Universal Model Simulation: BG and Extended BG as Examples

Petr Kuznetsov

Télécom ParisTech
petr.kuznetsov@telecom-paristech.fr

Abstract. This paper focuses on simulations as a means of deriving the relative power of distributed computing models. We describe an abstract simulation algorithm that enables reducing the question of solvability of a generic distributed task in one model to an equivalent question in another model. The technique implies simple equivalents to the fundamental reduction by Borowsky and Gafni, known as BG simulation, as well as to Extended BG, a more recent extension of it to colored tasks. We also sketch how the parameters of our technique can be tuned to derive recent equivalence results for models that use, in addition to basic read-write memory, k-set agreement or k-process consensus objects, or make assumptions on active resilience.

1 Introduction

When do we say that a given problem is hard or even impossible to solve? In computing, this typically means that, in a given model of computation, the problem cannot be solved by any algorithm with desired properties. But if we found the answer to the question in one model, can we derive it for another? This is where *simulations* may be handy.

This paper focuses on simulations of distributed computing models. Here a model specifies a collection of computing units, called *processes*, that communicate via invoking operations on shared-memory variables or sending messages. We restrict our attention to a class of decision problems called *tasks*, where every process starts with its private *input* value and is expected, after some information exchange with other processes, to produce an *output* value, so that the inputs and the outputs are consistent with the task specification. For example, in the *consensus* task, the processes are expected to agree on one of the inputs.

In this setting, informally, to simulate a model A in a model B means to guarantee that in every execution of B, the processes in B reach a form of *agreement* on the behaviour of the processes in A that is (1) consistent with some execution of A, and (2) somehow reflects the *inputs* provided to the processes in B. The first condition means that the simulation is *correct*, i.e., it indeed produces something that could have happened in A. The second condition means that the simulation is *useful*, i.e., the simulated run allows the simulators to compute some outputs based on their inputs. These outputs depend on the goal

of the simulation, which in turn depends on the kind of relations between the models we intend to capture.

The main concern of this paper is *distributed computability*: what can and what cannot be computed in a given model. We aim at reductions of the question of whether a task T is solvable in model A to an equivalent question in model B.

We focus first on read-write shared-memory simulations. We assume that processes run the *full-information protocol* using the atomic snapshot memory [1] where every process first writes its input in the memory and then alternates snapshots with updates, where every next update writes the result of the preceding snapshot.

We define an abstract simulation technique in which a set of *simulators* use agreement protocols to reconcile on the evolution of the *simulated* processes. The input of the simulated processes is agreed upon too based on a specific *initialization* rule, which is a parameter of our simulation. A simulator may decide to join the simulation using an *activation* rule based on the inputs of other simulators, which is the second parameter of our simulation. At any point in the simulation, a simulator may decide to terminate, using a specific *termination* rule, which is the third parameter. By varying these three parameters, we can obtain a wide spectrum of computability results.

One application of our abstract techinque is the celebrated result that the question of t-resilient solvability of a colorless task T is equivalent to the question of wait-free solvability of T in a system of $t + 1$ processes [5, 7]. We derive the result by simply allowing each simulator to consider itself active, to use its input value in initializing any simulated process, and terminate as soon as one simulated process outputs.

We then go further and apply our simulation framework to the generic (not necessarily colorless) tasks and show that t-resilient solvability can be reduced to the wait-free solvability (obtained in [13] via *Extended* BG-simulation). We speculate that our technique can be extended to other classes of simulation algorithms, such as adversarial models [10, 16, 21] or models equipped with k-set agreement primitives [15, 14]. In particular, we sketch how a simple modification of parameters in our simulation framework may establish the recently shown equivalence between a system in which any number of processes communicate via read-write memory and k-set agreement objects and a system in which no more than $k - 1$ *active* processes fail (*k-active resilience*).

The rest of the paper is organized as follows. In Section 2, we overview existing model simulations and hint how they can be unified in a common framework. In Section 3, we briefly discuss our basic system model and introduce agreement protocols as principal building blocks of our simulation. In Section 4, we present our simulation framework and in Section 5, we use the framework to derive equivalence results analogous to BG [5, 7] and Extended BG [13]. Section 6 sketches applications of our framework to models beyond read-write t-resilience.

2 Related work

Simulations improve our understanding of distributed computing by establishing equivalence between seemingly different phenomena: synchrony and asynchrony [12], message-passing and read-write shared memory [3], read-write shared memory and atomic snapshot [1], atomic snapshot and immediate snapshot [6], wait-freedom and t-resilience for distributed tasks [7, 13], k-set agreement and k-concurrency [14], wait-freedom and superset-closed adversarial models [18], etc. The motivation behind this paper is to establish a unifying simulation framework that would encompass all existing and emerging equivalence results: by tuning a small set of well-defined parameters of our framework, we should be able produce the desired simulation protocol.

