset of $\{r_1, \ldots, r_5\}$ one can associate a process in $\{p_1, \ldots, p_4\}$ to cover each register in the selected subset. A possible selection to cover r_1, r_2, r_3 and r_5 is shown in Figure 7.2.

Confusion and graph connectivity.

In Figure 7.3, we propose an alternative representation of the confusion displayed in Figure 7.1. Registers are represented as nodes, and pairs of registers that a process might be covering are represented as edges. An example of a covering of $\{r_1, r_2, r_3, r_5\}$ for some particular execution is presented on the right side of Figure 7.3. The representation as a graph is interesting as it allows to easily check the validity of Condition 4. Indeed, we are going to show that Condition 4 is verified if and only if the corresponding graph is connected.

Intuitively, to show that it implies Condition 4, the proof consists in showing that any strict subset may be covered by taking one of the non-covered register as root of a spanning tree and making processes cover their edge child (see the right part of Figure 7.3). The reverse direction leverages the fact that if a non-intersecting cut is possible then, due to Condition 2, one part cannot be all covered at once. Formally, we show that Condition 4 is satisfied if and only if, for any partition of S into two non-empty subsets S_1 and S_2, there is a process in $\Pi \setminus P$ for which the two registers it may be covering in $\mathcal{D}$ intersect with both S_1 and S_2:



(a) A confusion of $\{p_5, \ldots, p_8\}$ on $\{r_1, \ldots, r_5\}$ (b) A covering of $\{r_1, r_2, r_3, r_5\}$ by $\{p_1, \ldots, p_4\}$.

Figure 7.3 – Graph representation of processes $\{p_5, p_6, p_7, p_8\}$ confused on $\{r_1, r_2, r_3, r_4, r_5\}$. A configuration corresponding with the covering from Figure 7.2 is given on the right.

Lemma 7.6. $\forall P \subseteq \Pi, \forall S \subseteq \mathcal{R}, \forall \mathcal{D} \subseteq \mathit{Reach}$ *satisfying Conditions* 1, 2 *and* 3 *of the confusion definition, we have* $\forall R \subsetneq S, \exists D \in \mathcal{D} : \mathit{Cover}(R, \Pi \setminus P, D)$ *if and only if:*

$$\forall S_1, S_2 \subseteq S, (S_1 \neq \emptyset \wedge S_2 \neq \emptyset \wedge S_1 \cup S_2 = S \wedge S_1 \cap S_2 = \emptyset) :$$

$$\exists r_1 \in S_1, r_2 \in S_2, p \in \Pi \setminus P, D_1, D_2 \in \mathcal{D} : (\mathit{Cover}(\{r_1\}, \{p\}, D_1) \wedge \mathit{Cover}(\{r_2\}, \{p\}, D_2)) .$$

Proof. Let us fix some $P \subseteq \Pi$, $S \subseteq \mathcal{R}$, and $\mathcal{D} \subseteq \mathit{Reach}$ satisfying Conditions 1, 2 and 3 of the confusion definition.

First, let us assume that Condition 4 is also satisfied and consider any partition of S into non-empty subsets S_1 and S_2 (i.e., $S_1 \neq \emptyset$, $S_2 \neq \emptyset$, $S_1 \cap S_2 = \emptyset$ and $S_1 \cup S_2 = S$). Assume now that there does not exist any process $p \in \Pi \setminus P$ such that p might be covering a register from S_1 or a register from S_2 in $\mathcal{D}$. This implies that processes in $\Pi \setminus P$ can be partitioned into two subsets Q_1 and Q_2 (with $Q_1 \cap Q_2 = \emptyset$ and $Q_1 \cup Q_2 = \Pi \setminus P$) such that processes in Q_1, respectively Q_2, may cover registers from S_1, respectively S_2, in $\mathcal{D}$. By construction of the partitions, we have $|Q_1| + |Q_2| = |\Pi \setminus P|$ and $|S_1| + |S_2| = |S|$. Using the fact that Condition 2 is satisfied by P and S we obtain from $|S| + |P| = n + 1$ that $|S_1| + |S_2| + (n - (|Q_1| + |Q_2|)) = n + 1$, and thus, that $|S_1| + |S_2| = |Q_1| + |Q_2| + 1$. This implies that either $|Q_1| < |S_1|$ or $|Q_2| < |S_2|$, w.l.o.g., let $|Q_1| < |R_1|$. Now consider $r \in S_2$, $S \setminus \{r\}$ is a strict subset of S, and therefore Condition 4 implies that there exists $D \in \mathcal{D}$ such that $\mathit{Cover}(S \setminus \{r\}, \Pi \setminus P, D)$. As registers in S_1 can only be covered by processes from Q_1, then we have $\mathit{Cover}(S_1, Q_1, D)$. Recall that, by the pigeonhole principle, a set of processes cannot cover more registers than processes it contains. But $|Q_1| < |R_1|$ — a contradiction.

Now let us assume that given any partition of S into non-empty subsets S_1 and S_2, there exists a process $p \in \Pi \setminus P$ such that p might be covering a register in S_1 or a register in S_2 in $\mathcal{D}$. Let us show that any strict subset R of S is covered in some configuration from $\mathcal{D}$ and, hence, that Condition 4 is satisfied. This is done by inductively restricting the set of configurations from $\mathcal{D}$, by selecting the maximal subset in which some process from $\Pi \setminus P$ may cover only one register. The idea is to select a process which may cover only one not-yet covered register in R, and to select the subset in which this process covers this register.

Let S_0 be a non-empty subset of S. Let p_0 be a process from $\Pi \setminus P$ which might be covering a register r_0 in S_0 or a register r'_0 in $S \setminus S_0$ in $\mathcal{D}$. Let us assume that such a process exists and consider $\mathcal{D}_0$ to be the subset of $\mathcal{D}$ including all configurations in which p_0 is covering r'_0. Now let $S_1 = S_0 \cup \{r'_0\}$ and repeat this procedure using S_1 to select some p_1 and compute $\mathcal{D}_1$, etc...

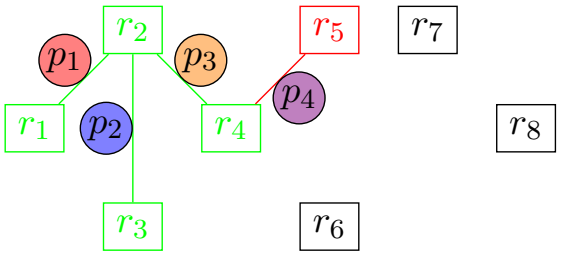


Figure 7.4 – Reduced confusion from Figure 7.3 by removing r_5 and p_4.

As long as a process can be selected satisfying the condition, the sets S_i keep increasing with i. Consider the round j at which the procedure fails to find such a process. This implies that there is no process which might be covering a register from either S_j or $S \setminus S_j$ in $\mathcal{D}_{j-1}$. Note that by construction $\mathcal{D}_{j-1}$ is a non-empty subset of $\mathcal{D}$.

If $S_j \neq S$, then S_j and $S \setminus S_j$ forms a partition of S into two non-empty subsets. Thus, by assumption, there exists a process q which might be covering a register r_q in S_j or a register r'_q in $S \setminus S_j$ in $\mathcal{D}$. Consider some configuration $D \in \mathcal{D}_{j-1}$. According to Condition 3 of the confusion definition, as $D \in \mathcal{D}$, q is covering either r_q or r'_q in D, and there exists a configuration D' in which q is covering the other register in $\{r_q, r'_q\}$, relatively to D, and such that D and D' are indistinguishable to all other processes. As $I(\{D, D'\}, \Pi \setminus \{q\})$, if D was kept in some restriction of $\mathcal{D}_{i-1}$ towards $\mathcal{D}_i$, then D' was also kept unless q was the corresponding selected process p_i. But if q was selected in an earlier iteration, both r_q and r'_q would be included in S_j. Thus D and D' belong to $\mathcal{D}_{j-1}$ and therefore q is a valid selection for p_j. This contradiction implies that $S_j = S$.

By construction, all registers in $S_j \setminus S_0$ are covered in all configurations in $\mathcal{D}_{j-1}$. As this is true for any non-empty S_0 and as $S_j = S$, any strict subset of S is covered in some configuration of $\mathcal{D}$ and therefore Condition 4 is satisfied. □

Extracting smaller confusions. Using the representation of the confusion as a connected graph, it is now relatively simple to extract a confusion on one less register. Intuitively, one of the edges of a connected graph can always be found such that its removal maitains connectivity of at least all but one node.

On acyclic graphs as for confusions (due to Condition 2), it corresponds to removing a leaf and its connecting edge. Figure 7.4 shows for example the confusion of Figure 7.3 where r_5 is removed and p_4 is no longer used, alternatively r_1 and p_1 or r_3 and p_2 could have been used.

Formally, given any confusion for some P distinct from Π, S and $\mathcal{D}$, we can identify a process $p \in \Pi \setminus P$ and a register $r \in S$ such that $P \cup \{p\}$ is confused on $S \setminus \{r\}$ for a subset $\mathcal{D}'$ of $\mathcal{D}$ which includes any given $C \in \mathcal{D}$:

Lemma 7.7. *Given* $P \subsetneq \Pi$, $S \subseteq \mathcal{R}$, $\mathcal{D} \subseteq$ *Reach:* $\mathit{Confused}(P, S, \mathcal{D}) \implies$
$\exists p \in \Pi \setminus P, \exists r \in S, \forall C \in \mathcal{D}, \exists \mathcal{D}' \subseteq \mathcal{D} : (C \in \mathcal{D}') \wedge \mathit{Confused}(P \cup \{p\}, S \setminus \{r\}, \mathcal{D}')$.

Proof. According to Condition 3, a process may be covering exactly two registers from S in $\mathcal{D}$, thus, the sum over S of how many distinct processes may cover each register is equal to $2|\Pi \setminus P| = 2(n - |P|)$. Note that any register $r \in S$ may be covered by at least one process in $\Pi \setminus P$ in $\mathcal{D}$ as any strict subset of S may be covered (Condition 4). Therefore, there exists a register $r_c \in S$ which can be covered by a single process $p_c \in \Pi \setminus P$ in $\mathcal{D}$. Indeed, if all

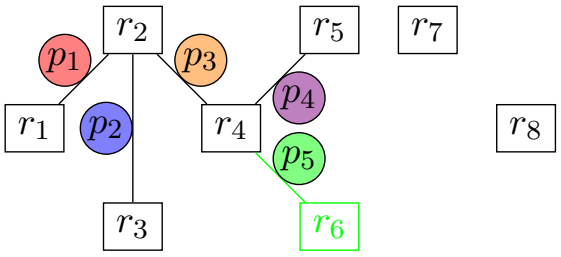


Figure 7.5 – Extended confusion from Figure 7.3 by making p_5 indistinguishably cover r_4 or r_6.

registers in S might be covered by two distinct processes, then, the sum over S of how many distinct processes may cover each register (equal to $2(n-|P|)$), would be greater than or equal to $2|S|$, or $n-|P|<|S|$ as $|P|+|S|=n+1$ (Condition 2).

Let $\mathcal{D}_c$ be the subset of $\mathcal{D}$ which includes all configurations in $\mathcal{D}$ that are indistinguishable to p_c from any configuration $C \in \mathcal{D}$. Let us show that $Confused(P \cup \{p_c\}, S \setminus \{r_c\}, \mathcal{D}_c)$. Condition 1 holds as by construction all configurations in $\mathcal{D}_c$ are indistinguishable to p_c and as they are indistinguishable to all processes in P, since $\mathcal{D}_c$ is a subset of $\mathcal{D}$. It is immediate, as we remove a register from S and add a process to P, that Condition 2 holds.

