# Understanding the Set Consensus Partial Order Using the Borowsky-Gafni Simulation *
# (Extended Abstract)

Soma Chaudhuri and Paul Reiners

Iowa State University, Ames, IA 50011, USA

**Abstract.** We present a complete characterization of the Set Consensus Partial Order, a refinement of the Consensus Hierarchy of Herlihy. We define the $(n,k)$-*set consensus problem* as the $k$-set consensus problem for $n$ processors. We then answer the question of whether an $(n,k)$-set consensus object (an object which solves the $(n,k)$-set consensus problem) can be implemented using a combination of $(m,\ell)$-set consensus objects and snapshot objects, for all possible values of $n, k, m, \ell$, creating a *partial order* of set consensus objects. The model we consider is the asynchronous shared memory model.

To prove our results, we use the Borowsky-Gafni Simulation technique, a powerful tool which has been used to prove several impossibility results about shared memory algorithms. Lynch and Rajsbaum gave a formal description of the basic technique, along with a proof of its correctness. We extend their results to include simulations of algorithms which access set consensus objects. Our description of the simulation, and its proof of correctness, are also in terms of I/O Automata. We need this stronger version of the simulation algorithm to obtain our results on the Set Consensus Partial Order. We state a general Simulation Theorem which specifies the properties of the simulation, and characterizes all the impossibility results that can be obtained using this technique. Our partial order result can then be derived as a special case of this theorem.

## 1 Introduction

The CONSENSUS PROBLEM is a fundamental problem in distributed computing, where a set of $n$ processors communicate among each other to decide on a common output value from among their input values. While the problem itself seems simple, in a surprising result, Fischer *et. al.* [6] showed that in a *totally asynchronous* system, the CONSENSUS PROBLEM could not be solved deterministically even in the presence of only *one* fail-stop fault.

The $k$-SET CONSENSUS PROBLEM, a generalization of CONSENSUS introduced by Chaudhuri [5], requires each processor to decide on some processor's input value as its output (this is called the *validity condition*), and the set of values decided upon must be of size at most $k$ (this is called the *$k$-agreement condition*).

* Supported in part by NSF grant CCR-93-08103.

Chaudhuri conjectured that the $k$-SET CONSENSUS problem is unsolvable in the presence of $k$ fail-stop faults, while showing that it was solvable in the presence of $k-1$ faults. Since the 1-SET CONSENSUS problem is equivalent to the CONSENSUS problem, this is a generalization of the FLP impossibility result, and it was later proven correct by three teams of researchers, Borowsky and Gafni [2], Herlihy and Shavit [9], and Saks and Zaharoglu [15].

In his seminal paper on wait-free synchronization [7], Herlihy introduced the consensus hierarchy by defining an $n$-consensus object, an object which solves the consensus problem for $n$ processors. Herlihy showed a hierarchy among these objects, and his result was later extended by Jayanti and Toueg [13] to show a strict linear hierarchy among the $n$-consensus objects for all values of $n$. In particular, they showed that there is a wait-free implementation of an $n$-consensus object, using any number of read/write objects and $m$-consensus objects, if and only if $n \leq m$.

Continuing along those lines in trying to refine this hierarchy, we study the $(n,k)$-set consensus problem (and its corresponding $(n,k)$-set consensus object), as defined by Borowsky and Gafni [3]. The *$(n,k)$-set consensus problem* is the $k$-set consensus problem for $n$ processors. An *$(n,k)$-set consensus object* (henceforth referred to as an *$(n,k)$-setcon* object) is a data object which supports a one-time computation of $k$-set consensus by up to $n$ processors. The object has to be wait-free, meaning that it is $(n-1)$-resilient, where the number of processors that can access it is $n$.

We answer the question of whether an *$(n,k)$-setcon* object can be implemented *in a wait-free manner* by any combination of *$(m,\ell)$-setcon* objects and snapshot objects. Since snapshot objects can be implemented by read/write registers [1], this restriction imposed on implementations is consistent with the $h_m^r$ hierarchy defined by Jayanti [10], where an object $A$ can be implemented by any combination of objects $B$ and read/write registers. We show the relationship (that is, whether a wait-free implementation is possible or not) between every pair of *setcon* object types. We determine that the relationships define a *Set Consensus Partial Order* rather than a hierarchy because we will show the existence of pairs of objects, neither of which can implement the other. We, thus, completely characterize these relationships in the theorem stated below (the $\preceq$ relation stands for "can be implemented by").

**Theorem 1. [Partial Order Theorem]** *Let $n$, $k$, $m$, and $\ell$ be any positive integers. Then*

1. IF $n \leq k$, THEN $(n,k)$-*setcon* $\preceq$ $(m,\ell)$-*setcon*,
2. IF $n > k$ AND $m \leq \ell$, THEN $(n,k)$-*setcon* $\not\preceq$ $(m,\ell)$-*setcon*,
3. IF $n > k$ AND $m > \ell$, THEN
   (a) IF $k \geq \ell\lceil n/m \rceil$, THEN $(n,k)$-*setcon* $\preceq$ $(m,\ell)$-*setcon*,
   (b) IF $k \geq \ell\lfloor n/m \rfloor + (n - m\lfloor n/m \rfloor)$, THEN $(n,k)$-*setcon* $\preceq$ $(m,\ell)$-*setcon*,
   (c) IF $k < \ell\lceil n/m \rceil$ AND $k < \ell\lfloor n/m \rfloor + (n - m\lfloor n/m \rfloor)$, THEN $(n,k)$-*setcon* $\not\preceq$ $(m,\ell)$-*setcon*.

The first two cases are pretty straightforward. The interesting point about the third case is that the intuition behind the impossibility result of the last sub-case is based on the algorithms of the two other sub-cases.

Borowsky and Gafni [3, 4] were the first to consider the question of whether a certain set consensus object can implement another. They obtained some partial results, including both impossibility results and algorithms (cases 3(a) and (b) of the Partial Order Theorem). Their impossibility results use a powerful simulation technique. Our result described above is an extension of their work and uses the same technique. The same result is obtained by Herlihy and Rajsbaum [8] using more complicated topological techniques.

The impossibility results we obtain are derived using the Borowsky-Gafni simulation technique, a powerful tool for studying possibility and impossibility results in the asynchronous shared memory system with failures. It is therefore important to understand exactly what the simulation allows. This was not completely clear in the informal presentation of the technique by Borowsky and Gafni, which left open some questions. To answer these questions, Lynch and Rajsbaum [11] recently studied the simulation technique and came up with a precise, formal description of the basic technique, in terms of I/O automata, along with a formal proof of correctness.

We extend the results of Lynch and Rajsbaum, also using I/O automata, to include simulations of algorithms which access *setcon* objects, since we will be using such simulations in obtaining our results on the Set Consensus Partial Order. We present our results about the simulation as a Simulation Theorem—a general characterization of the properties of the simulation—which characterizes all the results that can be obtained using the technique. Our results can then be derived as a special case of the Simulation Theorem. What makes the Simulation Theorem easy to use is that, while it is proven using I/O automata, it is stated in more general terms, which makes it easy to see in which situations it can be applied.