Our abstract simulation builds upon a weak simulation algorithm that always ensures *safety* [2], i. e., it guarantees that the simulated execution indeed could have taken place in the simulated model. A basic building block of the simulation is an agreement protocol, e.g., the safe agreement protocol of [7] or obstruction-free consensus [19, 4]. Since no asynchronous fault-tolerant protocol can achieve agreement providing both safety and liveness [11], the liveness properties of the simulation depend on the liveness properties exported by the agreement protocols it employs. Interestingly, different simulated steps can use different agreement protocols which enables a variety of simulation protocols suitable for different models. The variants of BG and Extended BG proposed in this paper employ obstruction-free consensus [19, 4] as an agreement protocol. Intuitively, to make progress with this kind of agreement protocols, we must ensure that eventually at least some of concurrently simulated processes is driven forward by exactly one correct simulator, which was inspired by the simulations proposed earlier in [17, 9].

Gafni and Guerraoui [15] have recently established that providing the processes with k-set agreement objects is, in a precise sense, equivalent to having access to k state machines, where at least one is guaranteed to progress. In particular, as is informally shown in [14], providing k-set agreement is equivalent, with respect to task solvability, to assuming k-concurrency or active $(k-1)$-resilience. In this paper, we sketch how the latter equivalence result can be seen as a straightforward application of our simulation framework.

3 Model

Processes. We consider a system Π of n processes, $p_1, \ldots, p_n$, that communicate via reading and writing in the shared memory. We assume that the system is *asynchronous*, i.e., relative speeds of the processes are unbounded. Without loss of generality, we assume that processes communicate via an *atomic snapshot* memory [1], where every process may update its dedicated position and take atomic snapshot of the whole memory. More precisely, atomic snapshot memory exports two atomic operations: $update(i, v)$ $(i \in \{1, \ldots, n\})$ that writes value v

to position i, and *scan*() that returns the vector of the most recently written values to positions $1, \ldots, n$.

Simulators. An execution of the processes $p_1, \ldots p_n$ can be *simulated* by a set of *simulators* $s_1, \ldots, s_\ell$ that mimic the steps of the full-information protocol in a *consistent* way: for every execution E_s, there exists an execution E of the full-information protocol on $p_1, \ldots, p_n$ such that the sequence of simulated snapshots for every process p_i in E_s have also been observed by p_i in E.

A process or a simulator may only fail by crashing, and otherwise it must respect the algorithm it is given. A *correct* process or simulator never crashes.

Tasks. In this paper, we focus on a specific class of distributed computing problems, called *tasks* [20]. In a distributed task [20], every participating process starts with a unique input value and, after the computation, is expected to return a unique output value, so that the inputs and the outputs across the processes satisfy certain properties. More precisely, a *task* is defined through a set $\mathcal{I}$ of input vectors (one input value for each process), a set $\mathcal{O}$ of output vectors (one output value for each process), and a total relation $\Delta : \mathcal{I} \mapsto 2^{\mathcal{O}}$ that associates each input vector with a set of possible output vectors. An input $\perp$ denotes a *not participating* process and an output value $\perp$ denotes an *undecided* process.

For example, in the task of *k-set consensus*, input values are in $\{\perp, 0, \ldots, k\}$, output values are in $\{\perp, 0, \ldots, k\}$, and for each input vector I and output vector O, $(I, O) \in \Delta$ if the set of non-$\perp$ values in O is a subset of values in I of size at most k. The special case of 1-set consensus is called *consensus* [11].

We assume that every process runs a *full-information* protocol: initially it writes its input value and then alternates between taking snapshots of the memory and writing back the result of its latest snapshots. After a certain number of such asynchronous rounds, a process may gather enough state to *decide*, i.e., i.e., to produce an irrevocable non-$\perp$ output value. There is no loss of generality in this assumption since the full-information protocol provides at least as much information about the execution as any other protocol.

In *colorless* tasks (also called *convergence* tasks [7]), processes are free to use each others' input and output values, so the task can be defined in terms of input and output *sets* instead of vectors. Formally, let *val*(U) denote the set of non-$\perp$ values in a vector U. In a colorless task, for all input vectors I and I' and all output vectors O and O', such that $(I, O) \in \Delta$, *val*(I) $\subseteq$ *val*(I'), *val*(O') $\subseteq$ *val*(O), we have $(I', O') \in \Delta$. The k-set consensus task is colorless.

