Faithful Simulation of Randomized BFT Protocols on Block DAGs

HAGIT ATTIYA, Technion, Israel
CONSTANTIN ENEA, Ecole Polytechnique, France and CNRS, France
SHAFIK NASSAR, Technion, Israel

Byzantine Fault-Tolerant (BFT) protocols that are based on *Directed Acyclic Graphs* (DAGs) are attractive due to their many advantages in asynchronous blockchain systems. Many DAG-based BFT protocols rely on randomization, since they are used for agreement and ordering of transaction, which cannot be achieved deterministically in asynchronous systems. Randomization is achieved either through local sources of randomness, or by employing shared objects that provide a common source of randomness, e.g., *common coins*.

This paper shows how to simulate DAG-based BFT protocols that use public coins and shared objects, like common coins. Our simulation is *faithful* in the sense that it precisely preserves the safety and liveness properties of the original BFT protocol, and in particular, their probability distributions.

1 INTRODUCTION

Asynchronous distributed computation is naturally captured by a *directed acyclic graph* (DAG), whose nodes describe local computation and edges correspond to causal dependency between computation at different processes. Lamport's *happens-before* relation [11] is an example of such DAG, where each node is a single local computation event, and each edge is a single message delivery event. *Block* DAGs [20] go one step further and incorporate more than one local computation step in each block (node); these steps may even belong to several *independent* protocols.

By exchanging blocks in a manner that preserves their dependencies, a distributed protocol can now be abstracted as a joint computation of a block DAG. In particular, a general *Byzantine fault-tolerant* (BFT) DAG-based algorithm combines two components: one component builds the DAG using a communication protocol that tolerates malicious failures, and the other component performs the local computation embodied in each node of the DAG. The first component can be used to separate the task of injecting user input to the system, such as transactions, from the task of processing these inputs and producing an output, e.g., an ordering of those transactions.

This generality makes block DAGs an attractive approach for designing coordination protocols for, e.g., Byzantine Atomic Broadcast [8, 10, 19], consensus [2, 14] and cryptocurrencies [5]. A block DAG can be seen as a strict extension of a *blockchain*, which is a DAG where all blocks are *totally ordered*, i.e., a directed path. The DAG approach was shown to achieve high throughput [18] due to the flexibility it provides over the standard blockchain approach.

Schett and Danezis [16] have shown that any *deterministic* BFT protocol can be simulated as a block DAG. They provide generic mechanisms for processes to maintain a consistent view of the block DAG, and for each process to *interpret* that DAG as an execution of some protocol. Because several protocols can be interpreted on the same block DAG, this provides a powerful ability to run multiple protocols, or multiple instances of the same protocol, at the same cost.

The restriction to deterministic protocols, however, handicaps the applicability of this result. DAG-based agreement protocols with provable security, like Aleph [8] or DAG-Rider [10], are either randomized or assume the existence of a shared source of randomness. (This is not surprising since consensus cannot be *deterministically* solved in an asynchronous system, when even a single process may fail by crashing [7].) More generally, practical DAG-based algorithms like Narwhal [6] can be

Authors' addresses: Hagit Attiya, Technion, Department of Computer Science, Israel, hagit@cs.technion.ac.il; Constantin Enea, Ecole Polytechnique, LIX, France and CNRS, France, cenea@lix.polytechnique.fr; Shafik Nassar, Technion, Department of Computer Science, Israel, shafiknassar@campus.technion.ac.il.

composed with other "base" consensus protocols to achieve the performance of the DAG-based approach while inheriting liveness and safety from the base protocol.

This calls for a framework that can handle *randomized* BFT protocols; those that either utilize local randomness or even a shared object.

1.1 Our Results

This paper takes upon the task of developing a general approach for DAG-based execution of BFT protocols, even with randomization and shared objects. Specifically, we consider *randomized* BFT protocols in which the local coin flips of each process may be public, we call those protocols *public-coin* protocols. We prove that any public-coin protocol that uses shared objects, e.g., common coins, can be simulated on a block DAG, preserving its usage of shared objects.

It is crucial to show that this simulation is *faithful*, i.e., it preserves properties of the original protocol, including security and probabilistic properties, for example expected termination time. To achieve this, the DAG-based protocol is modelled as a *labeled transition systems* (LTS), and shown to be a *forward* simulation [12] of the original protocol(s). This implies that it precisely preserves the trace distribution and the probabilistic relationship between inputs and outputs. From a security point of view, this means that whatever adverse effects can occur in the simulation, can already be demonstrated in the original protocol.

We believe that our result can help design future DAG-based protocols, by providing an expressive and systematic framework to analyze such protocols. We demonstrate this utility by presenting a high level analysis of existing DAG-based protocols using our result.

1.2 Related Work

Our results can be viewed as a generalization of the work of Schett and Danezis [16]. They show how block DAGs can be used to simulate deterministic protocols, which are a special case of the protocols that we handle here. Readers that are familiar with their work will notice that we were able to achieve a simulation that is a natural extension of theirs. We emphasize, however, that our techniques for proving the faithfulness of our simulation are novel and different from theirs. This is because their techniques do not capture the probabilistic guarantees of randomized protocols. Additional details appear in Section 5.1.

Several recent works have adopted the block DAG approach, e.g., Aleph [8], DAG-Rider [10] and Bullshark [19]. All of these protocols are randomized. Many DAG-based practical systems have been deployed, e.g., ByteBall [5], Flare [14] and Hashgraph [2]. For a survey of the techniques used in block DAG approaches, see [20]. While each of these works presents a new protocol, we provide a formal and systematic framework for analyzing DAG-based protocols, especially *randomized* block DAG protocols. We discuss how our framework can be used to analyze Aleph and DAG-Rider in Section 5.2.

1.3 Organization

Section 2 describes the model and introduces some definitions and notations. Section 3 formally defines block DAGs. Our results are presented and proved in Section 4. Applications of our simulation and its relation to [16] appear in Section 5, and we summarize with future work, in Section 6.

2 PRELIMINARIES

For any $n \in \mathbb{N}$, we denote $[n] = \{1, \ldots, n\}$. For any two strings s_1 and s_2, we denote by $s_1 \circ s_2$ the concatenation of the two strings.

2.1 Model

We consider an asynchronous network with n processes $p_1, \ldots, p_n$. Each process p_i has a local process state PS_i, and buffers $In_{j\to i}$ and $Out_{i\to j}$, for each $j \in [n]$, that serve for communicating with p_j, as well as a buffer $Rqsts_i$ that contains incoming user requests. A schedule consists of two types of events:

- A compute(i) event lets process p_i pass all the messages in the buffers $In_{j\to i}$, as well as the requests in $Rqsts_i$, to PS_i. Then a local computation is performed which may update PS_i, deposit new messages in $Out_{i\to j}$ and return indications to the user.
- A deliver(i, j) event moves the oldest message in $Out_{i\to j}$ to $In_{j\to i}$.

