



**HAL**  
open science

# Faithful Simulation of Randomized BFT Protocols on Block DAGs

Hagit Attiya, Constantin Enea, Shafik Nassar

► **To cite this version:**

Hagit Attiya, Constantin Enea, Shafik Nassar. Faithful Simulation of Randomized BFT Protocols on Block DAGs. 34th International Conference on Concurrency Theory, CONCUR 2023,, Sep 2023, Antwerp, Belgium. 10.4230/LIPIcs.CONCUR.2023.27 . hal-04465413

**HAL Id: hal-04465413**

**<https://hal.science/hal-04465413v1>**

Submitted on 19 Feb 2024



**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Faithful Simulation of Randomized BFT Protocols on Block DAGs

Hagit Attiya  

Technion, Israel

Constantin Enea  

LIX, Ecole Polytechnique, CNRS and Institut Polytechnique de Paris, France

Shafik Nassar  

Technion, Israel

---

## Abstract

Byzantine Fault-Tolerant (BFT) protocols that are based on *Directed Acyclic Graphs* (DAGs) are attractive due to their many advantages in asynchronous blockchain systems. These DAG-based protocols can be viewed as a *simulation* of some BFT protocol on a DAG. Many DAG-based BFT protocols rely on randomization, since they are used for agreement and ordering of transactions, 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*.

A DAG simulation of a randomized protocol should be *faithful*, in the sense that it precisely preserves the properties of the original BFT protocol, and in particular, their probability distributions. We argue that faithfulness is ensured by a *forward simulation*. We show how to faithfully simulate any BFT protocol that uses public coins and shared objects, like common coins.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms; Theory of computation → Distributed computing models; Computing methodologies → Distributed algorithms; General and reference → Verification

**Keywords and phrases** Distributed Algorithms, Byzantine failures, Hyperproperties, Forward Simulation

**Digital Object Identifier** 10.4230/LIPIcs.CONCUR.2023.27

**Related Version** <https://eprint.iacr.org/2023/192>

**Funding** *Hagit Attiya*: partially supported by the Israel Science Foundation (grants 380/18 and 22/1425).

*Constantin Enea*: partially supported by the project AdeCoDS of the French ANR Agency.

*Shafik Nassar*: 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.

## 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 [14] 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 [21] 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.



© Hagit Attiya, Constantin Enea and Shafik Nassar;  
licensed under Creative Commons License CC-BY 4.0

34th International Conference on Concurrency Theory (CONCUR 2023).

Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 27; pp. 27:1–27:17



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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

51 This generality makes block DAGs an attractive approach for designing coordination  
 52 protocols for, e.g., Byzantine Atomic Broadcast [10, 13, 20], consensus [4, 16] and cryptocur-  
 53 rencies [6]. (For a survey of the techniques used in block DAG approaches, see [21].) A block  
 54 DAG can be seen as a strict extension of a *blockchain*, which is a DAG where all blocks  
 55 are *totally ordered*, i.e., a directed path. The DAG approach was shown to achieve high  
 56 throughput [19] due to the flexibility it provides over the standard blockchain approach.

57 Schett and Danezis [17] show that any *deterministic* BFT protocol can be simulated as a  
 58 block DAG. They provide generic mechanisms for processes to maintain a consistent view of  
 59 the block DAG, and to individually *interpret* the DAG as an execution of some protocol.

60 The restriction to deterministic protocols, however, handicaps the applicability of this  
 61 result, since many algorithms in the asynchronous domain are necessarily non-deterministic,  
 62 due to the FLP impossibility result [9]. For example, DAG-based agreement protocols with  
 63 provable security, like Aleph [10] or DAG-Rider [13], are either randomized or assume the  
 64 existence of a shared source of randomness. This calls for a framework that can handle  
 65 *randomized* BFT protocols; those that either utilize local randomness or even a shared object.

66 The problem of using or defining block DAG simulations in the context of *randomized*  
 67 *protocols* has two aspects: (1) using a block DAG simulation of a *deterministic* protocol  
 68 as a building block of a *randomized protocol*, and (2) defining block DAG simulations of  
 69 *randomized protocols*.

70 Concerning the first aspect above, we aim to enable modular reasoning when using such  
 71 simulations instead of the original protocols (Section 2 describes a concrete example). Schett  
 72 and Danezis [17] establish that the traces of the block DAG simulation are included in  
 73 the set of traces of the original protocol (for some notion of trace which is not important  
 74 for this discussion). However, as shown in other contexts, e.g., concurrent objects [2, 11],  
 75 such a notion of refinement is not sufficient to conclude that relevant specifications of a  
 76 randomized protocol that builds on some other deterministic protocol are preserved when  
 77 the latter is replaced by the block DAG simulation. Indeed, the specifications of randomized  
 78 protocols characterize sets (probabilistic distributions) of executions and are instances of  
 79 *hyper-properties* which are not preserved by standard trace inclusion [2].

80 Therefore, we establish a stronger notion of refinement between a block DAG simulation  
 81 and the original protocol, namely, that there exists a *forward simulation* between the two.  
 82 (A forward simulation maps every step of one protocol to a sequence of steps of the other  
 83 protocol, starting from the initial state of the first and advancing in a forward manner; a  
 84 backward simulation is similar, but it goes in the reverse direction, from end states back to  
 85 initial states.). Based on the results in [2], this implies that any finite-trace specification of  
 86 a randomized protocol against an adaptive adversary is preserved when a sub-protocol is  
 87 replaced by its block DAG simulation. We recall that an *adaptive adversary* is a scheduler  
 88 that resolves all the non-determinism introduced by the interleaving semantics and which  
 89 can observe everything about the local state of a process or the messages in transit.

90 Armed with this understanding of the precise nature of block DAG simulation, we present

■ **Algorithm 1** Binary consensus using a common coin

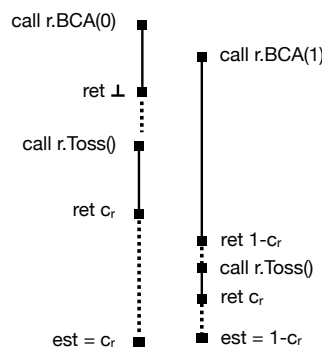
---

**Input:**  $x$

```

1:  $r := 0$ ;  $est := x$ ;
2: while true do
3:    $r++$ ;
4:    $val := r.BCA(est)$ ;
5:    $c := r.Toss()$ ;
6:   if  $val \neq \perp$  and  $c = val$  then
7:     output  $val$ ;
8:      $est := val$ ;
9:   else if  $val \neq \perp$  then
10:     $est := val$ ;
11:   else
12:     $est := c$ ;
```

---



■ **Figure 1** A randomized consensus algorithm on the left, and an execution template ( $c_0 \in \{0, 1\}$ ) on the right, which represents the executions of an adaptive adversary which disallows termination.

91 an extension of the construction of Schett and Danezis [17], which applies also to protocols  
 92 using randomization and shared objects. Specifically, we consider *randomized* protocols in  
 93 which the local coin flips of each process may be public, we call those protocols *public-coin*  
 94 protocols. We prove that any public-coin protocol that uses shared objects, e.g., common  
 95 coins, can be simulated on a block DAG, preserving its usage of shared objects.

