

## The Iterated Restricted Immediate Snapshot Model\*

Corentin Travers\*\* Sergio Rajsbaum\*\*\* Michel Raynal\*\*\*\*

**Abstract:** In the *Iterated Immediate Snapshot* model (*IIS*) the memory consists of a sequence of one-shot *Immediate Snapshot* (*IS*) objects. Processes access the sequence of *IS* objects, one-by-one, asynchronously, in a *wait-free* manner; any number of processes can crash. Although more restricted (each *IS* object can be accessed only once), the *IIS* model is equivalent to the read/write model for wait-free solvability of decision tasks. Its interest lies in the elegant recursive structure of its runs, which facilitates its analysis, round by round.

Although there are by now quite a few papers that use the *IIS* model or its variants, the approach has not yet been used to study failure detectors. The paper shows that an elegant way of capturing the power of a failure detector and other partially synchronous systems is by considering appropriate subsets of runs of the *IIS* model, giving rise to the *Iterated Restricted Immediate Snapshot* model (*IRIS*).

The proposed approach has several benefits. First it provides us with new simulations in presence of asynchrony and failures. Then, it gives new insights on the very nature of failure detectors, and on how to represent them in an iterated model. Finally, it allows designing simpler proofs of existing results. As a study case, the paper considers a system enriched with a *limited-scope accuracy* failure detector, where there is a cluster of processes such that some correct process is eventually never suspected by any process in that cluster. A new proof of the  $k$ -set agreement Herlihy and Peaso's lower bound for shared memory system augmented with a limited-scope accuracy failure detector is provided. The proof is based on an extension of the Borowsky-Gafni *IIS* simulation to encompass failure detectors, followed by a very simple topological argumentation. The paper describes similar applications for other failure detectors including the classes  $\Omega^z$  and  $\diamond\psi^y$ .

**Key-words:** Algorithmic reduction, Asynchronous system, Distributed algorithm, Distributed Computability, Failure detectors, Fault-tolerance, Round-based computation, Shared memory, Topology.

---

### *Un modèle de calcul réparti itéré généralisé*

**Résumé :** *Ce rapport présente un modèle de calcul réparti itéré généralisé.*

**Mots clés :** *Calcul réparti, modèle, réduction.*

---

\* Earlier versions of this paper appeared in [44, 45, 46]

\*\* LaBRI, Université de Bordeaux 1, France

\*\*\* Instituto de Mathematicas, UNAM, D.F. 04510, Mexico

\*\*\*\* Institut Universitaire de France et IRISA (équipe commune à l'Université de Rennes 1 et Inria)

# 1 Introduction

A distributed model of computation consists of a set of  $n$  processes communicating through some medium (some form of message passing or shared memory), satisfying specific timing assumptions (process speeds and communication delays), and failure assumptions (their number and severity). A major obstacle in the development of a theory of distributed computing is the wide variety of models that can be defined – many of which represent real systems – with combinations of parameters in both the (a)synchrony and failure dimensions [6, 36]. Thus, an important line of research is concerned with finding ways of unifying results, impossibility techniques, and algorithm design paradigms of different models.

An early approach towards this goal has been to derive direct simulations from one model to another, e.g., [2, 5, 8, 10]: to show how to transform a protocol running in an asynchronous message passing model to one for a shared memory model [2], or from an asynchronous model to a synchronous model [5], or from a protocol tolerating some number of failures to one tolerating more failures [10] or more severe ones. A more recent approach has been to devise models of a higher level of abstraction, where results about various more specific models can be derived (e.g., [19, 30, 37]). Two main ideas are at the heart of the approach, which has been studied mainly for crash failures only, and is the topic of this paper.

**Two bedrocks: wait-freedom and round-based execution** It has been discovered [8, 31, 51] that the *wait-free* case, where any number of processes can crash (“wait statements” to hear from another process are useless) is fundamental. In particular, [31] provided a characterization of the tasks that are wait-free solvable in a read/write shared memory system. One can derive characterizations of task solvability in other models, by reduction (via simulations e.g., [10, 20, 33, 34]) to the wait-free model and then applying the characterization of [31].

The wait-free characterization of [31] is topological in nature, and it is based on a representation of the executions of a protocol as a *simplicial complex*, i.e., a discrete geometric object, whose interesting properties are invariant over continuous deformations, namely, subdivisions. In more detail, one considers the *simplicial complex of global states* of the system after a finite number of steps. Various papers have analyzed topological invariants about the structure of such a complex, for wait-free and other models, to derive impossibility results, and sometimes also protocols. Such invariants are based on the notion of *indistinguishability*, which has played a fundamental role in nearly every lower bound in distributed computing. Two global states are indistinguishable to a set of processes if they have the same local states in both.

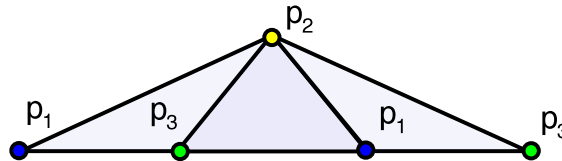


Figure 1: A simple complex with three simplexes

As an example let us consider Figure 1 that represents a complex with three triangles. Each triangle is a *simplex* representing a global state. The corners of a simplex represent local states of processes in the global state. The center simplex and the rightmost simplex represent global states that are indistinguishable to  $p_1$  and  $p_2$ , which is why the two triangles share an edge. Only  $p_3$  can distinguish between the two global states.

Most attempts at unifying models of various degrees of asynchrony restrict attention to a subset of well-behaved, *round-based* executions. Given a model of distributed computation, one considers subsets of executions, generated by particular legal sequences of actions for the scheduler, each of which produces a layer. Thus, in a precise sense, such a layering can be viewed as defining a sub-model of the original model. Lower bounds and impossibility results proven for the sub-model translate directly into the original model. For example, [37] presents a uniform approach to the study of solvability of consensus in various models of computation in which, crash failure behavior can occur. The use of layerings facilitates performing round-by-round analysis: essentially, results regarding consensus follow from analyzing a single layer of computation.

The approach in [9] goes beyond and defines an *iterated* round-based model (*IIS*), where each communication object can be accessed only *once* by each process. In its basic form, the iterated model assumes the objects are *Immediate Snapshot (IS)* objects [7], that are accessed by the processes with a single operation denoted `write_snapshot()`, that writes the value provided by the invoking process and returns to it a snapshot [1] of its content. A benefit of using immediate snapshot operations is that the resulting complex is a *manifold*: as in the figure above, where for three processes, each edge is contained in at most two triangles. The sequence of *IS* objects are accessed asynchronously, and one after the other by each process. It is shown in [9] that the *IIS* model is equivalent (for bounded wait-free task solvability) to the usual read/write shared memory model. A simpler and more general simulation appeared recently [23].

Thus, the runs of the *IIS* model are not a subset of the runs of a standard (non-iterated, where a process can access the same object more than once) model as in other works, and the price that has to be payed is a simulation algorithm showing that the model is equivalent to a read/write shared memory model (w.r.t. wait-free task solvability). But the reward is a model that has an elegant recursive structure: the complex of global states after  $i + 1$  rounds is obtained by replacing each simplex in the complex of global states after  $i$  rounds, by a one round complex (see Figure 2). Thus, as in [19, 30, 37] impossibility results follow from analyzing a single layer of computation, but in the *IIS* the layers are by definition independent. Furthermore, the design of algorithms is also facilitated. Actually, roughly speaking, the *IIS* is the model resulting from programming distributed algorithms in a recursive manner [22]. Indeed, the *IIS* model was the basis for the proof in [9] of the wait-free characterization theorem of [31] that holds for any task. Also, the *IIS* model, enriched with objects more powerful than read/write registers, was instrumental for the results in [24] showing that renaming is a strictly weaker task than set agreement. Later on it was shown that this enrichment is equivalent to its non-iterated version [23]. See [43] for an overview of results related to the *IIS* model, and more recent papers that take advantage of the *IIS* model and its variants, such as [28, 29].

**Failure detectors** Although there are by now quite a few papers that use the *IIS* model or its variants, the approach has not yet been used to study failure detectors. Recall that a *failure detector* [12] is a distributed oracle that provides each process with hints on process failures (see [49, 50] for an introduction to failure detectors). According to the type and the quality of the hints, several classes of failure detectors have been defined (e.g., [12, 38, 54]). Failure detectors are used as an abstraction of reliability assumptions, to design modular protocols in distributed systems, and also as a theoretical device, to study models of various degrees of synchrony.

The family of *limited scope* accuracy failure detectors, is denoted  $\diamond\mathcal{S}_x$  [27, 53]. These capture the idea that a process may detect failures reliably on the same local-area network, but less reliably over a wide-area network. They are a generalization of the class denoted  $\diamond\mathcal{S}$  that has been introduced in [12] ( $\diamond\mathcal{S}_n$  is  $\diamond\mathcal{S}$ ). Informally, a failure detector  $\diamond\mathcal{S}_x$  ensures that there is a non-faulty process that is eventually never erroneously suspected by any process in a cluster of  $x$  processes. A failure detector of the class  $\diamond\mathcal{S}_x$  is for a system made up of a single cluster of processes. The family  $(\diamond\mathcal{S}_{x,q})_{1 \leq x \leq n, 1 \leq q \leq x}$  extends the notion of limited scope failure detector to a system where the processes are partitioned into multiple disjoint clusters. There are  $q$  disjoint clusters denoted  $X_1, \dots, X_q$ , where  $|X_i| = x_i$ ,  $X = \bigcup_{1 \leq i \leq q} X_i$  and  $x = \sum_{i=1}^q x_i$ . Informally, there is a process that is never suspected in each cluster  $X_i$ . Thus, as the parameters  $x, q$  vary, systems of different degree of synchrony are obtained.

Many other families of failure detectors have been considered. Notably,  $\{\Omega^z\}_{1 \leq z \leq n}$ , and  $\{\diamond\psi^y\}_{1 \leq y \leq n}$ . The failure detector class  $\Omega^z$  [42] is a generalization of the class  $\Omega$  [13]; in particular,  $\Omega^1$  is the class  $\Omega$ , that is necessary and sufficient to solve consensus. A failure detector of the class  $\Omega^z$  controls a local variable  $\text{LEADER}_i$  containing a set process identities, and captures weaker synchrony assumptions. A failure detector of the class  $\diamond\psi^y$  outputs at each process  $p_i$  an integer  $\text{NBC}_i$  that is an estimate of the number of processes that have crashed. The family  $\{\diamond\psi^y\}_{1 \leq y \leq n}$  was introduced in [39] (although with a different formulation).

**Context and goals of the paper** The paper introduces the *IRIS model*, which consists of a subset of runs of the *IIS* model of [9]. The aim is to obtain the benefits of the round by round and wait-freedom approaches in one model, where any number of processes can fail (by crashing), but the executions represent those of a partially synchronous model. The proposed approach has several benefits. First it provides us with new simulations in presence of asynchrony and failures. Then, it gives new insights on the very nature of failure detectors, and on how to represent them in an iterated model. Finally, it allows designing simpler proofs of existing results.

In the construction of a distributed computing theory, a central question has been understanding how the degree of synchrony of a system affects its power to solve distributed tasks. The degree of synchrony has been expressed in various ways, typically either by specifying a bound  $t$  on the number of processes that can crash, as bounds on delays and process steps [16], or by a failure detector [12], or by using powerful shared memory objects [26]. It has been shown multiple times that systems with more synchrony can solve more tasks. Previous works in this direction have mainly considered an asynchronous system enriched with a failure detector that can solve consensus. Some works have identified this type of synchrony in terms of fairness properties [52]. Other works have considered round-based models with no failure detectors [19]. Some other works [35] focused on performance issues mainly about consensus. Also, in some cases, the least amount of synchrony required to solve some task has been identified, within some paradigm. A notable example is the weakest failure detector to solve consensus [13] or set agreement [54].  $k$ -set agreement [14] (see [48] for a short survey) represents a desired coordination degree to be achieved in the system, requiring processes to agree on at most  $k$  different values (consensus is 1-set agreement), and hence is natural to use it as a measure for the *synchrony degree* in the system. The fundamental result of the area is that  $k$ -set agreement is not solvable in a wait-free, i.e., fully asynchronous system even for  $k = n - 1$  [8, 31, 51].

However, a clear view of what exactly “degree of synchrony” means is still lacking. For example, the same power as far as solving  $k$ -set agreement can be achieved in various ways, such as via failure detectors [38] or  $t$ -resilience assumptions. A second goal for introducing the *IRIS* model, is to have a mean of precisely representing the degree of synchrony of a system, and this is

achieved with the *IRIS* model by considering particular subsets of runs of the *IIS* model. We observe in [47] that directly including failure detectors in the *IIS* model is useless, instead we consider subsets of runs to model partial synchrony.

Our representation of synchrony complements previous results about  $t$ -resilient systems, derived by reduction to the wait-free case [10], or using bivalency arguments (e.g., [17, 37]) which do not seem to be generalizable from consensus to set agreement. The 1-resilient characterization of [11] is by reduction to the consensus impossibility of [17], and in general dealing with  $t$ -resilient executions is more difficult than the wait-free case; compare for example the wait-free consensus impossibility proof in [26] with the one of [17], which is much more subtle.

**Contributions** This paper shows that the *IIS* model has yet another fundamental advantage, namely, it allows studying the computability power of the read/write shared memory model equipped with a failure detector, when any number of processes can crash. More specifically, the paper presents several results in that direction.

1. Given that directly adding a failure detector to the *IIS* model does not allow solving more tasks [47], an iterated model is defined by a subset of its executions. For a failure detector of a class  $C$ , a corresponding restricted *IIS* model is defined. This model is denoted  $IRIS(PR_C)$ . *IRIS* stands for *Iterated Restricted Immediate Snapshot* model.  $PR_C$  denotes a property, derived from the failure detector class  $C$ , that is encapsulated in the `write_snapshot()` operation. The  $IRIS(PR_C)$  model is induced by the runs in which the `write_snapshot()` operations satisfy the corresponding  $PR_C$  property. Every run of  $IRIS(PR_C)$  is a run of the *IIS* model, but the opposite is not necessarily true.

To illustrate the approach, the paper considers three families of failure detector classes:  $\{\diamond S_x\}_{1 \leq x \leq n}$ ,  $\{\Omega^z\}_{1 \leq z \leq n}$ , and  $\{\diamond \psi^y\}_{1 \leq y \leq n}$ . For a failure detector  $C$  in each one, it defines a corresponding  $IRIS(PR_C)$  model.

2. The paper shows that the synchrony exhibited by the  $IRIS(PR_C)$  model characterizes the power of the read/write model enriched with  $C$ . It presents a simulation from the shared memory model with  $C$  to the  $IRIS(PR_C)$  model. Conversely, it shows how to extract  $C$  from  $IRIS(PR_C)$ , and then simulate the read/write model with  $C$ . A noteworthy corollary follows from that simulation, namely, a task is solvable in the read/write model with  $C$  if and only if it is solvable in the  $IRIS(PR_C)$  model. Thus, the paper shows that the simulation of [23] (an improvement on the original one in [9]) of the read/write model in the *IIS* model, can be extended to work also with failure detectors.
3. As an application of the previous simulations, new, simple proofs of the impossibility of solving  $k$ -set agreement in the read/write model equipped with a failure detector from the above classes are derived. Such direct proofs were known only for the  $\{\diamond S_x\}_{1 \leq x \leq n}$  family [27], using combinatorial topology techniques from [30]. Impossibility proofs for the other families are by reduction to this result [38].

Conversely, the results presented in the paper open the possibility of designing new set agreement (and in particular consensus) algorithms: design an algorithm in an  $IRIS(PR_C)$  model, and then using the simulation mentioned above, transform it into an algorithm for the read/write model with  $C$ .

We remark that the definition of an  $IRIS(PR_C)$  model is not in terms of process failures or failure detectors. The characterization of a failure detector class  $C$  appears as a restriction of the set of runs that would be produced if the corresponding failure detector was used in a certain canonical way and the schedules of read and write operations follow a certain form. So, the  $IRIS(PR_C)$  model captures the synchronization/scheduling power of the corresponding failure detector class. In that sense, a failure detector is a scheduler with specific fairness properties<sup>1</sup>.

**Roadmap** The paper is divided in 8 sections. Section 2 describes the computational model and the classes of failure detectors we are interested in. Iterated restricted models corresponding to the failure detector classes  $\diamond S_x$ ,  $\Omega^z$  and  $\diamond \psi^y$  are presented in 3. The computational equivalence of these models with the standard read/write model enriched with the corresponding failure detectors is proved in sections 4,5 and 6. Section 7 presents simple proofs of impossibility results regarding the computational power of the read/write model augmented with failure detectors. This section also shows that the *IRIS* model can be used to analyze the  $t$ -resilient model. Finally, section 8 concludes the paper.

## 2 Computation model and failure detector classes

This section presents a quick overview of the background needed for the rest of the paper, more detailed descriptions can be found elsewhere, e.g., [6, 9, 12]. We describe here the two main models we are concerned with, in Section 2.1 the standard shared memory

<sup>1</sup>This is similar to the *linearizability* consistency criterion [32] that restricts the set of runs generated by processes that access concurrently shared objects.

model enriched with failure detector, and in Section 2.2 the *IIS* model. In Section 2.3 we define tasks, and the known equivalence between these models.

## 2.1 Shared memory model enriched with failure detectors

**Asynchronous shared memory** The paper considers a standard asynchronous system made up of  $n$  processes,  $p_1, \dots, p_n$ . A process can fail by *crashing*, i.e., by prematurely halting. It behaves correctly (i.e., according to its specification) until it (possibly) crashes. A process is *correct in a run* if it takes an infinite number of steps and *faulty* otherwise. If not otherwise indicated, we do not assume any upper bound on the number of faulty processes. In the case where no failure detector is available, this is called the *wait-free* environment, because “wait statements” used by a process to hear from another process are useless. A system where any number of processes may crash is sometimes called wait-free even if a failure detector is available, despite the fact that wait statements may be useful in this case.

The shared memory is structured as an array  $SM[1..n]$  of atomic registers. Each register  $SM[i]$  supports two operations  $write(v)$  and  $read()$  that allow to store the value  $v$  and retrieve the current value of the register respectively. Only  $p_i$  can write to  $SM[i]$ , but every process  $p_j$  can read  $SM[i]$ . Uppercase letters are used to denote shared registers. It is often useful to consider higher level abstractions constructed out of such registers, that are implementable on top of them, such as snapshots objects. In this case, a process can read the entire memory  $SM[1..n]$  in a single atomic operation, denoted  $snapshot()$  [1].

**Failure detectors** As explained in the Introduction, a failure detector [12] is a distributed oracle that provides possibly unreliable information about failures to the processes. Operationally, each process  $p_i$  is endowed with a read-only variable  $FD_i$  that contains the information provided by the failure detector. Several classes of failure detectors can be defined according to the kind and the quality of failures information they provide.

**The family  $(\diamond S_x)_{1 \leq x \leq n}$**  A failure detector of the class  $\diamond S_x$  [25, 40, 27] provides each process  $p_i$  with a variable  $TRUSTED_i$  that contains identities of processes that are believed to be currently alive. When  $j \in TRUSTED_i$  we say “ $p_i$  trusts  $p_j$ ”. The class  $\diamond S_x$  is a simple generalization of the class  $\diamond S$  introduced in [12] (in particular,  $\diamond S_n$  is  $\diamond S^2$ .)

By convention, a crashed process trusts all processes. The failure detector class  $\diamond S_x$  is defined by the following properties:

- *Strong completeness.* There is a time after which every faulty process is never trusted by every correct process and,
- *Limited scope eventual weak accuracy.* There is a set  $Q$  of  $x$  processes containing a correct process  $p_\ell$ , and a (finite) time after which each process of  $Q$  trusts  $p_\ell$ .

The time  $\tau$ , the set  $Q$  and the process  $p_\ell$  are not known by the processes. Moreover, some processes of  $Q$  could have crashed. The parameter  $x$ ,  $1 \leq x \leq n$ , defines the scope of the eventual accuracy property. When  $x = 1$ , the failure detector provides no information on failures, when  $x = n$  the failure detector can be used to solve consensus [12]. In a system where no more than  $t$  processes may crash, all the classes  $\diamond S_x$ ,  $t < x \leq n$ , are equivalent [4].