Now consider any process $p \in \Pi \setminus (P \cup \{p_c\})$ and any configuration $D \in \mathcal{D}_c$. As $p \in \Pi \setminus P$ and $D \in \mathcal{D}$, Condition 3 of the confusion definition implies that there exists $D' \in \mathcal{D}$, $I(\{D, D'\}, \Pi \setminus \{p\})$, such that p covers r_p and r'_p in D and D' (or vice-versa). Since $I(\{D, D'\}, \Pi \setminus \{p\})$, we have $D' \in \mathcal{D}_c$. Moreover, as p_c is the only process which may cover r_c in $\mathcal{D}$, r_p and r'_p must belong to $S \setminus \{r_c\}$. Condition 3 is therefore verified for $P \cup \{p\}$, $S \setminus \{r_c\}$ and $\mathcal{D}_c$.

Lastly, let us consider some partition of $S \setminus \{r_c\}$ into two non-empty subsets S_1 and S_2. Both $(S_1 \cup \{r_c\}, S_2)$ and $(S_1, S_2 \cup \{r_c\})$ form a partition of S in two non-empty subsets. Thus, as $Confused(P, S, \mathcal{D})$, we can apply Lemma 7.6 and obtain that $\exists p_1, p_2 \in \Pi \setminus P$ such that p_1, respectively p_2, might cover registers from either $S_1 \cup \{r_c\}$ or S_2, respectively either S_1 or $S_2 \cup \{r_c\}$, in $\mathcal{D}$. It follows that p_c cannot be both p_1 and p_2 as p_c might cover only two registers in $\mathcal{D}$, one of which is r_c. W.l.o.g, assume that $p_1 \neq p_c$. As p_c is the only process which may be covering r_c, this implies that p_1 might be covering a register from either S_1 or S_2.

Furthermore, since Conditions 1, 2 and 3 applies to $P \cup \{p\}$, $S \setminus \{r_c\}$ and $\mathcal{D}_c$, we can apply Lemma 7.6 to obtain that Condition 4 is also verified. □

Extending confusions. Increasing the size of the confusion can be done similarly to its reduction. We need to show that a couple of executions can be applied to all configurations of the confusion. This executions must be indistinguishable to all but one of the confused processes, and, moreover, lead this process to be covering a register from the confusion in one case and outside of the confusion in the other. Indeed, adding an edge from one node of the connected graph to a non-connected node both maintains connectivity of the edges and connects a new node. Figure 7.5 shows an example of an extended confusion from Figure 7.3.

Lemma 7.8. *Let $P \subsetneq \Pi$, $S \subseteq \mathcal{R}$ and $\mathcal{D} \subseteq Reach$ such that $Confused(P, S, \mathcal{D})$.*
Given $C \in \mathcal{D}$, if $\exists p \in P, r_1 \in S, r_2 \in \mathcal{R} \setminus S$ and if there exists P-only executions α_1 and α_2 which are applicable to C, and such that $I(\{C\alpha_1, C\alpha_2\}, \Pi \setminus \{p\})$, $Cover(\{r_1\}, \{p\}, C\alpha_1)$ and $Cover(\{r_2\}, \{p\}, C\alpha_2)$, then we have $Confused(P \setminus \{p\}, S \cup \{r_2\}, (\mathcal{D}\alpha_1) \cup (\mathcal{D}\alpha_2))$.

Proof. Following Observation 7.1, as α_1 and α_2 are P-only, and as $\mathcal{D}$ satisfies Condition 1 for P, $(\mathcal{D}\alpha_1) \cup (\mathcal{D}\alpha_2)$ satisfies Condition 1 for $P \setminus \{p\}$. Condition 2 trivially holds for $P \setminus \{p\}$ and $S \cup \{r\}$ as it holds for P and S and we remove a process from P and add a register to S.

Condition 3 is satisfied for all processes in $\Pi \setminus P$ and configurations in $(\mathcal{D}\alpha_1) \cup (\mathcal{D}\alpha_2)$ as α_1 and α_2 are P-only, and as $\mathcal{D}$ satisfies Condition 3 for any process in $\Pi \setminus P$. Moreover, as configurations in $\mathcal{D}$ are indistinguishable to $p \in P$, p may only cover r_1 if $D \in \mathcal{D}\alpha_1$ and cover r_2 if $D \in \mathcal{D}\alpha_2$. But as given any $D \in \mathcal{D}$ we have $I(\{D\alpha_1, D\alpha_2\}, \Pi \setminus \{p\})$, Condition 3 is also satisfied for p.

Since Conditions 1, 2 and 3 are satisfied, we can apply Lemma 7.6 and obtain that $\mathit{Confused}(P \setminus \{p\}, S \cup \{r_2\}, (\mathcal{D}\alpha_1) \cup (\mathcal{D}\alpha_2))$. Indeed, a partition of two non-empty subsets of $S \cup \{r_2\}$ can be reduced, unless the partition is $(S, \{r_2\})$, to a partition of two non-empty subsets of S. In this case, Lemma 7.6 can be applied for P, S and $\mathcal{D}$, which provides us, for any partition of S, with a process that may cover in $\mathcal{D}$ a register from either set of the partition. As α_1 is P-only, it still holds for $\mathcal{D}\alpha_1$. For the partition $(S, \{r_2\})$, p may cover either $r_1 \in S$ or r_2 in $(\mathcal{D}\alpha_1) \cup (\mathcal{D}\alpha_2)$. □

7.4.3 Lower Bound Proof

To establish our lower bound, we show that there is a set of reachable configuration $\mathcal{D}$ in which there is a process confused on all n registers. Intuitively, we proceed by induction on the number of "confusing" registers.

Initialization. For the base case, we show that the initial configuration can lead to a confusion of all but one process on *two registers*:

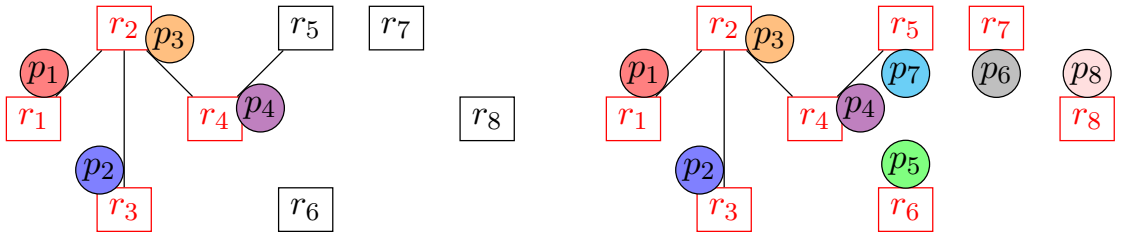
Lemma 7.9. $\exists \mathcal{D} \in \mathit{Reach}, \exists p \in \Pi, \exists S \subseteq \mathcal{R} : \mathit{Confused}(\Pi \setminus \{p\}, S, \mathcal{D})$.

Proof. Consider any two processes p_1 and p_2. Since the algorithm is comparison-based, the first write that the two processes perform in a solo execution is on the same register, let us call it r. Let p_1 execute solo until it is about to write to r and then do the same with p_2, let C be the resulting configuration. Consider the execution α from C in which p_1 executes until it is poised to write to a register $r' \neq r$ and then p_2 executes its pending write on r. This execution is valid as p_1 must eventually write to an uncovered register.

We obtain $\mathit{Confused}(\Pi \setminus \{p_1\}, \{r, r'\}, \{C\alpha, C\alpha|_{\{p_1\}}\})$. Indeed, as p_1 is hidden in α, following Observation 7.2, we have $I(\{C\alpha, C\alpha|_{\{p_1\}}\}, \Pi \setminus \{p_1\})$ (Condition 1). We have $|\Pi \setminus \{p_1\}| + |\{r, r'\}| = n + 1$ (Condition 2). As p_1 covers r' in $C\alpha$ and p_1 covers r in $C\alpha|_{\{p_1\}}$, we have Condition 4. Condition 3 directly follows from Conditions 1 and 4 in this setting. □

Induction step. We now prove our inductive step: That, given a set of configurations in which a set of processes, $P \neq \Pi$, is confused on a set of registers, $S \neq \mathcal{R}$, we can obtain a set of configurations in which a set P' of processes are confused on a set S' of strictly more than $|S|$ registers. To do this, we can show how to construct two executions applicable to $\mathcal{D}$, indistinguishable to all but one of the confused processes and such that this process covers a "confusing" register after one of the executions and a non "confusing" register after the other execution. Then Lemma 7.8 can be applied to obtain the desired extended confusion.

Lemma 7.10. $\exists \mathcal{D} \subseteq \mathit{Reach}, P \subsetneq \Pi, S \subsetneq \mathcal{R} : \mathit{Confused}(P, S, \mathcal{D})$
$\implies \exists \mathcal{D}' \subseteq \mathit{Reach}, P' \subseteq \Pi, S' \subseteq \mathcal{R}, S \subsetneq S' : \mathit{Confused}(P', S', \mathcal{D}')$.



(a) Selecting a cover of $|S|-1$ "confusing" registers. (b) Covering the rest with confused processes.

Figure 7.6 – Covering registers outside of the covering with all but one of the confused processes.

Proof. Given $\mathit{Confused}(P, S, \mathcal{D})$, consider $C \in \mathcal{D}$ such that exactly $|S| - 1$ registers in S are covered by processes in $\Pi \setminus P$. See for example the selected configuration illustrated in Figure 7.6a for the confusion from Figure 7.3. Then we can reach a configuration in which all registers not in S are covered by processes in P. Indeed, when executed solo starting from C, a process must eventually write to a register that is not covered in C. Thus, it must eventually write either to a register in $\mathcal{R} \setminus S$ or to the uncovered register in S. Recall that, as $|S|+|P| = n+1$, we have $|\mathcal{R} \setminus S| = |P|-1$. Thus, by concatenating solo executions of processes in P until they are poised to write to uncovered registers, we reach a configuration $C\alpha$ in which *all* registers are covered. In our example case, we obtain a setting such as the one illustrated in Figure 7.6b.

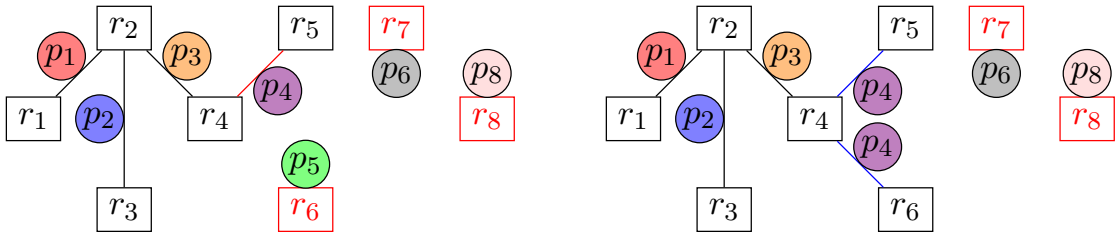
Let p be the process in P covering a register from S in $C\alpha$ (e.g., p_7 in Figure 7.6). Note that, as α is P-only, we have $\mathit{Confused}(P, S, \mathcal{D}\alpha)$. Thus:

$$\mathit{Cover}(\mathcal{R} \setminus S, P \setminus \{p\}, C\alpha) \wedge \mathit{Confused}(P, S, \mathcal{D}\alpha).$$

We now extend this set of configurations to make P confused on *two* distinct sets of registers. By Lemma 7.7, there exists $p_c \in \Pi \setminus P$ and $r \in S$ such that for any $C' \in \mathcal{D}$ we have $\mathit{Confused}(P \cup \{p_c\}, S \setminus \{r\}, \mathcal{D}')$ with $C' \in \mathcal{D}'$. Let us select $C' \in \mathcal{D}$ to be a configuration in which p_c covers $r_c \in S \setminus \{r\}$ (Since we have $\mathit{Confused}(P, S, \mathcal{D})$, $p_c \in P$ may cover two registers from S in $\mathcal{D}$ and so at most one can be r). The selection of a reduced confusion is illustrated in Figure 7.7a, where r_5 is removed from the confusion and thus p_4 is no longer used.

