The BG Distributed Simulation Algorithm[*]

Elizabeth Borowsky
Computer Science Department
Boston College
Chesnut Hill, MA 02467
borowsky@bc.edu

Eli Gafni
Computer Science Department
University of California, Los Angeles
CA 90024
eli@cs.ucla.edu

Nancy Lynch[†]
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
lynch@theory.lcs.mit.edu

Sergio Rajsbaum[‡]
Instituto de Matemáticas, UNAM
Ciudad Universitaria
D.F. 04510, México
rajsbaum@servidor.unam.mx

January 3, 2001

Abstract

We present a shared memory algorithm that allows a set of $f + 1$ processes to wait-free "simulate" a larger system of n processes, that may also exhibit up to f stopping failures.

Applying this simulation algorithm to the k-set-agreement problem enables conversion of an arbitrary k-fault-tolerant n-process solution for the k-set-agreement problem into a wait-free $k + 1$-process solution for the same problem. Since the $k + 1$-process k-set-agreement problem has been shown to have no wait-free solution [5, 18, 26], this transformation implies that there is no k-fault-tolerant solution to the n-process k-set-agreement problem, for any n.

More generally, the algorithm satisfies the requirements of a *fault-tolerant distributed simulation.* The distributed simulation implements a notion of *fault-tolerant reducibility* between decision problems. This paper defines these notions and gives examples of their application to fundamental distributed computing problems.

The algorithm is presented and verified in terms of I/O automata. The presentation has a great deal of interesting modularity, expressed by I/O automaton composition and both forward and backward simulation relations. Composition is used to include a *safe agreement* module as a subroutine. Forward and backward simulation relations are used to view the algorithm as implementing a *multi-try snapshot* strategy.

The main algorithm works in snapshot shared memory systems; a simple modification of the algorithm that works in read/write shared memory systems is also presented.

---

[*] Preliminary versions of this paper appeared in [5, 22].

[†] Supported by Air Force Contracts AFOSR F49620-92-J-0125 and F49620-97-1-0337, and NSF contract 9225124CCR and CCR-9520298, and DARPA contracts N00014-92-J-4033 and F19628-95-C-0118.

[‡] Part of this work was done at the Laboratory for Computer Science of MIT and at the Cambridge Research Laboratory of DEC. Supported by DGAPA and CONACYT Projects.

1 Introduction

One of the fundamental goals of theoretical computer science is to determine the boundary between problems that are, and are not, computable. In distributed computing, the large number of system parameters compounds this problem. Computability results depend heavily on the communication medium, the number of processes in the system, and the number and type of possible faults. It is difficult in practice to extend a result obtained in one system to apply in another, even if only one of the many system parameters differs between the two systems. In this paper, we take the first steps toward a formal theory for reduction among problems in different models of distributed computing. We consider asynchronous read/write shared memory systems where processes may exhibit stopping failures. There is a parameter f associated to a system, which specifies the maximum number of processes that can fail.

We describe an algorithm, the *BG-simulation algorithm*, that allows a set of $f + 1$ processes with at most f failures, to "simulate" a larger number n of processes, also with at most f failures. The BG-simulation algorithm is a powerful tool for proving solvability and unsolvability results for fault-prone asynchronous systems.

To illustrate the power of the BG-simulation algorithm, consider the n-process k-set agreement problem [8], in which all n processes propose values and decide on at most k of the proposed values. We use the BG-simulation algorithm to convert an arbitrary k-fault-tolerant n-process solution for the k-set-agreement problem into a wait-free $k + 1$-process solution for the same problem. (A wait-free algorithm is one in which any non-failing process terminates, regardless of the failure of any number of the other processes.) Since the $k+1$-process k-set-agreement problem has been shown to have no wait-free solution [5, 18, 26], this transformation implies that there is no k-fault-tolerant solution to the n-process k-set-agreement problem, for any n.

As another application, we show how the BG-simulation algorithm can be used to obtain results of [12, 16] about the computability of some decision problems. Other applications of the algorithm (including variants, related algorithms discussed below, and extensions of it) have appeared in [6, 7, 9, 10, 21, 25, 17].

As these examples suggest, the BG-simulation algorithm is an important tool when studying reducibility among problems in different models of distributed computing. Thus, it is important to understand precisely what the algorithm guarantees. In this paper, we present a complete and careful description of the BG-simulation algorithm, plus a careful description of what it accomplishes, plus a proof of its correctness.

In order to specify the contribution of the BG-simulation algorithm, we define a notion of *fault-tolerant reducibility* between decision problems, and a notion of *fault-tolerant simulation* between shared memory systems. We show that, in a precise sense, any algorithm that implements the fault-tolerant simulation between two systems also implements the reducibility between decision problems solved by the systems. Then we describe a specific version of the BG-simulation algorithm that implements the simulation. These notions are quite natural (although specially tailored to the BG-simulation algorithm) and we believe they can serve as a basis for more general notions of reducibility between decision problems and simulation between systems.

To highlight the limits of the current reducibility, we give examples of pairs of decision problems that do and do not satisfy our notion of fault-tolerant reducibility. For example, the n-process k-set-agreement problem is f-reducible to the n'-process k'-set-agreement problem if $k \geq k'$ and $f \leq \min\{n, n'\}$. On the other hand, these problems are not reducible if $k \leq f < k'$. The moral is that one must be careful in applying the simulation – there are scenarios for which it is appropriate and scenarios for which it is not. One must verify that the conditions for reducibility hold.

We present and verify the BG-simulation algorithm in terms of I/O automata [23]. The presentation has a great deal of modularity, expressed by I/O automaton composition and both forward and backward simulation relations (see [24], for example, for definitions). Composition includes a *safe agreement* module, a simplification of one in

[5], as a subroutine. Forward and backward simulation relations are used to view the algorithm as implementing a *multi-try snapshot* strategy. The most interesting part of the proof is the safety argument, which is handled by the forward and backward simulation relations; once that is done, the liveness argument is straightforward.

We present our main version of the BG-simulation algorithm for a snapshot shared memory system. This makes the correctness proof more modular, and the whole presentation clearer, and is no loss of generality, since a system using snapshot shared memory can be implemented in a wait-free manner in terms of single-writer multi-reader read/write shared variables [1]. For completeness, we briefly present a version that works in read/write shared memory systems. Essentially, the version for read/write systems is obtained by replacing each snapshot operation by a sequence of reads in arbitrary order. The correctness of the resulting read/write systems is proved by arguments analogous to those used for snapshot systems, combined with a special argument showing that the result of a sequence of reads is the same as the result of a snapshot taken somewhere in the interval of the reads.

The original idea of the BG-simulation algorithm and its application to set agreement are due to Borowsky and Gafni [5]. The first precise description of the simulation, including a decomposition into modules, the notion of *fault-tolerant reducibility* between decision problems, and a proof of correctness appeared in Lynch and Rajsbaum [22]. The present paper combines the results of [5] and [22], and adds the abstract notion of fault-tolerant simulation, extensions for read/write systems, computability results, and other details that were not included in [5, 22] for lack of space.

Borowsky and Gafni extended the BG-simulation algorithm to systems including set agreement variables [6]; Chaudhuri and Reiners later formalized this extension in [10, 25], following the techniques of [22].

In the context of consensus, variants of the BG-simulation are used in [9, 21] to simulate systems with access to general shared objects. The BG-simulation requires processes to agree on the outcome of each step by solving (a restricted form of) Consensus using only (read/write) registers. Instead of having processes agree on the outcome of the step as in the BG-simulation, these papers use (a similarly restricted form of) test&set registers to ensure that only one process simulates each step. The simulation of Chandra et al. applies to a context in which TAS registers are available directly and need not be implemented, while Lo and Hadzilacos present a test test&set implementation.

Afek and Stupp [3] use simulation to prove a lower bound on the time-space tradeoff of leader election algorithms that use compare&swap registers. Their simulation reduces a leader election algorithm for a system with compare&swap registers with limited time and space resources to a set agreement algorithm with only read/write variables. Each simulating process simulates a group of statically pre-assigned processes in the simulated algorithm. The coordination is loose, so different executions may end-up being simulated by processes in different groups.

This paper is organized as follows. We start with the model in Section 2. In Section 3 we define decision problems, what it means to solve a decision problem, reducibility between decision problems, and simulation between shared memory systems that solve decision problems. In Section 4 we describe a safe agreement module that is used in the BG-simulation algorithm. In Section 5 we present the BG-simulation algorithm. In Section 6 we present the formal proof of correctness for the BG-simulation algorithm. This implies Theorem 6.10, our main result, which asserts the existence of a distributed algorithm that implements the reducibility and simulation notions of Section 3. In Section 7 we show how to modify the BG-simulation algorithm (for snapshot shared memory), to work in a read/write memory system. In Section 8 several applications of the BG-simulation algorithm are described. A final discussion appears in Section 9.

2 The Model

The underlying model is the I/O automaton model of Lynch and Tuttle [23], as described, for example, in Chapter 8 of [19]. Briefly, an I/O automaton is a state machine whose transitions are labelled with actions. Actions are classified as *input*, *output*, or *internal*. The automaton need not be finite-state, and may have multiple start states. For expressing liveness, each automaton is equipped with a *task* structure (formally, a partition of its non-input actions), and the execution is assumed to give fair turns to each task. The *trace* of an execution is the sequence of external actions occurring in that execution.

Most of the systems in this paper are *asynchronous shared memory* systems, as defined, for example, in Chapter 9 of [19]. Briefly, an n-process asynchronous shared memory system consists of n processes interacting via instantaneously-accessible shared variables. We allow finitely many or infinitely many shared variables. (Allowing infinitely many shared variables is a slight generalization over what appears in [19], but it does not affect any of the properties we require.) Formally, we model the system as a single I/O automaton, whose state consists of all the process local state information plus the values of the shared variables, and whose task structure respects the division into processes. When we discuss fault-tolerance properties, we model process stopping explicitly by means of $stop_i$ input actions, one for each process i. The effect of the action $stop_i$ is to disable all future non-input actions involving process i. When we discuss safety properties only, we omit consideration of the *stop* actions.

In most of this paper, we focus on shared memory systems with *snapshot shared variables*. A snapshot variable for an n-process system takes on values that are length n vectors of elements of some basic data type R. It is accessible by *update* and *snap* operations. An $update(i, r)$ operation has the effect of changing the i'th component of the vector to r; we assume that it can be invoked only by process i. A *snap* operation can be invoked by any process; it returns the entire vector.

We often assume that the i'th component of a snapshot variable is itself divided into components. For example, we use a snapshot variable *mem*, and denote the i'th component by $mem(i)$; this component includes a component $sim\text{-}mem(j)$, denoted $mem(i).sim\text{-}mem(j)$, for each j in some range. We sometimes allow process i to change only one of its components, say $mem(i).sim\text{-}mem(j_0)$, with an *update* operation; this is permissible since process i can remember all the other components and overwrite them.

As we have defined it, a snapshot system may have more than one snapshot shared variable. However, any system with more than one snapshot variable (even with infinitely many snapshot variables) can easily be "implemented" by a system with only a single snapshot variable, with no change in any externally-observable behavior (including behavior in the presence of failures) of the system. Likewise, a system using snapshot shared memory can be "implemented" in terms of single-writer multi-reader read/write shared variables, again with no change in externally-observable behavior; see, e.g., [1] for a construction.

In Section 7 we also consider shared memory systems with single-writer multi-reader read/write shared variables (as defined, for example, in [19]).

3 Decision Problems, Reducibility and Simulation

In Section 3.1 we define decision problems and in Section 3.2 we say what it means for a system to solve a decision problem. In Section 3.3 we define the fault-tolerant reducibility between decision problems. In Section 3.4 we present the notion of simulation.

While the notion of reducibility relates decision problems, we show that the notion of simulation is the equivalent counterpart that relates systems. The following diagram represents these relations, where D and D' are decision problems, and $\mathcal{P}$ and $\mathcal{P}'$ are systems.

$$\begin{tikzcd}
D \arrow[r, "reducible"] & D' \\
\mathcal{P} \arrow[u, "solves"'] \arrow[r, "simulates"] & \mathcal{P}' \arrow[u, "solves"']
\end{tikzcd}$$

We use the following notation. A *relation from X to Y* is a subset of $X \times Y$. A relation R from X to Y is *total* if for every $x \in X$, there is some $y \in Y$ such that $(x, y) \in R$. We write $R(x)$ as shorthand for $\{y : (x, y) \in R\}$. For a relation R from X to Y, and a relation S from Y to Z, $R \cdot S$ denotes the relational composition of R and S, which is a relation from X to Z.

3.1 Decision Problems

Let V be an arbitrary set of values; we use the same V as the input and output domain for all the decision problems in this paper, and V^n denotes the set of all length n vectors with entries from the set V.

An *n-port decision problem* $D = \langle \mathcal{I}, \mathcal{O}, \Delta \rangle$ consists of a set $\mathcal{I}$ of *input vectors,* $\mathcal{I} \subseteq V^n$, a set $\mathcal{O}$ of *output vectors,* $\mathcal{O} \subseteq V^n$, and Δ, a total relation from $\mathcal{I}$ to $\mathcal{O}$.

Example 1 *In the n-process k-set-agreement problem over a set of values V, $|V| \geq k+1$, which we abbreviate as the (n,k)-set-agreement problem, $\mathcal{I}$ is the set of all length n vectors over V, and $\mathcal{O}$ is the set of all length n vectors over V containing at most k different values. For any $w \in \mathcal{I}$, $\Delta(w)$ is the set of all vectors in $\mathcal{O}$ whose values are included among those in w.*

3.2 Solving Decision Problems

Let $D = \langle \mathcal{I}, \mathcal{O}, \Delta \rangle$ be an n-port decision problem; we define what it means for an I/O automaton A (in particular, a shared memory system) to solve D. A is required to have inputs $\mathit{init}(v)_i$ and outputs $\mathit{decide}(v)_i$, where $v \in V$ and $1 \leq i \leq n$. Each such i is associated to a process of a A, and is used to communicate with other modules via the corresponding input and output actions. We say that an $\mathit{init}(v)_i$ or $\mathit{decide}(v)_i$ occurs in *port* i.

We consider A composed with any user automaton U that submits at most one init_i on each port i. We say that a set of $\mathit{init}(v_i)_i$ actions, one for each i, $1 \leq i \leq n$, *forms* the vector $(v_1, \ldots, v_n)$. A set of $\mathit{decide}(v_i)_i$ actions for different values of i can be *completed* to a vector in a given set of n-vectors, if there is one vector in the set, w, such that $w(i) = v_i$ for every $\mathit{decide}(v_i)_i$ action. We require the following conditions:

Well-formedness: A only produces a decide_i if there is a preceding init_i, and A never responds more than once on the same port.

Correct answers: If *init* events occur on all ports, forming a vector $w \in \mathcal{I}$, then the outputs that appear in *decide* events can be completed to a vector in $\Delta(w)$.

We say that A *solves* D provided that for any such U, the composition $A \times U$ guarantees well-formedness and correct answers. In addition, we consider a liveness condition expressing fault-tolerance:

f-failure termination: In any fair execution of $A \times U$, if *init* events occur on all ports and *stop* events occur on at most f ports, then a *decide* occurs on every non-failing port.