It is sometimes convenient to use the following equivalent formulation of  $\diamond S_x$  [38]. Assuming the local variable controlled by the failure detector is  $REPR_i$ :

- *Limited eventual common representative.* There is a set  $Q$  of  $x$  processes containing a correct process  $p_\ell$ , and a (finite) time after which, for any correct process  $p_i$ , we have  $i \in Q \implies REPR_i = \ell$  and  $i \notin Q \implies REPR_i = i$ .

Clearly, a failure detector that satisfies the previous property can be transformed into one of the class  $\diamond S_x$  (define  $TRUSTED_i = \{REPR_i\}$ ). Conversely, an algorithm that transforms any failure detector of  $\diamond S_x$  into a failure detector satisfying the limited eventual common representative property is described in [38].

**The family  $(\diamond \psi^y)_{1 \leq y \leq n}$**  A failure detector of the class  $\diamond \psi^y$  outputs at each process  $p_i$  an integer  $NBC_i$  that is an estimate of the number of processes that have crashed. The class  $\diamond \psi^y$  is defined by the following property, where  $f$  is the number of actual crashes in a run.

- *Eventual accuracy.* There is a time from which  $NBC_i = \max(n - y, f)$  at each correct process  $p_i$ .

<sup>2</sup>The original definition of the failure detector calls  $\diamond S$  [12] provides each process  $p_i$  with a set denoted  $SUSPECTED_i$ . Using the set  $TRUSTED_i$  is equivalent to using the set  $SUSPECTED_i$ . (more precisely,  $TRUSTED_i = \Pi \setminus SUSPECTED_i$ ). We use  $TRUSTED_i$  to emphasize the fact that what is important to ensure progress is the set of processes that are alive.



The family  $\{\diamond\psi^y\}_{1 \leq y \leq n}$  was introduced in [39] although with a different formulation<sup>3</sup>. It is shown in [38] that  $\diamond\psi^n$  is equivalent to  $\diamond\mathcal{P}$ , the class of eventually perfect failure detectors [12] (a failure detector of that class is strictly stronger than  $\diamond\mathcal{S}_n$ ), while  $\diamond\psi^1$  provides no information on failures. A failure detector of that class provides each process with a set, denoted  $\text{TRUSTED}_i$ , such that after an arbitrary but finite time, the set of any correct process contains all the correct processes and only them.

**The family  $(\Omega^z)_{1 \leq z \leq n}$**  The failure detector class  $\Omega^z$  [42] is a generalization of the class  $\Omega$  [13]; in particular,  $\Omega^1$  is the class  $\Omega$ . A failure detector of the class  $\Omega^z$  controls a local variable  $\text{LEADER}_i$  containing a set process identities, and satisfies the following property :

- *Eventual leadership.* There is a set  $L$ , of size at most  $z$  and containing a correct process, and a (finite) time after which the set  $\text{LEADER}_i$  of every correct process  $p_i$  remains forever equal to  $L$ .

Let us notice that when  $z = n$  a failure detector of the class  $\Omega^z$  provides no information on failures; when  $z = 1$ ,  $\Omega^z$  is equivalent to  $\diamond\mathcal{S} = \diamond\mathcal{S}_n$  [13], and hence powerful enough to solve consensus. However, as shown in [38], while it is possible to build a failure detector of the class  $\Omega^z$  from a failure detector of the class  $\diamond\mathcal{S}_x$  iff  $x + z > t + 1$  (where  $t$  is an upper bound on the number of processes that may crash), it is impossible to build a failure detector of the class  $\diamond\mathcal{S}_x$  from a failure detector of the class  $\Omega^z$  for  $1 < x, z \leq t$ . On another side, while  $\diamond\psi^y$  can be transformed into  $\Omega^z$  iff  $y + z > t$ ,  $\Omega^z$  cannot be transformed into  $\diamond\psi^y$  [38].

Equivalently, the class  $\Omega^z$  can be defined by the following property:

- *Weak eventual leadership.* There is a set  $L$ , of size at most  $z$  and containing a correct process, and a time  $\tau$  such that for every correct process  $p_i$  and every time  $\tau' \geq \tau$ ,  $\text{LEADER}_i^{\tau'} \subseteq L$  and  $\text{LEADER}_i^{\tau'} \cap \text{Correct} \neq \emptyset$  where  $\text{LEADER}_i^{\tau'}$  is the output of the failure detector at time  $\tau'$  at process  $p_i$  and  $\text{Correct}$  is the set of correct process.

Clearly, the weak eventual leadership property is implied by the eventual leadership property. Conversely, a failure detector that satisfies the eventual leadership property can be emulated when a failure detector with the weak eventual leadership property is available. Initially, an array  $\mathcal{L}[0..m-1] = [L_0, \dots, L_{m-1}]$  made of all possible sets of size  $z$  is shared by the processes ( $m = \binom{n}{z}$ ). Each process  $p_i$  maintains a counter  $cn_i$  initially equal to 0. At any time, and at each process  $p_i$ , the output of the emulated failure detector is defined as  $\mathcal{L}[cn_i \bmod m]$ . Each process  $p_i$  periodically checks whether  $\mathcal{L}[cn_i \bmod m]$  is contained in the current set output by the failure detector. If this is not true,  $p_i$  increments its counter  $cn_i$  and writes the new value in its shared register  $SM[i]$ . Each process also periodically reads every register and set its counter to the largest value it sees.

By the weak eventual leadership property, there exists a set  $L$  such that eventually the output of the failure detector is always a subset of  $L$ . This property implies that the counters  $cn_i$  are bounded. Moreover, the fact that each process  $p_i$  periodically writes  $cn_i$  in shared memory and updates it with the largest value its sees implies that all counters converge to the same value  $x$ . Therefore, the output of the emulated failure detector is eventually always equal to  $\mathcal{L}[x \bmod m] = L'$ . By definition,  $|L'| = z$  and it contains a correct process (Otherwise, the counters cannot converge to the value  $x$ ). The emulated outputs thus satisfy the eventual leadership property.

## 2.2 The Iterated immediate snapshot (IIS) model

A *one-shot immediate snapshot* object  $IS$  is accessed with a single operation denoted  $\text{write\_snapshot}()$ . That operation replaces both the  $\text{write}()$  and the  $\text{snapshot}()$  operations on the shared memory  $SM[1..n]$ . Intuitively, when a process  $p_i$  invokes  $\text{write\_snapshot}(v)$  it is as if it instantaneously executes write operation  $IS[i].\text{write}(v)$  immediately followed by a snapshot  $IS.\text{snapshot}()$ . If several  $IS.\text{write\_snapshot}()$  operations are executed simultaneously, then their corresponding writes are executed concurrently, and then their corresponding snapshots are executed concurrently (each of the concurrent operations sees the values written by the other concurrent operations): they are set-linearizable [41].

The semantics of the  $\text{write\_snapshot}()$  operation is characterized by the three following properties, where  $v_i$  is the value written by  $p_i$  and  $sm_i$ , the *immediate snapshot* or *view* it gets back from the operation, for each  $p_i$  invoking the operation. A view  $sm_i$  is a set of pairs  $\langle k, v_k \rangle$ , where  $v_k$  corresponds to the value in  $p_k$ 's entry of the array  $SM$ . If  $SM[k] = \perp$ , the pair  $\langle k, \perp \rangle$  is not placed in  $sm_i$ . Moreover, we have  $sm_i = \emptyset$ , if the process  $p_i$  never invokes  $\text{write\_snapshot}()$  on the corresponding object. For every  $p_i$  and  $p_j$  that invoke  $\text{write\_snapshot}()$  with values  $v_i$  and  $v_j$  as parameters and obtains sets  $sm_i$  and  $sm_j$  respectively, the three following properties are satisfied:

- *Self-inclusion.*  $\forall i : (i, v_i) \in sm_i$ .
- *Containment.*  $\forall i, j : sm_i \subseteq sm_j \vee sm_j \subseteq sm_i$ .

<sup>3</sup>The Chandra-Toueg original definition of failure detector required that the local output of a failure detector is a function of the failure pattern, while the failure detectors of  $\diamond\psi^y$  as defined in [38] allowed processes to interact with the failure detector providing a parameter to a query.

- *Immediacy*.  $\forall i, j : (i, v_i) \in sm_j \implies sm_i \subseteq sm_j$ .

These properties are represented in the first image of Figure 2, for the case of three processes. The image represents a *simplicial complex*, i.e., a family of sets closed under containment; each set is called a *simplex*, and it represents the views of the processes after accessing the *IS* object. The *vertices* are the 0-simplexes, of size one; edges are 1-simplexes, of size two; triangles are of size three (and so on). Each vertex is associated with a process  $p_i$ , and is labeled with  $sm_i$  (the *view*  $p_i$  obtains from the object).

The highlighted 2-simplex in the figure represents a run where  $p_1$  and  $p_3$  access the object concurrently, both get the same view seeing each other, but not seeing  $p_2$ , which accesses the object later, and gets back a view with the 3 values written to the object. But  $p_2$  can't tell the order in which  $p_1$  and  $p_3$  access the object; the other two runs are indistinguishable to  $p_2$ , where  $p_1$  accesses the object before  $p_3$  and hence gets back only its own value or the opposite. These two runs are represented by the corner 2-simplexes. Thus, the vertices at the corners of the complex represents the runs where only one process  $p_i$  accesses the object, and the vertices in the edges connecting the corners represent runs where only two processes access the object. The triangle in the center of the complex, represents the run where all three processes access the object concurrently, and get back the same view.

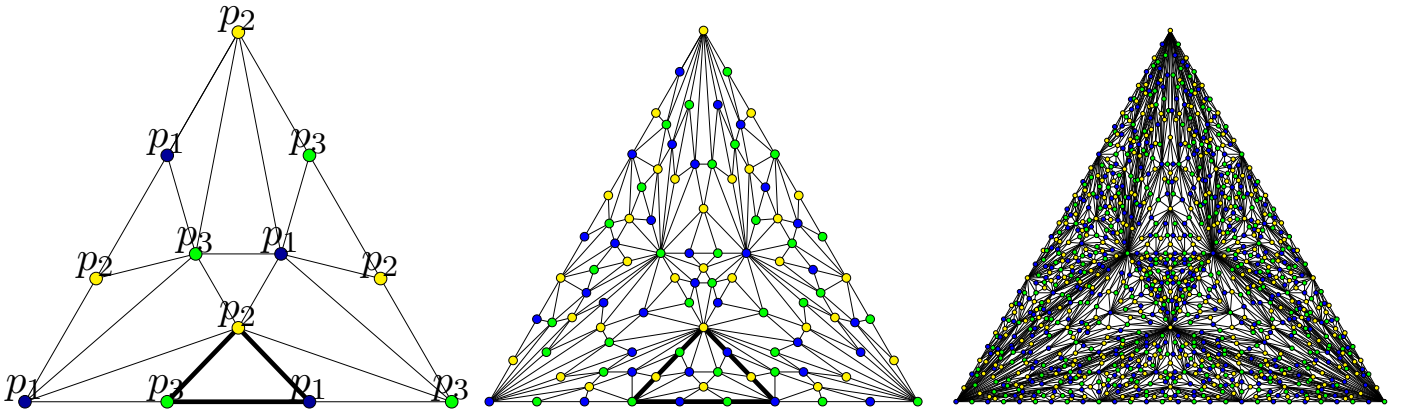


Figure 2: One, two and three rounds in the *IIS* model

In the *iterated immediate snapshot model (IIS)* the shared memory is made up of an infinite number of one-shot immediate snapshot objects  $IS[1], IS[2], \dots$ . These objects are accessed sequentially and asynchronously by each process, according to the round-based pattern described in Figure 3. In Figure 2 one can see that the *IIS* complex is constructed recursively by replacing each simplex by the one round complex.

```

 $r_i \leftarrow 0; val_i \leftarrow input;$ 
loop forever  $r_i \leftarrow r_i + 1;$ 
                $val_i \leftarrow \text{local computations};$ 
                $sm_i \leftarrow IS[r_i].write\_snapshot(val_i)$ 
end loop.

```

Figure 3: Generic algorithm in the Iterated Immediate Snapshot model

### 2.3 Tasks and computational equivalence of the two models

**Distributed tasks** A distributed *task*  $T$  is defined by two sets of  $n$ -dimensional vectors  $\mathcal{I}$  and  $\mathcal{O}$  and an input-output relation  $\Delta \subseteq \mathcal{I} \times \mathcal{O}$ . The relation  $\Delta$  specifies, for each input vector  $I$  the set of allowed output vectors. Operationally, in an execution, each process  $p_i$  is initially provided a private input value  $v_i$  and is required to *decide* and output value  $w_i$ . The input  $I$  and output vectors  $O$  of the execution are then the vectors formed by the input and output values respectively of the process (i.e.,  $I[i] = v_i$  and  $O[i] = w_i$  or  $\perp$  if  $p_i$  never decides).

A distributed algorithm  $\mathcal{A}$  *solves task*  $T$  if, in any execution of the algorithm with input vector  $I$ , each non-faulty process eventually decides on a private output value satisfying the task specification. Thus, it is required that the vector of output values can be extended to a vector  $O \in \mathcal{O}$  such that  $(I, O) \in \Delta$ . In the *k-set agreement* task,  $1 \leq k \leq n$ , the initial values are taken from some arbitrary set  $\mathcal{V}$ ,  $|\mathcal{V}| > k$ .  $\mathcal{I} = \mathcal{O} = \mathcal{V}^n$ , and for any pair  $(I, O) \in \mathcal{I} \times \mathcal{O}$ ,  $(I, O) \in \Delta$  if and only if

- *Validity.*  $\forall j, \exists i : O[j] = I[i]$  and,
- *Agreement.*  $|\{O[j], 1 \leq j \leq n\}| \leq k$ .

As can be seen in Figure 2, the *IIS* complex of global states at any round is a subdivided simplex, and hence Sperner's Lemma implies that  $k$ -set agreement is not solvable in the *IIS* model if  $k < n$ .

**Computational equivalence** Consider two models of computation  $M_1$  and  $M_2$  for  $n$  processes, and let  $S$  a set of tasks. Model  $M_1$  is *at least as strong as model  $M_2$  with respect to set  $S$*  if each task  $T \in S$  that can be solved in model  $M_2$  can also be solved in model  $M_1$ . The two models are said to be *equivalent* if  $M_1$  is *at least as strong as model  $M_2$*  and vice versa.

A task with a finite set of input vectors is *bounded*. The  $k$ -set agreement task is a bounded task. The following equivalence was proved in [9].

**Theorem 2.1.** [9] *The IIS model and the standard wait-free asynchronous shared memory model are equivalent with respect to bounded tasks.*

Therefore, as  $k$ -set agreement is not solvable in the *IIS* model for  $k < n$ ,

**Theorem 2.2.** [8, 31, 51] *In the  $n$ -processes wait-free shared memory model, the  $k$ -set agreement task cannot be solved if  $k < n$ .*

### 3 The IRIS model

This section presents the *IRIS* model associated with a failure detector class  $C$ , denoted  $IRIS(PR_C)$ , where  $C$  is a class in the families  $(\diamond S_x)_{1 \leq x \leq n}$ ,  $(\diamond \psi^y)_{1 \leq y \leq n}$  or  $(\Omega^z)_{1 \leq z \leq n}$ . As in the *IIS* model, the processes share an infinite sequence  $IS[1], IS[2], \dots$  of *IS* objects. In an execution, each process accesses sequentially the sequence of objects via `write_snapshot()` operations, following the round-based pattern described in Figure 3. To distinguish the operation in the *IIS* model and its more constrained counterpart of the *IRIS* model, the former is denoted  $IS[r].write\_snapshot()$ , while the latter is denoted  $IS[r].WRITE\_SNAPSHOT()$ .

The model consists of a subset of runs of the *IIS* model, that satisfy a corresponding  $PR_C$  property. `WRITE\_SNAPSHOT()` operations on a given objects  $IS[r]$  have the same semantic as `write_snapshot()` in the *IIS* model. However, the sequence of views returned in every infinite execution satisfies an additional property denoted  $PR_C$  that depends on the failure detector class  $C$  we consider.

Given an infinite execution,  $sm_i^r$  denotes the set returned by  $IS[r].WRITE\_SNAPSHOT()$  to process  $p_i$ . If  $p_i$  never accesses the  $r$ th *IS* object,  $sm_i^r = \emptyset$ . Note that in this case,  $sm_i^{r'} = \emptyset, \forall r' \geq r$ . Recall that each set  $sm$  is made up of pairs  $\langle j, val \rangle$  where  $j$  is the index of a process. We write  $j \in sm$  as a shorthand for  $\exists \langle j, val \rangle \in sm$ .

#### 3.1 The model $IRIS(PR_{\diamond S_x})$

The property  $PR_{\diamond S_x}$  states that there is a set  $Q$  of  $x$  processes, a process  $p_\ell$  and a round  $r$ , such that at each round  $r' \geq r$ , each process  $p_i \in Q$  either has crashed ( $sm_i^{r'} = \emptyset$ ) or obtains a view  $sm_i^{r'}$  that contains  $sm_\ell^{r'}$ . Formally,

**Definition 3.1.** (Property  $PR_{\diamond S_x}$ )  $PR_{\diamond S_x} \equiv \exists Q \subseteq \{1, \dots, n\}, \ell, r : |Q| = x \wedge \forall r' \geq r, \forall i \in Q : sm_i^{r'} = \emptyset \vee \ell \in sm_i^{r'}$

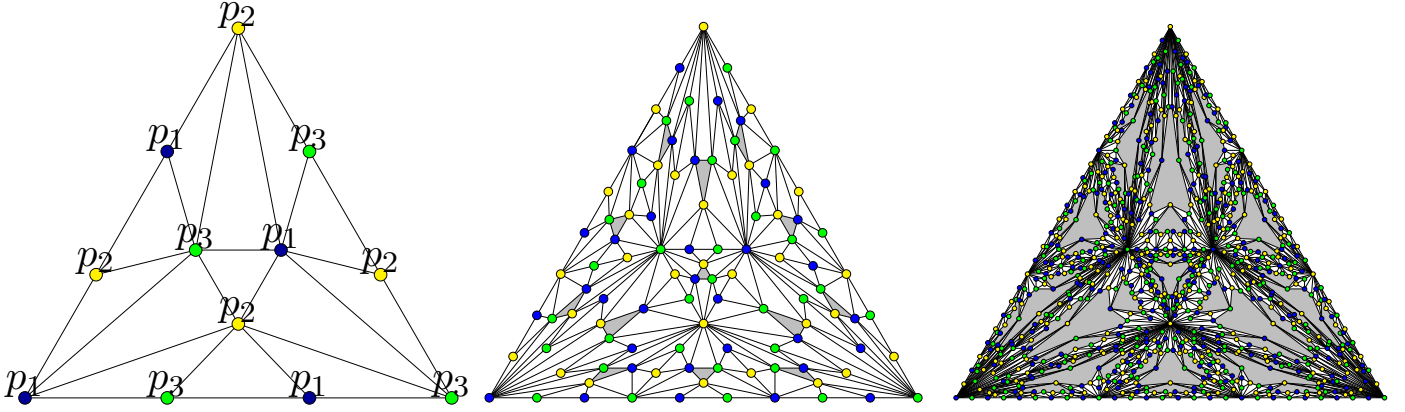
Figure 4 shows runs of the  $IRIS(PR_{\diamond S_x})$  model for  $x = 2$ . The complex remains connected in this case and consequently consensus is unsolvable in that model

Our first main result is the following.

**Corollary 3.2.** *A task is solvable in the read/write model equipped with a failure detector of the class  $\diamond S_x$  if and only if it is solvable in the  $IRIS(PR_{\diamond S_x})$  model.*

This result is a corollary of a more general theorem proved in Section 6. We prove that, for any arbitrary failure detector class  $C$ , if we are able to define an iterated model  $IRIS(PR_C)$ , that can be simulated in the shared memory model enriched with  $C$ , and conversely simulate a failure detector of the class  $C$  in  $IRIS(PR_C)$ , then the shared memory model enriched with  $C$  and the  $IRIS(PR_C)$  model have the same computational power. A simulation of  $IRIS(PR_{\diamond S_x})$  in the shared memory model enriched with  $\diamond S_x$  is presented in section 4.1, and Section 5.1 provides an iterated algorithm emulating a failure detector of the class  $\diamond S_x$  in  $IRIS(PR_{\diamond S_x})$ .