If p is executed solo from $C'\alpha$, it must write infinitely often to *all* registers in S to ensure that it writes to an uncovered register. Hence, in a $\{p, p_c\}$-only execution from $C'\alpha$, p_c can be hidden for arbitrary many steps as long as p_c does not write to a register outside of S. But, as the algorithm satisfies 2-obstruction-freedom, p_c *must* eventually write to a register outside of S in such an execution. Consider the $\{p, p_c\}$-only execution β from $C'\alpha$ in which p_c is hidden and such that p_c executes until it is poised to write to some register $r' \in \mathcal{R} \setminus S$. Thus, we get two configurations $C'\alpha\beta$ and $C'\alpha\beta|_{\{p\}}$, indistinguishable to all processes but p_c, in which p_c covers, respectively, $r' \in \mathcal{R} \setminus S$ and $r_c \in S$. Thus, the conditions of Lemma 7.8 hold for $\mathcal{D}'$, p_c, $\alpha\beta$ and $\alpha\beta|_{\{p\}}$ and so we obtain $\mathit{Confused}(P, (S \cup \{r'\}) \setminus \{r\}, (\mathcal{D}'\alpha\beta) \cup (\mathcal{D}'\alpha\beta|_{\{p\}}))$.

We therefore manage to obtain an alternative cover of size $|S|$ for the same set of confused processes but with a distinct set of registers. This new scenario is illustrated in Figure 7.7b, where r_5 is replaced by r_6. Note that the initial confusion is still valid after the $\{p\}$-only



(a) Selecting a reduced covering.

(b) Constructing an alternative cover.

Figure 7.7 – Addition of an alternative covering of the same size and for the same processes.

execution $\beta|_{\{p\}}$. Thus, we obtain the following result:

$$\mathit{Cover}(\mathcal{R} \setminus S, P \setminus \{p\}, C\alpha\beta) \wedge \mathit{Confused}(P, S, \mathcal{D}\alpha\beta|_{\{p\}}) \wedge$$
$$\mathit{Confused}(P, S \cup \{r'\} \setminus \{r\}, (\mathcal{D}'\alpha\beta) \cup (\mathcal{D}'\alpha\beta|_{\{p\}})).$$

Moreover, all configurations in the formula above are indistinguishable to processes in P, since $\mathcal{D}' \subseteq \mathcal{D}$, $I(\mathcal{D}, P)$, $\alpha\beta$ is $P \cup \{p_c\}$-only and p_c is hidden in it (Observation 7.2).

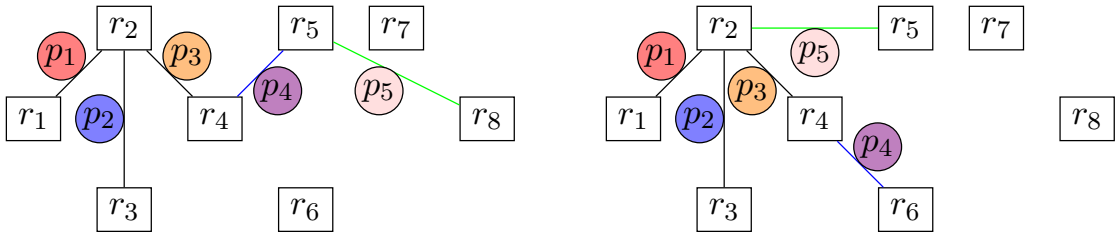
Let p' be the process from P that covers r' in $C\alpha\beta$ (e.g., p_5 in Figure 7.7). According to p or p', every proper subset of S or $S \cup \{r'\} \setminus \{r\}$ may be covered in the current configuration by $\Pi \setminus (P \cup \{p, p\})$ and all other registers covered by $P \setminus \{p, p'\}$. Thus, from $C\alpha\beta$, to complete a Write operation, p or p' must write to *all* registers in one of the sets S, $S \cup \{r'\} \setminus \{r\}$ or $\{r, r'\}$.

Consider any $\{p, p'\}$-only extension of $C\alpha\beta$. If one of $\{p, p'\}$ covers a register in $S \setminus \{r\}$, r or r', then the other process, in any solo extension, must write respectively to all registers in $\{r, r'\}$, S or $(S \cup \{r'\}) \setminus \{r\}$. In particular, since p' covers r' in $C\alpha\beta$, p running solo from $C\alpha\beta$ must eventually cover a register in $S \setminus \{r\}$ (Note that $S \setminus \{r\} \neq \emptyset$, since $|P| < n$ and $|P| + |S| = n + 1$). Then p' executes solo afterwards and hence must write to r and r'. Let us stop p' when it covers a register $r'' \neq r$ for the last time before writing to r. Let γ be the resulting execution, and $E = C\alpha\beta\gamma$ be the resulting configuration.

Let $\mathcal{E}$ and $\mathcal{E}'$ denote the sets of configurations indistinguishable from E to P defined as $\mathcal{D}\alpha\beta|_{\{p\}}\gamma$ and $(\mathcal{D}'\alpha\beta\gamma) \cup (\mathcal{D}'\alpha\beta|_{\{p\}}\gamma)$ respectively. Note that as γ is P-only, we still have $\mathit{Confused}(P, S, \mathcal{E})$ and $\mathit{Confused}(P, S \cup \{r'\} \setminus \{r\}, \mathcal{E}')$.

Now the following two cases are possible:

1. $r'' \notin S \cup \{r'\}$: In this case, we let p continue until it is poised to write on r, and then, we let the process from $P \setminus \{p, p'\}$ which covers r'' to proceed to its pending write on r''. Let δ be this P-only execution from E in which p' is hidden. As p' covers $r \in S$ in $E\delta$ and $r'' \in \mathcal{R} \setminus S$ in $E\delta|_{P \setminus \{p'\}}$, as $I(\{E\delta, E\delta|_{P \setminus \{p'\}}\}, \Pi \setminus \{p'\})$, and as $\mathit{Confused}(P, S, \mathcal{E})$, we can apply Lemma 7.8 and obtain $\mathit{Confused}(P \setminus \{p'\}, S \cup \{r''\}, (\mathcal{E}\delta) \cup (\mathcal{E}\delta|_{P \setminus \{p'\}}))$. This case where the initial confusion can be extended is illustrated in Figure 7.8a (in which $p' = p_5$, $r = r_5$ and $r'' = r_8$).
2. $r'' \in S \cup \{r'\}$, and so $r'' \in (S \cup \{r'\}) \setminus \{r\}$: Then we have the following sub-cases:
   — Some step performed by p in a solo execution from E makes p' to choose a register other than r to perform its next write in resumed solo extension of p'. Clearly, this step of p is a write. From the configuration in which p is poised to execute this "critical" write, let p' run solo until it is poised to write to r and then let p complete its pending write. Let $E\delta$ be the resulting configuration.



(a) Extension with initial confusion.

(b) Extension using alternative confusion.

Figure 7.8 – Two possible extensions of confusion size, either with initial or alternative confusions.

Now consider the execution in which p completes its "critical" write, then p' runs solo until it covers a register $r''' \neq r$. Let $E\delta'$ be the resulting configuration. Note that as the states of the memory in $E\delta$ and $E\delta'$ are identical, we have $I(\{E\delta, E\delta'\}, \Pi \setminus \{p'\})$. Note that δ and δ' are P-only executions, and that p' covers r in $E\delta$ and r''' in $E\delta'$.

(a) If $r''' \in S$, as we have $\mathit{Confused}(P, (S \cup \{r'\}) \setminus \{r\}, \mathcal{E}')$, applying Lemma 7.8, we obtain $\mathit{Confused}(P \setminus \{p'\}, (S \cup \{r'\}), (\mathcal{E}'\delta) \cup (\mathcal{E}'\delta'))$. In this case the initial confusion is extended as shown in Figure 7.8a.

(b) If $r''' \in \mathcal{R} \setminus S$, as we have $\mathit{Confused}(P, S, \mathcal{E})$, applying Lemma 7.8, we obtain $\mathit{Confused}(P \setminus \{p'\}, (S \cup \{r'''\}), (\mathcal{E}\delta) \cup (\mathcal{E}\delta'))$. In this case the alternative confusion is extended as shown in Figure 7.8b.

— Otherwise, no write of p is "critical", and we let it run from E until it covers r (recall that, as p' covers $r'' \in (S \cup \{r'\}) \setminus \{r\}$, p must eventually write to all registers in S or $\{r, r'\}$ and, thus, to r). Let then p' run until it covers r, as p, and let δ be this execution. From $E\delta$, let p' run until it becomes poised to write to a register $r''' \neq r$, and then let p perform its pending write on r. Let λ be this extension. Note that as p' is hidden in λ, we have $I(\{E\delta\lambda, E\delta\lambda|_{\{p'\}}\}, \Pi \setminus \{p\})$. Note also that $\delta\lambda$ and $\delta\lambda|_{\{p\}}$ are P-only executions such that p' covers r in $E\delta\lambda|_{\{p\}}$ and covers r''' in $E\delta\lambda$.
As before, depending on where r''' is situated, either the initial confusion or the alternative confusion may be extended (both cases are illustrated in Figure 7.8):

(a) If $r''' \in S$, as we have $\mathit{Confused}(P, (S \cup \{r'\}) \setminus \{r\}, \mathcal{E}')$, applying Lemma 7.8, we obtain $\mathit{Confused}(P \setminus \{p'\}, (S \cup \{r'\}), (\mathcal{E}'\delta\lambda) \cup (\mathcal{E}'\delta\lambda|_{\{p\}}))$.

(b) If $r''' \in \mathcal{R} \setminus S$, as we have $\mathit{Confused}(P, S, \mathcal{E})$, applying Lemma 7.8, we obtain $\mathit{Confused}(P \setminus \{p'\}, (S \cup \{r'''\}), (\mathcal{E}\delta\lambda) \cup (\mathcal{E}\delta\lambda|_{\{p\}}))$.

□

Proof conclusion. Our lower bound directly follows from the initialization case proved in Lemma 7.9 and the inductive step proof shown in Lemma 7.10:

Theorem 7.3. *Any n-process comparison-based 2-obstruction-free SWMR memory implementation requires $n + 1$ MWMR registers.*

Proof. By contradiction, suppose that an n-register algorithm exists. We show, by induction, that there is a reachable configuration in which a process is confused on *all* registers. Lemma 7.9 shows that there exists a reachable configuration in which $n - 1$ processes are confused on two

registers. We can therefore apply Lemma 7.10 and obtain a configuration with a confusion with strictly more registers. By induction, there exists then a set of configurations $\mathcal{D}$ and $p \in \Pi$ such that $\mathit{Confused}(\{p\}, \mathcal{R}, \mathcal{D})$.

Thus, any strict subset of $\mathcal{R}$ is covered by the remaining $n-1$ processes in some configuration in the (indistinguishable for p) set of configurations $\mathcal{D}$. But p may complete a Write operation if and only its write value is present in a register which is not covered (by a process not aware of its value) in any of the configurations indistinguishable to p. Therefore, in an infinite solo execution, p must write infinitely often to *all* registers. But then, any arbitrary long execution by any other process can be hidden by incorporating sufficiently many steps of p, violating 2-obstruction-freedom—a contradiction. □

7.5 Related Problems

In this section, we provide a brief look at a couple of related problems. We first show how resilience can be used to use less registers. Then, we introduce a stronger problem of SWMR allocation, necessary to solve for time optimal implementations. Lastly, we study the problem of stable storage in an anonymous system with an unbounded number of participants.

7.5.1 Resilient SWMR Memory Implementation.

Our algorithm presented in Section 7.3 works in a wait-free system in which all (but one) of the participating processes may fail by crashing. As we are going to show, if resilience assumptions are made, then less than n registers are sufficient. We consider here that at most t out of the n participants may fail by crashing. We show that in this setting, $(n-t)$-lock-freedom can be achieved using only $t+1$ registers.

