Visiting Gafni's Reduction Land: From the BG Simulation to the Extended BG Simulation

Damien Imbs and Michel Raynal

IRISA, Université de Rennes 1, 35042 Rennes, France

Abstract. The *Borowsky-Gafni* (BG) *simulation* algorithm is a powerful tool that allows a set of $t+1$ asynchronous sequential processes to wait-free simulate (i.e., despite the crash of up to t of them) a large number n of processes under the assumption that at most t of these processes fail (i.e., the simulated algorithm is assumed to be t-resilient). The BG simulation has been used to prove solvability and unsolvability results for crash-prone asynchronous shared memory systems.

In its initial form, the BG simulation applies only to colorless decision tasks, i.e., tasks in which nothing prevents processes to decide the same value (e.g., consensus or k-set agreement tasks). Said in another way, it does not apply to decision problems such as renaming where no two processes are allowed to decide the same new name. Very recently (STOC 2009), Eli Gafni has presented an *extended* BG *simulation* algorithm (GeBG) that generalizes the basic BG algorithm by extending it to "colored" decision tasks such as renaming. His algorithm is based on a sequence of sub-protocols where a sub-protocol is either the base agreement protocol that is at the core of BG simulation, or a commit-adopt protocol.

This paper presents the core of an extended BG simulation algorithm that is particularly simple. This algorithm is based on two underlying objects: the base agreement object used in the BG simulation (as does GeBG), and (differently from GeBG) a new simple object that we call *arbiter*. More precisely, (1) while each of the n simulated processes is simulated by each simulator, (2) each of the first $t+1$ simulated processes is associated with a predetermined simulator that we called its "owner". The arbiter object is used to ensure that the permanent blocking (crash) of any of these $t+1$ simulated processes can only be due to the crash of its owner simulator. After being presented in a modular way, the proposed extended BG simulation algorithm is proved correct.

Keywords: Arbiter, Asynchronous processes, Distributed computability, Fault-Tolerance, Process crash failure, Reduction, t-Resilience, Shared memory system, Wait-free environment.

1 Introduction

What is the Borowsky-Gafni (BG) simulation. Considering an asynchronous system where processes can crash, the (n, k)-set agreement problem is a basic decision task defined as follows [9]. Each of the n processes proposes a value, and every process that does not crash has to decide a value (termination), such that a decided value is a proposed value (validity) and at most k different values are decided (agreement). The consensus problem corresponds to the particular case $k = 1$.

A fundamental question related to asynchronous distributed computability is the following. Suppose we have an algorithm that solves the $(15, 4)$-set agreement problem. Can we use this algorithm as a subroutine to solve the $(12, 5)$-set agreement problem, assuming that at most $t < 12$ processes can crash? Intuitively, the answer might be "yes" (as we have less processes and more decided values are allowed). Let us now suppose that we want to use the same $(15, 4)$-set agreement subroutine to solve the $(100, 4)$-set agreement problem. As we have much more proposed values, and the same constraint on the number of decided values, an intuitive answer does not spring in an obvious way. And what is the answer if we want to solve the $(80, 7)$-set agreement problem (much more proposed values but only two more values can be decided), or (assuming $t = 4$) solve the $(5, 4)$-set agreement problem?

Stated in more general terms, the question is: *"Can we use a solution to the (n, k)-set agreement problem as a subroutine to solve the (n', k')-set agreement problem, when at most $t < \min(n, n')$ processes may crash?"* (We say that "the (n', k')-set agreement is reducible to (n, k)-set agreement".) The BG simulation (introduced in [6] and formalized and deeply investigated in [7] where is given a formal definition of "reducibility") answers this fundamental question. It states that the answer is "yes" if $k' \geq k$ and "no" if $k' \leq t < k$. As we can see, the answer "yes" does not depend on the number of processes.

To that end, a BG simulation algorithm is described that allows $n' = t + 1$ processes to simulate a large number n of processes that collectively solve a decision task in presence of at most t crashes. Each of the n' simulator processes simulates all the n processes. These n' simulator processes cooperate through underlying objects (the type of which is called here safe_agreement) that allow them to agree on a single output for each of the non-deterministic statements issued by every simulated process.

The important lesson learned from the BG simulation is that, in a failure-prone context, what is important is not the number of processes but the maximal number of possible failures and the actual number of values that are proposed to a decision task. An interesting application of the BG simulation (among several of its applications [7]) is the proof that there is no t-resilient (n, k)-set agreement algorithm for $t \geq k$. This is obtained as follows. As (1) the BG simulation allows reducing the $(k + 1, k)$-set agreement problem to the (n, k)-set agreement problem in a system with up to k failures, and (2) the $(k + 1, k)$-set agreement problem is known to be impossible in presence of k failures [6,13,17], it follows that there is no k-resilient (n, k)-set agreement algorithm.

The limit of the BG-simulation and the extended BG-simulation. The BG simulation characterizes t-resilient solvability by reducing it to the question of wait-free solvability (i.e., t-resilience in a system of $n = t + 1$ processes). Unfortunately, the BG simulation is limited to *colorless* decision tasks, i.e., tasks in which if a process decides a value v, then all the processes can decide that value (the class of colorless tasks is formally defined in [7]). The (n, k)-set decision problem is typically such a task. From an operational point of view, this is due to the fact that, in the BG simulation, each simulator simulates fairly all the processes, and consequently, the crash of a simulator process can manifest itself as the crash of any simulated process (the one it is currently simulating a critical part of code).

The extended BG simulation has been proposed by Eli Gafni to overcome this limitation and consequently fully capture t-resilience [12]. As stated in [12] "With the extended BG simulation we can reduce questions about t-resilience solvability to questions about wait-free solvability. The latter is characterized by the Herlihy-Shavit conditions [13]".

