On the Borowsky-Gafni Simulation Algorithm
(Extended Abstract)

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

Abstract

The first precise description of a version of the Borowsky-Gafni fault-tolerant simulation algorithm is given, along with a careful description of what it accomplishes and a proof of correctness. The algorithm implements a notion of fault-tolerant reducibility *between decision problems. This notion of reducibility is defined, and examples of its use are provided.*

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.*

1 Introduction

Consider a read/write asynchronous shared memory system. In [3], Borowsky and Gafni describe an algorithm that allows a set of $f+1$ processes, any f of which may exhibit stopping failures, to "simulate" a larger number n of processes, also with at most f failures. In the n-process k-set-agreement problem [6], all n processes propose values and decide on at most k of the proposed values. The simulation algorithm is used in [3] 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 [3, 8, 12], this transformation implies that there is no k-fault-tolerant solution to the n-process k-set-agreement problem, for any n. Other applications of the simulation algorithm appear in [4] and in [5].

These initial examples suggest that the Borowsky-Gafni simulation can become a powerful tool for proving solvability and unsolvability results for fault-prone asynchronous systems. However, in order for this to happen, it must be clear exactly what the simulation guarantees. Borowsky and Gafni's presentation is brief and informal, and does not include a careful specification of what the algorithm provides to its users. In fact, the emphasis in [3] is mainly on the application to the k-set-agreement problem rather than on the simulation itself.

We began this research with the modest aim of stating and proving precise correctness guarantees for the Borowsky-Gafni simulation algorithm, using the I/O automaton model and standard proof techniques (invariants, simulation relations, composition, etc.). However, the job turned out to be more difficult than we expected, because the description in [3] left some ambiguities that we needed to resolve.[1] The final product of our work is a complete and careful description of a version of the Borowsky-Gafni simulation algorithm, plus a careful description of what it accomplishes, plus a proof of correctness.

In order to specify what the simulation accomplishes, we define a notion of *fault-tolerant reducibility* between decision problems, and show that the algorithm implements this reducibility, in a precise sense. Although this notion

*Supported by Air Force Contract AFOSR F49620-92-J-0125, NSF contract 9225124CCR, 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] For example, the read/write shared memory model that we believe is intended in [3] turns out not to provide enough coherence among the data read by different processes. In order to rectify this, we switched to an atomic snapshot memory model, which can be done without loss of generality.

of reducibility is quite natural, it is specially tailored to the Borowsky-Gafni simulation algorithm; it does not seem suitable as a general notion of reducibility between decision problems. We give some examples of pairs of decision problems that do and do not satisfy this 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 $f \leq \min\{n, n'\}$. (The particular case where $n = k+1$ and $f = k = k'$ was used in [3].) On the other hand, these problems are not reducible if $k \leq f < k'$. Moreover, we only know trivial instances of the renaming problem that satisfy the reducibility. The moral is that one must be careful in applying the simulation – it does not work for all pairs of problems, but only those that satisfy the reducibility.

We present and verify the algorithm in terms of I/O automata [10]. The presentation has a great deal of interesting modularity, expressed by I/O automaton composition and both forward and backward simulation relations (see [11], for example, for definitions). Composition is used to include a *safe agreement* module, a simplification of one in [3], 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.

Some of the formal descriptions and proofs are omitted from this extended abstract.

2 The Model

The underlying model is the I/O automaton model of Lynch and Tuttle [10], as described, for example, in Chapter 8 of [9]. Briefly, an I/O automaton is a simple 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.

Most of the systems in this paper are *asynchronous shared memory* systems, as defined, for example, in Chapter 9 of [9]. 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 [9], 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. To model process stopping failures, we stipulate that each process i in the system has a $stop_i$ input action whose effect is to disable all future non-input actions involving that process.

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.

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.

3 Decision Problems and Fault-Tolerant Reducibility

First we define decision problems and say what it means for a system to solve a decision problem (e.g. [8]). Then we define the fault-tolerant reducibility between decision problems.