Algorithm description. The Algorithm is presented in Algorithm 7.2. An operation is represented as a quadruplet $(\mathit{ValueType}, \mathit{IdType}, \mathit{IdType}, \mathbb{N})$, corresponding respectively with, the written value, the identifier of the process which issues operation, the identifier of the process which propagated the operation and an operation counter. Processes go through a loop until the following condition is satisfied: If a write operation is known to have been propagated by m processes and if it is present in a snapshot in ℓ registers, then $m + \ell$ must be strictly greater than n.

In each loop, processes first take a snapshot of the memory. Each operation quadruplet that was not observed before is added to their view, along with the same operation in which the "propagator" is replaced by the process own identifier. Then, processes check whether there is a register which does not contain their own view. If so, the smallest such register is updated with the process current view. Otherwise, processes end the loop without any modification of the memory.

Note that it is assumed that processes must perform write operations infinitely often. This assumption is not meaningless, it restricts its use when dealing short lived problems. Indeed, $(n-t)$ processes make progress, thus not necessarily all correct processes. But, if they do not stop participating, then other processes cannot be guaranteed progress afterwards.

Algorithm's proof sketch. Let us give a proof sketch of the algorithm safety first. A process terminates only when it has observed m processes "propagating" their own write

Algorithm 7.2: $(n-t)$-lock-free t-resilient SWMR memory using $t+1$ registers.

```
1  View : list of quadruplets of type (ValueType, IdType, IdType, ℕ), initially set to ∅;
2  opCounter ∈ ℕ, initially set to 0;

3  Write(v):
4  |  View = View ∪ (v, id, id, opCounter);
5  |  do
6  |  |  Snap = MEM.snapshot();
7  |  |  forall (v, id1, id2, opCounter) ∈ Snap do
8  |  |  |  View = View ∪ {(v, id1, id2, opCounter), (v, id1, id, opCounter)};
9  |  |  if ∃j ∈ {1, ..., t+1}|Snap[j] ≠ View then
10 |  |  |  writePos = min({j ∈ {1, ..., t+1}|Snap[j] ≠ View});
11 |  |  |  Update(MEM[writePos], View);
12 |  while |{j ∈ {1, ..., t+1}|(v, id, _, opCounter) ∈ Snap[j]}|+
13 |  |      |{k ∈ IdType|∃j ∈ {1, ..., t} : (v, id, k, opCounter) ∈ Snap[j]}| ≤ n;
14 |  opCounter = opCounter + 1;
15 End Write;

16 Collect():
17 |  Reads = MEM.snapshot();
18 |  V = ∅;
19 |  forall pid such that (_, pid, _, _) ∈ Reads do
20 |  |  V = V ∪ {v} with v such that (v, pid, _, max{c ∈ ℕ, (_, pid, _, c) ∈ Reads}) ∈ Reads;
21 |  Return V;
22 End Collect;
```

operation (including itself). Therefore, at most $n-m$ processes may not have observed the write value. But, the write operation value is present in strictly more than $n-m$ registers. Thus one is not covered by a process unaware of the operation. This register will therefore always contain the operation value. In a read operation, processes will thus see this write value and return it, unless a value corresponding to a latter write operation is observed and returned. Thus the set of write operations returned is always valid.

Let us now give a proof sketch of liveness properties. By assumption, there are at least $n-t$ correct processes. Let us show that, if processes proceed to infinitely many write operations, then at least $n-t$ processes terminate infinitely many such operations. Assume that this is not the case. Thus, eventually, only $m < n-t$ processes complete new operations.

First, assume that $m > 1$. Let p be a process terminating infinitely many write operations. Each time p terminates an operation, p must have seen its operation propagated by at least $n-t$ processes. Indeed, the value can be observed in at most all the $t+1$ registers. Thus one of the processes propagating p operations must be eventually blocked. But this process propagates p operations only associated to its own blocked operation. As p operations complete, then all processes must see p values in any latter read. Thus all registers written infinitely often eventually always contain q blocked write operation and the fact that it is propagated by processes propagating p values infinitely often. Thus q must eventually pass the test, a contradiction.

Now assume that $m = 0$. Thus, eventually no new write operations are completed. It implies that processes views stop being modified eventually. At this point, a process cannot learn new information. Thus all processes with the less information stop writing the registers.

This implies that, if a correct process has more information, then it cannot be over-written. Thus its information will be seen by other processes with less information, a contradiction. All correct processes thus have the same information. There are at least $n - t$ correct processes. Each knows its current write operation. Thus all correct processes know all correct processes current operations. But once a process learns of an operation, it propagates it. Thus all correct processes know that all correct processes propagated their own value. Moreover, as the same information is present in all registers, all correct processes see their own operation present in the $t + 1$ registers. Thus correct processes pass the test of the while loop and complete their current operation, a contradiction.

Remarks. It is not too difficult to see that the t-resilient algorithm can, as it is done in Algorithm 7.1, use extra registers to provide extra liveness. Indeed, this time, the k^{th} extra register is used only if at least $n - t + k - 1$ processes are observed as making progress. This way, using $t + k + 1$ registers, we can provide a $(n - t + k)$-lock-freedom SWMR memory implementation. In particular, if $2t + 1$ registers are available, then a progress can be guaranteed to all correct processes in a t-resilient system. Thus, if $t < (n - 1)/2$ we obtain an implementation providing progress to all correct processes with less registers than the number of participating processes.

When dealing with task solvability issues, using only $t + 2$ registers may be sufficient to provide outputs to all processes. Indeed, the t-resilient algorithm can be used on the first $t + 1$ registers while the last register is used only by non-terminated processes. In this setting at least $n - t$ processes make progress, including a non-terminated one which must thus eventually terminate. Note that for colorless tasks, $t + 1$ registers are sufficient since a terminated process is guaranteed to proceed to infinitely write operation, hence its output can be returned by all processes. Thus, in particular, $(t + 1)$-set consensus can be solved in a t-resilient system with only $t + 1$ registers.

7.5.2 SWMR Allocation

Now, instead of wondering if less registers can be used in a stronger system, let us consider the time complexity of SWMR memory implementation. A natural question that arises when implementing an SWMR memory is whether or not writing to multiple registers to ensure persistence is a necessary requirement. If $2n - 1$ registers are available, one could be tempted to apply a renaming algorithm and then use the register associated with the new name as a SWMR register. But applying a renaming algorithm already requires some communication. While $2n - 1$ registers are sufficient to implement an SWMR memory, nothing ensures that when a process is attributed a name j, the renaming algorithm will not use anymore the register of rank j. This leads to an interesting problem of SWMR allocation.

Definition. In the SWMR allocation problem on m MWMR registers, processes have access to a single *one-shot* operation *Allocate*(*Id*) accessed with the process unique identifier. The operation returns an integer j in $\{1, \ldots, m\}$ such that the following properties are satisfied:

- *Unicity*: No two processes obtain the same integer as output.
- *Avaibility*: If j is returned to some process at time τ, then no process can write to register j after time τ in the allocation algorithm.
- *Termination*: All correct processes must eventually terminate.

Contrary to the SWMR memory problem, this is a *one-shot* problem. Yet, it is not too difficult to see that a solution to the SWMR allocation problem with a linear number of registers implies a time optimal, wait-free, SWMR memory implementation. Indeed, processes can use the allocation algorithm and then use the allocated register as an SWMR register. Write operations thus consists in a single write to the allocated registers, collect operations can just collect the content of all registers.

SWMR allocation vs. renaming. The SWMR allocation problem can be trivially solved if $3n-1$ registers are available. Indeed, processes can use the n first registers to implement a lock-free SWMR memory. Using this emulated SWMR memory, processes can execute a renaming algorithm providing all processes with distinct names in the range $\{1,\ldots,2n-1\}$. Processes can then simply return the name obtained plus n to solve the SWMR allocation problem. Note that lock-freedom is sufficient to ensure that all correct processes terminate. Indeed, a terminated process will no longer compete to proceed to write operations. Moreover, as allocated registers and registers used for the SWMR memory implementation do not intersect, there are no possible conflicts.

Concerning lower bounds, it is easy to see that a solution to the allocation problem on M registers implies a solution to the renaming task on M names. Indeed, the renaming task on M names is identical except for the removal of the *avaibility* property. This implies that the allocation problem requires at least $2n-2$ register or $2n-1$ registers depending on the value of n. Still, a gap of n registers remains between the lower and upper bounds trivially provided by the bounds known on the renaming task.

SWMR allacation vs. splitters. A splitter [MA95] is a very simple abstraction, which partitions participating processes in several groups. Formally, a splitter can be accessed by a single operation *Split*(*Id*) which returns *right*, *down* or *stop* and satisfies the following properties:

— *Solo-stop*: If a single process participates, *stop* must be returned.
— *Stop-unicity*: At most one process can return *stop*.
— *Partition*: If two or more processes participate, not all participating processes can return the same value.
— *Termination*: All correct processes must eventually terminate.

Splitters are pertinent as one can be implemented with only two MWMR registers, one of which may even be a boolean register. A sequence of m splitters can even be implemented using $m+1$ registers, by combining the use of the second register of a splitter with the first register of a subsequent splitter.

Using a single splitter, two processes can solve the SWMR allocation problem using only three registers. The solution is presented in Algorithm 7.3. Processes first put their identifier in the first register. Then they check if the second register is already written. In this case they return 2 for the SWMR allocation problem (corresponding with *down* for the splitter). Otherwise, the process writes the second register and after reads the first register. If the first register contains the process identifier, then 3 is returned (case *stop*), otherwise 1 is returned (case *right*).

Showing that Algorithm 7.3 solves the allocation problem is not very difficult. For two processes, a splitter can provide only distinct outputs. Thus the unicity property is verified (the proof of the splitter is assumed, see [MA95]). Termination is trivial. Now for *Avaibility*,

Algorithm 7.3: Optimal splitter-based SWMR allocation for 2 processes.

```
1  Allocate(Id):
2      Write(MEM[1], Id);
3      closed = Read(MEM[2]);
4      if closed ≠ true then
5          Write(MEM[2], true);
6          winner = Read(MEM[0]);
7          if winner = Id then Return 3;
8          else Return 1;
9      else Return 2;
10 End Allocate;
```

only the first two registers may be written to. If a process obtains 1, then the other process already wrote to it as its identifier is read from it. If 2 is obtained, the process does not write it and the register has been already written, hence by the other process.

Unfortunately, splitter-based renaming algorithms use a quadratic number of splitters and provide as many distinct names. Thus even if it can be used for an optimal 2 process SWMR allocation using only three registers, it does not seem to be generalizable to more. The 2 process case just provides evidence that allocating registers used by the allocation algorithm is feasible and that the upper bound is not tight.

7.5.3 Anonymous and Adaptive Stable Storage

Now, instead of increasing assumptions on the system or increasing the problem difficulty, one can look at how much the system can be weaken while still being able to implement a stable storage.

Adaptive SWMR memory. One way to weaken further the model assumptions is to assume that no bound on the number of potential participants is known. A finite number of processes may participate, but there can be arbitrary many of them. Such a system is called the *unknown* comparison-based model. A solution in this model is necessarily adaptive, the number of used registers depends on the actual execution.

This problem was recently studied in [DGFGL13]. They manage to provide an adaptation which uses a linear number of registers, relatively to the level of participation p. They first provide a participation estimation using at most $p + 1$ registers. This estimation ensures that all processes which observe the same participation size have the same estimated participation. Then, the true participation is computed by sharing the estimated participation sets to register corresponding to their size. Hence at most p additional registers may be used. Using the known participation, processes then execute an SWMR memory implementation for the participation size such as Algorithm 7.1. But each time an operation is about to complete, processes must check the participation to verify it has not evolved. If it changed, the operation is cancelled and tried again for the new participation size.

This adaptive implementation working in unknown comparison-based systems use $3p + k$ registers when k-lock-freedom is provided. No results exists for lower bounds on the number of required registers. It can be trivially shown that p registers are not sufficient, even for lock-freedom. But stronger lower bounds are much complex to derive.