Figure 4: One, two and three rounds in  $IRIS(PR_{\diamond S_x})$  with  $x = 2$  and  $r = 2$ 

**The  $k$ -set agreement tasks with  $\diamond S_x$**  The power of the  $IRIS$  model becomes evident when we use it to prove the lower bound for  $k$ -set agreement in the shared memory model equipped with a failure detector of the class  $\diamond S_x$ .

**Theorem 3.3.** *In the shared memory model, in which any number of processes may crash, there is no  $\diamond S_x$ -based algorithm that solves  $k$ -set agreement if  $k < n - x + 1$ .*

The proof is established in the  $IRIS(PR_{\diamond S_x})$  models via a simple reduction argument. The lower bound applies to the shared memory model as well thanks to corollary 3.2.

The lower bound is obtained by reduction to an  $n - x + 1$ -processes wait-free shared memory system. We partition the  $n$  processes in two sets  $L$  and  $H$  where  $L$  consists in the first  $n - x + 1$  low-order processes  $\{p_1, \dots, p_{n-x+1}\}$  and  $H = \{p_{n-x+2}, \dots, p_n\}$  is the set of the remaining high-order processes. We observe that every  $IIS$  runs where the processes in  $L$  never see the process in  $H$  trivially satisfy the  $PR_{\diamond S_x}$  property. More precisely, we consider the subset of executions in which for every round, and every  $p_i \in L, p_j \in H, j \notin sm_i^r$ . By Theorem 2.2 and the computational equivalence between the shared memory model and the  $IIS$  model (Theorem 2.1), there is no algorithm that solves the  $k$ -set agreement task for  $k < n - x + 1$  in these executions. Therefore, the task is unsolvable as well in our  $IRIS(PR_{\diamond S_x})$  model. Theorem 6.9 then implies that it is unsolvable in the read/write shared memory model equipped with a failure detector of the class  $\diamond S_x$ .

The argument is illustrated in Figure 5. It depicts the first three rounds of a subset of legal executions in the  $IRIS(PR_{\diamond S_2})$  model. Figure 5 shows all executions that satisfy property  $PR_{\diamond S_2}$  with the following parameters:  $Q = \{p_2, p_3\}$  and  $\ell = p_2$ . This set of executions contains all possible wait-free executions of processes  $p_1$  and  $p_2$  (these executions are highlighted in the picture). Moreover, neither  $p_1$  nor  $p_2$  see  $p_3$  in their successive views. Therefore, an algorithm designed for the  $IRIS(PR_{\diamond S_2})$  model that solves some task  $T$  can be directly used to wait-free solve the same task among  $p_1$  and  $p_2$ .

### 3.2 The models $IRIS(PR_{\diamond \psi^y})$ and $IRIS(PR_{\Omega^z})$

This section shows how to define iterated restricted immediate snapshot models  $IRIS(PR_C)$  for other families of when  $C$  is a failure detector class in the families  $(\diamond \psi^y)_{1 \leq y \leq n}$  or  $(\Omega^z)_{1 \leq z \leq n}$ .

Given an infinite execution, let  $f, 0 \leq f \leq n - 1$  denotes the actual number of processes that fail in that execution. The property  $PR_{\diamond \psi^y}$  is defined as follows.

**Definition 3.4.** (Property  $PR_{\diamond \psi^y}$ )  $PR_{\diamond \psi^y} \equiv \exists r : \forall r' \geq r : ((i - 1 = (r' - 1) \bmod n) \wedge (sm_i^{r'} \neq \emptyset)) \implies |sm_i^{r'}| \geq n - \max(n - y, f)$ .

The intuition that underlies this property is the following: there is a logical time (round number) after which each correct process obtains infinitely often a view that misses at most  $\max(n - y, f)$  processes. As we can see, when  $f < n - y$  such views can miss correct processes. As a particularly simple case, let us consider the instance  $y = n$  (as already noticed,  $\diamond \psi^n$  is equivalent to  $\diamond P$ ):  $PR_{\diamond \psi^n}$  states that after some round there is an infinite number of rounds at which  $p_i$  obtains a view containing the  $(n - f)$  correct processes.

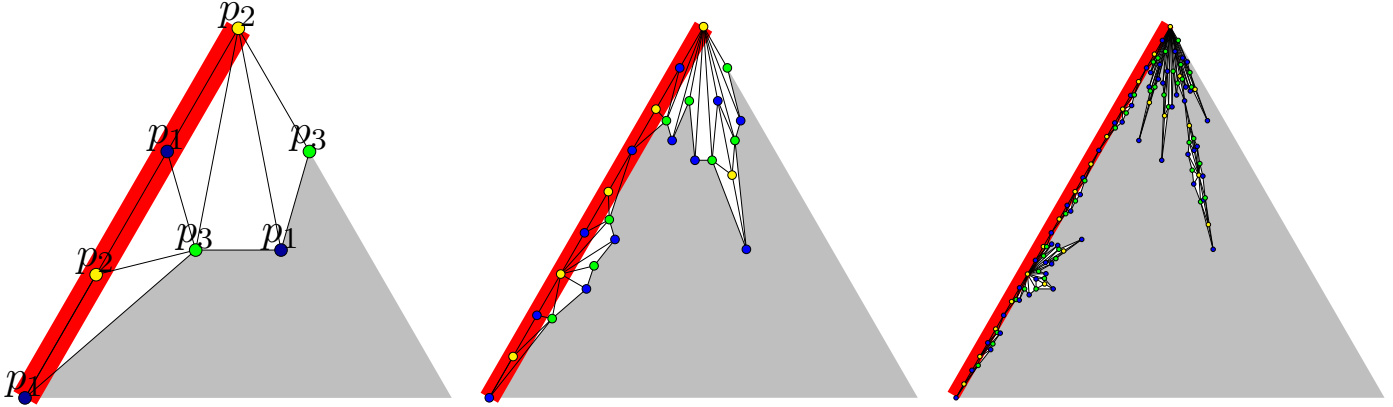


Figure 5: Subsets of  $IRIS(PR_{\diamond S_2})$  that contain all executions by  $p_1$  and  $p_2$

The property  $IRIS(PR_{\Omega^z})$  is defined as follows (where  $L$  is a set of process indexes). Recall that in each round the sets  $sm_i^r$  (seen as sets of processes indexes) returned by the operations  $WRITE\_SNAPSHOT()$  performed on the object  $IS[r]$  are ordered by containment. Let  $smin^r$  be the smallest set among the sets  $sm_i^r$ .

**Definition 3.5.** (Property  $PR_{\Omega^z}$ )  $PR_{\Omega^z} \equiv \exists L, r : |L| = z \wedge \forall r' \geq r, smin^{r'} \subseteq L$ .

The property  $PR_{\Omega^z}$  ensures that there exists a set  $L$  of  $z$  processes such that after round  $r$ , the smallest view  $smin$  is contain only indexes of processes of  $L$ . Thus, past this round, each view  $sm$  returned by every  $WRITE\_SNAPSHOT()$  invocation contains processes in  $L$ .

Let us consider the case  $z = 1$ , i.e., the simplest instance of  $PR_{\Omega^z}$ . In that case, there  $|L| = 1$  and there exists a process  $p_\ell$  such that  $L = \{p_\ell\}$ . Therefore, for every round  $r' \geq r$ , the value  $v$  written  $p_\ell$  in the object  $IS[r']$  (by calling  $IS[r'].WRITE\_SNAPSHOT(v)$ ) is seen by every non-faulty process  $p_i$ , (i.e.,  $\langle \ell, v \rangle \in sm_i^{r'}$ ). Said differently, whatever the concurrency degree among the  $IS[r'].WRITE\_SNAPS$  invocations during round  $r'$ , the invocation of  $p_\ell$  is always set-linearized first, and no other invocation is set-linearized together with it.

The instances  $z > 1$  are weaker in the sense that they allow several  $WRITE\_SNAPSHOT()$  invocations issued by the processes of a subset of  $L$  to be set-linearized first. Moreover this subset of  $L$  can differ from one round to another (This property is close to, but different from, the notion of  $z$ -bounded concurrency [21]).

## 4 Simulations of the $IRIS(PR_C)$ model in the shared memory model with $C$

This section presents simulations of the  $IRIS(PR_C)$  model from the shared memory model equipped with a failure detector of one the families  $(\diamond S_x)_{1 \leq x \leq n}$ ,  $(\diamond \psi^y)_{1 \leq y \leq n}$  and  $(\Omega_{1 \leq z \leq n}^z)$ . The aim is to produce subsets of runs of the  $IIS$  model that satisfy the property  $PR_C$ . To that end, each of the constructions described in Figure 6, 7, and 8 relies on an infinite array of immediate snapshot objects  $IS[1], \dots$  that can be in addition read in snapshots. Given an object  $IS[r]$ , views returned by  $snapshot()$  and  $write\_snapshot()$  are ordered by containment, and the corresponding invocations can be consistently set-linearized. Such objects can be implemented in the shared memory model tolerating an arbitrary number of failures [1, 7]. In addition to this shared array, each construction uses appropriate additional shared registers and local variables.

In each construction, the last operation on shared objects issued by a process in round  $r$  is an  $IS[r].write\_snapshot()$ . It consequently follows that the constructed  $IS[r].WRITE\_SNAPSHOT()$  automatically benefits from the self-inclusion, containment and immediacy properties of the underlying object. This means that only the property  $PR_C$  has to be proved.

### 4.1 From the shared memory model with $\diamond S_x$ to $IRIS(PR_{\diamond S_x})$

The algorithm is described in Figure 6. It relies on the representative-based definition of the class  $\diamond S_x$ .

When it invokes  $IS[r].WRITE\_SNAPSHOT(v_i)$ , a process  $p_i$  repeatedly (1) issues a snapshot operation on  $IS[r]$  in order to know which processes have already written in the object  $IS[r]$ , and (2) reads the value locally output by the failure detector ( $REPR_i$ ), until it discovers that it is its own representative ( $rp_i = i$ ) or its representative has already written in  $IS[r]$  ( $(rp_i, *) \in sm_i$ ). When

```

operation  $IS[r].WRITE\_SNAPSHOT(v_i)$ :
(1) repeat  $sm_i \leftarrow IS[r].snapshot()$ ;  $rp_i \leftarrow REPR_i$ 
(2) until  $((rp_i, *) \in sm_i) \vee (rp_i = i)$  end repeat;
(3)  $sm_i \leftarrow IS[r].write\_snapshot(v_i)$ ;
(4) return  $(sm_i)$ .

```

Figure 6: From the shared memory model with  $\diamond_{S_x}$  to the  $IRIS(PR_{\diamond_{S_x}})$  model (code for  $p_i$ )

this occurs,  $p_i$  invokes  $R[r].write\_snapshot(v_i)$  to write  $v_i$ . It finally returns the snapshot value obtained by that `write_snapshot()` invocation.

**Lemma 4.1.** *In the shared memory model equipped with a failure detector of the class  $\diamond_{S_x}$ , the algorithm described in Figure 6 simulates the  $IRIS(PR_{\diamond_{S_x}})$  model.*

*Proof.* Let us consider an infinite execution of the algorithm described in Figure 6. By the definition of the class  $\diamond_{S_x}$ , there exists a set  $Q$  of at least  $x$  processes, a correct process  $p_\ell$  and a time  $\tau$  after which  $REPR_i = \ell$  for every non-faulty process in  $Q$  and  $REPR_i = i$  for each  $i \notin Q$ . Let  $R$  the index of a  $IS$  objects that has not been accessed by time  $\tau$ , i.e., no processes has invoked  $IS[R].WRITE\_SNAPSHOT()$  by time  $\tau$ .

We first note that the simulation is non blocking. Suppose for contradiction that for round some round  $r$ , the invocation  $IS[r].WRITE\_SNAPSHOT()$  by some correct process never terminates. Eventually, at each correct process  $p_i$  the value of  $REPR_i$  is either  $i$  or the index  $\ell$  of the correct process  $p_\ell$ . In the first case, the condition that ends the repeat loop is eventually satisfied. In the second case, as the correct process  $p_\ell$  eventually exits the repeat loop (this follows from the first case), and therefore writes a value in the object  $IS[r]$ , the snapshots taken by the waiting process eventually contain the index of its representative, thus enabling the loop to terminate.

Finally, consider a round  $r \geq R$  and a non-faulty process  $p_i$  whose invocation of  $IS[r].WRITE\_SNAPSHOT()$  terminates. When  $p_i$  invokes  $IS[r].write\_snapshot()$ ,  $p_i$  has previously obtained a snapshot containing the index of  $p_\ell$ . Hence, the operation `write_snapshot()` issued by  $p_\ell$  is set-linearized after that operation by  $p_i$ . Therefore,  $sm_\ell^r \subseteq sm_i^r$  and we have  $\ell \in sm_i^r$  as required by the definition of  $PR_{\diamond_{S_x}}$ .  $\square$

## 4.2 From the shared memory model with $\diamond_{\psi^y}$ to $IRIS(PR_{\diamond_{\psi^y}})$

The construction described in figure 7 (that has some  $\ell$ -mutual exclusion flavor [3]) uses a deterministic function  $order(r)$ , where the parameter  $r$  is a round number. This function orders the process indexes as follows:  $order(r)$  returns a sequence of the indexes  $1, \dots, n$  in which the last element is the index  $i$  such that  $(i - 1) = (r - 1) \bmod n$ .

```

operation  $IS[r].WRITE\_SNAPSHOT(v_i)$ :
(1)  $sequence_i \leftarrow order(r)$ ;
(2)  $pred_i \leftarrow \{j : j \text{ appears before } i \text{ in } sequence_i\}$ ;
(3) repeat  $sm_i \leftarrow IS[r].snapshot()$ ;
(4)  $seen_i \leftarrow sm_i \cap pred_i$ ;
(5)  $nb_{c_i} \leftarrow NBC_i$ 
(6) until  $(|pred_i| - nb_{c_i} \leq |seen_i|)$  end repeat;
(7)  $sm_i \leftarrow IS[r].write\_snapshot(v_i)$ ;
(8) return  $(sm_i)$ .

```

Figure 7: From the base shared memory model with  $\diamond_{\psi^y}$  to  $IRIS(PR_{\diamond_{\psi^y}})$  (code for  $p_i$ )

The simulation is described in Figure 7. When it invokes  $IS[r].WRITE\_SNAPSHOT(v_i)$ , process  $p_i$  first computes the sequence  $(sequence_i)$  of process indexes associated with the round  $r$  (line 1), and determines the set of processes  $(pred_i)$  that are ordered before it in that sequence (line 2). Then,  $p_i$  enters a loop in which it determines the set of processes that have already written in  $IS[r]$   $(seen_i)$  and whose indexes precedes  $i$  in the sequence  $sequence_i$  (these are the processes in  $pred_i$ ).  $p_i$  also reads the value  $(NBC_i)$  currently provided by the underlying failure detector (line 5), which is an approximation of the number of crashed processes.

$p_i$  exits the loop when the processes of  $pred_i$  that it perceives as not crashed have written in  $IS[r]$  (line 6);  $p_i$  locally estimates there are at least  $(|pred_i| - nb_{c_i})$  such processes. As in the previous simulations, when this predicate becomes true,  $p_i$  writes its value in  $IS[r]$  (line 7) and returns the associated snapshot value it has just obtained (line 8).

**Lemma 4.2.** *Every invocation of `WRITE_SNAPSHOT()` by a correct process terminates.*

*Proof.* The proof is by contradiction. Let  $r$  the smallest round in which a `WRITE_SNAPSHOT()` invocation by a correct process never returns. Let  $B$  the set of correct processes whose round  $r$  invocation of `WRITE_SNAPSHOT()` never return. Let  $s$  the smallest process index in  $B$  in the sequence order( $r$ ). We show next that the invocation of  $IS[r].WRITE\_SNAPSHOT()$  by  $p_s$  returns: a contradiction.

Let us consider  $pred_s$  (i.e., the set of process indexes that are before  $s$  in order( $r$ )). We consider two cases:

- $|pred_s| \leq \max(n - y, f)$ . It follows from the eventual accuracy property of the class  $\diamond\psi^y$  that eventually we always have  $nbc_s = \max(t + 1 - y, f)$ . Consequently, eventually  $|pred_s| - nbc_s \leq 0$  and thus  $|seen_s| = |m_s \cap pred_s| \geq 0$ . Therefore, the predicate of line 6 is eventually true. Hence, the invocation  $IS[r].WRITE\_SNAPSHOT()$  by process  $p_s$  eventually returns.
- $|pred_s| > \max(n - y, f)$ . Let  $faulty(S)$  denote the set of faulty processes in the set  $S$ . We have  $|faulty(pred_s)| \leq |faulty(\{1, \dots, n\})| = f \leq \max(t + 1 - y, f)$ . Let  $\alpha$  be the number of correct processes in  $pred_s$ . Note that  $\alpha = |pred_s| - |faulty(pred_s)| \geq |pred_s| - \max(t + 1 - y, f)$ . Let us recall that these  $\alpha$  processes have a rank smaller than  $s$  in order( $r$ ). Moreover, it follows from the definition of  $p_s$  that all invocations of `WRITE_SNAPSHOT()` by every correct processes whose index is smaller than  $s$  in order( $r$ ) returns. Hence, there is a time  $\tau_1$  after which we always have  $|seen_s| = |m_s \cap pred_s| \geq \alpha$ .

Finally, due to the eventual accuracy property of  $\diamond\psi^y$ , there is time  $\tau_2$  after which  $nbc_s = \max(t + 1 - y, f)$ . Therefore, after time  $\tau = \max(\tau_1, \tau_2)$ ,  $|seen_s| = |m_s \cap pred_s| \geq \alpha$  and  $\alpha \geq |pred_s| - nbc_s$ , from which we conclude that  $p_s$  eventually exits the **repeat** loop. Hence, the invocation  $IS[r].WRITE\_SNAPSHOT()$  by process  $p_s$  eventually returns.  $\square$

**Lemma 4.3.** *In the shared memory model equipped with a failure detector of the class  $\diamond\psi^y$ , the algorithm described in Figure 7 simulates the  $IRIS(PR_{\diamond\psi^y})$  model.*

*Proof.* It follows from Lemma 4.2 that each correct process executes an infinite number of rounds (a requirement of any  $IRIS(PR_C)$  model). So, it remains to show that the property  $PR(\diamond\psi^y)$  is satisfied.

By the eventual the eventual accuracy property of the class  $\diamond\psi^y$  there is a time  $\tau$  after which  $NBC_i = \max(n - y, f)$  for every correct process  $p_i$ . Let  $R$  a round such that every  $IS[R].WRITE\_SNAPSHOT()$  invocation starts after  $\tau$ .

Let  $p_i$  a correct process and  $r \geq R$  a round such that  $i - 1 = (r - 1) \bmod n$ . We show that  $|sm_i^r| \geq n - \max(n - y, f)$ .

By the choice of  $r$ , the rank of  $i$  in order( $r$ ) is  $n$ , i.e.,  $|pred_i| = n - 1$ . As the invocation of `WRITE_SNAPSHOT()` by  $p_i$  terminates (Lemma 4.2), the predicate  $(|pred_i| - nbc_i \leq |seen_i|)$  (line 7) is true, from which we have  $|pred_i| - nbc_i = (n - 1) - \max(t + 1 - y, f) \leq |seen_i|$ . As  $i \notin seen_i$ , but  $i \in sm_i^r$  and  $seen_i \subset sm_i^r$  (immediacy and containment properties of immediate snapshots),  $n - 1 - \max(n - y, f) \leq |seen_i| < |sm_i^r|$ , which implies that  $n - \max(n - y, f) \leq |sm_i^r|$ , as required by definition 3.4.  $\square$

### 4.3 From the shared memory model with $\Omega^z$ to $IRIS(PR_{\Omega^z})$

This construction is described in Figure 8. As in the previous construction, a one-shot immediate snapshot object  $IS[r]$  is associated with each round  $r$ . When process  $p_i$  invokes  $IS[r].WRITE\_SNAPSHOT(v_i)$ , it first waits until either some process has written in  $IS[r]$ , or its index belongs to the set `LEADERi` managed by its local failure detector. When one of these conditions becomes true,  $p_i$  writes in  $IS[r]$  by invoking  $IS[r].write\_snapshot(v_i)$ . This invocation returns a snapshot of  $IS[r]$  that  $p_i$  returns as the result of its `WRITE_SNAPSHOT()` invocation.