As a result, it applies to both colorless tasks and *colored* decision tasks such as the renaming problem [3]. In that problem, each of the n processes has to decide a new name (from a given new name space) such that no two processes have the same new name. This problem has wait-free solutions when the new name space $[1..M]$ is such that $M \geq 2n - 1$ (see [8] for a deeper insight into the problem).

In his paper [12], Gafni presents several (un)decidability results that can be obtained in a simpler way from the BG simulation. He also uses the extended BG simulation to show that the t-resilient weak symmetry breaking problem is equivalent to t-resilient weak renaming problem.

The core of the BG simulation relies on the following principles: (1) each of the $(t + 1)$ simulators fairly simulates all the processes, and this simulation is such that (2) the crash of a simulator entails the crash of at most one simulated process. The BG simulation is "symmetric" in the sense that each of the n processes is simulated by every simulator, and the $(t + 1)$ simulators are "equal" with respect to each simulated process. One way to be able to simulate colored tasks (without preventing the simulation of colorless tasks), consists in introducing some form of *asymmetry*.

The extended BG simulation [12] realizes the appropriate asymmetry as follows. In addition of simulating an appropriate subset of the n simulated processes, each simulator q is statically associated with exactly one given simulated process p (in our terminology, q is the *owner* of p). This ownership notion is used to ensure that the corresponding simulated process p will not be blocked forever (perceived as crashed) if its owner simulator q does not crash. Hence, if a simulator does not crash, it can always decide the value decided by the simulated process p it "owns". As noticed and demonstrated in [12] "extending the BG simulation by this simple property results in a full characterization of t-resilience in terms of wait-freedom".

Content of the paper. In addition to the introduction of the notion of extended BG simulation, and a full characterization of t-resilience, Gafni presents in [12] an extended BG simulation algorithm (denoted here GeBG). This algorithm is based on a sequence of sub-protocols where each sub-protocol is either the base agreement protocol used in the BG simulation (safe_agreement type objects) or a commit-adopt protocol [11]. This algorithm is presented informally in English.

The present paper presents the core of an extended BG simulation algorithm that is particularly simple. This algorithm is based on two underlying object types: the type safe_agreement (the one used in the BG simulation algorithm and in GeBG), and (differently from GeBG) an object type that we call arbiter. An arbiter object allows exploiting the ownership notion in a simple way to ensure that (1) an object value is always decided when its owner does not crash, and (2) the value of that object is determined either by its owner simulator or by the other simulators.

As far as the whole simulation is concerned, while (as in the BG simulation) each of the n simulated processes is simulated by each simulator, (as in GeBG) each of the first

$t+1$ simulated processes is “associated” with exactly one simulator (its “owner”). As already said, it follows from the appropriate use of the arbiter objects that the permanent blocking (crash) of any of these $t+1$ simulated processes can only be due to the crash of its owner simulator.

The paper is made up of 7 sections. Section 2 presents the model and the definition of decision tasks. Section 3 explains the structure of the simulation. Section 4 defines the base object types used by the simulators to cooperate and realize a correct simulation. Then, the extended BG simulation algorithm is presented in an incremental and modular way. First Section 5 briefly presents the BG simulation algorithm, and then Section 6 enriches it to solve the extended BG simulation. This algorithm is proved in Section 7.

2 Solving Decision Tasks

2.1 Decision Tasks

The problems we are interested in are called *decision tasks*[1]. In every run, each process proposes a value and the proposed values define an input vector I where $I[j]$ is the value proposed by p_j. Let $\mathcal{I}$ denote the set of allowed input vectors. Each process has to decide a value. The decided values define an output vector O, such that $O[j]$ is the value decided by p_j. Let $\mathcal{O}$ be the set of the output vectors.

A decision task is a binary relation Δ from $\mathcal{I}$ into $\mathcal{O}$. A task is *colorless* if, when a value v is decided by a process p_j (i.e., $O[j] = v$), then v can be decided by all the processes). Consensus, and more generally k-set agreement, are colorless tasks. Otherwise the task is *colored*. Renaming is a colored task.

2.2 The Computation Model

Asynchronous processes and fault model. We are interested in distributed algorithms the aim of which is to solve a task in a system made up of n asynchronous sequential processes denoted p_1, ..., p_n. A process executes a sequence of atomic steps (as defined by its algorithm). Each process p_j is endowed with a write-once local variable $output_j$ where it deposits the value it decides.

A process can crash in a run. A process executes correctly the steps defined by its algorithm until it crashes (if it ever does). After it has crashed, a process executes no more steps. If it does not crash, a process executes an infinite number of steps.

It is assumed that an arbitrary subset (not known in advance) of up to $t < n$ processes can crash (the crash of one process being independent from the crash of other processes). A process that does not crash in a run is said to be *correct* in that run, otherwise it is faulty. This failure model is called the *t-resilient environment*, and an algorithm designed for such an environment is said to be *t-resilient*. The extreme case $t = n-1$ is called *wait-free environment*, and the corresponding algorithms are called *wait-free algorithms*.

[1] The reader interested in a more formal presentation of decision tasks can consult the literature (e.g., [2,7,12,13]).

Communication model. The n processes cooperate through a shared memory made up of a snapshot object [1] denoted mem. This means that a process p_j can write only the entry $mem[j]$ but can read all the entries by invoking the operation $mem.\mathsf{snapshot}()$. The write and snapshot operations appear as being executed atomically [1]. (These operations can be built on top of a single-writer/multiple-readers atomic registers [1,4]). Initially, $mem[j] = \perp$.

