VNIVERSIDAD NACIONAL
AVTNOMA DE
MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

Estrategias de transformación entre modelos de computación distribuida por rondas

TESIS

QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN CIENCIAS (COMPUTACIÓN)

PRESENTA:
VÍCTOR ADRIÁN VALLE RIVERA

DIRECTOR DE TESIS
DR. SERGIO RAJSBAUM GORODEZKY
INSTITUTO DE MATEMÁTICAS - UNAM

CODIRECTOR DE TESIS
DR. DAMIEN CLAUDE PAUL IMBS
INSTITUTO DE MATEMÁTICAS - UNAM

MÉXICO, D. F. MARZO 2014

A mi hijo Leonel, con quien deseo
compartir lo poco que sé.

Agradecimientos

A Martha por su comprensión cuando me ausentaba y su paciencia cuando la atormentaba con mis ideas sobre la tesis. A mi hijo por exigirme jugar con él cuando más presionado estaba. A mi madre por su apoyo incondicional y en general a mi familia que estuvo ahí cuando lo necesité. También a mis amigos de quienes a pesar de la distancia han sabido seguir presentes en mi vida.

Al Dr. Sergio Rajsbaum por sus brillantes clases y por aceptar ser mi tutor. También quiero agradecer especialmente al Dr. Damien Imbs por el tiempo que me brindó y sus valiosas ideas que sin duda contribuyeron en gran medida a culminar mi trabajo en el tiempo requerido. Finalmente, un agradecimiento a mis sinodales: Dr. David Flores, Dr. Carlos Velarde y Dr. Michel Raynal por su buena disposición.

Contents

List of Figures

List of Algorithms

Abstract

Due to the advent of multiprocessor computers, distributed computing models based on shared memory have received a major attention. Among them, the *Snapshot model* and its iterated counterpart have been widely studied. In particular, the *Iterated Snapshot model* (IS) lends itself more easily for combinatorial analysis and it has proved its simplicity to derive past results.

It was discovered that both the *Snapshot* and the *IS models* are equivalent when they are used to solve a certain kind of problems known as decision tasks [10]. A *decision task* is a distributed coordination problem in which a set of computing agents start with a private input value and after some communication they should individually decide a private output value in such a way that the collection of all such outputs satisfy a certain specification.

Recently, it was shown that both models are equivalent even when they are extended to work with 01-*exclusion tasks* in addition to *read/write registers* [20]. In order to prove the aforementioned result, Gafni and Rajsbaum constructed a novel simulation of the *Snapshot model* by the *IS model* and later it was shown how this simulation can be extended to prove the desired equivalence. The authors presented this algorithm informally and part of this thesis is dedicated to present it in algorithmic form and give all necessary proofs.

In addition, it is proposed for the very first time an extension derived of an old simulation presented by Borowsky and Gafni [10] that simulates the *Snapshot model* into the *Immediate Iterated Snapshot model*. Using the extension presented in this thesis it is shown that the *Snapshot model* extended to work with *x-consensus tasks* is equivalent to the *Iterated Snapshot model* extended with *x-consensus tasks*. Similarly to the work done in [20], it is presented a basic simulation and then it is constructed a more complex simulation upon it.

Introduction

The very foundation of sequential computing was primarily established by the work of Alan Turing and Alonzo Church in the 1930s. They introduced formal computing models currently known as the turing machine and the lambda calculus which turn out to be equivalent. Using these models, it is possible to show the existence of certain type of problems for which there is no solution by computational means. This theory is purely sequential, it assumes the existence of a unique computational device capable of executing instructions in a sequential fashion.

When studying distributed computing, we face a new kind of limitations different from its classical sequential counterpart. In general, we face problems related with the coordination among a set of computing devices. These devices can be for example computers located at the nodes of a network or processors in a multi core architecture. A distributed computing system models a set of computing devices that communicate with each other by some communication medium to solve a distributed problem.

Some of the major challenges to solve distributed problems are:

1. No global state knowledge. When designing sequential algorithms, it is possible to get a global coherent state of the system by simple inspection of the memory. Even when this inspection is not atomic, we assume that no data is modified while the inspection is carried out. When dealing with distributed algorithms, a computing device might be altering data while another is trying to read it.

2. Global time ordering. The set of events in an execution of a sequential system is related by a total order. It is always possible to order the events in order of their temporal occurrence. When dealing with distributed systems we just have a partial ordering of the events. There are times when for two or more events, neither of them occur before the other.

3. Non-determinism. In a distributed system, processes are inherently asynchronous, they run at different speed and can be halted or delayed without warning. This means that if you run a distributed algorithm with the same input several times, you might obtain different results. This does not mean that distributed algorithms are random, it only

means that given one algorithm and one input, you get a finite set of possible executions leading to different outputs.

For historical reasons, the first model of distributed computation consisted of a distributed system in which processes are computing nodes in a network modeled more conveniently as a digraph. The communication consists in messages passing through the edges of the digraph. This model is best known as the asynchronous message passing model and one of the first impossibility results was proved in it. In the leader election task, a set of computing devices have to decide one of them as the leader. In [16] it was shown that even when just one device might fail by halting, the election of a leader is not solvable. This result is a revelation that certain problems that are completely solvable by a turing machine are not solvable by a set of computing devices even if you assume they are as powerful as turing machines.

In recent years, the distributed computing area has received significant attention. With the advent of the Internet, and the industry of multi core machines, there is a growing excitement to take advantage of these technologies that interconnect computing devices. But few people are familiarized with the ideas and discoveries of the field and there is a lot of work to be done. In general, the distributed computing area tries to answer the question: What are the fundamental capabilities and limitations of distributed computing models? In this dissertation I try to increment this knowledge a tiny little bit further.

General Objective

Exhibit the equivalence between two well known distributed computing models namely, the *Snapshot model* and the *Iterated Snapshot model* when they are extended to work with additional communication objects, by means of simulation algorithms from one model to another.

Specific Objectives

1. Present an introduction to some basic ideas of the field required to serve as a solid basis for the objects that will be constructed later.

2. Introduce the *Snapshot model* and the *IS model*.

3. Devise an extension of both the *Snapshot model* and the *IS model* to work with additional communication objects.

4. Introduce a recent simulation from the *Snapshot model* into the *IS model* powerful enough to be extended to work with 01-*exclusion tasks* and construct upon it the algorithm that was presented by the authors.

5. Present an extension of the Borowsky and Gafni simulation that simulates the *Snapshot model* extended with x-*consensus objects* into the *IS model* extended with x-*consensus objects* in the adversary model of failures. Establishing the equivalence between both models.

Background

The notion of safe, regular and atomic registers was introduced in [32, 33]. The notion of linearizability which generalizes that of atomic register for any communication object was introduced in [27]. The shared memory model is due to the work done in [1, 2]. The adversary model of failures which generalizes the wait-free and t-resilient models was first proposed in [30]. The iterated version of the snapshot model is introduced in [10] but this is a more restrictive form of the iterated model used in this dissertation. The hierarchy of communication objects based on consensus numbers was introduced in [23]. In [18] was described for the first time the 01-exclusion family of tasks with the discovery that the relative power of this family is sandwiched between two subconsensus tasks: Set-Consensus on the top and Weak Symmetry Breaking on the bottom. Simulations in distributed computing have been effectively used to construct atomic registers from safe registers [3, 6, 11, 29, 31, 32, 33, 34, 35, 36, 37, 41, 42, 43], to construct snapshot objects from atomic registers [1, 2], to construct immediate snapshot objects from the shared memory model [8] to prove the equivalence between message passing and shared memory models when a majority of processes do not fail (otherwise a message passing system is less powerful) [4], the BG-simulation [7] which can be used to reduce impossibility results in t-resilient models to impossibility results in wait-free models. Finally, the simulations between the IS model and the Snapshot model [20, 10] are fundamental to this dissertation.

Part I

Distributed Computing Principles

Chapter 1

Modeling Distributed Systems

"Computer Science is a science of abstraction: -creating the right model for a problem and devising the appropriate mechanizable techniques to solve it".
A. Aho and J. Ullman

In its most general form, we consider a *distributed system* as a set of interconnected sequential machines, referred as *processes*, which communicate with each other to achieve a common goal. There are two paradigms to model communication within a distributed system: on the one hand, in *message passing models* processes communicate by sending and receiving messages over links in a communication network; on the other hand, in *shared memory models* processes communicate by writing and reading a memory shared by all processes. Hereafter, we consider only the shared memory approach and for the sake of this dissertation, we model the shared memory as consisting of *communication objects*. Different assumptions in timing, failures and the capabilities of communication objects give rise to different instances of our more general definition for a distributed system. It should be made clear that for our purposes it is not important the physical location of processes or the exact way they are interconnected with communication objects. Next, it is formalized the notion of processes and communication objects and then it is introduced the kind of distributed problems we deal with in this thesis.

1.1 Processes

We consider distributed systems consisting of $N = n + 1$ *processes*. A *process* is a purely (not necessarily finite) sequential automaton completely identified by its name, which is taken from

a totally ordered set of names Π. We refer to a process with name i as the i^{th} process or simply as P_i. In the most simple case, the universe of names is tight: $|\Pi| = N$, but in some situations we consider Π to be a larger name space: $|\Pi| > N$. With this in mind, we can use two different but equivalent ways of thinking about process: There may be $n+1$ processes with distinct names taken from Π, or there may be $|\Pi|$ processes where at most $n+1$ of them participate. Having made this clear, we simply assume that $\Pi = \{0, \ldots, n\}$.

Each process initially knows its name and the name space Π but it does not necessarily know the names of the participating processes. When communicating, each process might include its own name as part of the communication. This way, processes could learn (if needed) the names of their peers dynamically as the computation unfold.

As expected, the i^{th} process is associated with a set of states Q_i which includes a set of *initial states* Q_i^{in} and a set of *final states* Q_i^{fin}. We do not require Q_i^{in} nor Q_i^{fin} to be finite. Formally, a state $q_i \in Q_i$ consists of the collection of all values $q_i(a_0), \ldots, q_i(a_k)$ unequivocally assigned to the set of local variables $a_0, \ldots, a_k$ of process P_i at a given time and the state of the associated automaton. We only require that each process P_i maintains an immutable variable *name* whose value is precisely i and a variable *view* which keeps track of the entire communication history of P_i.

1.1.1 Failures

Processes can fail by *crashing*, this means that a faulty process in the system simply halts rather than communicate malicious information. Once a process has crashed, it does not recover. Since there is no bound on process relative speed nor on communication delays, failures are undetectable. No process in the system can tell for sure whether an unresponsive process has crashed or it is just slow.

To model failures, we consider three different approaches: the *wait-free case*, the *t-resilient* case and the *adversary case*. This approaches are explained below:

Wait-free Case

In the wait-free case, all but one out of N processes may crash. We require that every process completes its computation in a finite number of steps regardless of the failure or delay of any

other process. This approach is revised in much more detail in section 2.4.

t-resilient Case

In the t-resilient case, we model situations in which at most t out of N processes may fail. Any execution under this assumption guarantees that at least $N - t$ processes do not fail and then it is fair to design algorithms in which a process waits (it keeps reading the shared memory) until other $n - t$ processes communicate with it (they write the shared memory). In fact, we require that in every memory read, a process *learns* of at least $n - t$ processes besides itself. Note that the n-resilient case is in fact the wait-free case.

Adversary Case

In both the t-resilient and the wait-free case, processes can fail in an independent way. An adversary, originally proposed in [30], models situations in which failures may be correlated meaning that the failure of a process say P_i increases the probability of failure of some other process say P_j which in turn increases the probability of failure of another process and so on. Following this idea, the failure of a single process could cause the entire system crash which is not desirable. Instead, an adversary warranties the existence of one or more proper subsets of Π which we call *cores* with the following properties:

- In every run, a core has one or more processes that do not fail.
- A core C is minimal, for every $C' \subset C$, there is a run in which every process in C' fail.

Consider the set of all cores C_Π. A set S of processes is said to be a *survivor set* if $\forall\, C \in C_\Pi$, $S \cap C \neq \emptyset$ and $\forall\, P_i \in S$, $\exists\, C \in C_\Pi$ such that $P_i \in C$ and $(S - \{P_i\}) \cap C = \emptyset$. That is, a survivor set is a set of processes such that there exists an execution in which the set of all non faulty processes is exactly S. Moreover, in every execution the set of non faulty processes includes a survivor set.

Under this approach, all processes know which subsets of processes form a survivor set. Thus, we require that in every communication step, any non-faulty process *learns* at least $|S|$ new values written by a survivor set S. Note that we can model the t-resilient case by making every subset of Π of size $N - t$, a survivor set.

1.2 Communication Objects

We use the term process to refer to a device capable of performing collaborative computations in a distributed system. Since processes by themselves are purely sequential, we allow them to access one or more *communication objects* in order to perform distributed computations.

We consider a *communication object* as a data structure in memory shared by all processes in a distributed system. Each communication object is associated with a name that identifies it completely, a *type* that defines the class to which it belongs and finally a set of methods that provides the only mean to manipulate its content. Each method, consist of a sequence of operations starting with an *invocation event* and ending with a *response event*. If a method guarantees that every time it is invoked its sequence of operations is finite we say that the method is wait-free and a wait-free implementation of a communication object is one for which all its methods are wait-free. We write $(O_j\ op(args*)\ P_i)$ to denote an invocation event where O_j is an object name, *op* is a method name and $args^*$ stands for a sequence of argument values. The response for an invocation is written as $(O_j\ term(res*)\ P_i)$ where res^* is a sequence of results and *term* is used to indicate a termination condition, we usually write Ok for a normal termination but since we assume communication objects do not fail we can ignore it. A response event matches an invocation event if their object names and their process names agree. A *method call* is the interval that starts with an invocation event and ends with a matching response event. Two method calls a and b are said to be sequential w.l.o.g. if the response event of a occurs at an earlier time than the invocation event of b and are said to be concurrent otherwise.

Chapter 2

Modeling Problems

2.1 Tasks

Unlike classical sequential computing which gives a treatment and characterization of computable functions, we are interested in a particular class of problems known as *decision tasks* which are the analogous of decision problems in sequential computing. A *decision task* is a distributed problem in which each process is given a *private input value* and after some coordination among them by means of communication objects, we expect them to individually decide a *private output value*.

To formalize, let V^{in} be a domain of *input values* and V^{out} a domain of *output values*. An *assignment* is any non-empty subset A of $\Pi \times \left\{V^{in} \cup V^{out}\right\}$ such that the cardinality of A is at most $n+1$ and if $(i,u),(j,v) \in A$ then $i \neq j$. We refer to A as an *input assignment* if $A \subseteq \Pi \times V^{in}$ and as an *output assignment* if $A \subseteq \Pi \times V^{out}$. Two assignments A and A' are called *matching* if and only if for every pair $(i,u) \in A$, there exists a pair $(i,v) \in A'$ where u and v are not necessarily distinct. Thus, a *decision task* or *task* for short, is a triplet $(\mathcal{I}, \mathcal{O}, \Delta)$, where $\mathcal{I}$ is the set of input assignments, $\mathcal{O}$ is the set of output assignments, and $\Delta \subseteq \mathcal{I} \times \mathcal{O}$ is a map carrying each input assignment to a set of matching output assignments. We say that a task is *bounded* if it is defined over a finite set of input assignments $\mathcal{I}$. Hereafter we assume all tasks to be bounded.

When solving a task, any input assignment $I \in \mathcal{I}$ translates into a set of initial states of processes Q^I. If $(i,v) \in I$ then, there exists $q_i \in Q^I$ where $q_i(view) = v$. Moreover, consider the

set $\pi \subseteq \Pi$ of processes whose initial states appear in Q^I. If processes in π start to communicate, each non-faulty one should eventually reach a final state q'_i for which there exists a map function δ_i such that the set $\{(i, \delta_i(q'_i(view))) \mid i \in \pi\} \in \Delta(I)$.

2.2 Examples Of Tasks

The following tasks have been widely studied in the literature of distributed computing and each one of them is very important in its own right.

2.2.1 Consensus

The *consensus* problem is one of the most representative and fundamental problems in distributed computing. Reaching consensus among a set of processes is at the heart of many distributed algorithms. Perhaps the more generally known problem that requires consensus is *the commit problem* in distributed databases in which a set of data managers executing a transaction must agree on whether to commit the transaction or discard it. In [16], it was shown that it is impossible to reach consensus within an asynchronous distributed system even with only one faulty process.

The consensus problem can be naturally expressed in terms of our definition for a task. Each process starts with a private input value and after some communication it must decide a private output value such that the collection of all output values satisfies the following conditions:

1. **Agreement**: No two correct processes decide different values.
2. **Validity**: If some process decides v then there must exist some process whose private input value was v.
3. **Termination**: Every correct process must eventually decide some value.

Formally, for any input assignment $I \in \mathcal{I}$, $\Delta(I)$ is the set of all output assignments $O \in \mathcal{O}$ such that: $O = \{(P_i, v) \mid (P_i, x) \in I\} \wedge \exists (P_j, v) \in I$.