Lynch and Rajsbaum [11] used their results to come up with a notion of a fault-tolerant reducibility between any two problems. Our Simulation Theorem answers the question of whether a simulation of an algorithm for Problem $P_2$ *for* a set of $n_2$ processors with at most $f_2$ faults *by* a set of $n_1$ processors with at most $f_1$ faults will solve Problem $P_1$ for the set of $n_1$ processors. We specify what we mean for an $n_1$-processor system to successfully simulate an algorithm for an $n_2$-processor system. Our Simulation Theorem differs from the $f$-reducibility of [11] in several ways. It is more general in that it lets the number of faults allowed in the simulating system be different from the number of faults allowed in the simulated system, due to the additional implementations of set consensus objects. It is therefore not a reducibility in the same sense, since the simulation needs to implement these objects not provided by the simulated system. Also, while $f$-reducibility focuses on the *decision problems* themselves, our Simulation Theorem looks at the *algorithms* for the decision problems. This makes our result less abstract in this respect, but also gives us some advantages. Specifically, focusing on the algorithm instead of the problem allows our Simulation Theorem

to express both the safety and liveness properties of the technique, while $f$-reducibility is restricted to the safety properties.

## 2 The Model

We use the I/O Automata model of Lynch and Tuttle [12], which we briefly review here. An I/O automaton is a simple state machine, where the transitions are *actions*, classified as *internal, input*, or *output*. There is a *fairness* constraint on the executions, requiring that every enabled non-input action be given a fair turn.

We assume an *asynchronous shared memory system* with *snapshot variables*. The shared memory is accessed by *snap* and *update* operations. We assume that, given a set of $n_1$ processors, the memory consists of $n_1$ components. A *snap* operation by any processor will return the value of the entire memory, *i.e.*, all $n_1$ components. An *update*$(x)$ operation by the processor $i$ changes the value of the $i$th component to $x$. In particular, only processor $i$ can change the value of the $i$th component of memory while all other processors can read it.

## 3 The Borowsky-Gafni Simulation

We now give an informal overview of the Borowsky-Gafni simulation technique. The technique allows $n_1$ real processors to each simulate $n_2$ programs or sequences of code (the simulated processors), so that the $n_1$ sets of simulations are consistent with one another, as well as consistent with a real execution of the $n_2$ programs, even though the programs may invoke nondeterministic objects. We assume that the $n_2$ programs can solve some decision problem $P_2$, and a simulation of the $n_2$ programs, by the $n_1$ real processors, solves some other decision problem $P_1$. We allow for fault-tolerance; we assume that the set of $n_2$ programs is resilient to $f_2$ fail-stop faults, while the set of $n_1$ simulating processors may have as many as $f_1$ such faults.

The $n_1$ real processors each simulate a shared-memory algorithm $P_2$ involving $n_2$ programs. We now define what we mean by a simulation. Specifically, each real processor $i$ has a function $g_i$ which maps its input value $x_i$ to an $n_2$-length vector of proposed input values for the $n_2$ programs $j$. The execution simulated adopts the input value proposed by one of the processors $i$ for each program $j$. At the termination of the simulated execution, each processor $i$ observes a certain $n_2$-length vector of the outputs of the programs $j$ (with as many as $f_2$ null entries). Each of these vectors are consistent with each other in that, for all $k$, all non-null entries in the $k$th position of each vector are the same. Also, each output vector consists of a set of valid outputs for the programs $j$. Now, for each $i$, a function $h_i$ maps the $n_2$-length vector observed by processor $i$ to the decision value $y_i$ for processor $i$. These mapping functions follow the lines of the reducibility defined in [11].

**Definition 2.** An *f-fault tolerant decision problem* $P = \langle \mathcal{I}, \mathcal{R}, \Delta \rangle$ *for a distributed system of n processors* is a set, $\mathcal{I}$, of vectors of length $n$, a set, $\mathcal{R}$, of vectors of length $n$, each of which may have up to $f$ null entries, and a set $\Delta \subseteq \mathcal{I} \times \mathcal{R}$. The intuitive idea is that $\mathcal{I}$ is the set of possible inputs to the system, $\mathcal{R}$ is the set of possible outputs of the system, with null entries corresponding to processors which have failed, and $(I, R) \in \Delta$, if $R$ is a correct solution of problem $P$ with input $I$.

**Definition 3.** An algorithm $A$ *solves* an $f$-fault tolerant decision problem $P = \langle \mathcal{I}, \mathcal{R}, \Delta \rangle$ for a distributed system of $n$ processors if, given an input $I \in \mathcal{I}$, every execution of $A$ starting at input $I$ in which no more than $f$ processors fail will produce an output $R \in \mathcal{R}$, where $(I, R) \in \Delta$.

In the following, we let $P_1 = \langle \mathcal{I}_1, \mathcal{R}_1, \Delta_1 \rangle$ be an $f_1$-fault-tolerant decision problem for a system of $n_1$ processors and let $P_2 = \langle \mathcal{I}_2, \mathcal{R}_2, \Delta_2 \rangle$ be an $f_2$-fault-tolerant decision problem for a system of $n_2$ processors. Let $A_2$ be an algorithm (a set of $n_2$ programs) that solves $P_2$. A *simulation of $A_2$ by a set of $n_1$ processors* is an algorithm for a system of $n_1$ processors that simulates an algorithm $A_2$ of $P_2$. Both $A_2$ and the simulation of $A_2$ are shared-memory algorithms, and we need to specify the systems in which they run. In particular, in the simulations described in [11], both $A_2$ and its simulation run in systems which contain only snapshot objects. We will also consider simulations where, while the algorithm $A_2$ may access both snapshot and set consensus objects, its simulation can access only snapshot objects.

**Definition 4.** We say that the *simulation $S$ of an algorithm for $P_2$ by $n_1$ processors with at most $f_1$ faults solves problem $P_1$* (Figure 1), if there is a sequence of functions $g_1, \ldots, g_{n_1}$, where, for all $i$, $g_i : \mathcal{I}_1[i] \rightarrow \mathcal{I}_2$, and a sequence of functions $h_1, \ldots, h_{n_1}$, where, for all $i$, $h_i : \mathcal{R}_2 \rightarrow \mathcal{R}_1[i]$, and the following conditions hold for each execution of $S$ with no more than $f_1$ failures. Let the input values be the vector $\langle x_1, \ldots, x_{n_1} \rangle \in \mathcal{I}_1$.

1. The execution terminates with some output vector $\langle y_1, \ldots, y_{n_1} \rangle \in \mathcal{R}_1$. This is the liveness condition.
2. The outputs $y_1, \ldots, y_{n_1}$ are consistent with a valid solution for problem $P_1$ given inputs $x_1, \ldots, x_{n_1}$, where at most $f_1$ entries $y_i$ are null. More formally, $(\langle x_1, \ldots, x_{n_1} \rangle, \langle y_1, \ldots, y_{n_1} \rangle) \in \Delta_1$.
3. There exist an $n_2$-length input vector $\hat{I} \in \mathcal{I}_2$ and $n_1$ $n_2$-length output vectors $\hat{R}_1, \ldots, \hat{R}_{n_1} \in \mathcal{R}_2$ such that
   (a) For all $i$, $\hat{R}_i$ is a valid output of problem $P_2$ on input $\hat{I}$, *i.e.*, $(\hat{I}, \hat{R}_i) \in \Delta_2$.
   (b) The set of vectors $\hat{R}_i$ are consistent, *i.e.*, for all $j$, if the $j$th entries in two different vectors are both non-null, then they must be equal.
   (c) The vector $\hat{I}$ is derived from $\langle x_1, \ldots, x_{n_1} \rangle$ and the functions $g_1, \ldots, g_{n_1}$. Specifically, for all $j$, the $j$th entry of $\hat{I}$ is equal to the $j$th entry of $g_i(x_i)$, for some $i$.
   (d) The vector $\langle y_1, \ldots, y_{n_1} \rangle$ is derived from the vectors $\hat{R}_1, \ldots, \hat{R}_{n_1}$ and the functions $h_1, \ldots, h_{n_1}$. Specifically, for all $i$, $h(\hat{R}_i) = y_i$.