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 G from Y to Z, $R \cdot G$ denotes relational composition.

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

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 *init*$(v)_i$ and outputs *decide*$(v)_i$, where $v \in V$ and $1 \leq i \leq n$. We consider A in conjunction with any user automaton U that submits at most one *init*$_i$ on each port i. 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 U, the combination of A and U guarantees well-formedness and correct answers. In addition, we consider a resilience condition:

f-failure termination: In any fair execution of A with 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 *wait-free termination* provided that it guarantees n-failure termination (or, equivalently, $n-1$-failure termination).

A 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 $D' = \langle \mathcal{I}', \mathcal{O}', \Delta' \rangle$, where $0 \leq f \leq n'$.

The reducibility is motivated by the way the Borowsky-Gafni simulation operates. In that simulation, 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 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 form a vector in $\mathcal{O}$, according to the H relation. 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 $0 \leq f \leq n$, $views_f(w)$ denotes the set of length n vectors over $V \cup \{\perp\}$ that are obtained by changing at most f of the components of w to $\perp$. If W is a set of length n vectors, then $views_f(W)$ denotes $\cup_{w \in W}\{views_f(w)\}$.

Our reducibility is defined in terms of three auxiliary relations:

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 Borowsky-Gafni simulation, the examples indicate situations where the simulation can and cannot be used.

Example 1: (n, k)-set agreement is f-reducible to (n', k')-set agreement for $k \geq k'$, $f < \min\{n, n'\}$.

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 2: (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 6.9, together with the fact that (n', k')-set agreement is solvable when $f < k'$ [6], would imply the existence of an f-fault-tolerant algorithm to solve (n, k)-set-agreement. But this contradicts the results of [3, 7, 8, 12].

Example 3: $(2, 2)$-renaming (2-process renaming to a 2-valued name space) is 1-reducible to (n', n')-renaming, for any $n' \geq 2$. However, we cannot (yet) say anything about other cases of the renaming problem.

4 A Safe Agreement Module

The simulation protocol 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 *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 *propose*$_j$ occurs and *safe*$_j$ does not occur, then for any i, if a *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. The following is an informal description of a simple design (using snapshot shared memory) for a safe agreement module. The formal description in the Appendix is a simplification of the one in [3].

The snapshot shared memory contains a *val* component and a *level* component for each process i. When process i receives a *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 *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 *level* $= 1$. When this happens, the set of processes with *level* $= 2$ is nonempty. Let v be the *val* value of the process with the smallest index with *level* $= 2$. Then process i performs an *agree*$(v)_i$ output.

5 The Basic Borowsky-Gafni Simulation Algorithm

In this section, we present the basic algorithm as an n-process snapshot shared memory system $\mathcal{Q}$ with a single snapshot object. 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 object.

We assume that the simulated system, $\mathcal{P}'$, is an n'-process snapshot shared memory system. 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', guaranteeing f-failure termination.

We make some simplifying assumptions about $\mathcal{P}'$, without loss of generality: We assume that for each process, there is only one initial state, only one task, and, 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 any state s of a process j of $\mathcal{P}'$, define *nextop*(s) to be an element of $\{$*"init"*, *"snap"*, *"local"*$\} \cup \{($*"update"*$, r) : r \in R\} \cup \{($*"decide"*$, v) : v \in V\}$. 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*$(s) =$ *"snap"* and any $w \in R^{n'}$, define *trans-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 *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 $\mathcal{Q}$ 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 $\mathcal{P}'$, by 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.

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.

The shared memory of $\mathcal{Q}$ is a single snapshot variable *mem*, contains a portion *mem*(i) for each process i of $\mathcal{Q}$. In its component, process i keeps track of the latest values in all the registers of $\mathcal{P}'$, according to i's local simulation of $\mathcal{P}'$. 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 $\mathcal{P}'$ updated its register.