A is said to *guarantee f-failure termination* provided that it satisfies the f-failure termination condition for any U, and A is said to guarantee *wait-free termination* provided that it guarantees n-failure termination (or, equivalently, $n-1$-failure termination).

3.3 Fault-Tolerant Reducibility

We define the notion of f-reducibility from an n-port decision problem $D = \langle \mathcal{I}, \mathcal{O}, \Delta \rangle$ to an n'-port decision problem $D' = \langle \mathcal{I}', \mathcal{O}', \Delta' \rangle$, for an integer f, $0 \leq f \leq n'$.

The reducibility is motivated by the way the BG-simulation algorithm operates. In that algorithm, a shared memory system $\mathcal{P}$ simulates an f-fault-tolerant system $\mathcal{P}'$ that solves D'. The simulating system $\mathcal{P}$ is supposed to solve D, and so it obtains from its environment an input vector $w \in \mathcal{I}$, one component per process. Each process i, based on its own input value $w(i)$, determines a "proposed" input vector $g_i(w(i)) \in \mathcal{I}'$. The actual input for each simulated process j of $\mathcal{P}'$ is chosen arbitrarily from among the j^{th} components of the proposed input vectors. Thus, for each $w \in \mathcal{I}$, there is a set $G(w) \subseteq \mathcal{I}'$, of possible input vectors of the simulated system $\mathcal{P}'$.

When the "subroutine" that solves $\mathcal{P}'$ produces a result (a vector in $\mathcal{O}'$), different processes of $\mathcal{P}$ can obtain different partial information about this result. However, with at most f stopping failures, the only difference is that each process can miss at most f components; the possible variations are captured by the F relation below. Then each process i of $\mathcal{P}$ uses its partial information $x(i)$ to decide on a final value, $h_i(x(i))$. The values produced in this way, combined according to the H relation, must form a vector in $\mathcal{O}$. The formal definitions follow.

For a set W of length n vectors and index $i \in \{1, \ldots, n\}$, $W(i)$ denotes $\{w(i) : w \in W\}$, and $\bar{W}$ denotes the Cartesian product $W(1) \times W(2) \times \ldots \times W(n)$. Thus, $\bar{W}$ consists of all the vectors that can be assembled from vectors in W by choosing each component to be the corresponding component of some vector in W.

For a length n vector w of values in V, and $0 \leq f \leq n$, $\mathit{views}_f(w)$ denotes the set of length n vectors over $V \cup \{\bot\}$ that are obtained by changing at most f of the components of w to $\bot$. If W is a set of length n vectors, then $\mathit{views}_f(W)$ denotes $\cup_{w \in W}\{\mathit{views}_f(w)\}$.

Our reducibility is defined in terms of three auxiliary parameterized relations G, F and H, depicted in the following diagram. The relation G is defined by relations $g_1, \ldots, g_n$. The relation H is defined by relations $h_1, \ldots, h_n$, and f. And the relation F is defined by f. Thus we use the notation $G = G(g_1, g_2, \ldots, g_n)$, $H = H(f, h_1, h_2, \ldots, h_n)$, and $F = F(f)$ to emphasize that $g_1, g_2, \ldots, g_n$, $h_1, h_2, \ldots, h_n$, and f are the key parameters whose existence is asserted in the following definition of reducibility.

$$\begin{tikzcd}
\mathcal{I} \arrow[rr, "G"] \arrow[d, "\Delta"] & & \mathcal{I}' \arrow[d, "\Delta'"] \\
\mathcal{O} & F(\mathcal{O}') \arrow[l, "H"'] & \mathcal{O}' \arrow[l, "F"']
\end{tikzcd}$$

1. $G = G(g_1, g_2, \ldots, g_n)$, a total relation from $\mathcal{I}$ to $\mathcal{I}'$; here, each g_i is a function from $\mathcal{I}(i)$ to $\mathcal{I}'$.

   For any $w \in \mathcal{I}$, let $W \subseteq \mathcal{I}'$ denote the set of all vectors of the form $g_i(w(i))$, $1 \leq i \leq n$, and define $G(w) = \bar{W}$. We assume that for each $w \in \mathcal{I}$, $G(w) \subseteq \mathcal{I}'$.

2. $F = F(f)$, a total relation from $\mathcal{O}'$ to $(views_f(\mathcal{O}'))^n$.

   For any $w \in \mathcal{O}'$, $F(w) = (views_f(w))^n$.

3. $H = H(f, h_1, h_2, \ldots, h_n)$, a total (single-valued) relation from $(views_f(\mathcal{O}'))^n$ to V^n; here, each h_i is a function from $views_f(\mathcal{O}')$ to $\mathcal{O}(i)$.

   For any $x \in (views_f(\mathcal{O}'))^n$, $H(x)$ contains exactly the length n vector w such that $w(i) = h_i(x(i))$ for every i.

Definition 3.1 (f-Reducibility) *Suppose that $D = \langle \mathcal{I}, \mathcal{O}, \Delta \rangle$ is an n-port decision problem, $D' = \langle \mathcal{I}', \mathcal{O}', \Delta' \rangle$ is an n'-port decision problem, and $0 \leq f \leq n'$. Then D is* f-reducible *to D' via relations $G = G(g_1, g_2, \ldots, g_n)$ and $H = H(f, h_1, h_2, \ldots, h_n)$, written as $D \leq_f^{G,H} D'$, provided that $G \cdot \Delta' \cdot F \cdot H \subseteq \Delta$.*

The following examples give some pairs of decision problems that do and do not satisfy the reducibility. Because the reducibility expresses the power of the BG-simulation algorithm, the examples indicate situations where the algorithm can and cannot be used.

Example 2 *(n, k)-set agreement is f-reducible to (n', k')-set agreement for $k \geq k'$, $f < \min\{n, n'\}$.*

This is verified as follows. For $v \in V$, define $g_i(v)$ to be the vector $v^{n'}$. Also, for $w \in views_f(V^{n'})$, define $h_i(w)$ to be the first entry of w different from $\perp$. It is easy to check that Definition 3.1 is satisfied.

Example 3 *(n, k)-set agreement is not f-reducible to (n', k')-set agreement if $k \leq f < k'$.*

If this reducibility held, then the main theorem of this paper, Theorem 6.10, together with the fact that (n', k')-set agreement is solvable when $f < k'$ [8], would imply the existence of an f-fault-tolerant algorithm to solve (n, k)-set-agreement. But this contradicts the results of [5, 11, 18, 26].

3.4 Fault-Tolerant Simulation

We present a specification, in the I/O automata formalism, of a fault-tolerant distributed simulation. In Theorem 3.3 we show how this specification corresponds to the reducibility of Section 3.3. The reducibility relates two decision problems, while the simulation relates two shared memory systems.

We start, in Section 3.4.1, by describing the simulated system, $\mathcal{P}'$. Each of the processes in the system, $\mathcal{P}$, that is going to simulate $\mathcal{P}'$ gets its own input. These processes have somehow to produce, out of their inputs, inputs for the simulated processes. Also, out of the outputs produced by the simulated processes, they have somehow to produce outputs for themselves. These two (distributed) procedures, of input translation and of output translation, are what is unique to the fault-tolerant simulation. Together with the natural, step-by-step simulation of $\mathcal{P}'$, they are modeled by an I/O automata called *SimpleSpec*, which is described in Section 3.4.2. Finally, in Section 3.4.3, we present a formal definition of simulation, and show that it implements our reducibility notion.

3.4.1 The Simulated Algorithm $\mathcal{P}'$

We assume that the algorithm to be simulated is given in the form of an n'-process snapshot shared memory system, $\mathcal{P}'$. It has only a single snapshot shared variable, called *mem*$'$. We assume that each component of *mem*$'$ takes on values in a set R, with a distinguished initial value r_0. Thus, the snapshot shared variable *mem*$'$ has a unique initial value, consisting of r_0 in every component. Furthermore, we assume that $\mathcal{P}'$ solves a decision problem D'. In this subsection and the next, we consider only safety properties, and so we omit the *stop* actions.

We make some simplifying "determinism" assumptions about $\mathcal{P}'$, without loss of generality: We assume that each process has only one initial state. Also, each process has, in any state, at most one non-input action enabled. Moreover, for any action performed from any state, we assume that there is a uniquely-defined next state. Also, the initial state of each process is "quiescent" – no non-input actions are enabled (until an input arrives). For each other state, exactly one non-input action is enabled. In any state after a process has executed a *"decide"*, only local actions are enabled.

The following is some useful terminology about system $\mathcal{P}'$. For any state s of a process j of $\mathcal{P}'$, define $\mathit{nextop}(s)$ to be an element of $\{\text{"init"}, \text{"snap"}, \text{"local"}\} \cup \{(\text{"update"}, r) : r \in R\} \cup \{(\text{"decide"}, v) : v \in V\}$. Specifically, for a quiescent state s, $\mathit{nextop}(s) = \text{"init"}$; for a state s in which the next action is a *snap*, $\mathit{nextop}(s) = \text{"snap"}$; for a state s in which the next action is an $\mathit{update}(i, r)$, $\mathit{nextop}(s) = (\text{"update"}, r)$; for a state s in which the next action is local, $\mathit{nextop}(s) = \text{"local"}$; and for a state s in which the next action is to decide on value v, $\mathit{nextop}(s) = (\text{"decide"}, v)$. Our determinism assumptions imply that for each state s, $\mathit{nextop}(s)$ is uniquely defined.

For any state s of a process j such that $\mathit{nextop}(s) = \text{"init"}$ and any $v \in V$, define $\mathit{trans\text{-}init}(s, v)$ to be the state that results from applying $\mathit{init}(v)_j$ to s. For any state s of a process j such that $\mathit{nextop}(s) = \text{"snap"}$ and any $w \in R^{n'}$, define $\mathit{trans\text{-}snap}(s, w)$ to be the state that results from performing the snapshot operation from state s, with the return value for the snapshot being w. Finally, for any state s of a process j such that $\mathit{nextop}(s)$ is an *"update"*, *"local"*, or *"decide"* pair, define $\mathit{trans}(s)$ to be the state of j that results from performing the operation from state s.

3.4.2 The *SimpleSpec* Automaton

Consider algorithm $\mathcal{P}'$, which solves problem D' guaranteeing f-failure termination, together with relations G and H. The definition of what we mean by a simulation is based on a safety specification expressed by the $\mathit{SimpleSpec}_f^{G,H}(\mathcal{P}')$ automaton, or simply *SimpleSpec*. A system of n processes, $\mathcal{P}$, which is supposed to simulate $\mathcal{P}'$, should implement *SimpleSpec*, in a sense described in Section 3.4.3.

The *SimpleSpec* automaton directly simulates system $\mathcal{P}'$, in a centralized manner. Repeatedly, a process j of $\mathcal{P}'$ is chosen nondeterministically and its next step simulated. The only unusual feature is the way of choosing the inputs for the $\mathcal{P}'$ processes and the outputs for the $\mathcal{P}$ processes, using G and H relations. In order to determine an input v for a process j of $\mathcal{P}'$, a process i is chosen nondeterministically from among those that have received their inputs, and v is set to the j-th component of the vector $g_i(\mathit{input}(i))$. At any time after at least $n' - f$ of the processes of $\mathcal{P}'$ have produced decision values, outputs can be produced, using the functions h_i.

We give a formal description of the *SimpleSpec* automaton.

SimpleSpec:

Signature:

Input:	Internal:
$\mathit{init}(v)_i, i \in \{1, \ldots, n\}$	$\mathit{sim\text{-}init}_j, j \in \{1, \ldots, n'\}$
Output:	$\mathit{sim\text{-}snap}_j, j \in \{1, \ldots, n'\}$
$\mathit{decide}(v)_i, i \in \{1, \ldots, n\}$	$\mathit{sim\text{-}update}_j, j \in \{1, \ldots, n'\}$
	$\mathit{sim\text{-}local}_j, j \in \{1, \ldots, n'\}$
	$\mathit{sim\text{-}decide}_j, j \in \{1, \ldots, n'\}$

States:

```
    sim-mem, a memory of $\mathcal{P}'$ (an element of $R^{n'}$), initially the initial memory $(r_0)^{n'}$
    for each $i \in \{1, \ldots, n\}$:
        input($i$) $\in V \cup \{\perp\}$, initially $\perp$
        reported($i$), a Boolean, initially false
    for each $j \in \{1, \ldots, n'\}$:
        sim-state($j$), a state of $j$, initially the initial state
        sim-decision($j$) $\in V \cup \{\perp\}$, initially $\perp$
```

Transitions:

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

  sim-init$_j$
      Precondition:
          nextop(sim-state($j$)) = "init"
          for some $i$
              input($i$) $\neq \perp$
              $v = g_i$(input($i$))($j$)
      Effect:
          sim-state($j$) := trans-init(sim-state($j$), $v$)

  sim-snap$_j$
      Precondition:
          nextop(sim-state($j$)) = "snap"
      Effect:
          sim-state($j$) :=
              trans-snap(sim-state($j$), sim-mem)

  sim-update$_j$
      Precondition:
          nextop(sim-state($j$)) = ("update", $r$)
      Effect:
          sim-state($j$) := trans(sim-state($j$))
          sim-mem($j$) := $r$

  sim-local$_j$
      Precondition:
          nextop(sim-state($j$)) = "local"
      Effect:
          sim-state($j$) := trans(sim-state($j$))

  sim-decide$_j$
      Precondition:
          nextop(sim-state($j$)) = ("decide", $v$)
      Effect:
          sim-state($j$) := trans(sim-state($j$))
          sim-decision($j$) := $v$