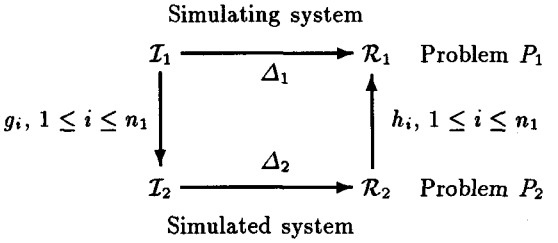


**Fig. 1.** A simulation of an algorithm which solves problem $P_2$ being used to solve problem $P_1$.

We assume that the simulated system $\mathcal{P}$ is an atomic snapshot memory system of $n_2$ programs, with a single snapshot variable $mem'$ consisting of $n_2$ components $mem'(j)$. We will refer to the processes whose executions are *being simulated* as *programs* and the *simulating* processes as *processors* to avoid confusion. As in [11], we let the simulating system $\mathcal{Q}$ be an atomic snapshot memory system of $n_1$ processors, with a single snapshot variable $mem$ consisting of $n_1$ components $mem(i)$. Each component $mem(i)$ is a copy of the entire memory $mem'$, and reflects processor $i$'s simulation of system $\mathcal{P}$. Along with a copy of $mem'(j)$, for each simulated process $j$, $mem(i)$ also includes a *sim-steps*$(j)$ counter which records the number of steps that $i$ has simulated for $j$ at the time of its last simulated update for $j$. This helps the other simulating processors identify the latest updated value, for any particular $j$, among all simulating processors $i$. A function *latest* can check, independently for each $j$, the *sim-steps*$(j)$ value in each $mem(i)$, and choose, for each $j$, the particular $mem(i).mem'(j)$ so that $mem(i).sim\text{-}steps(j)$ is highest over all $i$.

To maintain consistency among the real processors, it is important that each real processor agrees on the same snapshot value at each step so that they make the same state transitions. Simulating each snapshot therefore requires an agreement protocol, where different processors submit their individual versions of the snapshot variable $mem'$ and the agreement protocol chooses the version of a specific real processor, a value which is then adopted by all processors. In addition, the fact that each real processor submits snapshot values computed by the *latest* function makes sure that the snapshot values decided upon are consistent with a real run of the simulated processes.

We now consider the simulations of algorithms which access set consensus objects. Here, the simulation is further complicated by the fact that, while the simulated system has set consensus objects, there are no such objects in the simulating system. The simulation therefore also has to simulate the set consensus objects. Suppose the simulated algorithm has an $(m, \ell)$-*setcon* object $\mathcal{O}$ which is accessed by $m$ simulated programs $1, \ldots, m$, each with an input value. We need the *setcon* object $\mathcal{O}$ to return at most $\ell$ distinct values. We simulate the $m$ accesses to object $\mathcal{O}$ as follows. We let the $n_1$ real processors, each simulating the *setcon* operation of all of the $m$ programs to object $\mathcal{O}$ (this involves $n_1 m$ total

such simulated accesses), all participate in the same version of an $\ell$-agreement protocol, returning at most $\ell$ distinct values. We still need to ensure that the $n_1$ simulations of the *setcon* operation for *each particular program* $j \in 1, \ldots, m$, obtain the same value. We therefore have the $n_1$ real processors participate in a 1-agreement protocol, one for each program $j$. Therefore each *setcon* operation is simulated by an $\ell$-agreement followed by a 1-agreement. The simulation therefore requires the implementation of both 1-agreement and $\ell$-agreement protocols which are discussed in Section 5.

We now consider the fault-tolerance of the simulation. If the simulated algorithm is $f_2$-resilient, it is important that the simulations of no more than $f_2$ programs be blocked. As we will show in Section 5, our agreement (respectively, $\ell$-agreement) protocol is 0-resilient (respectively, $(\ell - 1)$-resilient). Therefore, a real processor $i$, simulating a snapshot of a program $j$, could fail while participating in the agreement algorithm and block the program $j$ from being simulated any further by other processors. Similarly, if $\ell$ real processors simulating the access to the $(m, \ell)$-*setcon* object $\mathcal{O}$ by the programs $1, \ldots, \ell$, respectively, fail in the middle of the $\ell$-agreement algorithm, they could potentially block all $m$ programs that can access $\mathcal{O}$. In general, a combination of these can happen, causing the simulation of certain programs to be blocked. To minimize the amount of damage caused by a faulty real processor and still allow progress to be made, we require that each real processor be in a position to block at most one agreement or $\ell$-agreement algorithm at any time. We will show how to achieve this in Section 5. We can now argue about the maximum number $f_1$ of faulty simulating processors that can be tolerated to successfully simulate the $f_2$-resilient algorithm. The Simulation Theorem, which will be proved formally in Section 6, is stated below.

**Theorem 5. [Simulation Theorem]** *A set of $n_2$ programs $j_1, \ldots, j_{n_2}$ solving the $f_2$-fault tolerant problem $P_2$, which accesses snapshot objects and $(m, \ell)$-setcon objects, can be simulated by a set of $n_1$ processors $i_1, \ldots, i_{n_1}$ solving the $f_1$-fault tolerant problem $P_1$, which only accesses snapshot objects, if*

1. *the sets of functions $g$ and $h$ exist as required by the simulation, and*
2. *$f_1 \geq a\ell + b$ implies $f_2 \geq am + b$, for all positive integers $a$ and $b$.*

We can now state a corollary to the Simulation Theorem, which applies to simulated systems with no *setcon* objects. It is obtained by setting $m = \ell$ in the Simulation Theorem, since $(m, m)$-*setcon* objects are trivial.

**Corollary 6.** *A set of $n_2$ programs solving the $f_2$-fault tolerant problem $P_2$, which accesses snapshot objects, can be simulated by a set of $n_1$ processors solving the $f_1$-fault tolerant problem $P_1$, which also accesses snapshot objects, if the set of functions $g$ and $h$ exist as required by the simulation and if $f_1 \leq f_2$.*

The following theorem, restated in terms of *setcon* objects, was originally proved by three teams of researchers [2, 9, 15]. It will be useful in proving the Partial Order Theorem.

**Theorem 7. [Set Consensus Impossibility Theorem]** *The $(k+1,k)$-setcon object does not have a wait-free implementation in an asynchronous snapshot shared memory system with $k+1$ processors.*

The impossibility of $k$-resilient solutions for the set consensus problem in the asynchronous shared memory model with $n$ processors, originally proven by Borowsky and Gafni, now follows from Theorem 7 and Corollary 6.

**Theorem 8.** *There is no solution to the $k$-set consensus problem in an asynchronous shared memory model with $n$ processors (where $k < n$), which is resilient to $k$ fail-stop faults.*

## 4 The Set Consensus Partial Order

By considering wait-free $(m,\ell)$-*setcon* objects, the consensus hierarchy can be refined into partial orders. The Partial Order Theorem, proved below, gives the relationships between different $(m,\ell)$-*setcon* objects for all values of $m$ and $\ell$. It is proven using the Set Consensus Impossibility Theorem and the Simulation Theorem.