Note that to solve a colorless task, it is sufficient to find a protocol (a decision function) that allows just one process to decide. Indeed, if such a protocol exists, we can simply convert it into a protocol that allows every correct process to decide: every process simply applies the decision function to the observed state of any other process and adopts the decision.

The task of (m, k)-*renaming* involves m participating processes (out of n) that are expected to select *names* in the range $\{1, \ldots, k\}$ so that no two processes choose the same name. Renaming is a *colored* (not colorless) task.

Agreement. A basic building block of our simulations is an *agreement* abstraction that can be seen as a safe part of consensus. It exports one operation

propose() taking $v \in V$ as a parameter and returning $w \in V$, where V is a (possibly infinite) *value set*. When a process p_i invokes *propose*(v) we say that p_i *proposes* v, and when the invocation returns v' we say that p_i *decides on* v. Agreement ensures three properties:

(i) every decided value has been previously proposed,
(ii) no two processes decide on different values, and
(iii) if a process decides, then, eventually, every process that takes sufficiently many steps decides.

There are many protocols that satisfy the three properties above, additionally offering some liveness guarantees.

The consensus protocol using the Ω failure detector [8] guarantees that every correct process eventually decides, where Ω, at every correct process, eventually outputs the same identifier of a correct process.

The *BG agreement* protocol [5, 7], guarantees that if every participating process takes enough steps, then eventually every correct participant decides.

The *obstruction-free* consensus protocol (*OF consensus*) [19, 4], which is of special interest for us here, guarantees that a process decides if it eventually runs *solo*, i.e., it eventually encounters no step contention.

4 Abstract simulation

We present our simulation algorithm in a modular way. First, we describe the procedure by which simulators advance one more step of a given simulated process p_i (Section 4.1). The procedure is using an agreement protocol that is, as stated above, safe but not necessarily live. Thus, a correct simulator may block forever in the middle of simulating a step.

Assuming that this procedure is used correctly, i.e., while a simulator is in the middle of simulating a step of p_i, it does not start simulating a new step of p_i, we show that the resulting simulated execution is *consistent* with some execution of the full-information protocol on the simulated processes, where inputs come from the simulators. simulation (Section 4.2).

4.1 Simulating one step

To simulate a step of a given process p_i, every simulator follows the algorithm in Figure 1. First, the simulator takes a snapshot of the current states of the simulated processes (line 1). The states of the simulated systems is stored in an atomic snapshot object St. Each simulator s_j stores its view of the states of all simulated processes in position j of St. The *getState*() call returns the vector of most recent simulated states, and the *updateState*$(i, [s, k+1])$ performed by s_j updates the state of process p_i in jth position of St. Both operations can be easily implemented using atomic snapshot memory shared by the simulators.

If p_i has not yet performed a single simulated step, then p_i's evaluated state is just its input value (line 4). Note that the procedure by which a simulator

Shared variables:
$A_1^i, A_2^i, \ldots$, agreement protocols { *used to simulate steps* $1, 2, \ldots$ *of* p_i }
St, atomic snapshot object, initially $[\perp, 0], \ldots, [\perp, 0]$

To simulate the next step of p_i:
1 $[S, K] := St.getState()$ { *get the most recent simulated states* }
2 $k := K[i]$ { *get the number of simulated steps of* p_i }
3 **if** $k = 0$ **then**
4 $s :=$ *p_i's input value (using the provided initialization procedure)*
5 **else**
6 $s := S$ { *the current simulated state* }
7 $s := A_{k+1}^i.propose(s)$ { *agree on the next state of* p_i }
8 $St.updateState(i, [s, k+1])$

Fig. 1. Safety: simulating one step of process p_i

chooses the input for p_i is a parameter of the simulation, and we give concrete examples in Sections 5.1 and 5.2. Otherwise, the state of p_i is evaluated as the result of the last snapshot of the simulated state (line 1).

Then the simulated process p_i is driven forward using a new instance of agreement A_{k+1}^i. When A_{k+1}^i returns s, the simulator *publishes* $[s, k+1]$ as the simulated state of p_i resulted after its $(k+1)$th snapshot (line 8) .

We assume that every simulator is *well-formed*: it never starts simulating a new step of p_i (using the algorithm in Figure 1) if it has not yet computed an input value for p_i and it is not yet done with simulating p_i's previous step. Respectively, an execution is *well-formed* if every simulator is well-formed in it. We say that a process p_i takes r simulated steps in a well-formed execution if at least one simulator returned from an invocation of $A_r^i.propose(s)$ (line 7).

Correctness. Now we show that any well-formed execution of the simulation using the agreement protocol in Figure 1 indeed produces an execution of the full-information protocol on $p_1, \ldots, p_n$. More precisely, the sequence of simulated snapshots obtained by every process p_i settled by the agreement protocols $A_1^i, A_2^i, \ldots$ in the well-formed execution could have been indeed observed in some execution of the full-information protocol.