Anonymous stable storage. Another way to weaken the system model consists in removing the assumption that processes have a unique identifier. In the worst setting, processes all have the same identifier, or equivalently, have no identifiers at all. If processes do not have any identifier, the system is said to be anonymous.

In an anonymous system, the SWMR memory problem is not well defined as it assumes that write operations are associated with processes. But two processes with the same write operation may not be identifiable. Yet, the notion of stable storage can still be provided. Yanagisawa introduced in [Yan16] the problem of *weak-set* implementation. The problem is defined using two operation $addToSet(v)$ which take a value and return nothing and $getSet()$ which takes no argument and must return a set of values. Each returned values must correspond to the argument of a previously initiated *addToSet* operation. Moreover, the set must contain all values corresponding with previously terminated *addToSet* operations.

In [DGFRY18], an optimal lock-free implementation using n registers is provided. It is almost identical to the comparison-based implementation, the use of identifiers is simply removed. Providing better liveness guarantees is not much more complicated. The principle of Algorithm 7.1 can be applied. The only difference concerns the detection of active processes. Without identifiers, counting processes is not possible and counting new added values can count the same process multiple times. Thus, processes need to share all their operations history. Then new added values are counted only if they correspond to distinct histories. Note that only the longest non-conflicting histories must be counted (they conflict if one is a prefix of another). Contrary to k-lock-free implementations, the algorithm improved for resilient systems does not generalize to anonymous systems. Indeed, correct processes may attempt to add the same value and therefore cannot be counted.

Unknown and anonymous value-based implementation. Difficulties arrive if the system is assumed to be anonymous and unknown at the same time. In this setting, implementing a weak-set abstraction is much more difficult. We provide here a very costly algorithm which adapts to the range of added values.

Algorithm 7.4: Weak-set implementation for unknown and anonymous systems.

```
1  addToSet(v):
2  |  Write(MEM[rank(v)], v);
3  |  forall j ∈ {1, ..., ⌈log2(rank(v))⌉} do
4  |  |  Write(INDEX[j], true);
5  End addToSet;

6  getSet():
7  |  indexCount = 1;
8  |  while Read(INDEX[indexCount]) do indexCount+ = 1 ;
9  |  view = ∅;
10 |  forall j ∈ {1, ..., 2^indexCount} do
11 |  |  if Read(MEM[j]) ≠ ⊥ then view = view ∪ {Read(MEM[j])} ;
12 |  else Return view;
13 End getSet;
```

The solution is presented in Algorithm 7.4. The principle is as simple as it can be. A process trying to add value v simply writes v in the register indexed with the rank of v among possible values. For example, if values are natural integers, processes adding k simply write k

to register k. A written value cannot be overwritten as all writes to the register have the same value. Yet, this is not sufficient as *getSet*() operations do not know when to stop searching for values. For this, a process which wrote to the register with index k, then write *true* to the first $\lceil log(k) \rceil$ registers of another array of MWMR registers called *INDEX*. As for other registers, all write operations put the same value, thus once a write is complete, the content does not change anymore.

A *getSet*() can thus simply read the *INDEX* array until a non-initialized register is found. Let k be this register. Then all registers with an index between 0 and 2^k return the set of all values found in the registers.

Registers instability. The difficulty in anonymous systems is the possible presence of clones, i.e., multiple processes with the same input and the same execution. Clones are undetectable in an execution, until they start behaving distinctly from each other. If a bound on the number of participants is known, then a bound on the number of clones is known. In an unknown system, there can be arbitrary many clones of any process. In particular, each time a register is written, with some value v, then an arbitrary large number of clones may be stopped poised to write v to the register. This leads to the following observation:

Observation 7.3. *In an anonymous and unknown system, a value may persist in memory only if it is written to a non-written and non-covered register (not covered by processes unaware of the written value).*

This observation can be used to simply show that the number of registers used in an implementation must scale linearly with the number of distinct values added to the weak-set.

Theorem 7.4. *A weak-set implementation in an anonymous and unknown system requires at least ℓ registers to complete ℓ addToSet operation of distinct values.*

Proof. Let us show this by induction on ℓ. Initialization is trivial for $\ell = 0$. Now assume that it is verified for $\ell = k$. Assume that a process starts from this point a new *addToSet* operation of a non-previously proposed value. According to Observation 7.3, a not-previously written register must be written in order to complete the operation. As by assumption k registers are already been written, after completion at least $k + 1$ registers have been written. □

The fact that the number of used register must scale with the number of distinct added values, shows that the solution of Algorithm 7.4 is almost optimal when added values are dense, i.e., if the added value with the greatest rank has rank k then most of values with a smallest rank are added to the weak-set.

Solo-space complexity. We have shown that the number of used register must scale with the number of distinct added values. But, if added values are very sparse, a better implementation which adapts to the number of added values could still be possible. We are going to see that it is not the case by showing that a single *addToSet* operation by a single process requires an unbounded number of registers.

A classical complexity metric is the *solo*-step complexity which counts how many steps a process execution solo must perform. Here we consider the *solo*-space complexity which counts how many distinct registers must be written in order to complete an operation.

Lemma 7.11. *The solo-space complexity of an operation is not constant, it depends on the rank of the written value.*

Proof. Assume by contradiction that ℓ registers are always sufficient when a single *addToSet* operation is performed by a single process. Let α_v be a solo execution of a process p performing *addToSet*(v). Let $v_1, \ldots, v_{m_v}$ be the sequence of all registers written in α_v, ordered by the first write performed on them. The sequence forms a word of size at most ℓ on an alphabet of ℓ letters. Thus, if arbitrary many values may be added to the weak-set, there must exists two values v and v' with the same sequence, i.e., with $m_v = m_{v'}$ and $v_1, \ldots, v_{m_v} = v'_1, \ldots, v'_{m_v}$.

Let us decompose the executions in $m_v + 1$ parts, $\alpha_v = \alpha_v^1 \ldots \alpha_v^{m_v+1}$, each part stopping just before a register is about to be written for the first time. We observed that clones can be stopped covering all written registers with the same value to write to it. Thus we can add in the execution, in between each part, a block write of clones which overwrite previously written registers with the last written value in the execution. Let β_v^i be the block write of all written registers in $\alpha_v^1 \ldots \alpha_v^i$. Thus $\alpha_v^1 \beta_v^1 \ldots \beta_v^{m_v} \alpha_v^{m_v+1}$ is indistinguishable from α_v.

Now, consider the following mix of these executions with block writes by clones for v and v'. First α_v^1, then $\alpha_{v'}^1$ followed by alternating sequences of $\beta_v^i \alpha_v^{i+1}$ and $\beta_{v'}^i \alpha_{v'}^{i+1}$. In α_v^1 and $\alpha_{v'}^1$, no registers are written, then before each α_v^{i+1} and $\alpha_{v'}^{i+1}$ a block write of all previously written registers is made by clones in order to restore the memory state to the case in which only v or v' is added to the weak-set. Thus both v and v' are added successfully to the weak-set. Thus, a *getSet* operation should return both values. But the memory is in the same state as after $\alpha_{v'}$ alone, in which a *getSet* operation should return only value v' — A contradiction. □

While our algorithm is probably far from optimal, the two lower bounds shows the inherent large complexity of the problem of implementing a stable storage in an anonymous and unknown system. Improving algorithms and lower bounds appears to be an interesting issue, but already with our partial results, it seems that in practice, providing a deterministic stable storage implementation is too costly. A probabilistic implementation may be more adequate in such systems.

7.6 Concluding Remarks

This chapter shows that the optimal space complexity of SWMR implementations depends on the desired progress condition: lock-free algorithms trivially require n registers, while 2-obstruction-free ones (and, thus, also 2-lock-free ones) require $n+1$ registers. We also extend the upper bound to k-lock-freedom, for all $k = 1, \ldots, n$, by presenting a k-lock-free SWMR implementation using $n+k-1$ registers. A natural conjecture is that the algorithm is optimal, i.e., no such algorithm exists for $n+k-2$ registers for all $k = 1, \ldots, n$. Since for $k = 1$, 2 and n, k-obstruction-freedom and k-lock-freedom impose the same space complexity, it also appears natural to expect that this is also true for all $k = 1, \ldots, n$.

An interesting corollary to our results is that to implement a 2-obstruction-free SWMR memory we need strictly more registers than to implement a 1-lock-free one. But the two properties are, in general, incomparable: a 2-solo run in which only one process makes progress satisfies 1-lock-freedom, but not 2-obstruction-freedom, and a run in which 3 or more processes are correct but no progress is made satisfies 2-obstruction-freedom, but not 1-lock-freedom. The relative costs of incomparable progress properties, e.g., in the (ℓ, k)-freedom spectrum [BG15], are yet to be understood.

We also introduced related problems and showed that the stronger problem of SWMR allocation is an intriguing problem with interesting links to the classical problem of renaming. Moreover, as SWMR allocation is the key to a step optimal SWMR memory implementation, understanding the subtleties of SWMR allocation is an interesting question to study further.

We also provide an interesting first glimpse at the inherent costly space complexity of a stable storage implementation in an unknown and anonymous system. In this setting, we saw that providing large registers is not very useful as they can be more or less only used as a one-shot object, and binary registers may prove to be as convenient. The most interesting question in such systems is how efficient a weak-set implementation can be when only a small number of values from a large set are added.

Chapter 8

Conclusion and Open Questions

This thesis studied task computability of a wide variety of shared-memory models. In particular, we suggested a simple combinatorial way to characterize task computability of a model via iterations of affine tasks, pure sub-complexes of finitely many iterations of the standard chromatic subdivision.

The last part of the thesis dealt with the space complexity of a class of comparison-based algorithms. In particular, we exhibited a tradeoff between the progress conditions satisfied by a stable-storage implementation and its memory requirements. See Chapter 7 for more details about these complexity issues and Section 7.6 for related concluding remarks.

Below we list implications and open questions that can be drawn from our work on the computability issues in shared-memory models.

8.1 Distributed Simulations

Distributed simulations are crucial tools when dealing with computability issues. Indeed, we can leverage the ability of a model to simulate another to reduce task computability issues of the former model to the latter. In Chapter 3, we reviewed existing simulation techniques available for shared memory models and generalized the agreement-based simulation prototypes proposed earlier [GG09, GG11]. It provides us with a compelling simulation technique that is relatively simple to apply, as shown in its applications in Chapter 4.

Universal simulation. With the improvement of the simulation techniques and their increased range of possible applications, a natural question arises: is there a universal and optimal simulation scheme?

A first issue lies in a proper formal definition of what is a distributed simulation. Intuitively, it is not too difficult to grasp what is a distributed simulation, relying on the ability to execute steps for simulated processes and an underlying synchronization protocol. But providing a formal framework is much more intricate. In Chapter 3, we relied on an approach consisting in having simulators selecting operations to try to simulate by choosing a set of suitable candidates based on the known state of the simulated system.

Another issue relies on a definition of what is a distributed model. One way can consist in considering infinite regular languages over the full information protocol on an atomic snapshot memory. But does it indeed grasp all classical models that we may wish to consider such as models enhanced with distributed objects? We also need to define what is a distributed

problem. On this issue, recent work from Castañeda et al. [CRR15, CRR17] proposed a formal definition of a notion of *long-lived tasks* and an equivalent implementation notion for distributed objects called *interval-linearizability* that may prove itself to grasp all pertinent distributed problems.

Only with a formal definition of distributed problems and models we can give a proper formalization of what does it means for a model to simulate another one: M simulates M' if M can solve any problem given a solution to it in M'. Note that in this thesis we already weaken this condition by only considering distributed tasks. Given this notion, a first natural requirement of a universal simulation should be that any model can simulate itself. While this may appear trivial, given an abstract layer of simulation, this is already a very complex issue. Moreover, if a model M can solve any problem solvable in M', then it is only natural to require that M should be able to simulate M'.