2.2.2 k-Set Consensus

The *k-set consensus problem* was proposed in [13] and since then it has been a widely subject of study: [5, 9, 12, 14, 21]. The *k-set consensus task* is a generalization of the consensus task. It requires that the set of all decided output values in any run is of size at most k. Observe that when $k = 1$, the task becomes an instance of the consensus problem and when $k \geq N$ the task becomes trivial since every process can simply decide its own value. This is why we require that $1 \leq k < N$. In addition, we assume that no two processes propose the same input value.

In a k-set consensus task we relax the agreement condition of the consensus task by allowing more than one value to be decided. Formally, for any input assignment $I \in \mathcal{I}$, $\Delta(I)$ consists of the set of all matching output assignments $O \in \mathcal{O}$ such that the set $K = \{v_i \mid (P_i, v_i) \in O\}$ is of size at most k.

2.2.3 01-Exclusion Tasks

In [18] was described for the first time *the* 01*-exclusion family of tasks*. A task belonging to this family is defined over a vector $\langle b_1^T, \ldots, b_n^T \rangle$ of bits. Each process solving a 01-exclusion task outputs either a one or a zero. There are two restrictions:

1. If $k < N$ processes participate solving the task, not all can decide bit b_k^T.

2. If all N processes participate solving the task, not all can decide 0 and not all can decide 1. That is, processes are partitioned into two non empty sets, the set of all processes that decide 0 and the set of all processes that decide 1.

Observe that unlike the consensus or the k-set consensus tasks, in a 01-exclusion task it is unimportant the exact values assigned to each process in any input assignment. We can simply assume that each process starts with its own name as input. That is, $V^{in} = \Pi$. What it is important is the size of the set of participating processes.

Formally, $V^{out} = \{0, 1\}$ and for each input assignment $I \in \mathcal{I}$, $\Delta(I)$ is a set of matching output assignments $O \in \mathcal{O}$ such that the following conditions hold:

1. If $|I| = N$, then the sets $\{i \mid (i, 1) \in O\}$ and $\{i \mid (i, 0) \in O\}$ are not empty.

2. On the other hand if $|I| = k < N$, then $0 \leq |\{i \mid (i, b_k) \in O\}| < k$. Alternatively, the set $\{i \mid (i, 1 - b_k) \in O\}$ is not empty.

For any member T of the 01-exclusion family, in a system with $N = n + 1$ processes, we define $D(T)$ as the dual for task T. Formally, if T is defined over $\langle b_1^T, \ldots, b_n^T \rangle$, then $D(T)$ is defined over $\langle b_1^{D(T)}, \ldots, b_n^{D(T)} \rangle$ where $b_k^{D(T)} = 1 - b_{N-k}^T$. Observe that if $b_k^{D(T)} = 1 - b_{N-k}^T$ then $b_k^{D(D(T))} = 1 - (1 - b_{N-(N-k)}^T) = b_k^T$ and then $D(D(T)) = T$.

Any member T of the 01-exclusion family of tasks satisfies the following properties:

1. Consider an $(n + 1)$-vector $\vec{o}$ of output values such that $\vec{o}[i]$ is the output of process P_i for task T. The vector $\vec{o}$ is a valid output vector for T if and only if it is also a valid output vector for $D(T)$. Observe that when all processes participate T and $D(T)$ behave the same, in both cases processes are partitioned into the set of processes deciding 1 and the set of processes deciding 0.

2. T and $D(T)$ are wait-free equivalent. This is proven in Chapter 4.

3. Let R, NR be a partition of the N processes into two non-empty sets. Let $\vec{O_R}$ be an output vector from T when processes in R participate, and $\vec{O_{NR}}$ be an output vector from $D(T)$ when processes in NR participate. The vector obtained by combining $\vec{O_R}$ and $\vec{O_{NR}}$ is a valid output vector for T. This is true because if $|R| = k$, $|NR| = N - k$ and then, not all processes in R can decide b_k^T and not all processes in NR can decide $b_{N-k}^{D(T)} = 1 - b_{N-(N-k)}^T = 1 - b_k^T$ which is the complement of b_k^T and therefore in the combination of O_R and O_{NR} not all bits are one neither zero. Thus, $O_R \cdot O_{NR}$ is a valid output vector for T when all processes participate. Consider Figure 2-1.

2.3 Protocols

Intuitively, a *protocol* is an algorithm run by processes in a distributed system in order to solve a distributed problem, in this case, a decision task. Recall that a decision task is a relation between input and output assignments. A protocol then specifies the communication steps processes should take and what information processes should communicate so that when they

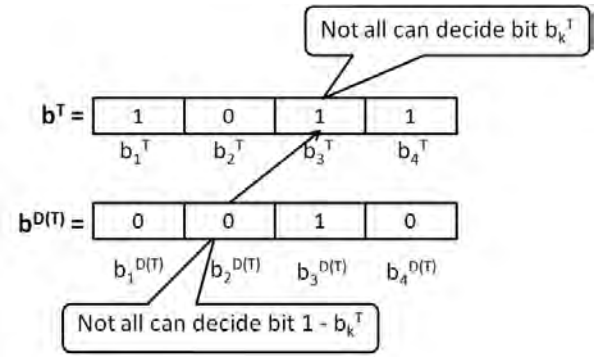


Figure 2-1: An illustration for property 3 of 01-exclusion task for 5 processes.

reach a final state, the task specification $(\mathcal{I}, \mathcal{O}, \Delta)$ is fulfilled. It is important to note that the final state of each process depends on the task it is trying to solve.

Following ideas from [24], it is natural to split a protocol into two parts: a *task-independent full-information part* in which each process repeatedly communicates its state to others, receives some of their states in response, and updates its own state to reflect what it has learned and a *task-dependent part* in which processes individually map their final state to an output value using decision map functions.

When solving a task $(\mathcal{I}, \mathcal{O}, \Delta)$, for each input assignment $I \in \mathcal{I}$ there is a set of initial states of processes Q^I such that $I = \{(q_i(name), q_i(view)) \mid q_i \in Q^I\}$. If processes whose initial state appear in Q^I start to communicate, each one of them will individually reach a final state so that the union of all such states $Q^{fin(I)}$ is a set of final states reachable from Q^I. Note that before the computation begins, for each set $Q^{I \in \mathcal{I}}$ we have a collection of possible sets of final states $\left\{Q_0^{fin(I)}, \ldots, Q_r^{fin(I)}\right\}$ and at the end of the computation, only one of such sets will represent the set of final states of processes. We denote as $\mathcal{P}$ the collection of all possible sets of final states for each set $Q^{I \in \mathcal{I}}$.

We think of a *protocol* as a quadruplet $(\mathcal{I}, \mathcal{P}, \Xi, \delta)$ that includes the set of input assignments $\mathcal{I}$, the set $\mathcal{P}$ of all final sets of states reached from Q^I for every $I \in \mathcal{I}$, a relation Ξ, that specifies the reachable sets of final states from each input assignment: If I is an input assignment in $\mathcal{I}$ then $\Xi(I)$ is a set $\left\{Q_0^{fin(I)}, \ldots, Q_m^{fin(I)}\right\} \subseteq \mathcal{P}$ of sets of final states. And finally $\delta_i \in \delta$ is a function mapping elements from $\Xi(I)$ into elements of $\mathcal{O}$. We are not concerned now with the specific algorithm that each process executes, or how it communicates with the others, only

with the sets of final states reached by the code.

To solve a task, each non faulty process P_i uses a decision map δ_i to select a decision value. The map δ_i is defined only over final states of P_i. If q_i' is one of the final states of process P_i then $\delta_i(q_i') \in V^{out}$ otherwise $\delta_i(q_i') = \perp$. Collectively, the set of functions δ_i define function δ and therefore if $Q^{fin(I)} \in \Xi(I)$ then $\delta(Q^{fin(I)}) \in \Delta(I) \subseteq \mathcal{O}$:

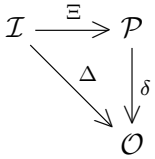
$$\delta(Q^{fin(I)}) = \left\{ \delta_i(q_i'(view)) \mid q_i' \in Q^{fin(I)} \right\}.$$

The map δ assigns to each final process state in $Q^{fin(I)}$ a pair consisting of a process name and output value.

A protocol $(\mathcal{I}, \mathcal{P}, \Xi, \delta)$ *solves* a task $(\mathcal{I}, \mathcal{O}, \Delta)$ if for every $I \in \mathcal{I}$, and every set of final states $Q^{fin(I)} \in \Xi(I)$, $\delta(Q^{fin(I)})$ is an output assignment $O \in \mathcal{O}$, and furthermore, this output assignment is allowed by the task's specification, i.e.,

$$O \in \Delta(I).$$

The following diagram semi-commutes, which in this context means that for every input configuration $I \in \mathcal{I}$, we have $\delta(\Xi(I)) \subseteq \Delta(I)$:



2.4 The Wait-Free Condition

In the model described so far, failures are undetectable. An unresponsive process may be slow or it may has crashed. In the wait-free case, all but one process may halt by crashing. This is why a process must not wait for the response of any of its peers in this model of failures. Classic techniques used to solve problems which involve waiting are completely ruled out due to this restriction. In the most extreme case, all but one process crash before taking any steps and the remaining process continues running and eventually reaching a final state without ever knowing about the rest of the processes. We refer to this extreme case as a *solo execution*. It follows

that the wait-free condition requires that when defining a task $(\mathcal{I}, \mathcal{O}, \Delta)$, $\mathcal{I}$ and $\mathcal{O}$ contain assignments of every size, between one and $n+1$. Formally, $\mathcal{I}$ and $\mathcal{O}$ must be *closed under containment*. If I is an input assignment in $\mathcal{I}$, then so is any subset $I' \subset I$. If the processes in I can start together in an execution, with input values given by I, then the processes in $I \setminus I'$ may fail before taking any steps, and the remaining processes will run as if the input assignment were I'. Furthermore, since $\Delta(I)$ contain matching assignments, then $\mathcal{O}$ will also be closed under containment and since the same argument goes for a protocol $(\mathcal{I}, \mathcal{P}, \Xi)$, it follows that $\mathcal{P}$ is also closed under containment.

Consider a process P_i that halts without ever hearing from a process P_j, this does not mean P_j must have crashed, because P_j may just be slow to start. The task specification $(\mathcal{I}, \mathcal{O}, \Delta)$ must ensure that any output value chosen by P_i remains compatible with decisions taken by late-starting processes. Formally, Δ is *monotonic*: if $I' \subseteq I$ are input assignments, then $\Delta(I') \subseteq \Delta(I)$. Operationally, the processes in I', running by themselves, may choose output values $O' \in \Delta(I')$. If the processes in $I \setminus I'$ then start to run, it must be possible for them to choose an output assignment $O \in \Delta(I)$ such that $O' \subseteq O$. Because $I \cap I'$ is a subset of both I and I', $\Delta(I \cap I') \subseteq \Delta(I)$ and $\Delta(I \cap I') \subseteq \Delta(I')$. In general, given a task $(\mathcal{I}, \mathcal{O}, \Delta)$ and for any two input assignments $I_1, I_2 \in \mathcal{I}$ if $I_1 \cap I_2$ is not empty then:

$$\Delta(I_1 \cap I_2) \subseteq \Delta(I_1) \cap \Delta(I_2). \tag{2-1}$$

Tasks that concern us here are all monotonic, in any case, for the sake of clarity here is an example of a non-monotonic task. In the *sum* task, processes output the number of processes participating in an execution. Consider three processes P_0, P_1 and P_2 and the set of all input assignments $\mathcal{I} = 2^{\Pi \times \{\perp\}} - \emptyset$. We define the task specification as $\Delta(I) = \{(P_i, |I|) \mid P_i \in \Pi\}$. To see that this task is not monotonic and it has no wait-free protocol consider an execution in which P_0 is very fast and ends its execution without ever hearing from other process, then process P_1 starts taking steps and it learns about P_0 only and then it finishes the execution. Finally P_2 starts the execution and it learns about P_0 and P_1.

In the above execution P_0 must choose output value 1 since it finishes without learning which of the others processes participated. It does not help to run longer, because no matter how long the protocol runs, P_0 might finish before the others start, forcing it to choose a value without

knowing who else is participating. Formally, this task is not monotonic, because, for example, $\Delta(\{(P_0, \perp)\})$ is not contained in $\Delta(\{(P_0, \perp), (P_1, \perp)\})$, because $\Delta(\{(P_0, \perp)\}) = \{(P_0, 1)\}$ while $\Delta(\{(P_0, \perp), (P_1, \perp)\}) = \{(P_0, 2), (P_1, 2)\}$.

We consider that a protocol wait-free solves a decision task if every non-faulty process decides an output value in a finite number of steps even if they participate in an infinite execution. On the other hand, we say that a protocol is *non-blocking* if it guarantees that at least one process will decide after a finite number of steps despite the underlying concurrency and failures of all other processes.

Chapter 3

Two Distributed Computing Models And Their Extensions

Informally, we consider a *distributed computing model* as the set of all interleavings of operations performed by a set of processes over communication objects. Here, we present two models of distributed computation widely found in the literature of the field: The *Snapshot model* and the *Iterated Snapshot (IS) model*. Next, we extend them to work with other communication objects.

As a note, the *IS model* should not to be confused with the *Iterated Immediate Snapshot (IIS) model* which is a more restricted form of the *IS model* (the *IIS model* is properly contained in the *IS model*).

3.1 Snapshot Model

3.1.1 An Overview Of The Shared Memory Model

Before introducing the *Snapshot model*, it is important to present the *Shared Memory Model* (or *SMM*), since this model represents the basis for the majority of concepts we will introduce later. Furthermore, the *Snapshot model* can be directly constructed upon the *SMM*.

Registers

A *read/write register* (or *register* for short) is a communication object that encapsulates a single value that can be observed by a *read*() method and modified by a *write*(v) method which replaces the current register value with v. A register can be characterized by the size of the values it can store, the allowed number of simultaneous readers and writers it can support and by the degree of consistency it can provide.

The size of a register is measured by the number of bits it can store, the simplest being a binary register capable of holding a single bit of information. With this in mind, we can naturally think of any value encapsulated by a register as a binary encoded integer. This should not represent an issue since we can always use integers to encode any kind of value, or simply use it to reference more abstract objects. In fact, when we use registers we assume their type consists of a set of more general object values as data vectors or data sets rather than just integers values.

The allowed number of simultaneous readers and writers is a very significant characteristic for a register object. Since we are dealing with distributed systems, we expect registers to be capable of supporting concurrent behavior. We refer to a register as a SRSW register if it can be written by a single process P_W and if can be read by a single process P_R. The acronym SRSW stands for *single reader single writer* and in the same way we use MRSW to mean *multiple reader single writer* and MRMW to mean *multiple reader multiple writer*. From now on, we consider MRSW registers only.

To specify the degree of consistency, we consider three definitions. We state them in an order of increasing consistency:

A MRSW register is safe if

- Any *read* that does not overlap a *write* returns the most recent value written by a *write* method.
- Otherwise, a *read* returns any value allowed by the type of the register.

A MRSW register is regular if

- Any *read* that does not overlap a *write* returns the most recent value written by a *write* method.

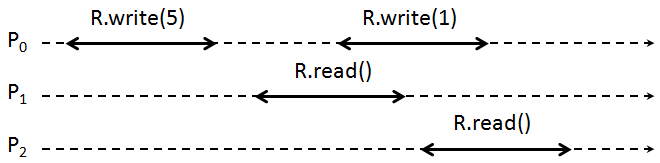


Figure 3-1: Example of a possible interleaving of method calls for a register R of integer type.

- Otherwise, a *read* that overlaps one or more writes returns any of the values written by the overlapping writes, or the last value written immediately before the *read* invocation event.

A MRSW register is atomic if it guarantees that every *read* and every *write* appear to happen at a single point in time occurring within its respective method call. We call such point a *linearization point* [27] and assume they are all distinct. Linearization points allow us to impose a total order on method calls and treat them as if they all were sequential. Therefore, in an atomic register, every linearized *read* returns the value written by the most recent linearized *write*.

Consider Figure 3-1 in which are depicted two *writes* performed by process P_0 to register R and two *reads* to register R performed by process P_1 and process P_2 respectively. Time flows from left to right and double arrows indicate the interval of a method call. If register R were safe, since both *reads* are overlapping a write, they could return any integer value allowed by the register type. If R were regular, then both *reads* could return either 1 or 5. If R were atomic, then the *read* by process P_1 could return either 1 or 5 depending on where its linearization point is located. If the returned value were 1, then no matter where you can place the linearization point for the *read* by P_2, it must return 1 since both method calls are sequential even before ordering them using linearization points.

A series of papers have shown how to construct wait-free implementations of MRSW atomic registers that can store multi-bits values in terms of the simplest registers. That is, binary SRSW safe registers [3, 6, 11, 29, 31, 32, 33, 34, 35, 36, 37, 41, 42, 43].

