Towards Formal Verification of HotStuff-based Byzantine Fault Tolerant Consensus in Agda⋆

Harold Carr[1], Christopher Jenkins[2], Mark Moir[1], Victor Cacciari Miraldo[3], and Lisandra Silva[4]

[1] Oracle Labs, USA and New Zealand
[2] University of Iowa, USA
[3] Tweag, The Netherlands
[4] Runtime Verification, USA

Abstract. LibraBFT is a Byzantine Fault Tolerant (BFT) consensus protocol based on HotStuff. We present an abstract model of the protocol underlying HotStuff / LibraBFT, and formal, machine-checked proofs of their core correctness (safety) property and an extended condition that enables non-participating parties to verify committed results. (Liveness properties would be proved for specific implementations, *not* for the abstract model presented in this paper.)
A key contribution is precisely defining assumptions about the behavior of honest peers, in an abstract way, *independent* of any particular implementation. Therefore, our work is an important step towards proving correctness of an entire class of concrete implementations, without repeating the hard work of proving correctness of the underlying protocol. The abstract proofs are for a single configuration (epoch); extending these proofs across configuration changes is future work. Our models and proofs are expressed in Agda, and are available in open source.

1 Introduction

There has been phenomenal interest in decentralized systems that enable coordination among peers that do not necessarily trust each other. This interest has largely been driven in recent years by the emergence of blockchain technology. When the set of participants is limited by *permissioning* or *proof of stake* [11, 23], Byzantine Fault Tolerant (BFT) [27] consensus—which tolerates some *byzantine* peers actively deviating from the protocol—is of interest.

Due to attractive properties relative to previous BFT consensus protocols, implementations based on HotStuff [41] are being developed and adopted. For example, the Diem Foundation (formerly Libra Association) was until recently developing LibraBFT based on HotStuff [5, 37]. (LibraBFT was renamed to DiemBFT before being discontinued; other variants are emerging.)

Many published consensus algorithms, including some with manual correctness proofs, have been shown to be incorrect [12, 38]. Therefore, precise, machine-checked formal verification is essential, particularly for new algorithms being

⋆ The final publication is available at https://link.springer.com/book/9783031067747.

adopted in practice. Some of the papers on HotStuff / LibraBFT include brief correctness arguments, but they lack many details and are not machine checked. Furthermore, LibraBFT uses data structures, messages and logic, that differ significantly from versions on which those informal proofs were based.

Our contributions are as follows:

- a precise, abstract model of the protocol underlying HotStuff / LibraBFT;
- precise formulation of assumptions; and
- formal, machine-checked proofs of core correctness (safety) properties, plus a novel extended condition that enables additional functionality.

Proving correctness for an *abstraction* of the protocol enables verifying any concrete implementation by proving that its handlers ensure the assumptions of our abstract proofs. Our contribution is thus an important step towards proving correctness for an entire class of concrete implementations. However, this class does not include all possible variants. In particular, DiemBFT recently added an option for committing based on 2-chains, rather than 3-chains, as our work assumes (see Section 3.1). Adapting our techniques to accommodate 2-chain-based implementations is future work.

This paper focuses on the metatheory around an *abstraction* of a system of peers participating in the HotStuff / LibraBFT protocol, and assumptions about which peers can participate, rules that honest peers obey, and the intersection of any two quorums containing at least one honest peer. We state and prove key correctness properties, such as that any two committed blocks do not conflict (i.e., they belong to the same ordered chain of committed blocks).

Our ongoing work [7] aims to use the results presented in this paper to verify a concrete Haskell implementation that we have developed based on the Diem Foundation's open-source Rust implementation [18]. We have built a system model that can be instantiated with data types and handlers, yielding a model of a distributed system in which honest peers execute those handlers and byzantine ones are constrained only by being unable to forge signatures of honest peers. We have ported this implementation to Agda, using a library we have developed [13] to enable the ported code to closely mirror the original, thus reducing the risk of error. We have made substantial progress towards proving that the resulting Agda port satisfies the assumptions established in this paper.

LibraBFT supports configuration changes (also known as epoch changes), whereby parameters such as the number and identities of participating peers can be changed. The contribution described in this paper is an abstract model for a single epoch and formal, machine-checked proofs of its correctness conditions. Stating and proving cross-epoch properties is future work. Nevertheless, the Haskell implementation we are verifying supports epoch changes, and our verification infrastructure is prepared for multiple epochs. In particular, our abstract modules are parameterized by an "epoch configuration" structure.

Our models, definitions and proofs are expressed in Agda [1, 32], a dependently-typed programming language and proof assistant. We chose Agda for this work because its syntax is similar to Haskell's, making it easier to develop and have confidence in a model of the implementation we aim to verify. This paper is intended to be reasonably self contained and does not require the reader to know Agda. To that end, we will explain Agda-specific features and syntax that are important for following the paper. We encourage interested readers to explore the open source proofs in detail, and we hope that this paper will provide a useful overview and guide that will make them more accessible. For readers who would like to learn about Agda, we recommend starting with the tutorial in [39].

In Section 2, we overview salient aspects of HotStuff / LibraBFT to motivate our approach to abstractly modeling the protocol and formally verifying correctness properties. In Section 3, we present the definitions used to develop the formal abstract model of a system of peers participating in the protocol, and to define traditional and extended correctness properties. We also describe their proofs, which are available in open source [7]. Related work is summarized in Section 4 and concluding remarks and future work appear in Section 5. Additional proof overviews are included in the extended version of this paper [14].

