Reusable Formal Verification of DAG-based Consensus Protocols

Nathalie Bertrand[1][0000−0002−9957−5394], Pranav Ghorpade[2][0009−0001−0421−4490], Sasha Rubin[2][0000−0002−3948−129X], Bernhard Scholz[3][0000−0002−7672−7359], and Pavle Subotić[3][0000−0002−6536−3932]

[1] Univ Rennes, Inria, CNRS, IRISA
[2] The University of Sydney
[3] Sonic Labs

Abstract. Blockchains use consensus protocols to reach agreement, e.g., on the ordering of transactions. DAG-based consensus protocols are increasingly adopted by blockchain companies to reduce energy consumption and enhance security. These protocols collaboratively construct a partial order of blocks (DAG construction) and produce a linear sequence of blocks (DAG ordering). Given the strategic significance of blockchains, formal proofs of the correctness of key components such as consensus protocols are essential. This paper presents safety-verified specifications for five DAG-based consensus protocols. Four of these protocols —DAG-Rider, Cordial Miners, Hashgraph, and Eventual Synchronous BullShark — are well-established in the literature. The fifth protocol is a minor variation of Aleph, another well-established protocol. Our framework enables proof reuse, reducing proof efforts by almost half. It achieves this by providing various independent, formally verified, specifications of DAG construction and ordering variations, which can be combined to express all five protocols. We employ TLA+ for specifying the protocols and writing their proofs, and the TLAPS proof system to automatically check the proofs. Each TLA+ specification is relatively compact, and TLAPS efficiently verifies hundreds to thousands of obligations within minutes. The significance of our work is two-fold: first, it supports the adoption of DAG-based systems by providing robust safety assurances; second, it illustrates that DAG-based consensus protocols are disposed to practical, reusable, and compositional formal methods.

Keywords: Formal verification · Theorem Proving · TLA+ · Consensus · Blockchain

1 Introduction

At the core of cryptocurrency lies blockchain technology, which relies on consensus protocols to coordinate network processes to achieve agreement on the state of the blockchain. Early blockchain consensus protocols, such as those used in Bitcoin and Tendermint, depend on varying degrees of synchrony within their operational environments to ensure safety (preventing conflicting transactions) and

liveness (guaranteeing eventual transaction confirmation) [39,10,34,24]. However, recent advances have introduced asynchronous probabilistic consensus protocols based on Directed Acyclic Graphs (DAGs) [28,29,4,21,19,25]. These protocols not only demonstrate high performance and guarantee Byzantine Fault Tolerance (BFT) but also utilize processes fairly and exhibit low communication complexity. Given these advantages, there has been a growing interest in DAG-based protocols in both industrial and academic circles [44]. Several leading blockchains have adopted DAG-based protocols as their primary consensus mechanisms [3,19,27]. Given the trillions of dollars locked in various blockchains [22], manipulating the consensus protocol of the blockchain is a natural attack vector. Double-spending attacks [33] exploit unsafe protocols by identifying inputs and conditions that lead to inconsistent blockchain states, allowing a currency unit to be spent multiple times. Although consensus protocols are generally designed to be safe and thus mitigate such attacks, ad hoc software design practices complicate the assurance of safety across all potential inputs. For example, even well-established, state-of-the-art blockchains are unsafe [40,46]. This underscores the need for solutions that ensure safety in the design of consensus protocols.

A common approach involves testing. However, a testing regime for a software implementation might not reveal deficiencies in the protocol design itself, and malicious behaviors of byzantine processes may not be fully understood and covered by test cases. Another approach is model checking, but model checking frequently encounters the state-space explosion problem, making it infeasible to exhaustively verify models as they scale. Together, both testing and traditional model checking cannot provide guarantees of correctness no matter the number of participating processes and no matter their configurations and behavior. Given the vast number of possible interleavings in an asynchronous environment, omitting rigorous verification is risky.

A more robust method for ensuring safety is *formal proofs*: constructing a mathematical model that describes the system behavior and providing a mathematical proof that the model is correct for all possible inputs and configurations. While such proof efforts exist for a small number of consensus protocols [43,11,45,9,6,2], the vast majority of consensus protocols operate with an informal assumption of safety. This is often due to perception that formal proofs are tedious, time-consuming, and complex, largely because of a lack of compositional *building blocks* that facilitate the proof of correctness for new protocols. While reuse is often achievable at the specification level, the corresponding safety proofs generally lack compositionality, making verification challenging.

In this paper, we present an industrial case study where proven models of consensus protocols are used to drive development. We emphasize making formal proofs more practical, reusable, and compositional for distributed systems that leverage DAG-based consensus protocols. This is achieved by creating compositional specifications and proofs that encapsulate commonalities between protocols, thereby allowing the reuse of proofs. Our overall approach relies on the principle of *equality by abstraction*[47]: by selecting appropriate abstractions for DAG-based protocols, we can decompose protocol properties and proofs into

modular components. Using this approach we provide safety-verified specifications for five DAG-based protocols DAG-Rider [28], Cordial Miners [29], Hashgraph [4], Eventually Synchronous BullShark [25], and a variant of Aleph. Our experience shows that while these (and many other) DAG-based protocols share fundamental principles, they differ in the variations introduced to improve performance. Nonetheless, these variations can significantly impact their correctness. Thus, our work demonstrates significant potential for reuse in handling other DAG based consensus protocols [44]. We implemented an open-source formal specification of above mentioned protocols (available at [23]). Our specification and proofs are in the Temporal Logic of Actions (TLA) [35]. For this we use TLA+, tool to write specification and proofs in TLA. We check proofs using the TLA+ proof system (TLAPS) [14], a proof system for mechanically checking proofs written in TLA. We use TLA+ as it is a popular specification language with numerous industrial case studies [41,26] and well supported tooling [31].

2 DAG-based Consensus Protocols in Blockchains

In this section we provide a synthesis of the fundamental principles shared by DAG-based consensus protocols, as well as common variations introduced to enhance performance. DAG-based consensus protocols [28,29,4,21,19,25] operate in a distributed environment consisting of n processes that communicate through reliable peer-to-peer message channels, meaning that messages are neither lost nor duplicated. Each process is assumed to have an id and all messages are signed by their sender, ensuring that they cannot be forged. Up to $f < \frac{n}{3}$ of the processes are Byzantine-faulty, which means that they may deviate from the protocol. Such processes can send any message to others or refuse to send certain messages. Correct processes, in contrast, adhere to the algorithm's specification and continue execution indefinitely. The system is typically asynchronous, meaning that processes operate at independent speeds and messages may experience arbitrary delays. However, there are exceptions, e.g., Eventual Synchronous Bullshark (ES Bullshark) model assumes certain bounds on message delays [25].