We assume a computationally bounded adversary that may adaptively corrupt up to f processes, and also controls the scheduling of the system. Initially, all n processes are *correct* and honestly follow the protocol. Once a process is corrupted, it may behave arbitrarily. The adversary can also read all messages in the system, even those sent by correct processes. Although the scheduling of message delivery is adversarial, we assume eventual delivery, i.e., every message sent is eventually delivered. We also assume that messages are delivered in the order they were sent.

2.2 Public-Coin Randomized Protocols

In a randomized protocol, the local computation of a process can depend on the result of local coin flips. To model this, we assume each process p_i has access to a random *tape*, from which it can draw a random string at each compute(i) event. Our simulation can be applied to *public-coin* protocols, which are randomized protocols that do no require processes to keep secrets, i.e., they can broadcast the random string they draw as soon as they use it. This definition captures protocols in the full-information model such as [9].

2.3 Shared Objects

To allow for easy composition, we define *shared objects*. A shared object is an implementation of an interface that is accessible by all processes. For example, the interface can be that of a *common coin* and the implementation can be a protocol that implements it. For reference, we formally define the interface of a perfect common coin:

Definition 1. *An f-resilient* perfect common coin *can be invoked using a* Toss(k) *request for any $k \in \mathbb{N}$ and returns an indication associated with that k, which satisfies the following properties if the adversary can corrupt up to f processes:*

(1) (Correctness) *If two correct processes p_i, p_j invoke* Toss(k) *then they both get the same indication. Furthermore, for any $w \in \{0,1\}^{\ell}$ it holds that the indication is equal to w with probability $2^{-\ell}$.*
(2) (Termination) *If all correct processes invoke* Toss(k) *then they all get an indication. Furthermore, if some correct process gets an indication, then every correct process that invokes* Toss(k) *eventually gets an indication.*
(3) (Unpredictability) *No Byzantine process can obtain the indication associated with k before at least one correct process invokes* Toss(k).

For any shared object o, each process p_i can invoke o as it performs any local computation. We assume that PS_i explicitly states the object invocations and their parameters. Invocations are non-blocking, and o may at any point return an indication in a designated buffer o.$buff_i$. Whenever a compute(i) event is scheduled, the contents of o.$buff_i$ are passed to PS_i and may affect the local computation.

2.4 Labeled Transition System

In this work, we model protocols as Labeled Transition Systems (LTSs), defined as follows:

Definition 2. *A* labeled transition system *is a tuple* $L = (Q, \Sigma, q_{start}, \delta)$ *where:*

1. *Q is a (possibly infinite) set of states.*
2. *Σ is a set of (transition) labels.*
3. *q_{start} is the starting state.*
4. *$\delta \subseteq Q \times \Sigma \times Q$ is a (possibly infinite) set of transitions, written as $q_1 \xrightarrow{l} q_2$ for any $(q_1, l, q_2) \in Q \times \Sigma \times Q$.*

An execution of L is an alternating sequence of states and transition labels $\alpha = q_0, l_0, q_1, l_1, \ldots$ such that for any $i \geq 0$ it holds that $q_i \xrightarrow{l_i} q_{i+1}$. If there exists any partial execution $q_i, l_i, \ldots, l_{j-1}, q_j$ then we write $q_i \xrightarrow{l_i, \ldots, l_{j-1}} q_j$. We define a subset of labels $\Sigma_E \subseteq \Sigma$ as the *external actions*, and define a *trace* of L to be the projection of an execution over Σ_E. Typically, external actions correspond to requests and indications in the interface of a protocol, and define the "observable" behavior of a protocol.

LTSs as defined above can be used to model deterministic protocols in a straightforward manner. Essentially, LTS states correspond to tuples of states of participating processes and communication channels, and each transition corresponds to a step of some process (more details are given below).

Randomized protocols can be modeled using an extension of LTSs called *(simple) probabilistic automata* [17] where a transition from a state q leads to a probability distribution over states instead of a single state. The semantics of a probabilistic automaton is formalized in terms of *probabilistic executions*, which are probability distributions over executions defined by a deterministic scheduler that resolves the non-determinism. The deterministic scheduler corresponds to the notion of adversary described above which controls message delivery and process scheduling. Since we consider protocols where the random choices follow the uniform distribution, we simplify the formalization and model them using LTSs instead of probabilistic automata by including results of random choices in the transition labels. Also, the transition labels corresponding to random choices are defined as external actions. The relevance of this modeling choice will be detailed later when discussing forward simulations.

2.5 Modeling the Protocol as an LTS

Let $\mathcal{P}$ be a public-coin protocol and O be a *set* of shared objects used by $\mathcal{P}$. We define the LTS of $\mathcal{P}$ as follows $L = (Q, \Sigma, q_{start}, \delta)$. A state $q \in Q$ consists of the local state PS_i, the incoming messages $(In_{j\to i})_{j\in[n]}$, the outgoing messages $(Out_{i\to j})_{j\in[n]}$ and the incoming object indications $(\mathrm{o}.buff_i)_{\mathrm{o}\in O}$ of each process p_i. For convenience, we assume that incoming user requests are stored in $In_{i\to i}$ and outgoing user indications are stored in $Out_{i\to i}$. Overall, $q = \big(PS_i, (In_{j\to i})_{j\in[n]}, (Out_{i\to j})_{j\in[n]}, (\mathrm{o}.buff_i)_{\mathrm{o}\in O}\big)_{i\in[n]}$. We use register notation to refer to the components of each state, e.g., $q.In_{j\to i}$ refers to the incoming messages buffer from j to i in the state q. In the initial state q_{start}, all of the processes have the initial local state and all of the message buffers are empty.

The transition labels Σ correspond to the different types of steps in a protocol execution, namely, local computation, message delivery, indications from objects in O, or user requests and indications. Observe that we do not need to label sending requests to $\mathrm{o} \in O$ as this is done in an ordinary local computation event. In addition, the local computation label would include the randomness (if any) that is used by the process in the said computation event. Formally, the labels in Σ are as follows:

(1) $\mathsf{compute}(i, \rho)$ denotes a transition where process p_i performs a local computation with ρ as its randomness.
(2) $\mathsf{deliver}(i \longrightarrow j)$ denotes a transition where all of the messages in $Out_{i\rightarrow j}$ are moved to $In_{i\rightarrow j}$.
(3) $\mathsf{o.indicate}(i, w)$ denotes a transition where the value w has been added to $\mathsf{o}.buff_i$.
(4) $\mathsf{request}(i, x)$ denotes a transition where process p_i receives x as input.
(5) $\mathsf{indicate}(i, y)$ denotes a transition where process p_i returns y as output.

The external actions in $\Sigma_E \subseteq \Sigma$ are user requests ($\mathsf{request}(i, x)$) and indications ($\mathsf{indicate}(i, y)$), and local computation events ($\mathsf{compute}(i, \rho)$). The latter are included in Σ_E in order to be able to relate probability distributions in different protocols, as discussed hereafter. A transition $(q_1, l, q_2) \in Q \times \Sigma \times Q$ is in δ if and only if the protocol can get from state q_1 to state q_2 by executing the step denoted by the label l.

Forward Simulations. Showing that a block DAG protocol is a simulation of some other protocol relies on the notion of *forward simulation* between the LTSs modeling the two protocols, respectively.