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 of 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 $\in V \cup \{$*null*$\}$, initially *null*
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* $\in R^{n'}$, initially arbitrary
- *sim-decision*$(j) \in V \cup \{$*null*$\}$, initially *null*

Transitions of i:

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

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

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

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

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

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

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

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

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

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

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

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

A function *latest* is used 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. This function operates "pointwise" for each j, selecting the *sim-mem*(j) value associated with the highest *sim-steps*(j), which must be unique.

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

The specification of safe-agreement stipulates that if a non-failing process i executes a $propose_{j,l,i}$ action it will get an $agree_{j,l,i}$ 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, 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. That is, i' can block at most one simulated process. In any execution with at most f simulators failing, 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.

6 Correctness Proof

In our proofs of safety properties for the main simulation algorithm, we would like to 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:

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

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

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

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

For the safety part of the proof, we use three levels of abstraction, related by forward and backward simulation relations. Informally, forward and backward simulation relations are techniques to show that one I/O automaton implements another [11]; they have nothing to do with "simulations" in the sense of the Borowsky-Gafni simulation algorithm.

6.1. The *SimpleSpec* Automaton

Our highest level is expressed by the *SimpleSpec* automaton, which 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 complication is the way of choosing the inputs for the $\mathcal{P}'$ processes and the outputs for the $\mathcal{Q}$ processes, using the 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 $g_i(input(i))$. At any time after at least $n' - f$ of the j processes have produced decision values, outputs can be produced, using the functions h_i.

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

SimpleSpec:

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

sim-init$_j$
  Precondition:
    nextop(*sim-state*$(j)) =$ *"init"*
    for some i
      input$(i) \neq$ *null*
      $v = g_i($*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:
    reported$(i) =$ *false*
    w is a "subvector" of *sim-decision*
    $|w| \geq n' - f$
    $v = h_i(w)$
  Effect:
    reported$(i) :=$ *true*

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.

Lemma 6.1 *If $\mathcal{P}'$ solves D' and $D \leq_f^{G,H} D'$, then* SimpleSpec *solves D.*

6.2. The *DelayedSpec* Automaton

Our second level is the *DelayedSpec* automaton. This is a slight modification of *SimpleSpec*, which replaces each snapshot step of a process j of $\mathcal{P}'$ (*sim-snap*$_j$) with a series of *snap-try*$_j$ steps during which snapshots are taken and their values recorded, followed by one *snap-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 *snap-set*(j), which keeps track of the set of snapshot vectors that result from doing *snap-try*$_j$ actions. The *sim-snap* actions are omitted. The new actions are:

---

snap-try$_j$
  Precondition:
    nextop(*sim-state*$(j)) =$ *"snap"*
  Effect:
    snap-set$(j) :=$ *snap-set*$(j) \cup \{$*sim-mem*$\}$

snap-succeed$_j$
  Precondition:
    nextop(*sim-state*$(j)) =$ *"snap"*
    $w \in$ *snap-set*(j)
  Effect:
    sim-state$(j) :=$ *trans-snap*(*sim-state*$(j), w)$
    snap-set$(j) := \emptyset$

It should not be hard to believe that *DelayedSpec* "implements" *SimpleSpec*, in the sense of trace inclusion – 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. (The "trace" of an execution is just the sequence of external actions occurring in that execution. Here, the external actions are just the *init* and *decide* actions) 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 is the relation b from states of *DelayedSpec* to states of *SimpleSpec* that is defined as follows. If s is a state of *DelayedSpec* and u is a state of *SimpleSpec*, then $(s, u) \in b$ provided that the following all hold:

1. u.*sim-mem* $= s$.*sim-mem*.
2. For each i,
   (a) u.*input*$(i) = s$.*input*(i).
   (b) u.*reported*$(i) = s$.*reported*(i).
3. For each j,

(a) $u.\textit{sim-state}(j) \in \{s.\textit{sim-state}(j)\} \cup \{\textit{trans-snap}(s.\textit{sim-state}(j), w) : w \in s.\textit{snap-set}(j)\}$.

(b) $u.\textit{sim-decision}(j) = s.\textit{sim-decision}(j)$.