<p><b>operation</b> <math>IS[r].WRITE\_SNAPSHOT(\langle i, v_i \rangle)</math>:</p> <ol style="list-style-type: none"> <li>(1) <b>repeat</b> <math>sm_i \leftarrow IS[r].snapshot(); ld_i \leftarrow LEADER_i</math></li> <li>(2) <b>until</b> <math>(sm_i \neq \emptyset) \vee (i \in ld_i)</math> <b>end repeat</b>;</li> <li>(3) <math>sm_i \leftarrow IS[r].write\_snapshot(v_i)</math>;</li> <li>(4) <b>return</b> <math>(sm_i)</math>.</li> </ol>
---

Figure 8: From the shared memory model with  $\Omega^z$  to  $IRIS(PR_{\Omega^z})$  (code for  $p_i$ )

**Lemma 4.4.** *In the shared memory model equipped with a failure detector of the class  $\Omega^z$ , the algorithm described in Figure 8 is a simulation of the  $IRIS(PR_{\Omega^z})$  model.*

*Proof.* The proof is made up of two parts: (1) any correct process executes an infinite number of rounds; and (2) the property  $PR_{\Omega^z}$  is satisfied.

1. To prove that every correct process  $p_i$  executes an infinite number of rounds, we have to show that the local predicate  $(sm_i \neq \emptyset) \vee (i \in ld_i)$  evaluated by  $p_i$  at line 2 is eventually true for each round  $r \geq 1$ . Let us proceed by contradiction. Let  $r$  be the first round at which a correct process  $p_i$  remains blocked forever, i.e.,  $(sm_i = \emptyset) \wedge (i \notin ld_i)$  remains forever true once. This means that, after some time,  $i$  never belongs to  $LEADER_i$  when  $p_i$  reads this set, and  $sm_i$  remains always empty. As  $m_i$  remains empty, no invocations of  $IS[r].WRITE\_SNAPSHOT()$  terminate (Observation O1).

However, due to the eventual leadership property of the class  $\Omega^z$ , there is a set  $L$  of size at most  $z$  containing at least one correct process  $p_\ell$  such that, after some arbitrary (but finite) time  $\tau$ , the predicate  $LEADER_\ell = L$  is true forever at  $p_\ell$ . It follows that while  $p_\ell$  is blocked at round  $r$ , the local predicate  $\ell \in ld_\ell$  becomes eventually true. Consequently, the round  $r$  invocation of  $WRITE\_SNAPSHOT()$  by  $p_\ell$  eventually terminates and  $p_\ell$  proceeds to the round  $r + 1$  (Observation O2). The observations O1 and O2 contradict each other, from which we conclude that every correct process executes an infinite number of rounds.

2. Let us now show that the property  $PR_{\Omega^z}$  is satisfied. By the eventual leadership property of the class  $\Omega^z$ , there are a set  $L$  containing at least one correct process  $p_\ell$  (and at most  $z$  processes) and a time  $\tau$  such that, after  $\tau$ , we always have  $LEADER_i = L$  for every correct process  $p_i$ . Due to the very existence of  $\tau$ , and the fact that the correct processes execute rounds infinitely often, we conclude that there is a round  $r$  such that, for every round  $r' \geq r$ , we have  $ld_i = L$  for every correct process  $p_i$ .

Let  $L(r')$  be the subset of the processes of  $L$  that stop waiting at line 1 because the predicate  $i \in ld_i$  is true while the predicate  $sm_i \neq \emptyset$  is false. Let us also notice that the invocations  $IS[r].write\_snapshot()$  issued by the processes of  $L(r')$  are set-linearized before the invocations issued by the processes that do not belong to  $L(r')$ . Therefore, sets returned by the invocations  $IS[r].write\_snapshot()$  satisfy the inclusion, immediacy and containment properties, the smallest set returned is contained in  $L(r')$ . As the set returned by the invocation of  $IS[r].write\_snapshot()$  is the set output by the  $write\_snapshot()$  operation at each process, we have  $sm_i^{r'} \subseteq L(r')$ . Since  $L(r') \subseteq L$ , we conclude that for every round  $r' \geq r$ , the smallest snapshot  $sm_i^{r'}$  is included in  $L$ . This completes the proof as  $|L| \leq z$ .  $\square$

## 5 Extracting a failure detector of the class $\mathcal{C}$ in the $IRIS(PR_{\mathcal{C}})$ model

Given the read/write model equipped with a failure detector of the class  $\mathcal{C}$ , the previous section has shown how to simulate the  $IRIS(PR_{\mathcal{C}})$  model. This section presents algorithms for the iterated model  $IRIS(PR_{\mathcal{C}})$  that construct a failure detector of the class  $\mathcal{C}$ . In each of these algorithms, a variable  $FD_i$  is maintained at each process  $p_i$ ; the successive values of this variable simulate the output of a failure detector of the class  $\mathcal{C}$ .

Section 6 provides a complete simulation from the  $IRIS(PR_{\mathcal{C}})$  model to the read/write model equipped with a failure detector of the class  $\mathcal{C}$ , provided that a failure detector of the class  $\mathcal{C}$  can be emulated in  $IRIS(PR_{\mathcal{C}})$ . Suppose that a task  $T$  is solvable in the shared memory model equipped with a failure detector of the class  $\mathcal{C}$ , where  $\mathcal{C}$  is a failure detector of one of the classes  $\{\diamond S_x\}$ ,  $\{\diamond \psi^y\}$  or  $\{\Omega^z\}$ . The emulations presented in this section, together with the general simulation described in Section 6 imply that  $T$  is also solvable in  $IRIS(PR_{\mathcal{C}})$ .

### 5.1 From $IRIS(PR_{\diamond S_x})$ to a failure detector of the class $\diamond S_x$

In a model equipped with a failure detector, each process can read at any time the output of the failure detector. We denote  $TRUSTED_i$  the variable that emulates the output of a failure detector of the class  $\diamond S_x$  at process  $p_i$ . A trivial algorithm that simulates a failure detector of the class  $\diamond S_x$  in the  $IRIS(PR_{\diamond S_x})$  model is described in figure 9.

```

init  $r_i \leftarrow 0$ ;  $TRUSTED_i \leftarrow \Pi$ .

repeat forever
(1)  $r_i \leftarrow r_i + 1$ ;  $sm_i \leftarrow IS[r_i].WRITE\_SNAPSHOT(i)$ ;
(2)  $TRUSTED_i \leftarrow \{j : j \in sm_i\}$ 
end repeat.
```

Figure 9: Emulation of a failure detector of the class  $\diamond S_x$  in  $IRIS(PR_{\diamond S_x})$

**Lemma 5.1.** *The algorithm of Figure 9 implements a failure detector of the class  $\diamond S_x$ .*



*Proof.* Consider an infinite execution. We prove that the values of the variable  $\text{TRUSTED}_i$  satisfy the first variant of the definition of the class  $\diamond\mathcal{S}_x$ . It is easy to see that strong completeness is ensured: a faulty process  $p_j$  accesses finitely many  $IS$  objects. Hence, after some time, no set  $\text{TRUSTED}$  contains  $j$ . For the limited accuracy property, let  $Q$ ,  $\ell$  and  $R$  be respectively the set of at least  $x$  processes, the index of a process and the round number as defined by the property  $PR_{\diamond\mathcal{S}_x}$  (Definition 3.1). Clearly, for every  $r \geq R$ , and every  $p_i \in Q$ , every set  $sm_i^r$  obtained in round  $r$  as a result of a  $\text{WRITE\_SNAPSHOT}()$  invocation contains  $\ell$ . Moreover,  $p_\ell$  is a correct process. Hence, there exists a correct process ( $p_\ell$ ) and a set of  $x$  processes ( $Q$ ) such that  $p_\ell$  is eventually always trusted by each member of the set, as desired.  $\square$

## 5.2 From $IRIS(PR_{\diamond\psi^y})$ to a failure detector of the class $\diamond\psi^y$

Figure 10 builds a failure detector of the class  $\diamond\psi^y$  from  $IRIS(PR_{\diamond\psi^y})$ . It has the same structure as the previous algorithm. The only lines that are modified are the initialization line and line 2. The aim of this new line is to take into account the property of  $PR_{\diamond\psi^y}$ . The emulated failure detector output is kept in the variable  $\text{NBC}_i$ .

```

init  $r_i \leftarrow 0$ ;  $\text{NBC}_i \leftarrow (n - y)$ .

repeat forever
(1)  $r_i \leftarrow r_i + 1$ ;  $sm_i \leftarrow IS[r_i].\text{WRITE\_SNAPSHOT}(i)$ ;
(2) if  $(i - 1) = ((r_i - 1) \bmod n)$  then  $\text{NBC}_i \leftarrow \max(n - y, n - |sm_i|)$  end if
end repeat.

```

Figure 10: Emulation of a failure detector of the class  $\diamond\psi^y$  in  $IRIS(PR_{\diamond\psi^y})$

**Lemma 5.2.** *The algorithm described in Figure 10 simulates a failure detector of the class  $\diamond\psi^y$  in the  $IRIS(PR_{\diamond\psi^y})$  model.*

*Proof.* The proof is nearly the same as for Lemma 5.1. It is left to the reader.  $\square$

## 5.3 From $IRIS(PR_{\Omega^z})$ to a failure detector of the class $\Omega^z$

The algorithm described in Figure 11 emulates a failure detector of the class  $\Omega^z$ . It provides each process  $p_i$  with a local variable  $\text{LEADERS}_i$  containing set of processes indexes. The successive values of the sets  $\text{LEADER}_i$  satisfy the weak eventual leadership of the failure detector class  $\Omega^z$ .

The algorithm consists in identifying “fresh” smallest snapshots. According to the definition  $PR_{\Omega^z}$ , we know that in each infinite run of the  $IRIS(PR_{\Omega^z})$  model, there is a round  $R$  and a set  $L$  such that for every  $r \geq R$ , the processes indexes that appear in the smallest snapshot  $smin^r$  of round  $r$  are contained in  $L$ . We also observe that eventually, only correct processes indexes appear in the snapshots returned by  $\text{WRITE\_SNAPSHOT}()$  operations. Hence, sequence of smallest snapshots ( $smin^r$ ) (or any infinite sub-sequence) is a valid output for a failure detector of the class  $\Omega^z$ .

The algorithm relies on the following characterization of the smallest snapshot of round  $r$ :

$$s = smin^r \iff \forall j \in s, s = sm_j^r.$$

Each process  $p_i$  maintains a local history  $h_i$  intended to record the snapshot that other processes get back as responses to  $\text{WRITE\_SNAPSHOT}()$  invocations.  $h_i$  is a two dimensional array;  $h_i[r][j]$  is initially equal to  $\emptyset$ ; if  $p_i$  learns the snapshot  $s$  obtained by  $p_j$  in round  $r$ ,  $h_i[r][j]$  is updated to contain that value. Let  $h_i[r : r'][i]$  denote the  $i$ th column of the rows  $r, \dots, r'$  (i.e., the values  $h_i[r][i], h_i[r + 1][i], \dots, h_i[r'][i]$ ).

Each time a new smallest snapshot  $smin$  is identified by  $p_i$ , the variable  $\text{LEADER}_i$  is set to  $smin$ , if the smallest snapshot is more recent than the previous smallest snapshot identified by  $p_i$ .  $p_i$  identifies smallest snapshots by observing  $h_i$ . If in row  $r$ , there exists a set  $s$  such that for each  $j \in s$ , the entry  $j$  is equal to  $s$ , then  $s$  is the smallest snapshot of round  $r$ . The correctness of the emulation relies on the fact that in round  $r$ ,  $p_i$  can always find the smallest snapshot of some round  $r'$  where  $r - n + 1 \leq r' \leq r$  (Lemma 5.3).

**Lemma 5.3.** *Let  $R \geq n$ ,  $p_i$  a process that has not failed by the end of round  $R$ , and  $h$  the value of the variable  $h_i$  after the update steps of round  $R$  (at line 6). There exists a round  $r$ ,  $n - R + 1 \leq r \leq R$  and a non-empty set  $s$  of processes indexes such that for each  $j \in s$ ,  $h[r][j] = s$ .*

*Proof.* Let assume for contradiction that the lemma is not true. To simplify the exposition, we number  $1, \dots, n$  the rounds  $R - n + 1, \dots, R$ . For each round  $r$ ,  $1 \leq r \leq n$ , let  $\sigma[r]$  denotes the set of processes indexes in the smallest snapshot of round  $r$  (i.e.,  $\sigma[r] = \{j : \langle j, * \rangle \in smin^r\}$ .) and  $s_i^r$  the indexes in the snapshot received by  $p_i$  in round  $r$  (i.e.,  $s_i^r = h_i[r][i]$ ).

```

init  $r_i \leftarrow 0$ ;  $h_i[1..+\infty][1..n] \leftarrow [1..+\infty][\emptyset, \dots, \emptyset]$ ;  $\text{LEADER}_i \leftarrow \{1, \dots, z\}$ .

repeat forever
(1)  $r_i \leftarrow r_i + 1$ ;  $sm_i \leftarrow IS[r_i].\text{WRITE\_SNAPSHOT}(\langle i, h_i[\max(1, n - r_i + 1) : r][i] \rangle)$ ;
(2)  $h_i[r_i][i] \leftarrow \{\ell : \langle \ell, * \rangle \in sm_i\}$ ;
    % update history %
(3) for each  $\ell : \langle \ell, h_\ell \rangle \in sm_i$  do
(4)   for each  $r \in \{n - r_i + 1, \dots, r_i - 1\}$  do
(5)     if  $h_i[r][\ell] = \emptyset$  then  $h_i[r][\ell] \leftarrow h_\ell[r][\ell]$  end if
(6)   end for end for;
    % look for smallest snapshots %
(7) for each  $r \in \{n - r_i + 1, \dots, r_i - 1\}$  do
(8)   if  $\exists s \subseteq \{1, \dots, n\}, s \neq \emptyset$  such that  $\forall j \in s, h_i[r][j] = s$  then  $\text{LEADER}_i \leftarrow s$  end if
(9) end for
end repeat.

```

Figure 11: Emulation of a failure detector of the class  $\Omega^z$  in  $IRIS(PR_{\Omega^z})$ 

Note that  $|s_i^n| \geq 2$ . Otherwise,  $|s_i^1| = 1$  and, by self-inclusion we have  $s_i^n = \{i\}$ . Hence  $\sigma[n] = s_i^n$ , which is known by  $p_i$ . Let  $j_{n+1}, j_n$  two distinct indexes in  $s_i^n$ .

Consider some round  $r$ ,  $1 \leq r < n$ . Notice that for each  $j \in s_i^r$ ,  $p_i$  knows the snapshot obtained by  $p_i$  in each round  $r' : 1 \leq r' \leq r - 1$ . More precisely, we have for each such  $j$   $h_i[r'][j] = s_i^{r'}$ . Suppose that  $s_i^{r-1} \subseteq \bigcup_{r \leq r' \leq n} s_i^{r'}$ . It then follows that for each subset  $s$  of  $s_i^{r-1}$  and each  $j \in s$ , the round  $r - 1$  snapshot of  $p_j$  is known by  $p_i$ . In particular, this holds for the smallest snapshot  $\sigma[r - 1]$ . So, there exists  $s$  such that for each  $j \in s$ ,  $h_i[r - 1][j] = s$ , which contradicts our initial assumption stating that the lemma is false. Therefore,  $s_i^{r-1} \not\subseteq \bigcup_{r \leq r' \leq n} s_i^{r'}$ . Let  $j_{r-1} \in s_i^{r-1} \setminus \bigcup_{r \leq r' \leq n} s_i^{r'}$ . Thus, we construct a sequence  $j_1, \dots, j_{n+1}$  of  $n + 1$  distinct processes indexes, which is impossible as the system consists of  $n$  processes.  $\square$

**Lemma 5.4.** *The algorithm described in Figure 11 emulates a failure detector of the class  $\Omega^z$  in  $IRIS(PR_{\Omega^z})$ .*

*Proof.* According to the property  $PR_{\Omega^z}$ , there exists a round  $R_0$ , and a set  $L$  of processes indexes such that the smallest snapshot of round  $R_0$  and every subsequent round contains indexes in  $L$ . Also, there exists a round  $R_1$  after which every snapshot contains only correct processes indexes. Let  $R = \max(R_0, R_1) + n$ . Let us consider a round  $r \geq R$  and a correct process  $p_i$ . By Lemma 5.3, the variable  $\text{LEADER}_i$  contains the processes indexes that appear in the smallest snapshot of some round  $r'$  where  $r - n + 1 \leq r' \leq r$ . In particular, this implies that  $r' \geq \max(R_0, R_1)$  from which we conclude that  $\text{LEADER}_i$  contains the index of a correct process and that  $\text{LEADER}_i \subseteq L$ . Therefore, for each correct process  $p_i$ , we eventually have  $\text{LEADER}_i \subseteq L$  and  $\text{LEADER}_i \cap \text{Correct} \neq \emptyset$ . Thus, the weak eventual leadership property is satisfied and the emulated failure detector is in the class  $\Omega^z$ .  $\square$

## 6 From $IRIS(PR_C)$ to the read/write model with $C$ : general case

This section presents a simulation of executions of the read/write model equipped with a failure detector of the class  $C$  in the model  $IRIS(PR_C)$ . The simulation does not depend on the failure detector class  $C$ , provided that an algorithm that emulates a failure detector of the class  $C$  in the  $IRIS(PR_C)$  model is given (Examples of such emulation have been described in Section 5 for the three failure detector classes  $\diamond\mathcal{S}_x$ ,  $\diamond\psi^y$  and  $\Omega^z$ ).

**Preliminaries** The aim is to establish that any task  $T$  solvable in the read/write model equipped with a failure detector of the class  $C$  is also solvable in the corresponding  $IRIS(PR_C)$  model. Thus, the simulation takes as parameter a read/write algorithm  $\mathcal{A}$  that solves a task  $T$  and an emulation  $\mathcal{E}$  of a failure detector of the class  $C$ .  $\mathcal{E}$  is an iterated algorithm that emulates the output of the failure detector. The simulation relies on the original simulation of the read/write model in the iterated model  $IIS$  [9] and on a recent improvement by Gafni and Rajsbaum [23].

Without loss of generality, we assume that algorithm  $\mathcal{A}$  is a full information protocol in normal form, as described in Figure 12. The state of process  $p_i$  is stored in the variable  $state_i$ , which is initialized with the input of  $p_i$  for the task.  $dec_i$  is a special write once variable intended to store the decision value of  $p_i$ .  $p_i$  queries its local failure detector module by reading the variable  $FD_i$ . In order to obtain a decision, process  $p_i$  enters an infinite loop. In each iteration, the failure detector is queried and the value returned is appended to the state of  $p_i$ .  $p_i$  then writes its entire state in its register and takes a snapshot. This snapshot constitutes the new state of  $p_i$ . Note that it includes its previous state. Finally, if  $p_i$  has not yet decided ( $dec_i = \perp$ ) but its current state allows deciding, it

```

init  $state_i \leftarrow input; dec_i \leftarrow \perp; k \leftarrow 0.$ 

repeat  $k \leftarrow k + 1;$ 
     $fd_i \leftarrow FD_i; state_i \leftarrow state_i \cdot fd_i;$  %  $k$ th failure detector query %
     $SM[i].write(state_i);$  %  $k$ th write %
     $state_i \leftarrow SM.snapshot();$  %  $k$ th read %
    if  $dec_i = \perp \wedge candecide(state_i)$  then  $dec_i \leftarrow \delta(s_i)$  end if
until  $dec_i \neq \perp.$ 

```

Figure 12: Full information normal form protocol with a failure detector.

decides by applying a function  $\delta$  to its state. Therefore, every full information protocol in normal form is completely determined by a predicate  $candecide$  and a decision function  $\delta$ . The predicate is defined over processes states; the decision function is defined only for states  $s$  such that  $candecide(s)$  is true.