Communication over the Shared Memory Model

In the *SMM*, processes communicate through a vector $mem := \langle R_0, \ldots, R_n \rangle$ of MRSW atomic registers that can hold arbitrarily large values from a certain domain of values. We use $mem[j]$ when referring to register R_j in vector mem. We assume that process P_i can only write to register $mem[i]$ but it can read all registers in mem.

We treat the mem vector as a single communication object that provides a $read_i(j)$ method used by the i^{th} process to atomically get the value from register $mem[j]$ and a $write_i(v)$ method used by the i^{th} process to atomically write the value v in its associated register $mem[i]$. Note that, $read_i(j)$ and $write_i(v)$ translate into $R_j.read()$ and $R_i.write(v)$ respectively.

In all models studied here, we assume the existence of a local variable *view* maintained by each process. This *view* variable initially contains a private input value, and subsequently as the computation unfolds, it will usually change from state to state encoding a not necessarily consistent global view of the distributed system. In a *full-information protocol*, each process is repeatedly exchanging its entire communication history with the rest of its peers. We use local variable *view* as a data structure capable of holding such communication history. In [17] was defined the equivalent of our *view* variable using the *s-expression* formalism invented for nested list in the programming language Lisp. The next definition avoids any Lisp parlance.

We refer as D^* to the domain of allowable values for variable *view*. It is constructed upon a simpler domain $D = V^{in}$. If $d \in D^*$ then d is inductively defined as either a value within D or an $(n+1)$-vector $\langle d_0, \ldots, d_n \rangle$ where $d_k \in D^* \cup \{\perp\}$. Consider the next grammar:

$$\begin{aligned} D^* &\rightarrow \quad d \in D \mid \langle d_0, \ldots, d_n \rangle \\ d_j &\rightarrow \quad D^* \mid \perp \end{aligned}$$

Defined in this way, *view* becomes a tree with at most $n+1$ children in each level. A leaf is either a value within D or the special null value $\perp$.

Configurations and executions over the Shared Memory Model

A configuration is an instantaneous view consisting of the states of processes and registers at a given time. Formally, a configuration in the SMM, is an instantaneous vector $C := \langle q_0, \ldots, q_n, v_0, \ldots, v_n \rangle$ where q_i is the local state of the i^{th} process and v_j stands for the value held by register $mem[j]$. We assume that crashed processes maintain their last state before their crash. An initial configuration in the SMM is a configuration in which every process state is in an initial state and every register is set to $\perp$. If $q_i \in Q_i^{in}$, then $q_i(view) \in D$. This restriction over the value $q_i(view)$ correspond to the fact that at the beginning, a process only knows its own private input value but does not know the private input values of its peers.

Each process P_i performs its computation by alternating a *write* in which it writes its current state q_i into mem using $write_i(view)$ and then reading the entire vector mem (as many times as necessary) one register at a time using $read_i(k)$ for $k \in \Pi$. The order in which P_i reads the registers in mem is a permutation p of the indexes $\{0, \ldots, n\}$. We do not assume that processes use the same permutation to read the memory.

Each time P_i obtains a response value v'_k from a $read_i(k)$ method, it updates the k^{th} entry of its $view$ vector with the value v'_k. Note that if $q^s(view)[k] = v_k$ and $q^{s+1}(view)[k] = v'_k$, then v_k and v'_k are not necessarily distinct. Finally, when P_i has finished reading the memory, it reaches a state q_i^t in which it is allowed to perform $write_i(q_i^t(view))$. When solving a task $(\mathcal{I}, \mathcal{O}, \Delta)$, P_i continues writing and reading the memory until it reaches a final state q_i^{fin} in which a decision can be made. It then applies a decision function δ_i to its current state, with $\delta_i(q_i^{fin}(view)) \in \Delta(I)$ for some $I \in \mathcal{I}$. We do not require that P_i halts after taking a decision.

We call this kind of behavior in which a process communicates everything it knows a full information protocol. Each process writes its entire communication history along with the communication history of all processes that appear in its last $view$. For this reason, we require registers to provide a large amount of memory space. Since we are dealing with computability issues, memory space is not a problem.

We call an *event* the change of state of some single process by the application of a single method over a communication object. In the SMM events can be either a $write_i$ or a $read_i$. An execution is then an alternating (not necessarily finite) sequence of configurations and events:

$$C_0, e_0, C_1, e_1, \ldots, e_r, C_{r+1}, \ldots$$

Where C_0 is the initial configuration and for any given configuration C_i, the next configuration C_{i+1} is the result of applying the event e_i to C_i: $C_{i+1} = e_i(C_i)$. Recall that registers are atomic and we can always impose a total order on method calls obtaining such an alternating sequence. Any local computation occurring between two consecutive method calls is irrelevant in the ordering of events and we do not model it. We simply assume such local computation is part of the method call preceding it. A canonical wait-free protocol over the SMM is illustrated in Algorithm 1. Note that in the canonical form, $view_i$ contains the entire communication history for process P_i.

It is important to note that after P_i has read the entire vector mem and updated its $view$ variable accordingly, its $view$ does not necessarily encode a consistent global view of the system. Consider for example the next execution in which only three process participate:

$$\begin{aligned}
&\ldots \\
&\langle q_0^{r_1}, q_1^{s_1}, q_2^{t_1}, u, v, w\rangle, read_0(0), \\
&\langle q_0^{r_2}, q_1^{s_1}, q_2^{t_1}, u, v, w\rangle, read_0(2), \\
&\langle q_0^{r_3}, q_1^{s_1}, q_2^{t_1}, u, v, w\rangle, write_2(w') \\
&\langle q_0^{r_3}, q_1^{s_1}, q_2^{t_2}, u, v, w'\rangle, write_1(v'), \\
&\langle q_0^{r_3}, q_1^{s_2}, q_2^{t_2}, u, v', w'\rangle, read_0(1), \\
&\langle q_0^{r_4}, q_1^{s_2}, q_2^{t_2}, u, v', w'\rangle, \ldots
\end{aligned}$$

At the end of the above fragment of an execution, P_0 is in a state $q_0^{r_4}$ in which $q_0^{r_4}(view) = \langle u, v', w\rangle$. This view is inconsistent because the vector $q_0^{r_4}(view)$ of values never occurs in vector mem.

Algorithm 1 Canonical Shared Memory Protocol Solving a Task (code for P_i)

```
1: procedure solveGenericTask(v)                          ▷ v ∈ V^in
2:     view_i ← v;                                        ▷ At first, a process only knows its input
3:     while δ_i(view_i) ≠ ⊥ do
4:         write_i(view_i)
5:         for j from 0 to n do
6:             view_i[j] ← read_i(j);
7:         end for
8:     end while
9:     output δ_i(view_i);
10: end procedure
```

3.1.2 Snapshot Model

Communication over the Snapshot Model

In the SMM, it could be useful for a process to have the ability to take an instantaneous snapshot of the entire shared memory. By doing so, a process could obtain consistent views, simplifying the design and verification of distributed algorithms. The *Snapshot model* is a distributed computing model which encapsulates the *mem* vector used in the SMM inside a *snapshot object* which is a communication object capable of obtaining such consistent views. A snapshot object provides an $update_i$ method that behaves in the same way as the $write_i$ method presented before and a $snap_i$ method which returns a snapshot from vector *mem* (hence the name of snapshot object). Formally, the $snap_i$ method has no arguments and it atomically reads *mem* returning a vector $\langle v_0, \ldots, v_n \rangle$ which is an instantaneous copy of the *mem* vector occurring at a single instant between its invocation and response events.

In [1] and [2] were presented separately different algorithms that allow constructing snapshot objects from MRSW atomic registers. Therefore, the Snapshot model can be constructed upon the SMM. For completeness of this dissertation, Algorithm 2 and Algorithm 3 are two simple algorithms taken from [1] and slightly adapted to fit our notation. Both algorithms construct a snapshot object directly from the SMM. The key idea is that every *update* method uses a unique timestamp. If a process performs two consecutive sets of *reads* with identical timestamps, then either set of reads represents an atomic snapshot of the memory. In addition, the algorithm uses a helping mechanism to avoid executions in which a process can never obtain two identical sets of reads because another process is making progress and it is constantly

writing new timestamps.

Algorithm 2 An implementation of the $update_i$ method for a snapshot object (code for P_i)

```
1: procedure update_i(v)
2:     sequence ← sequence + 1;
3:     s ← snap_i();                          ▷ Before writing, take a snapshot
4:     write_i((v, sequence, s));             ▷ Write your snap to help your peers
5: end procedure
```

Algorithm 3 An implementation of the $snap_i$ method for a snapshot object (code for P_i)

```
1: procedure snap_i
2:     ∀j ∈ Π : moved[j] ← 0;
3:     while true do
4:         for j ∈ Π do a[j] ← read_i(j) end for      ▷ Sequentially read the memory
5:         for j ∈ Π do b[j] ← read_i(j) end for      ▷ Read it again
6:         if ∀j ∈ Π: a[j].seq = b[j].seq then
7:             return ⟨b[0].data, ..., b[n].data⟩;   ▷ There was no changes during the reads
8:         else
9:             for j ∈ Π do
10:                if a[j].seq ≠ b[j].seq then
11:                    if moved[j] = 1 then
12:                        return b[j].view;          ▷ Borrow the snapshot taken by P_j
13:                    else
14:                        moved[j] ← moved[j] + 1;
15:                    end if
16:                end if
17:                moved[j] ← 0;
18:            end for
19:        end if
20:    end while
21: end procedure
```

In Algorithms 2 and 3, a register value val different from $\perp$ consists of a triplet $(d, sequence, \vec{d})$ where $val.data = d$, $val.seq = sequence$ and $val.\vec{data} = \vec{d}$. Roughly speaking, algorithm 2 works as follows: before updating, a process P_i takes a snapshot of the memory using Algorithm 3 and stores it in its local vector s (this is part of the helping mechanism). Once P_i has its snapshot, it is able to write its value v. P_i increments by one a variable $sequence$ (whose value is initially 0) and it then writes the triplet $(v, sequence, s)$. This way, if $read_j(i).seq = k$, then $read_j(i)$ is the k^{th} value written by P_i.

Algorithm 3 works as follows: a process P_i taking a snapshot first reads the memory one

register at a time and it stores this set of reads in its local vector a. It repeats this procedure in its local vector b (lines 4 and 5). If the sequence numbers of both vectors agree (line 6), between both sets of reads there was no updates, and it is safe to deliver b as the response of the *snap* method (line 7). Otherwise, at least one other process performed an *update*. Each j such that $a[j].seq \neq b[j].seq$ (line 10) is marked in $moved[j]$. If this is the second time P_i observes that P_j completed an *update* during its method call, it is safe to return the snapshot taken by P_j as if it were taken by P_i since the *update* performed by P_j is completely contained within the method call of P_i (line 12).

This was just an outline of the ideas of the proof. The complete proof can be found in [1].

Configurations and executions in the Snapshot Model

Just as in the SMM, an execution in the Snapshot model consists of an alternating (not necessarily finite) sequence of configurations and events starting from an initial configuration C_0:

$$C_0, e_0, C_1, e_1, \ldots, e_r, C_{r+1},$$

An event is either an $update_i$ or a $snap_i$ operation. Algorithm 4 represents a canonical form for writing wait-free snapshot protocols. In this canonical form, each process P_i proceeds its computation by alternating one $update_i$ and one $snap_i$ operations:

$$update_i^1, snap_i^1, \ldots, update_i^k, snap_i^k$$

In the adversary model of failures, between two consecutive *updates* performed by a process, there is a sequence of one or more *snap* operations. A process takes as many snapshots as necessary in order to read new values written by a survivor set. In this case, only the last snapshot is considered to be part of the execution. Using such alternating sequence and the total order $\underset{ot}{\rightarrow}$ arising between all events, we can define some properties satisfied by all snapshots taken in a snapshot protocol:

1. $\forall\ update_i(v)^k, k > 1$:

   (a) $snap_i()^{k-1} \underset{ot}{\rightarrow} update_i(v)^k \wedge update_i(v)^k \underset{ot}{\rightarrow} snap_i()^k$

2. $\forall\ snap_i^k, j \in [n] : snap_i^k[j] = \perp \vee \exists\ update_j(v)^{k'}$ such that:

   (a) $snap_i^k[j] = v$.

   (b) $update_j(v)^{k'} \underset{ot}{\rightarrow} snap_i^k$.

We define a *one shot use* of a snapshot object as an execution in which each process asynchronously executes just one *update* followed by just one *snap* (recall that in the adversary case only the last *snap* is considered). Let S_1 and S_2 be two snapshots. We say $S_1 \subseteq S_2$ if for every k, $S_1[k] = S_2[k]$ or $S_1[k] = \perp$. In a one shot execution, snapshots are ordered by containment. For any two snapshots S_1 and S_2 it is the case that $S_1 \subseteq S_2$ or $S_2 \subseteq S_1$.

Algorithm 4 Canonical Snapshot Protocol Solving a Task (code for P_i)

```
1: procedure solveGenericTask(v)
2:     view_i ← v;
3:     while δ_i(view_i) ≠ ⊥ do
4:         update_i(view_i);
5:         view_i ← snap_i();
6:     end while
7:     output δ_i(view_i);
8: end procedure
```

3.2 The Iterated Snapshot Model

Iterated Models

For any distributed computing model M using communication objects of class X for which there exists the notion of one-shot use, it is possible to define an iterated model IM. Processes running an IM, go through a sequence of layers (or rounds) one at a time in the same order. The main characteristic is that any communication object used in a given round is never accessed in some other round. When a process reaches a layer r, it asynchronously applies a one-shot use of object O_r and then it uses the output obtained as the input for object O_{r+1} in round $r + 1$. The resulting iterated model is related to the notion of a *communication-closed layer* [15]. The intuition behind this concept is that there is a synchronization barrier at the end of each layer preventing processes to access the next layer until all of them had executed (asynchronously) the previous one. Iterated models have been widely used for instance [21, 25, 26, 38, 39].

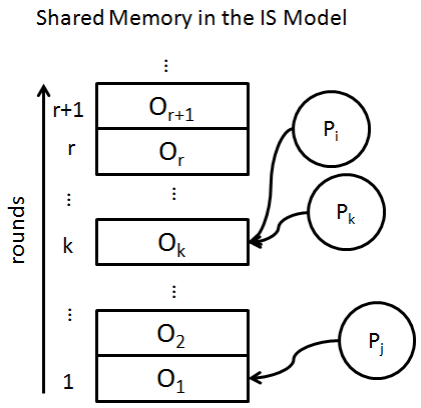


Figure 3-2: Processes accessing the memory in the IS model.

Communication over the Iterated Snapshot Model

The *Iterated snapshot model* (*IS*) is a distributed computing model in which processes communicate through a sequence (not necessarily finite) of snapshot objects. We write $update_i(r, v)$ to mean an $update_i(v)$ over the snapshot object O_r, likewise, we write $snap_i(r)$ to mean an $snap_i()$ over the snapshot object O_r.

In the *IS model*, processes access the snapshot objects in a sequence of rounds. In each round r, processes asynchronously perform just one $update(r, view)_i$ followed by one $snap_i(r)$ (recall that in the adversary case only the last *snap* is considered). When a final state is reached a decision is made as a function of the state. Due to the asynchrony described before, we do not require that any process in round r waits for the rest of processes before moving forward to the next round. Each process advances at its own pace as depicted in Figure 3-2.

Configurations and executions over the Iterated Snapshot Model

We will say that an execution $C_0, e_0, C_1, e_1, \ldots, e_k, C_{k+1}$ with $1 < k \leq 2n + 2$ is a *layer* if each non faulty process P_i participates with just one $update_i$ and just one $snap_i$ events (in that order) in the sequence of events $e_0, \ldots, e_k$. An execution of the IS model is then described by a layered execution which consists of several concatenated layers. A canonical wait-free protocol over the IS model is illustrated in Algorithm 5.

Algorithm 5 Canonical IS Protocol Solving a Task (code for P_i)

```
1: procedure solveGenericTask(v)
2:     r ← 0;
3:     view_i ← v;
4:     while δ_i(view_i) ≠ ⊥ do
5:         r ← r + 1;
6:         update_i(r, view_i);
7:         view_i ← snap_i(r);
8:     end while
9:     output δ_i(view_i);
10: end procedure
```

3.3 Extending The Models

As we have seen, snapshot objects can be constructed upon read/write registers and therefore we cannot use them to solve problems not solvable using only registers. In this section we extend the Snapshot and the IS models to use additional communication objects.

A communication object is precisely intended to allow communication between a set of processes. A snapshot object for instance, provides access to all processes. In the more general case, a communication object can be accessed by $k \leq |\Pi|$ processes. We then provide an infinite number of communication objects for every subset of Π of size k:

$$O_1^{\pi_1}, \ldots, O_1^{\pi_{\binom{N}{k}}}, O_2^{\pi_1}, \ldots, O_2^{\pi_{\binom{N}{k}}}, \ldots$$

with the following considerations:

1. All objects belong to the same class of communication objects.
2. $\pi_i \in [\Pi]^k$ where $[\Pi]^k$ stands for all subsets of Π of size k.
3. Object $O_r^{\pi_i}$ is intended to be accessed only by processes belonging to π_i.