That is, all state components are the same in u and s, with the sole exception that $u.\textit{sim-state}(j) \in \{s.\textit{sim-state}(j)\} \cup \{\textit{trans-snap}(s.\textit{sim-state}(j), w) : w \in s.\textit{snap-set}(j)\}$, i.e., $u.\textit{sim-state}(j)$ 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 6.2 *Relation* b *is a backward simulation from* DelayedSpec *to* SimpleSpec.

Sketch of proof: Let (s, π, s') be a step of *DelayedSpec*, and let $(s', u') \in f$. We produce a corresponding execution fragment of *SimpleSpec*, 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'. Then we know that, either $u'.\textit{sim-state}(j) = s'.\textit{sim-state}(j)$, or $u'.\textit{sim-state}(j) \in \{\textit{trans-snap}(s'.\textit{sim-state}(j), w) : w \in s'.\textit{snap-set}(j), w \neq x\}$. We need to know that $u'.\textit{sim-state}(j)$ is in the set $\{s.\textit{sim-state}(j)\} \cup \{\textit{trans-snap}(s.\textit{sim-state}(j), w) : w \in s.\textit{snap-set}(j)\}$. But this follows easily from the facts that $s.\textit{sim-state}(j) = s'.\textit{sim-state}(j)$, $s.\textit{snap-set}(j) \supseteq s'.\textit{snap-set}(j) - \{x\}$, and that $u'.\textit{sim-state}(j) \neq \textit{trans-snap}(s'.\textit{simstate}(j), x)$.

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, $u'.\textit{sim-state}(j) \in \{s.\textit{sim-state}(j)\} \cup \{\textit{trans-snap}(s.\textit{sim-state}, w) : w \in s.\textit{snap-set}\}$, as needed.

■

Corollary 6.3 *Every trace of* DelayedSpec *is a trace of* SimpleSpec.

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

Finally, our third 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 "implements" *DelayedSpec* in the sense of trace inclusion. 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. The word "weak" simply indicates that the proof uses invariants. We need the invariants for the specification as well as for the proof of the forward simulation. Strictly speaking, the definition of the forward simulation we use is ambiguous without the invariants.

The basic invariant is a "coherence" invariant. It does require us to talk about a run, but not of the entire system. We only look at an execution of each process j of $\mathcal{P}'$ (with initial values and snapshot values arriving from the safe-agreement protocols). All we claim is that it is an execution of the individual process of $\mathcal{P}'$. That is not too difficult to see just by simple following of the code. The argument about a global execution of the entire system $\mathcal{P}'$ falls into the simulation proof in the next section, rather than an invariant.

Express a *run* of process j in $\mathcal{P}'$ in the form $s_0, c_1, s_1, c_2, s_2, \ldots, s_k$, where each s_i is a state of process j, and each c_i is a "change", i.e., one of the following: $(\textit{"init"}, v)$, $(\textit{"snap"}, w)$, $(\textit{"update"}, r)$, *"local"*, $(\textit{"decide"}, v)$; the first state is the unique start state, and all the changes are correct.

The key coherence invariant is the following. The invariant is asserting consistency among three things: the run of an individual process, ρ, the information kept by the i processes, and the information in the safe agreement modules. Invariant 1 relates the information of the processes and of the safe agreement, and therefore, it does not involve ρ. Invariants 2 and 3 relate the process information and ρ, and the last two invariants relate ρ and safe-agreement.