Definition. The previous computation model (asynchronous crash-prone processes that communicate through snapshot objects) is called *snapshot model*.

2.3 Algorithm Solving a Task

An algorithm solves a task in a t-resilient environment if, given any $I \in \mathcal{I}$, each correct process p_j decides (i.e., writes a value v in $output_j$) and there is an output vector O such that $(I, O) \in \Delta$ where O is defined as follows. If p_j decides v, then $\mathcal{O}[j] = v$. If p_j does not decide, $O[j]$ is set to any value v' that preserves the relation $(I, O) \in \Delta$.

A task is solvable in a t-resilient environment if there is an algorithm that solves it in that environment. As an example, consensus is not solvable in the 1-resilient environment [10,15,16]. Differently, renaming with $2n - 1$ names is solvable in the wait-free environment [3,5,13].

3 Simulated Processes vs. Simulator Processes

Aim. Let A be an n-process t-resilient algorithm that solves a decision task in the base snapshot model described previously. The aim is to design a $(t + 1)$-process wait-free algorithm A' that simulates A in the same snapshot model. (The reader is referred to [7] for a formal definition of a simulation.)

Notation. A simulated process is denoted p_j with $1 \leq j \leq n$. Similarly, a simulator process (in short "simulator") is denoted q_i with $1 \leq i \leq t + 1$.

As far as the objects accessed by the simulators are concerned, the following convention is adopted. The objects denoted with upper case letters are the objects shared by the simulators. Differently, an object denoted with lower case letters is local to a simulator (in that case, the associated subscript denotes the corresponding simulator).

What a simulator does. Each simulator q_i is given the code of all the simulated processes $p_1, \ldots, p_n$. It manages n threads, each one associated with a simulated process, and locally executes these threads in a fair way. It also manages a local copy mem_i of the snapshot memory mem shared by the simulated processes.

The code of a simulated process p_j contains writes of $mem[j]$ and invocations of $mem.\mathsf{snapshot}()$. These are the only operations used by the processes $p_1, \ldots, p_n$ to cooperate. So, the core of the simulation is the definition of two algorithms. The first (denoted $\mathsf{sim\_write}_{i,j}()$) has to describe what a simulator q_i has to do in order to correctly simulate a write of $mem[j]$ issued by a process p_j. The second (denoted $\mathsf{sim\_snapshot}_{i,j}()$) has to describe what a simulator q_i has to do in order to correctly simulate an invocation of $mem.\mathsf{snapshot}()$ issued by a process p_j.

4 Base Object Types Used in the Simulation

In addition to snapshot objects, the simulator processes also cooperate through atomic read/write register objects, and specific objects the types of which (safe_agreement and arbiter) are defined in this section. These types can be implemented from multi-reader/multi-writer atomic registers, which in turn can be implemented from snapshot objects. Hence, all the base objects used in the simulation can be implemented in the snapshot computation model described in the previous section.

4.1 The safe_agreement Object Type

The safe_agreement *type.* This object type (defined in [6,7]) is at the core of the BG simulation. It provides each simulator q_i with two operations, denoted $\mathsf{propose}_i(v)$ and $\mathsf{decide}_i()$, that q_i can invoke at most once, and in that order. The operation $\mathsf{propose}_i(v)$ allows q_i to propose a value v while $\mathsf{decide}_i()$ allows it to decide a value. The properties satisfied by an object of the type safe_agreement are the following.

- Termination. If no simulator q_x crashes while executing $\mathsf{propose}_x()$, then any correct simulator q_i that invokes $\mathsf{decide}_i()$, returns from that invocation.
- Agreement. At most one value is decided.
- Validity. A decided value is a proposed value.

An implementation. The implementation of the safe_agreement type described in Figure 1 is from [7]. This construction is based on a snapshot object SM (with one entry per simulator q_i). Each entry $SM[i]$ of the snapshot object has two fields: $SM[i].value$ that contains a value and $SM[i].level$ that stores its level. The level 0 means the corresponding value is meaningless, 1 means it is unstable, while 2 means it is stable.

When a simulator q_i invokes $\mathsf{propose}_i(v)$, it first writes the pair $(v, 1)$ in $SM[i]$ (line 01), and then reads the snapshot object SM (line 02). If there is a stable value in SM, p_i "cancels" the value it proposes, otherwise it makes it stable (line 03).

A simulator q_i invokes $\mathsf{decide}_i()$ after it has invoked $\mathsf{propose}_i()$. Its aim is to return the same stable value to all the simulators that invoke this operation (line 06). To that end, q_i repeatedly computes a snapshot of SM until it sees no unstable value in SM (line 04). Let us observe that, as a simulator q_i invokes $\mathsf{decide}_i()$ after it has invoked $\mathsf{propose}_i(v)$, there is at least one stable value in SM when it executes line 05. Finally, in order that the same stable value be returned to all, q_i returns the stable value proposed by the simulator with the smallest id (line 05).

A formal proof that this algorithm implements the safe_agreement type is given [7]. Another proof is also given [14].