Definition 3. *Let $L = (Q, \Sigma, q_{start}, \delta)$ and $L' = (Q', \Sigma', q'_{start}, \delta')$ be two LTSs with the same set of external actions Σ_E. A relation $R \subseteq Q \times Q'$ is a forward simulation from L to L' if both of the following hold:*

- $(q_{start}, q'_{start}) \in R$
- *For any $(q_1, l, q_2) \in \delta$ and any q'_1 such that $(q_1, q'_1) \in R$, there exists $q'_2 \in Q'$ such that:*
  - $(q_2, q'_2) \in R$,
  - *$q'_1 \xrightarrow{\sigma} q'_2$ is a partial execution of L' (σ is a sequence of labels in Σ'), and*
  - *if $l \in \Sigma_E$, then the projection of the label sequence σ over Σ_E is exactly l.*

When L is an LTS modeling a block DAG simulation of a deterministic protocol that is modeled as an LTS L', the existence of a forward simulation R from L to L' implies that the set of traces of L is included in the set of traces of L' [13]. It also implies the preservation of hyperproperties in programs that use a block DAG simulation instead of the original protocol as shown in [1].

These results extend to randomized protocols as well. Assuming that the random choices follow the uniform distribution, a forward simulation would imply that any random choice in L is mimicked in precisely the same manner by L'. This is because the label of every step that includes a random choice is an external action and the result of that random choice is included in the label itself. More formally, it will imply the existence of a *weak probabilistic simulation* which is known to imply that the probability distributions over traces of L defined by a deterministic scheduler are included in the probability distributions over traces of L' defined by a deterministic scheduler [17]. Moreover, it will also imply the preservation of probability distributions over executions of programs that use the block DAG simulation instead of the original protocol (this is a consequence of weak probabilistic simulations being sound for the trace distribution precongruence [17]).

Therefore any standard specification of a protocol, e.g., safety or (almost-sure) termination, is preserved by a block DAG simulation provided the existence of a forward simulation. Moreover, typical specifications of programs using the DAG simulation instead of the original protocol will also be preserved.

2.6 Cryptographic Primitives

Our block DAG simulation uses secure hash function and signatures. These cryptographic primitives are traditionally defined with a security parameter, but here, we assume they are *perfect.* This is justified because we can choose a security parameter that is sufficiently small so that the probability of failure is negligible.[1] We omit the security parameter from the definitions and assume zero probability of failure.

Definition 4. *An efficiently-computable function $h : X \to Y$ is a* secure cryptographic hash function *if for any computationally bounded adversary:*

(1) *(Preimage resistance) It is infeasible to find a preimage $x \in X$ s.t. $h(x) = y$ for any $y \in Y$ for which no preimage is known.*
(2) *(Collision resistance) It is infeasible to find two preimages $x_1, x_2 \in X$ s.t. $h(x_1) = h(x_2)$.*

Definition 5. *A* cryptographic signature scheme *consists of three efficiently-computable functions* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ *with the following properties:*

(1) *The key-generation function* Gen *produces a pair of keys* (pk, sk).
(2) *The signing function* Sign *takes a secret key sk and a string m and outputs a signature σ.*
(3) *The verification function* Vrfy *takes a public key pk, a message m and a signature σ and outputs a bit b.*

The scheme must satisfy that for any honestly generated $(pk, sk) \leftarrow \mathsf{Gen}$ *and any message m, it holds that* $\mathsf{Vrfy}(pk, \mathsf{Sign}(sk, m), m) = 1$. *The scheme is* secure *if it is infeasible for any message m and any computationally bounded adversary that does not know sk to produce a signature σ^* s.t.* $\mathsf{Vrfy}(pk, \sigma^*, m) = 1$.

We assume the existence of a secure cryptographic hash function $h : \{0,1\}^* \to \{0,1\}^*$ and a secure cryptographic signature scheme $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$. Since we assume the computational power of the adversary is bounded, it cannot forge messages or break cryptographic primitives. We also assume that every process p has already generated a key-pair (sk_p, pk_p) and distributed the public key pk_p to all other processes, which allows other processes to verify signatures of p. We denote $\mathsf{Sign}_p(\cdot) := \mathsf{Sign}(sk_p, \cdot)$ and $\mathsf{Vrfy}_p(\cdot, \cdot) := \mathsf{Vrfy}(pk_p, \cdot, \cdot)$.

3 BLOCK DAGS

A *block* is the main type of message that is exchanged in DAG-based protocols and our block DAG simulations. A block contains the identity of the issuing process and some data, and is additionally signed by the issuing process to prevent forgery. Formally:

Definition 6 (Block). *A block B is a message that consists of the identity of the issuing process p, some data d, a hash $h(B.p \circ B.d)$ of the data and the issuing process which we denote by* $\mathsf{ref}(B)$ *and a signature* $\sigma = \mathsf{Sign}_p(\mathsf{ref}(B))$ *of the issuing process p over the hash. The data consists of:*

(1) *A sequence number k.*
(2) *A finite list of hashes of predecessor blocks $preds$.*
(3) *A finite list of requests $rqsts$ that come from the user.*
(4) *Auxiliary information aux.*

This definition is similar to that of [16], with the added auxiliary information (4) on which we will elaborate in Section 4.1. We use register notation, e.g., $B.p$ is the issuing process of B. We say that a block $B' \in B.preds$ is the parent of B if $B.p = B'.p$ and $B.k = B'.k + 1$, and we denote $B' = B.parent$. A *genesis* block is a block B with sequence number $B.k = 0$. We define the *ancestors* of a block B to be all of the predecessors of B, and their predecessors and so on; this set is denoted $\mathsf{ancestors}(B)$.

Correct processes in DAG-based protocols usually add a block to their block DAG only if they can validate it. Validation typically entails checking that block was created according to the rules, but

[1] This is also a standard assumption in distributed algorithms [3, 16].

in addition, a process cannot add a block without adding its predecessors, therefore the validation process is recursive, meaning that a block can be valid only if its predecessors are valid. Formally:

Definition 7 (Valid Block). *A correct process p_i considers a block B* valid*, denoted* $\mathsf{valid}(p_i, B)$*, if all of the following hold:*

- *It considers all blocks in $B.preds$ valid.*
- *It can verify the signature* $\mathsf{Vrfy}_{B.p}(B.\sigma, \mathsf{ref}(B))$.
- *B has exactly one parent or is a genesis block.*

Since a process only validates blocks that are properly signed and we assume that the adversary cannot forge signatures, then we may assume that only authentic blocks are exchanged in the system.

As mentioned before, each process uses the valid blocks it has received to build a DAG which we call a *block DAG*. The vertices are all valid blocks and the edges correspond to the predecessor relation between blocks, that is (B', B) is an edge if and only if $B' \in B.preds$. Formally:

Definition 8 (Block DAG). *For a correct process p_i, $\mathcal{G} = (V_{\mathcal{G}}, E_{\mathcal{G}})$ is a* Block DAG of p_i *if:*

- $V_{\mathcal{G}} \subseteq \{B : B \text{ is a block that } p_i \text{ considers valid}\}$.
- *If $B \in V_{\mathcal{G}}$ then for all $B' \in B.preds$ it holds that $B' \in V_{\mathcal{G}}$.*
- $E_{\mathcal{G}} = \{(B', B) \in V_{\mathcal{G}} \times V_{\mathcal{G}} : B' \in B.preds\}$.