2 An Overview of HotStuff / LibraBFT

The following overview does *not* fully describe HotStuff and LibraBFT: it highlights aspects that our abstraction must accommodate to enable our proofs. Details are in the relevant papers and repositories [5, 7, 18, 37, 41].

Peers participating in the HotStuff / LibraBFT protocol repeatedly agree to extend a chain of *blocks* that is initially empty (represented by a *genesis* record). Each block identifies (directly or indirectly) the block that it extends (or the genesis record if none) via one or more cryptographic hashes. This common *hash chaining* [36] technique ensures that each block uniquely identifies its predecessor, unless an adversary finds a hash collision (e.g., two different blocks that hash to the same value); it is a standard assumption that a computationally bounded adversary cannot do so [30, Chapter 9].

We require that two (*honest*) peers that faithfully follow the protocol cannot be convinced to extend the chain in conflicting ways: if honest peer p_1 (resp., p_2) determines that block b_1 (resp. b_2) is in the chain, then the chain up to one of the blocks *extends* the chain up to the other. This must hold even if some (*byzantine*) peers (up to some threshold, as discussed below) actively misbehave.

A peer can *propose* to add a new block to a chain, and others can *vote* to support the proposal. A proposed block can include a special *reconfiguration* (*epoch change*) transaction, which would change the set of peers participating and/or other parameters. To prevent impersonation, messages are signed.

A valid proposal contains or identifies a *quorum certificate* that represents a *quorum* of votes supporting the previous block. Based on assumptions discussed below, we can be sure that any two quorums each contain a vote from at least one honest peer in common. An honest participant will refuse to vote for a proposal if the requirements for the quorum certificate and previous blocks are not met. This ensures that the quorum certificate associated with each block in a chain satisfies these requirements, even though some peers that contributed votes to the quorum certificates may be dishonest. The conditions for *committing* a block are designed to ensure that honest peers never contribute votes to two quorums that cause conflicting blocks to be committed.

If a byzantine proposer sends different proposals to different peers, a quorum of votes for the same proposal may not be generated. In this case, waiting peers may time out, and initiate a new effort to extend the chain; this can result in competing proposals to extend the same chain with different blocks. To distinguish between proposals, each proposed block has an associated *round*, which must be larger than that of the block that it extends. Because competing proposals are possible, peers collectively build a *tree* of records, and follow specified rules to determine when a given proposal has been committed. The essence of the protocol is in the rules that honest peers must follow, and what information a peer must verify before committing a proposal.

The goal of this work is an *abstract* model of the protocol that is independent of all these details, capturing just enough detail to prove that, if the assumptions are not violated, then honest peers will not commit conflicting proposals.

3 Correctness properties and proofs

We prove our high-level abstract correctness properties in module *LibraBFT.Abstract.Properties* (in file `LibraBFT/Abstract/Properties.agda`), which receives several module parameters that can be instantiated in order to relate a particular implementation to the abstract machinery.

```
module LibraBFT.Abstract.Properties
  (ℰ : EpochConfig) (UID : Set)
  (_≟UID_ : (u₀ u₁ : UID) → Dec (u₀ ≡ u₁))
  (𝒱 : VoteEvidence ℰ UID)
  where ...
```

We first describe *EpochConfig*; the other module parameters are explained later. *EpochConfig* represents configuration information for an epoch, including: how many peers participate in the epoch (*authorsN*), their identities (*toNodeId*), and their public keys (*getPubKey*), as well as requirements such as each member having a different public key (*PK-inj*). Members are identified by values of type *Fin authorsN*: the natural numbers less than *authorsN*; for example, we have *getPubKey* : *Member* → *PK* where *Member* = *Fin authorsN*.

An *EpochConfig* also provides *IsQuorum*, a predicate indicating what the implementation considers to be a quorum. The type of *IsQuorum* is *List Member* → *Set*; *Set* is Agda's way of representing an arbitrary type. This definition is then used to define another important field of an *EpochConfig*:

```
bft-assumption : ∀ {xs ys} → IsQuorum xs → IsQuorum ys
                 → ∃[a](a ∈ xs × a ∈ ys × MetaHonestPK (getPubKey a))
```

Here, *bft-assumption* requires that the intersection of any two quorums contains at least one honest peer.[5] Agda supports *implicit* arguments, listed in curly braces, which need not be provided explicitly if their values can be inferred from context, e.g., *IsQuorum xs* implies that *xs* is of type *List Member*. The $\exists[a]\cdot$ notation says that there is an a which satisfies the condition—a product of three conditions, in this case. The type of a must be implied by context; here, $a \in xs$ implies that a is of type *Member*.

To inherit the correctness properties we prove, an implementation must provide an *EpochConfig* as a module parameter. Part of constructing it is proving *bft-assumption* based on whatever assumptions and definition of *IsQuorum* the implementation uses. One common approach is to assume n peers with equal "voting power", at most f of which are byzantine, and to ensure that $n > 3f$; in this case, a set of peers is a quorum iff it contains at least $2n/3$ distinct peers. *LibraBFT.Abstract.BFT* contains a lemma that can be used to prove that such assumptions imply *bft-assumption*. The lemma is sufficiently general to accommodate LibraBFT's approach of assigning (potentially non-uniform) *voting power* to peers, and considering a set of peers to be a quorum iff its combined voting power exceeds two thirds of the total voting power.