DAG-based consensus protocols solve the Byzantine Atomic Broadcast (BAB) problem [38,12,17]. BAB provides a mechanism to propose sets of transactions (blocks) and totally order them in blockchain systems with Byzantine faulty processes. Informally, Byzantine Atomic Broadcast guarantees the following:

1. **Agreement:** All correct processes agree on the set of messages they deliver.
2. **Validity:** All messages broadcast by correct processes are eventually delivered with probability 1 [4].
3. **Integrity:** A correct process delivers a message at most once, and only if it was previously broadcast.
4. **Total Order:** All correct processes agree on the delivery order of the set of messages they deliver.

[4] Randomization is needed due to asynchronous nature of the communication and Fischer, Lynch, and Paterson (FLP) impossibility result [20].

Although we have not found an agreed-upon definition of "DAG-based consensus protocols", we propose an informal definition that matches many published protocols [28,29,4,21,19,25], further detailed in Figure 1:

DAG-based consensus protocols solve the BAB problem in two phases: (1) In the **DAG construction phase** processes communicate their blocks and construct a Directed Acyclic Graph (DAG) of the exchanged blocks. (2) In the **ordering phase**, processes use their locally constructed DAGs to determine a total order of the blocks without requiring additional communication.

In the rest of this section, we broadly outline our view of these two phases. We do this in enough detail that a reader who wants to model their own DAG-based consensus protocol can appreciate how to separate their protocol into the two phases, and how to apply our abstractions in Section 3.

2.1 DAG construction phase

In the DAG construction phase, processes create and communicate blocks in the form of vertices. A vertex is defined recursively and consists of three components: the creator's ID, the block the creator proposes, and a possibly empty set of references to other vertices. Each process p builds a Directed Acyclic Graph (DAG) $G(p)$, called its *local DAG*, which includes vertices created by process p, as well as some of those received from other processes. References act as outgoing edges: if v, v' are vertices in $G(p)$ and v' is referenced by v, then $G(p)$ has a directed edge from v to v'. Vertices are added sequentially to $G(p)$. The key property of the construction of $G(p)$ is that a newly created or received vertex v is only added to $G(p)$ once all the vertices referenced by v have been added to $G(p)$ (in the meantime, v is stored in a buffer). Thus, $G(p)$ is indeed acyclic: if a vertex v is added to $G(p)$, and if there is a reference path from v to some vertex v', then v' was previously added to $G(p)$. Note that the local DAGs of two correct processes need not be identical or even subgraphs of each other. However, if a vertex v appears in two local DAGs, the sub-DAG rooted at v, also called the "causal history of v", is identical in both. Variations in the DAG construction phase in different protocols arise from two main factors: the communication primitive used to exchange blocks and the type of DAG being constructed (see Fig. 1).

Communication Some protocols, e.g., DAG-Rider, BullShark, and Aleph, use a *reliable broadcast* primitive to communicate vertices (the specification of Reliable broadcast is the same as that of Byzantine Atomic Broadcast but without Total Order). As a result, these protocols prevent equivocation, meaning that no vertex appears more than once in the local DAGs of correct processes. In contrast, other protocols such as Cordial Miners, Hashgraph, and Lachesis, rely on unreliable communication to spread vertices, e.g., plain broadcast [29] or gossip [8]. While this approach may allow Byzantine processes to introduce conflicting vertices, it offers significantly lower latency compared to reliable broadcast.

DAG Type In protocols like DAG-Rider, Cordial Miners, BullShark, and Aleph, processes create new vertices in rounds, one in each. Thus each vertex also contains its creation round number. A process has *completed* round r once its local DAG contains vertices created by at least $n - f$ processes in round r. To create a vertex v in round $r > 0$, a process must wait to complete round $r - 1$ and ensure that v references all the vertices in its local DAG from round $r - 1$ at the time of creation. This round-based construction results in a *layered structure*: vertices in round $r = 0$ form the first (bottom) layer, while vertices in round $r > 0$ have edges to at least $n - f$ vertices in round $r - 1$ created by distinct processes. Figure 2a illustrates this structure for DAG-Rider.

On the other hand, in Hashgraph and Lachesis there is no use of rounds. Process p creates a new vertex v upon receiving some vertex v' from another process. Vertex v references exactly two vertices: the last vertex created by p, and v'. As a result, these DAGs lack a layered structure as that of previous protocols. While ordering protocols are simpler to implement on layered structured DAGs, unstructured DAGs capture a finer partial order of vertices, helping provide additional fairness guarantees [4].

As discussed above, there are performance trade-offs in the way local DAGs are constructed. While we have outlined some common configurations, the design space offers numerous possibilities. For instance, one could combine reliable and unreliable communication to balance the trade-off between latency and fault tolerance when exchanging blocks [18].

2.2 Ordering phase

In this phase, each process has a local DAG, representing a partial order of vertices, that must be totally ordered. This must be done independently, without additional communication, while ensuring that all correct processes order them identically (Total Order).

Recall that while correct processes might have different local DAGs, the causal histories of common vertices are identical. We now outline a four-step method for ordering the vertices, which is applicable to all the protocols discussed in this paper:

1. Partition the local DAG into "frames": The vertices in the local DAG are divided into sequentially numbered frames, starting from 0. For example, in a structured DAG, a frame could correspond to a round or a fixed number of consecutive rounds (aka "wave").
2. Agree on anchor vertices for each frame: For each frame x, agree on a set of vertices A_x within that frame to serve as "anchors". The key property of an anchor is that it must eventually be present in the local DAGs of all correct processes.
3. Group (aka batch) vertices with the help of causal histories of anchors: There are various ways to do this. E.g., for a frame x, define the x^{th} group as those vertices in the local DAG that are in the union of the causal histories of the vertices in A_x, but are not in the union of causal histories of the vertices in A_i, for any $0 \leq i < x$ (or, e.g., intersection can be used instead of union).

4. Order groups: The groups are first sorted by their frame numbers. Vertices within each group are then sorted using a deterministic criterion, e.g., by hash value.

For example, in Fig. 2b, vertex 1 and vertex 2 are the singleton sets of anchors for waves 1 and 2, respectively. The shaded regions between anchors 1 and 2 represent differences in causal history, forming groups 1 and 2. These batches can then be ordered deterministically by topological sort.

Notice that the only non-trivial part of the above procedure is agreeing on anchors for each frame. Simple strategies for choosing anchors are insufficient in an asynchronous network. For instance, always selecting the set of vertices created by the process with the lowest ID in the local DAG as anchors might seem plausible. However, this approach fails because an anchor must exist in the local DAGs of all correct processes; having a vertex in one's local DAG does not guarantee its presence in the local DAGs of others. To address this, one needs to solve an instance of Binary Agreement to determine whether a vertex exists in local DAG of every correct process. Due to the Fischer, Lynch, and Paterson (FLP) impossibility result [20], which states that no deterministic protocol can guarantee consensus in an asynchronous system with faults, randomization must be incorporated into anchor agreement (to ensure progress). In the literature, there are two main variations, discussed below and summarized in Table 1.