We also suppose that an algorithm extracting a failure detector of the class  $C$  in  $IRIS(PR_C)$  is given. Again, we assume that the extraction algorithm can be written as a full information protocol in normal form, as described in Figure 13. Hence, such an extraction algorithm is completely determined by the pair of functions  $initfd$  and  $updatefd$ .  $initfd$  provides an initial value for the failure detector. The function  $updatefd$  outputs failure detector values, and takes as parameter the full information state of a process. The invocation of  $updatefd$  may not produce a fresh failure detector value each time it is invoked. Nevertheless, we assume that there exists a bound  $M$  on the number of rounds needed to actually update the simulated failure detector output. For example, in the simulation of  $\Omega^z$  in  $IRIS(PR_{\Omega^z})$ ,  $M = n$  (Figure 11).

```

init  $h_i \leftarrow i; r_i \leftarrow 0; FD_i \leftarrow initfd().$ 

repeat forever
     $r_i \leftarrow r_i + 1;$ 
     $h_i \leftarrow IS[r_i].WRITE\_SNAPSHOT(\langle i, h_i \rangle);$ 
     $FD_i \leftarrow updatefd(h_i);$ 
end repeat.

```

Figure 13: Normal form simulation of a failure detector of the class  $C$  in  $IRIS(PR_C)$ .

**The simulation** The algorithm extends the simulation given in [9] and improved in [23] to the context of the read/write model equipped with failure detectors. The simulation, described in Figure 14, solves a task  $T$  in the  $IRIS(PR_C)$  model, provided that an algorithm  $\mathcal{A}$  to solve  $T$  in the read/write model with  $C$  and an emulation  $\mathcal{E}$  of a failure detector of the class  $C$  are provided. In a nutshell, the simulation is the algorithm in [23] augmented with an helping mechanism.

The simulation takes as parameter  $input$  which is the process input to the task. Each process  $p_i$  maintains two variables a state  $state_i$  and an history  $h_i$ , as well as a write-once decision variable  $dec_i$  initialized to  $\perp$ . Variables  $state_i$  and  $h_i$  are updated following the normal form pattern in which algorithm  $\mathcal{A}$  and extraction  $\mathcal{E}$  are given.

To simulate write and snapshot operations, each process  $p_i$  maintains a vector  $c_i$  with one entry per process. Each entry has two fields denoted  $clock$  and  $val$  respectively. We denote  $c_i.clock$  and  $c_i.val$  the vectors formed by taking the  $clock$  and  $val$  field respectively of each entry. At each process, the execution of the normal form protocol  $\mathcal{A}$  proceeds by  $cycle$ . In its each  $k$ th cycle, the process queries its failure detector, writes in its register and performs a snapshot operation. Process  $p_i$  starts the simulation of its  $k$ th cycle by incrementing the clock entry of  $c_i[i]$  and placing in  $c_i[i].val$  the value to write in this cycle, that is its current state  $state_i$  (lines 19 – 21).  $c_i$  hence represents process  $p_i$  estimate of the state of the simulated shared memory. When  $c_i[j] = \langle v, k \rangle$ ,  $p_i$  knows that  $p_j$  is currently simulating its  $k$ th cycle and that the simulation of the previous cycles of  $p_j$  have been successfully completed.

Vectors  $c$  are partially ordered according to the  $clock$  fields.  $c \leq c'$  if and only if  $c[i].clock \leq c'[i].clock$  for every entry  $i$ , an  $c < c'$  if in addition,  $c[i].clock < c'[i].clock$  for some entry  $i$ .  $|c|$  denotes the sum of the clock entries in vector  $c$  and for a set  $s$  of vectors,  $top(s)$  denote the component-wise maximum of the vectors in  $s$ . Formally,

$$|c| = \sum_{1 \leq i \leq n} c[i].clock,$$

$$\forall 1 \leq j \leq n, top(s)[j] = c'[j] \text{ s.t. } c'[j].clock = \max\{c[j].clock, c \in s\}.$$

In every round, process  $p_i$  updates the emulated failure detector output (line 9) and updates its vector  $c_i$  by performing a  $top$  operation on the sets of vector that appear in its view (line 10). Recall that  $M$  is an upper bound on the number of rounds needed

by the extraction algorithm to produce a new failure detector output. Every  $M$  rounds,  $p_i$  checks if  $M|c_i| = r$ . When this condition is verified, the cycle currently simulated by  $p_i$  terminates, and  $c_i.val$  is the value returned by the snapshot of that cycle. Intuitively, vectors  $c$  satisfying  $M|c| = r$  are totally ordered and thus the values  $c.val$  are valid snapshots of the simulated shared memory. If it has not already decided,  $p_i$  then checks whether it can decide (line 17). If  $p_i$  is still undecided,  $c_i[i].clock$  is incremented, and the simulation of the next cycle starts (lines 18–22). Otherwise,  $p_i$  does not increment any more its clock entry, but keeps participating in the simulation. This is required for the correctness of the simulated run, as the failure detector extraction algorithm assumes that every correct process takes steps forever.

However, by not increasing their clock entries forever, correct processes do not impede undecided processes. Indeed, the simulation is non-blocking, as demonstrated by Lemma 6.4. To ensure wait-freedom, a simple helping mechanism is implemented by the variables  $CC_i$  and  $cc_i$ . Each vector  $c$  such that  $M|c| = r$  for some round  $r$  is a valid snapshot of the simulated shared memory. Hence, a process  $p_j$  simulating its  $k$ th cycle may adopt such a vector  $c$  as the output of its  $k$ th snapshot provided that  $c$  contains the  $k$ th write of  $p_j$ , i.e.,  $c[j].clock = k$ . Each process  $p_i$  therefore stores its last simulated snapshot in  $cc_i$  (lines 14–15).  $CC_i[j]$  then contains, according to the knowledge of  $p_i$ , the value of last snapshot completed by  $p_j$  (line 11). When  $p_i$  is simulating its  $k$ th cycle,  $p_i$  also checks every  $M$  rounds if its matrix  $CC_i$  contains a vector  $c$  such that  $c[i].clock = k$  (line 13) which it can adopt as a result of its  $k$ th snapshot.

```

simulation(input)
(1) init  $r_i \leftarrow 0$ ;  $dec_i \leftarrow \perp$ ;  $FD_i \leftarrow initfd()$ ;  $state_i \leftarrow input \cdot FD_i$ ;
(2)   for each  $j \neq i$  do  $c_i[j].clock \leftarrow 0$ ;  $c_i[j].val \leftarrow \perp$  end do;
(3)    $c_i[i].clock \leftarrow 1$ ;  $c_i[i].val \leftarrow state_i$ ;  $h_i \leftarrow i$ ;  $k \leftarrow 1$ ;
(4)   for each  $j$  do  $cc_i[j].clock \leftarrow 0$ ;  $cc_i[j].val \leftarrow \perp$  end do;
(5)    $CC_i[1..n][1..n] \leftarrow [[\perp, \dots, \perp], \dots, [\perp, \dots, \perp]]$ .

(6) repeat forever
(7)   repeat  $M$  times
(8)      $r_i \leftarrow r_i + 1$ ;  $sm_i \leftarrow IS[r_i].WRITE\_SNAPSHOT(\langle i, h_i, c_i, cc_i \rangle)$ ;
(9)     append  $\{ \langle j, h_j \rangle : j \in sm_i \}$  to  $h_i$ ;  $FD_i \leftarrow updatefd(h_i)$ ;
(10)     $c_i \leftarrow top(\{ \langle j, *, c_j, * \rangle \in sm_i \})$ ;
(11)    for each  $\langle j, h_j, c_j, cc_j \rangle \in sm_i$  do  $CC_i[j] \leftarrow cc_j$  end for
(12)  end repeat;
(13)  if  $|c_i|M = r_i \vee \exists j : CC_i[j][i].clock = k$  then %  $k$ th read %
(14)    if  $|c_i|M = r_i$  then  $state_i \leftarrow c_i.val$ ;  $cc_i \leftarrow c_i$ 
(15)      else  $state_i \leftarrow CC_i[j].val$ ;  $cc_i \leftarrow CC_i[j]$ 
(16)    end if;
(17)    if  $dec_i = \perp \wedge candecide(state_i)$  then  $dec_i \leftarrow \delta(state_i)$  end if;
(18)    if  $dec_i = \perp$  then
(19)       $k \leftarrow k + 1$ ;  $fd_i \leftarrow FD_i$ ; %  $k + 1$ th failure detector query %
(20)       $state_i \leftarrow state_i \cdot fd_i$ ; % input of the  $k + 1$ th write %
(21)       $c_i[i].val \leftarrow state_i$ ;  $c_i[i].clock \leftarrow c_i[i].clock + 1 (= k)$ 
(22)    end if
(23)  end if
(24) end repeat.

```

Figure 14: Simulating an algorithm for the read/write model +  $C$  in  $IRIS(PR_C)$  (code for  $p_i$ )

**Proof of correctness** The proof structure follows the one given in [23].  $c_i^r$  denotes the value of the vector  $c_i$  of process  $p_i$ , right after the execution of the *top* operation in round  $r$  (line 10), but before it is possibly modified in line 20. The first lemma states that the vectors corresponding to the values of the variables  $c_i$  right after the execution of line 10 of round  $r$  are totally ordered, for every  $r$ .

**Lemma 6.1.** *Let  $r$  a round number and  $p_i, p_j$  two processes.*

1. Assuming that  $p_i$  and  $p_j$  do not fail by the end of round  $r$ ,  $c_i^r \leq c_j^r \vee c_j^r \leq c_i^r$ ,
2. Assuming that  $p_i$  does not fail by the end of round  $r + 1$ ,  $c_i^r \leq c_i^{r+1}$ .

*Proof.* To prove the first part, let us observe that the views  $sm_i^r$  and  $sm_j^r$  that  $p_i$  and  $p_j$  get back by performing  $IS[r].WRITE\_SNAPSHOT()$  are related by containment. Without loss of generality, let us assume that  $sm_i^r \subseteq sm_j^r$ . Let  $\ell \in \{1, \dots, n\}$ . By definition of the top operation,  $c_i^r[\ell].clock = \max\{c_k[\ell].clock : \langle k, c_k, *, * \rangle \in sm_i^r\}$ . Since  $sm_i^r \supseteq sm_j^r$ ,  $c_j^r[\ell].clock \leq c_i^r[\ell].clock$ . Therefore  $c_j^r \geq c_i^r$ .

The second part follows from the self-inclusion property of immediate snapshot. In round  $r + 1$ ,  $sm_i$  contains an element  $\langle i, c, *, * \rangle$  where  $c = c_i^r$  or  $c > c_i^r$  (this occurs if  $p_i$  initiates a the simulation of a new cycle in round  $r$ ). Then, by the definition of the top operation, we have  $c_i^{r+1} \geq c_i^r$ .  $\square$

We say that a process is *undecided* at round  $r$  if it does not decide before round  $r$ , i.e., when the process invokes  $IS[r].WRITE\_SNAPSHOT()$   $dec_i = \perp$ . A process undecided at round  $r$  might however decide in round  $r$ . Next lemma presents two simple invariants on the values of the vector  $c_i$  of undecided processes.

**Lemma 6.2.** *Let  $r$  a round and  $p_i$  an undecided process at round  $r$ . Let  $c$  the value of  $c_i$  just before line 10 and  $c' (= c_i^r)$  its value after executing line 10. We have (1)  $c \leq c'$  and (2)  $r \leq M|c'|$ .*

*Proof.* Inequality (1) directly follows of the self-inclusion property of immediate snapshot and the definition of the top operation. We prove invariant (2) by induction. We first show that  $r \leq M|c_i^r|$  for each  $1 \leq r \leq M$ . Clearly, we have  $|c_i^{r-1}| \leq |c_i^r|$ . Initially,  $r = 0$  and  $|c_i| = 1$ , since  $c_i[j].clock = 0$  for each  $j \neq i$  and  $c_i[i].clock = 1$ . Hence, we have  $1 \leq |c_i^1|$ , and thus  $r \leq M \leq M|c_i^1|$  for each  $1 \leq r \leq M$ .

Let  $\alpha > 1$  and assume now that the invariant holds for each round  $\leq (\alpha - 1)M$ . Let  $r'$  such that  $(\alpha - 1)M < r' \leq \alpha M$ . In round  $r = (\alpha - 1)M$ , either  $r = M|c_i^r|$  or  $r < M|c_i^r|$ .

- In the first case, either  $p_i$  decides in round  $r$  (line 17) or  $c_i[i].clock$  is incremented. If  $p_i$  decides, the invariant is not required to hold in subsequent rounds. Otherwise, we have  $|c_i^r| + 1 \leq |c_i^{r'}|$ , from which we get  $(\alpha - 1)M + M \leq M|c_i^{r'}|$ . Consequently,  $r' \leq M|c_i^{r'}|$  for  $r' \leq \alpha M$ .
- In the second case,  $M(\alpha - 1) < M|c_i^r|$ . Therefore,  $\alpha \leq |c_i^r|$ , and as  $|c_i^r| \leq |c_i^{r'}|$  for every  $r \leq r'$ ,  $M\alpha \leq M|c_i^{r'}|$ . We conclude that  $r' \leq M|c_i^{r'}|$  for every  $(\alpha - 1)M \leq r' \leq \alpha M$ .  $\square$

An immediate consequence of Lemma 6.1 is that in each round  $r$  at most one vector  $c$  satisfies  $M|c| = r$ .

**Lemma 6.3.** (Simultaneity [23]) *Let  $r$  a round in which the condition  $M|c_i| = r$  is true for some process  $p_i$ . There exists a unique vector denoted  $c^r$  such that  $M|c_i^r| = r \implies c_i^r = c^r$ . Moreover, for each undecided process  $p_k$ ,  $c_k^r \geq c^r$  assuming that  $p_k$  does not fail by the end of round  $r$ .*

*Proof.* Let  $r$  a round in which the condition  $M|c| = r$  (line 14) is true for some process. Let  $p_i, p_j$  two processes such that  $M|c_i^r| = r$  and  $M|c_j^r| = r$ . By Lemma 6.1,  $c_i^r \leq c_j^r$  or  $c_j^r \leq c_i^r$ . Assume without loss of generality that  $c_i^r \leq c_j^r$ . Hence, for every entry  $\ell$ ,  $c_i^r[\ell].clock \leq c_j^r[\ell].clock$ . Therefore, as  $\sum_{\ell} c_i^r[\ell].clock = |c_i^r| = |c_j^r| = \sum_{\ell} c_j^r[\ell].clock$ ,  $c_i^r[\ell].clock = c_j^r[\ell].clock$  for every entry  $\ell$ . To conclude, observe that  $c[\ell].clock = c'[\ell].clock \implies c[\ell].val = c'[\ell].val$ . This completes the proof of the first part of the Lemma.

Let  $p_k$  a process that has not decided by the end of round  $r - 1$ . By Lemma 6.2(2)  $r \leq M|c_k^r|$ . As  $r = M|c^r|$ ,  $|c^r| \leq |c_k^r|$ . Moreover,  $c^r = c_i^r$  for some process  $p_i$ , and  $c_i^r \leq c_k^r \vee c_k^r \leq c_i^r$  by Lemma 6.1. We thus conclude that  $c^r \leq c_k^r$ , since  $|c| \leq |c'| \implies c' \not\prec c$ .  $\square$

As round numbers keep increasing, Lemma 6.2(2) implies the following property. An operation *completes* in round  $r$  if the simulation of a cycle terminates in round  $r$ . More precisely, the  $k$ th cycle of process  $p_i$  completes in round  $r$  if the condition of line 13 is satisfied at process  $p_i$  in round  $r$ .

**Lemma 6.4.** (Non-blocking progress [23]) *Let  $r$  a round in which some correct process is undecided. There exists a round  $r' \geq r$  in which some process completes its operation.*

*Proof.* Let  $p_j$  a correct undecided at round  $r$ . Let  $S$  the largest snapshot returned in round  $r$  (as snapshots are ordered by containment,  $S$  is well defined) and  $cm = top(S)$ . By lemma 6.1 and the definition of the top operation,  $|c_j^r| \leq |cm|$  for each process  $p_j$  that has not failed prior to round  $r$ . Assume for contradiction that no process completes an operation after round  $r$ . Hence, after round  $r$ , the *clock* fields are never incremented. Therefore, for every round  $r' \geq r$ ,  $|c_j^{r'}| \leq |cm|$ . However, by Lemma 6.2(2), we have  $\forall r', r' \leq M|c_i^{r'}|$ : a contradiction.  $\square$



We next show that every simulated execution is a valid execution of the read/write model equipped with a failure detector of the class  $\mathcal{C}$  executing algorithm  $\mathcal{A}$ . To do so, we prove that simulated operations (i.e., write, snapshot and failure detector queries) can be linearized. Following [23], operations are linearized in rounds; an operation linearized in round  $r$  precedes in the linearization order every operation linearized in round  $r'$ , for every  $r' > r$ . When several operations are linearized in the same round, we define how they are ordered.

Let  $\mathcal{R}$  the set of rounds for which there exists some process  $p_i$  such that  $M|c_i^r| = r$ . Intuitively,  $\mathcal{R}$  is the set of rounds at which the simulation produces valid output for the simulated snapshot operation. Formally,

$$r \in \mathcal{R} \iff \exists p_i : M|c_i^r| = r.$$

Per Lemma 6.3, every round  $r \in \mathcal{R}$  is uniquely associated with a vector denoted  $c^r$ .

**Lemma 6.5.** *Let  $r < r' \in \mathcal{R}$ .  $c^r < c^{r'}$ .*

*Proof.* Let  $r_i < r_j \in \mathcal{R}$  such that  $\forall r_i < r < r_j, r \notin \mathcal{R}$ . By definition of  $\mathcal{R}$ , there exists two processes  $p_i, p_j$  such that  $c_i^{r_i} = c^{r_i}$  and  $c_j^{r_j} = c^{r_j}$ . By Lemma 6.1(1), the vectors  $c_i^{r_i}$  and  $c_j^{r_j}$  are ordered. We consider two cases, according to the relative order of the two vectors.

- $c_j^{r_j} \geq c_i^{r_i}$ . By Lemma 6.1(2) for every round  $r > r_i$ ,  $c_j^r \geq c_i^r$ . Therefore, as  $c^{r_j} = c_j^{r_j}$ ,  $c_i^{r_i} \leq c_j^{r_j}$ .
- $c_j^{r_j} < c_i^{r_i}$ . Let  $S$  the immediate snapshot obtained by  $p_i$  in round  $r_i$  and  $L$  the set of processes that obtain immediate snapshots smaller than  $S$ . Let  $U$  the complement of  $L$ : the set of processes that have not failed by the end of round  $r_i$  and obtain immediate snapshots larger than or equal to  $S$  in round  $r$ .

Note that for each process  $p_\ell \in L$ ,  $c_\ell^{r_i} < c_i^{r_i}$  and  $p_j \in L$ . Therefore, it follows from Lemma 6.3 that each process included in  $U$  has decided before round  $r_i$ . After it has decided, a process no longer increments its clock entry in its vector  $c$  (at line 18). In round  $r_j$ , we have  $c_j^{r_j} = c^{r_j}$ , with  $|c^{r_j}| > |c^{r_i}| > |c_j^{r_i}|$ . Hence, in some round  $r$  between  $r_i$  and  $r_j$ , the immediate snapshot of  $p_j$  must include a vector  $c$  such that  $c$  is the vector of some process  $p$  included in  $U$  or  $c \geq c_u$  where  $p_u \in U$ . In both case, this implies that  $c_j^r \geq c_i^{r_i}$ , and thus by Lemma 6.1(2)  $c_j^{r_j} \geq c_i^{r_i}$ .

We have shown that  $c_i^{r_i} \leq c_j^{r_j}$ . Since  $r_i < r_j$ , we have  $|c^{r_i}| < |c^{r_j}|$  and thus  $c_i^{r_i} < c_j^{r_j}$ .  $\square$

We show by induction that each simulated snapshot operation that completes returns a vector  $c^r.val$  where  $r \in \mathcal{R}$ .