Any well-formed execution of the algorithm exports, for each process p_i, a sequence of *states* $v_1^i, v_2^i, \ldots$ of the full-information protocol returned by the agreement instances $A_1^i, A_2^i, \ldots$, each next state "extending" the previous one. This sequence is well defined, since each of these agreement instances returns at most one value and the instances are used one-by-one.

Intuitively, if the sequence of states $v_1^i, v_2^i, \ldots$ is finite (only finitely many invocations of $A_1^i, A_2^i, \ldots$ return), p_i takes only finitely many steps in the simulation. Otherwise, if the sequence of p_i's states is infinite, then p_i appears correct (takes infinitely many steps) in the simulated execution.

Now we say an execution E_s of the algorithm in Figure 1 is *consistent with* an execution E of the full-information protocol for processes $p_1, \ldots, p_n$ if, for

each p_i, the sequence of states $v_1^i, v_2^i, \ldots$ exported to p_i by E_s is the sequence of states of p_i observed in E.

Lemma 1. *Let E_s be any well-formed execution on $s_1, \ldots, s_\ell$ using the algorithm in Figure 1. There exists an execution E of the full-information protocol on $p_1, \ldots, p_n$, such that E_s is consistent with E and the input of every process participating in E is proposed in E_s by some simulator in line 4.*

Proof. Given a well-formed execution E_s, we construct the corresponding execution E of the full-information protocol as follows.

First of all, we observe that, thanks to the use of agreement protocols, the evolution of the state of every simulated process is observed consistently by all simulators: the outcome of the kth snapshot of each process p_i is witnessed in the same way by all simulators in E_s.

The inputs of each simulated process is agreed using the agreement protocol in line 7, where each decided value is chosen by some simulator in line 4.

Further, since all snapshots of St are totally ordered, we can also totally order all snapshots of the simulated states that were agreed in line 7 for some simulated processes, so that every next snapshot contains the preceding one. Thus, all snapshots obtained by the simulated processes in E_s are related by *containment*.

Also, a simulator only accesses agreement protocol A_{k+1}^i if it observes (in line 1) that p_i performed k simulated snapshots so far. Since the full-information state proposed by a simulator to the agreement protocol in line 7 contains p_i's most recent snapshot, we also have *self-inclusion*: every simulated snapshot of a process p_i contains the most recent update of p_i.

Now we construct E as follows: we place simulated snapshot operations in E_s respecting the containment order, and then place each kth update of a process p_i before the first snapshot operation that returns a vector containing the $(k-1)$th snapshot of p_i. Here the first update operation of p_i simply writes the input of p_i. □

4.2 Simulating a run using OF consensus

Our simulation algorithm using obstruction-free (OF) consensus as an agreement protocol is presented in Figure 2. The simulation is parameterized by:

- The *initialization* condition: how a simulator computes an input of any process it is simulating (line 9).
- The *activation* condition: how a simulator decides when to participate in the simulation (line 10).
- The *termination* condition: how a simulator decides when to depart from the simulation (line 28).

Concrete examples of how these parameters can be defined are given in Sections 5.1 and 5.2. For now we only assume that the activation and termination conditions are publicly known: each simulator can look at the simulated state

St and decide which simulators are active and which are able to terminate. We assume that the conditions are *monotone*: if a condition holds given the current snapshot of the simulated state, it holds given any subsequent snapshot in the simulation.

In our simulation, each simulator s_i first *registers* its input (line 9) and then waits until it is *activated*. Once s_i becomes active, it writes 1 as its round number in register R_i (line 14). Therefore, we say that a simulator s_i is *active* in a given state of the simulations if $R_i \neq \perp$.

Every active simulator s_i proceeds in rounds. In each round, s_i picks one simulated process p_ℓ and tries to move it forward using the algorithm in Figure 1 with OF consensus objects [19, 4] as the agreement abstractions. The simulated process p_ℓ is chosen based on the following rule:

- s_i computes the set U of currently active (but not yet terminated) simulators. Let $m = |U|$.
- s_i computes its *rank* k in U, i.e., the number of simulators in U with ids $j \leq i$.
- If $m \leq n$, i.e., the number of simulators in U does not exceed the number of processes to simulate, s_i chooses p_ℓ as the k-th smallest process in $S^m_{r \bmod \binom{n}{m}}$, the "next" set of m simulated processes. Here, for all m, the set of process subsets of size m is ordered as $S^m_0, ..., S^m_{\binom{n}{m}-1}$. (Note that the simulation may block if the number of active simulators exceeds the number of simulated processes, which is not extremely surprising.)