While showing that a simulation technique is optimal may be too optimistic, with so many complicated issues to resolve, a more straightforward yet still pertinent question would be to compare simulations techniques. For example, we believe that our agreement-based simulation is strictly stronger than all other simulations presented in this thesis. But formally showing such a claim requires to construct reductions between simulation techniques and their possible applications that is interesting but simple to do.

Simulations and topological reductions. Another interesting issue with simulations concerns their topological interpretations. As discussed in this thesis, the possible executions of a model lead to sets of possible configurations of the system which can be represented as a simplicial complex. In this setting, a simulation can be seen as transforming a set of configurations into another one. Hence, a simulation can be seen as merely a simplicial map from a base model to the simulated model. Trying to understand the type of mapping which can be constructed by a simulation technique and trying to understand topological transformation tools regarding them as distributed simulations are compelling issues to investigate.

Trying to interpret the simplicial approximation and the convergence algorithms as simulations may provide interesting insights about the possible simulation techniques. In the latter case, the construction in an algorithmic fashion can already help the interpretation. The convergence algorithm is implemented in the wait-free model, and an interesting issue could be to understand what can be solved using a combination of a distributed simulation with the convergence algorithm or how the convergence algorithm can be improved with resilience-based or agreement-based synchronization capabilities.

Similarly, trying to understand what type of topological transformation can be carried out using our agreement-based simulation is an intriguing issue. Probably, the structure of the solvable affine task can shed some light about the possible topological transformations. But whether such a simulation can be used directly or if some additional layer should be used to transform simplicial complexes is yet to be determined.

Reduction to wait-free solvability. Traditional applications of distributed simulations concern the reduction of task solvability issues from one model to another. In this thesis, we focused on showing that different models can solve the same set of tasks. Hence, the reduction is direct as the simulated model is used to solve the same task in another model. A more generic approach can be used to show that the solvability of one task in a given model can be reduced to the solvability of another task in the simulated model. It allows for example to

reduce task computability issues between models corresponding to distinct system sizes.

In particular, task computability issues have been reduced to the wait-free model. For example, in [Gaf09], the solvability of a task in the n-process t-resilient model is reduced to the solvability of another task in the $n-t$-process wait-free model. Surprisingly, this reduction is made for all tasks and not just colorless tasks which may be defined independently of the system size. As reducing computability issues to a single class of models is very convenient, trying to use the improved simulation technique to reduce task solvability of other models such as adversaries or collections of set-consensus objects to a wait-free model is an interesting issue. But it is not clear whether such a reduction is even possible in general, yet it can probably be done for more adversaries than just the small class of t-resilient adversaries.

8.2 Measuring Models Relative Task Computability

In Chapter 4, we introduced the notion of an *agreement function*. Agreement functions express the ability of models to solve variations of set consensus tasks. Compared to our original proposal [KR17], the definition of an agreement function proposed in this thesis accounts for both restrictions on (1) the set of participating processes in the non-trivial executions and, additionally, (2) the set of processes which may compete for agreement. Agreement functions are then used, through our agreement-based simulation, to relate the computational power of models such as adversaries, active resilience and collections of set-consensus objects.

We believe that agreement functions can be a pertinent notion in search of a measure of the relative task computability power of shared-memory models. Unfortunately, many questions are left to answer to understand what can be shown as regards this issue.

Computability of agreement functions. A first issue that should be resolved is to understand if agreement functions are computable. The solvability of a task is undecidable in general, but we believe that the solvability of the set consensus tasks considered in the computation of agreement function is likely decidable. But while the solvability of set consensus tasks has been studied in many models, no results are known for generic model definitions. Hence, showing that agreement functions are computable and, even more, providing an algorithm providing them for any shared-memory model is an interesting problem to investigate.

Agreement functions of affine models. A probably more straightforward issue to resolve would be to compute the agreement function of affine models. We believe that this should be a much simpler issue than computing agreement functions for models defined as restrictions of sets of AS runs for example. For example, computing the agreement power under a limited participation P can be reduced to the sub-complex intersecting the face of $\mathbf{s}$ correspond ing to P. When dealing with a limited set of competitors, one can merely remove vertices corresponding to other processes and thus obtain a lower dimensional simplex.

The issue is then to show how solutions to the distinct agreement tasks can be combined in a global simulation. We think that such a simulation could be constructed, at least when the agreement tasks can be solved using a single iteration of the affine task. Given such a generic construction could then help to simplify the proof of equivalence from an affine model to a model with the same agreement function. For affine tasks defined for fair adversarial models, we showed that the operations of a complex adaptive set-consensus protocol can be simulated. Instead, we could merely have to prove that some agreement tasks are solvable.

Relations between simulations and agreement functions. The main issue with our definition of agreement functions and their implication concerning the measure of the relative task computability of shared-memory models is its correspondence with the agreement-based synchronization of our simulation technique. In this thesis, we did not resolve this issue satisfyingly, as we focused on suffix-closed models with possible accesses to infinitely many copies of some shared objects. Hence, if a task is solvable, it is solvable with external inputs provided at any time.

In general, this is not the case. For example for a model ensuring that the participating set includes at least two processes, we cannot solve $(n-1)$-set agreement with external inputs provided after the first steps of the processes. But solving operations with external inputs is stronger than what is required to run our agreement-based simulation. Indeed, while any number of set-consensus operations with inputs that are not initially known to the processes must be solved, the inputs are not external but based on the result of preceding operations. Hence, it is possible that the agreement function of a model determines its synchronization power in the simulation protocol. Determining whether this is the case or not is a crucial issue that is not answered in this thesis.

Answering this question is primordial in understanding if the agreement function measure can indeed be used to compute at least a partial order on models relative task computability power or not. Only once this first question is answered, one will be able to investigate another question: Are models with the same agreement power indeed equivalent regarding task solvability? Or, does a refined measure should be provided to reach this goal?

8.3 Affine Tasks

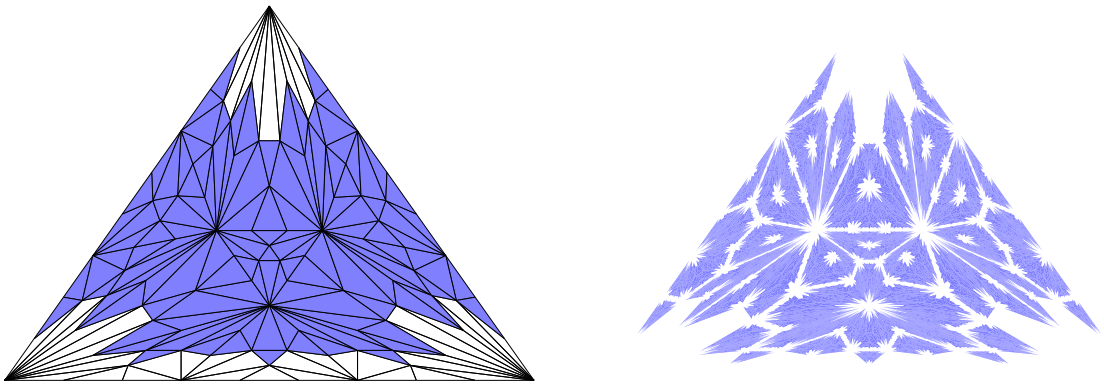
In Chapter 6, we generalized all existing topological characterizations of distributed computing models [HS99, HR12, GKM14, GHKR16, SHG16]. It applies to all tasks (not necessarily colorless) and all *fair* adversarial models (not necessarily t-resilience or k-obstruction-freedom). Additionally, we considered models beyond the scope of conventional adversaries: k-test-and-set models. Just as the wait-free characterization [HS99] implies that the *IS* task captures the wait-free model, our characterization equates considered models with a (compact) *affine task* embedded in the second degree of the standard chromatic subdivision.

We believe that our work is merely a first glimpse at affine tasks and their pertinence for the study of task computability of shared-memory models. We know that affine tasks can be used to grasp the task computability of shared-memory models beyond our examples. Moreover, we think that studying their structures may yield a better understanding of distributed computability. In particular, they can be used to determine whether the general undecidability of task solvability also applies to the question of models relative task computability.

Affine Tasks for all Shared-Memory Models?

In this thesis, we have shown that affine tasks can be used to characterize large classes of shared-memory models. But the main issue is left open, that is, whether affine tasks apply to all shared-memory models or just to well-behaved families of models.

Limits of the affine models. A first issue lies in the definition we provide for affine models. We only considered models defined by iterating a single affine task. But it can be checked



(a) One iteration of the affine task

(b) Second iteration of the affine task

Figure 8.1 – First and second iteration of the candidate affine task for the wake-up adversarial model with participation equal to $\{p_1, p_2\}$, $\{p_1, p_3\}$ or $\{p_1, p_2, p_3\}$.

relatively easily that such affine models cannot grasp all shared-memory models. One can look at the *wake-up* adversarial model, stating which set of processes may form the set of participating processes in a valid run. For example, the t-resilient wake-up adversary stipulates that the participating set must include at least $n - t$ processes.

Let us consider the 3-process wake-up adversarial model where the participating set might be equal to $\{p_1, p_2\}$, $\{p_1, p_3\}$ or $\{p_1, p_2, p_3\}$. Runs in which p_1 takes the first step and crash followed by a wait-free execution of $\{p_2, p_3\}$ are valid runs for the model. By reduction to the 2-process wait-free model, it is easy to see that consensus among $\{p_2, p_3\}$ is not solvable. But, it is easy to see that an affine task capturing the task computability of this model should have no vertices with a carrier equal to $\{p_2, p_3\}$ or $\{p_1\}$ since they are not valid participating sets. But the weakest affine task with this property, depicted in Figure 8.1a, can solve consensus among $\{p_2, p_3\}$. Indeed, if we iterate the affine task twice, as depicted in Figure 8.1b, we obtain a disconnected affine task for which there is a trivial solution to this agreement task. Therefore, no affine models can be defined to capture this model in our current definition.

Generalized Affine Models. One way to resolve this issue brought by models with short-lived properties is to replace the definition of affine models to use two affine tasks instead of one. A first affine task is used initially once, while the other is used afterward in an iterated manner. The definition is similar to constructs used in [SHG16], which shows that iterating the t-resilient affine task is equivalent to the model consisting on a single iteration of this affine task followed by iterations of the standard chromatic subdivision. It leads to a generalized definition of an affine model by a couple of affine tasks $\mathcal{R}$ and $\mathcal{Q}$ and the of IIS runs corresponding to $\mathcal{R}\mathcal{Q}^*$.

It can be observed that the composition of affine tasks of the same dimension also forms an affine task. Hence, allowing the combination of one iteration of an affine task with infinitely many iterations of another one is equivalent to allowing any finite prefix followed by a repeating pattern. It does not necessarily correspond to all infinite sequences of affine tasks, but it is close to matching any regular infinite sequence.

We believe that such a generalized affine model is likely to grasp most if not all shared-memory models. A first point of interest is to determine if such a definition indeed captures any possible infinite affine task, that is an infinite regular language over possible IIS runs.

Composition and extrapolation of affine tasks. As we observed, composing two affine tasks of the same dimension creates an affine task. Hence, a natural way to construct elaborate affine tasks is to compose a finite set of "basic" affine tasks.

Another complementary way to construct affine tasks can be done by "extrapolating" on lower dimension affine tasks. Given an m-process affine task, one can build an n-process affine task for $m > n$ by merely taking all simplices for which the face corresponding to the m particular processes belongs to the provided affine task.

We believe that using this two operations should result in a simple construction of affine tasks corresponding to many models. For example, it is relatively easy to be convinced that by extrapolating an affine task corresponding to the 2-process test-and-set model to a 3-process affine task, we obtain an affine task corresponding to a wait-free model enhanced with test-and-set objects among a single selected couple of processes. But we can create three affine tasks, each one corresponding to the wait-free model with test-and-set objects among a given possible couple of processes. Lastly, by combining these three affine tasks, we should obtain an affine task corresponding to the 3-process test-and-set model. While this example can be rather easily shown to be correct and indeed capture the 3-process test-and-set model, it does not provide a solution more straightforward than the one proposed in this thesis. Indeed, this would produce by construction an affine task corresponding to the third iteration of Chr instead of a single iteration. But this approach could be applied to more complex models.