Lemma 6.4 *In every reachable state of $\mathcal{Q}$ with abstract safe agreement modules, for each 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) $\text{sim-steps}(j)_i \geq 1$ *if and only if* $i \in \text{agreed-procs}_{j,0}$.
   (b) $\text{sim-snaps}(j)_i \geq \ell$ *if and only if* $i \in \text{agreed-procs}_{j,\ell}$.
   (c) $i \in \text{proposed-procs}_{j,0} - \text{agreed-procs}_{j,0}$ *if and only if* $\text{nextop}(\text{sim-state}(j)_i) = \text{"init"}$ *and* $\text{status}(j)_i \in \{\text{unsafe}, \text{safe}\}$.
   (d) *For any* $\ell \geq 1$, $i \in \text{proposed-procs}_{j,\ell} - \text{agreed-procs}_{j,\ell}$ *if and only if* $\text{nextop}(\text{sim-state}(j)_i) = \text{"snap"}$, $\text{sim-snaps}(j)_i = \ell - 1$, *and* $\text{status}(j)_i \in \{\text{unsafe}, \text{safe}\}$.
2. $k = \max_i\{\text{sim-steps}(j)_i\}$.
3. *For any i, if* $\text{sim-steps}(j)_i = \ell$ *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) $(\text{"init"}, v)$ *appears in ρ if and only if* $\text{agreed-val}_{j,0} = v$.
   (b) $(\text{"snap"}, w)$ *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* $\text{agreed-val}_{j,\ell} = \text{null}$ *then*
   (a) *If $\ell = 0$ then ρ consists of only one state s, and* $\text{nextop}(s) = \text{"init"}$.
   (b) *If $\ell \geq 1$, then* $\text{nextop}(s) = \text{"snap"}$*, where s is the final state of ρ, and the number of snaps in ρ is $\ell - 1$.*

We need a consistency invariant relating the *sim-mem-local* values for "current" processes to the *proposed-vals*:

Lemma 6.5 *Let* $k = \max_i\{\text{sim-steps}(j)_i\}$*, and i' an index achieving the maximum. If* $\text{nextop}(\text{sim-state}(j)_{i'}) = \text{"snap"}$ *then* $\text{proposed-vals}_{j,\ell} = \{\text{sim-mem-local}(j)_i : \text{sim-steps}(j)_i = k$ *and* $\text{status}(j)_i \in \{\text{safe}, \text{unsafe}\}\}$*, where* $\ell - 1 = \text{sim-snaps}(j)_{i'}$.

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

1. $u.\mathit{sim\text{-}mem} = \mathit{latest}(s.\mathit{mem})$.
2. For every i,
   (a) $u.\mathit{input}(i) = s.\mathit{input}_i$.
   (b) $u.\mathit{reported}(i) = s.\mathit{reported}_i$.
3. 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 $s.\mathit{sim\text{-}decision}(j)_i \neq \mathit{null}$ then $u.\mathit{sim\text{-}decision}(j) = s.\mathit{sim\text{-}decision}(j)_i$ for some such i, else $u.\mathit{sim\text{-}decision}(j) = \mathit{null}$.
   (c) If $\mathit{nextop}(u.\mathit{sim\text{-}state}(j)) = \text{"}\mathit{snap}\text{"}$ then $u.\mathit{snap\text{-}set}(j) = \{s.\mathit{sim\text{-}mem\text{-}local}_i(j) : s.\mathit{sim\text{-}steps}(j)_i = \max_k\{s.\mathit{sim\text{-}steps}(j)_k\}$ 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 $\mathcal{Q}$ have 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 $\mathcal{Q}$. Each *snap-try* step of *DelayedSpec* is "implemented" by a current *snap* of $\mathcal{Q}$. 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 6.6 *Relation f is a weak forward simulation from $\mathcal{Q}$ composed with safe agreement modules to* DelayedSpec*.*

Sketch of proof: Let (s, π, s') be a step of $\mathcal{Q}$, and let u be any state of *DelayedSpec* such that $(s, u) \in f$. We produce an execution fragment of *DelayedSpec*, from u to a state u', $(s', u') \in f$. The proof is by cases, according to what π is. 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 simulates no steps.

   Formally, since (s, π, s') is a step of $\mathcal{Q}$, then the precondition for π holds in s and $\mathit{nextop}(s.\mathit{sim\text{-}state}(j)_i) = \text{"}\mathit{snap}\text{"}$. In the first case, since $(s, u) \in f$, then $\mathit{nextop}(u.\mathit{sim\text{-}state}(j)) = \text{"}\mathit{snap}\text{"}$, by 3(a) of the definition of f. Therefore, the precondition for $\mathit{snap\text{-}try}_j$ holds in u, and the corresponding execution fragment is $(u, \mathit{snap\text{-}try}_j, u')$, where u' is the same as u except that $u'.\mathit{snap\text{-}set}(j) = u.\mathit{snap\text{-}set}(j) \cup \{u.\mathit{sim\text{-}mem}\}$, is well defined.