Note that by the preimage resistance of h, it holds that if $B' \in \mathsf{ancestors}(B)$ then $B \notin \mathsf{ancestors}(B')$ That is, the blocks guarantee that there are no cycles, and $\mathcal{G}$ is indeed acyclic. If p_i receives a new block B that it considers valid, then it can add it to $V_{\mathcal{G}}$ and update $E_{\mathcal{G}}$ accordingly and the result would be a new block DAG. We abuse notation and write $B \in \mathcal{G}$ if $B \in V_{\mathcal{G}}$.

Observe that by definition, if a block B is in some block DAG $\mathcal{G}$, then all of the ancestors of B are in $\mathcal{G}$ as well.

4 SIMULATING PUBLIC-COIN PROTOCOLS WHICH USE SHARED OBJECTS

Simulating a protocol on a block DAG consists of two components: first, a mechanism that allows processes to build and maintain a *joint block DAG* and second, an algorithm to *interpret* this joint block DAG as an execution of the original protocol. Given those two ingredients, we can execute an instance of the protocol without sending any actual messages that are specific to the protocol itself. Of course, maintaining the joint block DAG would require exchanging one type of messages (blocks), but those messages are agnostic to the protocol being simulated. This means that we can use the same joint block DAG to interpret multiple instances of the same protocol or even instances of different protocols.

Fig. 1 describes how to simulate a public-coin protocol $\mathcal{P}$ using the components mentioned above. We refer to this protocol as the *block DAG simulation of $\mathcal{P}$* and denote it by $\mathsf{BD}(\mathcal{P})$. We allow $\mathsf{BD}(\mathcal{P})$ to access the same shared objects as $\mathcal{P}$.

Interpreting the block DAG as an execution of $\mathcal{P}$ is done using the interpret algorithm. This algorithm runs locally and involves no communication, yet guarantees that if two correct processes are interpreting the same (partial) block DAG, then their interpretations would be identical. This algorithm is presented and discussed in Section 4.1.

Maintaining the joint block DAG is done using the genBlock and echo algorithms: genBlock is responsible for creating and disseminating new blocks and echo is responsible for passing those blocks around to ensure that all correct processes receive the same blocks even if the process that issued the block is corrupted. These algorithms are presented and discussed in Section 4.2.

The aforementioned components, together, ensure that correct processes have consistent views of the execution of $\mathcal{P}$ at all times. However, this does not guarantee that the execution is useful,

Simulation of Public-Coin Protocols on Block DAGs

From the perspective of process p_i, user requests go directly to $Rqsts_i$.
Initialize G_i as an empty block DAG and *blks* as an empty set of blocks.
On every compute(i) event:

(1) Run genBlock$(G_i, blks)$.
(2) If new blocks were added to G_i, then run interpret$(G_i, \mathcal{P})$.
(3) Run echo$(G_i, blks)$.

Fig. 1. The simulation algorithm for public-coin protocols

e.g., it might give the adversary more power or it might be a "liveless" execution where the correct processes are not making any progress. For that reason, we prove in Section 4.3 that the execution (defined by the views) is faithful in the sense that there exists a forward simulation towards the original protocol. This guarantees that the simulation of $\mathcal{P}$ on the block DAG preserves $\mathcal{P}$'s original specification.

4.1 Common Interpretation

Given a block DAG $\mathcal{G} = (V, E)$, we want to interpret it as an execution of the protocol. We call this execution the *simulated execution.* Furthermore, we need the interpretation to be consistent among all correct processes doing it.

The idea is to view $\mathcal{G}$ as a causality graph, where a block in $\mathcal{G}$ issued by some process p_i corresponds to a node that belongs to p_i in the causality graph, and such a node corresponds to a compute(i) in the simulated execution. In order to interpret $\mathcal{G}$, we need to interpret each block separately, where the interpretation of the block consists of the local process state and the outgoing messages of the issuing process after the corresponding compute(i) event. For convenience, we also treat the incoming messages (right before the event) as part of the interpretation. Formally:

Definition 9 (Block Interpretation). *The* interpretation *of a block B has the following fields:*

(1) *A local process state $B.PS$.*
(2) *A list of incoming messages $B.M_{in}$.*
(3) *A list of outgoing messages $B.M_{out}$. For convenience, we denote by $M_{out}[j]$ the outgoing messages in M_{out} that are designated to p_j.*

Note that the interpretation of a block is not sent over the network. That is, when a process receives or even generates a new block, it does not automatically have its interpretation. This is crucial because we do not want the size of the block to increase with the number of messages, and instead we only want the block to include information that processes cannot locally compute unambiguously. As such, it is the responsibility of each process to interpret each block it has locally.

In a regular execution of a *deterministic* protocol, whenever a compute(i) event is scheduled, the process p_i performs the following: it passes all of the message in $In_{j\to i}$ to the local state of its protocol instance PS_i and performs a local computation. This updates the local state PS_i, produces new outgoing messages that are deposited into $Out_{i\to j}$ and may return user indications. Our interpretation protocol tries to mimic the execution by assigning to $B.PS$ the local state of the process after the corresponding event, $B.M_{out}[j]$ the messages that would be deposited in $Out_{i\to j}$ and $B.M_{in}$ the messages that would have been in $In_{j\to i}$ before the event. In addition, if the block B was issued by the process doing the interpretation and $B.PS$ produces a user indication, then the process must actually return the indication to the user.

Algorithm 1 interpret$(G_i, \mathcal{P})$ for process p_i

$G_i = (V_i, E_i)$ is a block DAG and $\mathcal{P}$ is a public-coin protocol.
G_i is process-local variable that maintains its value across different invocations

```
while ∃B ∈ G_i s.t. B is not interpreted s.t. ∀B′ ∈ B.preds : B′ is interpreted do
    if B.k = 0 then
        Initialize B.PS as a new state according to the protocol P and process B.p
    else
        B.PS := B.parent.PS
    for all B′ ∈ B.preds do
        Copy messages from B′.M_out[B.p] to B.M_in
    Pass the user requests B.rqsts, messages B.M_in, random tape B.rand and the object indications B.buff to the state B.PS
    Overwrite the new state in B.PS
    Store the outgoing messages in B.M_out
    if B.p = i then
        Return user indications produced by B.PS to the user
        Perform object invocations as dictated by B.PS
```

When extending this approach to *randomized* protocols, we need to account for the local randomness. In this case, the process state expects to receive a random tape and the result of the computation, i.e., the new PS_i and $Out_{i \to j}$. To guarantee that the interpretation is consistent for all correct processes, we use the same randomness tape when interpreting a block. For that reason, we need the random tape to be attached to the block, which is where $B.aux$ comes in: each process that issues a block B, attaches the random tape in a specific field in $B.aux$ which we call $B.rand$, thus forcing all other processes to use the same randomness when interpreting its blocks.