A process P_i running a protocol of an extended model, will access the previously described communication objects depending on its knowledge about its peers. To model this behavior, we extend our protocol definition of Section 2.3 by adding a new function ϕ composed of a collection of functions ϕ_i. The intent of function ϕ_i is to indicate to process P_i which is the next communication object to invoke. Formally, function ϕ_i takes as argument the *view* of

process P_i and returns a special null symbol $\top$ if the local algorithm of P_i indicates that no additional communication object should be invoked given the local state of P_i, otherwise it returns a pair $\langle O, \lambda \rangle$ where O is the identity of the next task to invoke and λ is a list of parameters to invoke O. If the requested object does not accept parameters, then λ is an empty list.

We also change our data structure *view* to include this extra information. Recall that the domain for *view* is D^* (see Subsection 3.1.1). Consider the domain $\mathbb{I}$ which consists of the symbol $\top$ and the set of all triplets of the form $\langle O, \lambda, \omega \rangle$ where O is the identity of a communication object, λ is a list of parameters and ω is in the domain of output values of the object with identity O. In this extension, the domain of allowable values for D^* is inductively defined using the next grammar:

$$\begin{aligned} D^* \rightarrow &\quad (D, v \in \mathbb{I}) \mid D \\ D \rightarrow &\quad \langle d_0, \ldots, d_n \rangle \mid (u \in V^{in}, v \in \mathbb{I}) \\ d_j \rightarrow &\quad D^* \mid \perp \end{aligned}$$

If *view* is of the form $\langle d_0, \ldots, d_n \rangle$ and d_j is of the form (u, v), we use $view[j].data$ to access u and $view[j].call$ to access v.

An extended protocol is then a quintuplet $(\mathcal{I}, \mathcal{P}, \Xi, \delta, \phi)$. In general, the inclusion of ϕ into the definition of a protocol P, causes that P can not be longer split into a task independent full information part and a decision part because the order in which P_i chooses to access communication objects depends on ϕ. We can still split the decision part from the communication one, the communication part is just not independent anymore.

In addition to $update_i$ and $snap_i$ operations, we introduce a generic operation $invoke_i(\langle O, \lambda \rangle)$ which invokes the only method of the communication object O with the list of parameters λ returning the corresponding response ω. We require that every invocation performed by a process P_i be encoded in the corresponding triplet $\langle O, \lambda, \omega \rangle$ and added into the next *update*. If there was no invocation before an *update* operation, then P_i simply uses the symbol $\top$ in its *update*.

3.3.1 Snapshot Model Extended With Tasks

In the extension of the Snapshot model, each process P_i that solves a task, performs an *update*, it may invoke an additional communication object using the corresponding *invoke* method and then it takes a snapshot using *snap*. In this model, P_i is allowed to access (additional) communication objects in any order. The only restriction is that P_i must not invoke the same communication object more than once. Algorithm 6 is the canonical form we just introduced of an extended Snapshot protocol.

Algorithm 6 Extended Canonical Snapshot Protocol Solving a Task (code for P_i)

```
1: procedure solveGenericTask(input)
2:     view_i ← (input, ⊤);
3:     call ← ⊤;
4:     while δ_i(view_i) ≠ ⊥ do
5:         update_i(view_i);
6:         call ← φ_i(view_i);
7:         if call is of the form ⟨O, λ⟩ then
8:             call ← ⟨O, λ, invoke_i(call)⟩;
9:         end if
10:        view ← (snap_i(), call);
11:    end while
12:    output δ_i(view_i);
13: end procedure
```

3.3.2 IS Model Extended With Tasks

Unlike the extension of the Snapshot Model, we impose an additional structure in the order in which communication objects should be invoked. Following the paradigm of iterated models, we associate each round with one or more additional communication objects. There are two possible approaches when the number of process that may access this objects is $k < N$:

1. **One Round One Object**: In this approach, in round r only the subset of processes $\pi_{\mu(r)}$ where $\mu(r) = ((r + \binom{N}{k} - 1) \, mod \, \binom{N}{k}) + 1$, is allowed to access object $O_r^{\pi_{\mu(r)}}$ where *mod* stands for the modulus. This is a typical round robin approach.

2. **One Round Multiple Objects**: In this approach we associate round r with the sequence of objects $O_r^{\pi_1}, \ldots, O_r^{\pi_{\binom{N}{k}}}$.

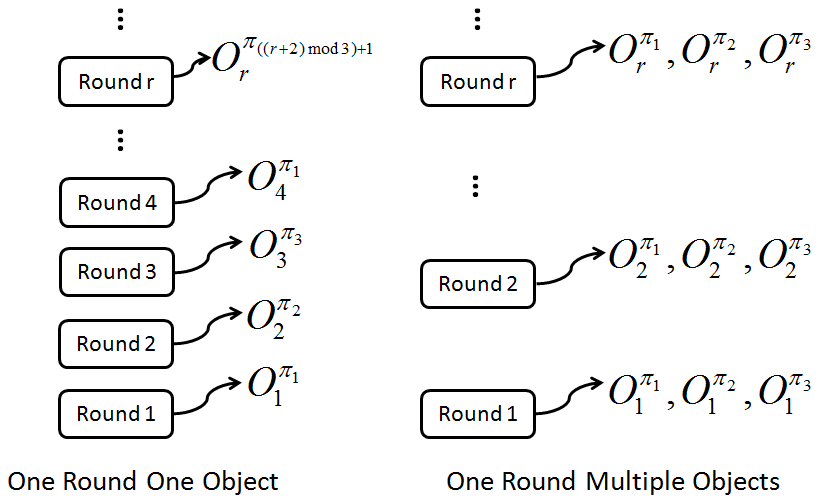


Figure 3-3: Ilustration of One Round One Object and One Round Multiple Objects for 3 processes and objects that allow communication for 2 processes only.

Both approaches are depicted in Figure 3-3. For convenience of the algorithms presented in chapter 5 we will use the One Round One Object approach. Algorithm 7 is the canonical form of an extended IS protocol using this approach.

Algorithm 7 Extended Canonical IS Protocol Solving a Task (code for P_i)

```
1:  procedure solveGenericTask(input)
2:      r ← 0;
3:      view_i ← (input, ⊤);
4:      call ← ⊤;
5:      while δ_i(view_i) ≠ ⊥ do
6:          r ← r + 1;
7:          update_i(r, view_i);
8:          call ← φ_i(view);
9:          if call = ⟨O, λ⟩ then
10:             call ← ⟨O, λ, invoke_i(call)⟩;
11:         end if
12:         view_i ← (snap_i(r), call);
13:     end while
14:     output δ_i(view_i);
15: end procedure
```

If you look closely at Algorithm 6 and Algorithm 7 you will note that despite the differences underlying the models in which they were designed, a process running either of both canonical forms will execute the operations *update*, *invoke* and *snap* in blocks as depicted in Figure 3-4.

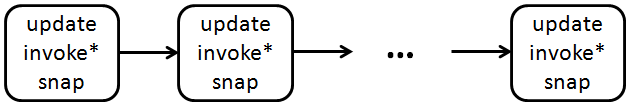


Figure 3-4: Ilustration of how you can group the operations produced by a single process running a protocol in canonical form.

A natural question that may arise is whether this is the most appropriate order for operations. Why not use *invoke*, *update* and *snap* instead? Let us suppose that someone claims that he came up with an extended IS algorithm that uses the order *invoke*, *update* and *snap* (IUS). Can we rewrite his algorithm to fit our canonical version? This is not a problem at all, we can simply add a first *update* that will be ignored, perform the *invoke*, take a snapshot that will be ignored, then in the next round perform the missing *update*, and finally take the missing snapshot. The converse is not necessarily true. Not all UIS (*update*, *invoke* and *snap*) protocols can be rewritten in the IUS form. Consider for instance, a one shot protocol $\mathcal{P}$ in our model UIS. Processes running $\mathcal{P}$, propose a value in the *update* operation, then they perform an *invoke* followed by a *snap* and finally they decide a value. Let P be a predicate over the value obtained as a response for the *invoke* operation. Processes that obtain a value v such that $P(v)$ is *true* decide v. The rest, decide the value proposed by one of their peers. We assume that the value obtained will satisfy P only if a process perform the invocation alone. Observe that in the IUS form, a process P_i might obtain v such that $P(v)$ is $false$ but if P_i is fast enough, it will perform the *update* and the *snap* operations alone. In this case, P_i cannot decide a value because it does not have any information about the proposed values except its own. It does not help to add an *update* and *snap* in a previous round and it does not help to run longer. This is just an informal reasoning with the intent of justifying the order of the operations in the canonical form presented.

Chapter 4

About The Power Of Tasks

Consider two different communication objects X and Y. An interesting question that naturally arises is whether we can wait-free implement object X using one or more copies of object Y. If so, for any wait-free distributed algorithm that solves task T using one or more copies of object X, we immediately know that there exists a distributed wait-free algorithm that solves T using Y objects. It is important to observe that a protocol $\mathcal{P}$ solving a decision task can be seen as a communication object. Any process running $\mathcal{P}$ starts with a private input value v which can be considered as the parameter for $\mathcal{P}$ and it obtains a private output value which can be seen as the response of the invocation of $\mathcal{P}$ with parameter v. The converse is not true, not any communication object can be seen as a protocol solving a task. Having made this clear, in this section we use the word task to mean a communication object solving a task. For instance, if X and Y are tasks we could rephrase our original consideration and say: An interesting question is whether we can wait-free solve task X using one or more copies of task Y.

If there is a wait-free distributed algorithm solving task X using task Y we say that Y is at least as powerful as X denoted as $Y \succeq X$. If the converse is also true, we then say that both task are equivalents under wait-free solvability.

In [23] was proposed a hierarchy of communication objects (tasks) such that no object at one level can wait-free implement any object at higher levels using the concept of *consensus number*. A communication object has consensus number k if k is the largest integer such that there exists a wait-free algorithm that solves the consensus task for a system of k processes (or infinite if there is no such integer). An object with consensus number $k+1$ can trivially

Consensus number	Communication Objects
1	read/write registers
2	test & set, swap, fetch & add, queue, stack
$\vdots$	$\vdots$
$2n-2$	n-register assigment
$\vdots$	$\vdots$
∞	memory-to-memory move and swap, augmented queue, compare & swap, fetch & cons, sticky byte

Table 4-1: Hierarchy based on consensus numbers

solve consensus for k of fewer processes but it was also shown that an object with consensus number k cannot be used to construct an object that solves consensus for strictly more than k processes. The hierarchy is shown in Table 4.

As you can see in the hierarchy, read/write registers can only solve consensus for a single process. There is however a special class of tasks that cannot solve consensus for two processes and yet they are more powerful that read/write registers. Any task belonging to this class is known as a *subconsensus task*. Examples of subconsensus tasks are the *renaming*, *weak symmetry breaking*, 01-*exclusion* (for $N > 2$) and *set-agreement*. How do we compare all these subconsensus tasks? We still do not have a definitive answer but some results have been presented, for instance [12, 14, 18, 21, 40].

Algorithm 8 is a snapshot protocol extended to use 01-exclusion tasks that wait-free solves the n-set agreement task for $n+1$ processes using a particular instance of a 01-exclusion family of tasks in which $\forall j : b_j^T = 0$, proving that the latter is at least as powerful as the former. This should not be interpreted to mean that the whole 01-exclusion family is at least as powerful as k-set consensus, in fact we know that: weak symmetry breaking family $\prec$ 01-exclusion family $\preceq$ k-set agreement family [18]. For simplicity, the algorithm is not in the canonical form of Algorithm 6.

Lemma 4.0.1. *Algorithm 8 wait-free solves n-set agreement for $n+1$ processes.*

Proof. Observe that the set of processes that obtain 1 in line 4 is at most n by the specification of a $0-1$ task whose all bits are 0. Recall that when $k < n+1$ processes participate, not all can decide b_k^T. Since $b_k^T = 0$, then at most $k < N$ processes decide 1, if all N processes participate

not all should decide 1. Therefore, if processes which obtain 1 in line 4 decide their own value, the set of decided values is at most n. Consider then the first process P_i which obtained 0 in line 4, this process is excluding its own value from the set of decided values and no process will choose its value by line 10 and the task specification is satisfied. The only problem that remains is that P_i can decide a value at all. If P_i obtained 0 in line 4 then there is at least another process P_j participating with P_i. Process P_j has to write its input value before invoking the task, and this input value must appear in the snapshot of P_i, thus P_i is able to use it as its decided value. □

Algorithm 8 Wait-Free Snapshot Protocol Solving k-set agreement (code for p_i)

Require: T is a shared 01-exclusion task shuch that $b_k^T = 0$ for all k.

```
1: procedure k-SET(input)
2:     inv ← ⊥;
3:     update_i((input, inv));
4:     inv ← T.invoke();
5:     update_i((input, inv));
6:     view ← snap_i();
7:     if inv ≠ 0 then
8:         output input;
9:     else
10:        output v for any k such that view[k] = (v, inv) with inv ≠ 0;
11:    end if
12: end procedure
```

Finally let us prove the following lemma that we left open in subsection 2.2.3.

Lemma 4.0.2. *For any* 01*-exclusion task* T*,* T *and* $D(T)$ *are wait-free equivalent.*

Proof. Since $D(D(T)) = T$, it is sufficient to prove that we can use T to wait-free solve $D(T)$. Consider Algorithm 9. Any non-faulty process running this algorithm eventually decides since no instruction requires waiting. To see that this algorithm is correct observe that if the number of non faulty process solving $D(T)$ is k with $0 < k < N = n + 1$, then not all should decide $1 - b_{n-k}^T$, this requirement is fulfilled since at least one process outputs b_{n-k}^T see line 6 of Algorithm 9. On the other hand if all N processes execute line 8 then by property 1 of the 01-exclusion family of tasks, the output vector obtained is a valid one for $D(T)$. Suppose for the sake of contradiction that r, $k > r > 0$ processes execute line 8, for the definition of T at

least one will output $1 - b_r^T$, consider the $(k - r)$ processes that did not execute line 8, at least one of them observe in its view $n - r$ participants and output $b^T_{n-(n-r)}$, that is b_r^T. Therefore at least one outputs $1 - b_r^T$ and at least one outputs b_r^T and the task specification is fulfilled.

□

Algorithm 9 Wait-Free Snapshot Protocol Solving $D(T)$ using a 01-exclusion task T (code for p_i)

Require: T is a shared 01-exclusion task such that $D(D(T)) = T$.

```
1:  procedure DUAL-01-EXCLUSION-TASK
2:      update_i(i);
3:      view ← snap_i();
4:      let k = |{j | view[j] ≠ ⊥}|;
5:      if 0 < k < n + 1 then
6:          output b^T_{n−k};
7:      else
8:          output T.invoke();
9:      end if
10: end procedure
```

Part II

Equivalence

Chapter 5

Equivalence Between IS And Snapshot Models

"A mind is a simulation that simulates itself".
Erol Ozan

5.1 The Strategy Of The Simulation

Transforming a space into another is a common technique used to solve problems. In classic computing theory we say that a problem A *reduces* to a problem B if there exists a function mapping every instance of A into an equivalent instance of B. In computability theory we use the reduction technique for proving the decidability of problems: if A reduces to B and B is decidable so is A. On the other hand, if A is undecidable and reduces to B then B is undecidable. In complexity theory we use reductions from one problem into another to show that the latter is as difficult as the former. For instance in [22] it is shown that the satisfiability problem belongs to the NP-Complete class of decision problems and then it is shown other problems belong to this class by using reductions. The key idea is that if $A \in$ NP-Complete and A reduces to B, then B should be as least as hard as A.

A *simulation* is another kind of transformation that arises in the study of computability theory. We say that a computational model R known as the real model, simulates a computational model V, known as the virtual model if there exists an algorithm $\mathcal{A}$ in R that takes as entries an algorithm $\mathcal{B}$ of V and an input x and simulates the execution of $\mathcal{B}$ with input x step

by step. This way, if $\mathcal{B}$ halts with input x so does $\mathcal{A}$. This leads us to the conclusion that the existence of $\mathcal{A}$ in R implies that any solvable problem in V is also solvable in R.

Consider two different computing models M_1 and M_2. If M_1 simulates M_2 and vice versa we say that M_1 and M_2 are equivalent. Observe that when two models of computation turn out to be equivalent they can solve the same class of problems. In computability theory, they recognize the very same languages. Any computability question about one model can be answered in the other.