To prove that $(s', u') \in f$, the only non trivial part of the definition of f to check is 3(c). Since *nextop*$(u'.$*sim-state*$(j))$ = *"snap"*, we do have to verify that u' satisfies part 3(c) of the definition of f. This follows from the fact that $(s, u) \in f$. Thus $u.$*snap-set*(j) is equal to the set $\{s.\textit{sim-mem-local}(j)_i : s.\textit{sim-steps}(j)_i = \max_k\{s.\textit{sim-steps}(j)_k\}$ and $s.\textit{status}(j)_i \neq \textit{idle}\}$. Since $s'.\textit{status}(j)_i \neq \textit{idle}$, we do have to add the value $s'.\textit{sim-mem-local}(j)_i$ to the set $u'.\textit{snap-set}(j)$. Which is done because $s'.\textit{sim-mem-local}(j)_i = \textit{latest}(\textit{mem})$, from the effect of $\textit{snap}_{j,i}$, and $u.\textit{sim-mem} = \textit{latest}(s.\textit{mem})$, since $(s, u) \in f$, and part 1 of the definition of f.

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

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

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

   Formally, in the first case, $\max_t\{s'.\textit{sim-steps}(j)_t\}$ is achieved only by $s'.\textit{sim-steps}(j)_i$, and hence $s.\textit{sim-steps}(j)_i = \max_t\{s.\textit{sim-steps}(j)_t\}$. By Lemma 6.4(2) for state s, there is a run for j, $\rho = s_0, \ldots, s_k$, such that $k = \max_t\{s.\textit{sim-steps}(j)_t\}$. Thus $s.\textit{sim-state}(j)_i = s_k$, by part(3.a) of the Lemma. By well-formedness $s.\textit{sim-snaps}(j)_i = \ell - 1$. Thus, by Lemma 6.4(3.b), $\ell - 1$ is the number of *snap*'s in ρ. By Lemma 6.4(4.b), $s.\textit{agreed-val}_{j,\ell} = \textit{null}$. Since $\textit{proposed-vals}_{j,\ell} \neq \emptyset$, Lemma 6.4(5.b) implies that $\textit{nextop}(s_k)$ = *"snap"*. Thus $\textit{nextop}(s.\textit{sim-state}(j)_i)$ = *"snap"*.

   We conclude that $\textit{nextop}(u.\textit{sim-state}(j))$ = *"snap"*, since $(s, u) \in f$. We need to show that $w \in u.\textit{snap-set}(j)$. First notice that $w \in \textit{proposed-vals}_{j,\ell}$ from the specification of safe-agreement. By part 3(c) of the definition of f and the fact that $(s, u) \in f$, $w \in u.\textit{snap-set}(j)$ if $w = s.\textit{sim-mem-local}(j)_t$ for some t with $s.\textit{sim-steps}(j)_t = s.\textit{sim-steps}(j)_i$, and $s.\textit{status}(j)_t \neq \textit{idle}$. This follows from Lemma 6.5 because $w \in \textit{proposed-vals}_{j,\ell}$. The corresponding execution fragment is $(u, \textit{snap-succeed}_j, u')$, where u' is the same as u except that $u'.\textit{sim-state}(j) = \textit{trans-snap}(u.\textit{sim-state}(j), w) = s'.\textit{sim-state}(j)_i$ and $u'.\textit{snap-set}(j) = \emptyset$, which is well defined because the preconditions of $\textit{snap-succeed}_j$ hold in u.