Global Perfect Coin (GPC) ordering. In protocols such as DAG-Rider, Cordial Miners, and BullShark, each process divides its local DAG into a fixed number of consecutive rounds, called *waves* (4 in DAG-Rider, 5 in Cordial Miners, and 2 in BullShark). Waves act as frames for anchor agreement. Processes then use a Global Perfect Coin (GPC) [5], a cryptographic primitive to agree on a random leader process in each wave.[5] Intuitively, GPC uniformly samples a process ID at random for each wave and incorporates a holding mechanism to enforce unpredictability, especially when only Byzantine processes query it. The vertices created by the leader process in the first round of each wave are called leader vertices. Subsequently, a deterministic leader commit protocol is executed to decide which leader vertex, if any, will be committed as an anchor. This commit protocol relies on the properties of the local DAG. For instance, protocols that use a reliable broadcast primitive during the DAG construction phase ensure that each process creates a unique vertex in every round, leading to at most one leader vertex per wave in the local DAG. In contrast, protocols that use unreliable communication cannot guarantee a unique leader vertex in each wave, complicating the agreement on the anchor.

Virtual Voting (VV) ordering. Protocols such as Hashgraph and Aleph simulate a round-based Binary Agreement Protocol (BAP) to determine whether a vertex

[5] Eventual Synchronous Bullshark, a variant of Bullshark, assumes a certain degree of synchrony in communication. As a result, processes do not have to rely on GPC to compute the leader process; instead, the leader is chosen deterministically, for example, using a round-robin approach.

PROTOCOL	DAG CONSTRUCTION PHASE		ORDERING PHASE				
	Communication	DAG Type	DAG Processing	Wave Size	Anchor Agreement	Coin Rounds	Equivocation
DAG Rider	Reliable (RB)	Structured	-	4 rounds	GPC-based	-	-
Cordial Miners	Unreliable	Structured	-	5 rounds	GPC-based	-	Need to handle
ES BullShark	Reliable (RB)	Structured	-	2 rounds	Deterministic	-	-
Hashgraph	Unreliable	Unstructured	Required	-	Randomised VV	After 10 Rounds	Need to handle
Aleph	Reliable (RB)	Structured	-	-	Randomised VV	After 4 Rounds	-

RB: Reliable broadcast, GPC: Global Perfect Coin, VV: Virtual Voting

Fig. 1

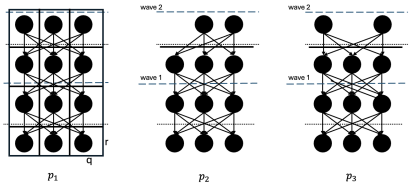
v from a frame in the local DAG becomes part of the anchor set. Round-based BAPs enable processes to achieve consensus on a binary decision (0 or 1). Each process participates in asynchronous rounds of voting, starting with an initial vote (initial value) in round 1. In each round, processes broadcast their votes and wait to receive votes from at least $n - f$ processes. The decision rules applied to the received votes dictate the next round's vote and, the final decision. Certain rounds use randomization in their decision rules (to ensure progress).

In Virtual Voting (VV) ordering, the votes of each process p in each round r (as required by the round-based BAPs) are computed virtually from $G(p)$, based on the relationship between the vertex v (the anchor to be decided) and the vertices created by p. Protocols that construct structured DAGs can directly compute these necessary virtual votes. In contrast, protocols that construct an unstructured DAG must first convert them into a structured DAG, referred to as the witness DAG (inspired by [4]), before computing these votes. The frames used for anchor agreement then correspond to the rounds of the structured DAG. (e.g., the "local DAG" in Aleph or the "witness DAG" in Hashgraph).

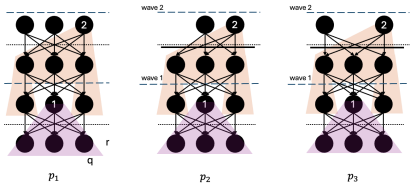
3 Abstractions towards simple and generic specifications

In this section, we describe the overall architecture of our specifications and the abstractions employed to maximize proof reuse for verifying five protocols: DAG-Rider, Cordial Miners, ES Bullshark, Hashgraph, and a variant of Aleph.

First, we divide each protocol specification into two modular components: DAG construction and ordering. This separation reflects the inherent structure of DAG-based consensus protocols (Section 2) and facilitates modular verification. A naive such approach to verifying five protocols would require $5+5 = 10$ verified specifications. However, our analysis in Section 2.1, summarized in Figure 1, highlights overlapping patterns in the DAG construction phases. Specifically, the



(a) DAG construction phase: building local DAGs for each processes and our grid abstraction depicted for P1.



(b) Ordering phase: selecting anchors across local DAGs.

Fig. 2: High-level depiction of the two phases in DAG-Rider. The black dots depict vertices that form a local DAG. The dotted blue lines represent a round that ends a wave, and the dotted black line represents a round in a wave.

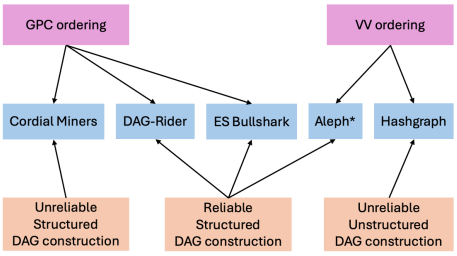


Fig. 3: Building blocks and their usage in verifying the 5 DAG-based consensus protocols. Each block's/protocol's TLA+ specification can be found in the corresponding `Specification.tla` file, and proofs in the corresponding `Proofs.tla` file localted in [23].

DAG construction phase of DAG-Rider, ES Bullshark, and Aleph is identical. By leveraging these similarities, we devise 3 distinct DAG construction specifications (Figure 3): (1) Reliable structured, (2) Unreliable structured, and (3) Unreliable unstructured. We discuss the abstractions in them in Subsection 3.1.

Next, we consolidate the five ordering variations discussed in Section 2.2 and summarized in Figure 1 into two fundamental specifications: GPC ordering and VV ordering. This is achieved by isolating the key functionalities of the ordering phase and integrating them into the DAG construction specifications. A detailed discussion is provided in Subsection 3.2.

Finally, we refine the ordering specifications by incorporating the DAG construction specifications to produce complete verified specifications for the five protocols (Figure 3). This refinement step is elaborated in Subsection 3.3. We remark that in Figure 3, our complete specifications for ES Bullshark and DAG-Rider rely on the same DAG construction and ordering specifications. The differences arise in the refinement step, which defines the wave size, leader position for each wave, and constraints for wave commitment.

In the remainder of the section, we write "GPC ordering specification" to refer to the TLA+ specification of GPC ordering as found in the file `GPCorderingSpecification.tla`, and similarly for other specifications.

3.1 Abstractions in DAG construction specifications

Causal histories as vertices. In our DAG construction specifications, a vertex in a given position represents not only a block but also its entire causal history within the DAG (inspired by [28]). Listing 1.1 describes the set of vertices (`VertexSet`). This abstraction significantly simplifies proofs by ensuring that agreement on anchor vertices immediately implies agreement on their causal histories as required in ordering phase (see Section 2.2).