When further extending this to protocols with *shared objects*, we need to handle object invocations and object indications. When an object o returns an indication to p_i, there is not necessarily a way for all other processes to learn that indication on their own. (This can happen, for example, in a *secret sharing* object, where the different indications of processes are independent.) For that reason, we need processes to attach those indications to their blocks as well: when p_i receives an indication from object o, that indication is added to a specific field in $B.aux$ which we call $B.buff[\mathrm{o}]$. These indications are then passed to $B.PS$ (alongside $B.M_{in}$ and $B.rand$) when interpreting the block. As for object invocations, they should be handled in a similar fashion to user indications, that is if the interpreting process is $B.p$ and $B.PS$ dictates that o must be invoked, then the interpreting process actually performs the invocation.

With all of this in mind, the interpretation is described in Algorithm 1. At a high level, we interpret the blocks in a topological order and rely on the fact that if we feed the same messages, randomness and object indications to the same process state, then we would always get the same interpretation. We then apply this idea inductively and prove that the interpretation of any specific block is consistent among all correct processes. To that end, for any block block DAG G and $B \in G$, we denote by interpret$(G, \mathcal{P}).B$ the interpretation of B when running interpret$(G, \mathcal{P})$. Lemma 1 formalizes the main guarantee of Algorithm 1.

Lemma 1. *For any two block DAGs G_1 and G_2, if $B \in G_1$ and $B \in G_2$ then* interpret$(G_1, \mathcal{P}).B =$ interpret$(G_2, \mathcal{P}).B$.

Proof. For any block $B \in G_1 \cap G_2$, let $B.PS^1, B.M^1_{in}, B.M^1_{out}$ and $B.PS^2, B.M^2_{in}, B.M^2_{out}$ be the interpretations interpret$(G_1, \mathcal{P}).B$ and interpret$(G_2, \mathcal{P}).B$, respectively. Recall that by definition,

$B.p, B.preds, B.rqsts$ and $B.aux$ (which consists of $B.rand$ and $B.buff$) are identical in G_1 and G_2. We note that any path in G_1 that ends in B is also a path in G_2 and vice versa since both G_1 and G_2 include all of the ancestors of B. Define $\ell(B)$ to be the length of the longest such path from a genesis block to B. We utilize the fact that if $\ell(B) \geq 1$ then $\ell(B') < \ell(B)$, for any $B' \in B.preds$. This allows us to prove the lemma by induction on ℓ.

- **Base:** If $\ell(B) = 0$ then B is a genesis block. By the construction of interpret, it holds that both $B.PS^1$ and $B.PS^2$ are initialized with the initial state of $B.p$ w.r.t. $\mathcal{P}$. Since $\ell(B) = 0$, we know that $B.preds = \emptyset$ Therefore, there are no incoming messages, i.e., $B.M^i_{in} = \emptyset$ for all $i \in \{1, 2\}$. By the construction of interpret, $\mathsf{interpret}(G_i, \mathcal{P})$ is computed by feeding $B.M^i_{in}, B.rqsts$ and $B.aux$ to $B.PS^i$ and since $B.M^1_{in} = B.M^2_{in}$ and $B.PS^1 = B.PS^2$, we get that $\mathsf{interpret}(G_1, \mathcal{P}) = \mathsf{interpret}(G_2, \mathcal{P})$.
- **Hypothesis:** Assume that for all blocks B' with $\ell(B') < l$ it holds that $\mathsf{interpret}(G_1, \mathcal{P}).B' = \mathsf{interpret}(G_2, \mathcal{P}).B'$.
- **Step:** Let B be a block with $\ell(B) = l$. By the induction hypotheses, $\mathsf{interpret}(G_1, \mathcal{P}).B' = \mathsf{interpret}(G_2, \mathcal{P}).B'$ for all $B' \in B.preds$. This implies that both $B.PS^1$ and $B.PS^2$ are initialized with the same value and that $B.M^1_{in} = B.M^2_{in}$. By the same argument we have used in the base of the induction, it follows that $\mathsf{interpret}(G_1, \mathcal{P}) = \mathsf{interpret}(G_2, \mathcal{P})$.

■

4.2 Joint Block DAG

We have shown that processes that interpret the same blocks reach the same conclusion. But for this to be useful, we must show that correct processes eventually receive the same blocks. This is where communication is used. Recall that our model assumes eventual delivery, that is, every message sent is eventually delivered, albeit with a potentially unbounded delay. This is immediately inherited by blocks: if a process sends a block, then all other processes eventually receive that block.

We utilize this to allow processes to generate and disseminate new blocks in Algorithm 2. Since blocks are the only way for processes to inject information into the system, user requests are injected using blocks in Line 2. In our context of simulating public-coin protocols with shared objects, blocks must also include the local randomness and the object indications that might have been returned due to previous object invocations. We utilize the function $\mathsf{fillAux}(B)$ in Line 7 to perform those two tasks:

(1) Generate a random string ρ and assign it to $B.rand := \rho$.
(2) For each $\mathrm{o} \in O$, move the object indications from $\mathrm{o}.buff_i$ to $B.buff[\mathrm{o}]$.

While generating a new block B, processes also add valid blocks they receive to their block DAGs and to $B.preds$. This way, each correct process p_i extends its block DAG G_i gradually. By construction, only valid blocks are added to G_i and $B.preds$. Finally, the block is signed and sent to everyone. The condition in step 6 only guarantees that we do not send out blocks that do no contain new information.

We would like to claim now that if a correct process p_i issues a block $B := \mathsf{genBlock}(G_i, blks)$, then any other correct process p_j will eventually consider B valid and thus, add it to its block DAG. However, for that to happen, p_j must receive (and consider as valid) all the predecessors of B. Consider an adversarial process p^* that sends a block B^* to p_i but not to p_j. In this case, p_i generates a B such that $B^* \in B.preds$, but p_i will not consider B valid unless it also considers B^* valid. This issue can be easily solved with a simple *echoing* mechanism, presented in Algorithm 3.

Now we can claim that all correct processes eventually receive the same blocks:

Algorithm 2 genBlock(G_i, *blks*) for process p_i

$G_i = (V_i, E_i)$ is a block DAG and *blks* is a set of blocks. In the initial invocation, a variable k is initialized to 0 and its value is preserved throughout all invocations.

G_i, *blks* and k are process-local variables that maintain their values across different invocations.

Initialize a new block B as follows $B.p = p_i, B.k = k, B.preds = \emptyset, B.rqsts = \emptyset$
Move all user requests (if any) from $Rqsts_i$ to $B.rqsts$.
Move all blocks from all $In_{j\to i}$ to *blks*.
while $\exists B' \in blks$ s.t. valid(p_i, B') **do**
    Add B' to G_i and add B' to $B.preds$
if $B.preds \neq \emptyset \vee B.rqsts \neq \emptyset \vee \exists \mathrm{o} \in \mathcal{O}$ s.t. $\mathrm{o}.buff_i \neq \emptyset$ **then**
    Fill the auxiliary information fillAux(B).
    Sign the block B, i.e., $B.\sigma := \mathsf{Sign}_{p_i}(\mathsf{ref}(B))$, add it to G_i and broadcast it to everyone.
    Increment $k := k + 1$.
else
    $B := \perp$
return B

Algorithm 3 echo(G_i) for process p_i

$G_i = (V_i, E_i)$ is a block DAG and *blks* is a set of blocks.