4.2 The arbiter Object Type

Definition. Each object of the type arbiter has a statically predefined owner simulator q_j. Such an object provides the simulators with a single operation denoted $\mathsf{arbitrate}_{i,j}()$ (where i is the id of the invoking simulator and j the id of the owner). A simulator q_i invokes $\mathsf{arbitrate}_{i,j}()$ at most once, and, when it terminates, this invocation returns a value to q_i. The properties of an object of the type arbiter owned by q_j are the following.

```
init: for each $x : 1 \le x \le t+1$ do $SM[x] \leftarrow (\perp, 0)$ end for.

operation propose$_i(v)$:   % $1 \le i \le t+1$ %
(01) $SM[i] \leftarrow (v, 1)$;
(02) $sm_i \leftarrow SM$.snapshot();
(03) if $(\exists x : sm_i[x].level = 2)$ then $SM[i] \leftarrow (v, 0)$ else $SM[i] \leftarrow (v, 2)$ end if.

operation decide$_i()$:   % $1 \le i \le t+1$ %
(04) repeat $sm_i \leftarrow SM$.snapshot() until $(\forall x : sm_i[x].level \neq 1)$ end repeat;
(05) let $x = \min(\{k \mid sm_i[k].level = 2\}$; $res \leftarrow sm_i[x].value$;
(06) return($res$).
```

Fig. 1. An implementation of the safe_agreement type [7] (code for q_i)

- Termination. If the owner q_j invokes arbitrate$_{j,j}()$ and is correct, or does not invoke arbitrate$_{j,j}()$, or if a simulator q_i returns from its invocation arbitrate$_{i,j}()$, then all the correct simulators return from their arbitrate$_{i,j}()$ invocation.
- Agreement. No two processes return different values.
- Validity. The returned value is 1 (*owner*) or 0 (*not_owner*). Moreover, if the owner does not invoke arbitrate$_{j,j}()$, 1 cannot be returned, and if only the owner invokes arbitrate$_{i,j}()$, 0 cannot be returned.

An implementation. An implementation of an object of the type arbiter is described in Figure 2. It is based on a snapshot object $PART$ (initialized to $[false, \ldots, false]$), and a multi-writer/multi-reader atomic register $WINNER$ (initialized to $\perp$).

When it invokes arbitrate$_{i,j}()$, the simulator q_i announces that it participates (line 01), and issues a snapshot to know the simulators that are currently participating (line 02). If q_i is the owner of the object ($i = j$, line 03), it checks if it is the first participant (predicate $part_i = \{i\}$). If it is, it sets $WINNER$ to 1, otherwise it sets it to 0 (line 04). If q_i is not the owner of the object ($i \neq j$), it checks if the owner is a participating simulator (predicate $j \in part_i$). If it is, q_i waits to know which value has been assigned to $WINNER$. If it is not, it sets $WINNER$ to 0. Finally, q_i terminates by returning the value of $WINNER$.

A proof that this construction implements the arbiter object type is given in [14].

```
operation arbitrate$_{i,j}()$:   % $1 \le i, j \le t+1$ %
(01) $PART[i] \leftarrow true$;
(02) $aux_i \leftarrow PART.snapshot()$; $part_i \leftarrow \{x \mid aux_i[x]\}$;
(03) if $(i = j)$   % $p_i$ is the owner of the associated arbiter type object %
(04)    then if $(part_i = \{i\})$ then $WINNER \leftarrow 1$ else $WINNER \leftarrow 0$ end if
(05)    else  if $(j \in part_i)$ then wait $(WINNER \neq \perp)$ else $WINNER \leftarrow 0$ end if
(06) end if;
(07) return($WINNER$).
```

Fig. 2. The arbitrate$_{i,j}()$ operation of the arbiter object type (code for q_i)

5 The BG Simulation

This section presents the BG simulation [6,7]: its main principles and the algorithms implementing its base operations $\mathsf{sim\_write}_{i,j}()$ and $\mathsf{sim\_snapshot}_{i,j}()$.

5.1 The Shared Memory $MEM[1..(t+1)]$

The snapshot memory mem shared by the processes $p_1, \ldots, p_n$ is emulated by a snapshot object MEM shared by the simulators (so, MEM has $(t+1)$ entries).

More specifically, $MEM[i]$ is an (unbounded) atomic register that contains an array with one entry per simulated process p_j. Each $MEM[i][j]$ is made up of two fields: a field $MEM[i][j].value$ that contains the last value of $mem[j]$ written by p_j, and a field $MEM[i][j].sn$ that contains the associated sequence number. (This sequence number, introduced by the simulation, is a control data that will be used to produce a consistent simulation of the $mem.\mathsf{snapshot}()$ operations issued by the simulated processes).

5.2 The $\mathsf{sim\_write}_{i,j}()$ Operation

The algorithm, denoted $\mathsf{sim\_write}_{i,j}(v)$, executed by q_i to simulate the write by p_j of the value v into $mem[j]$ is described in Figure 3 [7]. Its code is pretty simple. The simulator q_i first increases a local sequence number $w\_sn_i[j]$ that will be associated with the value v written by p_j into $mem[j]$. Then, q_i writes the pair $(v, w\_sn_i[j])$ into $mem_i[j]$ (where mem_i is its local copy of the memory shared by the simulated processes) and finally writes atomically its local copy mem_i into $MEM[i]$.

```
operation $\mathsf{sim\_write}_{i,j}(v)$:
(01) $w\_sn_i[j] \leftarrow w\_sn_i[j] + 1$;
(02) $mem_i[j] \leftarrow (v, w\_sn_i[j])$;
(03) $MEM[i] \leftarrow mem_i$.
```

Fig. 3. $\mathsf{sim\_write}_{i,j}(v)$ executed by q_i to simulate $\mathsf{write}(v)$ issued by p_j (from [7])

5.3 The $\mathsf{sim\_snapshot}_{i,j}()$ Operation

This operation is implemented by the algorithm described in Figure 4 [7].

Additional local and shared objects. For each process p_j, a simulator q_i manages a local sequence number generator $snap\_sn_i[j]$ used to associate a sequence number with each $mem.\mathsf{snapshot}()$ it simulates on behalf of p_j (line 04).