96 A relationship based on a forward simulation allows to conclude that probabilistic  
 97 specifications of a randomized protocol, e.g., termination time, are preserved by its block  
 98 DAG simulation. Such a simulation precisely preserves the finite trace distribution and the  
 99 probabilistic relationship between inputs and outputs. This means that whatever “adverse”  
 100 effects can occur in the simulation, can already be demonstrated in the original protocol.

101 **Organization.** Section 2 presents an example that demonstrates why simulations should  
 102 preserve hyperproperties. Sections 3–5 describe the model and introduces important defini-  
 103 tions and notations. Section 6 formally defines block DAGs. Our results are presented and  
 104 proved in Section 7. The relation of our simulation to [17], and some applications appear  
 105 in Section 8. We summarize with future work, in Section 9.

## 106 2 Motivating Example

107 We describe a class of protocols solving *Binary Crusader Agreement*, and a hyperproperty  
 108 about them, called *binding* [1], which is assumed when such protocols are used to solve  
 109 randomized consensus. This motivates the need for establishing a notion of refinement  
 110 for block DAG simulations that is stronger than trace inclusion and which enables the  
 111 preservation of such hyperproperties.

### 112 2.1 Randomized consensus based on Binary Crusader Agreement

113 Let us consider the consensus protocol listed in Algorithm 1 (from [1]). This is a randomized  
 114 protocol based on two sub-protocols, Binary Crusader Agreement, invoked as BCA, and a

115 common coin, invoked via `Toss`. Every process participating in this consensus protocol goes  
 116 through a sequence of asynchronous rounds (the current round is stored in the variable  $r$ ),  
 117 and each round consists of one instance of `BCA` followed by one instance of `Toss`. We prefix  
 118 invocations with the value of  $r$  in order to emphasize that these instances are different from  
 119 one round to another.

120 *Binary Crusader Agreement* [7] is a weak form of consensus, where processes start with a  
 121 value in  $\{0, 1\}$  and can return a value in  $\{0, 1, \perp\}$  (note the special value  $\perp$ ). The requirements  
 122 are: (1) validity: if all non-faulty processes start with the same input, then this is the only  
 123 output, (2) agreement: no two non-faulty processes output two distinct non- $\perp$  values, and  
 124 (3) termination: every non-faulty process eventually outputs a value. It is weaker than  
 125 consensus because a process can output the “don’t know” value  $\perp$  instead of one of the inputs.  
 126 The common coin protocol allows to implement a shared source of *uniform* randomness, it  
 127 guarantees that all processes receive the same output in  $\{0, 1\}$  (drawn with equal probability)  
 128 and that this output is unpredictable to an outsider (adversary).

129 Each round of the consensus protocol starts with a round of `BCA` where each process  
 130 inputs the current estimation of the agreement value  $est$  (initially, this is the input  $x$ ),  
 131 followed by a round of the common coin. If `BCA` returns a non- $\perp$  value then this will be  
 132 the value of  $est$  in the next round. Otherwise, the value of  $est$  is the value returned by the  
 133 coin protocol. Furthermore, if the values returned by `BCA` and `Toss` are the same, then the  
 134 process outputs the decision value. A process continues running the protocol after outputting  
 135 the decision in order to “help” other processes reach a decision (e.g., so that future instances  
 136 of `BCA` and the common coin satisfy honest super majority assumptions).

## 137 2.2 Termination under binding

138 We say that the protocol *terminates* when all non-faulty processes output a decision. It has  
 139 been shown [1] that the protocol of Figure 1 terminates against an adaptive adversary with  
 140 probability 1, provided that `BCA` satisfies a property called *binding*. The binding property  
 141 states that for every execution prefix of `BCA` that ends with a process returning  $\perp$ , there is  
 142 a *single* non- $\perp$  value that can be returned by a process in *any* future extension of this prefix.  
 143 It is important to note that this is an instance of a *hyperproperty* because it characterizes  
 144 *sets* of executions, i.e., all possible extensions of a prefix, instead of individual executions as  
 145 in standard safety or liveness properties.

146 To explain the usefulness of binding, we use the execution template on the right of  
 147 Figure 1. This defines non-terminating executions of the consensus protocol against a specific  
 148 adaptive adversary assuming a “worst-case” `BCA` protocol, which satisfies the specification  
 149 described in Section 2.1 but does *not* satisfy binding. Therefore, assuming two processes  
 150 with different inputs, for every round  $r$ , the adversary schedules `BCA` so that a first process  
 151 returns  $\perp$  and the second process’s return value is not yet fixed. Then, it schedules the first  
 152 process to get a value  $c_r \in \{0, 1\}$  from the common coin and after observing this value, it  
 153 resumes `BCA` so that the second process gets the value  $1 - c_r$  (this is admitted by the `BCA`  
 154 specification). The conditional at lines 6–12 implies that the first process will enter the next  
 155 round with  $est$  being the outcome of the coin toss, and the second process with  $est$  being the  
 156 value returned by `BCA`. Therefore, they enter the next round with different estimations of  
 157 the agreement value, and the same can be repeated infinitely often. Since this repeats for all  
 158 possible outcomes of the coin tosses, non-termination happens with probability 1.

159 Note that this would not be possible for both outcomes  $c_r \in \{0, 1\}$  of the coin toss if  
 160 `BCA` satisfies binding. Indeed, after the first process gets  $\perp$  from `BCA` (and before the coin  
 161 toss), the value returned by `BCA` to the second process is *fixed* in *any* possible extension, i.e.,

162 it is the same no matter the outcome of the coin toss. Therefore, for one of the two possible  
 163 outcomes of the coin toss, this return value equals that outcome, and the two processes will  
 164 enter with equal values of *est* in the next round.

165 When binding holds, an adaptive adversary can *not* impose the schedule described above  
 166 and the protocol terminates with probability 1. In every round, if the BCA value is not  $\perp$ ,  
 167 then it equals the outcome of the coin toss with probability  $1/2$ , which leads to outputting a  
 168 decision. If all processes get  $\perp$  from BCA, then the common coin leads directly to agreement.  
 169 Therefore, the protocol terminates within a constant expected number of rounds.

## 170 2.3 Preserving binding

171 In the context of this consensus protocol, we discuss the possibility of replacing a given BCA  
 172 protocol with a block DAG simulation as defined by Schett and Danezis [17]. The results  
 173 in [17] are not sufficient to deduce that the block DAG simulation satisfies binding if the  
 174 original protocol did, because, as mentioned above, binding is an instance of a hyper-property  
 175 and hyper-properties are not preserved by standard trace inclusion [2]. Therefore, based on  
 176 the results in [17], the proof of termination that assumed binding is not applicable to the  
 177 block DAG simulation.