The simulation of p_ℓ in round r is performed until p_ℓ moves forward (takes one more simulated snapshot according to the algorithm in Figure 1) or another simulator reaches a round higher than r. Note that in the special case when the input value for p_ℓ is not yet known, the simulation simply moves round $r + 1$.

I_i is an n-vector that contains inputs for processes in Π proposed by simulator s_i. If the jth position in the vector is $\perp$, it means that s_i does not have an input for p_i. The inputs given to different simulators do not have to be mutually consistent: different simulators can be given different input values for the same simulated process. We say that a process p_j is *initialized* if in a given execution of our simulation, at least one simulator s_i has written a non-$\perp$ value in the jth position of I_i (line 9).

Note that the simulation is well-formed: no simulator starts simulating a step of a process p_i before it finishes simulating p_i's previous step. Thus, by Lemma 1, it produces a correct execution of the full-information protocol where every participating simulated process starts with an input proposed by one of the active simulators.

Moreover, our abstract simulation ensures the following property that will be instrumental (in our concrete examples in Section 5.1 and 5.2).

Theorem 1. *If, eventually, there are exactly m active and not terminated simulators, at least one of which is correct, and at least $\ell \geq m$ processes are initialized, then at least $\ell - m + 1$ processes take infinitely many steps in the simulated execution.*

Shared variables:
  I_i, for each simulator s_i, initially $\perp$
  R_i, for each simulator s_i, initially $\perp$

Code for each simulator s_i with input V:
9 $I_i := \mathit{initialize}(V)$ { *the initialization rule* }
10 wait until $\mathit{active}()$ { *the activation rule* }
11 $r := 0$
12 **repeat**
13 $r := r + 1$
14 $R_i := r$
15 **repeat**
16 $U :=$ active and not yet terminated simulators
17 $m := |U|$
18 $k :=$ rank of s_i in U
19 **if** $m \leq n$ **then**
20 $p_\ell :=$ the k-th process in $S^m_{r \bmod \binom{n}{m}}$ { *pick a process in* $S^m_{r \bmod \binom{n}{m}}$ }
21 **if** p_ℓ is initialized in one of $I_1, \ldots, I_n$ **then**
22 run one step of the algorithm in Figure 1 for p_ℓ using OF consensus (start a new snapshot simulation if done with the previous one)
23 **else**
24 break
25 **else**
26 break
27 **until** $\exists s_j : R_j > r$ **or** p_ℓ moves forward
28 **until** decided in St { *the termination rule* }
29 *return the output*

Fig. 2. Abstract simulation: the code of each simulator s_i

Proof. We show first that every correct simulator proceeds through infinitely many rounds of the algorithm in Figure 2. Suppose, without loss of generality, that r is the smallest round that is never completed by some correct simulator s_i. Since a simulator completes round r as soon as it observes that another simulator reaches a higher round (line 28), we derive that every correct process is blocked forever in round r.

Consider the moment after which the set W of active and not yet terminated simulators is of size m. Recall that the termination condition for each simulator is publicly known. Therefore, there is a time after which every correct simulator evaluates the set of such simulators as W in line 16.

Every simulator with rank k in W chooses kth process in the set $S^m_{r \bmod \binom{n}{m}}$ to simulate. Since no simulator reaches a round higher than r by breaking in line 24, we observe that every correct active simulator is blocked in simulating a step of an initialized process. But there are exactly m processes in $S^m_{r \bmod \binom{n}{m}}$,

thus, eventually, at most one simulator is promoting every initialized process in $S^m_{r \bmod \binom{n}{m}}$.

Since we use OF consensus as the agreement protocol in Figure 1 and s_i is the only process to take steps of the protocol, eventually, the agreement protocol in line 7 returns. Thus, s_i eventually simulates one more step of its process in $S^m_{r \bmod \binom{n}{m}}$ and moves to round $r+1$—a contradiction.

Thus, a correct simulator s_i goes through infinitely many rounds of the algorithm in Figure 2. Suppose, by contradiction, that there is a set W of m initialized simulated processes that take only finitely many steps in the resulting simulated execution. Note that since all sets of size m are continuously explored in the round-robin fashion, eventually, s_i infinitely often reaches round r such that $S^m_{r \bmod \binom{n}{m}} = W$. Since all processes in W are initialized, at least one of them is takes at least one simulated step. By repeating this argument, we derive that at least one process in W takes infinitely many simulated steps—a contradiction.

Thus, at most $m-1$ initialized processes can take only finitely many steps in the simulated execution, i.e., at least $\ell - m + 1$ processes take infinitely many steps. □

5 Applications

Now we apply our abstract simulation to establish the equivalence of t-resilient systems and wait-free $(t+1)$-process systems, first for colorless tasks [5, 7] and then for generic (colored) tasks [13].