  decide($v$)$_i$
      Precondition:
          input($i$) $\neq \perp$
          reported($i$) = false
          $w$ is a "subvector" of sim-decision
          $|w| \geq n' - f$
          $v = h_i(w)$
      Effect:
          reported($i$) := true
```

Tasks:

Arbitrary. They are not used in the proof.

A *sim-init*$_j$ action is used to simulate an *init* step of process j. To simulate any other step of j, the function *nextop* is used to determine what the next operation is: *"init"*, *"snap"*, (*"update"*, r), *"local"*, or (*"decide"*, v). Then the state transition specified by $\mathcal{P}'$ is performed, using the appropriate function: *trans-init*, *trans-snap* or *trans*. Once the simulation of at least $n' - f$ processes has been completed a decision value for i can be produced, using h_i. In the code this is expressed by a "subvector" of *sim-decision*, where "subvector" means replacing zero or more entries of the vector *sim-decision* by $\perp$, and $|w|$ is the number of entries different from $\perp$.

Theorem 3.1 *Assume $\mathcal{P}'$ solves D' and $D \leq_f^{G,H} D'$.*
Then SimpleSpec$_f^{G,H}(\mathcal{P}')$ *solves D.*

Proof: Following Section 3.2, we consider $\mathit{SimpleSpec}_f^{G,H}(\mathcal{P}')$ composed with any user automaton U that submits at most one init_i on each port i.

To prove well-formedness, we note that it follows directly from the code that $\mathit{SimpleSpec}_f^{G,H}(\mathcal{P}')$ only produces a decide_i if there is a preceding init_i, and it never responds more than once on the same port.

To prove correct answers, assume *init* events occur on all ports, forming a vector $w \in \mathcal{I}$. Then the code for *sim-init* guarantees that the inputs for $\mathcal{P}'$ that are produced can be completed to a vector $w' \in G(w)$. Then the code of $\mathit{SimpleSpec}_f^{G,H}(\mathcal{P}')$ simulates a centralized execution of $\mathcal{P}'$ with these inputs, and hence the vector w'' of output values that is stored in *sim-decision* can be completed to a vector in $\Delta'(w')$. Then the code for *decide* guarantees that the outputs that appear in *decide* events can be completed to a vector in $H(F(w''))$. It follows that the outputs appearing in *decide* events can be completed to a vector in $H(F(\Delta'(G(w))))$, and hence (since $D \leq_f^{G,H} D'$) to a vector in $\Delta(w)$. Thus, $\mathit{SimpleSpec}_f^{G,H}(\mathcal{P}')$ produces correct answers. ■

3.4.3 Definition of Simulation

We now define a notion of fault-tolerant simulation; our definition includes both safety and liveness conditions. We had to make two choices for this definition. First, on the way the simulating processes produce inputs for the simulated processes from their own inputs, and on the way they produce outputs from the outputs of the simulated processes. Our choice was defined by the way the the BG-simulation algorithm operates. The second choice is about how detailed the simulation should be. One possibility that comes to mind is to require a step-by-step simulation, executing each instruction of each simulated program. Our choice was to use the weakest notion of simulation that would still be sufficient for the applications we present. Our simulation specification deals only with external behaviors, and does not require that the program given by $\mathcal{P}'$ be simulated step-by-step. The key property guaranteed by such a simulation is formally stated in Theorem 3.3.

We need a preliminary definition and lemma. Suppose that A and B are two I/O automata with the same inputs $\mathit{init}(v)_i$ and outputs $\mathit{decide}(v)_i$, $v \in V$, $1 \leq i \leq n$. We consider A and B composed with any user automaton U that submits at most one init_i on each port i. We say that A *solves* B provided that for any such U, every trace of the composition $A \times U$ is also a trace of the composition $B \times U$.

Lemma 3.2 *Suppose that A and B are two I/O automata with the same inputs $\mathit{init}(v)_i$ and outputs $\mathit{decide}(v)_i$, $v \in V$, $1 \leq i \leq n$. If A solves B and B solves an n-port decision problem D then A solves D.*

Proof: By assumption, every trace of $A \times U$ is also a trace of $B \times U$. Since B solves D, every trace of $B \times U$ satisfies well-formedness and correct answers. Therefore, every trace of $A \times U$ satisfies well-formedness and correct answers, so A solves D. ■

Definition 3.2 (fault-tolerant simulation) *Suppose that $\mathcal{P}$ is an n-process shared memory system, $\mathcal{P}'$ is an n'-process shared memory system, and $0 \leq f \leq n'$. Then $\mathcal{P}$ f*-simulates *$\mathcal{P}'$ via relations $G = G(g_1, g_2, \ldots, g_n)$ and $H = H(f, h_1, h_2, \ldots, h_n)$, written as $\mathcal{P}$ simulates$_f^{G,H}$ $\mathcal{P}'$, provided that both of the following hold:*
(1) $\mathcal{P}$ solves $\mathit{SimpleSpec}_f^{G,H}(\mathcal{P}')$.
(2) If $\mathcal{P}'$ guarantees f-failure termination then $\mathcal{P}$ guarantees f-failure termination.

Note that condition (1) involves safety only, and so we follow the convention (of Section 2) of not including the *stop* actions in $\mathcal{P}$ and $\mathcal{P}'$. However, condition (2) is a fault-tolerance condition, and so we assume there that the *stop* actions are included, according to the convention.

The relationship between our simulation and reducibility notions is as follows:

Theorem 3.3 *Assume $\mathcal{P}'$ solves D' and guarantees f-failure termination. Assume that $D \leq_f^{G,H} D'$ and $\mathcal{P}$ $simulates_f^{G,H}$ $\mathcal{P}'$. Then $\mathcal{P}$ solves D and guarantees f-failure termination.*

Proof: We first show that $\mathcal{P}$ solves D. Theorem 3.1 implies that $SimpleSpec_f^{G,H}(\mathcal{P}')$ solves D. By property (1) of the definition of f-simulation, we have that $\mathcal{P}$ solves $SimpleSpec_f^{G,H}(\mathcal{P}')$. Therefore, Lemma 3.2 implies that $\mathcal{P}$ solves D, as needed.

Now we show that $\mathcal{P}$ guarantees f-failure termination. We know that $\mathcal{P}'$ guarantees f-failure termination. Since $\mathcal{P}$ $simulates_f^{G,H}$ $\mathcal{P}'$, property (2) of the definition of f-simulation implies that $\mathcal{P}$ guarantees f-failure termination, as needed. ■

Later we use Theorem 3.3 to show that if $\mathcal{P}'$ solves D' with f-failure termination and $D \leq_f^{G,H} D'$, then there exists a snapshot shared memory system $\mathcal{P}$ that solves D with f-failure termination. The proof consists of describing a specific snapshot shared memory system $\mathcal{P}$ such that $\mathcal{P}$ $simulates_f^{G,H}$ $\mathcal{P}'$. This result is stated in Theorem 6.10; the corresponding version for read/write shared memory systems is stated in Theorem 7.5.

Notice that this simulation specification deals only with external behaviors, and does not require that the program given by $\mathcal{P}'$ be simulated step-by-step. This requirement is sufficient for the applications we present.

4 A Safe Agreement Module

The simulation algorithm uses a component that we call a *safe agreement* module. This module solves a variant of the ordinary agreement problem and guarantees failure-free termination. In addition, it guarantees a nice resiliency property: its susceptibility to failure on each port is limited to a designated "unsafe" portion of an execution. If no failure occurs during these unsafe intervals, then decisions are guaranteed on all non-failing ports on which invocations occur.

Formally, we assume that the module communicates with its "users" on a set of n *ports* numbered $1, \ldots, n$. Each port i supports input actions of the form $propose(v)_i$, $v \in V$, by which a user at port i proposes specific values for agreement, and output actions of the form $safe_i$ and $agree(v)_i$, $v \in V$. The $safe_i$ action is an announcement to the user at port i that the unsafe portion of the execution corresponding to port i has been completed, and the $agree(v)_i$ is an announcement on port i that the decision value is v. In addition, we assume that port i supports an input action $stop_i$, representing a stopping failure.

We say that a sequence of $propose_i$, $safe_i$ and $agree_i$ actions is *well-formed* for i provided that it is a prefix of a sequence of the form $propose(v)_i, safe_i, agree_i$. We assume that the users preserve well-formedness on every port, i.e., there is at most one $propose_i$ event for any particular i. Then we require the following properties of any execution of the module together with its users:

Well-formedness: For any i, the interactions between the module and its users on port i are well-formed for i.

Agreement: All agreement values are identical.

Validity: Any agreement value must be proposed.

In addition, we require two liveness conditions, which are stated in terms of fair executions. The first condition says that any *propose* event on a non-failing port eventually receives a *safe* announcement. This guarantee is required in spite of any failures on other ports.

Wait-free progress: In any fair execution, for any i, if a $\mathit{propose}_i$ event occurs and no stop_i event occurs, then a safe_i event occurs.

The second liveness condition says that if the execution does not remain unsafe for any port, then any *propose* event on a non-failing port eventually receives an *agree* announcement.

Safe termination: In any fair execution, if there is no j such that $\mathit{propose}_j$ occurs and safe_j does not occur, then for any i, if a $\mathit{propose}_i$ event occurs and no stop_i event occurs, then agree_i occurs.

An I/O automaton with the appropriate interface is said to be a *safe agreement module* provided that it guarantees all the preceding conditions (for all users).

We now describe a simple design (using snapshot shared memory) for a safe agreement module. It is a slight simplification of the one in [5].

The snapshot shared memory contains a *val* component and a *level* component for each process i. When process i receives a $\mathit{propose}(v)_i$, it records the value v in its *val* component and raises its *level* to 1. Then i uses a snapshot to determine the *level*'s of the other processes. If i sees that any process has attained $\mathit{level} = 2$, then it backs off and resets its *level* to 0, and otherwise, it raises its *level* to 2.

Next, process i enters a wait loop, repeatedly taking snapshots until it sees a situation where no process has $\mathit{level} = 1$. When this happens, the set of processes that it sees with $\mathit{level} = 2$ is nonempty. Let v be the *val* value of the process with the smallest index with $\mathit{level} = 2$. Then process i performs an $\mathit{agree}(v)_i$ output.

In the following code, we do not explicitly represent the stop_i actions. We assume that the stop_i action just puts process i in a special "stopped" state, from which no further non-input steps are enabled, and after which any input causes no changes.

SafeAgreement:

Shared variables:

x, a length n snapshot value; for each i, $x(i)$ has components:
  $\mathit{level} \in \{0, 1, 2\}$, initially 0
  $\mathit{val} \in V \cup \{\bot\}$, initially $\bot$

Actions of i:

Input:
  $\mathit{propose}(v)_i$, $v \in V$
Output:
  safe_i
  $\mathit{agree}(v)_i$

Internal:
  $\mathit{update1}_i$
  $\mathit{snap1}_i$
  $\mathit{update2}_i$
  wait_i

States of i:

$\mathit{input} \in V \cup \{\bot\}$, initially $\bot$
$\mathit{output} \in V \cup \{\bot\}$, initially $\bot$
x-local, a snapshot value; for each j, $\mathit{x\text{-}local}(j)$ has components:
  $\mathit{level} \in \{0, 1, 2\}$, initially 0
  $\mathit{val} \in V \cup \{\bot\}$, initially $\bot$
$\mathit{status} \in \{\mathit{idle}, \mathit{update1}, \mathit{snap1}, \mathit{update2}, \mathit{safe}, \mathit{wait}, \mathit{report}\}$, initially *idle*

Transitions of i:

```
$propose(v)_i$
    Effect:
        $input := v$
        $status := update1$

$update1_i$
    Precondition:
        $status = update1$
    Effect:
        $x(i).level := 1$
        $x(i).val := input$
        $status := snap1$

$snap1_i$
    Precondition:
        $status = snap1$
    Effect:
        $x\text{-}local := x$
        $status := update2$

$update2_i$
    Precondition:
        $status = update2$
    Effect:
        if $\exists j : x\text{-}local(j).level = 2$
        then $x(i).level := 0$
        else $x(i).level := 2$
        $status := safe$

$safe_i$
    Precondition:
        $status = safe$
    Effect:
        $status := wait$

$wait_i$
    Precondition:
        $status = wait$
    Effect:
        if $\nexists j : x(j).level = 1$
            and $\exists j : x(j).level = 2$ then
          $k := \min\{j : x(j).level = 2\}$
          $output := x(k).val$
          $status := report$

$agree(v)_i$
    Precondition:
        $status = report$
        $v = output$
    Effect:
        $status := idle$
```

Tasks of i:

All actions comprise a single task.

Theorem 4.1 *SafeAgreement is a safe agreement module.*

Proof: Well-formedness and validity are easy to see. We argue agreement, using an operational argument. Suppose that process i is the first to perform a successful *wait* step, that is, one that causes it to decide, and suppose that it decides on the *val* of process k. Let π be the successful $wait_i$ step; then at step π, process i sees that $x(j).level \neq 1$ for all j, and k is the smallest index such that $x(k).level = 2$.

We claim that no process j subsequently sets $x(j).level := 2$. Suppose for the sake of contradiction that process j does subsequently set $x(j).level := 2$ in an $update2_j$ step, ϕ. Since $x(j).level \neq 1$ when π occurs, it must be that process j must perform an $update1_j$ and a $snap1_j$ after π and before ϕ. But then process j must see $x(k).level = 2$ when it performs its $snap1_j$, which causes it to back off, setting $x(j).level := 0$. This is a contradiction, which implies that no process j subsequently sets $x(j).level := 2$. But this implies that any process that does a successful *wait* step will also see k as the smallest index such that $x(k).level = 2$, and will therefore also decide on k's *val.*

The wait-free progress property is immediate, because process i proceeds without any delay until it performs its $safe_i$ output action.

To see the safe termination property, assume that there is no j such that $propose_j$ occurs and $safe_j$ does not occur. Then there is no j such that $x(j).level$ remains equal to 1 forever, so eventually all the *level* values are in

$\{0, 2\}$. Then any non-failing process i will succeed in any subsequent $wait_i$ statement, and so eventually performs an $agree_i$ output action. ■

5 The BG Simulation Algorithm

In this section, we present the basic snapshot shared memory simulation algorithm, which we will show satisfies Definition 3.2.

We present the algorithm as an n-process snapshot shared memory system $\mathcal{Q}$ with a single snapshot shared variable. This algorithm is assumed to interact not only with the usual environment, via *init* and *decide* actions, but also with a two-dimensional array of safe agreement modules $A_{j,\ell}$, $j \in \{1, \ldots, n'\}$, $\ell \in N$, $N = \{0, 1, 2, \ldots\}$. In the final version of the simulation algorithm, system $\mathcal{P}$, these safe agreement modules are replaced by implementations and the whole thing implemented by a snapshot shared memory system with a single shared variable. The system $\mathcal{Q}$ is assumed to interact with each $A_{j,\ell}$ via outputs $propose(w)_{j,\ell,i}$ and inputs $safe_{j,\ell,i}$ and $agree(w)_{j,\ell,i}$. Here, we subscript the safe agreement actions by the particular instance of the protocol. For $\ell = 0$, we have $w \in V$. For $\ell \in N^+$, we have $w \in R^{n'}$.

System $\mathcal{Q}$ simulates the n' processes of $\mathcal{P}'$ ($\mathcal{P}'$ is described in Section 3.4.1), using a safe agreement protocol $A_{j,0}$ to allow all processes of $\mathcal{Q}$ to agree on the input of each process j, and also a safe agreement protocol $A_{j,\ell}$, $\ell \in N^+$ to allow all processes to agree on the value returned by the ℓ'th simulated snapshot statement of each process j. Other steps are simulated directly, with no agreement protocol. Each process i of $\mathcal{Q}$ simulates the steps of each process j of $\mathcal{P}'$ in order, waiting for each to complete before going on to the next one. Process i works concurrently on simulating steps of different processes of $\mathcal{P}'$. However, it is only permitted to be in the "unsafe" portion of its execution for one process j of $\mathcal{P}'$ at a time.

To simulate process j, process i keeps locally the current value of the state of j, in *sim-state(j)*, the number of steps that it has simulated for j, in *sim-steps(j)*, and the number of snapshots that it has simulated for j, in *sim-snaps(j)*. The shared memory of $\mathcal{Q}$ is a single snapshot variable *mem*, containing a portion $mem(i)$ for each process i of $\mathcal{Q}$. In its component, process i keeps track of the latest values of all the components of the snapshot variable of $\mathcal{P}'$, according to i's local simulation of $\mathcal{P}'$. Process i keeps the value of j's component in $mem(i).sim\text{-}mem(j)$. Along with this value, it keeps a counter in $mem(i).sim\text{-}steps(j)$, which counts the number of steps that it has simulated for j, up to and including the latest step at which process j of $\mathcal{P}'$ updated its component.

A function *latest* is used in the *snap* action to combine the information in the various components of *mem* to produce a single length n' vector of R values, representing the latest values written by all the processes of $\mathcal{P}'$. This function operates "pointwise" for each j, selecting the $sim\text{-}mem(j)$ value associated with the highest $sim\text{-}steps(j)$. More precisely, assume $k = \max_i\{mem(i).sim\text{-}steps(j)\}$. Then, let $\hat{i}$ be an index such that $mem(\hat{i}).sim\text{-}steps(j) = k$. The function *latest* selects, for j, the value $mem(\hat{i}).sim\text{-}mem(j)$. As we shall see (in Lemma 6.3), this value must be unique.

When process i simulates a decision step of j, it stores the decision value in the local variable $sim\text{-}decision(j)$. Once process i has simulated decision steps of at least $n' - f$ processes, that is, when $|sim\text{-}decision| \geq n' - f$, it computes a decision value v for itself, using the function h_i, that is, $v := h_i(sim\text{-}decision)$.

In the following code, we do not represent the *stop* actions, since the difficult part of the correctness proof is the safety argument. After the safety argument we give the fault-tolerance argument, and introduce the *stop* actions.

Simulation System $\mathcal{Q}$:

Shared variables:

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

Actions of i:

Input:
  init$(v)_i$, $v \in V$
  safe$_{j,\ell,i}$, $\ell \in N$
  agree$(v)_{j,\ell,i}$, $\ell = 0$ and $v \in V$,
    or $\ell \in N^+$ and $v \in R^n$
Output:
  decide$(v)_i$, $v \in V$
  propose$(v)_{j,\ell,i}$, $\ell = 0$ and $v \in V$,
    or $\ell \in N^+$ and $v \in R^{n'}$

Internal:
  sim-update$_{j,i}$
  snap$_{j,i}$
  sim-local$_{j,i}$
  sim-decide$_{j,i}$

States of i:

input $\in V \cup \{\bot\}$, initially $\bot$
reported, a Boolean, initially *false*
for each j:
  sim-state(j), a state of j, initially the initial state
  sim-steps$(j) \in N$, initially 0
  sim-snaps$(j) \in N$, initially 0
  status$(j) \in \{$*idle*, *propose*, *unsafe*, *safe*$\}$, initially *idle*
  sim-mem-local$(j) \in R^{n'}$, initially arbitrary
  sim-decision$(j) \in V \cup \{\bot\}$, initially $\bot$

Transitions of i:

```
$\textit{init}(v)_i$
    Effect:
        $\textit{input} := v$

$\textit{propose}(v)_{j,0,i}$
    Precondition:
        $\textit{status}(j) = \textit{idle}$
        $\nexists k : \textit{status}(k) = \textit{unsafe}$
        $\textit{nextop}(\textit{sim-state}(j)) = \text{“}\textit{init}\text{”}$
        $\textit{input} \neq \perp$
        $v = g_i(\textit{input})(j)$
    Effect:
        $\textit{status}(j) := \textit{unsafe}$

$\textit{safe}_{j,\ell,i}$
    Effect:
        $\textit{status}(j) := \textit{safe}$

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

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

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

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

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

$\textit{sim-local}_{j,i}$
    Precondition:
        $\textit{nextop}(\textit{sim-state}(j)) = \text{“}\textit{local}\text{”}$
    Effect:
        $\textit{sim-state}(j) := \textit{trans}(\textit{sim-state}(j))$
        $\textit{sim-steps}(j) := \textit{sim-steps}(j) + 1$

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

$\textit{decide}(v)_i$
    Precondition:
        $\textit{input} \neq \perp$
        $\textit{reported} = \textit{false}$
        $|\textit{sim-decision}| \geq n' - f$
        $v = h_i(\textit{sim-decision})$
    Effect:
        $\textit{reported} := \textit{true}$
```

Tasks of i:

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

---

6 Correctness Proof

The liveness proof, which is quite simple, is postponed to the end of this section. We start with the proofs of safety properties for the main simulation algorithm. For these, we use invariants involving the states of the safe agreement modules. Since we do not want these invariants to depend on any particular implementation of safe agreement, we add abstract state information, in the form of history variables that are definable for all correct safe agreement implementations:

$\mathit{proposed\text{-}vals} \subseteq V$, initially $\emptyset$
$\mathit{agreed\text{-}val} \in V \cup \{\bot\}$, initially $\bot$
$\mathit{proposed\text{-}procs} \subseteq \{1, \ldots, n\}$, initially $\emptyset$
$\mathit{agreed\text{-}procs} \subseteq \{1, \ldots, n\}$, initially $\emptyset$

These history variables are maintained by adding the following new effects to actions:

$\mathit{propose}(v)_i$
Effect:
$\mathit{proposed\text{-}vals} := \mathit{proposed\text{-}vals} \cup \{v\}$
$\mathit{proposed\text{-}procs} := \mathit{proposed\text{-}procs} \cup \{i\}$

$\mathit{agree}(v)_i$
Effect:
$\mathit{agreed\text{-}val} := v$
$\mathit{agreed\text{-}procs} := \mathit{agreed\text{-}procs} \cup \{i\}$

For the safety part of the proof, we use three levels of abstraction, related by forward and backward simulation relations. Forward and backward simulation relations are notions used to show that one I/O automaton implements another [24], or in our case, that one I/O automaton solves another; they have nothing to do with "simulations" in the sense of the BG simulation algorithm. The first level of abstraction is the specification itself; that is, the *SimpleSpec* automaton. The second level of abstraction is the *DelayedSpec* automaton described next in Section 6.1. The third level of abstraction is the simulation algorithm $\mathcal{P}$ itself (obtained by composing $\mathcal{Q}$ with safe agreement implementations). We will prove in Section 6.1 that *DelayedSpec* solves *SimpleSpec*, and in Section 6.2 that $\mathcal{P}$ solves *DelayedSpec*. This implies that $\mathcal{P}$ solves *SimpleSpec*, which is what is needed for the safety part of Definition 3.2.

6.1 The *DelayedSpec* Automaton

Our second level of abstraction is the *DelayedSpec* automaton. This is a slight modification of *SimpleSpec*, which replaces each snapshot step of a process j of $\mathcal{P}'$ ($\mathit{sim\text{-}snap}_j$) with a series of $\mathit{snap\text{-}try}_j$ steps during which snapshots are taken and their values recorded, followed by one $\mathit{snap\text{-}succeed}_j$ step in which one of the recorded snapshot values is chosen for actual use.

The *DelayedSpec* automaton is the same as *SimpleSpec*, except for the snapshot attempts. There is an extra state component $\mathit{snap\text{-}set}(j)$, which keeps track of the set of snapshot vectors that result from doing $\mathit{snap\text{-}try}_j$ actions. The *sim-snap* actions are omitted.

DelayedSpec:
Signature:

Input: As in *SimpleSpec* Output: As in *SimpleSpec*	Internal: As in *SimpleSpec* but instead of $\mathit{sim\text{-}snap}_j$, $j \in \{1, \ldots, n'\}$: $\mathit{snap\text{-}try}_j$ $\mathit{snap\text{-}succeed}_j$

States:

As in *SimpleSpec* but in addition:
$\mathit{snap\text{-}set}(j)$, a set of vectors in $R^{n'}$, initially empty

Transitions: As in *SimpleSpec* but instead of $\mathit{sim\text{-}snap}_j$:

```
snap-try_j
    Precondition:
        $nextop(sim\text{-}state(j)) = \text{“snap”}$
    Effect:
        $snap\text{-}set(j) := snap\text{-}set(j) \cup \{sim\text{-}mem\}$
```