**Lemma 6.6.** *Assume that the simulation of the  $k$ th snapshot operation by  $p_i$  completes in round  $r$ . There exists  $r' \in \mathcal{R}$ ,  $r' \leq r$  such that the value returned by this operation is  $c^{r'}.val$ .*

*Proof.* Suppose that the lemma holds for every round  $< r$  and that the simulation of the  $k$ th snapshot of  $p_i$  completes in round  $r$ . By the code, the condition of line 13 is satisfied for process  $p_i$ . There are two cases to consider:

- $M|c_i^r| = r$  (line 14). In this case,  $r \in \mathcal{R}$ , and the  $k$ th snapshot of  $p_i$  is  $c^r.val$ ;
- $\exists j : CC_j[j][i].clock = c_i[i].clock = k$ . In that case, the  $k$ th snapshot of  $p_i$  is  $CC_j[j].val$ . Note that  $CC_j[j]$  contains  $\perp$  or a vector  $c$  such that  $c.val$  is the value returned by a simulated snapshot of  $p_j$  completed in some round  $< r$  (lines 14–15 and line 11). Therefore, by the induction hypothesis,  $c = c^{r'}$  for some round  $r' : r' < r \wedge r \in \mathcal{R}$ .

In the first round, each entry of  $CC_i$  is equal to  $[\perp, \dots, \perp]$ . By the analysis of the first case above, the lemma thus holds for  $r = 1$ .  $\square$

We now define the linearization of simulated operation. A simulated snapshot operation  $op$  that completes can be uniquely associated to a round  $r$  in  $\mathcal{R}$  by Lemma 6.6. We linearize each snapshot in its associated round. More precisely, if  $c$  is the value returned by a snapshot operation  $op$ , there exists a unique  $r \in \mathcal{R}$  such that  $c^r = c$  by Lemma 6.6.  $op$  is then linearized in round  $r$ . A simulated write is linearized right before the first snapshot that includes it. A failure detector query by process  $p_i$  is linearized in the round in which the corresponding instance of  $fd_i \leftarrow \text{FD}_i$  (line 19) is performed. Finally, when several operations are linearized in same round, snapshot are ordered first, followed by failure detector queries and then writes. Several operations of the same type linearized in the same round are arbitrary ordered.

More formally, for each round  $r \in \mathcal{R}$ , let  $S[r]$  the set of simulated snapshot operations that return  $c^r.val$ . Similarly, for  $r \in \mathcal{R}$ ,  $W[r]$  is the set of write operations defined as follows. The  $k$ th write operation of  $p_i$  is included in  $W[r]$  if and only if  $c^r[i].clock = k$  and  $\forall r' < r$ ,  $c^{r'}[i].clock < k$ . Let  $c$  the vector such that  $c.val$  is the output of the  $k$ th snapshot of  $p_i$ . By the code  $c[i].clock = k$  and by Lemma 6.6,  $c = c^r$  for some  $r \in \mathcal{R}$ . Therefore, each completed write operation is included in some set  $W[r]$ . Finally, for

each round  $r$  (not necessarily in  $\mathcal{R}$ ), let  $Q[r]$  the set of failure detector queries that are simulated in round  $r$ . The  $k$ th failure detector query of  $p_i$  is simulated in round  $r$  if the  $k$ th instance of  $fd_i \leftarrow \text{FD}_i$  (line 19) performed by  $p_i$  occurs in round  $r$ . In each round  $r$ , the operation included in  $W[r] \cup S[r] \cup Q[r]$  are linearized whenever  $W[r], S[r]$  or  $Q[r]$  is defined. Write operations are linearized first, followed by snapshots and finally failure detector queries. Operations of the same type linearized in the same round are ordered arbitrarily.

The discussion above is summarized by the following lemma where, abusing notations,  $X[r]$ ,  $X$  being  $S, W$  or  $Q$ , is the operations included in set  $X[r]$  ordered arbitrarily if  $X[r]$  is defined and  $\emptyset$  otherwise.

**Lemma 6.7.** (Linearizability) *Consider a finite execution of the simulation protocol that consists in  $r_e$  rounds.  $\sigma = W[1], S[1], Q[2], W[2], S[2], \dots$  is a valid linearization of some execution of  $\mathcal{A}$  in the read/write model equipped with  $C$ .*

*Proof.* Each operation whose simulation completes is linearized in  $\sigma$ . It follows from Lemma 6.5, and the way write operations are ordered in  $\sigma$  that each snapshot contains the values written by the last write operations that precede it in  $\sigma$ .

By the correctness of the algorithm that simulates a failure detector of the class  $C$  in  $\text{IRIS}(\text{PR}_C)$ , the results of the failure detector queries are a valid history with respect to the specification of the class  $C$  and the failure pattern of the  $\text{IRIS}(\text{PR}_C)$  run. If a process  $p_i$  fails in round some round  $r$ , no operation of that process starts after that round, and the failure detector values obtained after that round are compatible with the failure of  $p_i$ . Reciprocally, if  $p_i$  does not fail, the failure detector values are compatible with that fact by the correctness of the extraction algorithm. It might however be the case that  $p_i$  does not complete a write or snapshot operation, although it is correct. As the execution we consider is finite and asynchronous, it is possible that  $p_i$  does not fail and does not complete a write or snapshot operation.  $\square$

To argue about the correctness of the simulation, it remains to show that every correct process eventually obtains a decision value (at line 17).

**Lemma 6.8** (Termination). *Every correct process eventually decides.*

*Proof.* Let us consider an infinite execution of the simulation. Let  $T$  the set of correct processes. Define the relation  $\rightsquigarrow$  between correct processes as follows: for every  $p_i, p_j \in T$ ,  $p_i \rightsquigarrow p_j$  if and only if  $p_i$  “sees”  $p_j$  in infinitely many rounds. More precisely,

$$p_i \rightsquigarrow p_j \iff \text{the set } \{r : j \in \text{sm}_i^r\} \text{ is infinite.}$$

Let  $\rightsquigarrow^*$  the transitive closure of  $\rightsquigarrow$ :

$$p_i \rightsquigarrow^* p_j \iff \exists p_{i1} = p_i, p_{i2}, \dots, p_{ix} = p_j : p_{i1} \rightsquigarrow p_{i2}, p_{i2} \rightsquigarrow p_{i3}, \dots, p_{i(x-1)} \rightsquigarrow p_{ix}$$

and  $\text{seen}(p_i)$  the set of processes that  $p_i$  sees directly or indirectly infinitely many often:

$$\text{seen}(p_i) = \{p_j : p_i \rightsquigarrow^* p_j\}.$$

Assume for contradiction that there exists a correct process that never decides. Let  $T_d$  and  $T_u$  be the set of correct process that decide and do not decide respectively. Per lemma 6.4, and the fact that the code of the simulated algorithm  $\mathcal{A}$  is in normal form, at least one process  $p_u \in T_u$  completes infinitely many write/snapshot operations. The proof relies on the following claim:

*Claim C1.* For each correct process  $p_i \in \text{seen}(p_u)$ ,  $p_i$  either decides or completes infinitely many write/snapshot operations.

Claim C1 tell us that every process seen directly or indirectly infinitely many often by  $p_u$  either decides or simulates infinitely many write/snapshot operations. Suppose for contradiction that the set  $\text{seem}(p_u)$  includes a process  $p_i$  that never decides. Consider now a process  $p_j \notin \text{seen}(p_u)$ . Note that this may happen even if  $p_j$  is a correct process. However,  $p_u$  and every process in  $\text{seen}(p_u)$  cannot distinguish the considered execution from an execution in which  $p_j$  fails. In particular, by the correctness of the extraction algorithm, the output of the simulated failure detector at the process included in  $\text{seen}(p_u)$  must be consistent with the failure of  $p_j$ . Hence, we simulate an execution of  $\mathcal{A}$ , with set of correct processes  $\text{seen}(p_u)$ , where every correct process performs infinitely many operation or decides and in which a correct process never decides. This execution is a valid execution of  $\mathcal{A}$  in the read/write model equipped with a failure detector of the class  $C$ , as every finite prefix is linearizable (Lemma 6.7). This thus contradicts the correctness of  $\mathcal{A}$ , which requires that every correct process decides in each execution of  $\mathcal{A}$ .

*Proof of the Claim C1.* Let  $p_i \in \text{seen}(p_u)$  an arbitrary correct process. For each write/snapshot operation completed by  $p_u$  there, there exists a vector  $c$  and a round  $r \in \mathcal{R}$  such that  $c^r = c$  and  $c^r.\text{val}$  is the value returned by the snapshot operation. This vector is either the value of the variable  $c_u$  at round  $r$  (if the operation completes at line 14) or the value of the variable  $c_j$  of some other process (if the operation completes at line 15). In the last case,  $p_u$  is helped by  $p_j$  to complete its operation. As  $p_u$  completes

infinitely many snapshot operations, there exists a process  $p_\ell$  such that, infinitely often, the value returned by these snapshot operations is the value  $c_\ell.val$  of the variable  $c_\ell$  of process  $p_\ell$ . Note that  $p_\ell$  is not necessarily distinct from  $p_u$ . Clearly,  $p_\ell \in seen(p_u)$  and  $p_u \in seen(p_\ell)$ , as in order that the  $k$ th snapshot of  $p_u$  is  $c_\ell.val$ ,  $p_\ell$  has to perform  $c_\ell[u].clock \leftarrow k$ .

Let  $r \in \mathcal{R}$  such that  $M|c_\ell^r| = r$ . By definition of  $p_\ell$ , there are infinitely many such rounds. As  $p_i$  does not decide  $M|c_i^r| \geq r$  (Lemma 6.3). Moreover, since  $p_i$  completes finitely many operations, we have  $M|c_i^r| > r$ , for large enough  $r$  which implies that  $sm_i^r \supseteq sm_\ell^r$ . Hence,  $p_i \overset{*}{\rightsquigarrow} p_\ell$ , and as the relation  $\overset{*}{\rightsquigarrow}$  is transitive and because  $p_\ell \overset{*}{\rightsquigarrow} p_u$  and  $p_u \overset{*}{\rightsquigarrow} p_i$ , we get  $p_\ell \overset{*}{\rightsquigarrow} p_i$ .

When  $p_i$  initiates its  $k$ th operation, it sets  $c_i[i].clock$  to  $k$  (line 21). Suppose the last cycle initiated by  $p_i$  is its  $k$ th. As  $p_i \in seen(p_\ell)$ , and for every vector  $c$ ,  $c[i].clock > k$  can occur only after the  $k$ th operation of  $p_i$  is completed, there is a round after which we always have  $c_\ell[i].clock = k$ . It then follows from the fact that the predicate  $M|c_\ell^r| = r$  is satisfied infinitely often that the vector  $cc$  of process  $p_\ell$  is eventually such that  $cc[i].clock = k$  (lines 14). Since  $p_\ell \in seen(p_i)$ ,  $p_i$  thus eventually observes a vector  $cc$  such that  $cc[i].clock = k$  and completes its  $k$ th operation at line 15: a contradiction. *End of the proof of claim C1.*  $\square$

**Computational equivalence between the read/write model with C and the  $IRIS(PR_C)$  model** The correctness of the simulation implies the following main theorem.

**Theorem 6.9.** *Let  $C$  be a failure detector class and  $IRIS(PR_C)$  be the corresponding iterated restricted immediate snapshot model. Let us assume that there are two algorithm  $\mathcal{T}_1$  and  $\mathcal{T}_2$  such that (1)  $\mathcal{T}_1$  simulates  $IRIS(PR_C)$  in the read/write model equipped with a failure detector of class  $C$  and, (2)  $\mathcal{T}_2$  emulates a failure detector of the class  $C$  in  $IRIS(PR_C)$ . A task  $T$  is solvable in  $IRIS(PR_C)$  if and only if it is solvable in the read/write model equipped with a failure detector of class  $C$ .*

*Proof.* Let us first consider the  $\Rightarrow$  direction. Let  $\mathcal{A}$  be an algorithm that solves  $T$  in the  $IRIS(PR_C)$  model. It follows that by stacking  $\mathcal{A}$  on top of the algorithm  $\mathcal{T}_1$  we obtain an algorithm that solves  $T$  in the read/write model equipped with a failure detector of the class  $C$ .

Let us now consider the  $\Leftarrow$  direction. Let  $\mathcal{A}$  be an algorithm that solves the task  $T$  in the read/write model equipped with a failure detector of the class  $C$ . Without loss of generality, we can assume that  $\mathcal{A}$  is written in normal form. The simulation describes in Figure 14 then provides an algorithm that solves  $T$  in the  $IRIS(PR_C)$  model. By Lemma 6.8, every correct process decides. Lemma 6.7, applied to the prefix of the simulated execution that ends after the last decision, then implies that the decisions obtained are valid decisions for  $T$ .  $\square$

## 7 Benefiting from the $IRIS$ model

This section presents several applications of our previous results.

### 7.1 Characterizing tasks solvable with failure detectors

The previous characterization in the  $IRIS(PR_C)$  framework of the synchrony achievable by the failure detector families  $(\diamond\mathcal{S}_x)_{1 \leq x \leq n}$ ,  $(\Omega^z)_{1 \leq z \leq n}$ , and  $(\diamond\psi^y)_{1 \leq y \leq n}$  can be used to study their computational power in the read/write shared memory model. As a particular example, we have the following.

**Theorem 7.1.** *The  $k$ -set agreement problem is not solvable in a read/write shared memory system with a failure detector of the class  $\Omega^z$  if  $k < z$ .*

This result was proved in [38] by reduction to a similar impossibility for  $\{\diamond\mathcal{S}_x\}$  proved in [27] using combinatorial topology techniques from [30]. A simple proof of the theorem is described next.

Consider the  $IRIS(PR_{\Omega^z})$  model. Notice that all runs of the IIS model where are most  $z$  processes are correct (and the others crash initially) are runs of the  $IRIS(PR_{\Omega^z})$  model. This is because these processes do not see a write by any other process (i.e., their views are always contained in a set  $L$  of size at most  $z$ , as required by property  $PR_{\Omega^z}$  property). But it is known that in the IIS model of  $z$  processes,  $(z - 1)$ -set agreement is not solvable [9] (because it is similar to a wait-free system of  $z$  processes).

More generally, thanks to Theorem 6.9, the  $IRIS(PR_C)$  allows characterizing the agreement tasks wait-free solvable in the read/write model enriched with a failure detector of the class  $C$ .

### 7.2 $t$ -resiliency

We may want to solve tasks  $t$ -resiliently, i.e., tolerating only  $t < n$  failures. A task  $T$  has a  $t$ -resilient solution if there exists an algorithm  $\mathcal{A}$  that solves  $T$  in every execution in which the number of failures is  $\leq t$ . That is, in every such execution, each non-faulty process outputs, and the outputs are valid outputs for the execution inputs according to the task specification. In execution with

strictly more than  $t$  failures, however, nothing is required. In particular, non-faulty processes may produce arbitrary outputs or may not output at all. The question of whether a task has a  $t$ -resilient solution or not may be studied via an iterated model as explained below. We show that  $t$ -resilient computability is captured by a class of failure detector, namely, the class  $\psi^{n-t}$  which is the perpetual counterpart of the class  $\diamond\psi^{n-t}$  defined in Section 2.1. More precisely, we establish that every task  $T$  has a  $t$ -resilient solution if and only if  $T$  is (wait-free) solvable in the asynchronous read/write model equipped with a failure detector of the class  $\psi^{n-t}$ . Moreover, every failure detector class  $\psi^y$  induces a iterated model denoted  $IRIS(PR_{\psi^y})$  of equivalent computational power. The question of whether a task has a  $t$ -resilient solution or not thus reduces to the solvability of the task in the iterated model  $IRIS(PR_{\psi^{n-t}})$ . This is summarized by the following theorem.

**Theorem 7.2.** *Let  $T$  a task. The following propositions are equivalent:*

1.  $T$  is  $t$ -resiliently solvable in the read/write model;
2.  $T$  is wait-free solvable in the read/write model equipped with a failure detector of the class  $\psi^{n-t}$ ;
3.  $T$  is wait-free solvable in the iterated model  $IRIS(PR_{\psi^{n-t}})$ .

The remainder section is devoted to prove this theorem.

**The family of failure detector classes  $(\psi^y)_{1 \leq y \leq n}$**  As indicated before, the failure detector class  $\psi^y$  is the perpetual counterpart of the failure detector class  $\diamond\psi^y$  [38]. A failure detector of the class  $\psi^y$  outputs integers, which are estimates of the current number of failed processes. More precisely, the outputs satisfy the following properties, where  $f$  is the actual number of failures that occur in the execution:

1. *Perpetual safety.*  $\forall \tau, n - y \leq NBC_i^\tau \leq \max(f, n - y)$ , where  $f$  is the number of failures that have occurred before by time  $\tau$ ;
2. *Eventual accuracy.* There is a time from which  $NBC_i = \max(n - y, f)$  at each correct process  $p_i$ .

The failure class  $\psi^y$  induces an iterated model  $IRIS(PR_{\psi^y})$  where the property  $PR_{\psi^y}$  is the perpetual counterpart of the property  $PR_{\diamond\psi^y}$  ( $f$  is the total number of failures in the execution):

$$PR_{\psi^y} \equiv \forall r' \geq 0 : ((i - 1 = (r' - 1) \pmod n) \wedge (sm_i^{r'} \neq \emptyset)) \implies |sm_i^{r'}| \geq n - \max(n - y, f)$$

The  $IRIS(PR_{\psi^y})$  model can be simulated in the read/write model equipped with a failure detector of the class  $\psi^y$ . The simulation is the same as the simulation of  $IRIS(PR_{\diamond\psi^y})$  (Figure 7) in the read/write model equipped with  $\diamond\psi^y$ . Reciprocally, one can check that the algorithm describes in Figure 10 emulates a failure detector of the class  $\psi^y$  when the underlying  $IRIS$  model satisfies the property  $PR_{\psi^y}$ . By Theorem 6.9, we thus obtain :

**Lemma 7.3.** *A task  $T$  is wait-free solvable in the read/write model equipped with a failure detector of the class  $\psi^y$  if and only if it is solvable in  $IRIS(PR_{\psi^y})$ .*

**Solvability with  $\psi^{n-t}$  implies  $t$ -resiliency** Suppose that task  $T$  is solvable in the read/write model equipped with a failure detector of the class  $\psi^{n-t}$ . This means that there exists an algorithm  $\mathcal{A}$  using failure detector  $\psi^{n-t}$  to solve  $T$ . Note that  $\mathcal{A}$  is wait-free, i.e., it tolerates an arbitrary number of failures. By the definition of the class  $\psi^{n-t}$ , every query to the failure detector returns an integer  $x$  such that  $n - (n - t) = t \leq x \leq \max(n - (n - t), f) = \max(t, f)$  where  $f$  is the total number of failures in the execution. Hence, in every execution in which the number of failures  $f$  is at most  $t$ , every failure detector query returns  $t$ . We use this fact to show that  $T$  can be solved  $t$ -resiliently in the read-write model without failure detector.

Let  $\mathcal{A}'$  the algorithm identical to  $\mathcal{A}$  except that every query to  $\psi^{n-t}$  is emulated by always returning  $t$ . Since  $t$  is a valid output of a query to  $\psi^{n-t}$  in every execution in which no more than  $t$  processes fail, each non-faulty process must produce a valid output according to the specification of  $T$  in every execution of  $\mathcal{A}'$  where no more that  $t$  processes fail. Therefore  $\mathcal{A}'$  is a  $t$ -resilient solution for  $T$ .

**$t$ -resiliency implies solvability with  $\psi^{n-t}$**  The remaining of this section is devoted to prove the reverse direction: if task  $T$  is  $t$ -resilient solvable,  $T$  is wait-free solvable in the read/write model equipped with a failure detector of the class  $\psi^{n-t}$ . We consider tasks  $T = (\mathcal{I}, \mathcal{O}, \Delta)$  that satisfy the following natural *monotony* condition:

**Definition 7.4** ([18]). *Let  $T = (\mathcal{I}, \mathcal{O}, \Delta)$  a task. Task  $T$  satisfies the monotony property if and only if for every  $(u, v) \in \mathcal{I} \times \mathcal{O}$  and  $u' \in \mathcal{I}$  such that  $(u, v) \in \Delta$  and  $u \subset u'^A$ , there exists  $v' \in \mathcal{O}$  such that  $v' \in \Delta(u')$  and  $v \subset v'$ .*

<sup>4</sup>For two  $n$ -vectors  $v, v'$ ,  $v \subseteq v'$  if and only if  $v[i] \neq v'[i] \implies v[i] = \perp$  for every  $i \in \{1, \dots, n\}$ .