*Proof. (of Partial Order Theorem)* Cases 1 and 2 are straightforward. We prove the other cases below. Cases 3a and 3b are based on the protocols in [3]. Case 3(c) is our main impossibility result.
*Case 3(a)*: We implement a wait-free $(n,k)$-*setcon* object using only wait-free $(m,\ell)$-*setcon* objects and snapshot objects. Divide the set of $n$ processors into $\lceil n/m \rceil$ groups of at most $m$ processors each. Let each group invoke a different $(m,\ell)$-*setcon* object and decide on the values returned. The number of different decisions made by the processors in any given group is at most $\ell$. This implies that the total number of different values decided by all the processors is at most $\ell\lceil n/m \rceil$. Since, by assumption, $k \geq \ell\lceil n/m \rceil$, at most $k$ values are decided, as required.
*Case 3(b)*: We implement a wait-free $(n,k)$-*setcon* object using just wait-free $(m,\ell)$-*setcon* objects and snapshot objects. Divide the set of $n$ processors into $\lfloor n/m \rfloor$ groups of $m$ processors each. There will be $n - m\lfloor n/m \rfloor$ processors remaining which are not part of any group. Let each group of $m$ processors invoke a different $(m,\ell)$-*setcon* object and decide on the values returned. The number of different decisions made by the $m$ processors in each of these groups is at most $\ell$. Let each of the remaining $n - m\lfloor n/m \rfloor$ processors decide on its own value. The total number of values decided by all the processors is therefore at most $\ell\lfloor n/m \rfloor + (n - m\lfloor n/m \rfloor)$. Since, by assumption, $k \geq \ell\lfloor n/m \rfloor + (n - m\lfloor n/m \rfloor)$, at most $k$ values are decided, as required.
*Case 3(c)*: Suppose $k < \ell\lceil n/m \rceil$ and $k < \ell\lfloor n/m \rfloor + (n - m\lfloor n/m \rfloor)$. Hence, $k+1 \leq \ell\lceil n/m \rceil$ and $k+1 \leq \ell\lfloor n/m \rfloor + (n - m\lfloor n/m \rfloor)$.

Suppose a wait-free $(n,k)$-*setcon* object can be implemented using wait-free $(m,\ell)$-*setcon* objects and snapshot objects. Then, we will show, using the Simulation Theorem, that $(k+1,k)$-*setcon* can be implemented by snapshot objects, contradicting the Set Consensus Impossibility Theorem.

By the assumption above, there is an $n$ program wait-free algorithm which implements an $(n,k)$-*setcon* object using wait-free $(m,\ell)$-*setcon* objects and snapshot objects. We use the Borowsky-Gafni simulation technique to simulate these $n_2 = n$ programs of code using $n_1 = k+1$ simulating processors $i_1, \ldots, i_{k+1}$, with input values $x_1, \ldots, x_{k+1}$. The simulating processors only use snapshot objects while the programs of code use $(m,\ell)$-*setcon* objects and snapshot objects. Since problem $A$ of the simulated system and problem $B$ solved by the simulating processors are both the $k$-set consensus problem with different numbers of inputs, our set of functions $g$ and $h$ are trivial. Let $g_i(v)$ be the $n_2$-length vector $(v, \ldots, v)$. The function $h_i$ maps an output vector of the simulated program to *any* non-null value in the vector. Since the set of $n$ programs $j_1, \ldots, j_n$ is wait-free, it is resilient to $n-1$ faults, and, since we would like our simulation to be wait-free, we may have as many as $k$ faulty simulating processors.

Now, by the Simulation Theorem, it follows that the simulation will succeed as long as, for all non-negative integer values of $a$ and $b$, if $k \geq a\ell + b$ then $n - 1 \geq am + b$. Let $a$ and $b$ be non-negative integers such that $k \geq a\ell + b$. We will show $n - 1 \geq am + b$, thus proving that every non-faulty simulating processor will terminate.

By assumption, $a\ell + b < k + 1$. Since $k + 1 \leq \ell\lceil n/m \rceil$, it follows that $a < \lceil n/m \rceil$, implying that $a \leq \lfloor n/m \rfloor$. Also, since $k + 1 \leq \ell\lfloor n/m \rfloor + (n - m\lfloor n/m \rfloor)$, it follows that $a\ell + b < \ell\lfloor n/m \rfloor + (n - m\lfloor n/m \rfloor)$.
Now,

$$\begin{aligned} & a\ell + b < n - m\lfloor n/m \rfloor + \ell\lfloor n/m \rfloor \\ & \qquad = n - (m - \ell)\lfloor n/m \rfloor \\ \Longrightarrow \quad & b < n - (m - \ell)\lfloor n/m \rfloor - a\ell \\ \Longrightarrow \quad & am + b < n - (m - \ell)\lfloor n/m \rfloor - a\ell + am \\ & \qquad = n - (m - \ell)\lfloor n/m \rfloor + a(m - \ell) \\ & \qquad = n - (m - \ell)(\lfloor n/m \rfloor - a) \end{aligned}$$

Since $m > \ell$ and $a \leq \lfloor n/m \rfloor$, it follows that $(m-\ell)(\lfloor n/m \rfloor - a) \geq 0$, thus proving that $am + b < n$, as required.

So, the conditions of the Simulation Theorem hold. By Definition 4, each non-faulty real processor $i \in \{i_1, \ldots, i_{k+1}\}$ will terminate with an output value $y_i$, where $y_i$ is the output of some program $j \in \{j_1, \ldots, j_n\}$ in an execution where the input of each program $j$ is in the set $\{x_1, \ldots, x_{k+1}\}$. Since the set of programs implementing the $(n,k)$-*setcon* object, satisfies *validity*, the output of each program $j$ is an input of some program $j'$ in the execution, and $y_i \in \{x_1, \ldots, x_{k+1}\}$, where $y_i$ is the output of real processor $i$. Also, since the set of programs $j$ satisfies *k-agreement*, the number of distinct outputs of the programs is no more than $k$, and, thus, the number of distinct values $y_i$ is also no more than $k$, where $y$ is the output of real processor $i$. Thus, the simulation represents a valid implementation of a $(k+1,k)$-*setcon* object using only snapshot objects. This contradicts the Set Consensus Impossibility Theorem. Therefore, an $(n,k)$-*setcon* object cannot be implemented by any number of $(m,\ell)$-*setcon* objects and snapshot objects. □

The theorem takes care of all possible cases, and completely characterizes the relationship between different *setcon* objects. Note that this does *not* define a total order, since there are pairs of objects, where neither implements the other. $(2,1)$-*setcon* and $(5,2)$-*setcon* is an example of such a pair.

## 5 The Agreement Protocol

We now define the agreement protocol as an I/O automaton, very much the same way as done in [11]. Our contribution here is in extending their definitions to include $\ell$-agreement protocols. Each agreement module has $N$ ports, numbered $1, \ldots, N$. The agreement protocol may fail due to failures at the ports (the users of the agreement module); however, this can only happen during an 'unsafe' portion of the execution of a particular port. Each port $i$ supports input actions of the form $propose_i(v)$ and $stop_i$, and output actions of the form $safe_i$ and $agree_i(w)$. A value $v$ is proposed for agreement by user $i$ with the input action $propose_i(v)$, and the output action $safe_i$ tells user $i$ that the unsafe portion of its execution is over. Finally, the agreed upon value $w$ is returned by the output action $agree_i(w)$. If user $i$ fails before receiving $agree_i(w)$, the input action $stop_i$ announces this failure.