Structured-DAG data-structure. In our reliable structured and unreliable structured DAG construction specifications, a structured DAG is modeled as a two-dimensional array where the index (q, r) stores vertices v created by process q in round r. We refer to index (q, r) as the vertex v's *position*. Listing 1.1 describes the type of local DAGs for every process (`dag`).

Listing 1.1: Data Structures.

```
VertexSet==[creator:ProcessSet, block:BlockSet, references:SUBSET(VertexSet)]
dag \in [ProcessSet->[RoundSet->[ProcessSet->SUBSET(VertexSet)]]]
broadcastNetwork \in [ProcessSet \cup {"History"}->SUBSET(VertexSet)]
broadcastRecord \in [ProcessSet->[RoundSet -> BOOLEAN]]
```

Abstraction of Reliable Broadcast. Reliable broadcast ensures three properties (Section 2.1): validity, agreement, and integrity. While validity and agreement are essential for proving consensus liveness, only integrity is required for safety. Therefore, we weaken the reliable broadcast abstraction so that it only guarantees integrity, omitting the guarantees of validity and agreement. This weakening does not imply that the proofs fail when validity and agreement hold; rather, it ensures the proofs remain applicable even if these properties do not hold.

Our specification of reliable broadcast also assumes that only the first broadcast vertex of a process in round r can be delivered, thus ensuring integrity. although this weakens the integrity property, we argue that it is reasonable in this context. Notably, only faulty processes send multiple messages in a single round. If a faulty process broadcasts several messages and one of the later messages is to be delivered, this scenario can be equivalently modeled as an execution where the faulty process sends the later message first.

In our weakened specification, reliable broadcast is modeled using two state variables as shown in Listing 1.1. The variable `broadcast-Network` maintains the sets of vertices to be delivered to each process. Meanwhile, `broadcast-Record` maintains a log of processes that have broadcast a vertex in each round. A vertex v broadcast in round r is added to `broadcast-Network` only if `broadcast-Record` confirms that the same process has not previously broadcast in round r.

Abstraction of Unreliable communication. In our unreliable structured and unreliable unstructured DAG construction specifications, we abstract away the specifics of unreliable communication implementations. A process p receiving

a vertex from process q is modeled as p non-deterministically choosing a vertex from q's local DAG and copying it. Listing 1.2 illustrates when the Receive predicate is enabled. This non-deterministic approach captures all possible interleavings that a specific communication implementation might produce. This abstraction gives a simpler and more generic specification as there are various ways to implement a unreliable communication [8].

Listing 1.2: Receive in unrelaible communication.

```
\* Process p receiving vertex v of round r from process q.
Receive(p, r, v, q) ==
    /\ v \notin dag[p][r][v.creator]
    /\ v \in dag[q][r][v.creator]
    /\ dag' = [ dag EXCEPT ! [p][r][v.creator] =
             dag[p][r][v.creator] \cup {v}]
```

3.2 Abstractions in ordering specifications

In this subsection, we discuss the ideas and abstractions that enable us to consolidate the five ordering variations into two fundamental specifications: GPC ordering and VV ordering.

Global Perfect Coin (GPC) ordering

Separation of concerns for uniform GPC ordering. DAG construction using reliable broadcast ensures a single vertex at each position in the DAG, while unreliable DAG construction may result in multiple vertices per position. This follows from the Integrity property of reliable broadcast, as discussed in Section 2.1. To handle both these cases in a uniform way, and to exploit compositionality, we divide the property of agreeing on committed wave leader vertices (see Section 2.2) into two sub-properties: *Vertex Agreement*: agreeing on a vertex for each position (to address cases with multiple vertices), and *Position Agreement*: agreeing on committed wave leader positions. We push *Vertex Agreement* into the DAG construction specifications (specifically, into reliable and unreliable structured DAG construction), and *Position Agreement* into the GPC ordering specification. The safety of *Vertex Agreement* in reliable and unreliable structured DAG construction specifications is formalized as the state invariants `DAGConsistency` and `RatificationConsistency`, respectively, as shown in Listing 1.3. On the other hand, in GPC ordering specification the state invariant `LeaderConsistency` formalizes the safety of *Position Agreement* as detailed in Listing 1.3.

Listing 1.3: Safety Invariants.

```
\* Safety of relaible DAG construction specification.
\* Note that the dag data structure has been modified to store a single
     vertex in each position instead of a set of vertices.

DagConsistency ==
   \A p, q \in ProcessorSet, r \in RoundSet, o \in ProcessorSet:
     ( /\ p \notin faulty /\ q \notin faulty /\ r \= 0
       /\ dag[p][r][o] \in VertexSet
```

```
       /\ dag[q][r][o] \in VertexSet ) => dag[p][r][o] = dag[q][r][o]
-----------------------------------------------------------------------
\* Safety of unelaible DAG construction specification.

RatificationConsistency ==
   \A p, q \in ProcessorSet, r \in RoundSet, o \in ProcessorSet:
     ( /\ p \notin faulty /\ q \notin faulty /\ r \= 0
       /\ dag[p][r][o].ratifiedVertex \in VertexSet
       /\ dag[q][r][o].ratifiedVertex \in VertexSet )
       => dag[p][r][o].ratifiedVertex = dag[q][r][o].ratifiedVertex
-----------------------------------------------------------------------
\* Safety of GPC ordering specification.

LeaderConsistency ==
   \A p, q \in ProcessorSet:
     ( /\ p \notin faulty /\ q \notin faulty
       /\ decidedWave[p] <= decidedWave[q] )
       => IsPrefix(leaderSeq[p].current, leaderSeq[q].current)
```

Abstraction of local DAG into wave-DAG. To simplify the GPC ordering specification, we abstract the local DAGs by focusing only on wave leader positions. Since each wave has a unique leader position (see Section 2.2), we construct a *wave-DAG* consisting of vertices nominated as wave leaders during the ordering phase. Edges in the wave-DAG represent a relevant relation between wave leaders; either the transitive closure relation of strong edges (DAG-Rider) [28] or the ratification relation (Cordial Miners) [30].

As shown in Listing 1.4, the variable `LeaderRelation` represents a snapshot of the wave-DAG: For every process p and wave ID w, `LeaderRelation` stores the following information: (i) the presence of the leader vertex of wave w in the wave-DAG of p, and (ii) the set of all waves w' such that the leader vertex of wave w satisfies the corresponding relation with the leader vertex of wave w'.

Listing 1.4: Some more data structures.

```
\* state varible leaderRelation capturing the snapshot of wave-DAG
leaderRelation \in [ ProcessSet->[WaveSet->[exists: BOOLEAN, edges: SUBSET(
    WaveSet)]]]
-----------------------------------------------------------------------
\* GPC as a function parameter to the specification
CONSTANT chooseLeader
chooseLeaderTypeAs == chooseLeader \in [WaveSet->ProcessSet]
```

Abstraction of Global Perfect Coin. Global Perfect Coin (GPC) (see Section 2.2) ensures unpredictability, agreement, termination, and fairness when selecting candidate wave-leaders [28]. While unpredictability, termination, and fairness are crucial for proving consensus liveness, agreement alone is sufficient for ensuring safety [28]. We relax the guarantees of GPC to make its specification compatible with TLA+, which lacks support for expressing randomness. This relaxed version of GPC does not guarantee unpredictability and need not imply fairness, however it ensures agreement and termination. Effectively, this relaxation models a scenario where Byzantine processes have increased power, yet the protocol's safety remains uncompromised. By removing the requirements of unpredictability and fairness, we represent the global perfect coin as an arbi-

trary function `chooseLeader`, shared among all processes, mapping waves to processes, as demonstrated in Listing 1.4.