In addition to the snapshot object $MEM[1..(t+1)]$, the simulators $q_1, \ldots, q_{t+1}$ cooperate through an array $SAFE\_AG[1..n, 0...]$ of $\mathsf{safe\_agreement}$ type objects.

Underlying principle of the BG simulation [6,7]*: obtaining a consistent value.* In order to agree on the very same output of the $snapsn$-th invocation of $mem.\mathsf{snapshot}()$ that is issued by p_j, the simulators $q_1, \ldots, q_{t+1}$ use the object $SAFE\_AG[j, snapsn]$.

Each simulator q_i proposes a value (denoted $input_i$) to that object (line 05) and, due to its agreement property, that object will deliver them the same output at line 06. In order to ensure the consistent progress of the simulation, the input value $input_i$ proposed by the simulator q_i to $SAFE\_AG[j, snapsn]$ is defined as follows.

```
operation sim_snapshot$_{i,j}$():
(01) $sm_i \leftarrow MEM$.snapshot():
(02) for each $y : 1 \leq y \leq n$: do $input_i[y] = sm_i[s][y].value$
(03)         where $\forall x : 1 \leq x \leq t+1 : sm_i[s][y].sn \geq sm_i[x][y].sn$ end for;
(04) $snap\_sn_i[j] \leftarrow snap\_sn_i[j] + 1$; let $snapsn = snap\_sn_i[j]$;
(05) enter_mutex; $SAFE\_AG[j, snapsn]$.propose$_i(input_i)$; exit_mutex;
(06) $res \leftarrow SAFE\_AG[j, snapsn]$.decide$_i$()
(07) return($res$).
```

Fig. 4. sim_snapshot$_{i,j}$() executed by q_i to simulate mem.snapshot() issued by p_j (from [7])

- First, q_i issues a snapshot of MEM in order to obtain a consistent view of the simulation state. The value of this snapshot is kept in sm_i (line 01).
  Let us observe that $sm_i[x][y]$ is such that (1) $sm_i[x][y].sn$ is the number of writes issued by p_y into $mem[y]$ that have been simulated up to now by q_x, and (2) $sm_i[x][y].value$ is the value of the last write into $mem[y]$ as simulated by q_x on behalf of p_y.
- Then, for each p_y, q_i computes $input_i[y]$. To that end, it extracts from $sm_i[1..t+1][y]$ the value written by the more advanced simulator q_s as far as the simulation of p_y is concerned. This is expressed in lines 02-03.

Once $input_i$ has been computed, q_i proposes it to $SAFE\_AG[j, snapsn]$ (line 05), and then returns the value decided by that object (lines 06-07).

The previous description shows an important feature of the BG simulation. A value $input_i[y] = sm_i[s][y].value$ proposed by simulator q_i can be such that $sm_i[s][y].sn > sm_i[i][y].sn$, i.e., the simulator q_s is more advanced than q_i as far as the simulation of p_y is concerned. This causes no problem, as when q_i will simulate mem.snapshot() operations for p_y (if any) that are between the $(sm_i[i][y].sn)$-th and the $(sm_i[s][y].sn)$-th write operations of p_y, it will obtain a value that has already been computed and is currently kept in the corresponding $SAFE\_AG[y, -]$ object.

Underlying principle of the BG simulation [6,7]*: from wait-freedom to t-resilience.* Each simulator q_i simulates the n processes $p_1, \ldots, p_n$ "in parallel" and in a fair way. But any simulator q_i can crash. The crash of q_i while it is engaged in the simulation of mem.snapshot() on behalf of several processes p_j, $p_{j'}$, etc., can entail their definitive blocking, i.e., their crash. This is because each $SAFE\_AG[j, -]$ object guarantees that its $SAFE\_AG[j, -]$.decide() invocations do terminate only if no simulator crashes while executing $SAFE\_AG[j, -]$.propose() (line 05 of Figure 4).

The simple (and bright) idea of the BG simulation to solve this problem consists in allowing a simulator to be engaged in only one $SAFE\_AG[-, -]$.propose() invocation at a time. Hence, if q_i crashes while executing $SAFE\_AG[j, -]$.propose(), it can entail the crash of p_j only. This is obtained by using an additional mutual exclusion object offering the operations enter_mutex and exit_mutex. (Let us notice that such a mutex object is purely local to each simulator: it solves conflicts among the simulating threads inside each simulator, and has nothing to do with the memory shared by the simulators).

From t-resilience to wait-freedom. As an example let us consider we have a t-resilient algorithm that solves the (n, t) agreement problem. We obtain a wait-free algorithm that solves the $(t + 1, t)$ agreement problem as follows. Each simulator q_i $(1 \leq i \leq t + 1)$ is initially given a proposed value v_i, and the base objects $SAFE\_AG[1..n, 0]$ are used by the $(t + 1)$ simulators as follows to determine the value proposed by p_j. For each j, $1 \leq j \leq n$, the simulator q_i invokes first $SAFE\_AG[j, 0].\mathsf{propose}_i(v_i)$ and then $SAFE\_AG[j, 0].\mathsf{decide}_i()$ that returns it a value that it considers as the value proposed by p_j. It is easy to see that, for any j, all the simulators obtain the same value for p_j. Moreover, this value is one of the $t + 1$ values proposed by the simulators. Finally, simulator process q_i can decide any of the values decided by the processes p_j it is simulating. (It is easy to see that the BG simulation is for colorless decision tasks.) A formal proof of this reduction (based on input/output automata) can be found in [7].

From wait-freedom to t-resilience. For colorless decision tasks, t-resilience can easily be reduced to wait-freedom as follows. First, each application process deposits its input value in a shared register. Then, every process of the $t + 1$ processes of the wait-free algorithm takes one of those values as its input value and executes its code. Finally, each application process decides any value decided by a process of the wait-free algorithm.