Now, clearly a *well-formed* execution would require that any $propose_i$, $safe_i$ and $agree_i$ actions for a given agreement module be in that order. Also, as in [11], we require that the programs preserve well-formedness on every port, that is, there is at most one $propose_i$ for any particular $i$. In addition, each module must satisfy the VALIDITY condition, which says that any agreement value must be a proposed value.

We will be looking at both 1-agreement and $\ell$-agreement modules, where $\ell > 1$. They need to satisfy the AGREEMENT and $\ell$-AGREEMENT conditions, respectively. The $\ell$-agreement condition requires that there are at most $\ell$ distinct agreement values. AGREEMENT is defined as 1-agreement.

We have two liveness conditions for each of our agreement modules. The first liveness condition, WAIT-FREEDOM, says that a $propose_i$ action on a non-faulty port will eventually receive a $safe_i$. In other words, every non-faulty port is guaranteed to complete the unsafe portion of its execution, no matter what happens at the other ports. More formally, WAIT-FREEDOM requires that in any execution, for any $i$, if $propose_i$ occurs and no $stop_i$ occurs, then $safe_i$ occurs.

The second liveness condition, SAFE TERMINATION, deals with the fault-tolerant behavior and depends on the specific agreement module. In the case of 1-agreement, SAFE TERMINATION says that, if no ports remain unsafe, then any *propose* event on a non-faulty port will receive an *agree*. In the case of $\ell$-agreement, the condition is stronger; $\ell$-SAFE TERMINATION says that, if *no more than* $\ell - 1$ ports remain unsafe, then any *propose* event on a non-faulty port will receive an *agree*. More formally, $\ell$-SAFE TERMINATION requires that in any execution, if there are no more than $\ell - 1$ indices $j$ such that $propose_j$ occurs and $safe_j$ does not occur, then, for any $i$, if $propose_i$ occurs and $stop_i$ does not occur, then $agree_i$ occurs. SAFE TERMINATION is defined as 1-safe termination.

These conditions on the agreement protocol allow the real processors to make sure that they are in a position to block at most one agreement protocol at any time, and still make progress simulating programs. In particular, a real processor can be involved in several 1-agreement (or $\ell$-agreement) protocols at the same time, as long as it is not in the unsafe portion of more than one (that is, there is at most one module for which it has made the *propose* request but has not received the *safe* announcement). In that case, if a real processor fails, then only one agreement module would be blocked. This is ensured by having a *status* variable, for each processor $i$, which has the value *unsafe* when $i$ has a $propose_i$ pending at some agreement module and which is changed to *safe* when $i$ receives the *safe* announcement. A value *unsafe* in the *status* variable disallows $i$ from sending another $propose_i$ action to some other agreement module.

The agreement module in [11] works as follows. The snapshot shared memory contains a *val* component and a *level* component, which is initialized t 0, for each port $i$. When a $propose_i(v)$ is received, the value $v$ is recorded in *val* and *level* is raised to 1. Then a snapshot is taken to determine whether $level = 2$ for any other port $i'$. If so, $level_i$ is set to 0, and otherwise it is set to 2.

Now, repeated snapshots are taken until there is no $i'$ such that $level_{i'} = 1$, in the case of the 1-agreement module, and until there are no more than $\ell - 1$ ports $i'$ such that $level_{i'} = 1$ in the case of the $\ell$-agreement module. Now, the *val*, $w$ corresponding to port $i_0$, where $i_0$ is the smallest index such that $level_{i_0} = 2$, is chosen and returned by the output action $agree_i(w)$.

The $\ell$-agreement module is a general version of the 1-agreement module of [11]. We do not describe it in detail here due to lack of space. The only difference between the two modules is in the condition required for exiting the repeated snapshot loop described above. In addition, we define the actions $\ell$-$propose_i$, $\ell$-$safe_i$ and $\ell$-$agree_i$ in the $\ell$-agreement module, to distinguish them from the equivalent actions $propose_i$, $safe_i$ and $agree_i$, respectively, in the 1-agreement module.

**Lemma 9.** *$\ell$-SafeAgreement satisfies the $\ell$-agreement condition.*

**Lemma 10.** *The $\ell$-agreement module defined above satisfies the safety conditions of* VALIDITY *and* $\ell$-AGREEMENT *and the liveness conditions of* WAIT-FREEDOM *and* $\ell$-SAFE TERMINATION.

## 6 Proof of the Simulation

We now give the details of our simulation. Given the system $\mathcal{P}$ of $n_2$ programs with access to the snapshot variable $mem'$ and a set $O$ of $(m, \ell)$-*setcon* objects, we want to simulate it in the system $\mathcal{Q}$ of $n_1$ processors with access only to the snapshot variable *mem*. In Figure 2, we give the automaton for $\mathcal{Q}$, an extension of the same in [11].

We define $\mathcal{P}$ and $\mathcal{Q}$ as I/O automata, and show that system $\mathcal{Q}$ simulates system $\mathcal{P}$. We do this in two stages, by first defining a new system $\mathcal{C}$. We then show that $\mathcal{C}$ simulates $\mathcal{P}$, and $\mathcal{Q}$ simulates $\mathcal{C}$, thus obtaining our result. Since

**Simulation System $Q$:**

**Shared variables:**

$mem$, a length $n_1$ snapshot value;
  for each $i$, $mem(i)$ has components:
    $sim\text{-}mem$, a vector in $R^{n2}$, initially everywhere $r_0$
    $sim\text{-}steps$, a vector in $N^{n2}$, initially everywhere 0

**Actions of $i$:**

**Input:**
  $init(v)_i$, $v \in V$
  $\ell\text{-}agree(w)_{j,\mathcal{O},i}$, $\mathcal{O} \in \mathbf{O}$ and $w \in W_{\mathcal{O}}$
  $\ell\text{-}safe_{j,\mathcal{O},i}$, $\mathcal{O} \in \mathbf{O}$
  $agree(v)_{j,k,i}$,
    $k = 0$ and $v \in V$, or $k \in N^+$ and $v \in R^{n2}$
  $agree(w)_{j,\mathcal{O},i}$, $\mathcal{O} \in \mathbf{O}$ and $w \in W_{\mathcal{O}}$
  $safe_{j,k,i}$, $k \in N$
**Output:**
  $decide(v)_i$, $v \in V$
  $\ell\text{-}propose(w)_{j,\mathcal{O},i}$, $\mathcal{O} \in \mathbf{O}$ and $w \in W_{\mathcal{O}}$
  $propose(v)_{j,k,i}$,
    $k = 0$ and $v \in V$, or $k \in N^+$ and $v \in R^{n2}$
  $propose(w)_{j,\mathcal{O},i}$, $\mathcal{O} \in \mathbf{O}$ and $w \in W_{\mathcal{O}}$
**Internal:**
  $sim\text{-}update_{j,i}$
  $snap_{j,i}$
  $sim\text{-}local_{j,i}$
  $sim\text{-}decide_{j,i}$

**States of $i$:**

$input \in V \cup \{null\}$, initially $null$
$reported$, a Boolean variable, initially $false$
for each $j$:
  $sim\text{-}state(j)$, a state of $j$, initially the initial state
  $sim\text{-}steps(j) \in N$, initially 0
  $sim\text{-}snaps(j) \in N$, initially 0
  $status(j) \in \{idle, propose, unsafe, safe\}$,
    initially $idle$
  $sim\text{-}mem\text{-}local \in R^{n2}$, initially arbitrary
  $sim\text{-}decision(j) \in V \cup \{null\}$, initially $null$
  $object\text{-}val_{\mathcal{O}}(j) \in W_{\mathcal{O}} \cup \{null\}$, initially $null$, $\mathcal{O} \in \mathbf{O}$