The remainder of this section is in context of a single *EpochConfig* called $\mathcal{E}$.

3.1 Abstract *Record*s and *RecordChain*s

A *Record* can be a *Block*, a quorum certificate (*QC*) or the epoch's *genesis* (initial) *Record*; precise definitions are below. (These are *abstract* records that may not correlate closely to data structures and message formats used by an implementation; for example, in LibraBFT, blocks *contain* the previous QC.) HotStuff-based algorithms grow a tree of *Record*s rooted at the epoch's genesis record, where nodes contain a *Block* or a *QC*. Paths (called *RecordChain*s) from the root begin with the genesis record and then alternate between *Block*s and *QC*s. For example, the existence of a path from the root to a record r is captured by the type *RecordChain r* being inhabited. Figure 1 illustrates a tree of *Record*s.

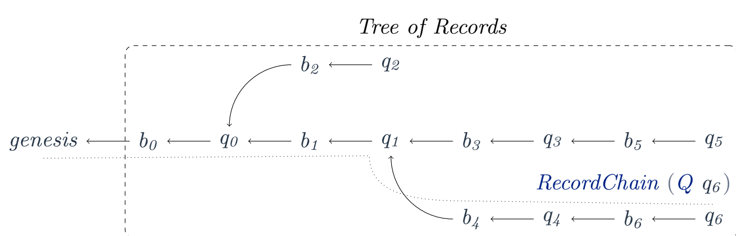

Fig. 1. A tree of *Record*s with a *RecordChain* from *genesis* to abstract *Record Q* q_6.

[5] *MetaHonestPK* is a predicate representing whether a peer owning a key behaves honestly. The *Meta* prefix identifies this as being part of the formal model and not accessible to implementations, which must not depend on knowing who is honest.

While typical implementations carry more information, abstractly, a *Block* comprises its round number, an identifier of type *UID* for itself and for the quorum certificate it extends, if any (a value of type *Maybe UID* is either *nothing* or *just* x for some x of type *UID*). *UID* can be any type that has decidable equality, as represented by the second and third module parameters; these are passed to other modules in the *Abstract* namespace as needed. Definitions below are in modules *LibraBFT.Abstract.Records* and *LibraBFT.Abstract.RecordChain*.

Typical implementations obtain a *Block*'s id by applying a cryptographic hash function to some or all of its contents; thus identifiers may not be unique. Our correctness properties are therefore proved modulo "injectivity failures" on (supposedly) unique ids. We do *not* assume that such injectivity failures do not exist, which would make our proofs meaningless because they *can* occur in practice, however unlikely. We elaborate below and in Sections 4 and 5.

Abstractly, a *Vote* is by a member of the epoch, for a round and *Block* id.

```
record Block : Set where
  constructor mkBlock
  field bRound   : Round
        bId      : UID
        bPrevQC  : Maybe UID
```

```
record Vote : Set where
  constructor mkVote
  field vRound    : Round
        vMember   : Member
        vBlockUID : UID
```

A quorum certificate (*QC*) represents enough *Vote*s to *certify* that a *Block* has been accepted by a quorum of members. It includes the *Block*'s id and round, and a list of *Vote*s and evidence that the QC is "valid" (representing properties that honest peers verify before accepting the QC), i.e.,:

1. The list of voting *Member*s represents a quorum.
2. All *Vote*s are for the *Block*'s id.
3. All *Vote*s are for the same round.

Honest peers accept a (concrete) *Vote* only if it satisfies implementation-specific conditions captured by the module parameter $\mathcal{V}$ of type *VoteEvidence* $\mathcal{E}$ *UID*, an implementation-specific predicate on abstract *Vote*s. To enable proofs to access the verified conditions, we add a fourth coherence clause to QCs:

4. For each *Vote* in the *QC*, there is evidence that a message was sent containing a concrete representation of the (abstract) *Vote* that satisfies the implementation-specific conditions.

Putting this all together, we have:

```
record QC : Set where
  constructor mkQC
  field qRound       : Round
        qCertBlockId : UID
        qVotes       : List Vote
        qVotes-C1    : IsQuorum (List-map vMember qVotes)
        qVotes-C2    : All (λ v → vBlockUID v ≡ qCertBlockId) qVotes
        qVotes-C3    : All (λ v → vRound v    ≡ qRound)       qVotes
        qVotes-C4    : All 𝒱 qVotes
```

All (from the Agda standard library) accepts a predicate and a list, and requires that each element of the list satisfies the predicate.

Next, we define a *Record* to be either a *Block*, a *QC*, or the special genesis record *I*. There is a constructor for each case, and the *B* and *Q* constructors take arguments of the appropriate type to form a *Record*.

```
data Record : Set where
  I :           Record
  B : Block  →  Record
  Q : QC     →  Record
```

We then say that a record r' *extends* another record r, denoted $r \leftarrow r'$, whenever one of the following conditions is met:

1. r is the genesis *Record* and r' is a *Block* for round greater than 0 and not identifying any previous *Block*.
2. r is a *QC* and r' is a *Block* with a round higher than r's and with a *bPrevQC* field identifying r.
3. r is a *Block* and r' is a *QC* certifying r.