178 In this work, we present a block DAG simulation that handles protocols that use public-  
 179 coins and shared objects (including a common coin like *Toss*). We establish that it is a  
 180 *forward simulation*, which by previous work [2], implies that the set of traces defined by an  
 181 adaptive adversary of the consensus protocol with the original BCA protocol is the same  
 182 when the latter is replaced with the block DAG simulation (the results in [2] were applied in  
 183 the context of concurrent objects and programs using such objects, but they are stated in  
 184 terms of LTSs models of such programs and apply more generally to distributed protocols  
 185 as well). Therefore, if one satisfies binding, then the other one satisfies it as well. This is  
 186 enough to conclude that the termination argument used for the original protocol holds for  
 187 the block DAG simulation as well.

## 188 3 Preliminaries

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

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

- 195 ■ A **compute**( $i$ ) event lets process  $p_i$  receive *all* the messages in the buffers  $In_{j \rightarrow i}$ , as well as  
 196 the requests in  $Rqsts_i$ , and update the local state  $PS_i$ . The local computation performed  
 197 to update  $PS_i$  may result in new messages being deposited in the outgoing buffers  $Out_{i \rightarrow j}$   
 198 and indications being sent to the user.
- 199 ■ A **deliver**( $i, j$ ) event moves the *oldest* message in  $Out_{i \rightarrow j}$  to  $In_{j \rightarrow i}$ .

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

206 In a randomized protocol, the local computation of a process can depend on the result of  
 207 local coin flips. To model this, we assume each process  $p_i$  has access to a random *tape*, from  
 208 which it can draw a random string at each `compute( $i$ )` event. Our simulation can be applied  
 209 to *public-coin* protocols, which are randomized protocols that do not require processes to keep  
 210 secrets, i.e., they can broadcast the random string they draw as soon as they use it. This  
 211 definition captures protocols in the full-information model such as [12].

212 To allow for easy composition, we define *shared objects*. A shared object is an implementa-  
 213 tion of an interface that is accessible by all processes. For example, in the context of the  
 214 randomized consensus protocol in Fig. 1 we used a shared object called *common coin* with  
 215 a method `Toss`. For any shared object  $o$ , each process  $p_i$  can invoke  $o$  as it performs any  
 216 local computation. Invocations are non-blocking, and  $o$  may at any point return a value in a  
 217 designated buffer  $o.buff_i$ . Whenever a `compute( $i$ )` event is scheduled, the contents of  $o.buff_i$   
 218 are dequeued and may affect the local computation.

## 219 4 Modeling protocols with Labeled Transition Systems

220 We model a protocol as a *Labeled Transition System (LTS)*, which is a tuple  $L = (Q, \Sigma, q_{start}, \delta)$   
 221 where:

- 222 1.  $Q$  is a (possibly infinite) set of states.
- 223 2.  $\Sigma$  is a set of (transition) labels.
- 224 3.  $q_{start}$  is the starting state.
- 225 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  
 226  $(q_1, l, q_2) \in Q \times \Sigma \times Q$ .

227 An execution of  $L$  is an alternating sequence of states and transition labels  $\alpha = q_0, l_0, q_1, l_1, \dots$   
 228 s.t.  $q_i \xrightarrow{l_i} q_{i+1}$  for any  $i \geq 0$ . If there exists any partial execution  $q_i, l_i, \dots, l_{j-1}, q_j$  then  
 229 we write  $q_i \xrightarrow{l_i, \dots, l_{j-1}} q_j$ . We define a subset of labels  $\Sigma_E \subseteq \Sigma$  as the *external actions*,  
 230 and define a *trace* of  $L$  to be the projection of an execution over  $\Sigma_E$ . Typically, external  
 231 actions correspond to requests and indications in the interface of a protocol, and define the  
 232 “observable” behavior of a protocol. For instance, the external actions of a consensus protocol  
 233 are about setting the input of each process and outputting their decisions.

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

238 Randomized protocols can be modeled using an extension of LTSs called (*simple*) *probabilistic automata* [18] where a transition from a state  $q$  leads to a probability distribution over  
 239 states instead of a single state. The semantics of a probabilistic automaton is formalized in  
 240 terms of *probabilistic executions*, which are probability distributions over executions defined by  
 241 a deterministic scheduler that resolves the non-determinism. *Probabilistic traces* are defined  
 242 as projections of probabilistic executions to external actions (similarly to the non-probabilistic  
 243 case). The deterministic scheduler corresponds to the notion of adaptive adversary described  
 244 above which controls message delivery and process scheduling. To simplify the formalization,  
 245 we model randomized protocols using LTSs instead of probabilistic automata by including  
 246 results of random choices in the transition labels. The transition labels corresponding to  
 247 random choices are defined as external actions. The relevance of this modeling choice will be  
 248 detailed later when discussing forward simulations.

249 Let  $\mathcal{P}$  be a public-coin protocol and  $\mathcal{O}$  be a *set* of shared objects used by  $\mathcal{P}$ . We define the  
 250 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  
 251

252 incoming messages  $(In_{j \rightarrow i})_{j \in [n]}$ , the outgoing messages  $(Out_{i \rightarrow j})_{j \in [n]}$  and the incoming object  
 253 return values  $(o.buff_i)_{o \in \mathcal{O}}$  of each process  $p_i$ . For convenience, we assume that incoming user  
 254 requests are stored in  $In_{i \rightarrow i}$  and outgoing user indications are stored in  $Out_{i \rightarrow i}$ . Overall,  
 255  $q = (PS_i, (In_{j \rightarrow i})_{j \in [n]}, (Out_{i \rightarrow j})_{j \in [n]}, (o.buff_i)_{o \in \mathcal{O}})_{i \in [n]}$ . We use register notation to refer to  
 256 the components of each state, e.g.,  $q.In_{j \rightarrow i}$  refers to the incoming messages buffer from  $j$  to  
 257  $i$  in the state  $q$ . In the initial state  $q_{start}$ , all of the processes have the initial local state and  
 258 all of the message buffers are empty. For the consensus protocol in Fig. 1, local states are  
 259 valuations of  $r$ ,  $val$ ,  $c$ , and  $est$ , and the buffer for incoming object return values will contain  
 260 values returned by **Toss**. User indications are decision values outputted at line 7.

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

- 267 1. **compute** $(i, \rho)$  denotes a transition where process  $p_i$  performs a local computation with  $\rho$   
 268 as its randomness. For the consensus protocol in Fig. 1, a local computation step would  
 269 consist in assigning a value to  $est$  depending on the conditions starting with line 6.
- 270 2. **deliver** $(i \rightarrow j)$  denotes a transition where all messages in  $Out_{i \rightarrow j}$  are moved to  $In_{i \rightarrow j}$ .
- 271 3. **o.indicate** $(i, w)$  denotes a transition where the value  $w$  has been added to  $o.buff_i$ . In  
 272 Fig. 1, this would correspond to the common coin object returning a value for **Toss**.
- 273 4. **request** $(i, x)$  denotes a transition where process  $p_i$  receives  $x$  as input. In Fig. 1, this  
 274 models a process receiving an input value to use in the consensus protocol.
- 275 5. **indicate** $(i, y)$  denotes a transition where process  $p_i$  returns  $y$  as output. In Fig. 1, this  
 276 corresponds to the output at line 7.