G_i and *blks* are process-local variables that maintain their values across different invocations.

for all $B' \in blks$ s.t. B' is not valid **do**
    for all $B'' \in B'.preds$ s.t. $B'' \notin G_i$ **do**
        Send FWD(B'') to $B'.p$
for all FWD$(B'') \in In_{j\to i}$ **do**
    If $B'' \in G_i$, send B'' to p_j
    Remove FWD(B'') from $In_{j\to i}$

LEMMA 2. *For any two correct processes p_i, p_j executing the protocol of Fig. 1, if p_i adds a block to its block DAG G_i, then p_j eventually receives B.*

Proof. Let B be a block added to G_i. If B was issued by p_i, then p_i broadcasts that block to everyone and therefore p_j eventually receives it. Otherwise, by Algorithm 2, it must be that p_i created a new block B' and added B as the predecessor of B' (because this is the only a correct process adds a block it has not issued to its block DAG). By Algorithm 2, p_i sends B' to p_j and p_j eventually receives B'. If p_j considers B' valid upon reception, then it must hold that p_j received B by the definition of validity. Otherwise, it will send a FWD(B) to p_i. This request will be eventually received p_i, who in return will send B to p_j. Again, p_j will eventually receive B and the lemma follows. ∎

Furthermore, we claim that all correct processes validate the same blocks:

LEMMA 3. *For any two correct processes p_i and p_j executing the protocol of Fig. 1, if p_i adds a block to its block DAG G_i, then p_j eventually considers B valid.*

Proof. Let B be a block that was added to G_i. By Lemma 2, p_j will eventually receive B. Since p_i is a correct process, it must hold that valid(p_i, B) and specifically, that $B.p = p_s$ for some process p_s and $B.\sigma = \mathsf{Sign}_{p_s}(\mathsf{ref}(B))$, therefore p_j can verify the signature $B.\sigma$. In addition, B is either a genesis or has exactly one parent. It remains to show that all predecessors of B will eventually be considered valid by p_j. Similarly to Lemma 1, we prove this by induction on $\ell(B)$, the length of the longest path from a genesis block to B:

- **Base:** If $\ell(B) = 0$ then B is a genesis block. This means that the predecessors condition holds trivially.
- **Hypothesis:** Assume that all blocks B' with $\ell(B') < l$ will eventually be considered valid by p_j.
- **Step:** Let B be a block with $\ell(B) = l$. By the induction hypotheses, all of the predecessors of B will eventually be considered valid by p_j, since they must have a shorter path to a genesis block that l. Therefore, the predecessors condition of validity will eventually hold for B and thus p_j will consider it valid.

■

Lemma 3 essentially states that correct processes are building a joint block DAG in the sense that if a block B is added to the block DAG of a correct process, then it will eventually be added to the block DAGs of all other correct processes. We note that Lemmas 2 and 3 really refer to any protocol in which Algorithms 2 and 3 are continuously run, and are not specific to Fig. 1.

Liveness of Block DAG Simulations. Combining Lemma 3 with Lemma 1 and assuming eventual delivery of blocks, we get eventual delivery of simulated messages. In other words, if a correct process p_i wants to send a message m to some correct process p_j, then this expressed in the block DAG framework as a block B issued by p_i, such that $B.M_{out}[j]$ contains the message m. Delivering the message m to p_j is expressed by p_j creating a block B' such that $m \in B'.M_{in}$. Note that referring to unambiguous interpretations of B and B' is only possible through Lemma 1. By Lemma 3, we know that if p_i issues the block B then p_j eventually receives B and considers it valid. By the algorithm in Algorithm 2, eventually p_j creates a new block B' such that $B \in B'.preds$ and by Algorithm 1, m will be added to $B.M_{in}$. This discussion demonstrates that the block DAG framework guarantees eventual delivery of simulated messages, if we assume eventual delivery of blocks. Essentially, this guarantees the liveness of block DAG simulations.

4.3 Correctness of the Simulation

We show that the block DAG simulation of a protocol $\mathcal{P}$ inherits all of the properties of $\mathcal{P}$, which we accomplish by proving that there exists a *forward simulation* from the block DAG simulation denoted as $\mathrm{BD}(\mathcal{P})$ to $\mathcal{P}$ (modeled as LTSs). Section 2.5 describes the modeling of $\mathcal{P}$ using LTSs. We describe below a modeling of $\mathrm{BD}(\mathcal{P})$ using an LTS which simplifies the forward simulation proof.

Modeling the Block DAG simulation as an LTS. We describe the components of an LTS $L' = (Q', \Sigma', q'_{start}, \delta')$ that we use to model $\mathrm{BD}(\mathcal{P})$. A state $q' \in Q'$ contains the block DAG G_i of each process p_i and $(In^B_{j\to i})_{j\in[n]}$ and $(Out^B_{i\to j})_{j\in[n]}$ for each process p_i, where $In^B_{j\to i}$ is the incoming buffer of process i with blocks sent by process j and $Out^B_{i\to j}$ is the outgoing buffer with blocks sent by i to j. As before, we assume that incoming user requests are stored in $In^B_{i\to i}$ and outgoing user indications are stored in $Out^B_{i\to i}$. The shared object indications are stored in separate buffers $(\mathrm{o}.\mathit{buff}_i)_{\mathrm{o}\in O}$ as before. Overall, $q' = \left(G_i, (In^B_{j\to i})_{j\in[n]}, (Out^B_{i\to j})_{j\in[n]} (\mathrm{o}.\mathit{buff}_i)_{\mathrm{o}\in O}\right)_{i\in[n]}$. In the initial state q'_{start}, all of the block DAGs and the buffers are empty. The transition labels correspond to computing and validating blocks, exchanging blocks, and user requests or indications. In comparison to the "standard" model described in Section 2.5 we decompose a compute step of a process as defined in Fig. 1 into a sequence of steps. This simplifies the forward simulation proof. As before, we include the randomness (that is attached to the newly created block) in the computation label. Formally, the transition labels are as follows:

(1) validateBlock$(i \rightarrow j)$ denotes a transition where p_j validates a block issued by p_i (inside the genBlock algorithm).
(2) compute(i, ρ) denotes a transition where process p_i produces and disseminates a new block (inside the genBlock algorithm) with ρ as its randomness, and then runs interpret to interpret the new block (and other previously uninterpreted blocks).
(3) sendFWD$(i \rightarrow j)$ denotes a transition where p_i sends a FWD request to p_j.
(4) replyFWD$(i \rightarrow j)$ denotes a transition where p_i sends a reply to a FWD sent by p_j.
(5) deliverBlocks$(i \rightarrow j)$ denotes a transition where all of the blocks in $Out^B_{i\rightarrow j}$ are moved to $In^B_{i\rightarrow j}$.
(6) o.indicate(i, w) denotes a transition where the value w has been added to o.$buff_i$.
(7) labels for user requests (request(i, x)) or indications (indicate(i, y)) have the same meaning as in Section 2.5.

