Mechanising Blockchain Consensus

George Pîrlea
University College London, UK
george.pirlea.15@ucl.ac.uk

Ilya Sergey
University College London, UK
i.sergey@ucl.ac.uk

Abstract

We present the first formalisation of a blockchain-based distributed consensus protocol with a proof of its consistency mechanised in an interactive proof assistant.

Our development includes a reference mechanisation of the *block forest* data structure, necessary for implementing provably correct per-node protocol logic. We also define a model of a network, implementing the protocol in the form of a replicated state-transition system. The protocol's executions are modeled via a small-step operational semantics for asynchronous message passing, in which packages can be rearranged or duplicated.

In this work, we focus on the notion of global system safety, proving a form of eventual consistency. To do so, we provide a library of theorems about a pure functional implementation of block forests, define an inductive system invariant, and show that, in a quiescent system state, it implies a global agreement on the state of per-node transaction ledgers. Our development is parametric *wrt.* implementations of several security primitives, such as *hash*-functions, a notion of a *proof object*, a *Validator Acceptance Function*, and a *Fork Choice Rule*. We precisely characterise the assumptions, made about these components for proving the global system consensus, and discuss their adequacy. All results described in this paper are formalised in Coq.

CCS Concepts • **Theory of computation → Program verification**; • **Networks → Formal specifications**;

Keywords blockchain, consensus, protocol verification, Coq

ACM Reference Format:
George Pîrlea and Ilya Sergey. 2018. Mechanising Blockchain Consensus. In *Proceedings of 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3167086

CPP'18, January 8–9, 2018, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5586-5/18/01…$15.00
https://doi.org/10.1145/3167086

1 Introduction

The notion of decentralised blockchain-based consensus is a tremendous success of the modern science of distributed computing, made possible by the use of basic cryptography, and enabling many applications, including but not limited to cryptocurrencies, smart contracts, application-specific arbitration, voting, *etc.*

In a nutshell, the idea of a distributed consensus protocol based on *blockchains*, or *transaction ledgers*,[1] is rather simple. In all such protocols, a number of stateful nodes (participants) are communicating with each other in an asynchronous message-passing style. In a message, a node (a) can announce a *transaction*, which typically represents a certain event in the system, depending on the previous state of the node or the entire network (we intentionally leave out the details of what can go into a transaction, as they are application-specific); a node can also (b) create and broadcast a *block* that contains the encoding of a certain vector of transactions, created locally or received via messages of type (a) from other nodes. Each recipient of a block message should then *validate* the block (*i.e.*, check the consistency of the transaction sequence included in it), and, in some cases, append it to its local ledger, thus, extending its subjective view of the global sequence of transactions that have taken place in the system to date. The process continues as more messages are emitted and received.

In order to control the number of blocks in the system, distributed ledger protocols rely on certain cryptographic primitives, such as a hash-function *hash* defined both on transactions and blocks, a notion of a *proof object* necessary for defining the validity of a block, and an implementation of a *Validator Acceptance Function* (VAF) that is used to ensure that a block *b* is valid *wrt.* to a proof object *pf*. Having a block *b* and a proof object *pf*, one can check very fast whether *VAF b pf* is true or false. What appears to be difficult is to produce an instance of a proof object *pf*, as it requires computing a pre-image of the *hash* function with respect to the current state of the local ledger of a specific node. The exact specifics of designing a *VAF* and a discipline for *minting* blocks with *VAF*-valid proof objects, is a subject of active research, which is far beyond the scope of this paper, with the best known approaches being Proof-of-Work [9, 24] and Proof-of-Stake [3]. The computational hardness or

[1] Hereafter, we will be using the terms "(transaction) ledger" and "blockchain" interchangeably.

probabilistic rarity of minting valid blocks is what controls the overall block population.

However, this setup by itself does not deliver a global consensus between the nodes. Indeed, in an asynchronous network, where messages can be rearranged, duplicated, or arbitrarily delayed, two different nodes n_1 and n_2 can receive different, or even conflicting, sets of valid blocks and decide to adopt them in their local ledgers. Assuming that initially all nodes share the same initial block (so-called *Genesis Block*), at any further state of the network, each two nodes' ledgers can be in a *fork* relation, when neither of them is a prefix of the other. The consensus is enabled by fixing a globally known *Fork Choice Rule* (*FCR*) function, that provides a decidable strict total order on *all possible chains of blocks* and is transitive and irreflexive. Thus, upon receiving a block, a node must check whether appending it to its local ledger is going to increase the ledger's "weight", and keep it if so, discarding it otherwise. Assuming every node follows the same *FCR*-imposed discipline for chain comparison, all participants will *eventually* share the same blockchain/transaction ledger instance.

Alas, the reality is a bit more complicated than the description above. For example, in a realistic fault-tolerant system implementation, nodes cannot afford to ignore blocks that arrive "out of order", which is not uncommon in an asynchronous setting. Not registering such blocks in a node's local state would pose serious liveness problems, as such nodes would be stuck with a "stale" local ledger, unable to progress along with the rest of the world. Furthermore, some nodes may not be active or known system-wide at the very beginning of communication, so they will start by first manifesting themselves, interacting only with a small set of peers they know. Finally, any node in the system should be able to request from its peers the set of publicly announced blocks these peers have witnessed in the past, so it would be possible for the node to "catch up" with the global state of the system, if, for instance, it has joined the network late or has been offline for some time.

In light of these and multiple other possible scenarios of distributed interaction, we believe that having a clean and principled model for *rigorous formal reasoning about system-wide properties of distributed blockchain-based protocols* is of paramount importance for gaining trust in the foundational principles of algorithms underlying, in particular, implementations of modern cryptocurrencies, such as Bitcoin [24], Ethereum [37] and Tezos [13].

In this work we provide such a model.

Our contributions. We provide formal model of a blockchain-based consensus protocol, along with a set of necessary reference data structures and a network semantics, with an agenda to formally study its properties, abstracting away the implementation details of security-related primitives. Our contributions include the following formal artifacts:

- A description of a minimal set of security primitives: *hash*, *VAF*, *FCR*, along with a set of laws (axioms) they should abide, and a discussion of these laws' adequacy *wrt.* real-world implementations;
- A reference implementation of *block forests*—a purely functional data structure implementing the local state of a node in the protocol in the presence of adopted out-of-order blocks, as well as a library of theorems about block forests, necessary for proving the consensus property;
- A definition of a replicated state-transition machinery, implementing the per-node logic of the protocol, and semantics of the asynchronous network used for establishing protocol invariants;
- A formulated eventual consistency (global consensus) property for a blockchain network with a clique topology, a whole-system invariant implying the consensus in a quiescent state, and a proof of this invariant's inductivity, *i.e.*, preservation by the network semantics.

In this work, we focus exclusively on system safety properties, *i.e.*, proving that "nothing goes wrong". There are, indeed, more facts to establish about blockchain-based protocols, involving liveness (*aka* chain growth), probabilistic irrevocability, stronger notions of consistency, and various security properties [11, 20]. We do not address any of them in this paper, and consider statements and proofs of those properties as future applications of our formal model, discussing some of them in Section 7.

Our Coq development is publicly available online [29]:

https://github.com/certichain/toychain

Paper outline. We explain, by example, behaviours of blockchain networks in Section 2. We then describe the design and implementation of the core data structures, such as block forests, and their dependencies on the externally-provided security primitives in Section 3. In Section 4, we define the protocol machinery and the network semantics, elaborating on the statement and the proof of the consensus property in Section 5. We report on our mechanisation experience and lessons learned in Section 6. We then discuss limitations of our model in Section 7. We survey related verification and formalisation efforts in Section 8, and conclude in Section 9.

2 Overview

We begin by walking through an example that demonstrates interactions between nodes in a blockchain-based protocol and shows how consensus is achieved.