Since the simulating algorithm follows blindly the steps taken by the simulated algorithm it is possible to reconstruct the execution of the simulated algorithm using the execution generated by the simulating one. Formally, let $\mathcal{A}$ be the simulating algorithm of model R that simulates algorithm $\mathcal{B}$ from model V. If $E_X(Y, Z)$ denotes the set of all executions of model X given algorithm Y with a list of inputs Z. Then, there exists a mapping function φ such that $E_V(\mathcal{B}, \lambda) = \varphi(E_R(\mathcal{A}, \langle \mathcal{B}, \lambda \rangle))$. Observe that if V is a sequential model of computation then $E_V(\mathcal{B}, \lambda)$ is a set with just one element whereas if V is a distributed model, $E_V(\mathcal{B}, \lambda)$ may consists of more elements. In the context of distributed computation it might be the case that $\varphi(E_R(\mathcal{A}, \langle \mathcal{B}, \lambda \rangle)) \subseteq E_V(\mathcal{B}, \lambda)$. Consider a situation in which $\varphi(E_R(\mathcal{A}, (\mathcal{B}, \lambda))) \subsetneq E_V(\mathcal{B}, \lambda)$. In this case, $\mathcal{A}$ might solve problems that are not solvable in V. Recall that an inherent problem in distributed computing is the lack of determinism. A restriction on the set of possible executions might represent a reduction on the amount of uncertainty. This reduction, could correspond to an increment in the amount of solvable problems.

5.2 Wait-Free Equivalence Under Decision Tasks

Deciding whether two models of distributed computing have the same computational power is one of the main concerns for distributed computing theory. An excellent success example is the equivalence between message passing and shared memory models when a majority of processes do not fail (otherwise a message passing system is less powerful) [4]. Once two models turn out to be equivalent, we are able to choose the more suitable one to solve a certain problem and then transport the solution if needed. In the rest of this section we will prove the following theorem:

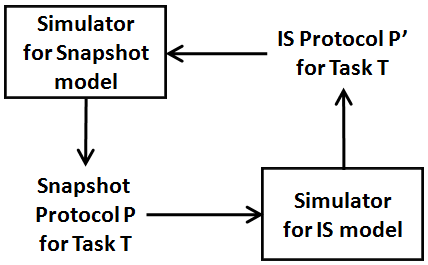


Figure 5-1: Equivalence between IS and Snapshot models from a black box perspective.

Theorem 5.2.1. *The Snapshot and the IS models are wait-free equivalent when solving decision tasks.*

Theorem 5.2.1 means that a bounded task T can be wait-free solved in the Snapshot model if and only if it can be wait-free solved in the IS model. In order to prove this equivalence, we take a constructive approach: Given a protocol $\mathcal{P}$ that solves a task in the Snapshot model, we exhibit a protocol in the IS model that simulates $\mathcal{P}$ and vice versa. Consider the Figure 5-1. That way, we don't just prove the equivalence but give a powerful tool to make the models interchangeable via a simulation. We begin the proof by stating the next lemma:

Lemma 5.2.2. *For any bounded task T wait-free solvable in the IS model, there exists a Snapshot protocol solving T.*

To avoid name conflicts, we shall consider two name spaces Π^S and Π^{IS}. We use $p_i \in \Pi^S$ to refer to a process running in the Snapshot model and use $P_i \in \Pi^{IS}$ to refer to a process running in the IS model.

To prove lemma 5.2.2, we exhibit a quite trivial protocol $\mathcal{P}$ that simulates any IS protocol $\mathcal{Q}$ into the Snapshot model. A process p_i running $\mathcal{P}$ will receive as parameters an input for process P_i running $\mathcal{Q}$ and its function δ'_i used to map a final state into a decision value.

To perform the simulation, every process p_i will hold a local variable r, initially 0 to keep track of the round in the IS model, a local variable $view_{is}$ which represents the local variable $view$ of P_i and finally an initially empty set U that will hold a pair (k, v) for every simulated $update_i(k, v)$ method call.

In order to simulate an $update_i(r, view)$ operation, process p_i adds the pair $(r, view_{is})$ to the set U and finally writes U into $mem[i]$ invoking $update_i(U)$. To simulate a $snap_i(r)$ operation,

p_i uses its $view_{is}$ variable to store a $snap_i()$ as usual, but additionally it inspects its snapshot in order to keep only the entries marked with the same value for its variable r. The complete protocol is shown in Algorithm 10. The correctness comes from the next lemma:

Lemma 5.2.3. *Algorithm 10 correctly simulates an IS protocol in the Snapshot model.*

Proof. The proof comes directly from the next observations:

1. Using variable r, p_i keeps track of the round it is simulating.
2. For each increment of the variable r, p_i performs just one *update* in which it includes the value of its counter r.
3. Process p_i only sees *updates* belonging to the same round it is simulating (lines 8 and 9).
4. By observations 2 and 3, every simulated *snap* has the effect of using a new snapshot object.

□

Algorithm 10 Simulation of the IS model in the Snapshot model (code for p_i)

```
1:  procedure TrivialSimulation(input, δ'_i)
2:      r ← 0; U ← ∅; view_is ← input;
3:      repeat
4:          r ← r + 1;
5:          U ← U ∪ {(r, view_is)};
6:          update_i(U);
7:          view_is ← snap_i();
8:          for j from 0 to n do
9:              view_is[j] ← v if (r, v) ∈ view[j] and ⊥ otherwise;
10:         end for
11:     until δ'_i(view_is) ≠ ⊥
12:     output δ'_i(view_is);
13: end procedure
```

To complete the proof of theorem 5.2.1, it is necessary to show how to take a protocol for the Snapshot model and transform it into a protocol for the IS model solving the same task in a similar way we did in Algorithm 10. The first simulation of this kind was presented in

Algorithm 11 Simulation of the Snapshot model in the IS model (code for P_i)

```
1:  procedure GRSimulation(input, δ'_i)
2:      r ← 0; c[j].clock ← 0 for j ≠ i; c[i].clock ← 1; c[i].val ← input;
3:      loop
4:          r ← r + 1;
5:          update_i(r, c);
6:          view ← snap_i(r);
7:          c ← top(view);
8:          if |c| = r then
9:              if δ'_i(c.val) ≠ ⊥ then
10:                 c[i].val ← c.val; c[i].clock ← c[i].clock + 1;
11:             else
12:                 output δ'_i(c.val) once;
13:             end if
14:         end if
15:     end loop
16: end procedure
```

[10]. It was a simulation of the Snapshot model into the Iterated Immediate Snapshot model. Algorithm 11 is a more recent simulation taken from [20].

All variables of Algorithm 11 are process-local. Each P_i maintains an $(n+1)$-vector c whose j^{th} entry $c[j]$, consists of two separated components: $c[j].clock$ is an integer value initially zero and $c[j].val$ belongs to D^*. In addition, we denote by $c.val$ the vector $c[j].val$ i.e., for all j, $c.val[j] = c[j].val$. Variable r keeps track of the current round in the execution and is inherited from the IS model. The $|c|$ and $top(view)$ operations are defined as:

$$|c| = \sum_{i=0}^{n} c[i].clock$$

$$top(view)[i] = view[j][i],\ s.t.\ \max_{j=0}^{n} view[j][i].clock,\ \text{for each } i$$

The idea behind Algorithm 11 is that every process P_i maintains three different logical clocks. One of this clocks is variable r which ticks at the beginning of every round. The second clock is $c[i].clock$ that ticks every time P_i tries to simulate a new *update* and finally a progress clock $|c|$ that measures the global progress of the simulation known by P_i.

The interpretation of the clocks is this: as usual, clock r indicates the r^{th} round reached by P_i. If $c[i].clock = k$, then $c[i].val$ represents the k^{th} simulated *update* value and when $c[i].clock$

is incremented, $c.val$ represents the k^{th} simulated snapshot. As processes do not share a global clock, process P_i needs a way to decide if it can progress its simulation or if it needs to wait for something to happen. This is precisely the intent of clock $|c|$. When $r < |c|$, we say that P_i is late and it needs to catch up its peers. When $r = |c|$, we say that P_i is on time and it can progress in its simulation.

Roughly speaking, Algorithm 11 works as follows: each time P_i is going to simulate a new *update*, it increments its own local progress clock $c[i].clock$. It then stores the new value it wants to communicate in $c[i].val$ (line 2 and 10). Finally P_i announces its c vector (line 5) and it reads the announcements of its peers by taking a snapshot (line 6). It then inspects every other process's progress and it updates its vector c accordingly (line 7). It then adds up all clocks appearing in its c vector and compares this addition with its clock r (line 8). If both clocks agree then P_i uses that last snapshot as the response of its $c[i].clock^{th}$ simulated *snap*. Otherwise, it tries to catch up its peers in the next tick of its clock r. An important thing to notice is that before line 10, the vector entry $c[i].val$ of process P_i contains the information of the last simulated *update* and after line 10, $c.val$ contains the information of the last simulated *snap*.

We assume that all simulated protocols are correct. This assumption will help us to justify all our argumentation about the correctness of the simulator protocol. The proof of Algorithm 11 comes from Invariant 5.2.4 and all subsequent lemmas. The following notation on vectors maintained by Algorithm 11 will be used:

$$c \leq c' \text{ if } c[i].clock \leq c'[i].clock, \text{for each } i$$
$$c < c' \text{ if } c \leq c' \text{ and } \exists\, i \text{ such that } c[i].clock < c'[i].clock$$

Invariant 5.2.4. *For undecided processes,* $r \leq |c|$.

Proof. Note that at the end of line 2, $r < |c|$ and r is only incremented at line 4. Therefore we must prove that after line 4 it is never the case that $r > |c|$. Let us assume for contradiction that this is not the case. Let the k^{th} iteration of the loop be the first time that r becomes

greater than $|c|$ in line 4. This implies that at the end of iteration $k-1$, $r = |c|$. Consider line 8 in iteration $k-1$. If $|c| = r$, after line 10 we get $r < |c|$, otherwise the condition at line 8 is not satisfied and $r < |c|$. Both cases lead us to a contradiction. □

Lemma 5.2.5. *All processes that complete a simulated snap operation in an iteration r, do so with the same vector c (line 7). If an undecided process does not complete a simulated snap operation in that iteration, then its vector c' satisfies $c' > c$.*

Proof. Consider all distinct snapshots obtained by undecided processes in their corresponding r^{th} round. These snapshots can be ordered by containment, $S_1 \subset S_2 \subset \cdots \subset S_m$. Consider any two such consecutive snapshots: $S_i \subset S_{i+1}$. By definition, for all $i \in [1..m]$ there exists at least one index, say j, such that $S_i[j] = \perp \neq S_{i+1}[j]$. Now take a look at $S_{i+1}[j]$. It may be the case that P_j completed a simulated *snap* operation in a previous round and now is announcing a new update, in such case we get $top(S_i) < top(S_{i+1})$ otherwise P_j did not complete a simulated *snap* operation in round $r-1$ and we get $top(S_i) = top(S_{i+1})$. Having said this, consider all distinct vectors resulting of applying *top* operation to each S_i. We can order them: $top(S_{i_1})_1, < \ldots, < top(S_{i_k})_k$. A process completes a simulated *snap* operation when $r = |c|$ in line 8, using invariant 5.2.4 we observe that all processes that complete a simulated *snap* operation ended with the same vector $c = top(S_{i_1})_1$ ($|top(S_{i_1})_1| = r$) and all processes that do not complete a simulated *snap* operation in such round end up with vector $c' > c$. □

Lemma 5.2.6. *A correct undecided process eventually completes a simulated snap and a simulated update operation.*

Proof. A process completes a simulated *update* operation with the first simulated snapshot that contains such update. When a process P_i completes a simulated *snap* operation, the resulting snapshot contains the last *update* P_i was trying to simulate (see lines 5 and 6), therefore we only need to prove that a simulated *snap* operation is eventually completed. A *snap* operation is completed when $r = |c|$. Invariant 5.2.4 establishes that for undecided processes it holds that $r \leq |c|$. Therefore, as r increases at the beginning of every round whereas $|c|$ may remain unchanged, eventually a correct undecided process completes a simulated operation. □

Lemma 5.2.7. *Every correct process eventually decides.*

Proof. Lemma 5.2.6 guaranties that, given enough iterations, a correct process completes a simulated operation. Since the number of processes is finite, eventually one correct process will complete enough simulated operations to decide. Recall that processes are simulating a protocol solving a decision task which by definition means that the simulated process must decide in a finite number of *updates* and *snaps*. Once a process decides, it stops incrementing its simulation progress clock allowing the rest of the processes to continue progressing. Repeating this argument, eventually all correct processes decide. □

Note from the last proof that if the protocol we are trying to simulate is not a protocol solving a decision task but the implementation of a *long-lived object*, then processes simulating the protocol do not necessarily end the simulation. A process can be forever ahead of the rest and its peers will never have the opportunity to catch up. A situation known as *starvation*. Examples of long-lived objects are *distributed queues*, *distributed stacks*, or even a snapshot object. Lemmas 5.2.6 and 5.2.7 deal with the termination of Algorithm 11. To prove that the algorithm reproduces a valid execution of the protocol it is simulating, let us prove Lemma 5.2.8.

Lemma 5.2.8. *An execution of Algorithm 11 encodes a valid execution of the Snapshot model within the IS model.*

Proof. Processes running Algorithm 11 move asynchronously through a sequence of rounds. In each round r, a subset of processes (possibly empty) $S_r \subseteq \Pi$ succeeded in completing a simulated *snap* operation. By Lemma 5.2.5, we know all processes belonging to S_r completed their simulated *snap* operation with the very same vector c. Consider the sequence of vectors: $c^0, c^1, \ldots, c^m$ where c^i is the vector obtained by processes $S_{|c^i|}$ and $|c^i|$ represents the i^{th} round such that $S_{|c^i|} \neq \emptyset$. Lemma 5.2.5 also implies that $c^i < c^{i+1}$. Moreover, if $c^i < c^{i+1}$ then there must exists a set of processes $U_{|c^{i+1}|}$ such that for every $j \in U_{|c^{i+1}|}$, $c^i[j] < c^{i+1}[j]$. Thus, processes $S_{|c^{i+1}|}$ completed their simulated *snap* in round $|c^{i+1}|$ and before this, processes $U_{|c^{i+1}|}$ completed a simulated *update*. If we abuse notation and use $U_{|c^i|}$ to mean a set of consecutive *update* operations linearized in any order and use $S_{|c^i|}$ to mean a set of consecutive *snap* operations linearized in any order, then we obtain the following execution:

$$C_0, U_{|c^1|}, C_1, S_{|c^1|}, C_2, \ldots, U_{|c^m|}, C_{2m-1}, S_{|c^m|}, C_{2m},$$

Which is a valid execution in the snapshot model. □

Lemma 5.2.9. *For any bounded task T wait-free solvable in the Snapshot model, there exists an IS protocol solving T.*

Proof. Let $\mathcal{P}$ be a protocol in the Snapshot model solving task T. By Lemmas 5.2.7 and 5.2.8, Algorithm 11 wait-free simulates $\mathcal{P}$. □

We can now prove Theorem 5.2.1 which can be restated as: A bounded task T can be wait-free solved in the Snapshot model if and only if it can be wait-free solved in the IS model.

Proof. By Lemma 5.2.2 proved using Algorithm 10, for any task T solvable in the IS model there exists a protocol in the Snapshot model solving T. On the other hand, by Lemma 5.2.9, any protocol solving task T in the Snapshot model can be simulated in the IS model. □

5.3 Adversary Equivalence Under Decision Tasks

In Section 5.1 the equivalence between Snapshot and IS models was shown when solving decision tasks in a wait-free model of failures. In this section, we show how to extend such equivalence in the adversary model of failures which is more general than the wait-free model.