Virtual Voting (VV) ordering

Separation of concerns for uniform VV ordering. The reliable structured DAG construction ensures the following (see Section 2.1): (1) a round-based structured local DAG and (2) a property we identify and call *reference consistency*, i.e., if v and v' are round-r vertices in the local DAGs of two correct processes, and if v references v_p and v' references v'_p, where v_p and v'_p were created in round $r-1$ by some process p, then $v_p = v'_p$. In contrast, unreliable and unstructured DAG constructions neither provide a round-based structured local DAG nor guarantee reference consistency. Since a vertex's vote is determined by its references, one of the requirements for underlying Byzantine Agreement Protocols (BAPs), as in Hashgraph and Aleph, is to construct a structured DAG that ensures reference consistency. To handle both cases uniformly and leverage compositionality, we divide the task of agreeing on anchors into two sub-tasks: (1) constructing a structured DAG whose vertices satisfy reference consistency, and (2) agreeing on anchors for each frame of the structured DAG, assuming its vertices satisfy reference consistency. The specification of (1) is pushed into the DAG construction specifications, while the specification of (2) forms the VV ordering specification.

Listing 1.5: Some more data structures.

```
WitnessSet == [source:ProcessSet, frame:FrameSet, vertex:VertexSet,
    stronglysees:SUBSET(WitnessSet)]
witnessDAG \in [ProcessSet->[Frames->[ProcessSet->SUBSET(WitnessSet)]]]
```

In the reliable structured DAG construction specification, the safety invariant `ReferenceConsistency`, defined in Listing 1.6, formalizes the reference consistency property for the local DAG. In the unreliable structured DAG specification, processes constructs a separate structured DAG, whose vertices are called "witnesses", referred to as the "witness-DAG". Listing 1.5 describes the definitions for the set of witnesses (`WitnessSet`) and type of the witness-DAG. The safety invariant `StronglySeenConsistency`, also defined in Listing 1.6, formalizes the references consistency property for the witness DAG. Finally, the state invariant `FameConsistency` formalizes the safety requirements of VV ordering specification, as described in Listing 1.6.

Listing 1.6: Some more safety invariants.

```
\* Safety property of relaible strucutred DAG construction, it is not hard to
     see that this follows from earler defined safety in Listing.
ReferenceConsistency ==
  \A p \in ProcessSet, q \in ProcessSet, s \in VertexSet, l \in VertexSet:
     ( /\ p \notin faulty /\ q \notin faulty
       /\ s \in dag[q][s.round][s.creator]
       /\ l \in dag[p][l.round][l.creator] )
       => ( \A a \in s.referneces, e \in l.referneces:
            a.round = e.round /\ a.creator = e.creator => a = e )
-----------------------------------------------------------------------
\* Safety of unrelialbe unstrucutred DAG construction. The invariant is
    repharsed version of TLA spec for clarity, however posses same meaning.
```

```
StronglyseeConsistency ==
  \A p \in ProcessSet, q \in ProcessSet, s \in WitnessSet, l \in WitnessSet:
     ( /\ p \notin faulty /\ q \notin faulty
       /\ s \in witnessDAG[q][s.round][s.creator]
       /\ l \in witnessDAG[p][l.round][l.creator] )
       => ( \A a \in s.stronglysee, e \in l.stronglysee:
            a.frame = e.frame /\ a.source = e.source => a = e )
------------------------------------------------------------------------
\* Safety of VV ordering specification. DecidedFrame[p][x] is a predicate
    variable set to true when all leaders of frame x are decided by process p
    . FamousWitnesses[p][x] stores the set of committed leaders in frame x by
     process p.

FameConsistency ==
  \A p \in ProcessSet, q \in ProcessSet, x \in Frames:
          DecidedFrames[p][x] /\ DecidedFrames[q][x]
            => FamousWitnesses[p][x] = FamousWitnesses[q][x]
```

To simplify the specifications across all the variants, we abstract away internal states of procedures and focus solely on their input/output behavior. For example, in the *wave-ready* procedure described in algorithm 3 in DAG-Rider [28], only the variables *decidedWave*, *deliveredVertices*, and *leadersStack* are retained, while the loop variable *w'* and the auxiliary variable *v'* are excluded.

3.3 Complete protocol specification

For a given protocol, our complete specification refines the ordering specification by incorporating the DAG construction specification. More precisely, we use the TLA+ *interface refinement* construct [35], which allows one to obtain a lower-level specification by instantiating the variables of a higher-level specification. To derive the complete protocol specification, we modify the DAG construction specification in three steps. 1. *Introduce new variables:* For each ordering specification state variable, introduce a corresponding state variable in the DAG construction specification. 2. *Map variables and constants:* Using the TLA+ *interface refinement* construct, assign values to the ordering specification constants as a function of the DAG construction specification constants. Also, clarify the mapping between the new state variables and the ordering specification state variables. 3. *Update actions:* For each action in the DAG construction specification, specify how it affects the newly introduced variables using the actions of the ordering specification. If no specific action is performed, define a stutter action where the state variables remain unchanged.

We illustrate the three steps for DAG-Rider specification, which refines the GPC ordering specification with a reliable structured DAG construction specification. The GPC ordering specification consists of three constants (`NumWaves, NumProcesses, Numfaulty`), five variables (`commitWithRef, decidedWave, leaderRelation, leaderSeq, faulty`), and two actions (`CreateNewWave, DecideWave`). Listing 1.7 describes the new variables in reliable structured DAG construction specification, their mapping to the variables of the GPC ordering specification, and the assignment to the constants. When a vertex is added at the leader position of a wave, the `CreateNewWave` action updates the newly introduced variables. Similarly, when a wave is completed and can be decided in the local DAG, the `DecideWave` action updates the newly introduced variables.

Table 1: Specification and proof metrics. Performed on a 2.10 GHz CPU with 8 GB of memory, running Windows 11 and TLAPS v1.4.5.

Metric \ Phase	Reliable structured	Unreliable structured	Unreliable unstructured	GPC Ordering	VV Ordering
Size of spec. (# loc)	403	160	230	272	136
Number of invariants	6	6	7	10	18
Size of proof (# loc)	460	594	554	822	2120
Max level of proof tree nodes	10	9	8	9	13
Max degree of proof tree nodes	7	8	7	7	11
# obligations in TLAPS	633	895	665	1302	3316
Time to check by TLAPS (s)	49	68	74	125	651

For all other actions in the reliable structured DAG construction, the newly introduced variables remain unchanged.