We capture these conditions in the following Agda datatype; _←_ indicates that ← is an infix operator with two arguments. Values of this type can be constructed using one of three constructors (*I←B*, *Q←B* or *B←Q*), each of which requires several arguments to establish a value of _←_ for a pair of *Record*s.

```
data _←_ : Record → Record → Set where
  I←B  : ∀ { b } → 0 < getRound b → bPrevQC b ≡ nothing
         → I ← (B b)
  Q←B  : ∀ { q b } → getRound q < getRound b
         → just (qCertBlockId q) ≡ bPrevQC b
         → Q q ← B b
  B←Q  : ∀ { b q } → getRound q ≡ getRound b → bId b ≡ qCertBlockId q
         → B b ← Q q
```

*RecordChain*s are in the reflexive, transitive closure of _←_, starting at the genesis record *I*. Sometimes, we reason about paths starting at records other than *I*; we therefore define *RecordChain* using the more specific *RecordChainFrom*.

```
data RecordChainFrom (o : Record) : Record → Set where
  empty : RecordChainFrom o o
  step  : ∀ { r r′ } → RecordChainFrom o r
          → r ← r′
          → RecordChainFrom o r′

RecordChain : Record → Set
RecordChain = RecordChainFrom I
```

Next, we present definitions needed to specify when a *Block* can be committed. For $k > 0$, a $\mathbb{K}$-*chain* is a sequence of k *Block*s, each of which is extended by a *QC*, such that each *Block* (except the first) extends the *QC* that extends the previous *Block*. Furthermore, each adjacent pair of *Block*s must satisfy the relation *R*, which can be instantiated with *Simple* (which holds for any pair of *Block*s) or *Contig* (which holds only if the rounds of the two *Block*s are contiguous: the second *Block*'s round is one greater than that of the first; the first parameter to *R* enables a definition of *Contig* that does not require a predecessor for the first *Block*; see module *LibraBFT.Abstract.RecordChain*). 𝕂*-chain*s are defined as follows.

```
data 𝕂-chain (R : ℕ → Record → Record → Set)
  : (k : ℕ) {o r : Record} → RecordChainFrom o r → Set where
  0-chain : ∀ {o r} {rc : RecordChainFrom o r} → 𝕂-chain R 0 rc
  s-chain : ∀ {k o r} {rc : RecordChainFrom o r} {b : Block} {q : QC}
          → (r←b : r ← B b) → (prf : R k r (B b))
          → (b←q : B b ← Q q) → 𝕂-chain R k rc
          → 𝕂-chain R (suc k) (step (step rc r←b) b←q)
```

Block b_0 (and those preceding it) are committed if b_0 is the head of a contiguous 3-chain: there is a *RecordChain* that contains b followed by blocks b_1 and b_2, such that the rounds of blocks b_0, b_1 and b_2 are consecutive. This is called a *CommitRule* (*kchainBlock* n c_3 is the nth *Block* from the end of c_3):

```
data CommitRuleFrom {o r : Record}
                          (rc : RecordChainFrom o r) (b : Block) : Set where
  commit-rule : (c₃ : 𝕂-chain Contig 3 rc) → b ≡ kchainBlock 2 c₃
              → CommitRuleFrom rc b
```

3.2 First correctness property: *thmS5*

We can now explain the first high-level property we prove for our abstract model, *thmS5*. (Because our work has been influenced by versions of the HotStuff [41] and LibraBFT papers [5, 37], some of our properties are named after properties presented informally in those papers. For example, *thmS5* is named after Theorem S5 in [5].) It states that, if two blocks b and b' are committed via *CommitRule* rc b and *CommitRule* rc' b', respectively, then one of the blocks is contained in the record chain of the other. This property ensures that all committed *Block*s are on a single non-branching path in the tree of *Record*s.

```
thmS5 : ∀ {q q'} → {rc : RecordChain (Q q)} → AllInSys rc
      → {rc' : RecordChain (Q q')} → AllInSys rc'
      → {b b' : Block} → CommitRule rc b → CommitRule rc' b'
      → Either NonInjective-≡ (Either ((B b) ∈RC rc') ((B b') ∈RC rc))
```

AllInSys rc means that each record in rc is "in" the abstract system according to an implementation-specific predicate over abstract *Record*s called *InSys*, which is provided as a module parameter. For purposes of *AllInSys*, a record r being "in" a record chain rc is captured by a simple recursive definition: if rc is formed by extending record chain rc' by record r', then r is "in" rc iff $r = r'$

or r is "in" rc'. On the other hand, as explained in Section 3.4, $\in RC$ represents a more complicated notion of a record being "in" a record chain.

Note that *thmS5* requires that *either* *NonInjective-≡* holds *or* one of the committed *Block*s is in a *RecordChain* ending at the other. The *NonInjective-≡* disjunct—which is shared by many of the properties discussed below—reflects that we prove *thmS5* modulo injectivity of *Block* ids, as discussed above.