In the IS model with an adversary $\mathcal{A}$, each process P_i *updates* the memory as usual but it then repeatedly takes as many snapshots as it needs in order to get a snapshot that includes a set of new values written by a survivor set. To achieve this behavior we replace line 6 of Algorithm 11 with the following:

6: **repeat** $view \leftarrow snap_i(r)$ **until** for some survivor set S, $S \subseteq \{j \mid view[j] \neq \perp\}$;

This change does not affect Lemma 5.2.6 since the replaced line always completes because adversary $\mathcal{A}$ guarantees that some survivor set eventually perform an *update* into snapshot object O_r and therefore Lemma 5.2.7 also holds. It might be the case that a process running the previous line observes old values. What we need to show is that the simulated protocol is

fair with respect adversary $\mathcal{A}$. That is, every simulated snapshot contains new values written by a survivor set. Consider the next Lemma:

Lemma 5.3.1. *If a process P_i completes a simulated snap operation in iteration r with a vector c, and the next round it completes a simulated snap is round $r + k$, with vector c', then there are at least $|S|$ entries j, for some survivor set S, such that either $c[j].clock < c'[j].clock$ or P_j is decided.*

Proof. Consider two sets A and $\overline{A}$ such that $\Pi = A \cup \overline{A}$ and $A \neq \emptyset$. The set A is the set of processes that complete a simulated operation in iteration r and $\overline{A}$ is the set of processes that do not complete a simulated *snap* operation in iteration r and may be empty. By Lemma 5.2.5, all processes in A end round r with the same vector c and all processes in $\overline{A}$ end with a vector larger than c. Observe that all undecided non faulty processes asynchronously reach round $r + 1$ with a vector c' larger than c. Consider a process $P_i \in A$ that reaches round $r + 1$. Process P_i invokes line 6 obtaining a *view* which includes a survivor set S which must satisfy $view[j].clock < view'[j].clock$ if $j \in S$ is undecided and therefore each simulated *snap* contains a survivor set which is fair with respect to $\mathcal{A}$. □

Now that the algorithm has been presented, we might be interested in answering some questions that arise naturally in the study of algorithms. As I already pointed out, this work has not to do with complexity issues. Nevertheless, in order to fully understand how Algorithm 11 works, we consider some complexity questions. First of all, consider a set of $n + 1$ processes in the Snapshot model executing a protocol $\mathcal{P}$ which needs k rounds to solve a decision task T. How many rounds of the IS model are sufficient for a process P_i executing Algorithm 11 to simulate a single *snap* in $\mathcal{P}$? Suppose in the first round, process P_0 executes line 6 obtaining a *view* in which it only sees its own write. After line 6, $|c|$ and r become equals and P_0 completes its first simulated *snap*. In this case, P_0 needed only one round to simulate an operation from $\mathcal{P}$. This is of course the best case. We will say that a process P_j *blocks* a process P_i in round r if the snapshot Si taken by P_i in round r is such that $S_i[i] < S_i[j]$. If in the first round, P_0 and P_1 concurrently write their respective states to memory, then they both block to each other and they cannot complete their corresponding first operation in the first round. Note that in the second round they are going to complete their operation independently of the order in which

they run unless a third process blocks them. How many times a process can be blocked? In the last example, neither P_0 can block P_1 nor P_1 can block P_0 because they have already read each other and they cannot write something new until they finish their current simulated operation. In a fully concurrent step, all N processes block each other in the first round. To proceed with the next simulated operation $|c|$ and r must be equal. Thus, they need to complete N rounds in any order before other blockage takes place. Then a process P_j can block a process P_i one time per operation and each time a process is blocked it delays the termination of its simulated operation one round. Consider the following scenario:

Suppose P_0 finishes its simulation in a solo execution fashion. Then P_1 starts taking steps, it cannot complete its first operation in round 1 since P_0 blocks him in the first round, then P_1 delays the termination of its first simulated operation to the next round. In its second round P_1 is blocked again by the new value written by P_0 and so on until the round k in which P_0 finishes the protocol and stop writing new values to memory. Then P_1 completes its first operation in round $k+1$ and finishes the simulation in round $2k$. Now a third process starts taking steps and complete its operation in round $2k+1$ and finishes the simulation in round $3k$, then process P_2 start taking steps and so on. Finally, process P_n starts taking steps and finishes its first operation in round $(n-1)k+1$. This is the worst possible scenario for a process to complete an operation and in this case, it finishes the protocol in nk rounds.

5.4 Extended Simulation With 01-Exclusion Tasks

Consider the Snapshot model extended with 01-exclusion tasks and its iterated counterpart. A natural question is whether these models are equivalent. An extension of our protocol 10 to simulate an IS protocol extended with objects of class x is quite trivial because the order in which invocations occurs in an extended IS model are a restricted form of the order in which invocations occurs in the appropriate extended Snapshot model, provided each communication object in both models is invoked only once. A more interesting question is whether we can simulate the Snapshot model extended with 01-exclusion tasks into its iterated counterpart.

Let us assume that we are given a canonical Snapshot protocol $\mathcal{P}$ extended to work with 01-exclusion tasks. We are asked to simulate it in the corresponding extended IS model. How

we should proceed with the simulation? If $\mathcal{P}$ does not use any additional object we could simply use Algorithm 11. The problem arises when $\mathcal{P}$ uses additional tasks, specially if $\mathcal{P}$ uses them in a mixed order. In [20] this simulation was addressed using the idea of announcements. The authors considered that we could ask $\mathcal{P}$ to include an announcement before every invocation. Our simulation is similar to [20] with a small difference: If p_i is about to invoke a task in $\mathcal{P}$, then P_i is responsible for adding the corresponding announcement. The reason to take this approach rather than the approach of the authors is that the problem of realizing that a simulated process it is going to perform an announcement seems as hard as the original problem of realizing if the simulated process is going to invoke a task. The authors of [20] did not include the algorithm but the underlying idea. This is perhaps the first place where the simulation is presented in algorithmic form. It is presented in Algorithm 12.

Processes running Algorithm 12 write announcements into memory. This announcements are not part of the simulated protocol $\mathcal{P}$. Thus, we modify the vector c used in Algorithm 11 in such a way that each entry in the vector c be a triplet. For all j, the entry $c[j]$ is of the form $\langle u, v, w\rangle$ where $c[j].clock = u$, $c[j].val = v$, and $c[j].ann = w$. In particular, $c[j].ann$ is the field used by process P_j to announce a task.

A process P_i running Algorithm 12 will receive as parameters an *input* for the simulated process p_i, the function δ'_i used by p_i to map a final state into a decision value and the function ϕ'_i used by p_i to evaluate the next object to be invoked. Algorithm 12 includes a local variable $flag$ not appearing in Algorithm 11. As long as variable $flag = 0$, the simulation proceeds equally to Algorithm 11. The only difference is that we need to communicate more information. Observe that $c[i].val$ is now a pair consisting of the last snapshot taken and the response of the last invocation. In Algorithm 11 when $|c| = r$, our simulated *snap* is completed and we can use $c.val$ as the input for the next simulated *update*. In this extended version of the simulation, we additionally need to ask if there is a pending *invoke* operation. That is, if previously to the *snap* operation, the simulated process p_i has to perform an *invoke* operation. This is done in line 22. Observe that $c[i].val$ is the last value written into memory, so we use $\phi'_i(c[i].val)$ to evaluate if p_i has a pending task. If it turns out that $\phi'_i(c[i].val) \neq \top$, we pause the simulation by incrementing $flag$ (see line 29). When $flag$ goes from 0 to 1, the simulator enters into an *announcement mode*. In this mode, the clock mechanism used to simulate snapshots is used to

Algorithm 12 Simulation of the Snapshot model in the IS model: extension to 01-exclusion tasks (code for P_i)

```
1: procedure 01GRSimulation(input, δ'_i, φ'_i)
2:     r ← 0; flag ← 0;
3:     c[j].clock ← 0, c[j].val ← ⊥, c[j].ann ← ⊤ for all j;
4:     c[i].clock ← 1; c[i].val ← (input, ⊤);                          ▷ Initialization
5:     loop
6:         r ← r + 1;
7:         update_i(r, c);
8:         if flag = 2 ∧ c[i].ann has the form ⟨T, λ⟩ then            ▷ φ'_i(c[i].val) = ⟨T, λ⟩
9:             flag ← flag + 1;
10:            if ∃ j : c.val[j] contains ⟨T, λ', b⟩ then
11:                call ← ⟨T, λ, 1 − b⟩;                               ▷ Use previous result
12:            else if |{P_j : c[j].ann is of the form ⟨T, −⟩}| = k < N then
13:                call ← ⟨T, λ, 1 − b_k^T⟩;                           ▷ Avoid forbidden output vector
14:            else
15:                call ← ⟨T, λ, invoke_i(⟨D(T), λ⟩)⟩;                  ▷ Invoke dual
16:            end if
17:        end if
18:        repeat view ← snap_i(r); until ∃ survivor set S : S ⊆ {j | view[j] ≠ ⊥};
19:        c ← top(view);
20:        if |c| = r then
21:            c[i].clock = c[i].clock + 1;
22:            if call ≠ ⊤ ∨ φ'_i(c[i].val) = ⊤ then                   ▷ P_i does not have pending tasks
23:                flag ← 0; c[i].ann ← ⊤;                             ▷ Exit invocation mode
24:                c[i].val ← (c.val, call);
25:                if δ'_i(c[i].val) ≠ ⊥ then
26:                    output δ'_i(c[i].val) once;
27:                end if
28:            else
29:                flag ← flag + 1;                                    ▷ Enter invocation mode
30:                c[i].ann ← φ'_i(c[i].val);                          ▷ Prepare announcement
31:            end if
32:        end if
33:    end loop
34: end procedure
```

announce that a task is going to be simulated (see line 30).

Once a process P_i enters the announcement mode, it spends one or more rounds writing its announcement (see line 7) until $|c| = r$. Note that when $|c| = r$ and $flag$ is set to 1, the condition in line 22 is not satisfied and then flag goes from 1 to 2 in line 29. Thus, in the next round the condition in line 8 must be satisfied and *call* must receive a value different from $\top$. So, the next time $|c| = r$, the condition in line 22 is satisfied and P_i exits the announcement mode (see line 23).

When the predicate of line 8 is satisfied, P_i executes the block between lines 8 and 17. In this block it will execute one of three cases. In the first case (line 11) it does not invoke the task but simulates it using a previous published result of some other process. In the second case (line 13) it again simulates its invocation by choosing a value that avoids the forbidden output of the task when $k < N$ processes participate. Finally, in the third case (line 15), it invokes the dual of the task. In any of the three cases it gets an output for the task and stores it in *call*. At this point $flag = 3$ so as long as $|c| > r$ the simulation will not execute any of the three cases again.

Note that in an execution in which the predicate in line 22 is always satisfied, Algorithm 12 behaves exactly the same as Algorithm 11 except for line 6. Moreover, even if the predicate in line 22 is eventually unsatisfied, it does not affect our Invariant 5.2.4: consider a round r in which after line 6, $r = |c|$. Note that $c[i].clock$ will be incremented in line 21. Thus, the invariant will hold in round r and in the next. As invariant 5.2.4 holds, so do our lemmas 5.2.5 and 5.2.6. Also note that once $flag$ is set to 1, by Lemma 5.2.6, $flag$ will eventually be set to 2 and the code within the condition of line 8 will be executed which means that the predicate in line 22 is eventually true. This implies that an *invoke* operation is eventually simulated and the Lemma 5.2.7 remains valid in Algorithm 12. We just need then to proof that the simulation of each task is correct and that the order in which the operations are simulated is a valid execution in the Snapshot model extended with 01-exclusion tasks.

Lemma 5.4.1. *All processes invoking $D(T)$, invoke it in the same round.*

Proof. Consider the first round r in which a set of processes A invokes $D(T)$. Suppose for the sake of contradiction that not all processes invoking $D(T)$, invoke it in the same round. Since

r is the first time in which a set of processes invoke $D(T)$, it must be the case that a set of processes B execute $D(T)$ in a round r' with $r' > r$. Observe that it is necessary that all processes in A running round $r-1$ obtain a vector c such that $|c| = r-1$, otherwise they do not increment their $flag$ in line 29 and they do not satisfy the condition in line 8 in round r. Moreover, since processes in A invoke $D(T)$ in line 15, processes in A read $n+1$ announcements for task T in round $r-1$. Processes in B did not invoke $D(T)$ in round r but processes in A read announcements of processes in B. By Lemma 5.2.5, processes in B obtain a vector c' in round $r-1$ such that $|c'| > r-1$ but this is impossible because vector c contains all values written in round $r-1$ and $|c| = r-1$. □

Lemma 5.4.2. *The simulation of each task T is correct.*

Proof. The simulation of an *invoke* has three cases:

1. Case 1 (line 11): It is safe to return $1-b$ because any output vector which contains two different values is a valid output vector for c. Recall from Subsection 2.2.3 that if $k < N$ process participate, not all can decide bit b_k^T. In addition when all processes participate any output vector which includes a one and a zero is a valid output vector.

2. Case 2 (line 13): There is a set of processes of size k trying to invoke task T. It is safe to output $1 - b_k^T$ since processes are simply avoiding the forbidden output vector in which all of them decide bit b_k^T. If a set of processes A observes all $n+1$ announces, they will invoke $D(T)$. The combined set of outputs from A and from processes invoking case 2 is valid for property 3 of 01-exclusion tasks. (see Subsection 2.2.3).

3. Case 3 (line 15): If all N processes invokes $D(T)$, the resulting output vector is a valid output vector for T. Recall that when all processes participate both tasks behave the same.

□

Lemma 5.4.3. *An execution of Algorithm 12, encodes a valid execution of the Snapshot model extended with 01-exclusion tasks within the IS model extended with 01-exclusion tasks.*

Proof. Let $\mathcal{P}$ denote the Snapshot protocol being simulated by Algorithm 12. As we discuss before, if $\mathcal{P}$ does not make invocations, Algorithm 12 behaves just like Algorithm 11 which by Lemma 5.2.8 encodes a valid execution of the Snapshot model:

$$C_0, U_{|c^1|}, C_1, S_{|c^1|}, C_2, \ldots, U_{|c^r|}, C_{2r-1}, S_{|c^r|}, C_{2r},$$

Assume protocol $\mathcal{P}$ includes invocation calls. Consider the moment in which a process P_i running Algorithm 12 does not satisfies the condition in line 22 (it has to invoke a task). What follows is that P_i enters the announcement mode by incrementing its $flag$ and adding the corresponding announcement (see line 30). From this moment, P_i stops updating $c[i].val$ until it finally completes the simulated invocation, that is, $call \neq \top$. P_i then replaces the announcement in $c[i].val.call$. When $flag$ goes from 3 to the value 0 in line 23, we linearize the simulated *snap* operation. In this case, the value of the snapshot is $c[i].val$. Just before this linearized *snap* there is a linearized *invoke* occurring within lines 8 to 17 occurring after the previous *update*. As usual, we linearize an *update* before the first snapshot that contains it. An invocation is then linearized between a linearized *update* and a linearized *snap*. This lead us to the next execution:

$$C_0, U_{|c^1|}, C_1, I_{|c^1|}, C_2, S_{|c^1|}, C_3, \ldots, U_{|c^r|}, C_{4r-3}, I_{|c^r|}, C_{4r-2}, S_{|c^{r+1}|}, C_{4r-1},$$

Where $I_{|c^i|}$ is a set of linearized *invoke* operations in round $|c^i|$ that may be empty. This is clearly a valid execution in the extended Snapshot model. □

Lemma 5.4.4. *The simulation by protocol 12 is fair with respect adversary $\mathcal{A}$.*

Proof. Each time a process P_i is going to simulate an *update*, an *invoke* and a *snap* operations in a row, it first behaves as in Algorithm 11 as if it were to simulate the *update* and the *snap* operation. By Lemma 5.3.1 this behavior is fair with adversary $\mathcal{A}$. P_i keeps its vector c updated so when P_i completes the simulation of the aforementioned operation, its vector c contains new values written by decided and undecided processes that belong to a survivor set S. □

Theorem 5.4.5. *Algorithm 12 correctly simulates an extended Snapshot protocol with* 01*-exclusion tasks in the IS model extended with* 01*-exclusion tasks.*

Proof. By Lemma 5.2.7 a process eventually decides. And by Lemma 5.4.2 all simulated 01-exclusion tasks are correct. In addition by Lemma 5.4.3 an execution of Algorithm 12 produces a correct linearized execution of the Snapshot model extended with 01-exclusion tasks. □

Corollary 5.4.6. *The Snapshot model extended with* 01*-exclusion tasks and the IS model extended with* 01*-exclusion tasks are equivalent when solving decision tasks.*

Chapter 6

Contribution: Equivalence Under x-consensus Objects

6.1 Basic Simulation

The following simulation is due to Gafni and Borowsky. It was informally presented in [10]. In this thesis, this simulation is revisited in detail. All necessary proofs are presented and later on it is extended to prove the equivalence between the Snapshot model extended with x-consensus objects and its iterated counterpart.

The difference between the next simulation and the one presented in section 5.3 lies in the conditions under which a simulated *snap* is linearized. Whereas in Algorithm 11, a *snap* is linearized when $|c| = r$, in this simulation we require that in each round, every process P_i takes a snapshot to see the progress of its peers and if it turns out that for every other process P_j whose write appears in its snapshot, P_j is aware of all operations known by P_i and P_i is aware of all operations known by P_j, then it is safe for P_i to finish its current simulated *update* operation and use this last snapshot as its simulated snapshot. This would be sufficient for a wait-free model of failures but in order to model the adversary case we additionally require that P_i finishes its current simulated *snap* only if its last snapshot contains either new values or a special symbol ∇ written by a survivor set S. This simulation is presented in Algorithm 13.