Listing 1.7: Instantiation of GPC ordering phase in protocol specification.

```
VARIABLES commitWithRefN, decidedWaveN, leaderRelationN, leaderSeqN, faultyN

GPCOrdering == INSTANCE GPCOrderingVerification
               WITH NumWaves <- w,
                    NumProcesses <- n,
                    Numfaulty <- f,
                    commitWithRef <- commitWithRefN,
                    decidedWave <- decidedWaveN,
                    leaderRelation <- leaderRelationN,
                    leaderSeq <- leaderSeqN,
                    faulty <- faultyN
\* w, n, f are constants in reliable structured DAG construction spec
```

4 Evaluation

The effort to understand, specify, and prove all the five protocols required approximately 14 person-months, distributed across 5 people. We evaluate the performance of our formal proof using TLA+ and TLAPS. We direct readers interested in background on TLA+, TLAPS, and proof strategies to Appendix A. Table 1 lists metrics, including the sizes of the DAG construction specifications and the ordering specifications. It also reports on metrics related to proof complexity, such as the number of invariants, the size of proof files, the maximum depth of the proof, the maximum branching in the tree-like proof structure, and the number of base (i.e., leaf) proof obligations. Finally, the table displays the verification time for proving the safety of each specification.

5 Related work

Parameterized verification is the problem of verifying a protocol for arbitrarily many processes [7]. For consensus protocols that tolerate Byzantine failures,

this means verifying the protocol is correct for all values of the parameters n, f with $n \geq 3f + 1$. In comparison, traditional model-checking only handles finite-state systems, and thus finitely many values of the parameters n, f. For instance, the Tendermint consensus protocol [34] has been modeled in TLA+, and certain properties, such as termination, were model-checked for small values of the parameters [9]. Our work can be seen as a contribution to the parameterized verification of distributed protocols by machine-checkable proofs. An alternative approach is to use parameterized model-checkers, e.g., ByMC [32]. For instance, Safety and Liveness of the consensus algorithm used in the Red Belly Blockchain [15] has been verified for all values of the parameters n, f with $n \geq 3f + 1$[6]. Although the algorithm only broadcasts binary values, that work verified both safety and liveness (under a weak fairness assumption).

Other BFT consensus protocols have been verified using interactive theorem provers. The work in [43] uses the IVy interactive theorem prover [1] to formally verify a variant of the Moonshine consensus protocol [43]. The work in [37] uses IVy and Isabelle/HOL [42] to verify the Stellar Consensus Protocol [36]. The Algorand [24] consensus protocol has been verified using Coq [13]. The safety of several non-Byzantine protocols such as variants of Paxos [11,45] have been verified using interactive theorem proving. However, these protocols are not robust to Byzantine faults and they have limited application in blockchains.

To the best of our knowledge, the only other work specific to DAG-based consensus protocols is an unpublished technical report by Crary [16] describing a formal verification of a DAG-based consensus algorithm, Hashgraph [4], with the theorem prover Coq. In contrast to that approach, our TLA+ specifications of DAG-based consensus protocols are lower-level; they are closer to actual implementations of the protocols and therefore require fewer manual abstractions. Clear advantages of the present approach are its separation of concerns between the communication phase and the ordering phase, and the reusability for several DAG-based consensus protocols.

6 Conclusion

Our motivation was to (1) provide a reusable and extensible framework for formally verifying DAG-based consensus protocols, (2) which can subsequently be leveraged to verify their implementations. Towards (1), we presented formally verified building-blocks and demonstrated their use in verifying five DAG-based consensus protocols. To achieve this, we designed the framework with reusability and extensibility in mind, and demonstrated its ability to reduce proof efforts by nearly half across the five protocols we analyzed. Other DAG-based consensus protocols can similarly benefit from our framework, even if one just re-uses one of the ordering specifications or DAG construction specifications. Regarding (2), we note that machine-checkable proofs can be leveraged to increase trust in implementations. Indeed, generating conformance tests from TLA+ specifications has found previously unknown bugs in distributed systems [48].

References

1. Ahn, K.Y., Denney, E.: Testing first-order logic axioms in program verification. In: Fraser, G., Gargantini, A. (eds.) Tests and Proofs - 4th International Conference, TAP@TOOLS 2010, Málaga, Spain, July 1-2, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6143, pp. 22–37. Springer (2010). https://doi.org/10.1007/978-3-642-13977-2_4
2. Aminof, B., Rubin, S., Stoilkovska, I., Widder, J., Zuleger, F.: Parameterized model checking of synchronous distributed algorithms by abstraction. In: Dillig, I., Palsberg, J. (eds.) Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10747, pp. 1–24. Springer (2018). https://doi.org/10.1007/978-3-319-73721-8_1
3. Aptos Foundation: Understanding Aptos: A comprehensive overview (2024), https://messari.io/report/understanding-aptos-a-comprehensive-overview
4. Baird, L., Luykx, A.: The hashgraph protocol: Efficient asynchronous BFT for high-throughput distributed ledgers. In: Proceedings of COINS 2020. pp. 1–7. IEEE (2020). https://doi.org/10.1109/COINS49042.2020.9191430
5. Bandarupalli, A., Bhat, A., Bagchi, S., Kate, A., Reiter, M.K.: Random beacons in monte carlo: Efficient asynchronous random beacon without threshold cryptography (version 1) (Apr 2024), https://doi.org/10.5281/zenodo.11044395
6. Bertrand, N., Gramoli, V., Konnov, I., Lazic, M., Tholoniat, P., Widder, J.: Holistic verification of blockchain consensus. In: Scheideler, C. (ed.) 36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA. LIPIcs, vol. 246, pp. 10:1–10:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPICS.DISC.2022.10, https://doi.org/10.4230/LIPIcs.DISC.2022.10
7. Bloem, R., Jacobs, S., Khalimov, A., Konnov, I., Rubin, S., Veith, H., Widder, J.: Decidability of Parameterized Verification. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers (2015). https://doi.org/10.2200/S00658ED1V01Y201508DCT013, https://doi.org/10.2200/S00658ED1V01Y201508DCT013
8. Boyd, S., Ghosh, A., Prabhakar, B., Shah, D.: Randomized gossip algorithms. IEEE Transactions on Information Theory **52**(6), 2508–2530 (2006). https://doi.org/10.1109/TIT.2006.874516
9. Braithwaite, S., Buchman, E., Konnov, I., Milosevic, Z., Stoilkovska, I., Widder, J., Zamfir, A.: Formal specification and model checking of the tendermint blockchain synchronization protocol (short paper). In: Bernardo, B., Marmsoler, D. (eds.) 2nd Workshop on Formal Methods for Blockchains, FMBC@CAV 2020, July 20-21, 2020, Los Angeles, California, USA (Virtual Conference). OASIcs, vol. 84, pp. 10:1–10:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/OASICS.FMBC.2020.10, https://doi.org/10.4230/OASIcs.FMBC.2020.10
10. Buterin, V.: Ethereum white paper: A next generation smart contract & decentralized application platform (2013), https://github.com/ethereum/wiki/wiki/White-Paper
11. Chand, S., Liu, Y.A., Stoller, S.D.: Formal verification of multi-paxos for distributed consensus. In: Fitzgerald, J.S., Heitmeyer, C.L., Gnesi, S., Philippou, A. (eds.) FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9995, pp. 119–136 (2016). https://doi.org/10.1007/978-3-319-48989-6_8