The external actions Σ_E are defined exactly as for the LTS L modeling $\mathcal{P}$, presented in Section 2.5 (Σ_E includes request(i, x), indicate(i, y), and compute(i, ρ)). A transition $(q'_1, e, q'_2) \in Q' \times \Sigma' \times Q'$ (denoted $q'_1 \xrightarrow{e} q'_2$) is in δ' if and only if the protocol BD($\mathcal{P}$) can get from state q'_1 to state q'_2 by executing the step denoted by the label e.

Theorem 1. *There exists a forward simulation from the LTS L' modeling* BD($\mathcal{P}$) *to the LTS L modeling $\mathcal{P}$.*

Proof. We define a relation $R \in Q' \times Q$ as follows: $q'\ R\ q$ if and only if all of the following holds for each $i, j\ in[n]$:

(1) $q.PS_i = B.PS$, where B is the most recent block issued by p_i in $q'.G_i$. If there is no such block, then $q.PS_i$ is the initial state.
(2) for every $i \neq j$, $q.Out_{i\rightarrow j}$ includes every message m such that there exists a block B with $m \in B.M_{out}[j]$, and B is created by p_i but not yet validated by p_j.
(3) for every $i \neq j$, $q.In_{j\rightarrow i}$ includes every message m such that there exists a block B with $m \in B.M_{out}[i]$, and B is created by p_j, validated by p_i, but not yet interpreted by p_i.
(4) for every i, $q.In_{i\rightarrow i} = q'.In^B_{i\rightarrow i}$ and $q.Out_{i\rightarrow i} = q.Out^B_{i\rightarrow i}$ (the same user requests and indications).
(5) for every i and $\mathrm{o} \in \mathcal{O}$, $q.\mathrm{o}.buff_i = q'.\mathrm{o}.buff_i$ (the same object indications).

Next, we show that R is indeed a forward simulation from L' tp L. It is clear that $q'_{start}\ R\ q_{start}$ by construction. Let q_1, q'_1 be two states such that $q'_1\ R\ q_1$ and let $q'_1 \xrightarrow{e'} q'_2$ be a transition in δ'. We show that there exists q_2 such that $q'_2\ R\ q_2$, $q_1 \xrightarrow{e} q_2$ is a transition in δ (or a stuttering step), and if $e \in \Sigma_E$, then $e = e'$. We do a case analysis based on the label e':

(1) If $e' \in \{$request(i, x), indicate(i, y), o.indicate$(i, w)\}$, then the only difference between the two states q'_1, q'_2 is in the requests, indications or object indications buffer of p_i. Let $q_2 \in Q$ be a state such that $q'_2\ R\ q_2$. By the definition of R, it holds that the only difference between q_1 and q_2 is in same buffer, so it holds that $q_1 \xrightarrow{e'} q_2$ for the same label e'.
(2) If $e' \in \{$sendFWD$(i \rightarrow j)$, replyFWD$(i \rightarrow j)$, deliverBlocks$(i \rightarrow j)\}$, then $q'_2\ R\ q_1$. This is because the definition of R does not look at FWD requests or replies, or delivery of created blocks (items 2 and 3 concern the creation or the validation of a block and not when a block is sent or received). Therefore we can choose $e = \epsilon$ and define $q_2 = q_1$ (stuttering step).
(3) If $e' =$ validateBlock$(i \rightarrow j)$, then q'_2 contains one more block B which is validated by p_j. Assume that B was created by process p_i. Since B had to exist in q'_1, for every j, $q_1.Out_{i\rightarrow j}$ includes every message in $B.M_{out}[j]$. We define q_2 as the state obtained from q_1 by moving

all messages from $q_1.Out_{i\to j}$ to $q_2.In_{j\to i}$. Also, let e be the label deliver$(j \to i)$. It is quite easy to check that $q_2' \; R \; q_2$ and $q_1 \xrightarrow{e} q_2$.

(4) If $e' = \mathsf{compute}(i,\rho)$ for some $i\in[n], \rho \in \{0,1\}^*$, then the only difference between q_1' and q_2' is in the block DAG G_i and the buffers $(In^B_{i\to j}, Out^B_{i\to j})_{j\in[n]}$ and $(\text{o}.\mathit{buff}_i)_{\text{o}\in O}$. Let B be the block that p_i disseminated in q_2' and $B' = B.\mathit{parent}$. By the definition of R, it holds that $q_1.PS_i = B'.PS$, and for every $j \neq i$, $q_1.In_{j\to i} = \{m : \exists B_0 \in B.\mathit{preds}, m \in B_0.M_{out}[i]\}$, and $q_1.In_{i\to i} = q_1'.In^B i \to i$. In addition, the object buffers in both q_1 and q_1' are equal, that is $(q_1.\text{o}.\mathit{buff}_i)_{\text{o}\in O} = (q_1'.\text{o}.\mathit{buff}_i)_{\text{o}\in O}$. Now let $q_2 \in Q$ be a state such that $q_1 \xrightarrow{\mathsf{compute}(i,\rho)} q_2$. We show that $q_2' \; R \; q_2$. Therefore, we have that $q_2.PS_i = B.PS$. This is because B is interpreted by feeding the messages $(q_1.In_{j\to i})_{j\in[n]}$ and the object indications $(q_1.\text{o}.\mathit{buff}_i)_{\text{o}\in O}$ with the randomness $B.\mathit{rand}$ to the state $q_1.PS_i = B'.PS$. Note that by the definition of the label $\mathsf{compute}(i,\rho)$ in L', it holds that $B.\mathit{rand} = \rho$.

This concludes the proof of Theorem 1. ■

5 RELATION TO PRIOR WORK

5.1 Comparison with Schett and Danezis

Now that we have presented our simulation and its proof, we can discuss how they are related to the work of Schett and Danezis [16].

Our network component which consists of genBlock and echo algorithms is a natural extension of the gossip algorithm of [16]. Indeed, the code responsible for generating new blocks and echoing them is almost identical to that of gossip. We only observe that blocks should carry enough information to resolve the randomized decisions that can come from local randomness or shared objects. This is because we want to exchange only blocks. In our protocol, each process is responsible to pass along its local randomness or the indications it got from the shared object in the blocks that it creates. The proofs of Lemmas 2 and 3 follow the proof of [16, Lemma 3.7].

Our interpretation algorithm is the natural extension of interpret algorithm of [16] for our context. That is, when interpreting a deterministic protocol, the computation of each process is only determined by the incoming messages and its state prior to processing those messages. When interpreting a randomized protocol with shared objects, the local computation may depend on local randomness and object indications. Our interpretation algorithm used those fields that were already attached to each block by our genBlock. We note that Lemma 1 that states the common interpretation of block DAGs, is analogous to [16, Lemma 4.2]. However, the proof of the latter had a very minor mistake (acknowledged by the authors [15]), and therefore our proof is slightly different.