The goal of the consensus protocol is to guarantee that network participants agree *wrt.* the order in which transactions happened. This is achieved not by ordering transactions directly, but rather by grouping them into blocks and then agreeing, via *FCR*—a comparison operation on block sequences (chains), which resulting blockchain to adopt. Assuming an agreement upon the rules of the protocol and

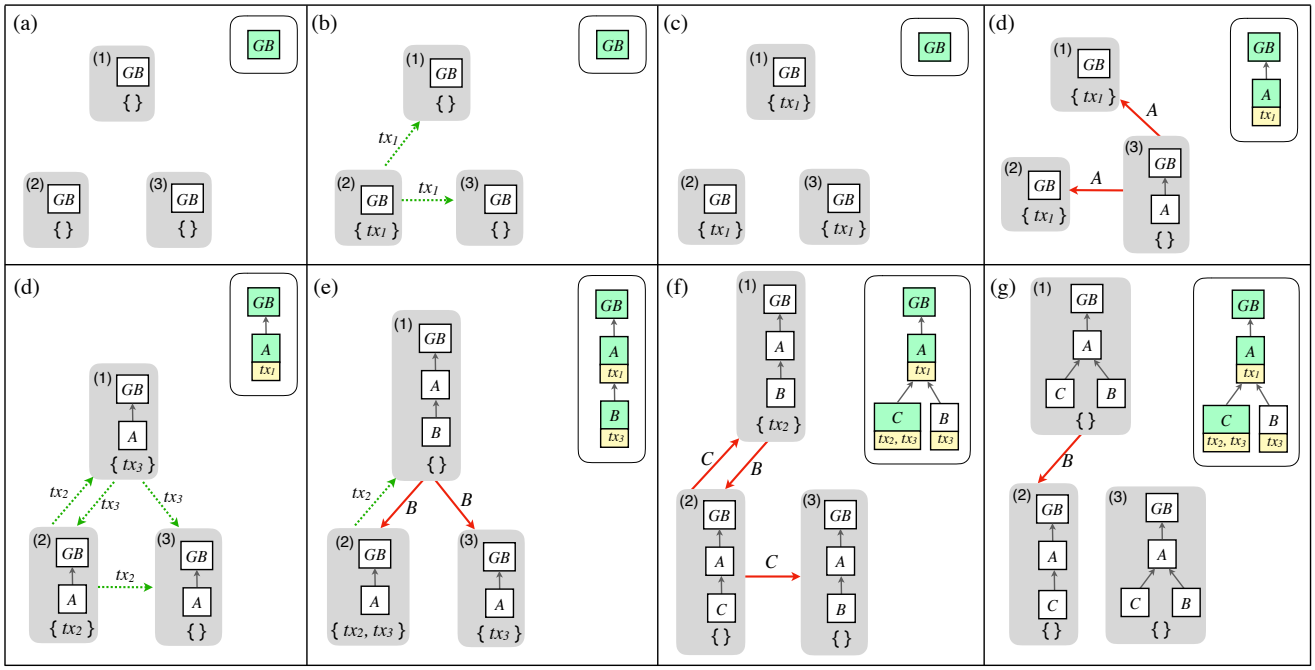

Figure 1. Progressive stages of interaction in a 3-node blockchain network, with "virtually shared" block-trees.

initial state of the system, (*i.e.*, everyone has the same *Genesis Block GB* in the local state, as shown in Figure 1(a)), and provided everyone follows the rules, consensus is guaranteed *once all block-carrying messages are delivered.*

The stages (b)–(g) of Figure 1 demonstrate further interactions in a 3-node blockchain-based system. Each stage also provides, in the top-right corner, the implicit "globally shared" tree of blocks, which will eventually be replicated in each of the participants and will serve to compute the globally shared ledger, thus delivering the desired ledger consensus. At any point in time, participants may issue transactions, which they broadcast to all their peers. For instance, a node (2) creates and announces a transaction tx_1 to its peers in (b). The messages containing the transactions propagate throughout the network, and each node temporarily stores them in its local *transaction pool* ($\{tx_1\}$ in stage (c)). When "minting" (*i.e.*, creating) a new block, nodes embed the transactions they know of inside the block and broadcast it to all participants in the network, as is done by node (3), which creates the block A with the transaction tx_1 in (d), setting A's "parent block" to be *GB*.

Since the system is distributed, multiple transactions can be issued and propagated concurrently. As a result, when a block is minted, the minter *does not necessarily know* of all pending transactions, as the messages containing them might still be in transit. This is what happens in stage (e), where the node (1), which creates block B, has yet no knowledge of tx_2. This delay in propagation also might cause certain transactions to be included in different, "conflicting" blocks. For example, in (f) the transaction tx_3 has been included in both block B, minted previously by node (1), and block C, which is newly-minted by node (2). As such, looking at the global tree view in the top-right corner of (f), tx_3 is part of two blockchains: $[GB, A, C]$ and $[GB, A, B]$. However, this is not an issue for the global agreement: as *FCR* imposes a *total order* on blockchains, it *uniquely determines* which of the candidate chains is the correct one—in this case, $[GB, A, C]$ (we decide so for the sake of this example, indicating this by the green colour in the "shared" tree). Thus, all transactions, including tx_3, appear only once within the correct chain. As more messages propagate, more participants agree on $[GB, A, C]$ to be the "correct", *i.e.*, canonical blockchain (g), until finally, everyone is agreement when the system is in a quiescent state with no messages in transit.

In the illustrative example above, we have seen how the system evolves over time and how it resolves conflicts by using the globally known *FCR* function. It is crucial for the purpose of eventually reaching a consensus that the *FCR* imposes a total order on all possible blockchains, such that conflicts are uniquely settled. Also note that *FCR* is computed locally by all participants, with no communication needed. Intuitively, these two properties put together imply that if two participants have the same blocks in their local "block forests", they will agree *wrt.* what the correct global chain is. Extended to an entire network, this means that when *all* blocks have been delivered, *all* nodes decide upon the same chain. That is, all nodes are in consensus.

In the following sections, we establish this formally.

$$\begin{aligned}
\mathsf{Time}, \mathsf{Addr} &\triangleq \mathbb{N} \\
b \in \mathsf{Block} &::= \{\, \texttt{prev} : \mathsf{Hash};\ \texttt{txs} : \mathsf{Tx}^*;\ \texttt{pf} : \mathsf{Proof} \,\} \\
c \in \mathsf{Chain} &\triangleq \mathsf{Block}^* \\
bf \in \mathsf{BForest} &\triangleq \mathsf{Hash} \xrightarrow{\text{fin}} \mathsf{Block} \\
tp \in \mathsf{TxPool} &\triangleq \mathcal{P}(\mathsf{Tx})
\end{aligned}$$

(a) Blocks, chains, and block forests

$$\begin{aligned}
&\mathsf{Hash}, \mathsf{Proof}, \mathsf{Tx} : \mathsf{eqType} \\
&GB : \mathsf{Block} \\
&hash_b : \mathsf{Block} \to \mathsf{Hash} \\
&hash_t : \mathsf{Tx} \to \mathsf{Hash} \\
&mkProof : \mathsf{Addr} \to \mathsf{Chain} \to \mathsf{option\ Proof} \\
&VAF : \mathsf{Proof} \to \mathsf{Time} \to \mathsf{Chain} \to \mathsf{bool} \\
&FCR : \mathsf{Chain} \to \mathsf{Chain} \to \mathsf{bool} \\
&txValid : \mathsf{Tx} \to \mathsf{Chain} \to \mathsf{bool} \\
&txExtend : \mathsf{TxPool} \to \mathsf{Tx} \to \mathsf{TxPool}
\end{aligned}$$

(b) Parameter primitive types and functions

Figure 2. Data structures and framework parameters.

3 Data Structures for Blockchains

We now present the data structures and primitives necessary for implementing the logic of the blockchain consensus protocol's replicated state machines.

3.1 Parameters and Core Data Structures

The top part of Figure 2 shows the definitions of the main data structures we are going to use. Some of the primitive data types are left undefined (*cf.* Figure 2 (b)), as they serve as parameters for the framework. For example, the types of time-stamps Time, necessary for modelling disciplines such as Proof-of-Stake, and network addresses Addr (ranged over by τ and a, correspondingly) are isomorphic to natural numbers $\mathbb{N}$. At the same time, the types Hash of hash values, proof objects Proof and transactions Tx (ranged over by h, pf, and tx, correspondingly) can be arbitrary (*e.g.*, natural numbers, strings, *etc*) as long as they come with decidable equality checking, which is indicated by the eqType type class annotation.

Blocks are represented as records with three fields. The first one, `prev`, stores the hash of some block (even the very same one, although in most cases such blocks will be deemed ill-formed). The field `txs` stores a sequence of transactions, contained within this block, in the order they are supposed to take place.[2] Finally, the proof object in `pf` is used to validate the block.

Two functions, $hash_t$ and $hash_b$, for computing hash-values of transactions and blocks correspondingly, are provided by

[2]For simplicity, we consider transactions located in the *same* block to be non-conflicting with each other.

$$\begin{aligned}
&hash_inj &&: \forall x\ y,\ \#x = \#y \implies x = y \\
&GB_hash &&: \texttt{prev}\ GB = \#GB \\
&GB_txs &&: \texttt{txs}\ GB = [\,] \\
&VAF_nocycle &&: \forall b\ \tau\ c,\ VAF\ (\texttt{pf}\ b)\ \tau\ c \implies b \notin c \\
&FCR_rel &&: \forall c_1\ c_2, c_1 = c_2 \vee c_1 > c_2 \vee c_2 > c_1 \\
&FCR_trans &&: \forall c_1\ c_2\ c_3, c_1 > c_2 \wedge c_2 > c_3 \implies c_1 > c_3 \\
&FCR_nrefl &&: \forall c, c > c \implies \mathsf{False} \\
&FCR_ext &&: \forall c_1\ c_2\ b, c_1 +\!\!+ (b :: c_2) > c_1 \\
&FCR_subch &&: \forall c_1\ c_2, c_1 \prec c_2 \implies c_2 \geq c_1 \\
&txValid_nil &&: \forall tx, txValid\ tx\ [\,] = \mathsf{true}
\end{aligned}$$

Figure 3. Axioms of the framework parameters.

the client of the framework. For the sake of brevity, in the remainder of the paper, we will use the overloaded notation $\#x$ for computing the hash of a value x, which is either a block or a transaction, using the corresponding hash-function. The only requirement imposed on the hash function is it being injective, as stated by the axiom *hash_inj* in Figure 3.