In this simulation, once a process becomes *decided*, it explicitly invalidates its vector c assigning it a special symbol ∇ (see line 15). This way, if a *decided* process belongs to some

Algorithm 13 Simulation of the Snapshot model in the IS model with adversary A (code for P_i)

```
1: procedure ASimulation(input, δ'_i)
2:     r ← 0; c[j].clock ← 0 for j ≠ i; c[i].clock ← 1; c[i].val ← input;
3:     loop
4:         r ← r + 1;
5:         update_i(r, c);
6:         repeat view ← snap_i(r); until for some survivor set S, S ⊆ {j | view[j] ≠ ⊥};
7:         if c ≠ ∇ then                          ▷ Prevent decided process to update its vector
8:             c ← top(view);
9:             if ∀a, b ∈ [n] : view[a] ∈ {⊥, ∇} ∨ view[a][b].clock = view[i][b].clock then
10:                if ∃ survivor set S: S ⊆ {j | c[j] = ∇ ∨ c[j] ≠ c[i].val[j]} then
11:                    if δ'(c.val) ≠ ⊤ then
12:                        c[i].val ← c.val; c[i].clock ← c[i].clock + 1;
13:                    else
14:                        output δ'_i(c.val);
15:                        c ← ∇;
16:                    end if
17:                end if
18:            end if
19:        end if
20:    end loop
21: end procedure
```

survivor set, an undecided process can distinguish it in line 6 but it will be ignored in line 9. It is important to note that unlike Algorithm 11, if a *decided* process P_i continues updating its vector it can indefinitely block another process P_j to make progress if each time P_j executes line 6 it gets a *view* in which $view[i][j].clock < view[j][j].clock$. The correctness of Algorithm 13 comes from the following lemmas:

Lemma 6.1.1. *If some process P_i completes a simulated snap operation in round r with vector $c = top(view)$ then $c[a] = view[i][a]$ for all $a \in [n]$.*

Proof. A process P_i completes a simulated *snap* operation when the predicates in lines 9 and 10 are satisfied and $c.val$ is the corresponding simulated snapshot. Observe that for the predicate in line 9 to become *true*, it is required that all clocks read by P_i in its *view* be equal to the clocks contained in $view[i]$. In addition, note that by line 12, a process updates both $c[i].clock$ and $c[i].val$ before performing an *update*. Both observations lead to the conclusion that the *top* operation returns the same values located at $view[i]$. □

Lemma 6.1.2. *Consider all vectors c_j obtained in round r at line 8. We can order them using the relation $\leq$.*

Proof. Consider all snapshots taken in round r at line 6, we can order them by containment: $view_1 \subseteq view_2 \subseteq \cdots \subseteq view_k$. Observe that by the definition of the *top* operation, for any two snapshots $view_i \subseteq view_j$ and for all $a \in [n]$, $top(view_i)[a].clock$ is at least as great as $top(view_j)[a].clock$. □

Lemma 6.1.3. *All undecided processes that complete a simulated snap operation in a round r (line 9), do so with the same vector c.*

Proof. Suppose, to derive a contradiction, that in a round, say r, two processes P_i and P_j completed a simulated operation with vectors $c_i = top(view_i)$ and $c_j = top(view_j)$ respectively such that $c_i \neq c_j$. By Lemma 6.1.2 we can assume without loss of generality that $c_i < c_j$. In addition, by Lemma 6.1.1 we know that $c_i[a] = view_i[i][a]$ for all $a \in [n]$. Now we have the required contradiction since $c_i < c_j$ means that there exists at least one x for which $c_i[x].clock < c_j[x].clock$ which implies that $view_i[i][x].clock < view_j[j][x].clock$. If we assume without loss

of generality that $view_j[i] \neq \perp$, then $view_j[i][x] \neq view_j[j][x]$ and then by Lemma 6.1.1 P_j did not complete an operation in round r which is a contradiction. □

Lemma 6.1.4. *Let c^r and c^s be two vectors obtained when completing a simulated operation in rounds r and s respectively. Then, it holds that $r < s \Rightarrow c^r \leq c^s$.*

Proof. In round r *undecided* processes can be split into two sets: the set A of all processes that completed a simulated *snap* operation and its complement $\overline{A}$. In turn, processes in A can be split into two categories, the ones that become *decided* in round r will not complete more simulated operations and we can discard them, by contrast the ones that do not become *decided* execute line 12 and the next time any of them completes an operation will be with a vector larger than c^r. On the other hand, processes in $\overline{A}$ can be split into two categories too. Those processes that become faulty in round r can be discarded since they will not complete more simulated operations in later rounds. Consider the first non-faulty process P_i in $\overline{A}$ that got $false$ in the predicate in line 9. It must be the case that its *view* contains the writes of all processes belonging to A and no process in A saw the write by P_i, otherwise at least some process in A say P_j obtains a $view_j$ such that $\forall\, b \in [n] : view_j[j][b].clock = view_j[i][b].clock$ and therefore either P_i should belong to A or P_j should belong to $\overline{A}$ because of the inclusion property of snapshots. This imply that all non-faulty process in $\overline{A}$ saw the writes of all process in A and therefore after line 8 in round r all non-faulty processes in $\overline{A}$ end up with a vector at least as updated as c^r and the next time any of them completes a simulated operation will be with a vector at least as large as c^r. □

Lemma 6.1.5. *Consider a round r in which no process satisfies the predicate in line 9. If the number of undecided non-faulty processes in round r is α, at most, in the following $\alpha - 1$ rounds, at least one process P_j takes a snapshot and obtains true for the predicate in line 9:* $\forall\, a, b \in [n] : view[a] \in \{\perp, \nabla\} \vee view[a][b].clock = view[j][b].clock$.

Proof. Consider a round r in which no process took a snapshot such that the predicate in line 9 was satisfied. Assume for a contradiction that in the following $\alpha - 1$ rounds, no process obtained *true* in the above mentioned predicate. All different snapshots taken in round r can be ordered by containment: $view_1 \subset \cdots \subset view_m$. Observe that $m < \alpha$, otherwise some process only reads

its own write and the predicate would become *true*. Consider all different vectors resulting from applying the *top* operation in line 8: $c_1 < \cdots < c_k$ where $k \leq m < \alpha$. Observe that in round $r+1$ no process is able to obtain vector c_1 at line 8, otherwise the one who obtains it satisfies the predicate. The same goes for vector c_2 in round $r+2$ and so on. Therefore, at the end of round $r+k-1$ all process will get vector c_k but then, at round $r+k < r+\alpha$ any process reaching line 9 will necessarily satisfy the predicate, which is a contradiction. □

Lemma 6.1.6. *Eventually, all non-faulty processes complete enough simulated snap operations and decide a value.*

Proof. We first show that at least one *snap* operation is simulated: By Lemma 6.1.5, there exists a first round in which a set of processes $A \subseteq \Pi$ satisfy the predicate in line 9. In that round, processes in A take a snapshot containing the values written by some survivor set S (see line 6). Moreover, all values read in that round are *new* in the sense that there is no previous simulated snapshot containing any such values. Thus, all processes in A satisfy the predicate in line 10, and the first simulated snapshot is linearized.

Assume for a contradiction that there exists a round r at which some correct processes are undecided and from which no more *snap* operations are simulated. As noted before, a process does not complete a simulated *snap* operation only if it does not satisfy the predicate in line 9, the predicate in line 10 or both. As the number of processes is finite and no process P_i updates its vector entry $c_i[i]$, there must exist a round $r' > r$ from which no process *learns* new information (i.e. its vector c remains immutable in later rounds). Consider the set B of processes that first completed line 6 in round r'. By assumption from round r', all processes in B satisfy the predicate in line 9. By line 6, this set of processes includes a survivor set S. Pick $P_i \in B$ such that the last round $k < r$ in which P_i completed its last simulated *snap* is minimum among B. As P_i does not satisfy the predicate in line 10 there exists at least one process $P_j \in B$ such that $\nabla \neq c_i[j] \neq c_i.val[j]$. As the last simulated snapshot taken by P_i ($c_i.val$) contains that *old* value ($c_i.val[j]$) there must be the case that P_j completed its last simulated *snap* operation before P_i, contradicting the election of P_i. Repeating this argument, eventually all non-faulty processes completes enough simulated *snap* operations and decide. □

Corollary 6.1.7. *Algorithm 13 is non-blocking.*

Proof. As a consequence of Lemma 6.1.6, there is no infinite execution of Algorithm 13 in which all correct processes stop making progress. □

Lemma 6.1.8. *An execution of Algorithm 13 encodes a valid execution of the Snapshot model within the IS model.*

Proof. Processes running Algorithm 13 move asynchronously through a sequence of rounds. In each round r, a subset of processes, possibly empty, succeed in completing a simulated *snap* operation. An *update* is linearized immediately before the first simulated *snap* that contains it. Consider the sequence of all vectors obtained in each round in which an operation is completed: $c_{r_1}, c_{r_2}, \ldots, c_{r_k}$ where c_{r_j} stands for the vector obtained by processes that completed a simulated operation in round r_j. By Lemmas 6.1.3 and 6.1.4 we have: $c_{r_1} \leq c_{r_2} \leq \cdots \leq c_{r_k}$. We can simplify this even more considering the sequence of all different vectors we get $c_{r_1} < c_{r_2} < \cdots < c_{r_j}$ where $j \leq k$. Observe that previous to every vector c_{r_i} there exists one or more updates, therefore if we define S^{r_i} as the set of *snaps* of processes that completed an operation with vector c_{r_i} and define U^{r_i} as the set of *updates* that appear in vector c_{r_i} then we have the following execution:

$$C_0, U^{r_1}, C_1, S^{r_1}, C_2, \ldots, U^{r_k}, C_{2k-1}, S^{r_k}, C_{2k},$$

Where C_i is a configuration and we can add the events in U^{r_i} in any order between C_i and C_{i+1} and the same goes for the events within S^{r_i}. This is a valid execution in the snapshot model.

□

Corollary 6.1.9. *The simulation of protocol $\mathcal{P}$ produced by Algorithm 13 is fair with respect to adversary $\mathcal{A}$.*

Proof. By Lemma 6.1.6 all non-faulty processes eventually decide a private output value and by Lemma 6.1.8 the simulated execution generated by Algorithm 13 of protocol $\mathcal{P}$ is valid. To see that every snapshot contains new values written by a survivor set it is sufficient to observe that when a process completes a simulated snapshot it satisfies the condition in line 10. □

Theorem 6.1.10. *Algorithm 13 simulates snapshot protocols with adversary $\mathcal{A}$.*

Proof. By Lemma 6.1.8 Algorithm 13 correctly simulates an execution of any protocol in the Snapshot Model. By Lemma 6.1.5, if the simulated snapshot protocol terminates with adversary $\mathcal{A}$, all non-faulty processes running Algorithm 13 eventually decide, thus Algorithm 13 is correct. □

6.2 The Extension

Now we are going to extend our Algorithm 13 to simulate a protocol in the snapshot model extended with x-consensus tasks into its iterated counterpart.

A x-consensus task is a communication object that solves consensus among x processes. Consider the One Round One Object approach described in Subsection 3.3.2. When $x = N$, all process might access a x-consensus task in each round. On the contrary, when $x < N$ then we enumerate all subsets of Π of size x so each subset is allowed to access a task by turns in a round robin fashion. Each process uses the function μ described in Section 3.3.2 to know if it is allowed to invoke an object.

Let $\mathcal{P}$ be a Snapshot protocol extended with x-consensus tasks that we want to simulate into its iterated counterpart. As in the simulation with 01-exclusion tasks, we assume each process P_i is given a function δ'_i used by p_i to decide an output value and a function ϕ'_i used by p_i to get the next object to be invoked. This time we use T to refer to a task used by processes running the simulation algorithm and O to refer to a task used by processes running $\mathcal{P}$. The simulator we are about to introduce may seem complicated at first glance but it lays upon five basic points informally explained below:

1. In every round in which a process P_i is allowed to invoke an x-consensus task, P_i must invoke it unless P_i already decided an output value. P_i must keep all responses from such invocations.

2. There is no one-to-one correspondence between the tasks used by the simulated protocol and the tasks used in the simulator. Let us suppose that a process P_i reaches a round r in which it is allowed to invoke task $T_r^{\mu(r)}$. In that round, P_i evaluates function ϕ'_i. If it gets $\langle O_k^{\pi_y}, \lambda \rangle$ as response such that $\pi_y = \mu(r)$, P_i must invoke task $T_r^{\mu(r)}$ using $\langle O_k^{\pi_y}, \lambda \rangle$ as parameter unless it observes task $O_k^{\pi_y}$ was already simulated among the set of all

invocations saved in the previous point. Consider the first thing to happen. Two cases arise:

(a) Task $T_r^{\mu(r)}$ returns the same parameter as the one used by P_i in the invocation (that is, the response of the invocation is $\langle O_k^{\pi_y}, \lambda \rangle$). In this case, P_i succeeds in simulating task $O_k^{\pi_y}$ and we say that P_i *won* the simulated x-consensus object $O_k^{\pi_y}$.

(b) If the object did not return the same parameter that the one used by P_i then two more cases arise:

i. The response of task $T_r^{\mu(r)}$ includes the same task that the one used by P_i in its invocation with a different parameter. In this case, P_i succeeds in simulating task $O_k^{\pi_y}$ and we say that P_i *loses* the simulated x-consensus object $O_k^{\pi_y}$.

ii. The response does not include the task used by P_i. In this case P_i does not succeed in simulating task $O_k^{\pi_y}$ and it needs to wait for another round to try it again.

3. If P_i cannot complete a simulated *snap* operation because it has not simulated the task indicated by ϕ_i', the process announces the task in the next *update* so the other processes read the announcement and they may help it to make progress.

4. If P_i reaches a round in which it is allowed to invoke a task $T_r^{\mu(r)}$ but it turns out that P_i does not have a pending task in its current state of the simulation (that is, the evaluation of ϕ_i' results in $\top$) or if its pending task is defined over a different set of processes that $\mu(r)$, it tries to help some process whose announcement appears in its vector.

5. If P_i does not have a pending task or if it can not use the current allowed task to simulate its pending one because it is not defined over the correct subset of processes, and there is no other process it can help: P_i tries to invalidate the current task by invoking it with a special value $\top$ so no other process can use it.

The simulation is presented in Algorithm 14. Observe that between the *update* and the *snap* operations of Algorithm 13 we introduce some conditions, none of which can affect Lemmas 6.1.1,6.1.2,6.1.3, 6.1.4 and 6.1.5. Nonetheless, we need now to correct our Lemmas 6.1.6, and 6.1.8. The proof for the correctness of Algorithm 14 comes from the following lemmas:

Algorithm 14 Simulation of the Snapshot model in the IS model: extension to x-consensus tasks (code for P_i)

```
1: procedure x-ConsensusSnapshotSimulation(input, δ'_i, φ'_i)
2:     r ← 0; bag ← {⊤}; call ← ⊤;
3:     c[j].clock ← 0, c[j].val ← ⊥, c[j].ann ← ⊤ for all j;
4:     c[i].clock ← 1; c[i].val ← (input, ⊤);
5:     loop
6:         r ← r + 1;
7:         update_i(r,c);
8:         if i ∈ π_{μ(r)} ∧ c ≠ ∇ then                          ▷ Is your turn to invoke an object
9:             if call = ⊤ ∧ φ'_i(c[i].val) is of the form ⟨O_k^{μ(r)}, λ⟩ then      ▷ Pending task
10:                if bag contains an entry ⟨O_k^{μ(r)}, res⟩ then      ▷ Has this task been simulated before?
11:                    call ← ⟨O_k^{μ(r)}, λ, res⟩;                    ▷ Invocation simulated!
12:                    inv ← invoke_i(⟨T_r^{μ(r)}, ⊤⟩);                ▷ Try to invalidate this task
13:                else
14:                    inv ← invoke_i(⟨T_r^{μ(r)}, ⟨O_k^{μ(r)}, λ⟩⟩);   ▷ Try to simulate O_k^{μ(r)}
15:                    if inv is of the form ⟨O_k^{μ(r)}, λ'⟩ then
16:                        call ← ⟨O_k^{μ(r)}, λ, λ'⟩;                 ▷ Invocation simulated!
17:                    end if
18:                end if
19:            else if ∃ j : j ∈ π_{μ(r)} ∧ c[j].ann is of the form ⟨O_x^{μ(r)}, λ⟩ ∧ ⟨O_x^{μ(r)}, −⟩ ∉ bag then
20:                inv ← invoke_i(⟨T_r^{μ(r)}, c[j].ann⟩);              ▷ Try to help process P_j
21:            else
22:                inv ← invoke_i(⟨T_r^{μ(r)}, ⊤⟩);                     ▷ Try to invalidate this task
23:            end if
24:            if inv ≠ ⊤ then bag ← bag ∪ {inv}; end if                ▷ Keep this response!
25:        end if
26:        repeat view ← snap_i(r); until for some survivor set S, S ⊆ {j | view[j] ≠ ⊥};
27:        if c ≠ ∇ then                                                ▷ If P_i is undecided
28:            c ← top(view);
29:            if φ'_i(c[i].val) = ⊤ ∨ call ≠ ⊤ then                    ▷ And you do not have pending task
30:                if ∀a, b ∈ [n] : view[a] ∈ {⊥, ∇} ∨ view[a][b].clock = view[i][b].clock then
31:                    if ∃ survivor set S: S ⊆ {j | c[j] = ∇ ∨ c[j] ≠ c[i].val[j]} then
32:                        if δ'_i((c.val, call)) ≠ ⊥ then
33:                            c[i] ← ⟨c[i].clock + 1, (c.val, call), ⊤⟩; call ← ⊤;
34:                        else
35:                            output δ'_i((c.val, call));
36:                            c ← ∇;
37:                        end if
38:                    end if
39:                end if
40:            else
41:                c[i].ann ← φ'_i(c[i].val);                           ▷ Add an announcement.
42:            end if
43:        end if
44:    end loop
45: end procedure
```

Lemma 6.2.1. *Eventually a process completes a simulated snap operation.*

Proof. A process running Algorithm 14 completes a simulated *snap* operation in a round if it simultaneously satisfies the predicates in lines 29, 30 and 31. Consider for the sake of contradiction that there exists an infinite execution in which from one round, no process completes a simulated *snap* operation. From that round we can group all *undecided non-faulty* processes into two sets: the set A consisting of all processes that satisfy predicate in line 29 and its complement $\overline{A}$.