Affine task for local models. We believe that the composition and extrapolation of affine tasks could provide a simple way to define affine tasks corresponding to at least the class of local models. Indeed, similarly to the test-and-set model, local models are defined by their ability to solve set consensus tasks among any given subset. As we have shown, such models can be reduced to the class of models defined through a collection of set consensus objects. Hence, it should be sufficient to construct affine tasks corresponding to each set consensus object in the collection (providing some level of agreement among a given set of processes). Given such affine tasks, there are good reasons to think that their composition should produce an affine task corresponding to the model defined by the collection of objects.

Moreover, to construct an affine task corresponding to the wait-free model with access to objects solving k-set consensus among a given subset Q, one could use the extrapolation method. Indeed, when we restrict the executions to steps of processes in Q, we are provided with a classical k-plain model for which we have proposed an equivalent affine model. Then, extrapolating this affine task defined on Q to the full system should indeed produce an affine task corresponding to the wait-free model enhanced with k-set consensus objects among Q. The composition of these affine tasks, providing an affine task sub-complex of many iterations of Chr, is then a very likely candidate to characterize the collection of set-consensus objects model. We hope that this methodology may even go further in providing an affine task characterizing all regular models, such as generic adversarial models.

Reduction to Wait-Free Solvability

Similarly to distributed simulations, affine tasks could provide us with a way to reduce the task computability issue in a given model to a task computability issue in the wait-free model.

Reduction for t-resilience. The characterization of t-resilience by Saraph et al. in [SHG16] can be used to reduce task solvability issues to the wait-free model. They showed that a task

is solvable in the t-resilient model if and only if it is solvable in the model where $\mathcal{R}_{\mathcal{A}_{t-res}}$ is applied once and is then followed by iterations of the standard subdivision. Hence, a task $\mathcal{T} = (\mathcal{I}, \mathcal{O}, \Delta)$ is solvable in the t-resilient model if and only if the task $\mathcal{T}' = (\mathcal{R}_{\mathcal{A}_{t-res}}(\mathcal{I}), \mathcal{O}, \Delta)$ is wait-free solvable.

We can apply this reduction to any generalized affine model $\mathcal{R}\mathcal{Q}^*$, by merely applying $\mathcal{R}$ to the input complex. It allows reducing the solvability of any task to the affine model $\mathcal{Q}^*$. Hence, for a generalized affine model under the form $\mathcal{R}\,\mathrm{Chr}^*$, this method enables to reduce task solvability issues to the wait-free model. Consequently, determining shared-memory models can be captured by such generalized affine models is an important issue to investigate.

The case of link-connected affine tasks. We believe that this could be achieved for all models which can be characterized by link-connected affine tasks. The approach from [SHG16] is likely to apply to all "stable" affine tasks, i.e., affine tasks conserving their carrier connectivities under iterations. One way to define stability could be for example to require that any variations of set consensus tasks considered in Chapter 4 solvable with iterations of the affine task should be solvable after a single iteration. An example of "unstable" affine task was given for our previous example, with the first and second iterations depicted in Figure 8.1. Moreover, we also believe that any affine task can be made stable by iterating it a computable number of times. Note that the decidability of stability could be determined as a corollary of the decidability of computing agreement functions for affine models.

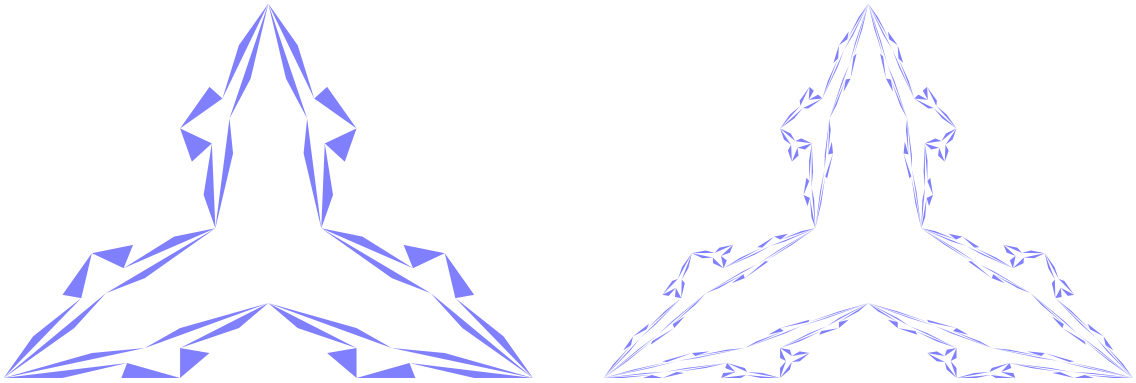
The case of link-disconnected affine tasks. We believe that this reduction to a generalized affine model with iteration of the standard chromatic subdivision applies also to models which cannot be characterized by link-connected affine tasks such as k-plain models. We believe that for these affine tasks, applying the affine task $\mathcal{R}_k$ once, followed by iterations of the standard chromatic subdivision, is indeed equivalent to $\mathcal{R}_k^*$. Intuitively, the reason is that the link-disconnectivity is somehow "orthogonal" to the mininal carrier-preserving continuous retraction of the affine task. We believe that this is also the case for fair adversarial models.

Unfortunately, this is not true for all models as it cannot be achieved for the test-and-set model. Consider the 3-process test-and-set model. Its matching affine task creates paths between each couple of corners through link-disconnected points. When iterated, the path gets longer and longer, and the number of link-disconnected vertices in the path strictly increases. The second and third iterations of the affine task are illustrated in Figure 8.2, and we can see the stable path with growing link-disconnectivity between the corners. But the image through a simplicial map of a link-connected component must be a link-connected component. Moreover, for a carrier-preserving map, the image of a path connecting two corners must also be such a path. Hence, $\mathcal{R}_{T\&S}^k\,\mathrm{Chr}^*$ cannot solve the task $\mathcal{R}_{T\&S}^{k+1}$.

While this is not sufficient to prove that task solvability cannot be reduced to wait-free solvability, it cannot be achieved through this technique. Understanding the limits of this methodology and the limits of reductions to wait-free solvability is a relevant issue that may also share insights about the possibilities and limits of distributed simulations.

Decidability of Affine Tasks

The last issue that we would like to point out is about whether the simulation or solvability of one affine model by another is a decidable question or not. It has been shown that in general, the problem of whether a task is solvable in a model or not is unfortunately



(a) Second iteration of $\mathcal{R}_{k-T\&S}$ (b) Third iteration of $\mathcal{R}_{k-T\&S}$

Figure 8.2 – Second and third iterations of the affine task characterizing the 3-proces test-and-set model.

undecidable [GK99, HR97]. But we believe that the solvability of the restricted class of affine tasks is decidable, and hence, the comparison of the relative task computability of models characterized by affine tasks.

Decomposing by connected components. The first observation that can be made is to see that the computability of affine tasks can be studied for each of their connected components. Indeed, a task is solvable in an affine model if and only if it is solvable in all affine models corresponding to a single of its connected component. Showing that it is a necessary condition is trivial as a single connected component is a subset of the global affine task. The reverse is not more complicated as after each iteration, processes know to which connected component the execution corresponds. Hence, processes can split their executions into smaller executions matching the restricted affine models. All executions must advance at the same pace in all projected models, and thus they agree on which model provide the first solution to the task.

The issue of stability. As illustrated by the example of Figures 8.1a and 8.1b, iterations of an affine task can modify its shape. We believe that computing a "stable" equivalent affine task is a decidable question. Solving this issue is likely to be required to show whether the solvability of affine tasks in affine models is a decidable problem. Providing an equivalent affine task which can solve all solvable variations of set consensus tasks among different subsets and under a varying participation in a single round can indeed be much helpful.

Link-connected affine tasks. For a single-component link-connected affine task, the decidability issue is very likely to be decidable. Indeed, we believe that the geometric realization of a single-component link-connected affine task can be continuously retracted in a carrier-preserving manner to a unique minimal sub-complex of the barycentric subdivision. Moreover, if the affine task is stable, iterations of it should retract to the same minimal sub-complex.

If this is shown to be the case, then one affine task can solve another affine task if and only if its minimal retraction is a subset of the other affine task retraction. Indeed, if it is the case, then the convergence algorithm provides a solution as the affine tasks are link-connected.

If it is not the case, then showing that no solution exists would come from the fact that a solution would imply a continuous map which can continuously retract in a carrier-preserving manner to a subset of the other affine task continuous retraction. Hence, contradicting the initial assumption.

Link-disconnected affine tasks. A first approach could consist in, as for link-connected affine tasks, to compute the minimal carrier-preserving continuous retraction to a sub-complex of the barycentric subdivision of the geometric realization of its connected components. It would correspond to considering the weaker link-connected affine model stronger of the affine task. Hence, it can be used to show that an affine task is not stronger than another, but it cannot be used to confirm the existence of a solution.

We think that some information about the type of link-disconnectivity can be added to the minimal retraction to a sub-complex of the barycentric subdivision of $\mathbf{s}$. It would have to provide information about whether the link-disconnectivity is "orthogonal" or if it presents a link-disconnectivity pattern repeating under iterations. We believe that if sufficient information is computed about the type of disconnectivity, then the existence of a solution could be computed by determining if the relative link-disconnectivity is weaker, stronger or incomparable.

While this question is somewhat complicated to tackle, we believe that it is a tractable issue and that the relative task computability power of models is a decidable question. Showing such a result would provide great insight about distributed computability of shared-memory models, especially if affine models can be used to characterize the task computability of all shared-memory models.

List of Figures

Chapter 7: Stable Storage in Comparison-Based Models

List of Algorithms

Bibliography

[AAD+93] 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, September 1993.

[ABND+90] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.

[AE14] James Aspnes and Faith Ellen. Tight bounds for adopt-commit objects. *Theory of Computing Systems*, 55(3):451–474, 2014.

[AGR+06] Yehuda Afek, Eli Gafni, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. Simultaneous consensus tasks: A tighter characterization of set-consensus. In *ICDCN*, pages 331–341, 2006.

[AM97] James H Anderson and Mark Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distributed Computing*, 11(1):1–20, 1997.

[BG93a] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100. ACM Press, May 1993.

[BG93b] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, pages 41–51, New York, NY, USA, 1993. ACM Press.

[BG97] Elizabeth Borowsky and Eli Gafni. A simple algorithmically reasoned characterization of wait-free computation (extended abstract). In *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 189–198, New York, NY, USA, 1997. ACM Press.

[BG15] Victor Bushkov and Rachid Guerraoui. Safety-liveness exclusion in distributed computing. In *34th ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 227–236, 2015.

[BGK14] Zohir Bouzid, Eli Gafni, and Petr Kuznetsov. Strong equivalence relations for iterated models. In *OPODIS*, pages 139–154, 2014.

[BGLR01] Elizabeth Borowsky, Eli Gafni, Nancy A. Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.

[BL93] James E Burns and Nancy A Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993.

[BRS15] Zohir Bouzid, Michel Raynal, and Pierre Sutra. Anonymous obstruction-free (n, k)-set agreement with n-k+1 atomic read/write registers. In *19th International Conference on Principles of Distributed Systems*, OPODIS '15, pages 18:1–18:17, 2015.

[Cha90] Soma Chaudhuri. Agreement is harder than consensus: Set consensus problems in totally asynchronous systems. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pages 311–324, Québec City, Québec, Canada, August 1990.

[Cha93] Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993.

[CR10] Armando Castañeda and Sergio Rajsbaum. New combinatorial topology bounds for renaming: the lower bound. *Distributed Computing*, 22(5-6):287–301, 2010.

[CR12] Armando Castañeda and Sergio Rajsbaum. New combinatorial topology bounds for renaming: The upper bound. *J. ACM*, 59(1):3:1–3:49, 2012.