We require the client of the framework to provide a dedicated *Genesis Block GB*, which is going to serve as an initial "seed" for all local ledgers, and is globally known in the system. This block is slightly different from other blocks we will observe in the system, as it is a subject of the first two hypotheses (axioms) we impose, which are shown in Figure 3. First, the hash of *GB* should be equal to *GB*'s `prev` value (*GB_hash*). This requirement seems quite artificial, but it is easy to achieve in practice by redefining the result of a typical hash-function for just one value, and it simplifies reasoning about forests, essentially ensuring that there can be no parent block of *GB*, in the presence of possible cycles between other (ill-formed) blocks referring to each other. An alternative to this construction would be to make a block's `prev` field optional, and ensure the GenesisBlock has no parent. The second axiom *GB_txs* ensures that the genesis block contains an empty transaction sequence [ ].

Blockchains (or simply *chains*) are defined as sequences of blocks, and block forests (or *forests*) are encoded as partial finite maps from hashes to blocks. The notions of "valid" chains and forests will follow below.

The next two parameter functions *mkProof* and *VAF* work in tandem: the former is used to obtain a proof object for a specific node and on top of a particular chain, and it might fail, hence the option result type. In real-world blockchain implementations, computing a value of *mkProof* is an expensive operation, as it controls the number of valid blocks in the system, but here we do not model computational complexity, as it is irrelevant for establishing consensus, which is a safety, not a liveness property. Dually, *VAF* is used to *validate* proof objects for a chain and it also takes a system-provided time-stamp as an additional parameter. The only *VAF*-related axiom we need, *VAF_nocycle*, ensures that

a freshly "minted" (*i.e.*, created) block b, for which a proof has been obtained *wrt.* to an "underlying" chain c, cannot be contained in the same c. An opposite situation would be an anomaly, and does not occur in real situations, in part due to the practical rareness of hash collisions.

The client-provided function *FCR* allows one to compare the weights of two chains. From now on, we will abbreviate ($FCR\ c_1\ c_2$) as $c_1 > c_2$. The axioms *FCR_rel*, *FCR_trans*, *FCR_nrefl* ensure that the order *FCR* imposes on chains is total, transitive, and irreflexive. The axiom *FCR_ext* states that any non-empty extension of a chain c_1 produces a strictly "heavier" chain. In turn, *FCR_subch* postulates that if a chain c_1 is a subchain of c_2 (*i.e.*, $c_1 \prec c_2 \triangleq c_2 = c' +\!\!+ c_1 +\!\!+ c''$ for some, possibly empty, c', c''), then c_2 is at least as "heavy" as c_1.

Finally, the transaction validation function *txValid* ensures the absence of conflicts between a transaction *tx* and a preceding chain c, being always true for an empty chain (as asserted by *txValid_nil*), and *txExtend*, which we did not have to constrain, is used to change a pool of pending transactions held by a particular node.

3.2 Largest Chains and Block Forest Evolution

Block forests are the main data structures nodes use to store incoming and locally minted blocks, and to reconstruct the actual *ledger* of transactions. The ledger of a forest *bf*, typeset as $\lceil bf \rceil$, is defined as the *largest* (*wrt. FCR*) chain starting at *GB* and ending with some block b, which has a corresponding entry in *bf*.

How do we construct such a chain? To do so, we should restrict the class of forests *bf* we are working with to those satisfying the following three properties:

$$\begin{array}{ll} 1. & \forall h_1, h_2 \in \mathrm{dom}(bf), h_1 = h_2 \Rightarrow bf(h_1) = bf(h_2); \\ 2. & \forall h\ b, bf(h) = b \Rightarrow h = \#b; \\ 3. & bf(\#GB) = GB. \end{array} \quad (1)$$

The first property states that every key in *bf* uniquely identifies its entry; the second ensures that for every block-entry in *bf*, its key is a hash of the corresponding block; finally, the third property makes sure that *bf* contains the Genesis Block with its key. We define a forest *bf* satisfying 1–3 as *valid*(*bf*) and will denote a block b having a corresponding entry $\#b \mapsto b$ in a valid forest *bf* as $b \in bf$, slightly abusing the $\in$-notation.

In the beginning of a system interaction, each node holds the same forest $bf_0 = \{\#GB \mapsto GB\}$, which is trivially valid. As the nodes start minting new blocks and broadcast them, local forests might be extended with new blocks, for which we define the following operation:

$$bf \triangleleft b \triangleq \textbf{if } \#b \in \mathrm{dom}(bf) \textbf{ then } bf \textbf{ else } bf \uplus \{\#b \mapsto b\} \quad (2)$$

That is, for any block, the result of $bf \triangleleft b$ is valid, if so was *bf*. Thus, fixing $\cdot \triangleleft \cdot$ as the only way to add a new block to a forest, in the rest of the paper we will be only dealing with valid forests, unless said otherwise.

Let us now compute the largest chain in a forest *bf*. Indeed, even a valid forest might not be a tree, due to gaps and possible cycles in the partial map that encodes it. We model cycles, even though they are implausible in a real-world setting, to account for the possibility that the hash functions used by the protocol might not be cryptographic. "Gaps", on the other hand, will appear frequently, as they correspond to blocks received out-of-order. To be considered a ledger candidate, a chain c should satisfy the following conditions:

1. It should contain no duplicate blocks;
2. For any block $b \in c$, $b = GB$ *or* `prev` $b = b'$, where b' is the block preceding b in c;
3. The first block of c should be *GB* (*gb-founded c*);
4. For any block $b \in c$, and any transaction $tx \in$ `txs` b, *txValid tx* c' should be True, where c' is a prefix of c, preceding b (*tx-valid c*).

To deliver such a candidate, we first construct a total function *chain*(*bf*, b) that returns a chain c, ending with a block b (or just one-element chain $[GB]$ if $\#b \notin \mathrm{dom}(bf)$) satisfying the conditions 1 and 2 above by implementing a "backwards walk" by `prev`-links from b in *bf* and ensuring that we do not visit the same block twice. Such a walk terminates if we encounter a cycle in *bf* or we reach *GB*, which is its own previous block. The exact code of *chain* can be found in our supplementary Coq sources. Using *chain*, we construct candidates considering *all* blocks in *bf* and choose the largest one from those that satisfy conditions 3 and 4. In set notation, for a valid *bf*, $\lceil bf \rceil$ is defined as follows:

$$\lceil bf \rceil \triangleq \max_{FCR} \left\{ c \;\middle|\; \begin{array}{l} c = chain(bf, b), b \in bf, \\ gb\text{-}founded\ c \ \wedge\ tx\text{-}valid\ c \end{array} \right\} \quad (3)$$

The function $\lceil bf \rceil$ is implemented to be total so it returns $[GB]$ as a default ledger, if no better one is found.