```
snap-succeed_j
    Precondition:
        $nextop(sim\text{-}state(j)) = \text{“snap”}$
        $w \in snap\text{-}set(j)$
    Effect:
        $sim\text{-}state(j) := trans\text{-}snap(sim\text{-}state(j), w)$
        $snap\text{-}set(j) := \emptyset$
```

Tasks:

As in *SimpleSpec*

It should not be hard to believe that *DelayedSpec* solves *SimpleSpec*– the result of a sequence of *snap-try* steps plus one *snap-succeed* step is the same as if a single *sim-snap* occurred at the point of the selected snapshot. Formally, we use a backward simulation to prove the implementation relationship. The reason for the backward simulation is that the decision of which snapshot is selected is made after the point of the simulated snapshot step.

The backward simulation relation we use (for any fixed U) is the relation b from states of $\mathit{DelayedSpec} \times U$ to states of $\mathit{SimpleSpec} \times U$ that is defined as follows. If s is a state of $\mathit{DelayedSpec} \times U$ and u is a state of $\mathit{SimpleSpec} \times U$, then $(s, u) \in b$ provided that the following all hold:

1. The state of U is the same in u and s.
2. $u.\mathit{sim\text{-}mem} = s.\mathit{sim\text{-}mem}$.
3. For each i,
    (a) $u.\mathit{input}(i) = s.\mathit{input}(i)$.
    (b) $u.\mathit{reported}(i) = s.\mathit{reported}(i)$.
4. For each j,
    (a) $u.\mathit{sim\text{-}state}(j) \in \{s.\mathit{sim\text{-}state}(j)\} \cup \{\mathit{trans\text{-}snap}(s.\mathit{sim\text{-}state}(j), w) : w \in s.\mathit{snap\text{-}set}(j)\}$.
    (b) $u.\mathit{sim\text{-}decision}(j) = s.\mathit{sim\text{-}decision}(j)$.

That is, all state components are the same in u and s, with the sole exception that $u.\textit{sim-state}(j) \in \{s.\textit{sim-state}(j)\} \cup \{\textit{trans-snap}(s.\textit{sim-state}(j), w) : w \in s.\textit{snap-set}(j)\}$, that is, $u.\textit{sim-state}(j)$ is either $s.\textit{sim-state}(j)$, or else the result of applying one of the snapshot results to $s.\textit{sim-state}(j)$. Each $\textit{sim-step}_j$ step of *SimpleSpec* is "implemented" by a chosen $\textit{snap-try}_j$ step of *Delayed Spec.*

Lemma 6.1 *Relation b is a backward simulation from DelayedSpec$\times U$ to SimpleSpec$\times U$.*

Sketch of proof: Let (s, π, s') be a step of $\textit{DelayedSpec} \times U$, and let $(s', u') \in b$. We produce a corresponding execution fragment of $\textit{SimpleSpec} \times U$, from u to u', with $(s, u) \in b$. The construction is in cases based on the type of action. The interesting cases are *snap-try* and *snap-succeed*:

1. $\pi = \textit{snap-try}_j$.

   Let x denote $s.\textit{sim-mem}$. If $u'.\textit{sim-state}(j) = \textit{trans-snap}(s'.\textit{simstate}(j), x)$, then let the corresponding execution fragment be $(u, \textit{sim-snap}_j, u')$, where u is the same as u', except that $u.\textit{sim-state}(j) = s.\textit{sim-state}(j)$. This is an execution fragment because $s.\textit{sim-state}(j) = s'.\textit{sim-state}(j)$.

   Otherwise, let the corresponding execution fragment be just the single state u'. That is, $u = u'$. Then we know that, either (i) $u'.\textit{sim-state}(j) = s'.\textit{sim-state}(j)$, or (ii) $u'.\textit{sim-state}(j) \in \{\textit{trans-snap}(s'.\textit{sim-state}(j), w) : w \in s'.\textit{snap-set}(j), w \neq x\}$. Since $u = u'$, we need to prove that $u'.\textit{sim-state}(j)$ is in the set $\{s.\textit{sim-state}(j)\} \cup \{\textit{trans-snap}(s.\textit{sim-state}(j), w) : w \in s.\textit{snap-set}(j)\}$. If case (i) holds the claim follows easily from the fact that $s.\textit{sim-state}(j) = s'.\textit{sim-state}(j)$. Hence, assume case (ii) holds. We know that $s.\textit{snap-set}(j) \supseteq s'.\textit{snap-set}(j) - \{x\}$, so $u'.\textit{sim-state}(j) = \textit{trans-snap}(s'.\textit{sim-state}(j), w)$, where $w \in s.\textit{snap-set}(j)$. The proof follows since $s.\textit{sim-state}(j) = s'.\textit{sim-state}(j)$.

2. $\pi = \textit{snap-succeed}_j$.

   The corresponding execution fragment consists of only the single state u'. We must show that $(s, u') \in b$. Fix $x \in s.\textit{snap-set}(j)$ to be the snapshot value selected in the step we are considering.

   Everything carries over immediately, except for the equation involving the $u'.\textit{sim-state}(j)$ component. For this, we know that $u'.\textit{sim-state}(j) \in \{s'.\textit{sim-state}(j)\} \cup \{\textit{trans-snap}(s'.\textit{sim-state}(j), w) : w \in s'.\textit{snap-set}(j)\}$. But by the code for $\textit{snap-succeed}_j$, the set $s'.\textit{snap-set}(j)$ is empty. So it must be that $u'.\textit{sim-state}(j) = s'.\textit{sim-state}(j)$.

   Now, the code implies that $s'.\textit{sim-state}(j) = \textit{trans-snap}(s.\textit{sim-state}(j), x)$, which implies that $u'.\textit{sim-state}(j) = \textit{trans-snap}(s.\textit{sim-state}(j), x)$. Therefore, $u'.\textit{sim-state}(j) \in \{s.\textit{sim-state}(j)\} \cup \{\textit{trans-snap}(s.\textit{sim-state}, w) : w \in s.\textit{snap-set}(j)\}$, as needed.

■

This lemma implies that every trace of $\textit{DelayedSpec} \times U$ is a trace of $\textit{SimpleSpec} \times U$ [24], that is (recall the definition of "solves" in Section 3.4.3):

Corollary 6.2 *DelayedSpec solves SimpleSpec.*

6.2 The System $\mathcal{Q}$ with Safe Agreement Modules

Our third and final level is the system $\mathcal{Q}$, composed with arbitrary safe agreement modules, and with the *propose* and *agree* actions reclassified as internal. We show that this system, composed with a user U that submits at most one $init_i$ action on each port, implements $DelayedSpec \times U$ in the sense of trace inclusion; that is, this system solves $DelayedSpec \times U$ (in the sense of Section 3.4.3). The idea is that individual processes of $\mathcal{Q}$ that are simulating a snapshot step of a process j of $\mathcal{P}'$ "try" to perform the simulated snapshot at the point where they take their actual snapshots. At the point where the appropriate safe agreement module chooses the winning actual snapshot, the simulated snapshot "succeeds". As in the *DelayedSpec*, this choice is made after the snapshot attempts.

Formally, we use a weak forward simulation [24]. The word "weak" simply indicates that the proof uses invariants. We need the invariants for the definition as well as for the proof of the forward simulation: strictly speaking, the definition of the forward simulation we use is ambiguous without them.

Lemma 6.3 gives "coherence" invariants, asserting consistency among three things: information kept by the processes of $\mathcal{Q}$, information in the safe agreement modules, and a "run" (as defined just below) of an individual process j of $\mathcal{P}'$. Note that Lemma 6.3 does not talk about global executions of $\mathcal{P}'$, but only about runs of an individual process of $\mathcal{P}'$.

Define a *run* of process j of $\mathcal{P}'$ to be a sequence of the form $\rho = s_0, c_1, s_1, c_2, s_2, \ldots, s_k$, where each s_i is a state of process j, and each c_i is a "change", that is, one of the following: (*"init"*, v), (*"snap"*, w), (*"update"*, r), *"local"*, (*"decide"*, v); the first state is the unique start state, and each change yields a transition from the preceding to the succeeding state.

A consequence of the next lemma is that every process i that simulates steps of a process j simulates the same run of j. As we shall see, the run is determined by the i process that is furthest ahead in the simulation of j; thus, only such an i process can affect the outcome of the next step of j. Moreover, it can affect only the outcome of snapshot steps. Once the outcome of a snapshot step is determined, i can proceed with the simulation of j locally (without reading the shared variable), up to the next snapshot step.

Invariant 1 relates the information in the processes of $\mathcal{Q}$ and the safe agreement modules. Invariants 2 and 3 relate the processes of $\mathcal{Q}$ and a given run ρ of process j. Invariants 4 and 5 relate ρ and the safe agreement modules. Invariant 6 relates all three types of information: it relates information in certain processes of $\mathcal{Q}$, the run ρ (those that are "current" in their simulation of j, according to ρ) and the safe agreement modules.

Lemma 6.3 *For every reachable state of $\mathcal{Q}$ composed with abstract safe agreement modules and a user U, and for each process j, there is a run $\rho = s_0, c_1, s_1, \ldots, s_k$ of process j such that:*

1. *For any i:*
   - *(a)* sim-steps$(j)_i \geq 1$ *if and only if* $i \in$ agreed-procs$_{j,0}$.
   - *(b) For any $\ell \geq 1$,* sim-snaps$(j)_i \geq \ell$ *if and only if* $i \in$ agreed-procs$_{j,\ell}$.
   - *(c)* $i \in$ proposed-procs$_{j,0}$ $-$ agreed-procs$_{j,0}$ *if and only if* nextop(sim-state$(j)_i$) $=$ "init" *and* status$(j)_i \in$ $\{$unsafe, safe$\}$.
   - *(d) For any $\ell \geq 1$, $i \in$* proposed-procs$_{j,\ell}$ $-$ agreed-procs$_{j,\ell}$ *if and only if* nextop(sim-state$(j)_i$) $=$ "snap", sim-snaps$(j)_i = \ell - 1$, *and* status$(j)_i \in \{$unsafe, safe$\}$.
2. $k = \max_i\{$sim-steps$(j)_i\}$.
3. *For any i, if* sim-steps$(j)_i = \ell$ *then:*

  (a) $\text{sim-state}(j)_i = s_\ell$.

  (b) $\text{sim-snaps}(j)_i$ *is the number of* "snap"*'s among* $c_1, \ldots, c_\ell$.

  (c) $\text{mem}(i).\text{sim-mem}(j)$ *is the value written in the last* "update" *among* $c_1, \ldots, c_\ell$*, if any, else* r_0.

  (d) $\text{mem}(i).\text{sim-steps}(j)$ *is the number of the last* "update" *among* $c_1, \ldots, c_\ell$*, if any, else* 0.

4. *(a)* ("init", v) *appears in* ρ *if and only if* $\text{agreed-val}_{j,0} = v$.

  (b) ("snap", w) *is the* ℓ*'th snapshot in* ρ *if and only if* $\text{agreed-val}_{j,\ell} = w$.

5. *If* $\text{proposed-vals}_{j,\ell} \neq \emptyset$ *and* $\text{agreed-val}_{j,\ell} = \perp$ *for some* ℓ *then*

  (a) If $\ell = 0$ *then* ρ *consists of only one state* s*, and* $\text{nextop}(s) =$ "init".

  (b) If $\ell \geq 1$*, then* $\text{nextop}(s_k) =$ "snap"*, and the number of snaps in* ρ *is* $\ell - 1$.

6. *For any* $\ell \geq 1$*, if* $\text{nextop}(s_k) =$ "snap" *and the number of* "snaps" *in* ρ *is* $\ell - 1$*, then* $\text{proposed-vals}_{j,\ell} = \{\text{sim-mem-local}(j)_i : \text{sim-steps}(j)_i = k \text{ and } \text{status}(j)_i \in \{\text{unsafe}, \text{safe}\}\}$.

Proof: Let s be any reachable state of $\mathcal{Q}$ composed with abstract safe agreement modules and a user U. For s equal to the initial state it is simple to check that the lemma holds. Assume it holds for some state s, and we prove that it holds for any state s', after a step (s, π, s'). Let $\rho = s_0, c_1, s_1, \ldots, s_k$ be a run of process j, corresponding to s, whose existence is guaranteed by the lemma. We prove there is a run ρ' corresponding to s', that satisfies the requirements of the lemma. The run ρ' will be either equal to ρ, or else obtained from ρ by appending a change c_{k+1} and a state s_{k+1}. We skip the proof of invariant 1, which is simple and does not talk about ρ.

For state s, $k = max_i\{s.sim\text{-}steps(j)_i\}$. Let k' be the corresponding value in s'; that is $k' = max_i\{s'.sim\text{-}steps(j)_i\}$.

First assume $k' = k + 1$. Then, for some i, π must be one of: $agree(w)_{j,0,i}$, $agree(w)_{j,\ell,i}$ for $\ell \in N^+$, $sim\text{-}update_{j,i}$, $sim\text{-}local_{j,i}$, or $sim\text{-}decide_{j,i}$, since these are the only cases that increment a *sim-steps* component. Moreover, $s.sim\text{-}steps(j)_i = k$, and hence, by part 3(a) of the lemma, $s_k = s.sim\text{-}state(j)_i$. For each one of these possibilities, ρ' is obtained from ρ by appending the corresponding change: (*"init"*, w) for an $agree(w)_{j,0,i}$; (*"snap"*, w) for an $agree(w)_{j,\ell,i}$, $\ell \in N^+$; (*"update"*, r) for a $sim\text{-}update_{j,i}$; *"local"* for a $sim\text{-}local_{j,i}$; (*"decide"*, v) for a $sim\text{-}decide_{j,i}$, and after the change, appending to the run the state s_{k+1}, resulting from the corresponding transition function (*trans-init*, *trans-snap*, or *trans*) applied to s_k. That is, $s_{k+1} = s'.sim\text{-}state(j)_i$. Thus, in s', process i is the first one to finish the simulation of the k'-th step of j and $s'.sim\text{-}steps(j)_i = k'$; while for every other process i', $s'.sim\text{-}steps(j)_{i'} < k'$.

First notice that part 2 of the lemma clearly holds for s'. Consider the case of $\pi = agree(w)_{j,\ell,i}$ for $\ell \in N^+$ (we omit the proofs of the other cases, which are analogous). For part 3 of the lemma, we need to consider only the case of $\ell = k + 1$, since the cases of $\ell < k + 1$ hold by the induction hypothesis. Thus, we need to consider only process i. Part (a) holds by the definition of s_{k+1}. Part (b) holds because $s.sim\text{-}snaps(j)_i$ is the number of *snap*'s among $c_1, \ldots, c_k$, and $s'.sim\text{-}snaps(j)_i = s.sim\text{-}snaps(j)_i + 1$, while $c_{k+1} =$ (*"snap"*, w). Part (c), (d), and part 4(a) of the lemma hold by induction hypothesis. For part 4(b) of the lemma, notice that there are $\ell - 1$ *snap*'s in ρ. Thus, in ρ' there are ℓ *snap*'s, and indeed $agreed\text{-}val_{j,\ell} = w$. Part 5 holds trivially because process i is the first one to finish the simulation of the ℓ-th *snap* of j, and hence $proposed\text{-}vals_{j,\ell'} \neq \emptyset$ and $agreed\text{-}val_{j,\ell'} \neq \perp$ for $\ell' \leq \ell$, while $proposed\text{-}vals_{j,\ell'} = \emptyset$ and $agreed\text{-}val_{j,\ell'} = \perp$ for $\ell' > \ell$. Finally, consider part 6. Since in s' there are no processes i' with $sim\text{-}steps(j)_{i'} = k + 1$ and $status(j)_{i'} \in \{unsafe, safe\}$, then we have to prove that $\mathit{proposed\text{-}vals}_{j,\ell+1} = \emptyset$. Observe that $s.\mathit{sim\text{-}snaps}(j)_{i'} = \ell - 1$ for any i' with $s.\mathit{sim\text{-}steps}(j)_{i'} = k$. Then, $s.\mathit{sim\text{-}snaps}(j)_{i'} < \ell$ for all i', and hence no i' has yet executed a $\mathit{propose}(w)_{j,\ell+1}$.

Now assume $k' = k$. In this case, $\rho' = \rho$. Clearly part 2 of the lemma holds. The cases of π equal to $\mathit{agree}(w)_{j,0,i}$, $\mathit{agree}(w)_{j,\ell,i}$, $\ell \in N^+$, $\mathit{sim\text{-}update}_{j,i}$, $\mathit{sim\text{-}local}_{j,i}$, or $\mathit{sim\text{-}decide}_{j,i}$, are similar to each other. Let us consider the most interesting: $\pi = \mathit{agree}(w)_{j,\ell,i}$. We have that $s.\mathit{sim\text{-}snaps}(j)_i = \ell - 1$ and $s'.\mathit{sim\text{-}snaps}(j)_i = \ell$. Assume $s.\mathit{sim\text{-}steps}(j)_i = k_1$, $k_1 < k$. To prove part 3 take $\ell = k_1 + 1$. Part (a) follows because $s.\mathit{sim\text{-}state}(j)_i = s_{k_1}$, and $w \in \mathit{agreed\text{-}val}_{j,\ell}$, so that the effect of π when *trans-snap* is applied gives $s_{k_1+1} = s'.\mathit{sim\text{-}state}(j)_i$. Part (b) follows because $s.\mathit{sim\text{-}snaps}(j)_i$ is the number of *snap*'s among $c_1, \ldots, c_\ell - 1$, and c_ℓ is a *snap*, and hence $s'.\mathit{sim\text{-}snaps}(j)_i = s.\mathit{sim\text{-}snaps}(j)_i + 1$ is the number of *snap*'s among $c_1, \ldots, c_\ell$. The other parts of the lemma follow easily by induction.

Another case is when π is $\mathit{propose}(v)_{j,0,i}$, or $\mathit{propose}(w)_{j,\ell,i}$, $\ell \in N^+$. Consider the second possibility. To check part 5 of the lemma assume $s'.\mathit{proposed\text{-}vals}_{j,\ell} \neq \emptyset$ and $s'.\mathit{agreed\text{-}val}_{j,\ell} = \perp$, while $s.\mathit{proposed\text{-}vals}_{j,\ell} = \emptyset$ and $s.\mathit{agreed\text{-}val}_{j,\ell} = \perp$. Then, π is the first *propose* for j and ℓ, and hence $k = s.\mathit{sim\text{-}steps}(j)_i$. Also, $s'.\mathit{nextop}(\mathit{sim\text{-}state}(j)_i) = \text{“}\mathit{snap}\text{”}$ because $s.\mathit{status}(j) = \mathit{propose}$. Thus $\mathit{nextop}(s_k) = \text{“}\mathit{snap}\text{”}$. To complete the proof of the claim notice that the number of *snaps* in ρ is $\ell - 1$, by the induction hypothesis for part 3 (a) and (b). Finally, part 6 of the lemma is easy to check because $w = s.\mathit{sim\text{-}mem\text{-}local}(j)_i$ is added to the set $\mathit{proposed\text{-}vals}_{j,\ell}$. ■

The forward simulation relation we use is the relation f from states of Q composed with safe agreement modules and U to states of *DelayedSpec* × U that is defined as follows. If s is a state of the Q system and u is a state of *DelayedSpec* × U, then $(s, u) \in f$ provided that the following all hold:

1. The state of U is the same in u and s.
2. $u.\mathit{sim\text{-}mem} = \mathit{latest}(s.\mathit{mem})$.
3. For every i,
   (a) $u.\mathit{input}(i) = s.\mathit{input}_i$.
   (b) $u.\mathit{reported}(i) = s.\mathit{reported}_i$.
4. For every j,
   (a) $u.\mathit{sim\text{-}state}(j) = s.\mathit{sim\text{-}state}(j)_i$, where i is the index of the maximum value of $s.\mathit{sim\text{-}steps}(j)$.