12. Coelho, P., Junior, T.C., Bessani, A., Dotti, F., Pedone, F.: Byzantine fault-tolerant atomic multicast. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 39–50 (2018). https://doi.org/10.1109/DSN.2018.00017
13. Coquand, T., Huet, G.P.: The calculus of constructions. Inf. Comput. **76**(2/3), 95–120 (1988). https://doi.org/10.1016/0890-5401(88)90005-3, https://doi.org/10.1016/0890-5401(88)90005-3
14. Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., Vanzetto, H.: TLA + proofs. In: Proceedings of FM 2012. Lecture Notes in Computer Science, vol. 7436, pp. 147–154. Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_14
15. Crain, T., Natoli, C., Gramoli, V.: Red belly: A secure, fair and scalable open blockchain. In: 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021. pp. 466–483. IEEE (2021). https://doi.org/10.1109/SP40001.2021.00087, https://doi.org/10.1109/SP40001.2021.00087
16. Crary, K.: Verifying the hashgraph consensus algorithm. Tech. rep., CMU (2021), https://arxiv.org/abs/2102.01167
17. Cristian, F., Aghili, H., Strong, R., Dolev, D.: Atomic broadcast: From simple message diffusion to byzantine agreement. Information and Computation **118**(1), 158–179 (1995). https://doi.org/https://doi.org/10.1006/inco.1995.1060, https://www.sciencedirect.com/science/article/pii/S0890540185710607
18. Dai, X., Wang, G., Xiao, J., Guo, Z., Hao, R., Xie, X., Jin, H.: LightDAG: A low-latency DAG-based BFT consensus through lightweight broadcast. Cryptology ePrint Archive, Paper 2024/160 (2024), https://eprint.iacr.org/2024/160
19. Fantom Foundation: Lachesis aBFT (2024), https://docs.fantom.foundation/technology/lachesis-abft
20. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32**(2), 374–382 (1985)
21. Gagol, A., Lesniak, D., Straszak, D., Swietek, M.: 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. pp. 214–228. ACM (2019). https://doi.org/10.1145/3318041.3355467
22. Geko, C.: Ethereum white paper: A next generation smart contract & decentralized application platform (2024), https://www.coingecko.com/en/chains
23. Ghorpade, P.: Reusable-Formal-Verification-of-DAG-based-Consensus. https://github.com/pranavg5526/Reusable-Formal-Verification-of-DAG-based-Consensus (2024)
24. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling Byzantine agreements for cryptocurrencies. In: Proceedings of SOSP 2017. pp. 51–68. ACM (2017). https://doi.org/10.1145/3132747.3132757
25. Giridharan, N., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Bullshark: Dag bft protocols made practical. Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (2022), https://api.semanticscholar.org/CorpusID:246015617
26. Hackett, F., Rowe, J., Kuppe, M.A.: Understanding inconsistency in azure cosmos db with tla+. In: 2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). pp. 1–12 (2023). https://doi.org/10.1109/ICSE-SEIP58684.2023.00006
27. Hedra: Streamlining consensus (2024), https://hedera.com/blog/streamlining-consensus-throughput-and-lower-latency-with-about-half-the-events

28. Keidar, I., Kokoris-Kogias, E., Naor, O., Spiegelman, A.: All you need is DAG. In: Proceedings of PODC 2021. pp. 165–175. ACM (2021). https://doi.org/10.1145/3465084.3467905
29. Keidar, I., Naor, O., Poupko, O., Shapiro, E.: Cordial miners: Fast and efficient consensus for every eventuality. In: Oshman, R. (ed.) 37th International Symposium on Distributed Computing, DISC 2023, October 10-12, 2023, L'Aquila, Italy. LIPIcs, vol. 281, pp. 26:1–26:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.DISC.2023.26, https://doi.org/10.4230/LIPIcs.DISC.2023.26
30. Keidar, I., Naor, O., Poupko, O., Shapiro, E.: Cordial Miners: Fast and Efficient Consensus for Every Eventuality. In: Proceeding of 37th International Symposium on Distributed Computing (DISC 2023). Leibniz International Proceedings in Informatics (LIPIcs), vol. 281, pp. 26:1–26:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPIcs.DISC.2023.26
31. Konnov, I., Kukovec, J., Tran, T.H.: Tla+ model checking made symbolic. Proc. ACM Program. Lang. **3**(OOPSLA) (oct 2019). https://doi.org/10.1145/3360549
32. Konnov, I.V., Lazic, M., Veith, H., Widder, J.: A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 719–734. ACM (2017). https://doi.org/10.1145/3009837.3009860
33. Kumar, A., Kumar Sah, B., Mehrotra, T., Rajput, G.K.: A review on double spending problem in blockchain. In: 2023 International Conference on Computational Intelligence and Sustainable Engineering Solutions (CISES). pp. 881–889 (2023). https://doi.org/10.1109/CISES58720.2023.10183579
34. Kwon, J.: Tendermint: Consensus without mining. https://tendermint.com/docs/tendermint.pdf (2014)
35. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002), http://research.microsoft.com/users/lamport/tla/book.html
36. Lokhava, M., Losa, G., Mazières, D., Hoare, G., Barry, N., Gafni, E., Jove, J., Malinowsky, R., McCaleb, J.: Fast and secure global payments with stellar. In: Brecht, T., Williamson, C. (eds.) Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019. pp. 80–96. ACM (2019). https://doi.org/10.1145/3341301.3359636
37. Losa, G., Dodds, M.: On the formal verification of the stellar consensus protocol. In: Bernardo, B., Marmsoler, D. (eds.) 2nd Workshop on Formal Methods for Blockchains, FMBC@CAV 2020, July 20-21, 2020, Los Angeles, California, USA (Virtual Conference). OASIcs, vol. 84, pp. 9:1–9:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/OASICS.FMBC.2020.9
38. Milosevic, Z., Hutle, M., Schiper, A.: On the reduction of atomic broadcast to consensus with byzantine faults. In: 2011 IEEE 30th International Symposium on Reliable Distributed Systems. pp. 235–244 (2011). https://doi.org/10.1109/SRDS.2011.36
39. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
40. Neu, J., Tas, E.N., Tse, D.: Two more attacks on proof-of-stake ghost/ethereum. In: Proceedings of the 2022 ACM Workshop on Developments in Consensus. p. 43–52. ConsensusDay '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3560829.3563560