5.1 BG simulation: characterizing t-resilient solvability of colorless tasks

BG simulation [5, 7] is a technique by which $k+1$ simulators $s_1, \ldots, s_{k+1}$ $(k < n)$ can wait-free simulate a *t-resilient* execution on processes $p_1, \ldots, p_n$ $(n > t)$ of a protocol solving a colorless task. The technique is applied to derive the following result which we now obtain using our abstract simulation:

Theorem 2. *A colorless task T is t-resiliently solvable if and only if it is wait-free solvable by $t+1$ processes.*

Proof. The "if" part of this result is straightforward. Suppose that T is solvable wait-free by $t+1$ processes. We just let processes $p_1, \ldots, p_{t+1}$ run the wait-free algorithm, where each p_i $(i = 1, \ldots, t+1)$ runs the algorithm of s_i. As soon as a process in $\{p_1, \ldots, p_{t+1}\}$ decides, it posts its decision value in the shared memory. Every process periodically checks the memory, and returns the first decision value it finds. In any t-resilient execution, every correct process returns.

To obtain the "only if" part, suppose that there is a t-resilient solution of T on processes $p_1, \ldots, p_n$. We want to show that T can thus be solved wait-free by

$s_1, \ldots, s_{t+1}$. To use our abstract simulation, we need to specify the initialization, activation, and termination parameters for the algorithm in Figures 1 and 2.

To initialize simulated processes, s_i puts its input value v_i in all positions of the vector I_i: $I_i = [v_i, ..., v_n]$. Every simulator s_i that reached line 10 is considered active.

The termination condition is also straightforward. A simulator terminates as soon it observes (in line 28) a simulated state St in which some simulated process decides. The simulator then simply returns the value decided by the simulated process.

By Theorem 1, the resulting simulated execution is going to be t-resilient and, thus, eventually, some simulated process must decide. Therefore, every correct simulator eventually decides and, since the task is colorless and the inputs of the simulated processes come from the simulators, the decisions of the simulators are consistent with the inputs in regard to the task specification. □

5.2 Extended BG simulation: characterizing generic tasks

In EBG [13], any task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ defined for n processes $p_1, \ldots, p_n$, is associated with a task $T' = (\mathcal{I}', \mathcal{O}', \Delta')$ defined for $t+1$ simulators $s_1, \ldots, s_{t+1}$ as follows:

- In every input vector $I' \in \mathcal{I}'$, each simulator s_i is given a set of input values for p_i and $n-t-1$ processes with ids higher than i. No two simulators are given different input values for the same process.
- In every output vector $O' \in \mathcal{O}'$, each simulator s_i obtains a set of output values for p_i and $n-t-1$ processes with ids higher than i. No two simulators obtain different output values for the same process.
- For every $(I', O') \in \Delta'$, the corresponding input vector I and output vector O for processes in Π satisfy $(I, O) \in \Delta$.

We apply our abstract simulation to derive the following result originally presented in [13]:

Theorem 3. *T can be solved t-resiliently if and only if T' can be solved wait-free.*

Proof. In both directions we use the simulation described in Figures 1 and 2 where the agreement protocols are instances of OF consensus. Recall that, in addition to the three properties of agreement, OF consensus guarantees that every process that, from some point on, runs solo eventually decides.

The "only if" part. Suppose we are given an algorithm that t-resiliently solves T. In a wait-free solution of T' for $t+1$ simulators $s_1, \ldots, s_{t+1}$, every simulator that reached line 10 is considered active. As in Section 5.1, every simulator s_i simply uses its input in T' to initialize its vector I_i. Then every active s_i runs the algorithm in Figures 1 and 2 until it observes outputs for p_i and $n-t-1$ processes with ids higher than i (the termination condition in line 28).

Initialization (line 9):
$I_i := V$ { V *is* s_i*'s input of* T'*, a vector of at least* $n-t$ *input values of* T }
Activation (line 10):
true
Termination (line 28):
p_i and $n-t-1$ processes with ids higher than i decided in St

Fig. 3. The parameters of $T \Rightarrow T'$ (lines 9, 10 and 28 in Figure 2)

Again, we show first that every correct simulator eventually terminates. Suppose, by contradiction, that there is a set W of exactly $m \geq 1$ active simulators that never terminate, and at least one of them is correct. Since there are m active simulators, exactly $\ell \geq n-t+m-1$ simulated processes are initialized.

By Theorem 1, in the resulting execution, at most $m-1 \leq t$ initialized processes take only finitely many steps in the resulting simulated execution. Since the simulated algorithm is t-resilient, we derive that at most $m-1$ initialized processes never decide.