Let  $T = (\mathcal{I}, \mathcal{O}, \Delta)$  denote a task that is  $t$ -resilient solvable. Suppose that we are given an algorithm  $\mathcal{A}$  that solves  $T$   $t$ -resiliently. Our goal is to construct an algorithm  $\mathcal{A}'$  that solves  $T$  independently of the number of failures, but with the help of a failure detector of the class  $\psi^{n-t}$ . On one hand, as long as the number of failures  $f$  remains bounded from above by  $t$ , the behavior of algorithm  $\mathcal{A}'$  might be identical as algorithm  $\mathcal{A}$ . On the other hand, when  $f > t$  a failure detector of the class  $\psi^{n-t}$  provides accurate information on the number of failures. In fact, one can show [38] that when the number of failures becomes larger than  $t$ ,  $\psi^{n-t}$  has the same power as a perfect failure detector  $\mathcal{P}$  [12].

Suppose we are given a perfect failure detector  $\mathcal{P}$ . It is then possible to implement consensus objects. Relying on these objects, task  $T$  can then be solved as follows. The construction used an array of  $n$  consensus objects denoted  $CONS[1..n]$ . Consensus object  $CONS[\ell]$  is used by the processes to agree on a valid pair of input and output vectors for  $T$  of size  $\ell^5$ . In more details, each process accesses in order the consensus objects  $CONS[1], CONS[2], \dots$  until one object returns a pair that contains an output for itself. It then decides that output. Let  $(u^\ell, v^\ell)$  denote the pair returned by the  $\ell$ th consensus object. If  $v^\ell[i] = \perp$ , the pair proposed by process  $p_i$  to the  $\ell + 1$ th consensus object extends the pair  $(u^\ell, v^\ell)$  with the input of  $p_i$  and a valid output for  $p_i$ . More precisely, the pair  $(u, v)$  of size  $\ell + 1$  vectors proposed by process  $p_i$  is such that  $u^\ell \subset u, v^\ell \subset v, (u, v) \in \Delta, v[i] \neq \perp$  and  $u[i]$  is equal to the input of  $p_i$ . As task  $T$  is monotone (Definition 7.4) and by construction  $(u^\ell, v^\ell) \in \Delta$ , finding such a pair is always possible. As the size of the pair returned by the consensus objects is increasing, each non-faulty process eventually finds an output for itself. Moreover, by construction, the outputs are valid outputs with respect to  $\Delta$ .

These ideas are implemented in the algorithm described in Figure 15. Without loss of generality, we assume that a  $t$ -resilient algorithm  $\mathcal{A}$  for  $T$  is given as a full-information algorithm in normal form.  $\mathcal{A}$  is thus completely determined by a decision predicate *candecide* and a decision function  $\delta$ .

Each process first executes algorithm  $\mathcal{A}$  until the number of failures becomes larger than  $t$  (lines 1–9). An estimate of the current number of failures is provided by the underlying failure detector of the class  $\psi^{n-t}$  (line 3). Whenever a value larger than  $t$  is returned, the process stop executing  $\mathcal{A}$  and instead tries to obtain an output using consensus objects (procedure **DecideCons**, lines 11–23). By the perpetual safety property, a failure detector query may return a value larger than  $t$  only if the number of failures is above  $t$ . Therefore, when processes switch to the procedure **DecideCons**, the underlying failure detector is as powerful as a perfect failure detector, and the consensus-based approach for solving  $T$  can thus be implemented.

However, in the same execution, some processes may obtain an output for  $T$  via the execution of  $\mathcal{A}$  (at line 6) while other processes may decide in the procedure **DecideCons** (at line 23). We rely on adopt-commit objects [53] to guarantee that outputs for  $T$  obtained in both parts of the algorithm are consistent, i.e., the vector formed by the output of each process is a valid output according to the input vector of the execution and the relation  $\Delta$ .

An adopt-commit object  $AC$  supports a single one-shot operation denoted *propose()* that takes as parameter values from some arbitrary set  $V$ . An invocation *propose*( $v$ ) returns a pair  $(b, v')$  where  $b \in \{\text{adopt}, \text{commit}\}$  and  $v' \in V$ . The pair returned satisfy the following properties:

- *Termination*: Each invocation by a correct process terminates.
- *Validity*: If an invocation returns  $(b, v)$  then some process invoked  $AC.\text{propose}(v)$ .
- *Agreement*: If an invocation returns  $(\text{commit}, v)$ , then every invocation returns  $(*, v)$ .
- *Convergence*: If every invocation has the same input value  $v$ , then  $(\text{commit}, v)$  is the only pair that can be returned.

Adopt-commit objects can be wait-free implemented in the read/write model, e.g., [19, 53].

We associate with every process  $p_i$  an adopt-commit object denoted  $AC[i]$ . Only two values may be proposed to object  $AC[i]$ :  $val_i$ , the output for  $T$  that  $p_i$  may obtain by executing algorithm  $\mathcal{A}$  (line 5) or the special value *abort* that we assume is never a valid output for  $T$ . The former case occurs when  $p_i$  obtains an output for  $T$  by executing  $\mathcal{A}$  (line 5). The latter case occurs in the procedure **DecideCons** (line 12–15). By proposing *abort* to  $AC[i]$ , a process executing **DecideCons** attempts to prevent  $p_i$  from deciding an output obtained by executing  $\mathcal{A}$ .

Suppose that process  $p_i$  obtains output  $v$  by executing  $\mathcal{A}$  (line 4).  $p_i$  then invokes *propose*( $v$ ) and is allowed to decide this value only if the invocation returns  $(\text{commit}, v)$ . If this does not occur, some process has proposed the value *abort* to  $AC[i]$ . This follows from the fact that *abort* is the only value  $\neq v$  that can be proposed and the convergence property of adopt-commit. This means that some process has discovered that the number of failures is above  $t$ .  $p_i$  thus switches to **DecideCons** procedure to attempt to obtain an output for  $T$  (line 7).

Process  $p_i$  executing **DecideCons** initiates the procedure by proposing *abort* to each adopt-commit object  $AC[j]$  for  $j \neq i$  (line 13). As  $p_i$  may have first proposed a value to  $AC[i]$  in the first part of the algorithm,  $p_i$  does not propose a value to  $AC[i]$ . The goal is to avoid  $p_j$  to obtain an output in the first phase of the algorithm, or – if  $p_j$  has already decided – to learn the output of  $p_j$ . In

<sup>5</sup>The size of an input or output vector is the number of its non- $\perp$  entries.



more details,  $p_i$  maintains two local vectors  $in_i$  and  $out_i$  which are intended to store each process input and output respectively. If the `propose()` invocation to  $AC[j]$  returns  $(commit, v)$  or  $(adopt, v)$ , with  $v \neq abort$ ,  $v$  has been proposed to  $AC[j]$  and thus may be the output for  $T$  decided by  $p_j$ . In that case,  $p_i$  stores this value in  $out_i$ . If  $(commit, abort)$  or  $(adopt, abort)$  is returned, the entry  $out_i[j]$  is left to its initial value  $\perp$ . Notice that in that case,  $p_j$  cannot decide and output for  $T$  in the first part of the algorithm. Inputs values are obtained by taking a snapshot of the shared  $I$  (line 16), to which each process initially writes its input.

Two processes  $p_i$  and  $p_\ell$  executing **DecideCons** may obtain different responses from their `propose()` operation on the same object  $AC[j]$ . In particular,  $(adopt, v)$  and  $(adopt, abort)$  may be returned. Similarly,  $p_i$  and  $p_\ell$  may have different views of the processes inputs. Thus, the pairs of vectors  $in_i/out_i$  and  $in_\ell/out_\ell$  may differ. Observe however that  $in_i \in \Delta(out_i)$  and  $in_\ell \in \Delta(out_\ell)$  as the non- $\perp$  values in  $out_i$  and  $out_\ell$  are decisions in a execution of  $\mathcal{A}$  in with no more than  $t$  failures. Moreover, if entry  $j$  differs in  $out_i$  and  $out_\ell$  then  $p_j$  cannot have decided in the first part of the algorithm. On the contrary, if  $p_j$  does decide  $v$  in the first part of the algorithm, then  $out_i[j] = out_\ell[j] = v$ .

```

init  $AC[1..n]$  array of adopt-commit objects
       $I[1..n], SM[1..n]$  arrays of atomic registers initially  $[\perp, \dots, \perp]$ 
       $ACONS, CONS[1..n]$   $\psi^{n-t}$ -based consensus objects

solveT( $candecide, \delta, input_i$ )
(1)  $state_i \leftarrow input_i; dec_i \leftarrow \perp; I[i] \leftarrow input_i;$ 
(2) repeat
(3)    $SM[i].write(state_i); state_i \leftarrow SM.snapshot(); nbc_i \leftarrow NBC_i;$ 
(4)   if  $(nbc_i \leq t) \wedge (dec_i = \perp) \wedge candecide(state_i)$  then
(5)      $val_i \leftarrow \delta(state_i); (b, v) \leftarrow AC[i].propose(val_i);$ 
(6)     if  $(b, v) = (commit, val_i)$  then  $dec_i \leftarrow val_i; decide(dec_i)$ 
(7)       else DecideCons() endif
(8)   end if
(9) until  $nbc_i > t$ 
(10) if  $dec_i = \perp$  then DecideCons() end if

DecideCons()
(11)  $out_i \leftarrow [\perp, \dots, \perp]; in_i \leftarrow [\perp, \dots, \perp];$ 
(12) for each  $j \in \{1, \dots, n\} \setminus \{i\}$  do
(13)    $(b, v) \leftarrow AC[j].propose(abort)$ 
(14)   if  $v \neq abort$  then  $out_i[j] \leftarrow v$  endif
(15) end for
(16)  $s_i \leftarrow I.snapshot();$  for each  $\langle j, v \rangle \in s$  do  $in_i[j] \leftarrow v$  end do
(17)  $(in_i, out_i) \leftarrow ACONS.propose(in_i, out_i);$ 
(18) let  $\ell$  the number of non- $\perp$  entries in  $out_i;$ 
(19) while  $out_i[i] = \perp$  do
(20)    $in_i[i] \leftarrow input_i; \text{let } out' \in \mathcal{O} \text{ s.t. } (in_i, out') \in \Delta \wedge \forall j \neq i, out_i[j] = out'[j] \wedge out'[i] \neq \perp;$ 
(21)    $\ell \leftarrow \ell + 1; (in_i, out_i) \leftarrow CONS[\ell].propose(in_i, out');$ 
(22) end while
(23)  $dec_i \leftarrow out[i]; decide(dec_i);$ 

```

Figure 15:  $(\psi^{n-t})$ -based algorithm for task  $T$  (code for process  $p_i$ )

Every process  $p_i$  agree at line 17 on the same input/output pair by proposing  $in_i/out_i$  to the consensus object  $ACONS$ . Let  $in/out$  denote the pair agreed upon. By the observation above, if  $p_j$  decides in the first part of algorithm,  $in[j]$  contains the input of  $p_j$  and  $out[j]$  the value decided by  $p_j$ . Processes then obtain an output for  $T$  following the consensus-based approach explained earlier (lines 19-22). The only difference is that processes start with the input/output pair  $in/out$  that already contains  $\ell$  input/output values instead of the empty pair.

**Proof** Consider an execution of the algorithm described in Figure 15. We denote by  $f$  the number of failures in this execution.

**Lemma 7.5.** *Every correct process decides.*

*Proof.* We consider two cases according to the value of  $f$ .

- $f \leq t$ . In that case, for every process  $p_i$  and every time  $\tau$ ,  $nbc_i^\tau \leq t$  and thus no processes invoke **DecideCons**(). As every correct process writes to  $SM$  infinitely often, and no more than  $t$  failures occur, each correct process  $p_i$  eventually reaches a

state  $state_i$  for which the predicate  $candecide()$  (line 4) is verified. Let  $d$  the value returned by the decision function  $\delta$  applied to this state. Since no processes invoke **DecideCons**(), process  $p_i$  only accesses the adopt-commit  $AC[i]$ . It thus follows from the convergence property of adopt-commit that the propose() operation performed by  $p_i$  returns  $(commit, d)$ , from which we conclude that  $p_i$  decides (line 5).

- $f > t$ . In that case, there exists a time  $\tau$  after which we always have  $nbc_i = f$ . Assume for contradiction that some correct process  $p_i$  does not decide. By the previous observation and the code (line 9), process  $p_i$  eventually invokes **DecideCons** (at line 7 or line 10). By the termination property of adopt-commit, every invocation of propose() on adopt-commit object performed by  $p_i$  terminates. Note also that, as  $f \geq t$ , the  $\psi^{n-t}$ -based implementations of consensus are correct, and thus each propose() operation on the consensus objects  $ACONS$  and  $CONS[j]$ ,  $1 \leq j \leq n$  performed by  $p_i$  terminates.

By the code, process  $p_i$  decides (at line 23) if the  $out$  vectors obtained as a response of a propose() operation on a consensus object  $CONS[j]$  contains an output value for itself, i.e.,  $out[i] \neq \perp$ . In the **while** loop (lines 19–22), process  $p_i$  accesses in this order the consensus objects  $CONS[\ell], \dots, CONS[n]$  for some  $\ell > 0$ . Each object is accessed once by  $p_i$ , and the vector  $out^j$  returned by the  $j$ th consensus object (line 21) contains  $j$  non- $\perp$  entries. Hence,  $out^n[i] \neq \perp$  and  $p_i$  eventually decides: a contradiction. □

Next Lemma establishes that decisions at line 6 or 23 follow the specification of task  $T$ .

**Lemma 7.6.** *Decided values are valid with respect to the task specification and the input values.*

*Proof.* Denote by  $D_t$  and  $D_\psi$  the sets of processes that decide at line 6 and at line 23 respectively. We consider two cases:

- $D_\psi = \emptyset$ . Consider a finite prefix  $\sigma$  of the execution of the full information protocol in normal form  $\mathcal{A}$ . This prefix can be extended to an infinite execution with no more than  $t$  failures. Therefore, each value decided (if any) during  $\sigma$  is valid according to the specification of  $T$ , since  $\mathcal{A}$  is a  $t$ -resilient algorithm and when the value is decided no process can distinguish the current execution with an execution with no more than  $t$  failures.

In the first part of the algorithm (lines 1–10), each process executes the full information protocol in normal form  $\mathcal{A}$ , until it possibly discovers that more than  $t$  failures occur. By the observation above, every decision is thus valid according to the task specification.

- $D_\psi \neq \emptyset$ . Let  $L$  the index of the largest  $CONS$  object accessed in the procedure **DecideCons**. Let  $(I_L, O_L)$  the pair of input/output vector returned by this object. By the code, the input of every propose() operation performed on  $CONS[L]$  is valid a input/output pair according to the specification of  $T$ . Therefore,  $O_L \in \Delta(I_L)$ .

Suppose that some process  $p_i$  decides in the first part of the algorithm at line 6. Let  $o_i$  the value decided by  $p_i$ . By the code, the propose() operation performed by  $p_i$  to  $AC[i]$  at line 5 returns  $(commit, o_i)$ . Consider now a process  $p_j$  that executes **DecideCons**. Note that by the code  $p_j \neq p_i$ . By the agreement property of adopt-commit, the propose() operation performed by  $p_j$  on  $AC[i]$  returns  $(commit, o_i)$  or  $(adopt, o_i)$  (line 13). Thus we have  $out_j[i] = o_i$  when  $p_j$  accesses the consensus object  $ACONS$  (line 17). Since this holds for every process accessing the object, the input/output pair  $(I_a, O_a)$  returned by the object is such that  $O_a[i] = o_i$ . By code,  $O_a \subseteq O_L$ , from which we conclude that  $O_L[i] = o_i$ .

Consider now some process  $p_j$  that decides in the procedure **DecideCons** value  $o_j$  (line 23). By the code,  $p_j$  gets back an input/output pair  $(I_\ell, O_\ell)$  from an invocation of propose() to  $CONS[\ell]$ , where  $|O_a| \leq \ell \leq L$ . Since  $O_\ell \subseteq O_L$ ,  $O_L[j] = o_j$ .

Therefore, for each process  $p_j$  that decides, the  $j$ th entry of the vector  $O_L$  contains the value decided by  $p_j$ . Similarly, for each  $j \in \{1, \dots, n\}$ , if  $I_L[j] \neq \perp$  then  $I_L[j]$  is the input value for task  $T$  of process  $p_j$ . As  $O_L \in \Delta(I_L)$ , the decided values are thus valid according to the specification of  $T$  and the input value of the execution. □

**Lemma 7.7.** *Let  $T = (\mathcal{I}, \mathcal{O}, \Delta)$  a task that satisfies the monotony property and is  $t$ -resiliently solvable. The algorithm described in Figure 15 solves  $T$  in the read/write model equipped with a failure detector of the class  $\psi^{n-t}$ .*

*Proof.* Immediately follows from Lemma 7.5 and Lemma 7.6. □

### 7.3 $k$ -set agreement with limited-scope failure detectors

To illustrate the advantage of the  $IRIS(PR_C)$  framework when one is interested in lower bounds, this section gives a new proof of the lower bound for the  $k$ -set agreement problem. That lower bound, conjectured in [40], has been proved in [27] in the context of  $t$ -resilient message-passing systems, using techniques borrowed from combinatorial topology. The new proof is on the *wait-free* case ( $t = n - 1$ ) in the read/write model enriched with a failure detector of the class  $\diamond\mathcal{S}_{x,q}$ . Technically speaking, the problem is reduced to the question of the  $k$ -set agreement wait-free solvability. No topology notion is required.

**The families of failure detector classes  $\{\mathcal{S}_{x,q}\}$  and  $\{\diamond\mathcal{S}_{x,q}\}$**  The family  $(\diamond\mathcal{S}_{x,q})_{1 \leq x \leq n, 1 \leq q \leq x}$  extends the notion of limited scope failure detector to a system where the processes are partitioned into multiple disjoint clusters. There are  $q$  disjoint clusters denoted  $X_1, \dots, X_q$ , where  $|X_i| = x_i$ ,  $X = \bigcup_{1 \leq i \leq q} X_i$  and  $x = \sum_{i=1}^q x_i$ . Informally, there is a process that is never suspected in each cluster  $X_i$ . More specifically, the variable  $TRUSTED_i$  provided by a failure detector of the class  $\diamond\mathcal{S}_{x,q}$  contains the identities of the processes that are believed to be currently alive. When  $j \in TRUSTED_i$  we say “ $p_i$  trusts  $p_j$ .” By definition, a crashed process trusts all the processes. The failure detector class  $\diamond\mathcal{S}_{x,q}$  is defined by the following properties:

- **Strong completeness.** There is a time after which every faulty process is never trusted by every correct process.
- **Eventual weak  $(x, q)$ -accuracy.** There are  $q$  disjoint sets  $X_1, \dots, X_q$  of cumulatively  $x$  processes,  $q$  processes  $p_{\ell_1} \in X_1, \dots, p_{\ell_q} \in X_q$  and a (finite) time  $\tau$  such that each process of  $X_i$  trusts  $p_{\ell_i}$ .

The time  $\tau$ , the set  $X_1, \dots, X_q$  and the processes  $p_{\ell_i}$  are not explicitly known. Moreover, some or all processes of  $X_i$  may be faulty (A cluster  $X_i$  of faulty processes trivially satisfies  $(x, q)$ -accuracy).

Recall the following equivalent formulation of  $\diamond\mathcal{S}_{x,q}$  [38], assuming the local variable controlled by the failure detector is  $REPR_j$ .

- **Eventual  $(x, q)$ -common representative.** There are  $q$  disjoint sets  $X_1, \dots, X_q$  of cumulatively  $x$  processes,  $q$  processes  $p_{\ell_1} \in X_1, \dots, p_{\ell_q} \in X_q$ , and a (finite) time  $\tau$  after which, for any correct process  $p_j$ , we have  $j \in X_i \implies REPR_j = \ell_i$  and  $j \notin \bigcup_{1 \leq i \leq q} X_i \implies REPR_j = j$ .