To get a better intuition on the dynamics of $\lceil bf \rceil$ as the forest *bf* keeps being extended with new blocks, let us take a look at Figure 4, which shows several states of a valid block forest with `prev`-links depicted by gray arrows. The stage (a) depicts a valid forest whose ledger is $c = [GB, A, B, C]$, with all other chains being less heavy or prefixes of c. In stage (b), due to out-of-order arrival, a block G has been added to the forest, but at that moment it is orphaned, hence a chain built from it is not *gb-founded*. Once the missing block F arrives in stage (c), a the forest gets a new ledger, namely $[GB, D, E, F, G]$, as it is more *FCR*-heavy than $[GB, A, B, C]$. Stages (d) and (e) show block forests that we account for in our implementation, but that will not correspond to local states of the protocol participants during a normal, non-Byzantine execution (we discuss Byzantine cases in Section 7). Specifically, the forest in (d) has a block M, which is non-*tx-valid wrt.* $[GB, A, B, C]$, and therefore does not contribute a ledger candidate, preventing all further chains

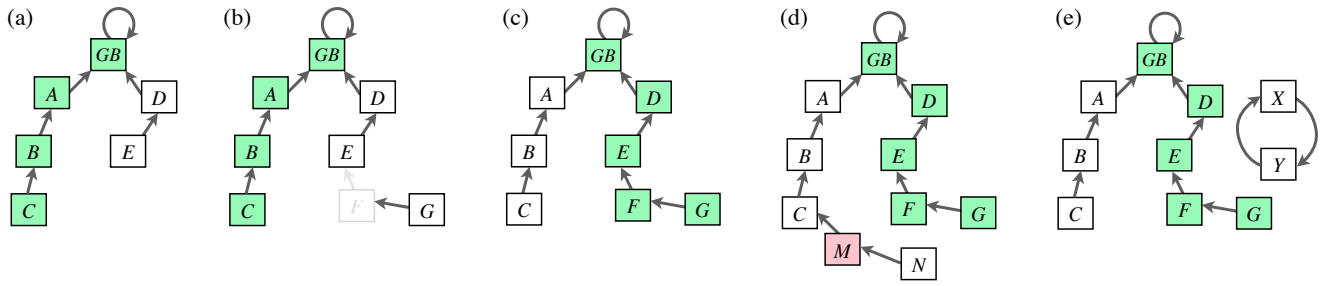

Figure 4. Different states of a valid block forest and its ledger chain (green) when extended with more blocks.

in the forest that include it from being considered for the largest ledger. Finally, the case (e) demonstrates a cycle between two blocks, X and Y, so neither of them is included into any chain to be considered a ledger candidate. In Section 4, we will show how the protocol prevents cases (d) and (e) from happening, when all participants are faithful.

3.3 Local Forests and System-Wide Union

The cases (a)–(c) from Figure 4 depict possible configurations of local forests, owned by particular participants of a blockchain protocol in the middle of a system's execution, when some of the blocks have already been received, while some others are in "in-flight" messages, *i.e.*, yet to be delivered. Furthermore, in the absence of Byzantine participants, a *system-wide union* $\widehat{bf}$ of all locally owned forests will be *defined*, *valid*, and furthermore satisfy the following property, dubbed $good(\widehat{bf})$:

$$\begin{aligned} &good(bf) \triangleq \forall b \in bf, \\ &\quad gb\text{-}founded(chain(bf, b)) \wedge tx\text{-}valid(chain(bf, b)) \end{aligned} \tag{4}$$

In other words, a good forest is a *tree*, such that *GB* is its root, and a chain built from any of its blocks starts with *GB* and has no blocks with invalid transactions. A pleasant consequence of the fact $gb\text{-}founded(chain(bf, b))$ is that the chain *has no gaps*, *i.e.*, the result of $chain(bf, b)$ will not be affected by adding new blocks to bf, which is stated formally by the following lemma:

Lemma 3.1. *For a valid forest bf and a block b, if* gb-founded($chain(bf, b)$) *then for any block* b', $chain(bf \triangleleft b', b) = chain(bf, b)$.

The following theorem is key for showing that locally minting a new block, when done right, *always increases* the local ledger, and, thus, has a chance to increase the global one, *i.e.*, the ledger of a system-wide forest union.

Theorem 3.2. *If a forest bf is valid and a block b is such that*

1. tx-valid($\lceil bf \rceil, b$),
2. `prev` $b = \#(last \lceil bf \rceil)$, *and*
3. *VAF* (`pf` b) τ ($\lceil bf \rceil$) *for some* τ,

then $\lceil bf \triangleleft b \rceil > \lceil bf \rceil$.

That is, if a block b is minted to extend the current bf's ledger, adding b will deliver a heavier one.

For a (multi-)set of blocks $bs = \{b_1, b_2, \ldots, b_n\}$ and a forest bf, we define the operator $bf \lhd\!\lhd bs$ as follows:

$$bf \lhd\!\lhd \{b_1, b_2, \ldots, b_n\} \triangleq (\ldots((bf \triangleleft b_1) \triangleleft b_2)\ldots) \triangleleft b_n \tag{5}$$

The definition (2) of $\triangleleft$ implies that reordering or duplication of blocks in bs does not affect the result of $\cdot \lhd\!\lhd bs$. We define the partial order $\sqsubseteq$ on *valid* forests as follows:

$$bf_1 \sqsubseteq bf_2 \triangleq \exists bs, bf_2 = bf_1 \lhd\!\lhd bs \tag{6}$$

We conclude this section with two theorems that are crucial for relating changes in a "node-local" ledger, due to minting a new block *wrt.* a node-local forest bf, and a "global" ledger built from the system-wide union of forests $\widehat{bf}$, such that $bf \sqsubseteq \widehat{bf}$. The first theorem states that, if a new block is global goodness-preserving, and if the *new* local ledger is *heavier than* the *old* global one, then the new *local* ledger *is* the new *global* one:

Theorem 3.3. *For valid forests bf,* $\widehat{bf}$ *and a block b, if* $good(\widehat{bf})$, *and* $good(\widehat{bf} \triangleleft b)$, *and* $\lceil bf \triangleleft b \rceil > \lceil \widehat{bf} \rceil$, *and* $bf \sqsubseteq \widehat{bf}$, *then* $\lceil bf \triangleleft b \rceil = \lceil \widehat{bf} \triangleleft b \rceil$.

The second theorem states that a correctly locally minted block b, if it does not contribute to create a ledger heavier than the *current* global one, will not change the global ledger even when added to the global forest:

Theorem 3.4. *For valid forests bf,* $\widehat{bf}$ *and a block b, if* $good(\widehat{bf})$, *and* $good(\widehat{bf} \triangleleft b)$, *and* tx-valid($\lceil bf \rceil, b$), *and* `prev` $b = \#(last \lceil bf \rceil)$, *and VAF* (`pf` b) τ ($\lceil bf \rceil$), *and* $bf \sqsubseteq \widehat{bf}$, *and* $\lceil \widehat{bf} \rceil \geq \lceil bf \triangleleft b \rceil$, *then* $\lceil \widehat{bf} \triangleleft b \rceil = \lceil \widehat{bf} \rceil$.

In a foresight of the development to be presented in the following sections, it should be intuitively clear that a ledger $\lceil \widehat{bf} \rceil$ of a system-wide forest union is what consensus is going to be reached upon. That is, each node, if it follows the rules of minting and adopting blocks, will eventually "align" its local ledger with $\lceil \widehat{bf} \rceil$.

4 The Protocol and its Semantics

From the description of the core data structures, we proceed to outline the logic of the protocol itself, represented as a family of replicated state-transition systems which communicate by asynchronously exchanging packets.

4.1 System State-Space

Figure 5 shows all state-space components of our system encoding. System configurations σ are pairs of a *global state* Δ and a *packet soup* P. The former is a finite partial mapping from node addresses to their local states; described further below, while the latter is a (multi-)set of packets.[3] Packets are simply triples, with the first component being the message sender, the second is the address of the destination, and the third one is the message content. We will further refer to the destination address of a package p as *dest* p.

A local state δ of a node is a quadruple $\langle \mathsf{this}, as, bf, tp \rangle$. Its first component this is the address of the node itself, which coincides with the key of this node in the system's global state; the second component is a set of addresses as representing the *peers* the node is aware of; the third component is a block forest bf, described in detail in Section 3, used to store the minted and received blocks; finally, the last component is a pool of transactions tp, storing locally created or received transactions to be included into minted blocks in the future.

Specific contents of the messages the nodes can send are defined by the data type Msg and include: NullMsg, which has no effect and is a message-passing analogue of the imperative skip command; ConnectMsg used by a node to announce itself to its peers; AddrMsg as used to propagate the set of peers as further into the system. TxMsg tx is used to announce or propagate a new transaction tx, and BlockMsg b serves the same purposes *wrt.* announcing a block b. InvMsg hs is sent to inform others of the transactions and blocks a node holds locally (represented by their hashes hs); GetDataMsg h is a request for a transaction or a block with a hash h, sent after having received an InvMsg.

In addition to emitting messages, nodes can perform internal operations, and we only capture the two that are relevant to the protocol in our semantics: creating and announcing a transaction, and minting a new block. A data type Instr for instructions serves to encode these two operations. We will use instructions to encode node-specific internal choices for modeling non-determinism in global network executions (described in Section 4.3) by parameterising them with *schedules*—finite sequences of *selectors* determining which node should act next and what is going to be its move.

[3] Our semantics is resilient with respect to packet duplication, so here for simplicity we assume packet soups to be sets, while in or mechanisation they are modeled as multi-sets.