Let $\{s_{i_1}, \ldots, s_{i_m}\}$ be the simulators in W sorted in the order of increasing ids. Note that there are at least $n-t+m-1$ initialized processes with ids i_1 and higher, and at most $m-1$ of them never decide. Hence, s_{i_1} eventually observes at least $n-t$ decided processes with ids i_1 or higher. We derive that p_{i_1} never decides, otherwise s_{i_1} would observe that p_{i-1} and $n-t-1$ processes with ids higher than i_1 are decided and terminate. Inductively, s_j observes at least $n-t+m-j$ initialized decided processes with ids i_j and higher and at most $m-j$ of them never decide. But for $j=m$ this gives at least $n-t$ decided processes with ids i_m and higher, and, thus, s_{i_m} terminates—a contradiction.

Thus, eventually, every correct simulator outputs.

The "if" part. Now suppose we are given a wait-free solution of T' for processes $p_1, \ldots, p_{t+1}$. We derive a t-resilient solution for T on n simulators $s_1, \ldots, s_n$ as follows.

Every simulator registers its participation by writing its input of T in the shared memory. As soon as a simulator $s_i \in \{s_1, \ldots, s_{t+1}\}$ witnesses the participation of at least $n-t-1$ processes with ids higher than its own, it joins the simulation of the wait-free algorithm solving T'. Respectively, in the simulation, a process p_i is considered initialized if s_i and $n-t-1$ simulators with ids higher than i have posted their inputs of T.

Note that, initially, every active simulator $s_i \in \{s_1, \ldots, s_{t+1}\}$ corresponds to a distinct initialized simulated process p_i. Thus, the number of active not yet terminated simulators does not exceed the number of simulated processes, and, by Theorem 1, at least one simulated process takes sufficiently many steps to decide (the simulated protocol is wait-free).

Once at least one simulated process decides (i.e., at least $n-t$ participating simulators terminate), every simulator s_i without output (whether it is in $\{s_1, \ldots, s_{t+1}\}$ or not) joins the simulation and runs it until some simulated pro-

Initialization (line 9):
$I_i := V$ { V *is* s_i*'s input in* T }
Activation (line 10):
if $s_i \notin \{s_1, \ldots, s_{t+1}\}$ **then**
wait until at least one simulated process in $\{p_1, \ldots, p_{t+1}\}$ decides
Termination (line 28):
some p_j decides with an output containing the value of s_i

Fig. 4. The parameters of $T' \Rightarrow T$ (lines 9, 10 and 28 in Figure 2)

cess produces an output for it (see the activation procedure in Figure 4). A simulator terminates as soon as its output is produced by some decided simulated process (the termination condition in 28).

Again, suppose, by contradiction that there are $\ell > 0$ participating simulators that never decide, at least one of which is correct. We observe first that there are at least ℓ initialized processes in $\{p_1, \ldots, p_{t+1}\}$ that never terminate. Indeed, since $n - t$ simulators decided in the first phase of our simulation, the total number of participating simulators is $n - t + k \geq n - t + \ell$, where k is the exact number of participating simulators in $\{s_1, \ldots, s_{t+1}\}$.

Note that, since exactly ℓ out of m simulators are undecided in the current simulation, exactly ℓ out of k initialized simulated processes never terminate. Indeed if we imagine that $k - \ell + 1$ out of k initialized simulators terminate, the total number of decided simulators must be $n - t + k - \ell + 1$ which, together with the ℓ participating simulators that never decide gives $n - t + k + 1$ participating simulators in total.

By Theorem 1, at most $\ell - 1$ simulated process takes only finitely many steps in the simulated execution. Thus, at least one of the ℓ never terminated simulators take infinitely many steps, and, since the simulated protocol is wait-free, eventually decides—a contradiction.

Thus, our construction indeed ensures that every live simulator eventually decides. Since the decision come from an execution of a protocol solving T' with the same inputs, the solution is correct with respect to T. □

6 Concluding remarks and speculations

This paper proposes a simple and intuitive simulation technique that is general enough to derive a wide variety of models equivalence results. At a high level, the technique maintains the invariant that the number of simulators conincides with the number of currently simulated processes. Therefore, as long as there is a live simulator, at least one of the simulated processes makes progress. To maintain the invariant, a terminated process may bring the number of simulators by one. As our algorithms in Section 5 suggest, multiple existing and new equivalence results can be established by simply parameterizing initialization, activation, and termination rules in our simulation framework.

Below we briefly sketch how the equivalence between the "generalized k-state machine"' [15] and active $(k-1)$-resilience. Sorting out details of the sketched algorithms and proving their correctness is left for (immediate) future work. We also show how to extend the technique to models equipped with k-process consensus objects.