(I) As a consequence of the proof of Lemma 6.1.6, eventually a set of processes satisfy both the conditions in line 30 and 31. Since by hypothesis no process satisfies all three predicates, there must be a round r from which the set $\overline{A}$ can not be empty anymore.

Consider all different snapshots taken in round $r+1$, we can order them by containment: $view_1 \subset \cdots \subset view_k$. Let Π_{view_1} be the set of all processes who obtained $view_1$ in line 26. Two possibilities emerge with respect to Π_{view_1}:

(1) $\overline{A} \cap \Pi_{view_1} \neq \emptyset$: Let P_i be a process belonging to $\overline{A} \cap \Pi_{view_1}$. Since snapshots are ordered by containment, the *update* performed by P_i in line 7 was read by all processes. As $P_i \in \overline{A}$ then, $\phi'_i(c[i].val)$ is of the form $\langle O_k^{\pi_y}, \lambda \rangle$ where $\pi_y \subset \Pi$. Let r' be a later round ($r' > r+1$) such that $\mu(r') = y$. In round r', two things may happen:

(a) P_i satisfies one of the predicates in lines 10 and 15. In this case, P_i will become a member of the set A.

(b) P_i does not satisfy any of the predicates in lines 10 and 15. Observe that whenever a process satisfies the predicate in line 8, it invariably invokes the task corresponding to the round (see lines 12, 14, 20 and 22). If P_i does not satisfy the predicate in line 10 it is because no process has simulated the task $\phi'_i(c[i].val.data)$. If P_i does not satisfy the predicate in line 15 is because inv is not of the form $\langle O_k^{\pi_y}, \lambda' \rangle$. Moreover, inv cannot be $\top$ because this would imply that some other process, say P_j, who did not satisfy the predicate in line 9 also did not satisfy the predicate in line 19. This contradicts the fact that the announcement made by P_i in line 7 in round $r+1$ was read by all processes. If $\langle O_k^{\pi_y}, \lambda' \rangle \in bag$ this would contradict the fact that P_i did

not satisfy any of the predicates in lines 10 and 15. Thus there must be some other process $P_j \notin A$, which had a pending task $O_{k'}^{\pi_y}$ with $k \neq k'$ who invoked the task $T_{r'}^{\mu(r')}$ and satisfied the predicate in line 15 and in turn satisfying the predicate in line 29 becoming a member of the set A.

(2) $\overline{A} \cap \Pi_{view_1} = \emptyset$: Observe that $|\Pi_{view_1} > 1|$ otherwise the process belonging this set should satisfy the predicate in line 30. Now consider the vector $c_1 = top(view_1)$ obtained by all processes in line 28. None of these processes can end up with the same vector on the next round otherwise they would satisfy the predicate in line 30.

By (a), for each round in which $view_1$ contains a process belonging to the set $\overline{A}$, the cardinality of $\overline{A}$ is eventually decreased. By (I), the set $\overline{A}$ cannot be empty in an infinite execution in which no process completes a simulated snapshot, thus there must be a round, say r' from which the setting explained in (a) stops occurring. Let us continue the analysis upon that round. By a similar argument that was used in Lemma 6.1.5, consider all different vectors resulting from applying the *top* operation in line 28 in round k: $c_1 < \cdots < c_m$, where m is at most $\alpha - 1$, otherwise some process reads its own *update* and it would satisfy the predicates in lines $29, 30$ and 31. This vector has now become forbidden in later rounds as expressed in (c), observe that the only processes that can obtain vector c_1 are those who obtained it in round r' and since all of them satisfy predicate in line 29, they would satisfy the predicate in line 30 as well. In next round no process can obtain vector c_2 and so on, since the number of vectors is finite, eventually, all processes belonging to the set A obtain vector c_m and any of them completes a simulated *snap* operation, which is a contradiction. □

Corollary 6.2.2. *Eventually all non-faulty undecided process simulate enough snapshots and decide.*

Proof. As a direct consequence of the proof of Lemma 6.2.1, eventually a set of non-faulty processes satisfy all three predicates in lines $29, 30$ and 31. Applying the last proof repeatedly, eventually all non-faulty processes complete enough simulated *snap* operations and decide. □

Corollary 6.2.3. *Algorithm 13 is non-blocking.*

Corollary 6.2.4. *All simulated snapshots are fair with respect adversary $\mathcal{A}$.*

Proof. This follows directly from the fact that a simulated snapshot is completed if it includes new values written by a survivor set. See condition of line 31. □

Lemma 6.2.5. *All simulated x-consensus tasks are correct.*

Proof. Recall that a x-consensus object must satisfy three conditions:

- Agreement: Processes running Algorithm 14 collect all responses different from $\top$ from all invocations (see line 24). A process will not generate a new response for a task that was already simulated (see lines 10, 11 and 19). In addition, in lines 14 and 20 processes use a $x-$consensus task to get a response so there is no risk to get duplicate responses.

- Validity: If some process got a response v for its task in line 11, then some other process say P_j proposed v. It might be the case that P_j executed line 14 which means that v was proposed by P_j, otherwise P_j executed line 20 which means that v was proposed by some other process for which the task was defined, see line 19.

- Termination: By Corollary 6.2.2, eventually all correct processes complete their respective simulated *invoke* operations.

□

Lemma 6.2.6. *An execution of Algorithm 14 encodes a valid execution of the Snapshot model extended with x-consensus tasks within the IS model extended with x-consensus tasks.*

Proof. If we remove the condition of line 8 along with its inner block of code, remove the condition of line 29 leaving its inner block of code from Algorithm 14, we obtain Algorithm 13 which by Lemma 6.1.8 encodes a valid execution of the Snapshot model:

$$C_0, U^{r_1}, C_1, S^{r_1}, C_2, \ldots, U^{r_k}, C_{2k-1}, S^{r_k}, C_{2k},$$

Moreover, when the simulated protocol $\mathcal{P}$ does not include invocations, Algorithm 14 behaves exactly the same as Algorithm 13 except for the extra information added in vector c and the invocations performed within the block in the condition of line 8. Consider the moment in which a process P_i running Algorithm 14 does not satisfy the condition in line 29. What follows

is that P_i adds an announcement in line 41 and continues the execution normally. From this point, consider the next time P_i satisfies the condition of line 29. In that precise moment, P_i linearizes its last simulated *snap* and just before this last *snap*, in the same round, P_i completes its simulated *invoke* operation in the block corresponding to the condition of line 8. Since this invocation is performed after an *update*, its linearized invocation is then linearized between a linearized *update* and a linearized *snap* operation.

□

We are now in conditions to state the next Theorem:

Theorem 6.2.7. *Algorithm 14 correctly simulates a Snapshot protocol extended with x-consensus tasks in the IS model extended with x-consensus tasks.*

Proof. By Corollary 6.2.2 all processes eventually decide and all simulated snapshots are fair with respect adversary $\mathcal{A}$ (see Corollary 6.2.4). And by Lemma 6.2.5 all simulated tasks are correct. In addition by Lemma 6.2.6 an execution of Algorithm 14 produces a correct linearized execution of the Snapshot model extended with x-consensus tasks. □

Corollary 6.2.8. *The Snapshot model extended with x-consensus tasks and the IS model extended with x-consensus tasks are equivalents when solving decision tasks.*

Chapter 7

Discussion

We have considered two asynchronous shared memory models, namely the Snapshot model and its iterated counterpart the IS model. We have shown how these models can be extended to work with additional communication objects. Using this extension, we were able to prove that the Snapshot model and the IS model are equivalent under many circumstances when they are used to solve decision tasks. Our approach was the use of the simulation technique. Our main result was to complete the description of the simulation presented in [20] and the construction of a simulation from the Snapshot model extended with x-consensus objects to its iterated counterpart based on the simulation informally presented in [10]. This simulation proves that we can always unravel the invocations of any Snapshot protocol extended with x-consensus objects obtaining an algorithm in the extended Snapshot model with x-consensus objects that invokes all objects in the same order.

The main difficulty of the algorithm shown in [20] is that it needs the notion of the dual of a task. An open interesting problem is a characterization of tasks which have a dual definition so we can find simulations similar to [20]. On the other hand, another open interesting problem is whether we can use a similar technique as that of Algorithm 14 to simulate protocols in the Snapshot model extended with k-set consensus objects into its iterated counterpart.

In [19], Gafni and Koutsoupias showed that no algorithm exists for deciding whether a bounded task for three or more processes is wait-free solvable in the Shared Memory model. Later on, Imbs and Raynal in [28] showed that for a special class of decision tasks known as *colorless tasks* (i.e. tasks such that if a process starts with an input value, any other process

may start with the same value and also if a process decides a value, any other process is allowed to decide the same value) the Shared Memory model with N_1 processes and at most t_1 failures is equivalent to the Shared Memory model with N_2 processes and at most t_2 failures and extended with x-consensus objects if and only if $t_1 = \lfloor \frac{t_2}{x} \rfloor$. Thus, if $\lfloor \frac{t_2}{x} \rfloor \geq 2$ the result by Gafni and Koutsoupias holds. With the equivalence under x-consensus objects presented in this thesis, I showed that this undecidability result applies to the IS model as well.

Bibliography

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

[2] James H. Anderson. Composite registers. *Distrib. Comput.*, 6(3):141–154, 1993.

[3] James H. Anderson and Mohamed G. Gouda. A criterion for atomicity. *Formal Aspects of Computing*, 4(3):273–298, May 1992.

[4] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, Jan 1995.

[5] Hagit Attiya and Armando Castañeda. A non-topological proof for the impossibility of k-set agreement. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 108–119. Springer Berlin Heidelberg, 2011.

[6] Bard Bloom. Constructing two-writer atomic registers. *IEEE Transactions on Computers*, 37(12):1506–1514, Dec 1988.

[7] E. Borowsky, E. Gafni, N. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. *Distrib. Comput.*, 14:127–146, October 2001.

[8] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *PODC '93: Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, pages 41–51, New York, NY, USA, 1993. ACM.

[9] Elizabeth Borowsky and Eli Gafni. The Implication of the Borowsky-Gafni Simulation on the Set-Consensus Hierarchy. Technical report, UCLA, 1993.

[10] Elizabeth Borowsky and Eli Gafni. A Simple Algorithmically Reasoned Characterization of Wait-Free Computations (Extended Abstract). In *PODC '97: Proceedings of the 16th annual ACM symposium on Principles of distributed computing*, pages 189–198, New York, NY, USA, Aug 1997. ACM.

[11] James E. Burns and Gary L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *PODC '87: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 222–231, Vancouver, British Columbia, Canada, Aug 1987. ACM.

[12] Armando Castañeda, Damien Imbs, Sergio Rajsbaum, and Michel Raynal. Renaming is weaker than set agreement but for perfect renaming: A map of sub-consensus tasks. In David Fernández-Baca, editor, *LATIN 2012: Proc. 10th Latin American Symposium Theoretical Informatics*, volume 7256 of *Lecture Notes in Computer Science*, pages 145–156. Springer Berlin Heidelberg, 2012.

[13] Soma Chaudhuri. Agreement Is Harder Than Consensus: Set Consensus Problems In Totally Asynchronous Systems. In *PODC '90: Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 311–324, New York, NY, USA, 1990. ACM.

[14] Soma Chaudhuri, Maurice Herlihy, Nancy A. Lynch, and Mark R. Tuttle. A Tight Lower Bound for k-Set Agreement. In *In Proceedings of the 34th IEEE Symposium on Foundations of Computer Science*, pages 206–215, 1993.

[15] Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2(3):155 – 173, 1982.

[16] M. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr 1985.

[17] Greg N. Frederickson and Nancy A. Lynch. Electing a leader in a synchronous ring. *J. ACM*, 34(1):98–115, January 1987.

[18] Eli Gafni. The 0—1-exclusion families of tasks. In Springer-Verlag, editor, *OPODIS '08: Proceedings of the 12th International Conference on Principles of Distributed Systems*, pages 246 – 258, Luxor, Egypt, Dec 2008. Springer Berlin Heidelberg.

[19] Eli Gafni and Elias Koutsoupias. Three-Processor Tasks Are Undecidable. *SIAM J. Comput.*, 28(3):970–983, 1999.

[20] Eli Gafni and Sergio Rajsbaum. Distributed programming with tasks. In *Proceedings of the 14th international conference on Principles of distributed systems*, OPODIS'10, pages 205–218, Berlin, Heidelberg, 2010. Springer-Verlag.

[21] Eli Gafni, Sergio Rajsbaum, and Maurice Herlihy. Subconsensus Tasks: Renaming Is Weaker Than Set Agreement. In *Distributed Computing, 20th International Symposium, Stockholm, Sweden, September 18-20, 2006, Proceedings(DISC)*, volume 4167 of *Lecture Notes in Computer Science*, pages 329–338. Springer, 2006.

[22] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[23] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[24] Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. Combinatorial topology and distributed computing. feb 2011.

[25] Maurice Herlihy and Sergio Rajsbaum. The topology of shared-memory adversaries. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 105–113, New York, NY, USA, 2010. ACM.

[26] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999.

[27] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[28] Damien Imbs and Michel Raynal. The multiplicative power of consensus numbers. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 26–35, New York, NY, USA, 2010. ACM.

[29] Amos Israeli and Ming Li. Bounded time-stamps (extended abstract). In *FOCS '87: Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, pages 371–382, Los Angeles, California, USA, Oct 1987. IEEE Computer Society.

[30] Flavio P. Junqueira and Keith Marzullo. Designing Algorithms for Dependent Process Failures. Technical report, 2003.

[31] Lefteris M. Kirousis, Evangelos Kranakis, and Paul M. B. Vitányi. Atomic multireader register. In *IWDC '87: Proceedings of the 2nd International Workshop on Distributed computing*, volume 312 of *Lecture Notes in Computer Science*, pages 278–296, Amsterdam,The Netherlands, Jul 1987. Springer Verlag.

[32] Leslie Lamport. On interprocess communication. part i: Basic formalism. *Distributed Computing*, 1(2):77–85, Jun 1986.

[33] Leslie Lamport. On interprocess communication. part ii: Algorithms. *Distributed Computing*, 1(2):86–101, Jun 1986.

[34] Ming Li and Paul M. B. Vitányi. How to share concurrent asynchronous wait-free varaibles (preliminary version). In *ICALP 89': Proceedings of the 16th International Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 488–505, Stresa, Italy, Jul 1989. Springer.

[35] Richard E. Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *PODC '87: Proceedings of the 6th Annual Symposium on Principles of Distributed Computing*, pages 232–248, Vancouver, British Columbia, Canada, Aug 1987. ACM.

[36] Gary L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, Jan 1983.

[37] Gary L. Peterson and James E. Burns. Concurrent reading while writing ii: The multi-writer case. In *FOCS '87: 28th Annual Symposium on Foundations of Computer Science*, pages 383–392, Los Angeles, California, USA, Oct 1987. IEEE Computer Society.

[38] Sergio Rajsbaum. Iterated shared memory models. In Alejandro López-Ortiz, editor, *LATIN 2010: Theoretical Informatics*, volume 6034 of *Lecture Notes in Computer Science*, pages 407–416. Springer Berlin / Heidelberg, 2010.

[39] Sergio Rajsbaum, Michel Raynal, and Corentin Travers. The Iterated Restricted Immediate Snapshot Model. In *Computing and Combinatorics, 14th Annual International Conference, COCOON 2008, Dalian, China, June 27-29, 2008, Proceedings*, volume 5092 of *Lecture Notes in Computer Science*, pages 487–497. Springer, 2008.

[40] Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. Comput.*, 29(5):1449–1483, Jul 2000.

[41] Ambuj K. Singh, James H. Anderson, and Mohamed G. Gouda. The elusive atomic register revisted. In *PODC '87: Proceeding of the 6th Annual Symposium on Principles of Distributed Computing*, pages 206–221, Vancouver, British Columbia, Canada, Aug 1987. ACM.

[42] John Tromp. How to construct an atomic variable (extended abstract). In *WDAG '89: Proceedings of the 3rd International Workshop on Distributed Algorithms*, volume 392 of *Lecture Notes in Computer Science*, pages 292–302, Nice, France, Sep 1989. Springer-Verlag.

[43] Paul M. B. Vitányi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware (detailed abstract). In *FOCS '86: Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 233–243, Toronto, Canada, Oct 1986. IEEE Computer Society.