6 The Extended BG Simulation

This section extends the previous algorithms in order to solve the extended BG simulation. Our aim is to obtain an implementation that is "as simple as possible". To that end, we proceed incrementally by "only" enriching the previous base BG simulation. The proposed implementation uses the same snapshot object MEM and the same $\mathsf{sim\_write}_{i,j}()$ operation (Figure 3) as the base BG simulation. It also uses the same $SAFE\_AG[1..n, 0...]$ array made up of safe_agreement type objects.

This section presents the additional shared objects that are used, the underlying principles on which relies the implementation of $mem.\mathsf{snaspshot}()$ issued by a simulator q_i on behalf of a simulated process p_j, and the algorithm (denoted $\mathsf{e\_sim\_snapshot}_{i,j}())$ that implements it.

6.1 The Additional Shared Objects

In addition to MEM and $SAFE\_AG[1..n, 0...]$, the memory shared by the simulators $q_1, \ldots, q_{t+1}$ contains the following objects.

- $ARBITER[1..t + 1, 0...]$ is an array of arbiter objects. The objects contained in $ARBITER[j, -]$ are owned by the simulator q_j $(1 \leq j \leq t + 1)$.
  The object $ARBITER[j, snapsn]$ is used by a simulator q_i when it simulates its $snapsn$-th invocation $mem.\mathsf{snapshot}()$ on behalf of the simulated process p_j for $1 \leq j \leq t + 1$. (As we will see, when $t + 1 < j \leq n$, the simulation of $mem.\mathsf{snapshot}()$ on behalf of p_j does not require the help of an arbiter object.)
- $ARB\_VAL[1..t + 1, 0...][0..1]$ is an array of pairs of atomic registers. The pair of atomic registers $ARB\_VAL[j, snapsn][0..1]$ is used in conjunction with the arbiter object $ARBITER[j, snapsn]$.

The aim of $ARB\_VAL[j, snapsn][1]$ is to contain the value that has to be returned to the $snapsn$-th invocation mem.snapshot(), on behalf of the simulated process p_j, if the owner q_j is designated as the winner by the associated object $ARBITER[j, snapsn]$. If the owner q_j is not the winner, the value that has to be returned is the one kept in $ARB\_VAL[j, snapsn][0]$.

6.2 The e_sim_snapshot$_{i,j}$() Operation

The enriched algorithm. The algorithm implementing the operation e_sim_snapshot$_{i,j}$() executed by q_i to simulate a mem.snapshot() operation issued by p_j is described in Figure 5. Its first four lines and its last line are exactly the same as in Figure 4. The lines 05-06 are replaced by the new lines N01-N14 that constitutes the "addition" that allows going from the BG to the extended BG simulation.

Underlying principle. Albeit each simulated process p_j ($1 \leq j \leq n$) is simulated by each simulator q_i ($1 \leq i \leq t+1$) as in the BG simulation, each simulated process p_j such that $1 \leq j \leq t+1$ is associated with exactly one simulator that is its "owner": q_i is the owner of p_j if $j = i$ (and also the owner of the corresponding objects $ARBITER[j, -]$). The aim is, for any $snapsn \geq 0$, to associate a single returned value with the $snapsn$-th invocations of e_sim_snapshot$_{i,j}$() issued by the simulators. The idea is to use the ownership notion to "shortcut" the use of $SAFE\_AG[j, snapsn]$ object in appropriate circumstances.

The operation e_sim_snapshot$_{i,j}$() for the simulated processes p_j such that $t+2 \leq j \leq n$, is exactly the same as sim_snapshot$_{i,j}$(). This appears in the lines N02-N03 that are the same as the lines 06-07 of Figure 4 (in that case, there is no ownership notion).

```
operation e_sim_snapshot$_{i,j}$():
(01)   $sm_i \leftarrow MEM$.snapshot():
(02)   for each $y : 1 \leq y \leq n$: do $input_i[y] = sm_i[s, y].value$
(03)                     where $\forall x : 1 \leq x \leq t+1 : sm_i[s, y].sn \geq sm_i[x, y].sn$ end for;
(04)   $snap\_sn_i[j] \leftarrow snap\_sn_i[j] + 1$; let $snapsn = snap\_sn_i[j]$;
(N01)  if $(j > t+1)$
(N02)     then enter_mutex; $SAFE\_AG[j, snapsn]$.propose$_i(input_i)$; exit_mutex;
(N03)          $res \leftarrow SAFE\_AG[j, snapsn]$.decide$_i$()
(N04)  else if $(i = j)$
(N05)     then $ARB\_VAL[j, snapsn][1] \leftarrow input_i$;
(N06)          enter_mutex; $win \leftarrow ARBITER[j, snapsn]$.arbitrate$_{i,j}$(); exit_mutex;
(N07)          if $(win = 1)$ then $res \leftarrow input_i$
(N08)                          else $res \leftarrow ARB\_VAL[j, snapsn][0]$ end if;
(N09)     else enter_mutex; $SAFE\_AG[j, snapsn]$.propose$(input_i)$; exit_mutex;
(N10)          $ARB\_VAL[j, snapsn][0] \leftarrow SAFE\_AG[j, snapsn]$.decide$_i$();
(N11)          $r \leftarrow ARBITER[j, snapsn]$.arbitrate$_{i,j}$();
(N12)          $res \leftarrow ARB\_VAL[j, snapsn][r]$
(N13)     end if
(N14)  end if;
(07)   return$(res)$.
```

Fig. 5. e_sim_snapshot$_{i,j}$() executed by q_i to simulate mem.snapshot() issued by p_j

The new lines N04-N14 address the case of the simulated processes owned by a simulator, i.e., the processes $p_1, \ldots, p_{t+1}$. The idea is the following: if q_i does not crash, p_i must not crash. In that way, if q_i is correct, p_i will always terminate whatever the behavior of the other simulators. To that end, q_i on one side, and all the other simulators on the other side, compete to define the snapshot value returned by the $snapsn$-th invocations $\mathsf{e\_sim\_snapshot}_{i,j}()$ issued by each of them. To attain this goal, the additional objects $ARBITER[j, snapsn]$ and $ARB\_VAL[j, snapsn]$ are used as follows.