Finally, the guarantees of randomized protocols, unlike those of deterministic protocols, cannot always be expressed as trace properties. Particularly, for our simulation to be faithful to the original protocol, we need a more careful and precise statement and proof. Therefore, the modeling in Sections 2.5 and 4.3 as well as the proof of Theorem 1 are totally different from what appears in [16].

5.2 Analyzing Existing Protocols

Here we discuss how our simulation applies to existing protocols, concentrating on Aleph [8] and DAG-Rider [10]. These protocols aim to order the blocks of the DAG, so as to implement *Byzantine Atomic Broadcast* (BAB). We briefly recall the problem of BAB and summarize how [8, 10] achieve this goal.

A BAB protocol is a broadcast protocol that allows all processes to receive the same messages in the *same order*. One natural way of implementing a BAB protocol using a block DAG is by having each process attach the messages it wants to broadcast to a block and then broadcast the block to everyone. The processes then just need to agree on an order of the blocks, which would induce an order of the messages.

Analogous to our simulation, both Aleph and DAG-Rider have a communication component that is responsible for building and maintaining the common DAG. In both protocols, each block in the DAG belongs to a specific round, and each correct process has a single block in each round.

In Aleph [8], ordering the blocks in the DAG is achieved by electing a leader block in each round, and then having that leader block (deterministically) dictate the order of its ancestor blocks that have not been ordered yet.

In DAG-Rider [10], the DAG is divided into *waves* where each one consists of four consecutive rounds, and a leader block is elected for each wave. The block leader election is done by interpreting the (same) block DAG as a consensus protocol and utilizing a shared object for generating randomness, namely, a common coin. It is critical to note that our simulation preserves the properties of the shared object, for example the *unpredictability* of the common coin. This is because our forward simulation preserves the compute events, in which the object invocations happen. This means that the object cannot distinguish if it is being used in the context of the original protocol or in the context of the block DAG simulation of the protocol. This means that its properties are preserved.

Aleph and DAG-Rider can be easily analyzed using our framework. The consensus protocol used can be analyzed independently of Aleph or DAG-Rider, while assuming it has access to a common coin. Then by Theorem 1, the simulation of the consensus protocol on the black DAG is faithful to the original consensus protocol. This would not only simplify reasoning about safety and liveness of Aleph and DAG-Rider, but it would support *modularity*: the simulated consensus protocol in Aleph or DAG-Rider can be seamlessly replaced using Theorem 1.

6 DISCUSSION

We have presented a faithful simulation of DAG-based BFT protocols, which use public coins and shared objects, including protocols that utilize a common source of randomness, e.g., a *common coin*. Being faithful, the simulation precisely preserves properties of the original BFT protocol, and in particular, their probability distributions.

One of the appealing properties of our block DAG framework is that it allows to minimize the communication when running multiple instances of potentially different protocols. This can be done by using the same joint block DAG to interpret multiple protocol instances. The logic of the communication layer does not change, other than the need to specify the associated instance for each user request and object indication that is attached to the blocks. Each process would then run multiple interpretation instances, one for each protocol instance. We note that a process does not necessarily need to attach a separate randomness tape for each instance, and can instead attach a small random seed. Processes can then use a *pseudorandom generator* to expand the seed to a large enough pseudorandom string that can be used for all of the instances. This ensures that block size does not grow beyond the size of the user requests and the object indications.

Our simulation relies on the fact that it is safe to reveal the randomness to the adversary as soon as it is used. We can similarly define *private-coin* protocols, whose security relies on processes ability to keep secrets from the adversary. A classical example would be any Asynchronous Verifiable Secret Sharing scheme (e.g. [4]). From a theoretical point of view, it would be interesting to demonstrate how we can simulate such algorithms on block DAGs. However, we note that some protocols are entirely public-coin other than a dedicated private-coin sub-protocol, such as Aleph-Beacon in

Aleph [8] (which is used to implement a common coin). In this case, the dedicated sub-protocol can be encapsulated as a shared object, thus factoring out the use of private-coin simulations.

ACKNOWLEDGMENTS

H. Attiya was partially supported by the Israel Science Foundation (grants 380/18 and 22/1425). S. Nassar was partially funded by the European Union (ERC, FASTPROOF, 101041208). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

REFERENCES

[1] Hagit Attiya and Constantin Enea. Putting strong linearizability in context: Preserving hyperproperties in programsthat use concurrent objects. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, volume 146 of *LIPIcs*, pages 2:1–2:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[2] Leemon Baird. The Swirlds Hashgraph consensus algorithm: Fair, fast, Byzantine fault tolerance. In *Technical Report*, 2016.

[3] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.).* Springer, 2011.

[4] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 42–51. ACM, 1993.

[5] Anton Churyumov. Byteball: A decentralized system for storage and transfer of value. In *Technical Report*, 2016.

[6] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis, editors, *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 34–50. ACM, 2022.

[7] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. In Ronald Fagin and Philip A. Bernstein, editors, *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 21-23, 1983, Colony Square Hotel, Atlanta, Georgia, USA*, pages 1–7. ACM, 1983.

[8] Adam Gagol, Damian Lesniak, Damian Straszak, and Michal Swietek. Aleph: Efficient atomic broadcast in asynchronous networks with Byzantine nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019*, pages 214–228. ACM, 2019.

[9] Shang-En Huang, Seth Pettie, and Leqi Zhu. Byzantine agreement in polynomial time with near-optimal resilience. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 502–514. ACM, 2022.

[10] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 165–175. ACM, 2021.

[11] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[12] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. Untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

[13] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

[14] Sean Rowan and Naïri Usher. The Flare consensus protocol: Fair, fast federated Byzantine agreement consensus. In *Technical Report*, 2019.

[15] Maria Schett. Personal communication, January 2022.

[16] Maria Anna Schett and George Danezis. Embedding a deterministic BFT protocol in a block DAG. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 177–186. ACM, 2021.

[17] Roberto Segala. A compositional trace-based semantics for probabilistic automata. In Insup Lee and Scott A. Smolka, editors, *CONCUR '95: Concurrency Theory, 6th International Conference, Philadelphia, PA, USA, August 21-24, 1995, Proceedings*, volume 962 of *Lecture Notes in Computer Science*, pages 234–248. Springer, 1995.

[18] Yonatan Sompolinsky, Shai Wyborski, and Aviv Zohar. PHANTOM GHOSTDAG: a scalable generalization of nakamoto consensus: September 2, 2021. In Foteini Baldimtsi and Tim Roughgarden, editors, *AFT '21: 3rd ACM Conference on Advances in Financial Technologies, Arlington, Virginia, USA, September 26 - 28, 2021*, pages 57–70. ACM, 2021.

[19] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2705–2718. ACM, 2022.

[20] Qin Wang, Jiangshan Yu, Shiping Chen, and Yang Xiang. SoK: Diving into DAG-based blockchain systems. *CoRR*, abs/2012.06128, 2020.