In Section 3.6, we explain how we refine the definition of *thmS5* and other properties in order to relate our abstract proofs to the security properties of a concrete implementation that is proved correct using them. For now, however, we can think of the following simplified definition of *NonInjective-≡*:

```
NonInjective-≡ : Set
NonInjective-≡ = Σ (Block × Block)
                   (λ { (b0 , b1) → b0 ≢ b1 × bId b0 ≡ bId b1 })
```

The Σ notation is similar to the $\exists[\cdot]\cdot$ notation introduced earlier, except that it specifies the *type* of the existentially quantified value (not just a name, as with $\exists[\cdot]\cdot$) and the condition on the value of that type is expressed as a predicate on that type. Thus, a value of type *NonInjective-≡* comprises a *pair* of (abstract) *Block*s—b_0 and b_1—that are *different* but have the *same* id.

3.3 Precisely defining protocol rules

Module *LibraBFT.Abstract.RecordChain.Properties* contains the proof of *thmS5*, which requires module parameters representing assumptions about *Record*s that are *InSys*. These assumptions capture the key properties that an implementation must ensure. Part of our contribution is to precisely define these assumptions in an abstract way, independent of any particular implementation.

Implementations described in various papers [5, 37, 41] are all based on the same core ideas, but differ substantially in detail. None of these papers gives a precise definition of the core protocol. Early versions of the LibraBFT papers [5] come closest, providing explicit statements of two "voting constraints".

These voting constraints ("Increasing Round" and "Preferred Round") were a starting point for us, but they are not entirely suitable for our purposes. For example, the "Increasing Round" constraint is originally stated as: *An honest node that voted once for* B *in the past may only vote for* B' *if* $round\ (B) < round\ (B')$. However, to interpret this as a protocol rule, we would need to define precisely what it means to have "voted in the past". Our proof efforts revealed that it suffices to require that an honest peer does not sign and send *different* (abstract) votes for the same round (regardless of order):

```
VotesOnlyOnceRule : Set ℓ
VotesOnlyOnceRule = (a : Member) → MetaHonestMember a
                    → ∀ {q q′} → InSys (Q q) → InSys (Q q′)
                    → (v : a ∈QC q) (v′ : a ∈QC q′)
                    → vRound (∈QC-Vote q v) ≡ vRound (∈QC-Vote q′ v′)
                    → ∈QC-Vote q v ≡ ∈QC-Vote q′ v′
```

For generality, *InSys* is assumed to return a type from some arbitrary universe [40] with level ℓ. The v parameter is evidence that there is a *Vote* by member a represented in q (a *QC*), and *∈QC-Vote* q v is that (abstract) *Vote*. Thus, *VotesOnlyOnceRule* requires that, if there are two *Vote*s for the same round by an honest member a in *QC*s in the system, then the *Vote*s are equal.

The second constraint—*PreferredRoundRule*—is more complicated. It is based on the voting constraint called “Locked Round” in early versions of the LibraBFT paper [5]; similar constraints on voting are followed by HotStuff [41] and by later versions of LibraBFT [37]. The essence of this rule is that, if an honest peer contributes a *Vote* to q (a *QC*) that commits a *Block* (c_3 is essentially a *CommitRule* that commits the *Block* identified by *kchainBlock 2* c_3), then it does not vote in a higher round for a *Block* unless the round of the *previous* *Block* is at least that of the committed *Block*. This is a key requirement to avoid voting to commit another *Block* that conflicts with the first.

```
PreferredRoundRule : Set ℓ
PreferredRoundRule
  = ∀ a {q q′} → MetaHonestMember a → InSys (Q q) → InSys (Q q′)
  → {rc : RecordChain (Q q)} {n : ℕ} → (c₃ : 𝕂-chain Contig (3 + n) rc)
  → (v : a ∈QC q) (rc′ : RecordChain (Q q′)) (v′ : a ∈QC q′)
  → vRound (∈QC-Vote q v) < vRound (∈QC-Vote q′ v′)
  → Either NonInjective-≡
           (getRound (kchainBlock (suc (suc zero)) c₃) ⩽ prevRound rc′)
```

3.4 The proof of *thmS5*

Our proof of *thmS5* is similar to the manual proof presented an early version of the LibraBFT paper [5]. However, a formal, machine-checked proof must address many details that are glossed over in the manual proof. Furthermore, as discussed in Section 3.3, making our assumptions about honest peers' *Vote*s precise and implementation-independent required somewhat different assumptions.

To help the reader approach the formal, machine-checked proofs in our open-source development [7], we describe below some of its key proofs and properties.

We first introduce two key lemmas. Roughly speaking, *lemmaS2* states that there can be at most one certified *Block* per round. Its proof depends on the *bft-assumption*: for two *QC*s, there is some honest peer with *Vote*s in each. By the assumption that honest peers obey *VotesOnlyOnceRule*, if the blocks certified by the two *QC*s have the same round, then both *Vote*s are for the same *BlockId*. However, this does *not* imply the *QC*s certify the same *Block*. For this reason, the conclusion of *lemmaS2* is that *either* *bId* is non-injective *or* $b_0 \equiv b_1$.

```
lemmaS2 : ∀ {b₀ b₁ : Block} {q₀ q₁ : QC} → InSys (Q q₀) → InSys (Q q₁)
        → (p₀ : B b₀ ← Q q₀) (p₁ : B b₁ ← Q q₁)
        → getRound b₀ ≡ getRound b₁
        → Either NonInjective-≡ (b₀ ≡ b₁)
```