**Transitions of $i$:**

$init(v)_i$
  Effect: $input(i) := v$

$propose(v)_{j,0,i}$
  Precondition: $status(j) = idle$
    $\neg\exists k : status(k) = unsafe$
    $nextop(sim\text{-}state(j)) = \text{"init"}$
    $input(i) \neq null$
    $v = g_i(input(i))(j)$
  Effect: $status(j) := unsafe$

$safe_{j,k,i}$
  Effect: $status(j) := safe$

$agree(v)_{j,0,i}$
  Effect: $sim\text{-}state(j)$
    $:= trans\text{-}init(sim\text{-}state(j), v)$
    $sim\text{-}steps(j) := 1$
    $status(j) := idle$

$snap_{j,i}$
  Precondition: $nextop(sim\text{-}state(j)) = \text{"snap"}$
    $status(j) = idle$
  Effect: $sim\text{-}mem\text{-}local(j) := latest(mem)$
    $status(j) := propose$

$propose(w)_{j,k,i}$, $k \in N^+$
  Precondition: $status(j) = propose$
    $\neg\exists m : status(m) = unsafe$
    $sim\text{-}snaps(j) = k - 1$
    $w = sim\text{-}mem\text{-}local(j)$
  Effect: $status(j) := unsafe$

$agree(w)_{j,k,i}$, $k \in N^+$
  Effect: $sim\text{-}state(j)$
    $:= trans\text{-}snap(sim\text{-}state(j), w)$
    $sim\text{-}steps(j)$
    $:= sim\text{-}steps(j) + 1$
    $sim\text{-}snaps(j) := sim\text{-}snaps(j) + 1$
    $status(j) := idle$

$\ell\text{-}propose(w)_{j,\mathcal{O},i}$, $\mathcal{O} \in \mathbf{O}$
  Precondition: $nextop(sim\text{-}state(j))$
    $= (\text{"set-consensus}_{\mathcal{O}}\text{"}, w)$
    $status(j) = idle$
    $\neg\exists k : status(k) = unsafe$
  Effect: $status(j) := unsafe$

$\ell\text{-}safe_{j,\mathcal{O},i}$, $\mathcal{O} \in \mathbf{O}$
  Effect: $status(j) := safe$

$\ell\text{-}agree(w)_{j,\mathcal{O},i}$, $\mathcal{O} \in \mathbf{O}$
  Effect: $object\text{-}val_{\mathcal{O}}(j) := w$
    $status(j) := propose$

$propose(w)_{j,\mathcal{O},i}$, $\mathcal{O} \in \mathbf{O}$
  Precondition: $status(j) = propose$
    $\neg\exists k : status(k) = unsafe$
    $nextop(sim\text{-}state(j)) = (\text{"set-consensus}_{\mathcal{O}}\text{"}, v)$
    $object\text{-}val_{\mathcal{O}}(j) = w$
  Effect: $status(j) := unsafe$

$agree(w)_{j,\mathcal{O},i}$, $\mathcal{O} \in \mathbf{O}$
  Effect: $sim\text{-}state(j) := trans\text{-}sc_{\mathcal{O}}(sim\text{-}state(j), w)$
    $sim\text{-}steps(j) := sim\text{-}steps(j) + 1$
    $status(j) := idle$

$sim\text{-}update_{j,i}$
  Precondition: $nextop(sim\text{-}state(j)) = (\text{"update"}, r)$
  Effect: $sim\text{-}state(j) := trans(sim\text{-}state(j))$
    $sim\text{-}steps(j) := sim\text{-}steps(j) + 1$
    $mem(i).sim\text{-}mem(j) := r$
    $mem(i).sim\text{-}steps(j) := sim\text{-}steps(j)$

$sim\text{-}local_{j,i}$
  Precondition: $nextop(sim\text{-}state(j)) = \text{"local"}$
  Effect: $sim\text{-}state(j) := trans(sim\text{-}state(j))$
    $sim\text{-}steps(j) := sim\text{-}steps(j) + 1$

$sim\text{-}decide_{j,i}$
  Precondition: $nextop(sim\text{-}state(j)) = (\text{"decide"}, v)$
  Effect: $sim\text{-}state(j) := trans(sim\text{-}state(j))$
    $sim\text{-}steps(j) := sim\text{-}steps(j) + 1$
    $sim\text{-}decision(j) := v$

$decide(v)_i$
  Precondition: $reported = false$
    $|sim\text{-}decision| \geq n_2 - f$
    $v = h_i(sim\text{-}decision)$
  Effect: $reported := true$

**Tasks of $i$:**

$\{decide(v)_i \mid v \in V\}$
for each $j$:
  all non-input actions involving $j$

**Fig. 2.** Automaton for $Q$.

our system $\mathcal{P}$ is basically the simulated system described by Lynch and Rajsbaum [11] with the addition of $(m, \ell)$-*setcon* objects, we will give an overview of their proof methodology and then focus on our extensions.

The operations of the algorithm described in the system $\mathcal{P}$ include *init*$(v)$, *snap*, *update*$(r)$, *setcons*$(\mathcal{O}, v)$ and *decide*$(v)$, for each program $j$. Let *trans-init*$(v)$ be the initial state of program $j$ given the input value $v$ (this gives the initial state with which to start the simulation of program $j$). Now, given any state $s$ of program $j$, let *nextop*$(s)$ be the next operation in the program. If *nextop*$(s)$ = *snap*, let *trans-snap*$(s, w)$ be the state resulting from a snapshot operation at state $s$ that returns $w$. Similarly, if *nextop*$(s)$ = *setcons*$(\mathcal{O}, v)$, let *trans-sc*$(s, w, \mathcal{O})$ be the state that results from a call to the *setcon* object $\mathcal{O}$ that returns the value $w$. Finally, if *nextop*$(s)$ is either *update* or *decide*, *trans*$(s)$ is the state resulting from the corresponding operation.

We now define the system $\mathcal{C}$, using I/O automata, which simulates $\mathcal{P}$ in a centralized manner. This corresponds to *SimpleSpec* in [11], except for the simulation of *setcon* operations. Here, a single processor selects a program $j$ in $\mathcal{P}$ nondeterministically and simulates its next operation. It chooses the input value for each program $j$ by picking *any* simulating processor $i$ and adopting its choice of the input value, *i.e.*, the $j$th component of the vector $g_i(input(i))$. When at least $n_2 - f_2$ programs have terminated with decision values, the central processor terminates with $n_1$ output values determined using the functions $h_i$.

This simulation is relatively straightforward except that each $(m, \ell)$-*setcon* object ($m$ and $\ell$ are fixed) in system $\mathcal{P}$ has to be directly simulated in system $\mathcal{C}$ (since the simulating system does not have access to *setcon* objects). Specifically, we have two new internal actions, *inv-setcons*$(v, \mathcal{O})$ and *ret-setcons*$(w, \mathcal{O})$, which simulate the proposal of a value $v$ to an $\ell$-set-consensus object $\mathcal{O}$ and the return of a value $w$ by the same object $\mathcal{O}$, respectively. We have defined these actions in the most general way possible, so that they can correspond to any possible correct implementation of set-consensus objects. That is, we make no assumptions about which of the proposed values can be returned by an object (such as, say, letting the set of returned values be the first $\ell$ values proposed, or the first value proposed), other than that no more than $\ell$ different values are returned.