[CRR15] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Specifying concurrent problems: beyond linearizability and up to tasks. In *International Symposium on Distributed Computing*, pages 420–435. Springer, 2015.

[CRR17] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Long-lived tasks. In *International Conference on Networked Systems*, pages 439–454. Springer, 2017.

[DFGR13] Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Sergio Rajsbaum. Black art: Obstruction-free k-set agreement with |mwmr registers| < |proccesses|. In *1st International Conference on Networked Systems*, NETYS '13, pages 28–41, 2013.

[DFKR15] Carole Delporte-Gallet, Hugues Fauconnier, Petr Kuznetsov, and Eric Ruppert. On the space complexity of set agreement. In *34th ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 271–280, 2015.

[DFRR16] Carole Delporte, Hugues Fauconnier, Sergio Rajsbaum, and Michel Raynal. t-resilient immediate snapshot is impossible. In *International Colloquium on Structural Information and Communication Complexity*, pages 177–191. Springer, 2016.

[DGFGK16] Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Petr Kuznetsov:. Set-consensus collections are decidable. In *OPODIS*, 2016.

[DGFGL13] Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Leslie Lamport. Adaptive register allocation with a linear number of registers. In *27th International Symposium on Distributed Computing*, DISC '13, pages 269–283, 2013.

[DGFGR15] Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Sergio Rajsbaum. Linear space bootstrap communication schemes. *Theoretical Computer Science*, 561:122–133, 2015.

[DGFGT11] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Andreas Tielmann. The disagreement power of an adversary. *Distributed Computing*, 24(3-4):137–147, 2011.

[DGFRY18] Carole Delporte-Gallet, Hugues Fauconnier, Sergio Rajsbaum, and Nayuta Yanagisawa. A characterization of t-resilient solvable colorless tasks in anonymous shared-memory model. In *SIROCCO*, page 15, 2018.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965.

[FFR06] Panagiota Fatourou, Faith Ellen Fich, and Eric Ruppert. Time-space tradeoffs for implementations of snapshots. In *38th ACM Symposium on Theory of Computing*, STOC '06, pages 169–178, 2006.

[FGRR14] Pierre Fraigniaud, Eli Gafni, Sergio Rajsbaum, and Matthieu Roy. Automatically adjusting concurrency to the level of synchrony. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, pages 1–15, 2014.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Gaf98] Eli Gafni. Round-by-round fault detectors (extended abstract): Unifying synchrony and asynchrony. In *Proceedings of the 17th Symposium on Principles of Distributed Computing*, 1998.

[Gaf09] Eli Gafni. The extended BG-simulation and the characterization of t-resiliency. In *STOC*, pages 85–92, 2009.

[GG09] Eli Gafni and Rachid Guerraoui. Simulating few by many: Limited concurrency = set consensus. Technical report, UCLA and EPFL, 2009.

[GG11] Eli Gafni and Rachid Guerraoui. Generalized universality. In *Proceedings of the 22nd international conference on Concurrency theory*, CONCUR'11, pages 17–27, Berlin, Heidelberg, 2011. Springer-Verlag.

[GHKR16] Eli Gafni, Yuan He, Petr Kuznetsov, and Thibault Rieutord. Read-Write Memory and k-Set Consensus as an Affine Task. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, volume 70 of *LIPIcs*, pages 6:1–6:17, 2016.

[GK99] Eli Gafni and Elias Koutsoupias. Three-processor tasks are undecidable. *SIAM J. Comput.*, 28(3):970–983, 1999.

[GK10] Eli Gafni and Petr Kuznetsov. Turning adversaries into friends: Simplified, made constructive, and extended. In *OPODIS*, pages 380–394, 2010.

[GK11] Eli Gafni and Petr Kuznetsov. Relating *L*-Resilience and Wait-Freedom via Hitting Sets. In *ICDCN*, pages 191–202, 2011.

[GKM14] Eli Gafni, Petr Kuznetsov, and Ciprian Manolescu. A generalized asynchronous computability theorem. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 222–231, 2014.

[GR10] Eli Gafni and Sergio Rajsbaum. Distributed programming with tasks. In *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, pages 205–218, 2010.

[GRT07] 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.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

[HKR14] Maurice Herlihy, Dmitry N. Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann, 2014.

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, 2003.

[HR97] Maurice Herlihy and Sergio Rajsbaum. The decidability of distributed decision tasks (extended abstract). In *STOC*, pages 589–598, 1997.

[HR12] Maurice Herlihy and Sergio Rajsbaum. Simulations and reductions for colorless tasks. In *PODC*, pages 253–260, 2012.

[HS93] Maurice Herlihy and Nir Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 111–120, May 1993.

[HS99] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(2):858–923, 1999.

[IKR17] Damien Imbs, Petr Kuznetsov, and Thibault Rieutord. Progress-Space Tradeoffs in Single-Writer Memory Implementations. In *21st International Conference on Principles of Distributed Systems (OPODIS 2017)*, volume 95 of *LIPIcs*, pages 9:1–9:17, 2017.

[IR09] Damien Imbs and Michel Raynal. Visiting gafni's reduction land: From the bg simulation to the extended bg simulation. In *SSS*, pages 369–383, 2009.

[JTT96] Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for non-blocking implementations (preliminary version). In *15th ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 257–266, 1996.

[Kön26] Dénes König. Sur les correspondances multivoques des ensembles. *Fundamenta mathematicae*, 1(8):114–134, 1926.

[Koz12] Dmitry N. Kozlov. Chromatic subdivision of a simplicial complex. *Homology, Homotopy and Applications*, 14(1):1–13, 2012.

[KR17] Petr Kuznetsov and Thibault Rieutord. Agreement functions for distributed computing models. In *Networked Systems (NETYS)*, volume 10299 of *LNCS*, pages 175–190, 2017.

[KRH17] Petr Kuznetsov, Thibault Rieutord, and Yuan He. Brief Announcement: Compact Topology of Shared-Memory Adversaries. In *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *LIPIcs*, pages 56:1–56:4, 2017.

[KRH18] Petr Kuznetsov, Thibault Rieutord, and Yuan He. An asynchronous computability theorem for fair adversaries. In *37th ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 387–396, 2018.

[Kuz12] Petr Kuznetsov. Understanding non-uniform failure models. *Bulletin of the EATCS*, 106:53–77, 2012.

[Kuz13] Petr Kuznetsov. Universal model simulation: Bg and extended bg as examples. In *Stabilization, Safety, and Security of Distributed Systems*, pages 17–31, 2013.

[LAA87] M.C. Loui and H.H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[Lam86] Leslie Lamport. On interprocess communication; part I and II. *Distributed Computing*, 1(2):77–101, 1986.

[MA95] Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, 1995.

[MRT06] Achour Mostéfaoui, Michel Raynal, and Corentin Travers. Exploring gafni's reduction land: From *mega* to wait-free adaptive (2p-[p/k])-renaming via k-set agreement. In *DISC*, pages 1–15, 2006.

[Ram30] Frank P. Ramsey. On a problem of formal logic. *Proceedings of the London Mathematical Society*, 30:264286, 1930.

[SHG16] Vikram Saraph, Maurice Herlihy, and Eli Gafni. Asynchronous computability theorems for t-resilient systems. In *DISC*, pages 428–441, 2016.

[SHG18] Vikram Saraph, Maurice Herlihy, and Eli Gafni. An algorithmic approach to the asynchronous computability theorem. *Journal of Applied and Computational Topology*, pages 1–24, 2018.

[Spa66] Edwin H. Spanier. *Algebraic topology*. McGraw-Hill Book Co., New York, 1966.

[Spe28] Emanuel Sperner. Neuer beweis für die invarianz der dimensionszahl und des gebietes. In *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, volume 6, pages 265–272, 1928.

[SZ93] Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 101–110. ACM Press, May 1993.

[SZ00] 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.

[Tau09] Gadi Taubenfeld. Contention-sensitive data structures and algorithms. In *23rd International Conference on Distributed Computing*, DISC'09, pages 157–171, 2009.

[Tau10] Gadi Taubenfeld. The computational structure of progress conditions. In *DISC*, 2010.

[Tur37] Alan M Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[Yan16] Nayuta Yanagisawa. Wait-free solvability of colorless tasks in anonymous shared-memory model. In *18th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS '06, pages 415–429, 2016.

[Zhu16] Leqi Zhu. A tight space bound for consensus. In *48th ACM Symposium on Theory of Computing*, STOC '16, pages 345–350, 2016.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Caractérisation Combinatoire de la Calculabilité Distribuée Asynchrone

Mots clés: calculabilité distribuée; topologie combinatoire; mémoire partagée asynchrone; simulations distribuées; tâches affines;

Résumé: Les systèmes informatiques modernes sont des systèmes distribués, allant de multiples processeurs sur une même puce à des systèmes internet de large échelle. Dans cette thèse nous étudions les problèmes de calculabilité et de complexité dans les systèmes distribués *asynchrones communiquant par mémoire partagée*.
Dans la première et majeure partie de cette thèse, nous étudions la capacité des modèles communiquant par mémoire partagée à résoudre des *tâches distribuées*. Notre première contribution est une technique de *simulation distribuée* utilisant la capacité d'accord du système afin de synchroniser les différents processus entre eux. Cette technique de simulation permet de comparer la capacité de différents modèles à résoudre des tâches distribuées. À l'aide de cet outil, nous montrons que pour les modèles d'adversaires en mémoire partagée, la capacité à résoudre un ensemble particulier de tâches d'accord permet de déterminer sa capacité à résoudre n'importe quelle tâche distribuée. Nous utilisons ensuite les outils issus de la topologie combinatoire afin de caractériser la calculabilité des modèles par le biais de *tâches affines:* des complexes simpliciaux extraits d'itérations finies de la sous-division colorée standard. Cette caractérisation s'applique aux modèles dits *sans-attente* avec accès à des objets de "k-test-and-set" ainsi qu'à un large ensemble de modèles d'adversaires en mémoire partagée dits *équitables*. Ces résultats généralisent et améliorent toutes les caractérisations topologiques connues de la capacité à résoudre des tâches pour les modèles communiquant par mémoire partagée.
Dans la seconde partie de la thèse, nous étudions la *complexité spatiale* de l'implémentation d'un *stockage fiable*, c.à.d., assurant qu'une valeur écrite en mémoire est persistante, dans le modèle à base de comparaison où seuls les identifiants peuvent être comparés. Nos résultats montrent l'existence d'un compromis non-trivial entre la complexité spatiale d'une implémentation et les garanties de vivacité qu'elle apporte.

Title: Combinatorial Characterization of Asynchronous Distributed Computability

Keywords: distributed computability; combinatorial topology; asynchronous shared memory; distributed simulations; affine tasks;

Abstract: Modern computing systems are distributed, ranging from single-chip multi-processors to large-scale internet systems. In this thesis, we study computability and complexity issues raising in *asynchronous crash-prone shared memory* systems.
The major part of this thesis is devoted to characterizing the power of a shared memory model to solve *distributed tasks*. Our first contribution is a refined and extended *agreement-based* simulation technique that allows us to reason about the relative task computability of shared-memory models. Using this simulation technique, we show that the task computability of a shared-memory adversarial model is grasped by its ability to solve specific agreement tasks. We then use the language of combinatorial topology to characterize the task computability of shared-memory models via *affine tasks*: sub-complexes of a finite iteration of the standard chromatic subdivision. Our characterization applies to the wait-free model enhanced with k-test-and-set objects and a to large class of *fair* adversarial models. These results generalize and improve all previously derived topological characterizations of the task computability power of shared memory models.
In the second part of the thesis, we focus on *space complexity* of implementing *stable storage*, i.e., ensuring that written values persists in memory, in the *comparison-based* model using multi-writer registers. Our results exhibit a non-trivial tradeoff between space complexity of stable-storage implementations and the progress guarantees they provide.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de lOrme aux Merisiers RD 128 / 91190 Saint-Aubin, France