Similarly, *lemmaS3* makes the *PreferredRoundRule* apply to *QC*s.

```
lemmaS3 : ∀ {r₂ q′} {rc : RecordChain r₂} → InSys r₂
        → (rc′ : RecordChain (Q q′)) → InSys (Q q′)
        → (c₃ : kchain Contig 3 rc) → round r₂ < getRound q′
        → Either NonInjective-≡ (getRound (kchainBlock (suc (suc zero)) c₃)
                                  ⩽ prevRound rc′)
```

The proof of *thmS5* depends on a non-symmetric variant of it called *propS4*:

```
propS4 : ∀ {q q′} {rc : RecordChain (Q q)} → AllInSys rc
       → (rc′ : RecordChain (Q q′)) → AllInSys rc′
       → (c₃ : 𝕂-chain Contig 3 rc)
       → getRound (kchainBlock (suc (suc zero)) c₃) ⩽ getRound q′
       → Either NonInjective-≡ (B (kchainBlock (suc (suc zero)) c₃) ∈RC rc′)
```

Recall that *∈RC* is a specific representation of what it means for a *Record* to be "in" a *RecordChain* that is precisely defined later, and note that c_3 is a 𝕂-*chain Contig 3 rc*, for some *rc*, i.e., a *CommitRule*.

Proof overviews for *thmS5* and *propS4* are in the extended paper [14].

Finally, we explain what it means for a *Block* to be "in" a *RecordChain*, as captured by the *∈RC* predicate. It is tempting to think that, if *RecordChain*s *rc* and *rc′* both end at block *b*, then the requirements of _←_ ensure that *rc* and *rc′* are the same *RecordChain*. However, suppose we have *q* ← *b* and *q′* ← *b*, where *q* and *q′* are *QC*s. The definition of _←_ requires that *just* (*qCertBlockId q*) ≡ *bprevQC b* ≡ *just* (*qCertBlockId q′*). This does *not* imply that *q* ≡ *q′* because *q* and *q′* may include different *Vote*s, reflecting the reality that two peers may be convinced to extend the same *Block* by two *different* valid *QC*s.

Therefore, we need a notion of *equivalent* *RecordChain*s that contain the same *Block*s and equivalent *QC*s: two *QCs* are equivalent iff they certify the same *Block* (i.e, their *qCertBlockId* components are equal). These notions are captured by *≈RC* (defined in *LibraBFT.Abstract.RecordChain*), which requires the two *RecordChain*s to be "pointwise equivalent" meaning that the corresponding *Record*s in the two *RecordChain*s are equivalent. A lemma *RC-irrelevant* shows that, if two record chains *rc* and *rc′* end at the same *Record*, then they are equivalent (i.e., *rc ≈RC rc′*), unless there is an injectivity failure.

The 𝕂-*chain-∈RC* property used in the proof of *propS4* states that, if a *RecordChain* rc_1 ends at a block *b* that is in a 𝕂-*chain* based on another record chain *rc*, then another *Block* that is earlier in the 𝕂-*chain* is also "in" rc_1. To enable proving this, *∈RC* must allow for the possibility that the other *Block* is contained in an equivalent *RecordChain*. The definition of *∈RC* therefore has an additional constructor beyond the two obvious ones, which enables the *Record* in question to be "transported" from an equivalent *RecordChain*:

```
data _∈RC_ {o : Record} (r0 : Record) :
            ∀ {r1} → RecordChainFrom o r1 → Set where
  here   : ∀ {rc : RecordChainFrom o r0} → r0 ∈RC rc
  there  : ∀ {r1 r2} {rc : RecordChainFrom o r1} → (p : r1 ← r2)
         → r0 ∈RC rc → r0 ∈RC (step rc p)
  transp : ∀ {r} {rc0 : RecordChainFrom o r} {rc1 : RecordChainFrom o r}
         → r0 ∈RC rc0 → rc0 ≈RC rc1 → r0 ∈RC rc1
```

3.5 Traditional and extended correctness properties

Our core correctness property *CommitsDoNotConflict* is *thmS5* without the *NonInjective-≡* disjunct. It is proved in *LibraBFT.Abstract.Properties*, which receives an additional module parameter *no-collisions-InSys* providing evidence that there are no injectivity failures between *Block*s that satisfy *InSys*. Note that, if an implementation reaches a state in which this does not hold, then there is an injectivity failure between *concrete Record*s at the implementation level; for a typical implementation, this signifies a collision for a cryptographic hash function among *Record*s that are actually in the system, contradicting the standard assumption that a computationally bounded adversary is unable to find such collisions. To prove *CommitsDoNotConflict*, we invoke *thmS5* and then use *no-collisions-InSys* to eliminate the possibility of an injectivity failure.

To invoke *CommitsDoNotConflict* for a particular implementation, we need to provide *AllInSys rc*, where *rc* is the *RecordChain* for the first *CommitRule* (and similarly for rc'). To enable this, honest voters in typical implementations will vote to extend a *Block* only after verifying that the *Block* extends a *QC* (or the initial *Record*) that the peer already knows is in the system. Thus, a peer that verifies a *CommitRule* based on a record chain *rc* that ends in a *QC* (q) knows that every *Record* in *rc* is "in the system": *AllInSys rc*.