41. Newcombe, C.: Why amazon chose tla + . In: Ait Ameur, Y., Schewe, K.D. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z. pp. 25–39. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
42. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002). https://doi.org/10.1007/3-540-45949-9
43. Praveen, M., Ramesh, R., Doidge, I.: Formally verifying the safety of pipelined moonshot consensus protocol. CoRR **abs/2403.16637** (2024). https://doi.org/10.48550/ARXIV.2403.16637
44. Raikwar, M., Polyanskii, N., Müller, S.: Sok: Dag-based consensus protocols. In: 2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). pp. 1–18 (2024). https://doi.org/10.1109/ICBC59979.2024.10634358
45. Schultz, W., Dardik, I., Tripakis, S.: Formal verification of a distributed dynamic reconfiguration protocol. In: Popescu, A., Zdancewic, S. (eds.) CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022. pp. 143–152. ACM (2022). https://doi.org/10.1145/3497775.3503688
46. Shoup, V.: Blue fish, red fish, live fish, dead fish. Cryptology ePrint Archive, Paper 2024/1235 (2024), https://eprint.iacr.org/2024/1235
47. Soffer, P.: Refinement equivalence in model-based reuse: Overcoming differences in abstraction level. J. Database Manag. **16**, 21–39 (07 2005)
48. Wang, D., Dou, W., Gao, Y., Wu, C., Wei, J., Huang, T.: Model checking guided testing for distributed systems. In: Proceedings of the Eighteenth European Conference on Computer Systems. p. 127–143. EuroSys '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3552326.3587442

A TLA+, TLAPS and Proof strategy

In this appendix, we first give a brief introduction to TLA+ and TLAPS. We then discuss the proof techniques for formally proving the safety invariants.

Background on TLA+ TLA+[35] is a tool for specifying systems, notably concurrent and distributed algorithms. It is based on Zermelo-Fraenkel set theory with choice for representing the data structures on which the algorithm operates, and the Temporal Logic of Actions (TLA), a variant of linear-time temporal logic (that allows binary *action* relations on states), for describing executions of the algorithm. To specify a system in TLA+ we use **state variables** and **actions**. A system state is an assignment of values to state variables. An action is a binary relation on states, specifying the effect of executing a sequence of instructions. An action is represented by a formula over unprimed and primed variables where unprimed variables refer to the values of the variables in the current state and primed variables refer to the values of the variables in the resulting state. For example, the instruction $x := x + 1$ is represented in TLA+ by the action `x' = x+1`. A system is specified by its actions and initial states, i.e., $\texttt{Spec} = \texttt{Init} \wedge \Box[\texttt{Next}]_{\texttt{vars}}$ where `Init` is the initial states predicate, and `Next` is a disjunction of all actions of the system, and `vars` is the tuple of all state variables. The expression $[\texttt{Next}]_{\texttt{vars}}$ is true if either `Next` is true, meaning that some action is true and therefore executed, or `vars` stutters, meaning that the values of the variables are same in the current and next states. The symbol $\Box$ is the temporal operator "always". Thus, `Spec` defines a set of infinite sequences of system states, i.e., those such that (1) the first state satisfies `Init`, and (2) every successive pair of states satisfies $[\texttt{Next}]_{\texttt{vars}}$. Such a sequence is called a *behavior*. Given a TLA+ specification one can then define safety properties of the set of behaviours. A safety property in TLA+ is a formula of the form $\texttt{Spec} \Rightarrow \Box\texttt{Inv}$, meaning that every state on every behavior that satisfies the system specification, satisfies the state predicate `Inv`, a formula over the (unprimed) state variables. We call the predicate `Inv` a **safety invariant**.

Background on TLAPS TLA+ Proof System (TLAPS) is a tool that mechanically checks proofs of properties specified in TLA+. The user writes proofs in TLA+ are written in a hierarchical style (see Listing 1.8), whose "terminal" proof obligations are simple enough for a proof tool to find the proof without any help from the user. The default behaviour of TLAPS is to try three backend provers in succession: CVC3 (an SMT solver), Zenon, and Isabelle [42]. If none of them find a proof, TLAPS reports a failure on the obligation. Other SMT solvers supported by TLAPS are Z3, veriT, and Yices. Temporal formulas are proved using LS4, a PTL (Propositional Temporal Logic) prover. Users can specify which prover they want to use by using its name and can specify the timeout for each obligation separately.

Proof Strategy Here, we simply give a high-level outline of the proof strategy for proving invariants. We outline the proof for an inductive invariant (`Inv1`) of the reliable structured DAG construction specification in Fig. 1.8. We aim to prove that the invariant holds on all reachable states of the ordering phase specification. This is formalized as: `LEMMA Inv1correctness` $=$ `Spec` $\rightarrow \Box$`Inv1`. Here we refer `Spec` for the ordering specification, recall that `Spec` has the form `Spec` $=$ `Init` $\wedge \Box$`[Next]`$_{\texttt{vars}}$, where `vars`, `Init`, `Next` are TLA+ macro names for set of variables, the initial state, and the possible actions leading to the next state. The lemma is proved by induction. The induction basis `Init` $\rightarrow$ `Inv1` is trivial and TLAPS proves it automatically. The induction step is captured by `Inv1` $\wedge \Box$`[Next]`$_{\texttt{vars}} \rightarrow$ `Inv1'` , where `[Next]`$_{\texttt{vars}}$ represents that either the state machine performs a `Next` step, or remains idle, and `Inv1'` states that the invariant holds on the state after performing `[Next]`$_{\texttt{vars}}$. These two cases are reflected in the code snippet as <2>1 and <2>2. Then, `Next` is a disjunction of three cases, each of which is proved separately in <3>1, <3>2, <3>3 and <3>4. Each case is proved independently by further breaking them into lower-level goals. TLAPS enables breaking these cases into progressively finer-grained proof obligations, ultimately reducing them to terms that follow from simple logical rules.

Listing 1.8: Proof structure of a specific invariant for the ordering specification.

```
LEMMA Inv1Lem == Spec => []Inv1
 <1>1 Init => Inv1
      ...
 <1>2 ASSUME StateType, StateType', Inv1, [Next]_vars
      PROVE Inv1'
      <2>1 ASSUME StateType, StateType', Next, Inv1
           PROVE Inv1'
           <3>1 ASSUME NEW p \in ProcessorSet, NEW b \in BlockSet,
                ProposeTn(p, b)
                PROVE Inv1'
                ...
           <3>2 ASSUME NEW p \in ProcessorSet, NEW r \in RoundSet,
                NEW q \in ProcessorSet, NEW v \in VertexSet, p # q,
                ReceiveVertexTn(p, q, r, v)
                PROVE Inv1'
                ...
           <3>3 ASSUME NEW p \in ProcessorSet, NEW v \in VertexSet,
                AddVertexTn(p, v)
                PROVE Inv1'
                ...
           <3>4 ASSUME NEW p \in ProcessorSet, NextRoundTn(p)
                PROVE Inv1'
                ...
           <3> QED BY <3>1, <3>2, <3>3, <3>4, <2>1 DEF Next
      <2>2 ASSUME UNCHANGED vars, Inv1
           PROVE Inv1'
           ...
      <2> QED BY <2>1, <2>2
 <1> QED BY <1>1, <1>2, TypeLem, PTL DEF Spec
```