We now describe how the automaton $\mathcal{C}$ works. $\mathcal{P}$ has invocations (by programs $j$) to a collection of $(m, l)$-set-consensus objects, indexed by $\mathcal{O}$. The value set of each set-consensus object is $W$, and the problem it solves is $\ell$-set consensus. $\mathcal{C}$ simulates $\mathcal{P}$ in a centralized manner, simulating all snapshots and updates as in [11]. In addition, $\mathcal{C}$ simulates each $setcons_{\mathcal{O}}(v)$ operation by two internal actions, *inv-setcons* and *ret-setcons*, which correspond to the invocation and response of the operation, respectively.
For each object $\mathcal{O}$, the state includes

$$inv\text{-}vals(\mathcal{O}), ret\text{-}vals(\mathcal{O}) \subseteq W, \text{ initially } \emptyset,$$

and, for each object $\mathcal{O}$ and program $j$,

$$inv\text{-}setcons(j, \mathcal{O}) \in \{yes, no\}, \text{ initially } no.$$

As specified earlier, $i \in \{1, \ldots, n_1\}$ is an index of a simulating processor while $j \in \{1, \ldots, n_2\}$ is an index of a simulated program.

**Simulation System $\mathcal{C}$**
**Transitions:**

$init(v)_i$
  Effect: $input(i) := v$

$sim\text{-}init_j$
  Precondition: $nextop(sim\text{-}state(j)) =$ "init"
    for some $i$
      $input(i) \neq null$
      $v = g_i(input(i))(j)$
  Effect: $sim\text{-}state(j) := trans\text{-}init(sim\text{-}state(j), v)$

$sim\text{-}snap_j$
  Precondition: $nextop(sim\text{-}state(j)) =$ "snap"
  Effect: $sim\text{-}state(j) := trans\text{-}snap(sim\text{-}state(j), sim\text{-}mem)$

$sim\text{-}update_j$
  Precondition: $nextop(sim\text{-}state(j)) = (\text{"update"}, r)$
  Effect: $sim\text{-}state(j) := trans(sim\text{-}state(j))$
    $sim\text{-}mem(j) := r$

$sim\text{-}local_j$
  Precondition: $nextop(sim\text{-}state(j)) =$ "local"
  Effect: $sim\text{-}state(j) := trans(sim\text{-}state(j))$

$sim\text{-}decide_j$
  Precondition: $nextop(sim\text{-}state(j)) = (\text{"decide"}, v)$
  Effect: $sim\text{-}state(j) := trans(sim\text{-}state(j))$
    $sim\text{-}decision(j) := v$

$inv\text{-}setcons_j(v, \mathcal{O})$
  Precondition: $nextop(sim\text{-}state(j)) =$ "$setcons_{\mathcal{O}}(v)$"
  Effect: $inv\text{-}vals(\mathcal{O}) := inv\text{-}vals(\mathcal{O}) \cup \{v\}$
    $inv\text{-}setcons(j, \mathcal{O}) := yes$

$ret\text{-}setcons_j(w, \mathcal{O})$
  Precondition: $nextop(sim\text{-}state(j)) =$ "$setcons_{\mathcal{O}}(v)$"
    $inv\text{-}setcons(j, \mathcal{O}) = yes$
    $w \in inv\text{-}vals_{\mathcal{O}}$
    $|ret\text{-}vals(\mathcal{O}) \cup \{w\}| \leq \ell$
  Effect: $ret\text{-}vals(\mathcal{O}) := ret\text{-}vals(\mathcal{O}) \cup \{w\}$
    $sim\text{-}state(j) := trans\text{-}sc_{\mathcal{O}}(sim\text{-}state(j), w)$
$decide(v)_i$
  Precondition: $reported(i) = false$
    $w$ is a 'sub-vector' of $sim\text{-}decision$
    $|w| \geq n_2 - f_2$
    $v = h_i(w)$
  Effect: $reported(i) := true$

It is easy to see now that system $\mathcal{C}$ simulates system $\mathcal{P}$. Clearly, the *inv-setcons* and *ret-setcons* actions do indeed solve $\ell$-set consensus, and all other actions are simulated properly. This result is stated, without proof, in the lemma below.

**Lemma 11.** *There is a relation between states in system $\mathcal{C}$ to states in system $\mathcal{P}$ such that any fair execution of system $\mathcal{C}$ corresponds to a fair execution of system $\mathcal{P}$.*

We now show that $\mathcal{C}$ can be simulated by $\mathcal{Q}$, along with the agreement modules. By the results of Lynch and Rajsbaum, $\mathcal{Q}$ will simulate $\mathcal{C}$, as long as we ignore the *setcon* operations. In particular, a snapshot operation of program $j$ is simulated by the pairs of actions $propose_j$ and $agree_j$, one for each real processor $i$, where the values proposed are the individual snapshots of each processor $i$. We now give an informal description of the simulation of the *setcon* operation. We omit the formal proof for lack of space. As mentioned in Section 3, each $inv\text{-}setcons_j(\mathcal{O}, v)$ and $ret\text{-}setcons_j(\mathcal{O}, w)$ pair is simulated by the sequences $\ell\text{-}propose_{j,\mathcal{O},i}(v)$, $\ell\text{-}agree_{j,\mathcal{O},i}(x_i)$, $propose_{j,\mathcal{O},i}(x_i)$, $agree_{j,\mathcal{O},i}(w)$, one for each simulating processor $i$. Recall that the $\ell$-agreement actions are used to limit the total number of different responses to $\ell$, while the agreement actions are used to guarantee consistency among the simulations. Specifically, for any particular program $j$, only the first $\ell\text{-}propose(v)$ action in $\mathcal{Q}$, among all processors $i$, is mapped to the $inv\text{-}setcons(v)$ action in $\mathcal{C}$. Similarly, for any particular program $j$, only the first $agree(w)$ action, among all processors $i$, in $\mathcal{Q}$ is mapped to the $ret\text{-}setcons(w)$ action in $\mathcal{C}$. For readers familiar with the simulation in [11], unlike in the simulation of the snapshot operation, where it is important to identify the winning proposed simulation of the snapshot since each simulation is different, here each simulation of the $inv\text{-}setcons(v)$ action *of a particular program* $j$ will $\ell\text{-}propose$ the same value $v$ *and agree* on the same value $w$. So, the complication of a backward simulation followed by a forward simulation can be avoided here.

We can now prove the safety and liveness conditions of the simulation of $\mathcal{C}$ by $\mathcal{Q}$, leading to the Simulation Theorem. The following safety conditions follow from Lemma 10.

**Lemma 12.** *For any setcon object $\mathcal{O}$ accessed in an execution of $\mathcal{P}$, let $V_{\mathcal{O}}$ be the set of values $v$ of all actions of the form $\ell\text{-}propose_{j,\mathcal{O},i}(v)$ in the execution, and let $W_{\mathcal{O}}$ be the set of values $w$ of all actions of the form $\ell\text{-}agree_{j,\mathcal{O},i}(w)$ in the execution. Then, $W_{\mathcal{O}} \subseteq V_{\mathcal{O}}$ and $|W_{\mathcal{O}}| \leq \ell$. Also, for any particular program $j$ and object $\mathcal{O}$, if $\ell\text{-}agree_{j,\mathcal{O},i_1}(w_1)$ and $\ell\text{-}agree_{j,\mathcal{O},i_2}(w_2)$ are actions by two real processors $i_1$ and $i_2$, then $w_1 = w_2$.*

We also have the following liveness condition.