Clearly, a failure detector that satisfies the previous property can be transformed into one of the class  $\diamond\mathcal{S}_{x,q}$  (define  $TRUSTED_i = \{REPR_i\}$ ). Conversely, one can easily extend the algorithm in [38] that transforms a failure detector of class  $\diamond\mathcal{S}_x$  into a failure detector satisfying the limited eventual common representative property to the context of the family  $(\diamond\mathcal{S}_{x,q})_{1 \leq x \leq n, 1 \leq q \leq x}$ .

**The lower bounds** The lower bounds established in [27] are on  $t$ -resilient asynchronous message-passing systems (i.e., systems prone to up to  $t$  process crashes). They are the following.

- If the system is equipped with  $\mathcal{S}_{x,q}$ , any  $k$ -set agreement protocol must satisfy  $t < k + x - q$  if  $q \leq k$  and  $t < x$  otherwise.
- If the system is equipped with  $\diamond\mathcal{S}_{x,q}$ , any  $k$ -set agreement protocol must satisfy  $t < \min(\frac{n}{2}, k + x - q)$  if  $q \leq k$  and  $t < \min(\frac{n}{2}, x)$  otherwise. (In the shared memory context, the requirement  $t < \frac{n}{2}$  is no longer needed, and the lower bound becomes  $t < \min(k + x - q)$ .)

**$IRIS(PR_{\diamond\mathcal{S}_{x,q}})$**  The property  $PR_{\diamond\mathcal{S}_{x,q}}$  extends the property  $PR_{\diamond\mathcal{S}_x}$  in a natural way. Informally,  $PR_{\diamond\mathcal{S}_{x,q}}$  is satisfied if  $PR_{\diamond\mathcal{S}_{x_i}}$  is satisfied for each cluster  $X_i$ .

$$\begin{aligned}
 PR_{\diamond\mathcal{S}_{x,q}} &\equiv \exists X_1, \dots, X_q : \left| \bigcup_{1 \leq j \leq q} X_j \right| \geq x \wedge \forall 1 \leq j < k \leq q : X_j \cap X_k = \emptyset, \\
 &\exists \ell_1, \dots, \ell_q : \forall 1 \leq j \leq q : \ell_j \in X_j, \\
 &\exists r : \forall r' \geq r, \forall 1 \leq j \leq q : (i \in X_j - \{\ell_j\}) \implies (sm_i^{r'} = \emptyset \vee \ell_j \in sm_i^{r'}).
 \end{aligned}$$

This property states that, for each cluster  $X_i$ , there is a process  $p_{\ell_i}$  that, from some round  $r$ , always belongs to the view of the processes of  $X_i$  that have not crashed.

**Building  $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$  in the read/write model equipped with  $\diamond\mathcal{S}_{x,q}$**  An algorithm that simulates the  $IRIS(PR_{\diamond\mathcal{S}_x})$  model from one-shot immediate snapshot objects is described in Figure 6. It can easily be checked that the very same construction can be used to simulate the  $WRITE\_SNAPSHOT()$  operations of the  $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$  in the read/write model equipped with a failure detector of the class  $\diamond\mathcal{S}_{x,q}$ . Thus we obtain:

**Lemma 7.8.** *In the shared memory model equipped with a failure detector of the class  $\diamond\mathcal{S}_{x,q}$ , the algorithm described in Figure 6 simulates the  $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$  model.*

**Simulating a failure detector of the class  $\diamond\mathcal{S}_{x,q}$  in the  $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$  model** A simple algorithm implementing a failure detector of the class  $\diamond\mathcal{S}_x$  in the  $IRIS(PR_{\diamond\mathcal{S}_x})$  model is described in Section 4.1 (Figure 9). Again, one can easily check that this algorithm executed in the  $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$  model emulates a failure detector of the class  $\diamond\mathcal{S}_{x,q}$ . Therefore,

**Lemma 7.9.** *The algorithm described in Figure 9 emulates a failure detector of the class  $\diamond\mathcal{S}_{x,q}$  in the  $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$  model.*

*Proof.* The property  $PR_{\diamond\mathcal{S}_{x,q}}$  states that there are  $q$  disjoint sets  $X_1, \dots, X_q$  of cumulatively  $x$  processes and  $q$  processes  $p_{\ell_1} \in X_1, \dots, p_{\ell_q} \in X_q$  and a round  $r$  after which,  $\forall j : 1 \leq j \leq q, \ell_j$  belongs to the views  $sm_i^{r'}$  of the processes  $p_i$  of  $X_j$  that have not yet crashed. Due to the assignment  $TRUSTED_i \leftarrow sm_i^{r'}$  executed during each round  $r' \geq r$ , this immediately translates as “there are  $q$  disjoint sets  $X_1, \dots, X_q$  of cumulatively  $x$  processes,  $q$  processes  $p_{\ell_1} \in X_1, \dots, p_{\ell_q} \in X_q$  and a time  $\tau$  after which, for each  $1 \leq j \leq q, p_{\ell_j}$  is not suspected by the processes of  $X_j$ ”.  $\square$

**Lower Bound** To prove the lower bound the following strategy is used. Given  $k < n - x + q$ , let us assume that there is an algorithm  $\mathcal{A}$  that solves wait-free solves the  $k$ -set agreement problem in the basic read/write model equipped with a failure detector of the class  $\diamond\mathcal{S}_{x,q}$ . From the Lemmas 7.8 and 7.9, the conditions required by the Theorem 6.9 hold. We can consequently conclude that there is an algorithm  $\mathcal{B}$  that solves  $k$ -set agreement in the  $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$  model. Then, analyzing a class of admissible runs in  $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$  model, it is possible to derive (from the algorithm  $\mathcal{B}$ ) a solution to the  $k$ -set agreement problem for ( $k <$ )  $n - x + q$  processes in the  $IIS$  model, which is known to be impossible ([8, 9, 31, 51]).

**Theorem 7.10.** *There is no algorithm that wait-free solves  $k$ -set agreement for  $n$  processes in the read/write model equipped with a failure detector of the class  $\diamond\mathcal{S}_{x,q}$ , for  $k < n - x + q$ .*

*Proof.* From the previous discussion, there is an algorithm  $\mathcal{B}$  that solves the  $k$ -set agreement task in the  $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$  model. We restrict our attention to a particular class of executions  $E$  defined iterated models. Let us partition the set of processes in two sets: the low-order processes  $L = \{p_1, \dots, p_{n-x+q}\}$  and the high-order processes  $H = \{p_{n-x+q+1}, \dots, p_n\}$ .  $E$  is a subset of all (infinite) executions admissible in the  $IIS$  model. Moreover, in an execution  $e \in E$ , there is at least one low-order process that is correct and, at each round, low-order processes that have not yet crashed are scheduled before any high-order process. In other words, a low order process  $p_i$  never observes a high order process in its view  $sm_i$ . More formally, an iterated execution  $e$  belongs to  $E$  iff the two following conditions hold:

- $\exists p_i \in L : \forall r : sm_i^r \neq \emptyset$ .
- $\forall r, \forall p_i \in L, \forall p_j \in H : (sm_i^r \neq \emptyset \wedge sm_j^r \neq \emptyset) \implies sm_i^r \subsetneq sm_j^r$ .

Let us observe (*observation O1*) that all wait-free runs in which only a subset of low-ordered processes participate are included in  $E$ . We next show that all executions that belong to  $E$  are admissible in the  $IRIS(PR_{\diamond\mathcal{S}_{x,q}})$  model (*observation O2*).

Let  $e \in E$ . There is a low-order process  $p_\alpha$  that takes infinitely many steps in  $e$ . W.l.o.g., let us assume that  $p_\alpha = p_q$  (as  $n-x \geq 0, n-x+q \geq q$ , i.e.,  $p_q$  is a low-order process). Consider the following  $q$  sets of processes:  $X_1 = \{p_1\}, \dots, X_{q-1} = \{p_{q-1}\}$  and  $X_q = \{p_q\} \cup H$ . These sets are disjoint and  $|\bigcup_{1 \leq i \leq q} X_i| = q-1+1+|H| = q+x-q = x$ . Define  $\ell_1 = 1, \ell_2 = 2, \dots, \ell_q = q$ . Finally, observe that  $\forall r, \forall j \in X_q - \{p_q\} = H, sm_{\ell_q}^r \subsetneq sm_j^r$ . The later follows from the fact that the low-order process  $p_q$  is always set linearized before any high-order process. To summarize,  $\forall r, \forall j, 1 \leq j \leq q, \forall i \in X_j - \{\ell_j\} : sm_i^r = \emptyset \vee sm_{\ell_j}^r \subsetneq sm_i^r$ , from which we conclude that the property  $PR_{\diamond\mathcal{S}_{x,q}}$  is satisfied in the execution  $e$ .

It follows from *O1* that in all the wait-free runs in which only low-ordered processes participate are included in  $E$ . Moreover, *O2* establishes that algorithm  $\mathcal{B}$  is a wait-free solution to  $k$ -set agreement in  $E$ . Consequently,  $\mathcal{B}$  is solution to  $k$ -set agreement in the  $IIS$  model for  $n - (x - q)$  processes. This would imply a wait-free solution for  $n - (x - q) > k$  processes to the  $k$ -set agreement problem in the read/write model [9], which is known to be impossible [8, 31, 51].  $\square$

Wait-free algorithms for solving  $k$ -set agreement for  $n$  processes in a message-passing system equipped with a failure detector of the class  $\mathcal{S}_{x,q}$ , such that  $q \leq k \wedge n - x + q \leq k$ , are given in [27, 40]. Such algorithms can easily be translated in the read/write model equipped with a failure detector of the class  $\mathcal{S}_{x,q}$ . Then, using the techniques developed in [53], these algorithms can be transformed to obtain solutions in the read/write model equipped with  $\diamond\mathcal{S}_{x,q}$ . We consequently obtain the following corollary.

**Corollary 7.11.** *Let  $q \leq k$ . There is a wait-free algorithm for solving  $k$ -set agreement among  $n$  processes in the read/write model equipped with a failure detector of the class  $\diamond\mathcal{S}_{x,q}$  iff  $n - x + q \leq k$ .*

## 8 Conclusion

This paper has shown that failure detectors can be represented as schedulers in the *IIS* model, the aim of which is to prevent some runs from occurring. To that end, the paper has investigated the Iterated Immediate Snapshot (*IIS*) model equipped with failure detectors. First, a companion paper [47] has shown that enriching such a model with a failure detector does not increase its computational power with respect to wait-free solvable tasks. Then, given a failure detector of a class  $C$  (where  $C$  is  $\{\diamond\mathcal{S}_x\}_{1 \leq x \leq n}$ ,  $\{\Omega^z\}_{1 \leq z \leq n}$ , or  $\{\diamond\psi^y\}_{1 \leq y \leq n}$ ), it has shown that the power of  $C$  can be added to the iterated model as soon as its base write-snapshot primitive satisfies an additional requirement, giving rise to the Iterated Restricted Immediate Snapshot model denoted  $IRIS(PR_C)$ . The paper has then shown that, for any the three previous failure detector classes  $C$ ,  $IRIS(PR_C)$  and the classical read/write model enriched with a failure detector of the class  $C$  have the same computational power.

In addition to providing a better insight on the very nature of failure detectors, the approach followed in the paper allows designing novel impossibility proofs, entirely based on an algorithmic reasoning.

## References

- [1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic Snapshots of Shared Memory. *Journal of the ACM*, 40(4):873-890, 1993.
- [2] Attiya H., Bar-Noy A., Dolev D., Sharing Memory Robustly in Message Passing Systems. *Journal of the ACM*, 42(1):124-142, 1995.
- [3] Afek Y., Dolev D., Gafni E., Merritt M. and Shavit N., A Bounded First-In, First-Enabled Solution to the  $\ell$ -Exclusion Problem. *ACM Transactions on Programming Languages and Systems*, 16(3):939-953, 1994.
- [4] Anceaume E., Fernández A., Mostéfaoui A., Neiger G. and Raynal M., A necessary and Sufficient Condition for Transforming Limited Accuracy Failure Detectors. *Journal of Computer Systems Science*, 68(1):123-133, 2004.
- [5] Awerbuch, B., Complexity of network synchronization. *Journal of the ACM*, 32:804-823, 1985.
- [6] Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*, Wiley, 2004.
- [7] Borowsky E. and Gafni E., Immediate Atomic Snapshots and Fast Renaming. *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 41-51, 1993.
- [8] Borowsky E. and Gafni E., Generalized FLP Impossibility Results for  $t$ -Resilient Asynchronous Computations. *Proc. 25th ACM Symposium on Theory of Computing (STOC)*, ACM Press, pp. 91-100, 1993.
- [9] Borowsky E. and Gafni E., A Simple Algorithmically Reasoned Characterization of Wait-free Computations. *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 189-198, 1997.
- [10] Borowsky E., Gafni E., Lynch N. and Rajsbaum S., The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127-146, 2001.
- [11] Biran, O., Moran, S., and Zaks, S., A Combinatorial Characterization of the Distributed 1-solvable Tasks. *Journal of Algorithms*, 11:420-440, 1990.
- [12] Chandra T., Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.
- [13] Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685-722, 1996.
- [14] Chaudhuri S., More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation*, 105:132-158, 1993.
- [15] Cornejo A., Rajsbaum S., Raynal M., Travers C., Failure Detectors as Schedulers (Brief Announcement). *Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp.308-309, 2007.
- [16] Dwork C., Lynch N., Stockmeyer L., Consensus in the Presence of Partial Synchrony, *Journal of the ACM*, 35(2):288-323, 1988.



- [17] Fischer M., Lynch N. and Paterson M., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.
- [18] Fraigniaud P., Rajsbaum S. and Travers C., Locality and Checkability in Wait-free Computing *Proc. 25th International Symposium on Distributed Computing (DISC)*, To appear, 2011.
- [19] Gafni E., Round-by-round Fault Detectors: Unifying Synchrony and Asynchrony. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 143-152, 1998.
- [20] Gafni E., The Extended BG-simulation and the Characterization of  $t$ -Resiliency. *Proc. 41st ACM Symposium on Theory of Computing (STOC)*, ACM Press, pp. 85-92, 2009.
- [21] Gafni E., Merritt M. and Taubenfeld G., The concurrency hierarchy and algorithms for unbounded concurrency. *Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 161-169, 2001.
- [22] Gafni E., Rajsbaum S. Recursion in Distributed Computing. *Proc. 12th Int'l Symposium Stabilization, Safety, and Security of Distributed Systems (SSS)*, Springer Verlag LNCS #6366, pp. 362-376, 2010.
- [23] Gafni E., Rajsbaum S. Distributed Programming with Tasks. *Proc. 14th Int'l Conference On Principles Of Distributed Systems (OPODIS)*, Springer Verlag LNCS #6490, pp. 205-218, 2010.
- [24] Gafni E., Rajsbaum S. and Herlihy M., Subconsensus Tasks: Renaming is Weaker than Set Agreement. *Proc. 20th Int'l Symposium on Distributed Computing (DISC)*, Springer Verlag LNCS #4167, pp.329-338, 2006.
- [25] Guerraoui R. and Schiper A., Gamma-Accurate Failure Detectors. *Proc 10th Int'l Workshop on Distributed Algorithms (WDAG)*, Springer Verlag LNCS #1151, pp. 269-286, 1996.
- [26] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.
- [27] Herlihy M. and Penso L. D., Tight Bounds for  $k$ -Set Agreement with Limited Scope Accuracy Failure Detectors. *Distributed Computing*, 18(2):157-166, 2005.
- [28] Herlihy M., and Rajsbaum S., The topology of shared-memory adversaries. *Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 105-113, 2010.
- [29] Herlihy M., and Rajsbaum S., Concurrent Computing and Shellable Complexes. *Proc. 24th Int'l Symposium on Distributed Computing (DISC)*, Springer Verlag LNCS #6343, pp. 109-123, 2010.
- [30] Herlihy M.P., Rajsbaum S., and Tuttle M., Unifying Synchronous and Asynchronous Message-Passing Models, *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 133-142, 1998.
- [31] Herlihy M., Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923, 1999.
- [32] Herlihy M.P. and Wing J.M., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.
- [33] Imbs D. and Raynal M., Visiting Gafni's Reduction Land: from the BG Simulation to the Extended BG Simulation. *Proc. 11th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, Springer-Verlag LNCS 5873, pp. 369-383, 2009.
- [34] Imbs D. and Raynal M., The Multiplicative Power of Consensus Numbers. *Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 26-35, 2010.
- [35] Keidar I., Shraer A., Timeliness, Failure-detectors, and Consensus Performance. *Proc. 25th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 169-178, 2006.
- [36] Lamport, L. and Lynch, N., Distributed Computing: Models and Methods. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pp. 1157-1199, 1990.
- [37] Moses Y. and Rajsbaum S., A Layered Analysis of Consensus. *SIAM Journal of Computing*, 31(4):989-1021, 2002.
- [38] Mostefaoui A., Rajsbaum S., Raynal M. and Travers C., On the Computability Power and the Robustness of Set Agreement-oriented Failure Detector Classes. *Distributed Computing* 21(3): 201-222 (2008)

- [39] Mostefaoui A., Rajsbaum S., Raynal M. and Travers C., The Combined Power of Conditions and Information on Failures to Solve Asynchronous Set Agreement. *SIAM Journal of Computing*, 38(4):1574-1601, 2008.
- [40] Mostefaoui A. and Raynal M.,  $k$ -Set Agreement and Limited Accuracy Failure Detectors. *19th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM press, pp. 143-152, 2000.
- [41] Neiger G., Set Linearizability. *Brief Announcement, Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 396, 1994.
- [42] Neiger G., Failure Detectors and the Wait-free Hierarchy. *Proc. 14th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 100-109, 1995.
- [43] Rajsbaum S., Iterated Shared Memory Models. *Proc. 9th Latin American Symposium Theoretical Informatics (LATIN'10)*, Springer Verlag LNCS #6343, pp.407-416, 2010.
- [44] Rajsbaum S., Raynal M., Travers C., Failure Detectors as Schedulers. *Technical Report #1838*, IRISA, Université de Rennes, France, 2007.
- [45] Rajsbaum S., Raynal M., Travers C., The Iterated Restricted Immediate Snapshot Model. *Technical Report # 1874*, IRISA, Université de Rennes, France, 2007.
- [46] Rajsbaum S., Raynal M., Travers C., The Iterated Restricted Immediate Snapshot Model. *Proc. 14th Annual Int'l Conference Computing and Combinatorics (COCOON 2008)*, Springer Verlag LNCS #5092, pp. 487-497, 2008.
- [47] Rajsbaum S., Raynal M., Travers C., An impossibility about Failure Detectors in the Iterated Immediate snapshot Model. *Information Processing Letters*, 108(3), 2008, 160–164.
- [48] Raynal M., Set agreement. *Encyclopedia of Algorithms*, Springer-Verlag, pp. 829-831, 2008 (ISBN 978-0- 387-30770-1).
- [49] Raynal M., Failure Detectors for Asynchronous Distributed Systems: an Introduction. *Wiley Encyclopedia of Computer Science and Engineering*, Vol. 2, pp. 1181-1191, 2009 (ISBN 978-0-471-38393-2).
- [50] Raynal M., Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems. *Morgan & Claypool Publishers*, 251 pages, 2010 (ISBN 978-1-60845-293-4).
- [51] Saks M. and Zaharoglou F., Wait-Free  $k$ -Set Agreement is Impossible: The Topology of Public Knowledge. *SIAM Journal on Computing*, 29(5):1449-1483, 2000.
- [52] Völzer H., On Conspiracies and Hyperfairness in Distributed Computing. *Proc. 19th Int'l Symposium on Distributed Computing (DISC)*, Springer Verlag LNCS #3724, pp. 33-47, 2005.
- [53] Yang J., Neiger G. and Gafni E., Structured Derivations of Consensus Algorithms for Failure Detectors. *Proc. 17th Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp.297-308, 1998.
- [54] Zieliński P., Anti-Omega: the Weakest Failure Detector for Set Agreement. *Proc. 27th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 55-64, 2008.