System configurations

$$\Delta \in \mathsf{GlobState} \triangleq \mathsf{Addr} \xrightarrow{\text{fin}} \delta$$
$$P \in \mathsf{PacketSoup} \triangleq \mathcal{P}(\mathsf{Packet})$$
$$\sigma \in \mathsf{Conf} \triangleq \mathsf{GlobState} \times \mathsf{MessageSoup}$$

Local states

$$\delta \in \mathsf{LocState} \triangleq \mathsf{Addr} \times \mathcal{P}(\mathsf{Addr}) \times \mathsf{BForest} \times \mathsf{TxPool}$$

Messages, instructions and schedules

$$p \in \mathsf{Packet} \triangleq \mathsf{Addr} \times \mathsf{Addr} \times \mathsf{Msg}$$
$$m \in \mathsf{Msg} ::= \mathsf{NullMsg} \mid \mathsf{ConnectMsg} \mid \mathsf{AddrMsg}\ (as \in \mathcal{P}(\mathsf{Addr})) \mid \mathsf{TxMsg}\ (tx : \mathsf{Tx}) \mid \mathsf{BlockMsg}\ (b : \mathsf{Block}) \mid \mathsf{InvMsg}\ (hs \in \mathcal{P}(\mathsf{Hash})) \mid \mathsf{GetDataMsg}\ (h : \mathsf{Hash})$$
$$i \in \mathsf{Instr} ::= \mathsf{DoTx}\ (tx : \mathsf{Tx}) \mid \mathsf{DoMint}$$
$$s \in \mathsf{Selector} ::= \mathsf{SelIdl} \mid \mathsf{SelRcv}\ (a : \mathsf{Addr}) \mid \mathsf{SelInt}\ (a : \mathsf{Addr})\ (\tau : \mathsf{Time})\ (i : \mathsf{Instr})$$
$$sc \in \mathsf{Schedule} \triangleq \mathsf{Selector}^*$$

Figure 5. System state-space and schedules.

4.2 Local Node Semantics

Figures 6 and 7 show the per-node transitions. While in our implementation they are encoded as executable Coq functions, in this paper we follow a more traditional relational style of presenting an operational semantics.

We split the semantics into receive-transitions and internal transitions. The former ones are of the form $\delta \xrightarrow[\rho]{p} (\delta', ps)$ taking a node from a state δ and to δ' when processing a package p, also emitting a new set of packages ps. The latter ones, $\delta \xrightarrow{i,\tau}_{\iota} (\delta', ps)$, describe change of a state from δ and δ' with emission of packages ps as a result of executing the instruction i at a globally synchronised time τ.

The receive-transitions (Figure 6) follow the intuition of the corresponding messages, and are mostly straightforward, so we only describe a few in prose. When taking a RcvAddr step, a node not only adds the new addresses to its local pool, but also sends a ConnectMsg-request to the new peers it has learned about (ps_1) and propagates the new information to its current peers (ps_2), which will then themselves connect to the new peers (if they hadn't already). When receiving a new transaction or a block (via RcvTx or RcvBlock), a node adds it to its local state and informs its peers of now possessing it by sending an InvMsg, potentially causing a lot of

Receive-step transitions: $\delta \xrightarrow{p}_{\rho} (\delta', ps)$

$$\textsc{RcvNull} \quad \delta \xrightarrow{\text{from, this, NullMsg}}_{\rho} (\delta,)$$

$$\textsc{RcvConnect} \quad \frac{as' = as \cup \{\text{from}\} \qquad hs = \text{dom}(bf) \cup \{\#tx \mid tx \in tp\} \qquad ps = \{\text{this, from, InvMsg } hs\}}{\text{this}, as, bf, tp \xrightarrow{\text{from, this, ConnectMsg}}_{\rho} (\text{this}, as', bf, tp, ps)}$$

$$\textsc{RcvAddr} \quad \frac{as_1 = \{a \mid a \in as' \wedge a \notin as\} \quad as_2 = as \cup as_1 \quad ps_1 = \{\text{this}, a, \text{ConnectMsg} \mid a \in as_1\} \quad ps_2 = \{\text{this}, a, \text{AddrMsg } as_2 \mid a \in as\} \quad ps = ps_1 \cup ps_2}{\text{this}, as, bf, tp \xrightarrow{\text{from, this, AddrMsg } as'}_{\rho} (\text{this}, as_2, bf, tp, ps)}$$

$$\textsc{RcvTx} \quad \frac{tp' = txExtend\ tp\ tx \qquad hs = \text{dom}(bf) \cup \{\#tx' \mid tx' \in tp'\} \qquad ps = \{\text{this}, a, \text{InvMsg } hs \mid a \in as\}}{\text{this}, as, bf, tp \xrightarrow{\text{from, this, TxMsg } tx}_{\rho} (\text{this}, as, bf, tp', ps)}$$

$$\textsc{RcvBlock} \quad \frac{bf' = bf \triangleleft b \qquad tp' = \{tx \mid tx \in tp \wedge txValid\ tx\ \lceil bf' \rceil\} \qquad hs = \text{dom}(bf') \cup \{\#tx \mid tx \in tp'\} \qquad ps = \{\text{this}, a, \text{InvMsg } hs \mid a \in as\}}{\text{this}, as, bf, tp \xrightarrow{\text{from, this, BlockMsg } b}_{\rho} (\text{this}, as, bf', tp', ps)}$$

$$\textsc{RcvInv} \quad \frac{hs_1 = \text{dom}(bf) \cup \{\#tx \mid tx \in tp\} \qquad hs' = hs \setminus hs_1 \qquad ps = \{\text{this, from, GetDataMsg } h \mid h \in hs'\}}{\text{this}, as, bf, tp \xrightarrow{\text{from, this, InvMsg } hs}_{\rho} (\text{this}, as, bf, tp, ps)}$$

$$\textsc{RcvGetData} \quad \frac{bs = \{b \mid b = bf(h)\} \qquad txs = \{tx \mid tx \in tp \wedge \#t = h\} \qquad m = \left(\begin{array}{l}\textbf{if } bs = \{b\} \textbf{ then BlockMsg } b \textbf{ else} \\ \quad \textbf{if } txs = \{tx\} \textbf{ then TxMsg } tx \textbf{ else NullMsg}\end{array}\right) \qquad ps = \{\text{this, from}, m\}}{\text{this}, as, bf, tp \xrightarrow{\text{from, this, GetDataMsg } h}_{\rho} (\text{this}, as, bf, tp, ps)}$$

Figure 6. Local semantics, Part I: receive-transitions.

redundant messages, which are nevertheless handled without any concerns for safety. Note that RcvBlock does not check whether the block it receives is valid before adding it to the local block forest. This seems unusual, but in reality is the only possible option, because a block's validity depends on the blocks that precede it, which the node may not yet have. Finally, the last two transitions serve to inform a node of new transactions and blocks in the system (via RcvInv), so it could request them by sending a GetDataMsg message, and the response to it will be sent (via RcvGetData) in the form of TxMsg or BlockMsg.

It is perhaps slightly non-obvious, but the rules allow to model the possibility of a node "joining late" and eventually "catching up" with the rest of the system, thanks to

Internal step transitions: $\delta \xrightarrow{i,\tau}_{\iota} (\delta', ps)$

$$\textsc{IntTx} \quad \frac{ps = \{\langle \text{this}, a, \text{TxMsg } tx\rangle \mid a \in as\}}{\langle \text{this}, as, bf, tp\rangle \xrightarrow{\text{DoTx } tx,\tau}_{\iota} (\langle \text{this}, as, bf, tp\rangle, ps)}$$

$$\textsc{IntMint} \quad \frac{mkProof\ \text{this}\ \lceil bf \rceil = \text{Some } pf \qquad VAF\ pf\ \tau\ \lceil bf \rceil = \text{true} \qquad b = \left\{\begin{array}{ll}\texttt{prev} & := \#(last\ \lceil bf \rceil); \\ \texttt{txs} & := [\ tx \mid tx \in tp \wedge txValid\ t\ \lceil bf \rceil\ ]; \\ \texttt{pf} & := pf\end{array}\right\} \qquad bf' = bf \triangleleft b \qquad ps = \{\langle \text{this}, a, \text{BlockMsg } b\rangle \mid a \in as\} \qquad tp' = \{tx \mid tx \in tp \wedge txValid\ tx\ \lceil bf' \rceil\} \setminus (\texttt{txs}\ b)}{\langle \text{this}, as, bf, tp\rangle \xrightarrow{\text{DoMint},\tau}_{\iota} (\langle \text{this}, as, bf', tp'\rangle, ps)}$$

Figure 7. Local semantics, Part II: internal transitions.

RcvConnect and other transitions that send known information about blocks transactions to the package origin from, so it could request them via RcvInv.

Figure 7 shows the two internal transitions that are triggered by the corresponding instructions. The IntTx simply adds a new transaction to the local pool, so it could be included into a block later, and announces it to the node's peers. The IntMint transition relies on the block forest machinery and related primitives described in the previous section. Specifically, we (rather optimistically) assume that a node locally checks the new minted block b with respect to its prefix chain, before adding it to its local forest and sending it to its peers.