**Lemma 13.** *Given a set of processors, $i_1, i_2, \ldots, i_{n_1}$, doing a B-G simulation of a set of programs, $j_1, j_2, \ldots, j_{n_2}$ in the system $\mathcal{Q}$, for each processor $i$, at any time within its simulation, there is at most one program $j$ such that $i$'s simulation of $j$ is within the 'unsafe' portion of a safe agreement module or $\ell$-safe agreement module.*

We now sketch the proof of the Simulation Theorem. For the complete proof refer to the full paper [14].

*Proof. (of Simulation Theorem)* Lemma 12 guarantees that the $(m, \ell)$-*setcon* objects are simulated correctly. It remains to be shown that the simulation terminates.

Suppose we have the required functions $g$ and $h$. In a B-G simulation, a processor halts after successfully completing its simulation of any program. Hence, the only reason the simulation could fail is if a non-faulty processor $i$ is unable to terminate the simulation of any of the programs $j_1, j_2, \ldots, j_{n_2}$, even though the relationship between the number of faulty processors, $f_1$ and the resiliency of the set of programs, $f_2$, is as required. It follows that either there is some program $j'$ such that, after some time $t_0$, $i$ does not simulate any steps of $j'$, or, for all $j$, $i$ simulates steps of $j$ infinitely often.

Suppose the first case holds. Since the fairness condition requires $i$ to simulate each program $j$ which has a step enabled, it must be true that $j'$ is in the 'unsafe' portion of an agreement algorithm at all times after $t_0$. This cannot happen, since, by the wait-free condition satisfied by the agreement module, $i$ will eventually execute $safe_i$ (and set *status* to *safe*), no matter how many other processors fail.

Suppose the second case holds. Then $i$ is blocked within its simulation of $j$, for each $j$, after some time $t_0$. Now, within $i$'s simulation of each $j$, $i$ is either blocked within a safe agreement or $\ell$-safe agreement module simulating a particular snapshot or set-consensus object invocation, or it is able to terminate each simulation of an individual snapshot or set-consensus object invocation statement in which case it is blocked within an infinite loop of the program $j$, itself. Let $L_1$ be the set of $\ell_1$ programs in which $i$ is blocked within a safe agreement or $\ell$-safe agreement module, and let $L_2$ be the set of the remaining $\ell_2 = n_2 - \ell_1$ programs. Since the set of programs is $f_2$-resilient, if at least $n_2 - f_2$ programs are allowed to each take sufficiently many steps, they will all terminate. Therefore, it follows that $\ell_2 < n_2 - f_2$, implying that $\ell_1 > f_2$. Now, $i$ must be within the busy-wait section, that is, the $wait_i$ or the $\ell$-$wait_i$ section, of a safe agreement or an $\ell$-safe agreement module simulating a snapshot or a set-consensus operation for each simulation of programs $j$, for $\ell_1$ programs $j$. Let $b_0$ be the number of snapshot operations being blocked, and let $a_0$ be the number of distinct set-consensus objects, $\mathcal{O}$, whose operations are being blocked. Since each set-consensus object can be accessed by at most $m$ programs, each blocked set-consensus operation can block at most $m$ programs. Therefore, it follows that $\ell_1 \leq a_0 m + b_0$. Now, the safe agreement modules satisfies the safe termination property, as proved in [11]. Hence, there is some processor which failed within the straight line section (the actions between and including $propose_i$ and $safe_i$) of the safe agreement module used in simulating a snapshot in $b_0$ distinct programs. Also, by Lemma 10, there must be at least $\ell$ processors which have failed within the straight line section (the actions between and including $\ell$-$propose_i$ and $\ell$-$safe_i$) of each of the $a_0$ $\ell$-safe agreement modules used in simulating the operations on $a_0$ distinct set-consensus objects. By Lemma 13, a processor cannot execute the actions between and including $\ell$-$propose_i$ and $\ell$-$safe_i$ or between and including $propose_i$ and $safe_i$ of more than one program simultaneously. Therefore, each processor can fail while executing the actions between and including $\ell$-$propose_i$ and $\ell$-$safe_i$ of at most one program, implying that at least $a_0\ell + b_0$ processors must have failed. Therefore, it follows that

$f_1 \geq a_0\ell + b_0$ and, since $\ell_1 > f_2$, we have $a_0 m + b_0 > f_2$. Thus, $f_1$ and $f_2$ do not satisfy the relationship, as assumed, and we have a contradiction. □

## 7 Conclusion

We have given a partial order of set consensus objects, refining the consensus hierarchy of Herlihy. To do so, we used the Borowsky-Gafni simulation technique, proving the stronger version of the technique required for our results. We therefore strengthened the reducibility result derived by Lynch and Rajsbaum, including simulations of algorithms which access set consensus objects. However, our notion of reducibility still allows the simulated programs to have access to just *one kind of* set consensus object (the values $m$ and $\ell$ are fixed for any particular reduction). The theorem could be generalized to allow access to several different kinds of set consensus objects. It would be interesting to see what other extensions of the Borowsky-Gafni simulation technique are possible, possibly including simulations of a wider variety of objects. This could also bring about a stronger, more general notion of reducibility.

Many of the proofs and algorithms are stated informally here due to lack of space. For the complete, more detailed version, refer to [14].

## References


1. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. "Atomic Snapshots of Shared Memory". *Journal of the Association for Computing Machinery*, 40(4):873–890, September 1993.
2. E. Borowsky and E. Gafni, "Generalized FLP Impossibility Result for $t$-resilient Asynchronous Computations", *ACM STOC*, 1993.
3. E. Borowsky and E. Gafni, "The Implication of the Borowsky-Gafni Simulation on the Set-Consensus Hierarchy", *UCLA Tech Report No. 930021.*
4. E. Borowsky and E. Gafni, pre-conference presentation at PODC 1995.
5. S. Chaudhuri, "More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems", *Information and Computation 105 (1)*, July 1993. Appeared earlier in *ACM PODC*, 1990.
6. M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process", *JACM 32*, April 1985. Appeared earlier in *ACM PODC*, 1983.
7. "Wait-Free Synchronization", *ACM TOPLAS 11 (1)*, 1991.
8. "Set Consensus Using Arbitrary Objects", In *Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 324–333. Association for Computing Machinery, ACM Press, 1993.
9. M. Herlihy and N. Shavit, "The Asynchronous Computability Theorem for $t$-Resilient Tasks", *ACM STOC*, 1993.
10. Prasad Jayanti. "On the Robustness of Herlihy's Hierarchy". In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 145–157. Association for Computing Machinery, ACM Press, 1993.
11. N. Lynch and S. Rajsbaum, "On the Borowsky Gafni Simulation Algorithm", *Israel Symposium on Theory of Computing and Systems*, 1996.

12. N. Lynch and M. Tuttle, "An Introduction to Input/Output Automata", TM-373, MIT Laboratory for Computer Science, November 1988.
13. P. Jayanti and S. Toueg, "Some Results on Impossibility, Universality and Decidability of Consensus", *6th WDAG*, Springer Verlag, 1992.
14. P. Reiners, "Understanding the Set Consensus Partial Order Using the Borowsky-Gafni Simulation", *M.S. Thesis*, Iowa State University, 1996.
15. M. Saks and F. Zaharoglou, "Wait-Free $k$-set Agreement is Impossible: The Topology of Public Knowledge", *ACM STOC*, 1993.