Bounded active resilience. Suppose that we have a protocol solving task T assuming that at most $k-1$ active processes may fail. Recall that a process is considered active if it has started the protocol and have not yet output. Without loss of generality, we assume that in the first step of the protocol, each process registers its input value in the shared memory. Thus, at any point of the execution, active processes are not yet terminated processes whose input values are registered.

In a $(k-1)$-active resilient model, we can easily solve k-set agreement as follows. Assuming active $(k-1)$-resilience, the active processes simulate the first steps of at most k processes $q_1, \ldots, q_k$. As soon as the first step of at least one simulated process q_i is completed, every process can decide on the posted value. A process participates in the simulation as long as it is among the first k active processes. In that case, the process with rank $\ell \in \{1, \ldots, k\}$ is assigned to simulate process q_ℓ. If there are k or more active processes with smaller ids, then the process simply waits until a decision value is posted. Since at most $k-1$ active processes may fail, at least one simulated process will eventually complete its first step, and every correct process will eventually decide.

In the other direction, we employ the simulation algorithm of Section 4.2 run on $\min(k, \ell)$ state machines, where ℓ is the number of active processes. The construction of [15] guarantees that at most $\min(k, \ell) - 1$ state machines may stall. Moreover, as long as there is at least one correct active process, at least one machine makes progress by simulating an active k-resilient execution. Therefore, every correct active process eventually terminates.

Beyond read-write. It is straightforward to extend our colorless simulations to the models where simulators can use k-process consensus objects so that, e.g., ℓ simulators can simulate a system of $\lceil \ell/k \rceil$ processes in the wait-free manner.

Indeed, consider the one-step simulation in Figure 1, where the agreement protocol A^i_{k+1} is augmented with k-processes consensus. "Augmented" means here that the protocol additionally guarantees that if at least one process among the first k to access it is correct, then every correct simulator returns. We can easily implement such an abstraction using k-process consensus object and read-write registers.

Now we apply our abstract simulation in Figure 2 and observe that a simulated process can only block forever if some k faulty simulators died in the middle of its simulation. As long as there is one correct simulator, at most $\lceil k/\ell \rceil - 1$ simulated processes can fail.

Acknowledgements. The author is grateful to Eli Gafni for multiple discussions on model simulations and to Armando Castañeda for sharing his confusions about the original EBG algorithm [13], which inspired deriving the technique proposed in this paper.

References

1. Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.
2. B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, Oct. 1985.
3. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. *J. ACM*, 42(2):124–142, Jan. 1995.
4. H. Attiya, R. Guerraoui, D. Hendler, and P. Kuznetsov. The complexity of obstruction-free implementations. *J. ACM*, 56(4), 2009.
5. E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100. ACM Press, May 1993.
6. E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, pages 41–51, New York, NY, USA, 1993. ACM Press.
7. E. Borowsky, E. Gafni, N. A. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.
8. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.
9. C. Delporte-Gallet, H. Fauconnier, E. Gafni, and P. Kuznetsov. Wait-freedom with advice. In *PODC*, pages 105–114, 2012.
10. C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and A. Tielmann. The disagreement power of an adversary. *Distributed Computing*, 24(3-4):137–147, 2011.
11. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.
12. E. Gafni. Round-by-round fault detectors (extended abstract): Unifying synchrony and asynchrony. In *Proceedings of the 17th Symposium on Principles of Distributed Computing*, 1998.
13. E. Gafni. The extended BG-simulation and the characterization of t-resiliency. In *STOC*, pages 85–92, 2009.
14. E. Gafni and R. Guerraoui. Simulating few by many: Limited concurrency = set consensus. Technical report, UCLA, 2009. http://www.cs.ucla.edu/∼eli/eli/kconc.pdf.
15. E. Gafni and R. Guerraoui. Generalized universality. In *Proceedings of the 22nd international conference on Concurrency theory*, CONCUR'11, pages 17–27, Berlin, Heidelberg, 2011. Springer-Verlag.
16. E. Gafni and P. Kuznetsov. On set consensus numbers. *Distributed Computing*, 24(3-4):149–163, 2011.
17. E. Gafni and P. Kuznetsov. Relating L-Resilience and Wait-Freedom via Hitting Sets. In *ICDCN*, pages 191–202, 2011.
18. E. Gafni and S. Rajsbaum. Distributed programming with tasks. In *OPODIS*, pages 205–218, 2010.
19. M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, 2003.
20. M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(2):858–923, 1999.
21. P. Kuznetsov. Understanding non-uniform failure models. *Bull. Eur. Assoc. Theor. Comput. Sci. EATCS*, 106:54–77, 2012.