With the rules in Figures 6 and 7, we intentionally define a non-optimal version of the protocol, such that nodes executing the transitions populate the packet soup with a lot of redundant messages. Yet, as we will show in Section 5, this does not pose problems for establishing consensus on the state of the global ledger.

4.3 Network Semantics

The network semantics rules, parameterised by a selector s, are shown in Figure 8. They are standard for modeling interleaved concurrency with non-deterministic internal choices and message delivery. The three rules account for a possibility of delivering a randomly picked package p from the soup P to a destination a (NetDeliver), a node a taking an internal step with an instruction i (NetInternal) or doing nothing (NetIdle).

While the rules do not change the global set of node addresses, we nevertheless can model a scenario of a node "joining" the network, assuming that it already has a predefined address and a correctly initialised initial state, so it

Network transitions: $\langle \Delta, P\rangle \overset{s}{\Longrightarrow} \langle \Delta', P'\rangle$

$$\textsc{NetDeliver}\quad \frac{p \in P \qquad dest\ p = a \qquad \Delta(a) = \delta \qquad \delta \xrightarrow{p}_{\rho} (\delta', ps)}{\langle \Delta, P\rangle \xRightarrow{\text{SelRcv } a} \langle \Delta[a \mapsto \delta'], P \setminus \{p\} \cup ps\rangle}$$

$$\textsc{NetInternal}\quad \frac{\Delta(a) = \delta \qquad \delta \xrightarrow{i,\tau}_{\iota} (\delta', ps)}{\langle \Delta, P\rangle \xRightarrow{\text{SelInt } a\ \tau\ i} \langle \Delta[a \mapsto \delta'], P \cup ps\rangle}$$

$$\textsc{NetIdle}\quad \langle \Delta, P\rangle \xRightarrow{\text{SelIdl}} \langle \Delta, P\rangle$$

Figure 8. Network semantics.

only needs to announce itself to its peers and requests the information about transactions and blocks.[4]

We conclude this section by defining the notion of *reachability* ($\rightsquigarrow$) between two configurations as follows:

$$\begin{aligned}\sigma \rightsquigarrow \sigma' \triangleq \sigma = \sigma' \vee \\ \exists sc = [s_1, \ldots, s_n], [\sigma_1, \ldots, \sigma_{n-1}],\ s.t. \\ \sigma \overset{s_1}{\Longrightarrow} \sigma_1 \wedge \ldots \wedge \sigma_{n-1} \overset{s_n}{\Longrightarrow} \sigma'.\end{aligned} \tag{7}$$

5 System Safety and Consensus

With the definitions of the protocol and a library of theorems about block forests at hand, we are now ready to establish several important safety properties, including the eventual consistency (*i.e.*, the consensus) of our system. It is customary to formulate safety properties as *inductive system invariants*, defined as follows:

Definition 5.1. The property $I : \text{Conf} \to \text{Prop}$ is an *inductive invariant* of a system if for the system's initial configuration σ_0, $I(\sigma_0)$ holds, and for any σ, σ' and s, such that $I(\sigma)$ holds, $\sigma \overset{s}{\Longrightarrow} \sigma'$ implies $I(\sigma')$.

Therefore, by induction, an inductive property I will hold for any system configuration σ, such that $\sigma_0 \rightsquigarrow \sigma$. Indeed, what can be proven inductive depends on the choice of the initial system state, which we have not specified so far. For the rest of this section, we will consider only the initial configurations of the form $\sigma_0 = \langle \text{GlobState}_0, \emptyset\rangle$, where for any node $a \in \text{dom}(\text{GlobState}_0)$, $\text{GlobState}_0(a) = \langle a, as_a, \{\#GB \mapsto GB\}, \{\ \}\rangle$, *i.e.*, leaving only the node-specific sets of peers as_a unconstrained.

[4] We could have added another internal transition rule for emitting a ConnectMsg, but this is orthogonal to our study of system safety.

5.1 System State Coherence

Before moving to the interesting (and, hence, complex) system safety properties, we start by establishing the inductivity of global state *coherence, i.e.*, proving that interaction between nodes does not violate the validity of the components of each node's local state. We thus define the global system state coherence as follows:

$$\begin{aligned}\text{Coh}(\langle \Delta, -\rangle) \triangleq \forall a \in \text{dom}(\Delta), \exists as\ bf\ tp, \\ \Delta(a) = \langle a, as, bf, tp\rangle \wedge valid(bf)\end{aligned} \tag{8}$$

The validity of each local forest bf is via the definition (1). Any of the σ_0 we consider satisfies it, and the property Coh is inductive, because all manipulations with node-local block forests are done using the $\triangleleft$ operation (2).

5.2 Eventual Ledger Consistency

Let us now formulate the eventual consistency of the system. Informally, it says that when there are no in-flight messages between any of the nodes, they all should agree on the local ledger, which can be, thus, thought of as a *globally shared* log of accepted transactions [33].

In practice, however, communication between nodes never stops. Our protocol features many "modes of communication" (announcing a block, requesting hashes, *etc*), and, as it turns out, not all of them should be ceased for reaching consensus on ledgers. What is essential is to have *no in-flight instances of* BlockMsg. [5] Having no in-flight block-messages, however, is not the only requirement for the *universality* of the consensus (*i.e.*, ensuring that each two nodes have the same ledger): it might be the case that some nodes joined late, and due to the delays in updating the topology, have not yet requested all missing data from their peers. Characterising consistency conditions in this case would require us to take the "late joiners" into account. While not impossible, this would make the whole consistency statement quite complicated. To avoid this, in this paper we decided to formulate the consistency in a simpler setting: a *clique network topology*, restricting the initial configurations to those where every node's known peers include *all* addresses in the global system state.[6]

We embed the clique topology assumption into the whole-system property Cliq, whose formal definition we postpone until Section 5.3. For now, let us present the eventual consistency result it implies. For this, we introduce two auxiliary definitions. The first one extracts a ledger for a node $a \in \text{dom}(\Delta)$ from a global state Δ.

$$ledger(\Delta, a) \triangleq \lceil bf \rceil,\ \text{s.t. } \langle a, -, bf, -\rangle = \Delta(a) \tag{9}$$

The second returns all in-flight blocks for a in a soup P:

$$blocksFor(P, a) \triangleq \{b \mid \langle -, a, \text{BlockMsg } b\rangle \in P\} \tag{10}$$

[5] The version we present is a form of *quiescent* consistency [2, 5].

[6] This situation is quite common for corporate blockchain-based protocols, where all peers know each other from the very beginning.

$$
\begin{aligned}
&\mathsf{Cliq}(\langle \Delta, P\rangle) \triangleq \\
&\quad \mathsf{Coh}(\langle \Delta, P\rangle) \wedge \\
&\quad \forall a \in \mathsf{dom}(\Delta), \mathsf{dom}(\Delta) \subseteq \mathit{peers}(\Delta, a) \wedge \\
&\quad \exists c, \widehat{bf}, \widehat{a} \in \mathsf{dom}(\Delta), \text{ such that} \\
&(i)\ \ \forall a \in \mathsf{dom}(\Delta), \widehat{bf} = \mathit{forest}(\Delta, a) \lhd \mathit{blocksFor}(P, a) \wedge \\
&(ii)\ \ \mathit{valid}(\widehat{bf}) \wedge \mathit{good}(\widehat{bf}) \wedge c = \lceil \widehat{bf} \rceil \wedge \\
&(iii)\ \ \forall a \in \mathsf{dom}(\Delta), c \geq \mathit{ledger}(\Delta, a) \wedge \\
&(iv)\ \ \mathit{ledger}(\Delta, \widehat{a}) = c \\
&\quad \text{where } \mathit{peers}(\Delta, a) \triangleq as, \text{ s.t. } \langle a, as, -, -\rangle = \Delta(a) \\
&\qquad\ \ \mathit{forest}(\Delta, a) \triangleq bf, \text{ s.t. } \langle a, bf, -, -\rangle = \Delta(a)
\end{aligned}
$$

Figure 9. Cliq system property.

The desired theorem is as follows:

Theorem 5.2 (Consensus in a clique topology)**.** *For $\sigma = \langle \Delta, P\rangle$, if* $\mathsf{Cliq}(\sigma)$ *holds, then there exists a chain c, such that for any node $a \in \mathsf{dom}(\sigma)$, $c \geq \mathit{ledger}(\Delta, a)$ and $\mathit{blocksFor}(P, a) = \emptyset$ implies $\mathit{ledger}(\Delta, a) = c$.*

The chain c from Theorem 5.2 statement is a *globally shared ledger*, and in a quiescent state, all nodes have it.

5.3 Clique Invariant

We now show the statement of the Cliq property, highlight its key insights, and convey the intuition of the proof that it is indeed inductive for systems that start in initial configurations σ_0 with a clique network topology.