Extended correctness condition We are also interested in enabling parties that do not participate in the protocol to verify commits. Suppose a peer p provides to a client c the contents of a *CommitRule* that c can verify. In this case, c cannot invoke *CommitsDoNotConflict* (or *thmS5*), because it does not know the *RecordChain* on which the *CommitRule* is based.

For this purpose, we define and prove a variant of *CommitsDoNotConflict* called *CommitsDoNotConflict′*. This condition ensures that even a party that does not participate in consensus can confirm commits and will not confirm conflicting commits.

```
CommitsDoNotConflict′ : ∀ {o o′ q q′}
  → {rcf : RecordChainFrom o (Q q)}  → AllInSys rcf
  → {rc′ : RecordChainFrom o′ (Q q′)} → AllInSys rc′
  → {b b′ : Block} → CommitRuleFrom rcf b → CommitRuleFrom rcf′ b′
  → Either Σ (RecordChain (Q q′)) ((B b) ∈RC_)
           Σ (RecordChain (Q q)) ((B b′) ∈RC_)
```

CommitsDoNotConflict′ does not require *CommitRule*s based on full *RecordChain*s; instead, *CommitRuleFrom*s based on *RecordChainFrom*s suffice. This property shows that a party can validate just the *Record*s required to form a *CommitRuleFrom*, and confirm that the *Block* it claims to commit has indeed been committed, and that there cannot be another commit that conflicts with it. Here, $(B\ b)\ \in RC\_$ is a predicate over values of type *RecordChain* $(Q\ q')$, so *CommitsDoNotConflict′* says that, if there are two *CommitRuleFrom*s based on *RecordChainFrom*s that end with a *QC* and have all of their *Record*s in the system, then (unless there is an injectivity failure), one of committed *Block*s is in a *RecordChain* that contains the other.

To prove this property, we require an additional assumption about the implementation, which is provided as a module parameter ∈QC⇒AllSent, of type *Complete InSys*, where:

```
Complete : ∀ {ℓ} → (Record → Set ℓ) → Set ℓ
Complete ∈sys  = ∀ {a q} → MetaHonestMember a
                → a ∈QC q → ∈sys (Q q)
                → ∃[b] (Σ (RecordChain (B b)) AllInSys × B b ← Q q)
```

Here, *Record* → *Set* ℓ is a predicate on (abstract) *Record*s representing what *Record*s an implementation considers to be "in the system".

This assumption (indirectly) requires that an honest peer sends a *Vote* for a *Block* id (which may subsequently be represented in a *QC*) only if it knows that there is a *Block* with that id and a *RecordChain* up to that *Block* whose *Record*s are all "in the system" (for example the peer may have validated all of those *Record*s itself, or it may have validated sufficient information to be confident that all of them have been validated by some honest peer, unless there is a hash collision among *Record*s that are in the system).

The extended version of this paper [14] includes proof overviews for *CommitsDoNotConflict′*, and for a lemma crf⇒cr on which it depends.

3.6 Relating non-injectivity to security properties

Recall from Section 3.2 that we prove our abstract properties modulo injectivity of *Block* ids. However, the simplified *NonInjective-≡* disjunct used in the property definitions presented so far is insufficient. The reason is that it is *trivial* to construct two different abstract *Block*s with the same id, meaning that we could prove *thmS5* with a single-line proof, independent of the actual protocol. Worse, we could accidentally do the same in context of legitimate-looking proofs.

The issue is that the abstract *Block*s we could trivially construct bear no relation to any real *Block*s and ids produced in the execution of a concrete implementation. To resolve this problem, we strengthen the first disjunct of *thmS5* to *NonInjective-≡-InSys*, defined as follows:

```
NonInjective-≡-InSys : Set
NonInjective-≡-InSys =
  Σ NonInjective-≡ λ {((b₀ , b₁) , _ , _) → InSys (B b₀) × InSys (B b₁)}
```

This definition requires that the proof not only provides different *Block*s b_0 and b_1 with the same id, but also proof that the implementation considers the *Record*s *B* b_0 and *B* b_1 to be "in the system". The meaning of "in the system" is specified by the implementation-provided predicate *InSys* and is thus beyond the scope of this paper. However, in ongoing work, we are proving a real implementation correct using the results presented here. In that broader context, we instantiate *InSys* with a predicate that holds only for *Block*s that are contained in network messages that have actually been sent. In this way, from the perspective of that concrete implementation, we ensure that our correctness properties hold unless and until an adversary *actually finds a hash collision* and introduces it into the system. We contrast this approach to some related efforts in Section 4.

The *NonInjective-≡* and *NonInjective-≡-InSys* definitions stated above are actually simplified versions of more general definitions we use in our proofs; details are available in our open source development [7]. These more general definitions are required because, at different stages of our proofs, we use different predicates to capture evidence collected so far about the conflicting *Block*s, so that we can build up to the proof for *thmS5* that both *Block*s satisfy *InSys*.

4 Related work

4.1 HotStuff/LibraBFT

Before open sourcing our work in December 2020 [7], we were not aware of any formal verification work related to the HotStuff / LibraBFT protocols beyond manual proof sketches [5, 37, 41]; these are useful and have influenced our work significantly, but are far from detailed, precise proofs. We have since learned of two other pieces of work involving mechanical proofs of correctness of variants of the HotStuff/LibraBFT algorithm, and one involving model checking.