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

## 282 5 Forward simulations

283 Showing that a block DAG protocol is a “correct” simulation of some other protocol relies on  
 284 the notion of *forward simulation* between the LTSs modeling the two protocols, respectively.

285 ► **Definition 1** (forward simulation). *Let  $L = (Q, \Sigma, q_{start}, \delta)$  and  $L' = (Q', \Sigma', q'_{start}, \delta')$  be*  
 286 *two LTSs with the same set of external actions  $\Sigma_E$ . A relation  $R \subseteq Q \times Q'$  is a forward*  
 287 *simulation from  $L$  to  $L'$  if both of the following hold:*

- 288 ■  $(q_{start}, q'_{start}) \in R$
- 289 ■ 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:
  - 290 ■  $(q_2, q'_2) \in R$ ,
  - 291 ■  $q'_1 \xrightarrow{\sigma} q'_2$  is a partial execution of  $L'$  ( $\sigma$  is a sequence of labels in  $\Sigma'$ ), and
  - 292 ■ if  $l \in \Sigma_E$ , then the projection of the label sequence  $\sigma$  over  $\Sigma_E$  is exactly  $l$ .

293 When  $L$  is an LTS modeling a block DAG simulation of a deterministic protocol  $\mathcal{P}$  that  
 294 is modeled as an LTS  $L'$ , the existence of a forward simulation  $R$  from  $L$  to  $L'$  implies  
 295 that the set of traces of  $L$  is included in the set of traces of  $L'$  [15]. It also implies the  
 296 preservation of (hyper-)properties of *finite* probabilistic traces of randomized protocols when



297 some sub-protocol  $\mathcal{P}$  is replaced by a block DAG simulation of it [2] (a concrete example  
 298 was given in Section 2). If the forward simulation is *weak progressive* [8], i.e., there exists a  
 299 well-founded order such that if  $\sigma = \epsilon$  in Definition 1 then either  $q_2$  is smaller than  $q_1$  in this  
 300 order or there exists an infinite execution from  $q_2'$  with empty trace, then (hyper-)properties  
 301 of *infinite* probabilistic traces are also preserved.

302 These results extend to randomized protocols as well. Assuming that the random choices  
 303 follow the uniform distribution, a forward simulation would imply that any random choice in  
 304  $L$  is mimicked in precisely the same manner by  $L'$ . This is because the label of every step  
 305 that includes a random choice is an external action and the result of that random choice is  
 306 included in the label itself. This holds even for *non-uniform* random sampling as long as  
 307 probabilities are recorded in transition labels. More formally, it will imply the existence of a  
 308 *weak probabilistic simulation* which is known to imply that the probability distributions over  
 309 traces of  $L$  defined by a deterministic scheduler are included in the probability distributions  
 310 over traces of  $L'$  defined by a deterministic scheduler [18]. Moreover, it will also imply  
 311 the preservation of probability distributions over executions of programs that use the block  
 312 DAG simulation instead of the original protocol (this is a consequence of weak probabilistic  
 313 simulations being sound for the trace distribution precongruence [18]).

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

## 318 **6** Block DAGs

319 A *block* is the main type of message that is exchanged in DAG-based protocols and our block  
 320 DAG simulations. A block issued by some process  $p_i$  allows  $p_i$  to: (1) inject data into the  
 321 system, e.g., user inputs or shared object outputs, and (2) establish a dependency between  
 322 events of different processes. To that end, the main fields of a block  $B$  are the identity of the  
 323 issuing process  $B.p$ , injected data  $B.d$ , and references to other blocks  $B.preds$  (on which  $B$   
 324 directly depends). The reference of  $B$  is denoted by  $\text{ref}(B)$ .

325 We require that each reference must uniquely identify a specific block. One way to achieve  
 326 this is using *cryptographic collision resistant hash functions*: the reference  $\text{ref}(B)$  consists of  
 327 a hash of the block  $B$ . By the collision resistance of the hash function, it is infeasible for a  
 328 computationally bounded adversary (or correct processes) to issue two distinct blocks that  
 329 hash to the same value and this ensures that the reference identifies a unique block.

330 Since blocks are supposed to represent local computation, and local computation steps of  
 331 any one process are always totally ordered, then each block  $B$  must include one reference to  
 332 a parent block which we denote by  $B.parent$ , except for one *genesis* block for each process  
 333 which does not have a parent. In addition, all of the blocks issued by one honest process  
 334 should form a chain, i.e., a directed path that starts with the genesis block.

335 We define the *ancestors* of a block  $B$  to be all of the predecessors of  $B$ , and their  
 336 predecessors and so on; this set is denoted  $\text{ancestors}(B)$ .

337 A block  $B$  is *authentic* if it was issued by the process  $B.p$ . It is crucial to ensure the  
 338 authenticity of each block before allowing it into the system. Otherwise, faulty processes can  
 339 impersonate honest processes and sabotage safety properties. We can ensure authenticity by  
 340 using a *cryptographic digital signature scheme*. That is each process must sign each block it  
 341 issues, and other processes validate the block by checking the signature attached to it.

342 Ensuring that each individual block is authentic is not enough to ensure that only

343 authentic blocks enter the system. We should also require that a block depends only on  
 344 authentic blocks, that is  $\text{ancestors}(B)$  must all be authentic in order for  $B$  to enter. We say  
 345 that a block is *valid* if it is authentic and all of  $B.\text{preds}$  are valid. Note that this recursive  
 346 definition is equivalent to requiring  $\text{ancestors}(B)$  all be authentic. Following this discussion,  
 347 to ensure safety, only valid blocks would be considered by correct processes. When a process  
 348  $p_i$  validates a block  $B$ , we write  $\text{valid}(p_i, B)$ .

349 Each process  $p_i$  maintains a local DAG  $G_i$  consisting of the valid blocks that  $p_i$  receives  
 350 as nodes and includes a directed edge  $B' \rightarrow B$  if and only if  $B' \in B.\text{preds}$ . Note that we  
 351 need a mechanism for  $p_i$  to ensure that  $G_i$  is a DAG. A simple mechanism would be for  $p_i$  to  
 352 validate  $B$  only after it has validated  $B.\text{preds}$  and not validate multiple blocks “atomically”.  
 353 This alongside the fact that each reference identifies a unique block, would ensure that no  
 354 block in a directed cycle would ever be considered valid. Formally, a *Block DAG of a correct*  
 355 *process*  $p_i$  is a graph  $\mathcal{G} = (V_{\mathcal{G}}, E_{\mathcal{G}})$  such that

- 356 ■  $V_{\mathcal{G}} \subseteq \{B : \text{valid}(p_i, B)\}$ .
- 357 ■ If  $B \in V_{\mathcal{G}}$  then for all  $B' \in B.\text{preds}$  it holds that  $B' \in V_{\mathcal{G}}$ .
- 358 ■  $E_{\mathcal{G}} = \{(B', B) \in V_{\mathcal{G}} \times V_{\mathcal{G}} : B' \in B.\text{preds}\}$ .
- 359 ■  $\mathcal{G}$  is acyclic.