All the simulators invoke $ARBITER[j, snapsn].\mathsf{arbitrate}_{i,j}()$ (at line N06 if q_i is the owner, and line N11 if it is not). According to the specification of the arbiter type, these invocations do not return different values, and do return at least when the owner q_j is correct and invokes that operation (as indicated in the specification, there are other cases where the invocations do terminate). Finally, the value returned indicates if the winner is the owner (1) or not (0).

If the winner is the owner q_j, the value returned by the $snapsn$-th invocations of $\mathsf{e\_sim\_snapshot}_{i,j}()$ (one invocation by simulator) is the value $input_j$ computed by the owner. That value is kept in the atomic register $ARB\_VAL[j, snapsn][1]$ (line N05).

If the owner is not the winner, the value returned is the one determined by the other simulators that invoked $SAFE\_AG[j, snapsn].\mathsf{propose}_i(input_i)$ (line N09) and $SAFE\_AG[j, snapsn].\mathsf{decide}_i()$ (line N10). The value they have computed has been deposited in $ARB\_VAL[j, snapsn][0]$ (line N10), and this value is used as the result of the $SAFE\_AG[j, snapsn]$ object.

It is important to notice that the owner q_j does not invoke the $\mathsf{propose_j}()$ and $\mathsf{decide_j}()$ operations on the objects it owns. Moreover, the simulator q_j is the only simulator that can write $ARB\_VAL[j, snapsn][1]$, while the other simulators can only write $ARB\_VAL[j, snapsn][0]$.

To summarize, if a simulator q_i crashes, it entails the crash of at most one simulated process. This is ensured thanks to the mutex algorithm. If the simulator q_i crashes, $1 \le i \le t + 1$, as far the simulated processes are concerned, it can entail either no crash at all (if q_i crashes outside a critical section), or the crash of p_i (if it crashes while executing $\mathsf{arbitrate}_{i,j}()$ inside the critical section at line N06), or the crash of a process p_j such that $1 \le j \ne i \le t + 1$ (this can occur only if q_j has crashed and was not winner, and q_i crashes inside the critical section at line N09), or the crash of one of the processes p_{t+2}, ..., p_n (if it crashes at line N02 inside the critical section).

t-Resilience vs wait-freedom. Given a BG simulation algorithm where a simulated process p_j ($1 \le j \le t+1 \le n$) can be blocked forever only if simulator q_j crashes, Gafni shows in [12] that wait-freedom and t-resilience are equivalent for decision tasks (he also shows strong results on equivalence between weak renaming and weak symmetry breaking).

7 Proof of the Extended BG Simulation

Lemma 1. *A simulator can block the progression of only one simulated process at a time.*

Proof. A simulator can block the simulation of a process only during the execution of an $\mathsf{e\_sim\_snapshot}()$ operation, when the simulator uses a $\mathsf{safe\_agreement}$ (lines N02-N03 or N09-N10) or an arbiter object because it is its owner (line N06). All these invocations are placed in mutual exclusion. Thus a simulator can block the simulation of only a single process at a time. $\square_{Lemma\ 1}$

Lemma 2. *The simulated process p_i is never blocked at the simulator q_i.*

Proof. The e_sim_snapshot() operation, when invoked by simulator q_i for the simulated process p_i (line N04, $i = j$) does not include any wait statement and does not use a safe_agreement object. Due to the properties of the arbiter object type, it cannot be blocked during its invocation of arbitrate(). Thus, the simulated process p_i can never be blocked at simulator q_i. $\square_{Lemma\ 2}$

Lemma 3. *Each simulator receives the decision value of at least $n - t$ simulated processes.*

Proof. Because at most t simulators may crash, and a simulator can block at most a single simulated process at a time (Lemma 1), each simulator can execute the code of at least $n-t$ simulated processes without being blocked forever. Because the simulated algorithm is t-resilient, these $n - t$ processes will then decide a value. $\square_{Lemma\ 3}$

Lemma 4. *All the simulators that return from the simulation of the k-th snapshot issued by the the simulated process p_j do return the same value for that snapshot.*

Proof. If p_j isn't owned by any simulator $(j > t + 1)$, because of the properties of the safe_agreement objects, the same value is always returned (lines N02-N03 of Figure 5).

If the owner of p_j chooses the value it has computed for p_j's k-th snapshot, it has written this value in $ARB\_VAL[j, snapsn][1]$ (line N05), and is the winner of the arbiter object (line N06). All other simulators will then read its value (line N12).

If the simulated process p_j has an owner but another process chooses the value it has computed for p_j's k-th snapshot, this process has already agreed on a value with all other non owner processes (safe_agreement object, lines N09-N10) and is the winner of the arbiter object (lines N11-N12). All non-owner processes will then write the same value in $ARB\_VAL[j, snapsn][0]$ (line N10) and the owner will read it (line N08).