The formal definition of Cliq, with the most important conjuncts labelled (i)–(iv) is given in Figure 9. The first two non-labelled conjuncts ensure that the property holds over configurations that are coherent (8) and have a clique topology, as discussed above. The rest of the definition is more interesting, as it exhibits an important property of blockchain-based protocols, which we call *the law of block conservation*. The "conservation" is expressed via the existence of a global forest $\widehat{bf}$ (foreshadowed as a system-wide forest union in Section 3.3), which is a superset of the local forest of any node a, as stated by conjunct (i), and can be obtained by adding all blocks currently in-flight towards a to a's local forest. The global forest $\widehat{bf}$ is also *valid* (1), *good* (4) and has the "canonical" ledger c (ii), which is larger or equal than any local ledger (iii). Finally, there is *always* a node $\widehat{a} \in \mathsf{dom}(\Delta)$ that *has* the canonical ledger c, even though $\widehat{a}$'s local forest might be a strict subset of $\widehat{bf}$ (iv).

The statement of Theorem 5.2 trivially follows from (i)–(ii) as then the subject node's forest is exactly $\widehat{bf}$.

Why is* Cliq *inductive? In our Coq development, we have proved that Cliq is preserved by the network semantics. The proof is of interest, as it heavily relies on the idempotence of the $\lhd$ operation, and the "goodness" of the global forest $\widehat{bf}$, whose ledger c (owned by at least one node in the network) serves as the *constructive witness* of what the consensus *is going to be reached upon*. The trickiest parts of the proof concern "restoring" conjuncts (ii) and (iv) when an arbitrary node takes the IntMint transition, with a chance of either (a) becoming the new owner $\widehat{a}$ of the global ledger c, or (b) minting a block that is already in $\widehat{bf}$ or simply does not deliver a heavier chain. The case (a) is handled by Theorem 3.3, while the case (b) is what is delivered by Theorem 3.4. The following theorem therefore holds:

Theorem 5.3. *For systems that initially have a clique network topology,* Cliq *is an inductive invariant.*

On the clique assumption. What would the invariant and the eventual consistency statements look like without the clique assumption? At the moment, the definition of Cliq ensures that for *any* node a and any block b in the system, b is either already in a's local forest or is "flying towards" it. With this assumption, our proofs do not rely on the more advanced features of the protocol, such as peer-exchange (via AddrMsg and ConnectMsg) and on-demand data exchange (via InvMsg and GetDataMsg). These features will become useful in the future, when we want to prove more interesting invariants. To illustrate this, let us consider a case of a node a' that has joined late, announcing itself (via ConnectMsg) only to a few other participants. Then, a' might not have yet requested or has not yet been forwarded all the blocks already minted in the system. Therefore, in order to relate its local state to $\widehat{bf}$, we would have to enhance the invariant with a conjunct for ongoing "propagation" of the known peers in the system, and replace (i) by it. In addition to that, we would need to consider situations when the topology is not a connected graph, in which case several "canonical" chains would co-exist without ever being reconciled. Stating the consensus property in such settings is our future work.

6 Elements of our Mechanisation

We mechanised all results described in this paper in Coq, making use of the Ssreflect/MathComp libraries [23, 31]. Our implementation of block forests builds on the library of partial finite maps by Nanevski *et al.* [25]. The size of our contributed codebase is pleasantly small, as demonstrated by the lines of code figures in Table 1.

In the proofs, we heavily relied on the small-scale reflection and rewriting machinery provided by Ssreflect [12]. For instance, our implementation of block forests features both constructive and computable definitions of prefix/fork relations on chains, as well as the corresponding `reflect`-view lemmas, to switch between the two representations. The definitions of all operations and predicates on block forests, such as (1)–(3), and on network configurations, such as (9) and (10), are also made decidable/computable. This design choice has paid off not only in reducing proof sizes, but also in the robustness of our proof scripts in the face of changes

Table 1. Sizes of definitions and proofs (LOC).

	Definitions	Proofs
Block Forests	579	1406
Protocol and Network	409	263
Consensus Properties	241	273

made in the definitions, which is surprising given how modest the amount of automation we used in the project was.

As an anecdote from our experience, while preparing this submission, a few days before the CPP'18 deadline and already after having completed proofs of all invariants, we have noticed an odd encoding of the rule RcvConnect. In our mechanisation to date, the corresponding transition was only adding the sender from to the local list of peers, but, rather selfishly, did not send a list of available hashes back as an InvMsg. We have changed the implementation so it would precisely match the rule from Figure 6, and, to our surprise, *no proofs of invariants broke*. We consider this an encouraging sign to investigate domain-specific automation for proofs about replicated state-transition systems.

7 Discussion

We now discuss the limitations of our protocol model and the implications of the assumptions we made.

Network semantics and system faults. As defined in Section 4, our network semantics is quite restricted. For instance, it does not include notions of packets being dropped or of participant faults. In practice, the clique assumption means that we can largely ignore crash faults, as we do not need other participants to relay our messages and we do not expect to receive any responses.

A possible complication arises when a participant crashes while in the process of broadcasting a newly-minted block, such that some peers receive it and others do not. This scenario, which is very similar to that of dropped packets, is difficult to accommodate in the current invariant. However, the problem with dropped packets would essentially disappear once we start making use of the protocol's peer-to-peer facilities, as they a provide a large amount of communication redundancy. That is, participants in the network advertise their *entire* knowledge every time they update it, and they request information they do not have *from all* peers that have advertised it. For reaching the consensus eventually, it is sufficient that one of these messages gets delivered. If none of them is, the process repeats the next time a peer updates its state and advertises.

Byzantine behaviours. A special case of faults is that of Byzantine faults, in which participants exhibit arbitrary behaviour [21]. These may arise due to software bugs, hardware malfunctions or through the actions of malicious actors. Our invariant is not resistant to Byzantine faults. For example, the proof relies on the fact that all blocks in the canonical block forest are *tx-valid*. This is true under normal operation, but can be invalidated at will by a malicious actor. In order to reason about the ineffectiveness of Byzantine faults, we will have to introduce to the invariant some notion of *honest participants* being in the majority and in communication with each other, and to find a way of accommodating within the proofs the presence of "bad" blocks.

It is important to stress that these invalid blocks, as seen in cases (d) and (e) of Figure 4, do not in any way prevent the protocol from operating correctly (from the perspective of faithful participants), but merely make it more difficult to *prove* that it does.

Other protocol properties. In this work, we have focused exclusively on the safety of the system, *i.e.*, the property that all correct nodes agree *wrt.* which ledger they adopt. Other properties, such as liveness and various security properties, depend on the choice of system parameters *hash*, *VAF*, and *FCR*. For example, system security almost certainly requires that we use a *cryptographic* hash function, *i.e.*, a hash function that is both collision-resistant (approx. injective) and pre-image resistant (given $h(m)$, finding m is computationally hard). For liveness, we will at the very least need to ensure, in the form of a new axiom, that hashes for blocks do not collide with hashes for transactions. Otherwise, information might not propagate correctly throughout the network. Moreover, we likely want *VAF* to impose a reasonable delay between consecutive block mintings, such that messages have time to propagate throughout the network—this would provide an adequate quiescent state, and thus consensus can be reached. Studying the full implications of different framework parameters is left for future work.

Towards a verified blockchain implementation. Our implementation of the protocol is intentionally non-optimal. Whenever faced with a decision of how to implement a function, we always chose simplicity over efficiency. That being said, all of our functions of, *e.g.*, processing block forests, are *pure*, so that they can be replaced with more efficient functional or imperative equivalents. For example, the reference implementation of *chain* performs a lot of redundant computation and would greatly benefit from a memoization strategy. Similarly, the message propagation strategy is very inefficient, and could be replaced with a more sensible one.

Because our mechanisation of the protocol is encoded as a library of computable Coq functions that implement state transformers and message handlers, it should be possible to extract it to OCaml and run on top of a trusted shim implementation, thus providing a formally verified blockchain *implementation*, in the same way it has been done in the recent work on the Disel framework [32, 35]. This setup, however, appears quite naïve and would be problematic in a realistic case of Byzantine faults, as the safety results

established in this work hold only as long as all participants follow the protocol and use exactly the same version of the shim, which is hard to guarantee in an adversarial distributed environment.

8 Related Work

The results we presented in this paper are related to the latest advances in the areas of computer security, formal methods, and distributed systems.

Consistency of Blockchain Protocols. In the past few years, there has been a lot of interest within the security and privacy community for notions of consistency in application to blockchain protocols.