360 Observe that by the definition of  $\mathcal{G}$ , for every  $B \in V_{\mathcal{G}}$  it holds that  $\text{ancestors}(B) \subseteq V_{\mathcal{G}}$ . When  
 361  $B' \in \text{ancestors}(B)$ , we write  $\text{path}(B', B)$ .

## 362 7 Simulating Public-Coin Protocols That Use Shared Objects

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

371 Figure 2 describes how to simulate a public-coin protocol  $\mathcal{P}$  using the components  
 372 mentioned above. We refer to this protocol as the *block DAG simulation of  $\mathcal{P}$*  and denote it  
 373 by  $\text{BD}(\mathcal{P})$ . We allow  $\text{BD}(\mathcal{P})$  to access the same shared objects as  $\mathcal{P}$ .

### Simulation of Public-Coin Protocols on Block DAGs

From the perspective of process  $p_i$ , user requests go directly to  $Rqsts_i$ .

Initialize  $G_i = (V_i, E_i)$  with  $V_i = \{B_j\}_{j \in [n]}$  where  $B_j$  is a dummy genesis block for the process  $p_j$ . On every  $\text{compute}(i)$  event:

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

■ **Figure 2** The simulation algorithm for public-coin protocols

374 Interpreting the block DAG as an execution of  $\mathcal{P}$  is done using the  $\text{interpret}$  algorithm,  
 375 described in Section 7.1. This algorithm runs locally and involves no communication, yet

376 guarantees that if two correct processes are interpreting the same (partial) block DAG, then  
 377 their interpretations would be identical.

378 Maintaining the joint block DAG is done using the `genBlock` and `exchangeBlocks` al-  
 379 gorithms (discussed in Section 7.2): `genBlock` is responsible for creating new blocks and  
 380 `exchangeBlocks` is responsible for passing those blocks around to ensure that all correct  
 381 processes receive the same blocks even if the process that issued the block is corrupted.

382 The aforementioned components, together, ensure that correct processes have consistent  
 383 views of the execution of  $\mathcal{P}$  at all times. However, this does not guarantee that the execution  
 384 is useful, e.g., it might give the adversary more power or it might be a “liveless” execution  
 385 where the correct processes are not making any progress. For that reason, we prove in  
 386 Section 7.3 that the execution (defined by the views) is faithful in the sense that there exists  
 387 a forward simulation towards the original protocol. This guarantees that the simulation of  $\mathcal{P}$   
 388 on the block DAG preserves  $\mathcal{P}$ ’s original specification.

## 389 7.1 Common Interpretation

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

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

399 ► **Definition 2 (Block Interpretation).** *The interpretation of a block  $B$  has the following fields:*  
 400 1. A local process state  $B.PS$ .  
 401 2. A list of incoming messages  $B.M_{in}$ .  
 402 3. A list of outgoing messages  $B.M_{out}$ . For convenience, we denote by  $M_{out}[j]$  the outgoing  
 403 messages in  $M_{out}$  that are designated to  $p_j$ .

404 Note that the interpretation of a block is *not* sent over the network. This is crucial  
 405 because we do not want the size of the block sent over the network to increase with the  
 406 number of protocol instances being interpreted, and instead we only want the block to  
 407 include information that processes cannot locally compute unambiguously. As such, it is the  
 408 responsibility of each process to interpret each block it has locally.

409 In a regular execution of a *deterministic* protocol, whenever a `compute( $i$ )` event is  
 410 scheduled, the process  $p_i$  performs the following: it passes all of the message in  $In_{j \rightarrow i}$  to  
 411 the local state of its protocol instance  $PS_i$  and performs a local computation. This updates  
 412 the local state  $PS_i$ , produces new outgoing messages that are deposited into  $Out_{i \rightarrow j}$  and  
 413 may return user indications. Our interpretation protocol tries to mimic the execution by  
 414 assigning to  $B.PS$  the local state of the process after the corresponding event,  $B.M_{out}[j]$  the  
 415 messages that would be deposited in  $Out_{i \rightarrow j}$ , and  $B.M_{in}$  the messages that would have been  
 416 in  $In_{j \rightarrow i}$  before the event. In addition, if the block  $B$  was issued by the process doing the  
 417 interpretation and  $B.PS$  produces a user indication, then the process must actually return  
 418 the indication to the user. The way to compute  $B.PS$  is as follows:  $B.PS$  is initially copied  
 419 from the parent block (or initialized as an initial state for genesis blocks), and then we feed  
 420 it all of the relevant outgoing messages from the interpretation of the predecessor blocks,  
 421 that is all messages in  $B'.M_{out}[i]$  for all  $B' \in B.preds$ , where  $B.p = p_i$ .

■ **Algorithm 2**  $\text{interpret}(G_i, \mathcal{P})$  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

- 1: **while**  $\exists B \in G_i$  s.t.  $B$  is not interpreted s.t.  $\forall B' \in B.preds : B'$  is interpreted **do**
- 2:     **if**  $B.k = 0$  **then**
- 3:         Initialize  $B.PS$  as a new state according to the protocol  $\mathcal{P}$  and process  $B.p$
- 4:     **else**
- 5:          $B.PS := B.parent.PS$
- 6:     **for all**  $B' \in B.preds$  **do**
- 7:         Copy messages from  $B'.M_{out}[B.p]$  to  $B.M_{in}$
- 8:     Pass the user requests  $B.rqsts$ , messages  $B.M_{in}$ , random tape  $B.rand$  and the object indications  $B.buff$  to the state  $B.PS$
- 9:     Overwrite the new state in  $B.PS$
- 10:    Store the outgoing messages in  $B.M_{out}$
- 11:    **if**  $B.p = i$  **then**
- 12:        Return user indications produced by  $B.PS$  to the user
- 13:        Perform object invocations as dictated by  $B.PS$

---

422     When extending this approach to *randomized* protocols, we need to account for the local  
 423 randomness. In this case, the process state expects to additionally receive a random tape. It  
 424 is the responsibility of the issuing process to include the tape in the block  $B$  and attach it as  
 425 a part of the block in a data field  $B.rand$ . The interpretation is thus similar to that of a  
 426 deterministic protocol, but  $B.rand$  is now also passed to the process state as randomness.