   (b) If there exists i with $s.\mathit{sim\text{-}decision}(j)_i \neq \perp$ then $u.\mathit{sim\text{-}decision}(j) = s.\mathit{sim\text{-}decision}(j)_i$ for some such i, else $u.\mathit{sim\text{-}decision}(j) = \perp$.
   (c) If $\mathit{nextop}(u.\mathit{sim\text{-}state}(j)) = \text{“}\mathit{snap}\text{”}$ then $u.\mathit{snap\text{-}set}(j) = \{s.\mathit{sim\text{-}mem\text{-}local}(j)_i : s.\mathit{sim\text{-}steps}(j)_i = \max_k\{s.\mathit{sim\text{-}steps}(j)_k\}$ and $s.\mathit{status}(j)_i \neq \mathit{idle}\}$ else $u.\mathit{snap\text{-}set}(j) = \emptyset$.

Thus, the simulated memory $u.\mathit{sim\text{-}mem}$ is determined by the latest information that any of the processes of Q has about the memory, and likewise for the simulated process states and simulated decisions. Also, the snapshot sets $u.\mathit{snap\text{-}set}(j)$ are determined by the snapshot values saved in local process states, in Q.

Each *snap-try* step of *DelayedSpec* is “implemented” by a current *snap* of Q. Each *snap-succeed* step is implemented by the first *agree* step of the appropriate safe agreement module, and likewise for each *sim-init* step. Each *sim-update* step is implemented by the first step at which some process simulates that update, and likewise for the other types of simulated process steps.

Lemma 6.4 *Relation f is a weak forward simulation from $\mathcal{Q}$ composed with safe agreement modules and U to $\mathit{DelayedSpec} \times U$.*

Sketch of proof: Let (s, π, s') be a step of the $\mathcal{Q}$ system, and let u be any state of $\mathit{DelayedSpec} \times U$ such that $(s, u) \in f$. We produce an execution fragment of $\mathit{DelayedSpec} \times U$, from u to a state u', such that $(s', u') \in f$. The proof is by cases, according to π. These are the most interesting cases:

1. $\pi = \mathit{snap}_{j,i}$.

   If $\mathit{sim\text{-}steps}(j)_i$ is the maximum value of $\mathit{sim\text{-}steps}(j)$ (in both s and s'), then this simulates $\mathit{snap\text{-}try}_j$, else it simulates no steps.

   Assume the first case: that $\mathit{sim\text{-}steps}(j)_i$ is the maximum value of $\mathit{sim\text{-}steps}(j)$. The corresponding execution fragment is $(u, \mathit{snap\text{-}try}_j, u')$, where u' is the same as u except that $u'.\mathit{snap\text{-}set}(j) = u.\mathit{snap\text{-}set}(j) \cup \{u.\mathit{sim\text{-}mem}\}$. Since (s, π, s') is a step of $\mathcal{Q}$, the precondition for π holds in s and $\mathit{nextop}(s.\mathit{sim\text{-}state}(j)_i) = \text{“}\mathit{snap}\text{”}$. Since $(s, u) \in f$, it follows that $\mathit{nextop}(u.\mathit{sim\text{-}state}(j)) = \text{“}\mathit{snap}\text{”}$, by 4(a) of the definition of f. Therefore, the precondition for $\mathit{snap\text{-}try}_j$ holds in u, and $(u, \mathit{snap\text{-}try}_j, u')$ is an execution fragment.

   To prove that $(s', u') \in f$, the only nontrivial part of the definition of f to check is 4(c); since $\mathit{nextop}(u'.\mathit{sim\text{-}state}(j)) = \text{“}\mathit{snap}\text{”}$, we do have to verify that u' satisfies part 4(c) of the definition of f. We know that $u.\mathit{snap\text{-}set}(j)$ is equal to the set $\{s.\mathit{sim\text{-}mem\text{-}local}(j)_i : s.\mathit{sim\text{-}steps}(j)_i = \max_k\{s.\mathit{sim\text{-}steps}(j)_k\}$ and $s.\mathit{status}(j)_i \neq \mathit{idle}\}$, because $(s, u) \in f$. Now, $u'.\mathit{snap\text{-}set}(j) = u.\mathit{snap\text{-}set}(j) \cup \{u.\mathit{sim\text{-}mem}\}$. Also, $u.\mathit{sim\text{-}mem} = \mathit{latest}(s.\mathit{mem})$, by part 3 of the definition of f. After the $\mathit{snap}_{j,i}$, we get $\mathit{latest}(s.\mathit{mem}) = s'.\mathit{sim\text{-}mem\text{-}local}(j)_i$. It follows that $u'.\mathit{snap\text{-}set}(j)$ is equal to $u.\mathit{snap\text{-}set}(j) \cup \{s'.\mathit{sim\text{-}mem\text{-}local}(j)_i\}$, and hence, $u'.\mathit{snap\text{-}set}(j)$ is equal to $\{s'.\mathit{sim\text{-}mem\text{-}local}(j)_i : s'.\mathit{sim\text{-}steps}(j)_i = \max_k\{s'.\mathit{sim\text{-}steps}(j)_k\}$ and $s'.\mathit{status}(j)_i \neq \mathit{idle}\}$, as desired.

   The case where $\mathit{sim\text{-}steps}(j)_i$ is not the maximum value of $\mathit{sim\text{-}steps}(j)$ is trivial.

2. $\pi = \mathit{agree}(w)_{j,\ell,i}$, $\ell \in N^+$.

   If this increases the maximum value of $\mathit{sim\text{-}steps}(j)$ then it simulates $\mathit{snap\text{-}succeed}_j$ with a decision value of w, else simulates no steps.

   Consider the case where π increases the maximum value of $\mathit{sim\text{-}steps}(j)$. Let $k = \max_i\{s.\mathit{sim\text{-}steps}(j)_i\}$. Then, $s.\mathit{sim\text{-}steps}(j)_i = k$, and $s'.\mathit{sim\text{-}steps}(j)_i = k + 1$. By Lemma 6.3, for state s, there is a run for j, $\rho = s_0, c_1, s_1, \ldots, s_k$, with $s_k = s.\mathit{sim\text{-}state}(j)_i$. Now, part 1(d) of Lemma 6.3 implies that $\mathit{nextop}(s.\mathit{sim\text{-}state}(j)_i) = \text{“}\mathit{snap}\text{”}$, $s.\mathit{sim\text{-}snaps}(j)_i = \ell - 1$, and $s.\mathit{status}(j)_i \in \{\mathit{unsafe}, \mathit{safe}\}$. Since $(s, u) \in f$, $u.\mathit{sim\text{-}state}(j) = s.\mathit{sim\text{-}state}(j)_i$, and hence, $\mathit{nextop}(u.\mathit{sim\text{-}state}(j)_i) = \text{“}\mathit{snap}\text{”}$. We want to prove that $(u, \mathit{snap\text{-}succeed}_j, u')$ with a decision value of w is an execution fragment. Since we already proved that $\mathit{nextop}(u.\mathit{sim\text{-}state}(j)_i) = \text{“}\mathit{snap}\text{”}$, to prove that the precondition of the $\mathit{snap\text{-}succeed}_j$ holds it remains to show that $w \in u.\mathit{snap\text{-}set}(j)$.

   To prove that $w \in u.\mathit{snap\text{-}set}(j)$, recall that $s.\mathit{sim\text{-}snaps}(j)_i = \ell - 1$, and hence, $\ell - 1$ is the number of “*snap*”'s in ρ, by part 3(b) of Lemma 6.3. Thus, the hypothesis of part 6 of Lemma 6.3 holds, and $s.\mathit{proposed\text{-}vals}_{j,\ell} = \{s.\mathit{sim\text{-}mem\text{-}local}(j)_i : s.\mathit{sim\text{-}steps}(j)_i = k$ and $s.\mathit{status}(j)_i \in \{\mathit{unsafe}, \mathit{safe}\}\}$. We know that w must be in the set $s.\mathit{proposed\text{-}vals}_{j,\ell}$, because $(s, \mathit{agree}(w)_{j,\ell,i}, s')$ is an execution fragment. Thus, $w = s.\mathit{sim\text{-}mem\text{-}local}(j)_{i'}$, for some i' with $s.\mathit{sim\text{-}steps}(j)_{i'} = k$ and $s.\mathit{status}(j)_{i'} \in \{\mathit{unsafe}, \mathit{safe}\}$. To complete the proof of the claim, notice that part 4(c) of the definition of f implies that $u.\mathit{snap\text{-}set}(j) =$

$\{s.\textit{sim-mem-local}(j)_i : s.\textit{sim-steps}(j)_i = \max_k\{s.\textit{sim-steps}(j)_k\}$ and $s.\textit{status}(j)_i \neq \textit{idle}\}$. Therefore, w must be in $u.\textit{snap-set}(j)$.

Finally, it is easy to verify that $(s', u') \in f$: we need only to check conditions 4(a) and 4(c) of the definition of f. Clearly 4(a) holds. For 4(c) observe that $u'.\textit{snap-set}(j) = \emptyset$. If $\textit{nextop}(u'.\textit{sim-state}(j)) \neq$ *"snap"* then 4(c) holds. But if $\textit{nextop}(u'.\textit{sim-state}(j)) =$ *"snap"* 4(c) also holds, since i is the only one achieving the maximum of $\max_k\{s'.\textit{sim-steps}(j)_k\}$, and $s'.\textit{status}(j)_i = \textit{idle}$.

The case where π does not increase the maximum value of $\textit{sim-steps}(j)$ is simple. Here no steps are simulated and $u = u'$. To see that $(s', u') \in f$, we need to check only that parts 4(a) and 4(c) of the definition of f hold. This follows easily from the fact that $(s, u) \in f$, and that the maximum value of $\textit{sim-steps}(j)$ does not change.

■

We conclude that every trace of $\mathcal{Q}$ composed with safe agreement modules and a user U is a trace of *DelayedSpec* $\times U$:

Corollary 6.5 *$\mathcal{Q}$ composed with safe agreement modules solves DelayedSpec.*

Combining Corollaries 6.5 and 6.2, we obtain:

Corollary 6.6 *$\mathcal{Q}$ composed with safe agreement modules solves SimpleSpec.*

Corollary 6.6 is almost, but not quite, what we need. It remains to compose the $\mathcal{Q}$ automaton with snapshot shared memory systems that implement all the safe agreement modules, then to merge all the processes of all these various components systems in order to form a single shared memory system. The resulting system has infinitely many snapshot shared variables; we combine all these to yield a system $\mathcal{P}$ with a single snapshot shared variable. We conclude that for every user U that submits at most one $\textit{init}_i$ action on each port, every trace of $\mathcal{P} \times U$ is a trace of *SimpleSpec* $\times U$. That is,

Lemma 6.7 *$\mathcal{P}$ solves SimpleSpec.*

Lemma 6.7 yields the safety requirements of a fault-tolerant simulation, as expressed by part (1) of Definition 3.2. Now we prove the fault-tolerance requirements, as expressed by part (2) of Definition 3.2. The argument is reasonably straightforward, based on the fact that each process of $\mathcal{Q}$ can, at any time, be in the unsafe region of code for at most one process of $\mathcal{P}'$. As before, since we are reasoning about fault-tolerance, we consider explicit *stop* actions.

Lemma 6.8 *If $\mathcal{P}'$ guarantees f-failure termination then $\mathcal{P}$ guarantees f-failure termination.*

Proof: Assume that $\mathcal{P}'$ guarantees f-failure termination.

Each process i of $\mathcal{P}$ simulates the steps of each process j of $\mathcal{P}'$ in order, waiting for each step to complete before going on to the next one. Process i works concurrently on simulating steps of different processes of $\mathcal{P}'$. However, it is only permitted to be in the "unsafe" portion of its execution for one process j of $\mathcal{P}'$ at a time.

Recall that the specification of safe-agreement stipulates that if a non-failing process i executes a $\textit{propose}_{j,\ell,i}$ action it will get an $\textit{agree}_{j,\ell,i}$ action, unless some other process i', simulating step ℓ of j, fails when "unsafe." In this case i' could block the simulation of j. However, since i' is allowed to participate in this safe agreement only if it is not currently in the "unsafe" portion of any other safe agreement execution, then i' can block at most one simulated process. In any execution in which at most f simulator processes fail, at most f simulated processes are blocked, and each non-failing simulator i can complete the simulation of at least $n' - f$ processes. Therefore, since $\mathcal{P}'$ satisfies f-failure termination, a non-failing simulator will eventually execute its *decide* step. Thus the whole system satisfies f-failure termination. ■

Lemmas 6.7 and 6.8 yield:

Theorem 6.9 *$\mathcal{P}$ is an f-simulation of $\mathcal{P}'$ via relations G and H.*

Now, from Theorem 6.9 and Theorem 3.3 we get the result that leads to the applications in Section 8:

Theorem 6.10 *Suppose that there exists a snapshot shared memory system that solves D' and guarantees f-failure termination, and suppose that $D \leq_f^{G,H} D'$. Then there exists a snapshot shared memory system that solves D and guarantees f-failure termination.*

7 Simulation in Read/Write Systems

A system using snapshot shared memory can be implemented in a wait-free manner in terms of single-writer multi-reader read/write shared variables [1]. It follows that Theorem 6.10 extends to read/write systems. However, in this section we provide a direct construction, showing how to produce a read/write shared memory system $\mathcal{P}$ that f-simulates a read/write shared memory system $\mathcal{P}'$. The read/write simulation algorithm is essentially the same as the snapshot simulation algorithm, except that a snapshot operation is replaced by a sequence of reads in arbitrary order.

The reasons why we presented the snapshot simulation algorithm first are that it is simpler, and that the correctness proof of the read/write simulation algorithm is based on that of the snapshot algorithm.

We assume that the system we want to simulate, $\mathcal{P}'_{RW}$, is an n'-process read/write shared memory system. We describe an n-process read/write simulating system $\mathcal{Q}_{RW}$. As before, this algorithm is assumed to interact with the usual environment, via *init* and *decide* actions, and also with a two-dimensional array of safe agreement modules $A_{j,\ell}$, $j \in \{1,\ldots,n'\}$, $\ell \in N$, $N = \{0,1,2,\ldots\}$. In the complete version of the simulation algorithm, denoted $\mathcal{P}_{RW}$, these safe agreement modules are replaced by read/write memory implementations and the whole thing implemented by a read/write shared memory system.

The simulated system $\mathcal{P}'_{RW}$ has a sequence *mem'* of n' read/write shared variables. Each variable *mem'*(j) is a single-writer multi-reader variable, written by process j of $\mathcal{P}'_{RW}$, taking on values in R, and with initial value r_0. Furthermore, we assume that $\mathcal{P}'$ solves a decision problem D', guaranteeing f-failure termination.

We use terminology about system $\mathcal{P}'_{RW}$ which is similar to that of system $\mathcal{P}'$, as described in Section 3.4.1. Namely, for any state s of a process j of $\mathcal{P}'_{RW}$, define *nextop*(s) to be an element of $\{$"*init*", "*local*"$\} \cup \{($"*read*"$, j') : 1 \leq j' \leq n'\} \cup \{($"*update*"$, r) : r \in R\} \cup \{($"*decide*"$, v) : v \in V\}$. As before, our determinism assumptions imply that each state s has a well defined and unique value of *nextop*(s). For any state s of a process j such that *nextop*$(s) =$ "*init*" and any $v \in V$, define *trans-init*(s, v) to be the state that results from applying *init*$(v)_j$ to s. For any state s of a process j such that *nextop*$(s) = ($"*read*"$, j')$ and any $w \in R$, define *trans-read*(s, w) to be the state that results from performing the read operation of the j'th variable from state s, with the return value for the read being w. Finally, for any state s of a process j such that *nextop*(s) is an "*update*", *"local"*, or *"decide"* pair, define *trans*(s) to be the state of j that results from performing the operation from state s.

The system $\mathcal{Q}_{RW}$ is assumed to interact with each $A_{j,\ell}$ via outputs *propose*$(w)_{j,\ell,i}$ and inputs *safe*$_{j,\ell,i}$ and *agree*$(w)_{j,\ell,i}$. In fact, $\mathcal{Q}_{RW}$ is very similar to $\mathcal{Q}$. The difference is that each snapshot operation used by $\mathcal{Q}$ (the only place snapshots are used is in the computation of *latest*) is replaced by a sequence of read operations in $\mathcal{Q}_{RW}$, as described next.

The shared memory of $\mathcal{Q}_{RW}$ consists of a sequence *mem-RW* of n read/write shared variables. Each variable *mem-RW*(i) is a single-writer multi-reader variable, written by process i of $\mathcal{Q}_{RW}$. In *mem-RW*(i), process i keeps track of the latest values in all the variables of $\mathcal{P}'_{RW}$, according to i's local simulation of $\mathcal{P}'_{RW}$. Along with each such value, *sim-mem*(j), it keeps a tag *sim-steps*(j), which counts the number of steps that it has simulated for j, up to and including the latest step at which process j of $\mathcal{P}'_{RW}$ updated its register.

The code of $\mathcal{Q}_{RW}$ has the same transitions as those of $\mathcal{Q}$, except that the *snap* is replaced by *reading* and *read-done*, and the necessary syntactic modifications are made to the *propose* and *agree* transitions. The formal description appears below. Process i simulates a "read" of variable j' by process j, by reading all the variables in *mem-RW* and combining the information in these variables to produce a single value in R: the value produced is the latest value written by any of the processes of $\mathcal{Q}_{RW}$ in its copy of the shared variable of j'. More precisely, process i executes a series of n *reading*$_{j,i}$ actions in arbitrary order, one for each i', selecting the *mem-RW*(i').*sim-mem*(j') value associated with the highest *mem-RW*(i').*sim-steps*(j') (this value must be unique). In the code below, $m(j)$ keeps track of the highest *mem-RW*(i').*sim-steps*(j') encountered so far. $m(j)$ is initialized to -1, because *mem-RW*(i').*sim-steps*(j') takes values greater or equal than 0. There is also *read-set*(j) which keeps track of the indexes of processes that have been considered. Thus, *read-set*(j) is initially empty. Once the n components of *mem-RW* have been read, *read-set*$(j) = \{1, \ldots, n\}$ and *read-done*$_{j,i}$ can be executed. This in turn allows completion of the simulation of the "read" with the execution of the *propose*$(w)_{j,\ell,i}$ and *agree*$(w)_{j,\ell,i}$ actions.

Simulation System $\mathcal{Q}_{RW}$
Same as $\mathcal{Q}$ but with the following changes:

Shared variables:

```
As in $\mathcal{Q}$ but instead of mem:
mem-RW, a sequence of $n$ read/write variables; for each $i$, mem-RW$(i)$ has components:
    sim-mem, a vector in $R^{n'}$, initially everywhere $r_0$
    sim-steps, a vector in $N^{n'}$, initially everywhere 0
```

Actions of i:

```
Input:              Internal:
    As in $\mathcal{Q}$         As in $\mathcal{Q}$ but instead of snap$_{j,i}$:
Output:                 reading$_{j,i}$
    As in $\mathcal{Q}$         read-done$_{j,i}$
```

States of i:

```
As in $\mathcal{Q}$ except for:
for each $j$,
instead of sim-snaps:
    sim-reads$(j) \in N$, initially 0
instead of sim-mem-local:
    $sim\text{-}mem\text{-}local\text{-}RW \in R$, initially arbitrary
and in addition:
    $read\text{-}set(j)$ a set of integers, initially empty
    $m(j) \in N \cup \{-1\}$, initially $-1$
```

Transitions of i:

As in $\mathcal{Q}$ but instead of $snap_{j,i}$,

```
$reading_{j,i}$
    Precondition:
        $nextop(sim\text{-}state(j)) = (\text{"read"}, j')$
        $status(j) = idle$
        $i' \in \{1, \ldots, n\} - read\text{-}set(j)$
    Effect:
        $read\text{-}set(j) := read\text{-}set(j) \cup \{i'\}$
        if $mem\text{-}RW(i').sim\text{-}steps(j') > m(j)$ then
            $sim\text{-}mem\text{-}local\text{-}RW(j) :=$
                    $mem\text{-}RW(i').sim\text{-}mem(j')$
            $m(j) := mem\text{-}RW(i').sim\text{-}steps(j')$

$read\text{-}done_{j,i}$
    Precondition:
        $nextop(sim\text{-}state(j)) = (\text{"read"}, j')$
        $status(j) = idle$
        $read\text{-}set(j) = \{1, \ldots, n\}$
    Effect:
        $read\text{-}set(j) := \emptyset$
        $m(j) := -1$
        $status(j) := propose$

$propose(w)_{j,\ell,i}$, $\ell \in N^+$
    Precondition:
        $status(j) = propose$
        $\nexists k : status(k) = unsafe$
        $sim\text{-}reads(j) = \ell - 1$
        $w = sim\text{-}mem\text{-}local\text{-}RW(j)$
    Effect:
        $status(j) := unsafe$

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

Tasks of i:

As in $\mathcal{Q}$.

To prove the correctness of the read/write simulation algorithm, we define an intermediate system, *SnapSim*. The only difference between $\mathcal{Q}_{RW}$ and *SnapSim* is that to simulate a read action of the j'th component, *SnapSim* performs a snapshot of *mem-RW* and applies a function $latest_{snp}$ to the result, instead of performing a series of reads. The function $latest_{snp}$ for j' is defined as follows. It returns a single value of R, representing the latest value written by all the processes in the *mem-RW* variable of j'. That is, let $k = \max_{i'}\{mem\text{-}RW(i').sim\text{-}steps(j')\}$, and choose any i'' such that $mem\text{-}RW(i'').sim\text{-}steps(j') = k$. Then $latest_{snp}(mem\text{-}RW, j') = mem\text{-}RW(i'').sim\text{-}mem(j')$. (We claim this is uniquely defined.) In the code of *SnapSim* the *reading* and *read-done* transitions are replaced by a *read* transition:

Simulation System *SnapSim*.:

Shared variables:

As in $\mathcal{Q}_{RW}$

Actions of i:

Input:
As in $\mathcal{Q}_{RW}$
Output:
As in $\mathcal{Q}_{RW}$

Internal:
As in $\mathcal{Q}_{RW}$, except that $\mathit{reading}_{j,i}$ and $\mathit{read\text{-}done}_{j,i}$ are replaced by $\mathit{read}_{j,i}$

States of i:

As in $\mathcal{Q}_{RW}$

Transitions of i:

As in $\mathcal{Q}_{RW}$, except that $\mathit{reading}_{j,i}$ and $\mathit{read\text{-}done}_{j,i}$ are replaced by $\mathit{read}_{j,i}$:

```
$\mathit{read}_{j,i}$
  Precondition:
    $\mathit{nextop}(\mathit{sim\text{-}state}(j)) = (\text{“}\mathit{read}\text{”}, j')$
    $\mathit{status}(j) = \mathit{idle}$
  Effect:
    $\mathit{sim\text{-}mem\text{-}local\text{-}RW}(j) := \mathit{latest}_{snp}(\mathit{mem\text{-}RW}, j')$
    $\mathit{status}(j) := \mathit{propose}$
```

Tasks of i:

As in $\mathcal{Q}_{RW}$.

It is not hard to verify that an execution of $\mathcal{Q}_{RW}$ corresponds to an execution of *SnapSim*: Consider a $\mathit{read\text{-}done}_{j,i}$ and the corresponding $\mathit{reading}_{j,i}$'s, for some fixed values j, i. Thus the precondition $\mathit{nextop}(\mathit{sim\text{-}state}(j)) = (\text{“}\mathit{read}\text{”}, j')$ holds for some particular j'”; fix j'. Also, $\mathit{sim\text{-}reads}(j) = \ell - 1$ for some value of ℓ. Thus, for the rest of the argument, we have fixed values of ℓ, i, j, j'.

Replace all of these $\mathit{read\text{-}done}_{j,i}$ and $\mathit{reading}_{j,i}$'s by a single $\mathit{read}_{j,i}$, which occurs somewhere between the first $\mathit{reading}_{j,i}$ and the last $\mathit{reading}_{j,i}$, at a point when the highest $\mathit{sim\text{-}steps}(j')$ takes the value recorded by the $\mathit{read\text{-}done}_{j,i}$. That is, the *read* is placed at a point where $\max_{i'}\{\mathit{mem\text{-}RW}(i').\mathit{sim\text{-}steps}(j')\}$ is equal to the value of $m(j)$ at the point of the *read-done*. Such a point exists because the *sim-steps* variables increase by one unit at a time, and because the final value of $m(j)$ satisfies the following: it is at least the value of $\max_{i'}\{\mathit{mem\text{-}RW}(i').\mathit{sim\text{-}steps}(j')\}$ at the moment of the first $\mathit{reading}_{j,i}$, and at most the value of $\max_{i'}\{\mathit{mem\text{-}RW}(i').\mathit{sim\text{-}steps}(j')\}$ at the moment of the last $\mathit{reading}_{j,i}$.

Note that the value of $\mathit{sim\text{-}mem\text{-}local\text{-}RW}(j)$ at the point of the *read-done* (which is the value returned by the sequence of *reading* steps in $\mathcal{Q}_{RW}$) is the same as the value of $\mathit{mem\text{-}RW}(i'').\mathit{sim\text{-}mem}(j')$ at the point where the *read* is placed, for any i'' with $\mathit{mem\text{-}RW}(i'').\mathit{sim\text{-}steps}(j') = \max_{i'}\{\mathit{mem\text{-}RW}(i').\mathit{sim\text{-}steps}(j')\}$.

It follows that every trace of $\mathcal{Q}_{RW}$ with safe-agreement modules and U is also a trace of *SnapSim* with safe-agreement modules and U. Now, the same proof technique that we used to proof that every trace of $\mathcal{Q}$ with safe-agreement modules and U is a trace of $\mathit{DelayedSpec} \times U$ can also be used to prove that every trace of *SnapSim* with safe-agreement modules and U is a trace of $\mathit{DelayedSpec}_{RW} \times U$, where $\mathit{DelayedSpec}_{RW}$ is the read/write memory version of *DelayedSpec*. Also, the proof technique used for Corollary 6.2 can be used to prove that every trace of $\mathit{DelayedSpec}_{RW} \times U$ is a trace of $\mathit{SimpleSpec}_{RW} \times U$, the read/write memory version of *SimpleSpec*. Combining all these facts, we see that every trace of $\mathcal{Q}_{RW}$ with safe-agreement modules and U is also a trace of $\mathit{SimpleSpec}_{RM} \times U$. Therefore:

Lemma 7.1 *$\mathcal{Q}_{RW}$ composed with safe agreement modules solves $\mathit{SimpleSpec}_{RW}$.*

As before, we compose $\mathcal{Q}_{RW}$ with read/write shared memory systems that implement all the safe agreement modules, and then merge all the processes of all these various components systems in order to form a single shared memory system, $\mathcal{P}_{RW}$. We see that, for every user U that submits at most one init_i action on each port, every trace of $\mathcal{P}_{RW} \times U$ is a trace of $\mathit{SimpleSpec}_{RW} \times U$. That is:

Lemma 7.2 *$\mathcal{P}_{RW}$ solves $\mathit{SimpleSpec}_{RW}$.*

The fault-tolerance argument is analogous to the one for snapshot shared memory systems:

Lemma 7.3 *If $\mathcal{P}'_{RW}$ guarantees f-failure termination then $\mathcal{P}_{RW}$ guarantees f-failure termination.*

Now Lemmas 7.2 and 7.3 yield (restating Definition 3.2, the definition of f-simulation, in terms of $\mathit{SimpleSpec}_{RW}$):

Theorem 7.4 *$\mathcal{P}_{RW}$ is an f-simulation of $\mathcal{P}'_{RW}$ via relations G and H.*

And we get the analogue of Theorem 6.10 (using the analogue of Theorem 3.3 for read/write systems):

Theorem 7.5 *Suppose that there exists a read/write shared memory system that solves D' and guarantees f-failure termination, and suppose that $D \leq_f^{G,H} D'$. Then there exists a read/write shared memory system that solves D and guarantees f-failure termination.*

8 Applications

In Section 8.1, we describe the notion of a *convergence task* [16], which is used to specify a family of decision problems, one for each number of processes. For example, binary consensus is a convergence task – it yields a decision problem for any number of processes. In Theorem 8.1, we show that one decision problem in the family of problems specified by a convergence task is solvable if and only if any other problem in the family is solvable. The proof is based on Theorem 6.10.

In Section 8.2 we use this theorem to obtain various possibility and impossibility results for read/write and snapshot shared memory systems.

8.1 Convergence Tasks

In Section 3.1 we defined an n-port decision problem in terms of two sets of n-vectors, $\mathcal{I}$ and $\mathcal{O}$, and a total relation Δ from $\mathcal{I}$ to $\mathcal{O}$. Thus, a decision problem is specified for a certain number of processes, n. For the applications in the next subsection, we would like to talk about a "problem" in general, without specifying the number of processes. For example, in the binary consensus problem, any number of processes start with binary inputs, and have to agree on some process' input value. Strictly speaking, this is not a decision problem, but a family of decision problems, one for each n.

In principle, one could define a family of decision problems, in a way that for two different values of n, the corresponding decision problems are completely unrelated. But this is not what one would mean by a "family." We now describe a way of defining a family of decision problems called convergence tasks [16]. We prove that it is a "family" in the sense, roughly, that one decision problem in the family is solvable if and only if any other is.

For defining convergence tasks, it will be convenient to talk about sets instead of vectors, since the position of an element in the vector will be immaterial. That is, in the kind of decision problems we will be considering, any permutation of an input (output) vector will also be an input (output) vector. We call a set a *simplex,* to follow the notation of topology. An element of a simplex is a *vertex.* A *complex* is a family of simplexes closed under containment.[1]

For a complex $\mathcal{K}$, $\mathit{skel}^k(\mathcal{K})$ denotes the subcomplex formed by all simplexes of $\mathcal{K}$ of size at most $k+1$. For example, $\mathit{skel}^0(\mathcal{K})$ consists of all the vertices of $\mathcal{K}$, and $\mathit{skel}^1(\mathcal{K})$ consists of all the vertices and all the simplexes of size two. Thus $\mathit{skel}^1(\mathcal{K})$ can be thought of as a graph, with simplexes of size 2 as edges and simplexes of size 1 as vertices.

Informally, if S is an input simplex of a convergence task, each process can receive as input value any vertex of S, such that the input values are a subset of S (two processes may receive the same vertex). The convergence task specifies a set of legal output simplexes for S, denoted $\Psi(S)$. Each process has to choose an output a vertex (two processes may choose the same vertex), such that the vertices form an output simplex of $\Psi(S)$. Let $n\text{-}\mathit{vectors}(S)$ be the set of n-vectors of values from S. Thus, if S is an input simplex, then $n\text{-}\mathit{vectors}(S)$ are input vectors, and if L is an output simplex then $n\text{-}\mathit{vectors}(L)$ are output vectors.

Let $\mathcal{K}$ be a complex. The corresponding n-port vector set $\widetilde{\mathcal{K}}_n$ is defined as follows. $\langle \vec{v}_1, \ldots, \vec{v}_n \rangle$ is a vector in $\widetilde{\mathcal{K}}_n$ if and only if $\vec{v}_1, \ldots, \vec{v}_n$ (not necessarily distinct) form a simplex in $\mathcal{K}$; that is, $\widetilde{\mathcal{K}}_n = \cup_{S \in \mathcal{K}}\, n\text{-}\mathit{vectors}(S)$. For a vector w, let $\mathit{set}(w)$ be the simplex of values of w. Thus, if $w \in \widetilde{\mathcal{K}}_n$ then $\mathit{set}(w) \in \mathcal{K}$.

Formally, a *convergence task* $[\mathcal{L}, \mathcal{K}, \Psi]$ consists of two arbitrary complexes, $\mathcal{L}$ and $\mathcal{K}$, called the *input complex* and the *output complex*, respectively, and a relation Ψ carrying each simplex of $\mathcal{L}$ to a non-empty subcomplex of $\mathcal{K}$, such that if L_0 is a face of L_1, then $\Psi(L_0) \subseteq \Psi(L_1)$.

For each n, the *n-port decision problem of* $[\mathcal{L}, \mathcal{K}, \Psi]$ is $\langle \widetilde{\mathcal{L}}_n, \widetilde{\mathcal{K}}_n, \widetilde{\Psi} \rangle$, where $\widetilde{\Psi}$ is as follows: $\widetilde{\Psi}(w)$ contains every n-vector w' such that $w' \in n\text{-}\mathit{vectors}(S)$, for $S \in \Psi(\mathit{set}(w))$.