   Finally, it is easy to verify that $(s', u') \in f$: we need only to check conditions 3(a) and 3(c) of the definition of f. Clearly 3(a) holds. For 3(c) observe that $u'.\textit{snap-set}(j) = \emptyset$. If $\textit{nextop}(u'.\textit{sim-state}(j)) \neq$ *"snap"* then 3(c) holds. But if $\textit{nextop}(u'.\textit{sim-state}(j))$ = *"snap"* 3(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 3(a) and 3(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.

■

Corollary 6.7 *Every trace of $\mathcal{Q}$ with safe agreement modules is a trace of* DelayedSpec*; therefore, every trace of $\mathcal{Q}$ with safe agreement modules is a trace of* SimpleSpec*.*

Corollary 6.7 and Lemma 6.1 yield the safety requirements. The liveness argument is then reasonably straightforward, based on the fact that each process of $\mathcal{Q}$ can be in the unsafe region of code for at most one process of $\mathcal{P}'$.

Theorem 6.8 *Suppose that $\mathcal{P}'$ solves D' and guarantees f-failure termination, and suppose that $D \leq_f^{G,H} D'$. Then $\mathcal{Q}$ with abstract safe agreement modules solves D and guarantees f-failure termination.*

Theorem 6.9 *Suppose that $\mathcal{P}'$ solves D' and guarantees f-failure termination, and suppose that $D \leq_f^{G,H} D'$. Then $\mathcal{P}$ solves D and guarantees f-failure termination.*

By translations back-and-forth between read/write shared memory and snapshot shared memory, we also obtain the same result for read/write shared memory systems.

7 Discussion

We have presented a precise description of a version of the Borowsky-Gafni fault-tolerant simulation algorithm, plus a careful description of what it accomplishes, plus a proof of correctness. In particular, we have defined a notion of *fault-tolerant reducibility* between decision problems, and showed that the algorithm implements this reducibility. The reducibility is specific to the simulation algorithm; it is not intended 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 – it does not work for all pairs of problems, but only for those that satisfy the reducibility.

Our formulation of the Borowsky-Gafni simulation algorithm is not the most general version possible. Some interesting extensions that appear simple 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 $\mathcal{Q}$. More sophisticated extensions are required in applications in [4, 5], where algorithms with access to set consensus objects need to be simulated[2].

We believe that an important contribution of this paper is providing the basis for the development of an interesting variety of extensions to the Borowsky-Gafni simulation algorithm.

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, e.g., for classifying decision problems according to their solvability in fault-prone asynchronous systems. Our reducibility appears somewhat too specially tailored to the Borowsky-Gafni simulation algorithm to serve as a useful general notion of reducibility. Further research is needed to determine the limitations of this reducibility, and to define a more general-purpose notion of reducibility between decision problems. This latter does not seem to be an easy problem.

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] 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.

[4] 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.

[5] E. Borowsky and E. Gafni, "A Computability Theorem for t-Resilient Computation Using Any Set Consensus Objects," manuscript, November 30, 1994.

[2]It is argued in [4] that this type of extension of the basic (read/write) Borowsky-Gafni simulation algorithm is simple, by using a variant of the safe-agreement algorithm.

[6] S. Chaudhuri, "Agreement is harder than consensus: set consensus problems in totally asynchronous systems," In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, August 1990, 311–234.

[7] 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.

[8] 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.

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

[10] N.A. Lynch, M.R. Tuttle, "An Introduction to input/output automata," TM-373, MIT Laboratory for Computer Science, November 1988.

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

[12] 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.

A A Safe Agreement Module Implementation

In the following code, we do not explicitly represent the $stop_i$ actions. We use the convention 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:
level $\in \{0,1,2\}$, initially 0
val $\in V \cup \{null\}$, initially *null*

Actions of i:

Input:	Internal:
$propose(v)_i$, $v \in V$	$update1_i$
Output:	$snap1_i$
$safe_i$	$update2_i$
$agree(v)_i$	$wait_i$

States of i:

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

update2$_i$
  Precondition:
    status = *update2*
  Effect:
    if $\exists j$: *x-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 A.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, i.e., 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. ■