427     When further extending this to protocols with *shared objects*, we need to handle object  
 428 invocations and object indications. In a regular execution of a protocol with a shared objects  
 429  $\circ$ , a process  $p_i$  might invoke  $\circ$  following a  $\text{compute}(i)$  event. Similarly, when interpreting a  
 430 block,  $B.PS$  might dictate that  $B.p$  should invoke  $\circ$ . In this case, the interpreting process  
 431  $p_i$  actually performs the invocation only if it is the issuing process of the block  $p_i = B.p$ .  
 432 The process states in the original protocol expect to receive indications from  $\circ$ , so these  
 433 indications should be passed to  $B.PS$  when interpreting  $B$ . When  $\circ$  returns an indication  
 434 to  $p_i$ , it is the responsibility of  $p_i$  to attach the indications to the block in a special buffer  
 435  $B.buff[\circ]$ . The contents of  $B.buff[\circ]$  are passed to  $B.PS$  when interpreting  $B$ . This concludes  
 436 the high level description of block interpretation. In order to interpret an entire block DAG,  
 437 we interpret blocks in a topological order since the interpretation of each block  $B$  depends  
 438 on the interpretation of its predecessors. Since the graph is a DAG, such an order exists and  
 439 every block can be interpreted. The full algorithm  $\text{interpret}(\mathcal{G}, \mathcal{P})$  is presented in Algorithm 2.  
 440 The main guarantee of  $\text{interpret}(\mathcal{G}, \mathcal{P})$  is the fact that the interpretation of  $B$  is independent  
 441 of  $\mathcal{G}$ . This is formalized in the following lemma (proved in the full version [3]):

442 ► **Lemma 3.** *For any two block DAGs  $G_1$  and  $G_2$ , if  $B \in G_1$  and  $B \in G_2$  then the*  
 443 *interpretation of  $B$  in both  $\text{interpret}(G_1, \mathcal{P})$  and  $\text{interpret}(G_2, \mathcal{P})$  is identical.*

## 444 7.2 Joint Block DAG

445 In this section, we demonstrate how processes build and maintain the block DAGs that they  
 446 interpret in Section 7.1. Algorithm 3 presents the  $\text{genBlock}(G_i)$  algorithm, which allows a  
 447 process to generate blocks and inject data into the system.

■ **Algorithm 3** `genBlock( $G_i$ )` for process  $p_i$

---

$G_i = (V_i, E_i)$  is a block DAG.

- 1: Initialize a new block  $B$  as follows  $B.p := p_i, B.preds := \emptyset, B.rqsts := \emptyset$
  - 2: Assign to  $B.parent$  the reference of the most recent block in  $G_i$  issued by  $p_i$ .
  - 3:  $B.k := B.parent.k + 1$
  - 4: **for all**  $B' \in V_i$  s.t.  $\neg \text{path}(B', B.parent)$  **do**
  - 5:      $B.preds := B.preds \cup \{\text{ref}(B')\}$
  - 6: Fill the external data `fillData( $B$ )`.
  - 7: **return**  $B$
- 

448     The algorithm gets a block DAG  $G_i$  which is assumed to be a valid block DAG of  $p_i$ .  
 449     It then generates a new block  $B$  and assigns it a parent from  $G_i$ , then adds to  $B.preds$  all  
 450     references to blocks in  $G_i$  that do not have a path to  $B.parent$ . Note that since  $B.preds \subseteq V_i$ ,  
 451     then  $B.pred$  only includes blocks  $B'$  s.t.  $\text{valid}(p_i, B')$ . This guarantees that  $B$  is a valid  
 452     block. Next the external data is filled into the block: this includes moving the user requests  
 453     from  $Rqsts_i$  to  $B.rqsts$ , moving the object indications from  $o.buff_i$  to  $B.buff[o]$  for each  
 454     relevant  $o \in \mathcal{O}$  and finally assigning a random string  $\rho$  to  $B.rand$ . Note that we do not know  
 455     exactly how long  $\rho$  needs to be until  $B$  is actually interpreted. Since all  $B' \in B.preds$  are  
 456     already in  $G_i$ , process  $p_i$  can already interpret  $B$  and generate  $\rho$  while generating  $B$ .

457     Next, we describe the communication component that is responsible for exchanging blocks  
 458     and growing the DAGs. We have shown that processes that interpret the same blocks reach  
 459     the same conclusion. But for this to be useful, the communication component must ensure  
 460     correct processes eventually interpret the same blocks. That is, if a correct process  $p_i$  adds  
 461     some  $B$  to  $G_i$ , then every correct process  $p_j$  eventually adds  $B$  to  $G_j$ . This can be viewed as  
 462     a consistency property between two processes.

463     Note that a naive approach of having each process simply send its blocks to everyone  
 464     does not guarantee consistency, since an honest process  $p_i$  may add a block  $B^*$  by some  
 465     corrupted process  $B^*$  as a predecessor for its own block  $B$ .  $p_i$  naturally considers  $B$  valid  
 466     and adds it to its block DAG, but for any other honest process  $p_j$ ,  $B$  will never be considered  
 467     valid until it receives  $B^*$  from  $p^*$ .

468     Consistency can be achieved using a simple *echoing* mechanism that we describe now. For  
 469     each block  $B$  that  $p_i$  issues using `genBlock`,  $p_i$  generates a signature for  $B$  which we denote  
 470     by  $B.\sigma$ , and sends  $(B, B.\sigma)$  to everyone. When  $p_i$  receives a block  $B$  by some other process,  
 471     it first ensures  $B$  is authentic (by verifying the signature). After collecting all authentic  
 472     blocks,  $p_i$  tries to validate as many of them as possible. The validation may only fail if some  
 473      $B' \in B.preds$  of  $B$  is missing, so  $p_i$  requests  $B'$  from the process  $B.p$  that issued  $B$ , using  
 474     a forward request message which we denote by `FWD(ref( $B'$ ))`. The idea is that if  $B.p$  is  
 475     correct then it must have those blocks, so it will eventually send them to  $p_i$ , allowing  $p_i$   
 476     to validate the block  $B.p$ . Finally,  $p_i$  of course has to respond to the forward requests it  
 477     has received. This concludes the informal description of `exchangeBlocks`. The consistency  
 478     guarantee is formalized in the following lemma:

479     ► **Lemma 4.** *For any two correct processes  $p_i$  and  $p_j$  executing the protocol of Figure 2, if*  
 480      *$p_i$  adds a block to its block DAG  $G_i$ , then  $p_j$  eventually inserts  $B$  into  $G_j$ .*

481     We note that Lemma 4 really refer to any protocol in which Algorithms 3 and 4 are  
 482     continuously run, and are not specific to Figure 2. The proof is deferred to the full version [3].

■ **Algorithm 4** `exchangeBlocks( $G_i$ )` for process  $p_i$

---

$G_i = (V_i, E_i)$  is a block DAG  
 $toValidate$  and  $isSent$  are process-local variables that maintain their values across different invocations  
Initialize  $toValidate := \emptyset$  and  $isSent := \emptyset$

- 1:
- 2: **for all**  $B \in G_i$  s.t.  $B.p = p_i$  and  $B \notin isSent$  **do**
- 3:     Sign  $B$  and denote the signature by  $B.\sigma$
- 4:     Send  $(B, B.\sigma)$  to everyone
- 5: Move all authentic blocks from all  $In_{j \rightarrow i}$  to a set  $auth$
- 6:  $toValidate := toValidate \cup auth$  ▷ Throw inauthentic blocks
- 7: **while**  $\exists B \in toValidate$  s.t.  $valid(p_i, B)$  **do**
- 8:      $G_i.insert(B)$
- 9:      $toValidate := toValidate \setminus \{B\}$
- 10:     $auth := auth \setminus \{B\}$
- 11: **for all**  $B \in auth$  **do** ▷ Try to validate all authentic blocks
- 12:     **for all**  $B' \in B.preds$  s.t.  $B' \notin G_i$  **do**
- 13:         Send `FWD(ref( $B'$ ))` to  $B.p$  ▷ Request missing blocks from  $B.p$
- 14:     **for all** `FWD(ref( $B'$ ))` in some  $In_{j \rightarrow i}$  **do** ▷ Respond to forward requests
- 15:         If  $B' \in G_i$ , send  $(B', B'.\sigma)$  to  $p_j$
- 16: Empty all  $In_{j \rightarrow i}$ .