In the next subsection, we consider the following convergence tasks.

1. The *N-consensus convergence task* is $[\mathcal{S}^{N-1}, \mathit{skel}^0(\mathcal{S}^{N-1}), \mathit{skel}^0]$, where $\mathcal{S}^{N-1}$ consists of a simplex of size N, $N > 1$, and its subsimplexes. Thus, for each n, it yields a consensus decision problem [11] for n processes, where the processes start with N possible input values, which are the vertices of $\mathcal{S}^{N-1}$. If the processes start with values that form an input simplex $S \in \mathcal{S}^{N-1}$, they have to decide values that form a simplex in $\mathit{skel}^0(S)$. Since the only simplexes of $\mathit{skel}^0(S)$ are the vertices of S, the processes have to decide on the same vertex, that is, they all have to agree on one of the input vertices of S.

2. The *(N,k)-set agreement convergence task,* $0 < k < N$, is $[\mathcal{S}^{N-1}, \mathit{skel}^{k-1}(\mathcal{S}^{N-1}), \mathit{skel}^{k-1}]$. Thus, for each n, it yields an n-process k-set-agreement problem over a set $\mathcal{S}^{N-1}$ of N values (see Example 1).

---

[1] Thus the complexes we consider here are "colorless," as opposed to the colored complexes considered usually in the topology approach to distributed computing (e.g. [7, 18, 15]), where each element of a simplex has associated a process id.

3. The *loop agreement convergence task* [16] is $[\mathcal{S}^2, \mathcal{K}, \Lambda]$, where $\mathcal{S}^2$ is the 2-simplex $(\vec{s}_0, \vec{s}_1, \vec{s}_2)$ and its subsimplexes, $\mathcal{K}$ is an arbitrary finite complex with three distinguished vertices $\vec{v}_0, \vec{v}_1, \vec{v}_2$, $\Lambda(\vec{s}_i) = \vec{v}_i$, $\Lambda(\vec{s}_i, \vec{s}_j)$ is some path (simplexes of size 1 and 2) λ_{ij} with end-points $\vec{v}_i$ and $\vec{v}_j$, and $\Lambda(\mathcal{S}^2) = \mathcal{K}$.

Other examples of convergence tasks appear in [16], like uncolored simplex agreement, barycentric agreement, and ϵ-agreement.

Theorem 8.1 *For a convergence task* $[\mathcal{L}, \mathcal{K}, \Psi]$, *let* $D = \langle \mathcal{I}, \mathcal{O}, \Delta \rangle$ *be the corresponding* n*-port decision problem,* $D' = \langle \mathcal{I}', \mathcal{O}', \Delta' \rangle$ *the* n'*-port decision problem, and* $f < \min\{n, n'\}$. *If there exists a snapshot shared memory system that solves* D *and guarantees* f*-failure termination then there exists a snapshot shared memory system that solves* D' *and guarantees* f*-failure termination.*

Proof: By Theorem 6.10, it suffices to show that $D \leq_f^{G,H} D'$, for some $G = G(g_1, g_2, \ldots, g_n)$ and $H = H(f, h_1, h_2, \ldots, h_n)$. Define $g_i(v)$ to be the n'-vector with all entries equal to v, and $h_i(w)$ to be any of the elements of w different from $\perp$.

Now we prove the requirement $G \cdot \Delta' \cdot F \cdot H \subseteq \Delta$ of Definition 3.1. Take any input vector $w \in \mathcal{I}$. Thus $set(w) \in \mathcal{L}$. For any $w_1 \in G(w)$,

$$set(w_1) \subseteq set(w), \tag{1}$$

and hence, $set(w_1) \in \mathcal{L}$, since $\mathcal{L}$ is closed under containment. That is, $w_1 \in \mathcal{I}'$.

Now, take any $w_2 \in \Delta'(w_1)$. Thus $set(w_2) \in \Psi(set(w_1))$. By definition of H and F, any $w_3 \in H(F(w_2))$ satisfies $set(w_3) \subseteq set(w_2)$. Thus, $set(w_3) \in \Psi(set(w_1))$, since $set(w_2) \in \Psi(set(w_1))$ and $\Psi(set(w_1))$ is (a complex) closed under containment.

Finally, we need to prove that $set(w_3) \in \Psi(set(w))$, since this implies that $w_3 \in \Delta(w)$. This holds because $\Psi(set(w_1)) \subseteq \Psi(set(w))$, by Equation 1. ■

Applying Theorem 7.5 (instead of Theorem 6.10), we get the same result for read/write systems.

8.2 Possibility and Impossibility Results

Theorem 8.1 can be used to extend results that are known for a small number of processes to larger numbers, for fixed f. In this section we present several applications of this kind. All the applications we present hold for read/write memory systems and for snapshot memory systems, since one can use the read/write memory or the snapshot memory version of Theorem 8.1.

Consensus. It is known [11, 20] that the consensus decision problem is not solvable with f-failure termination, when $f \geq 1$. In particular, wait-free 2-process consensus is unsolvable [13]. It is possible to use only this particular result, and Theorem 8.1 to prove the following:

Corollary 8.2 *The consensus problem is not solvable for* $f \geq 1$.

Set Agreement. It is known from [5, 26, 18] that the (n, k)-set agreement problem is not wait-free solvable. This result together with Theorem 8.1 implies:

Corollary 8.3 *There is no algorithm that solves the* (n, k)*-set agreement problem with* f*-failure termination if* $f \geq k$.

Computability. It is known [12] that the problem of telling if a decision problem for n processes, $n \geq 3$, has a wait-free solution is not computable (i.e., is undecidable). This was proved[2] in [16] by showing that the following problem is not computable: Given a loop agreement convergence task, tell if the n-port corresponding decision problem has a wait-free solution. This result, together with Theorem 8.1, implies the following:

---

[2] In fact, in [16], the result of Corollary 8.4 is proved directly, and in more general models of shared memory.

Corollary 8.4 *Let $2 \leq f < n$. The problem of telling if an n-port loop agreement decision problem has a solution with f-failure termination is not computable.*

Also, when $f = 1$, it was proved in [4] that the problem of telling if an arbitrary decision problem has solution with f-failure termination is computable. In particular, the problem is computable for any 2-port decision problem obtained from a convergence task. It is possible to use only this particular result, and Theorem 8.1, to prove the following:

Corollary 8.5 *The problem of telling if an n-port decision problem corresponding to a convergence task $\mathcal{T}$ has a solution with 1-failure termination is computable.*

Notice that the results in [4] apply to general decision problems, while this corollary is about decision problems produced by convergence tasks. Also, we stress that Corollary 8.5 follows from the results of [4]. The point here is that Corollary 8.5 can be proved by showing only the computability for 2-port, decision problems; a problem conceivably easier than to prove it directly for arbitrary n.

9 Discussion

We have presented the beginnings of a method to translate results in one distributed system model to another. We have introduced a general way of simulating a distributed algorithm of n processes and f fault-tolerance, by a distributed system with a different number of processes and the same fault-tolerance. We have presented a precise description of this fault-tolerant simulation algorithm, a careful description of what it accomplishes, as well as a proof of correctness.

Specifically, we have defined a notion of *fault-tolerant reducibility* between decision problems, and showed that the algorithm implements this reducibility. The reducibility is tailored to the simulation algorithm; it should not be used as a general notion of reducibility between decision problems. An important moral of this work is that one must be careful in applying the simulation algorithm– it does not work for all pairs of problems, but only for those that satisfy the reducibility. Nevertheless, we have shown that the simulation algorithm is a powerful tool for obtaining possibility and impossibility results.

Similarly, we have presented a specification of what it means for one shared memory system to simulate another, in a fault-tolerant manner. Again, this specification is intended to capture the type of simulation that is studied in this paper. We have given a full and detailed description of a version of the simulation algorithm for snapshot memory systems. We have proved that this algorithm satisfies the requirements of a fault-tolerant simulation.

We have also shown how to extend this basic snapshot memory simulation algorithm to read/write shared memory, and hence, have shown that it is useful for proving properties of these systems as well. We have first presented the snapshot algorithm and then the read/write variant due to the fact that in the snapshot model, the proof is more modular, and the whole presentation clearer.

We have presented several applications of the simulation algorithm to a class of problems that satisfy the reducibility, including consensus and set agreement, defined by convergence tasks [16]. The applications extend results about a system with some number of processes and f failures, to a system with any number of processes and the same number of failures. Further applications are described in [7].

Some possible variations on the simulation algorithm of this paper are: (a) Allow each process i of $\mathcal{Q}$ to simulate only a (statically determined) subset of the processes of $\mathcal{P}'$ rather than all the processes of $\mathcal{P}'$. (b) Allow more complicated rules for determining the simulated inputs of $\mathcal{P}'$ and the actual outputs of $\mathcal{Q}$; these rules can include f-fault-tolerant distributed protocols among the processes of $\mathcal{Q}$.

We hope that one of the greatest contributions of this paper will be in laying the foundation for the development of an interesting variety of extensions to the simulation algorithm. One extension is proposed in [6, 7], and later formalized (following our techniques) in [10, 25], where the processes of $\mathcal{Q}$ simulate a system $\mathcal{P}'$ that has access to set agreement variables. Other variants of the simulation, for consensus problems in systems with access to general shared objects appear in [9] and in [21].

Reducibilities between problems have proved to be useful elsewhere in computer science (e.g., in recursive function theory and complexity theory of sequential algorithms), for classifying problems according to their solvability and computational complexity. One would expect that reducibilities would also be useful in distributed computing theory, for example, for classifying decision problems according to their solvability in fault-prone asynchronous systems. Our reducibility appears somewhat too specially tailored to the simulation algorithm presented to serve as a useful general notion. Further research is needed to determine the limitations of this reducibility and to define a more general-purpose notion.

Stronger notions of reducibility (or fault-tolerant simulation) might include a closer, "step-by-step" correspondence between the execution of the simulating system $\mathcal{P}$ and the simulated system $\mathcal{P}'$. Such a stronger notion seems to be needed to obtain results [7] relating the topological structure of the executions of $\mathcal{P}$ and $\mathcal{P}'$. These results seem to indicate that the simulation plays an interesting role in the newly emerging topology approach to distributed computing (e.g. [7, 18, 15]).

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt and N. Shavit, "Atomic snapshots of shared memory," *Journal of the ACM,* Vol. 40, No. 4, September 1993, 873–890.

[2] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rudiger Reischuk, "Renaming in an asynchronous environment," *Journal of the ACM*, Vol. 37, No. 3, July 1990, 524–548.

[3] Yehuda Afek, Gideon Stupp, "Synchronization power depends on the register size," (Preliminary Version), *Proc. of the IEEE Symposium on Foundations of Computer Science* (FOCS), 1993, 196–205.

[4] O. Biran, S. Moran, S. Zaks, "A combinatorial characterization of the distributed 1-solvable tasks," *Journal of Algorithms*, vol. 11, 1990, 420–440.

[5] E. Borowsky and E. Gafni, "Generalized FLP impossibility result for t-resilient asynchronous computations," in *Proceedings of the 1993 ACM Symposium on Theory of Computing*, May 1993, 91–100.

[6] E. Borowsky and E. Gafni, "The implication of the Borowsky-Gafni simulation on the set consensus hierarchy," Technical Report 930021, UCLA Computer Science Dept., 1993.

[7] E. Borowsky, "Capturing the power of resiliency and set consensus in distributed systems," Ph.D. Thesis, University of California, Los Angeles, October 15, 1995.

[8] S. Chaudhuri, "More *choices* allow more *faults:* set consensus problems in totally asynchronous systems," *Information and Computation,* Vol. 105, 1993, 132–158.

[9] T. Chandra, V. Hadzilacos, P. Jayanti, S. Toueg, "Wait-freedom vs. t-resiliency and the robustness of wait-free hierarchies," in *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, August 1994, 334–343.

[10] S. Chaudhuri, P. Reiners, "Understanding the set consensus partial order using the Borowsky-Gafni simulation," 10th International Workshop on Distributed Algorithms, Oct. 9–11, 1996. Lecture Notes in Computer Science 1151, Springer-Verlag, 362–379.

[11] M.J. Fischer, N.A. Lynch, M.S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM,* Vol. 32, No. 2, April 1985, 374–382.

[12] E. Gafni and E. Koutsoupias, "3-processor tasks are undecidable," brief announcement in *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, August 1995, p. 271. Full version submitted for publication.

[13] M.P. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

[14] M.P. Herlihy and S. Rajsbaum, "Set consensus using arbitrary objects," 13th ACM Symposium on Principles of Distributed Computing (PODC '94), Aug. 14–17, Los Angeles, 1994, pp. 324–333.

[15] M.P. Herlihy and S. Rajsbaum, "A Primer on Algebraic Topology and Distributed Computing," in *Computer Science Today,* Jan van Leeuwen (Ed.), LNCS Vol. 1000, Springer Verlag, 1995, p. 203–217.

[16] M.P. Herlihy and S. Rajsbaum, "On the decidability of distributed decision tasks," 29th ACM Symp. on the Theory of Computation (STOC), May 1997, p. 589–598. Brief Announcement in 15th ACM Symposium on Principles of Distributed Computing (PODC), 1996, p. 279.

[17] M.P. Herlihy and E. Ruppert, "On the existence of booster types," in Proceedings of the 41st IEEE Symposium on Foundations of Computer Science (FOCS), 2000.

[18] M.P. Herlihy and N. Shavit, "The asynchronous computability theorem for t-resilient tasks," In *Proceedings of the 1993 ACM Symposium on Theory of Computing*, May 1993, 111–120.

[19] N.A. Lynch, *Distributed Algorithms,* Morgan Kaufmann Publishers, Inc. 1996.

[20] M.C. Loui and H.H. Abu-Amara, "Memory requirements for agreement among unreliable asynchronous processes," in F. P. Preparata (ed.), *Parallel and Distributed Computing,* vol. 4 of *Advances in Computing Research,* 163–183. JAI Press, Greenwich, Conn., 1987.

[21] W. Lo and V. Hadzilacos, "On the power of shared object types to implement one-resilient consensus," in *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 101–110, August 1997.

[22] N.A. Lynch and S. Rajsbaum, "On the Borowsky-Gafni Simulation Algorithm," In *Proceedings of the Fourth Israel Symposium on Theory of Computing and Systems*, June 1996, 4–15.

[23] N.A. Lynch, M.R. Tuttle, "An Introduction to input/output automata," *CWI-Quarterly,* Vol. 2, No. 3, September 1989, 219–246. Centrum voor Wiskunde en Informatica, Amsterdam. Also TM-373, MIT Laboratory for Computer Science, November 1988.

[24] Nancy Lynch and Frits Vaandrager. "Forward and Backward Simulations – Part I: Untimed Systems," *Information and Computation*, Vol. 121, No. 2, September 1995, 214–233.

[25] P. Reiners, "Understanding the Set Consensus Partial Order using the Borowsky-Gafni Simulation," M.S. Thesis, Iowa State University, 1996.

[26] M. Saks and F. Zaharoglou, "Wait-free k-set agreement is impossible: The topology of public knowledge," In *Proceedings of the 1993 ACM Symposium on Theory of Computing*, May 1993, 101–110.