Thus, all the simulators that return a value for the k-th snapshot of the simulated process p_j return the same value. $\square_{Lemma\ 4}$

Lemma 5. *At most one decision value can be decided by a simulated process on any simulator.*

Proof. Because every simulator computes the same value for any given snapshot and because the snapshot operations are the only non-deterministic parts of codes of the simulated processes, all simulators that decide a value for a given simulated process decide the same value. $\square_{Lemma\ 5}$

Lemma 6. *The sequences of all writes and snapshots for each simulated process correspond to a correct execution of the simulated algorithm.*

Proof. Every simulator that is not blocked while simulating a process simulates it in the same way (same values written and same snapshots read, Lemma 4).

When simulator q_i executes e_sim_snapshot() for p_j (i.e. the simulation of a snapshot for p_j), it stores in its $input_i$ variable the values written by the simulators that have advanced the most for each simulated process (Figure 3 and lines 01-03 of Figure 5). It can choose its own $input_i$ snapshot value only if no other simulator has already ended the execution of this e_sim_snapshot() (Lemma 4 implies that safe_agreement objects have a "memory" effect). Thus, for each e_sim_snapshot(), q_i returns an $input$ value computed by itself or another simulator. Let us notice that, when this $input$ value has been determined, no simulator had terminated its associated e_sim_snapshot(). (If this was not the case, that simulator would have provided the other simulators with its own $input$ value.) Because processes are simulated deterministically, the $input$ value returned contains the last value written by p_j as seen by q_i. This shows that the simulated process order is respected.

To ensure that the simulation is correct, we then have to show that the writes and snapshots of all processes can be linearized. The linearization point of the writes is placed at line 03 of Figure 3 of the first simulator that executes it. The linearization point of the snapshots is placed at line 01 of Figure 5 of the simulator q_i that imposes its $input_i$ value.

Because the simulator q_i that imposes its $input_i$ value in a e_sim_snapshot() operation reads the most advanced values at the time of its snapshot (lines 02-03 of Figure 5), and because once a simulator finishes the execution of e_sim_snapshot(), the value for this e_sim_snapshot() cannot change (Lemma 4), the linearization correspond to a linearization of a correct execution of the simulated algorithm. $\Box_{Lemma\ 6}$

Theorem 1. *The extended BG simulation algorithms described in Figures 3 and 5 are correct.*

Proof. Lemmas 2, 3, 5 and 6 show that the extended BG simulation algorithms presented here are correct. $\Box_{Theorem\ 1}$

Acknowledgments

The authors want to thank E. Gafni, S. Rajsbaum, and C. Travers for interesting discussions on the BG-simulation, and the referees for their constructive comments.

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic Snapshots of Shared Memory. Journal of the ACM 40(4), 873–890 (1993)
2. Afek, Y., Gafni, E., Rajsbaum, S., Raynal, M., Travers, C.: Simultaneous consensus tasks: a tighter characterization of set consensus. In: Chaudhuri, S., Das, S.R., Paul, H.S., Tirthapura, S. (eds.) ICDCN 2006. LNCS, vol. 4308, pp. 331–341. Springer, Heidelberg (2006)
3. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an Asynchronous Environment. Journal of the ACM 37(3), 524–548 (1990)

4. Attiya, H., Rachman, O.: Atomic Snapshots in O($n \log n$) Operations. SIAM Journal on Computing 27(2), 319–340 (1998)
5. Attiya, H., Welch, J.: Distributed Computing, Fundamentals, Simulation and Advanced Topics, 2nd edn. Wiley Series on Par. and Distributed Computing, 414 pages (2004)
6. Borowsky, E., Gafni, E.: Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. In: Proc. 25th ACM Symposium on Theory of Computing (STOC 1993), pp. 91–100. ACM Press, New York (1993)
7. Borowsky, E., Gafni, E., Lynch, N., Rajsbaum, S.: The BG distributed simulation algorithm. Distributed Computing 14(3), 127–146 (2001)
8. Castañeda, A., Rajsbaum, S.: New Combinatorial Topology Upper and Lower Bounds for Renaming. In: Proc. 27th ACM Symposium on Principles of Distributed Computing (PODC 2008), Toronto, Canada, pp. 295–304. ACM Press, New York (2008)
9. Chaudhuri, S.: More *Choices* Allow More *Faults:* Set Consensus Problems in Totally Asynchronous Systems. Information and Computation 105, 132–158 (1993)
10. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM 32(2), 374–382 (1985)
11. Gafni, E.: Round-by-round Fault Detectors: Unifying Synchrony and Asynchrony. In: Proc. 17th ACM Symp. on Principles of Distr. Computing (PODC 1998), pp. 143–152. ACM Press, New York (1998)
12. Gafni, E.: The Extended BG Simulation and the Characterization of t-Resiliency. In: Proc. 41st ACM Symposium on Theory of Computing (STOC 2009). ACM Press, New York (2009)
13. Herlihy, M., Shavit, N.: The Topological Structure of Asynchronous Computability. Journal of the ACM 46(6), 858–923 (1999)
14. Imbs, D., Raynal, M.: Visiting Gafni's Reduction Land: from the BG Simulation to the Extended BG Simulation. Tech Report #1931, IRISA, Université de Rennes (F) (2009)
15. Loui, M.C., Abu-Amara, H.H.: Memory Requirements for Agreement Among Unreliable Asynchronous Processes. In: Par. and Distributed Computing. Advances in Comp. Research, vol. 4, pp. 163–183. JAI Press (1987)
16. Lynch, N.A.: Distributed Algorithms, 872 pages. Morgan Kaufmann Pub., San Francisco (1996)
17. Saks, M., Zaharoglou, F.: Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. SIAM Journal on Computing 29(5), 1449–1483 (2000)