Garay *et al.* considered the core protocol underlying Bitcoin [24], focusing on its two properties, dubbed *Common Prefix* and *Chain Quality* [11]. The common prefix property is a probabilistic version of the notion of eventual consistency we have established in Section 5 of this paper. Specifically, they establish that all *honest* parties in the system agree on a common ledger prefix up to k last blocks, where k is a parameter of the system. The chain quality property tackles a Byzantine setting in which malicious participants may contribute ill-formed blocks, and states that the number of such blocks in the system in not very large, given that the majority of participants remain honest and follow the protocol. Unlike our formalization, the work by Garay *et al.* takes into the account possible adversarial behaviours of the protocol participants, but restricts the communication to fully synchronous, *i.e.*, messages in the system are instantly delivered without delays, whereas we allow for arbitrary delays and permutations in message delivery.

While that work focuses on proving the properties of Nakamoto's consensus based on Proof-of-Work [24], in a follow-up to that result, Kiayias *et al.* propose a blockchain consensus protocol based on Proof-of-Stake [3] and possessing the same properties, and also a new one, *Chain Growth*, which ensures overall liveness for the honest parties [20], under the assumption of synchronous message delivery in the network. Finally, in a recent work, Pass *et al.* provided probabilistic boundaries with respect to chain growth and quality, as well as the analysis of other consistency and liveness properties of blockchain consensus in a fully asynchronous environment [27].

In contrast with those and many other works [7, 11, 20, 27] that analyse blockchain consensus from the perspective of security properties, thus, focusing on probabilistic reasoning about a protocol modeled as a composition of distributions, we present a simple operational model that immediately provides an executable semantics of the system, but only allows us to prove "coarse-grained" correctness conditions, such as eventual consistency.

None of the proofs of security properties of blockchain consensus we are aware of were mechanised.

Formal Methods for Blockchain Applications. To date, the interest of the formal methods community *wrt.* blockchain-based systems is predominantly in applications of the technology, rather than reasoning about properties and invariants of the underlying protocols.

In the past two years, a number of works have been published on formal modeling and verification of *smart contracts*—a mechanism to associate executable code with certain blockchain transactions, providing a machinery for trusted decentralised arbitration, which gained a lot of attention thanks to its highly influential implementation in Ethereum [37]. Various formal semantics of Ethereum Virtual Machine (EVM) and its contract language Solidity were implemented in Coq [18], Isabelle/HOL [1, 17], F* [4], K [15], Idris [28], Why3 [10], and in custom tools for static and dynamic analysis of smart contract behaviours [22].

An implementation of an efficient data structure for transaction ledgers has been developed and verified in Coq by White [34], yet it has not been used in the context of verifying a protocol that employs this structure.

At the level of reasoning about protocols, Hirai has formalised a simple variant of a Proof-of-Stake protocol in Isabelle [16], proving a version of the protocols' *accountable safety*: if two conflicting blocks get adopted in a shared block tree, then at least $\frac{1}{3}$ of participants may lose their entire deposits (stakes). This property is specific to Ethereum's Casper protocol [6] and is orthogonal to the consensus result we established in this work.

Verification of Distributed Consensus. Several recent major efforts have fully mechanised and verified implementations of more traditional consensus protocols, such as versions of Paxos [8, 14, 19, 26, 30], Raft [36, 38], or the classical Two-Phase Commit [32, 35]. Even though none of those works consider blockchain consensus, we believe, many of those frameworks can handle it, as long as they can adopt our model and support reasoning about block forests. Therefore, we see our main conceptual contribution in distilling the protocol semantics and outlining the proof layout for blockchain consensus.

9 Conclusion and Future Work

In this work, we have presented a formal operational model of a distributed blockchain-based consensus protocol, implemented its core data structures, characterised the primitives it relies upon, and mechanically proved a form of the protocol's eventual consistency, *i.e.*, that a system that implements it does indeed reach a consensus.

In the future, we are going to enhance our mechanisation for reasoning about relevant security properties [20], modeling schedule-providing oracles as probabilistic distributions. We also plan to define an operational semantics for transactions run on top of the protocol, using it as a foundational platform for verified smart contracts.

Acknowledgments

We thank the CPP'18 reviewers for the careful reading and constructive suggestions on the paper and the formalisation. We also thank June Andronick and Amy Felty for their efforts as CPP'18 Program Co-Chairs.

Sergey's research was supported by EPSRC First Grant EP/P009271/1 "Program Logics for Compositional Specification and Verification of Distributed Systems".

References

[1] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *CPP*. ACM. To appear.

[2] James Aspnes, Maurice Herlihy, and Nir Shavit. 1994. Counting Networks. *J. ACM* 41, 5 (1994), 1020–1048.

[3] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. 2014. Cryptocurrencies without Proof of Work. *CoRR* abs/1406.5694 (2014).

[4] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *PLAS*. ACM, 91–96.

[5] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated data types: specification, verification, optimality. In *POPL*. ACM, 271–284.

[6] Vitalik Buterin and Virgil Griffith. 2017. Casper the Friendly Finality Gadget. *CoRR* abs/1710.09437 (2017).

[7] Phil Daian, Rafael Pass, and Elaine Shi. 2017. *Snow White: Robustly reconfigurable consensus and applications to provably secure proofs of stake.* Technical Report. Cryptology ePrint Archive, Report 2016/919.

[8] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*. ACM, 400–415.

[9] Cynthia Dwork and Moni Naor. 1992. Pricing via Processing or Combatting Junk Mail. In *CRYPTO (LNCS)*, Vol. 740. Springer, 139–147.

[10] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *ESOP (LNCS)*, Vol. 7792. Springer, 125–128.

[11] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. 2015. The Bitcoin Backbone Protocol: Analysis and Applications. In *EUROCRYPT (Part 2) (LNCS)*, Vol. 9057. Springer, 281–310.

[12] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. 2009. *A Small Scale Reflection Extension for the Coq system.* Technical Report 6455. Microsoft Research – Inria Joint Centre.

[13] L.M. Goodman. 2014. Tezos: A Self-Amending Crypto-Ledger. Position Paper. https://www.tezos.com/static/papers/position_paper.pdf.

[14] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *SOSP*. ACM, 1–17.

[15] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. 2017. *KEVM: A Complete Semantics of the Ethereum Virtual Machine.* Technical Report.

[16] Yoichi Hirai. 2017. A mechanized safety proof for PoS with dynamic validators. Published online on 18 March 2017.

[17] Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *1st Workshop on Trusted Smart Contracts*.

[18] Yoichi Hirai. 2017. Ethereum Virtual Machine for Coq (v0.0.2). Published online on 5 March 2017.

[19] Mauro Jaskelioff and Stephan Merz. 2005. Proving the Correctness of Disk Paxos. *Archive of Formal Proofs* 2005 (2005).

[20] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In *CRYPTO (Part 1) (LNCS)*, Vol. 10401. Springer, 357–388.

[21] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982), 382–401.

[22] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *CCS*. ACM, 254–269.

[23] Assia Mahboubi and Enrico Tassi. 2017. *Mathematical Components.* Available at https://math-comp.github.io/mcb.

[24] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf.

[25] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. 2010. Structuring the verification of heap-manipulating programs. In *POPL*. ACM, 261–274.

[26] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos made EPR: decidable reasoning about distributed protocols. *PACMPL* 1, OOPSLA (2017), 108:1–108:31.

[27] Rafael Pass, Lior Seeman, and Abhi Shelat. 2017. Analysis of the Blockchain Protocol in Asynchronous Networks. In *EUROCRYPT (Part 2) (LNCS)*, Vol. 10211. Springer, 643–673.

[28] Jack Pettersson and Robert Edström. 2016. *Safer Smart Contracts through Type-Driven Development.* Master's thesis. Chalmers University of Technology, Sweden.

[29] George Pîrlea and Ilya Sergey. 2017. Toychain: a Coq implementation of a minimalistic blockchain-based consensus protocol. A version accepted at CPP'18 is available at https://github.com/certichain/toychain/tree/cpp2018.

[30] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. 2015. Formal Specification, Verification, and Implementation of Fault-Tolerant Systems using EventML. In *AVOCS*. EASST.

[31] Ilya Sergey. 2014. *Programs and Proofs: Mechanizing Mathematics with Dependent Types.* Lecture notes with exercises. Available at http://ilyasergey.net/pnp.

[32] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and Proving with Distributed Protocols. *PACMPL* 2, POPL (2018), 28:1–28:30.

[33] Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP*. ACM, 172–183.

[34] Bill White. 2015. A Theory for Lightweight Cryptocurrency Ledgers. Unpublished draft. Code available at https://github.com/input-output-hk/qeditas-ledgertheory, accessed on 10 October 2017.

[35] James R. Wilcox, Ilya Sergey, and Zachary Tatlock. 2017. Programming Language Abstractions for Modularly Verified Distributed Systems. In *SNAPL (LIPIcs)*, Vol. 71. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 19:1–19:12.

[36] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*. ACM, 357–368.

[37] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. http://gavwood.com/paper.pdf.

[38] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. 2016. Planning for change in a formal verification of the Raft Consensus Protocol. In *CPP*. ACM, 154–165.