---

### 7.3 Correctness Proof

Combining Lemma 4 with Lemma 3 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 is 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 3. By Lemma 4, 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 3, eventually  $p_j$  creates a new block  $B'$  such that  $B \in B'.preds$  and by Algorithm 2,  $m$  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. This guarantees the liveness of the block DAG simulation.

We show that the block DAG simulation of a protocol  $\mathcal{P}$  is faithful in the sense that there exists a *forward simulation* from the block DAG simulation denoted as  $BD(\mathcal{P})$  to  $\mathcal{P}$  (modeled as LTSs). As mentioned in Section 5, this implies that the block DAG simulation inherits finite-trace probability distributions of  $\mathcal{P}$  and that typical specifications of programs using the DAG simulation instead of  $\mathcal{P}$  are preserved.

Section 3 describes the modeling of  $\mathcal{P}$  using LTSs. We describe below a modeling of  $BD(\mathcal{P})$  using an LTS  $L' = (Q', \Sigma', q'_{start}, \delta')$  which simplifies the forward simulation proof. A state  $q' \in Q'$  contains the block DAG  $G_i$  of each process  $p_i$  and  $(In_{j \rightarrow i}^B)_{j \in [n]}$  and  $(Out_{i \rightarrow j}^B)_{j \in [n]}$  for each process  $p_i$ , where  $In_{j \rightarrow i}^B$  is the incoming buffer of process  $i$  with blocks sent by process  $j$  and  $Out_{i \rightarrow j}^B$  is the outgoing buffer with blocks sent by  $i$  to  $j$ . As before, we assume that incoming user requests are stored in  $In_{i \rightarrow i}^B$  and outgoing user indications are stored in  $Out_{i \rightarrow i}^B$ . The shared object indications are stored in separate buffers  $(o.buff_i)_{o \in \mathcal{O}}$  as before.



Overall,  $q' = (G_i, (In_{j \rightarrow i}^B)_{j \in [n]}, (Out_{i \rightarrow j}^B)_{j \in [n]}(o.buff_i)_{o \in \mathcal{O}})_{i \in [n]}$ . 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 3 we decompose a compute step of a process as defined in Figure 2 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`( $i, \rho$ ) denotes a transition where process  $p_i$  produces and disseminates a new block (inside the `genBlock` algorithm) with  $\rho$  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`( $i \rightarrow j$ ) is a transition where all the blocks in  $Out_{i \rightarrow j}^B$  are moved to  $In_{i \rightarrow j}^B$ .
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`( $i, y$ )) are used as in Section 3.

The external actions  $\Sigma_E$  are defined exactly as for the LTS  $L$  modeling  $\mathcal{P}$ , presented in Section 3 ( $\Sigma_E$  includes `request`( $i, x$ ), `indicate`( $i, y$ ), and `compute`( $i, \rho$ )). A transition  $(q'_1, e, q'_2) \in Q' \times \Sigma' \times Q'$  (denoted  $q'_1 \xrightarrow{e} q'_2$ ) is in  $\delta'$  if and only if the protocol  $\text{BD}(\mathcal{P})$  can get from state  $q'_1$  to state  $q'_2$  by executing the step denoted by the label  $e$ . Theorem 5 is proved in the full version [3].

► **Theorem 5.** *There exists a forward simulation from the LTS  $L'$  modeling  $\text{BD}(\mathcal{P})$  to the LTS  $L$  modeling  $\mathcal{P}$ .*

## 8 Relation to Prior Work

**Comparison with the deterministic simulation.** We can now discuss how our simulation and proof are related to the work of Schett and Danezis [17]. 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 necessary because their techniques do not capture the probabilistic guarantees of randomized protocols.

Our network component which consists of `genBlock` and `exchangeBlocks` algorithms is a natural extension of the `gossip` algorithm of [17]. Indeed, the code responsible for generating new blocks and echoing them is almost identical to that of `gossip`. The difference is that because we want to exchange only blocks, they should carry enough information to resolve the randomized decisions that can come from local randomness or shared objects. 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. Lemma 4 is proved in a manner similar to [17, Lemma 3.7].

Our interpretation algorithm is the natural extension of `interpret` algorithm of [17] 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

554 fields that were already attached to each block by our `genBlock`. Lemma 3 that states the  
555 common interpretation of block DAGs, is analogous to [17, Lemma 4.2]. However, the proof  
556 of the latter had a minor mistake and our proof is slightly different.

557 Finally, the guarantees of randomized protocols, unlike those of deterministic protocols,  
558 cannot always be expressed as trace properties. Particularly, for our simulation to be faithful  
559 to the original protocol, we need a more careful and precise statement and proof. Therefore,  
560 the modeling in Sections 3 and 7.3 as well as the proof of Theorem 5 are totally different  
561 from what appears in [17].

562 **Analyzing existing protocols.** Several recent works rely on the block DAG approach, e.g.,  
563 Aleph [10], DAG-Rider [13] and Bullshark [20]. All of these protocols are randomized. While  
564 each of these works presents a new protocol, we provide a formal and systematic framework  
565 for analyzing DAG-based protocols, especially *randomized* block DAG protocols.

566 Here we discuss how our simulation applies to existing protocols, concentrating on  
567 Aleph [10] and DAG-Rider [13]. These protocols aim to order the blocks of the DAG, so  
568 as to implement *Byzantine Atomic Broadcast* (BAB). A BAB protocol allows all processes  
569 to receive the same messages in the *same order*. One natural way of implementing a BAB  
570 protocol using a block DAG is by having each process attach the messages it wants to  
571 broadcast to a block and then broadcast the block to everyone. The processes then just need  
572 to agree on an order of the blocks, which would induce an order of the messages.

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

577 Aleph orders the blocks in the DAG by electing a leader block in each round, and then  
578 having that leader block (deterministically) dictate the order of its ancestor blocks that have  
579 not been ordered yet.

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

589 Aleph and DAG-Rider can be analyzed using our framework. The consensus protocol  
590 used can be analyzed independently of Aleph or DAG-Rider, while assuming it has access  
591 to a common coin. By Theorem 5, the simulation of the consensus protocol on the block  
592 DAG is faithful to the original consensus protocol. This not only simplifies reasoning about  
593 safety and liveness of Aleph and DAG-Rider, but also supports *modularity*: the simulated  
594 consensus protocol in Aleph or DAG-Rider can be seamlessly replaced using Theorem 5.

## 595 **9 Discussion**

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



599 BFT protocol, and in particular, their probability distributions.

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

611 Our simulation relies on the fact that it is safe to reveal the randomness to the adversary  
 612 as soon as it is used. We can similarly define *private-coin* protocols, whose security relies  
 613 on processes ability to keep secrets from the adversary. A classical example would be any  
 614 Asynchronous Verifiable Secret Sharing scheme (e.g. [5]). From a theoretical point of view, it  
 615 would be interesting to demonstrate how we can simulate such algorithms on block DAGs.  
 616 However, we note that some protocols are entirely public-coin other than a dedicated private-  
 617 coin sub-protocol, such as Aleph-Beacon in Aleph [10] (which is used to implement a common  
 618 coin). In this case, the dedicated sub-protocol can be encapsulated as a shared object, thus  
 619 factoring out the use of private-coin simulations.

## 620 — References —

- 621 1 Ittai Abraham, Naama Ben-David, and Sravya Yandamuri. Efficient and adaptively secure  
 622 asynchronous binary agreement via binding crusader agreement. In Alessia Milani and Philipp  
 623 Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno,  
 624 Italy, July 25 - 29, 2022*, pages 381–391. ACM, 2022. doi:10.1145/3519270.3538426.
- 625 2 Hagit Attiya and Constantin Enea. Putting strong linearizability in context: Preserving  
 626 hyperproperties in programsthat use concurrent objects. In Jukka Suomela, editor, *33rd  
 627 International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Bu-  
 628 dapest, Hungary*, volume 146 of *LIPICs*, pages 2:1–2:17. Schloss Dagstuhl - Leibniz-Zentrum  
 629 für Informatik, 2019. doi:10.4230/LIPICs.DISC.2019.2.
- 630 3 Hagit Attiya, Constantin Enea, and Shafik Nassar. Faithful simulation of randomized bft  
 631 protocols on block dags. Cryptology ePrint Archive, Paper 2023/192, 2023. [https://eprint.  
 632 iacr.org/2023/192](https://eprint.iacr.org/2023/192). URL: <https://eprint.iacr.org/2023/192>.
- 633 4 Leemon Baird. The Swirls Hashgraph consensus algorithm: Fair,  
 634 fast, Byzantine fault tolerance. [https://www.researchhub.com/paper/337/  
 635 the-swirls-hashgraph-consensus-algorithm-fair-fast-byzantine-fault-tolerance](https://www.researchhub.com/paper/337/the-swirls-hashgraph-consensus-algorithm-fair-fast-byzantine-fault-tolerance),  
 636 2016.
- 637 5 Ran Canetti and Tal Rabin. Fast asynchronous Byzantine agreement with optimal resilience.  
 638 In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *Proceedings of the Twenty-  
 639 Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA,  
 640 USA*, pages 42–51. ACM, 1993. doi:10.1145/167088.167105.
- 641 6 Anton Churymov. Byteball: A decentralized system for storage and transfer of value.  
 642 <https://byteball.org/Byteball.pdf>, 2016.
- 643 7 Danny Dolev. The Byzantine generals strike again. *J. Algorithms*, 3(1):14–30, 1982. doi:  
 644 10.1016/0196-6774(82)90004-9.
- 645 8 Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. Weak progressive forward simulation  
 646 is necessary and sufficient for strong observational refinement. In Bartek Klin, Slawomir Lasota,  
 647 and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory, CONCUR*

- 648 2022, September 12-16, 2022, Warsaw, Poland, volume 243 of *LIPIcs*, pages 31:1–31:23. Schloss  
649 Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CONCUR.2022.31.
- 650 9 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus  
651 with one faulty process. In Ronald Fagin and Philip A. Bernstein, editors, *Proceedings of  
652 the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March  
653 21-23, 1983, Colony Square Hotel, Atlanta, Georgia, USA*, pages 1–7. ACM, 1983. doi:  
654 10.1145/588058.588060.
- 655 10 Adam Gagol, Damian Lesniak, Damian Straszak, and Michal Swietek. Aleph: Efficient atomic  
656 broadcast in asynchronous networks with Byzantine nodes. In *Proceedings of the 1st ACM  
657 Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October  
658 21-23, 2019*, pages 214–228. ACM, 2019. doi:10.1145/3318041.3355467.
- 659 11 Wojciech M. Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not  
660 suffice for randomized distributed computation. In Lance Fortnow and Salil P. Vadhan, editors,  
661 *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose,  
662 CA, USA, 6-8 June 2011*, pages 373–382. ACM, 2011. doi:10.1145/1993636.1993687.
- 663 12 Shang-En Huang, Seth Pettie, and Leqi Zhu. Byzantine agreement in polynomial time with  
664 near-optimal resilience. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th  
665 Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*,  
666 pages 502–514. ACM, 2022. doi:10.1145/3519935.3520015.
- 667 13 Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need  
668 is DAG. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21:  
669 ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30,  
670 2021*, pages 165–175. ACM, 2021. doi:10.1145/3465084.3467905.
- 671 14 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun.  
672 ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.
- 673 15 Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. untimed  
674 systems. *Inf. Comput.*, 121(2):214–233, 1995. doi:10.1006/inco.1995.1134.
- 675 16 Sean Rowan and Nairi Usher. The Flare consensus protocol: Fair, fast federated Byzantine  
676 agreement consensus. [https://flareportal.com/wp-content/uploads/simple-file-list/  
677 FCP.pdf](https://flareportal.com/wp-content/uploads/simple-file-list/FCP.pdf), 2019.
- 678 17 Maria Anna Schett and George Danezis. Embedding a deterministic BFT protocol in a block  
679 DAG. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21:  
680 ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30,  
681 2021*, pages 177–186. ACM, 2021. doi:10.1145/3465084.3467930.
- 682 18 Roberto Segala. A compositional trace-based semantics for probabilistic automata. In Insup  
683 Lee and Scott A. Smolka, editors, *CONCUR '95: Concurrency Theory, 6th International  
684 Conference, Philadelphia, PA, USA, August 21-24, 1995, Proceedings*, volume 962 of *Lecture  
685 Notes in Computer Science*, pages 234–248. Springer, 1995. doi:10.1007/3-540-60218-6\_17.
- 686 19 Yonatan Sompolinsky, Shai Wyborski, and Aviv Zohar. PHANTOM GHOSTDAG: a scalable  
687 generalization of nakamoto consensus: September 2, 2021. In Foteini Baldimtsi and Tim  
688 Roughgarden, editors, *AFT '21: 3rd ACM Conference on Advances in Financial Technologies,  
689 Arlington, Virginia, USA, September 26 - 28, 2021*, pages 57–70. ACM, 2021. doi:10.1145/  
690 3479722.3480990.
- 691 20 Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bull-  
692 shark: DAG BFT protocols made practical. In Heng Yin, Angelos Stavrou, Cas Cremers,  
693 and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and  
694 Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages  
695 2705–2718. ACM, 2022. doi:10.1145/3548606.3559361.
- 696 21 Qin Wang, Jiangshan Yu, Shiping Chen, and Yang Xiang. SoK: Diving into DAG-based  
697 blockchain systems. *CoRR*, abs/2012.06128, 2020. URL: <https://arxiv.org/abs/2012.06128>,  
698 arXiv:2012.06128.