Librachain [20] is a Coq-based model of the data structures used in LibraBFT. It contains a single commit from May 2020, described as "experimental"; we are not aware of any paper describing this work. The Librachain model commits to some structural details that are not central to the core protocol. For example, it assumes that the *QuorumCert* that a new *Block* extends is included in the *Block* record; this is one implementation choice, but certainly not fundamental. Furthermore, the proofs assume various conditions have been validated for the data structures, and are thus intimately tied to the particular implementation types. In contrast, we model an *abstraction* of the core protocol, and establish precise requirements for *any* implementation to enjoy the correctness properties we prove. The Librachain development also uses a hypothesis that the hash function used is injective, which is not true of hash functions that are used in practice. Our properties are proved to hold unless and until a *specific* injectivity failure exists between (abstract) *Record*s that are actually "in the system" (see Section 3.6); when instantiated with implementations that use cryptographic hash functions to assign ids, this ensures that the result holds unless and until a peer succeeds in finding a specific hash collision, violating the assumption that a computationally bounded adversary cannot do so.

More recently, Leander [22] has described work modeling and proving correctness for one specific, simplified variant of HotStuff. Hashes are not explicitly modeled, but the way the relationship between blocks is modeled amounts to an assumption that hashing is injective. Leander modeled this simplified variant in TLA+ and Ivy, and the paper is focused on comparing the tools for this purpose.

Kukharenko et al. [25] use TLA+ [26] to model check *basic* HotStuff, but not the more practical *chained* variant used by LibraBFT. Again, our work applies to an abstraction of the protocol that can be instantiated for all versions of HotStuff and LibraBFT, as well as variants that may not yet exist.

Model checking has the advantage of requiring less work (defining a model and correctness properties and then "pushing the button") than developing precise, machine-checked correctness proofs. It can also provide insight into errors found. Kukharenko et al. ran one of their models with seven participants of which three are byzantine (correctness is not guaranteed in this case), and found a counterexample showing *how* the byzantine peers can violate correctness.

To limit the state space, Kukharenko et al. developed a restricted model, in which a node (analogous to our *Block*) can be extended only by one of two nodes, and a more general model in which any node can extend any other (from some fixed set). The restricted model, with just four peers (one byzantine), took over seven hours to check. The more general model took over 17 days. Our approach imposes no such limitations, and Agda checks our proofs in under one minute. Finally, for the more general model, TLA+ estimates an "optimistic" probability of 0.3 that it has in fact not explored the entire state space due to hash collisions on states, leaving open the possibility of an unfound bug even for this minimal configuration. We consider that Kukharenko et al.'s work complements ours, but does not obviate the need for the machine-checked correctness proofs.

4.2 Other BFT consensus protocols

Pîrlea and Sergey present Toychain [33, 34], which models Nakamoto consensus [31] and proves correctness properties about it using Coq [6]. Although Nakamoto consensus differs substantially from HotStuff/LibraBFT, Toychain is the closest prior work to ours in terms of modeling structures (collections of trees of records) and reasoning about their properties. Their model can be instantiated with different implementation components, and they prove that any implementation that provides components satisfying certain requirements is correct. In contrast, each of the LibraBFT-related efforts mentioned above [20, 22, 25] proves properties about one particular model of HotStuff/LibraBFT.

While Toychain indeed establishes some generality by enabling instantiation with specific components, we impose no structure whatsoever on an implementation: if the externally visible behaviour of honest peers for a given implementation complies with two precisely stated rules, then that implementation can inherit the correctness properties we have proved of the abstract model.

Toychain initially assumed an injective hash function, which requires trusting that the proofs do not abuse the power granted by a false assumption. Interestingly, subsequent versions of Toychain addressed this issue by removing the assumption that the hash function used is injective. The bulk of Chapter 3 of Pîrlea's thesis [33] is devoted to describing the complexity that this undertaking involved, reporting that *every* proof had to be changed, and citing an example of one proof that grew from 10 lines to 150 to accommodate this enhancement!

In contrast, as described in Section 3.6, we have taken a different approach. Our abstract model is aware only of ids assigned to *Block*s that an implementation considers to be "in the system", not hash functions. We too rested our initial development on an unsound foundation by assuming that ids were injective. However, because our abstraction freed us from reasoning about hash functions in our correctness proofs, it was not particualrly disruptive to later augment our proofs to provide evidence of *specific* injectivity failures when necessary, tying those injectivity failures to *Record*s that the implementation considers to be in the system.

The work that is perhaps closest to our broader project is Velisarios [35], which uses the Coq theorem prover [6] and provides a framework for modeling distributed systems with byzantine peers, analogous to our system model. It is based on a Logic of Events [29] approach, in contrast to our state transition system approach. Velisarios is instantiated with definitions modeling PBFT [15] to prove PBFT correct. Coq supports extraction to OCaml, enabling an implementation to be derived from the PBFT model. Agda has support for extracting to Haskell or Javascript. However, we have not experimented with this. The goal of our ongoing work is to model our practical Haskell implementation in Agda and prove correctness for that model using the results presented in this paper.

Alturki et al. [2] use Coq to formally verify correctness of Algorand's [21] consensus protocol. Their correctness condition is slightly different as Algorand's protocol seeks to ensure that exactly one block is certified per round, implying a total order on all certified blocks. Crary [17] reports on work towards verifying correctness for the consensus mechanism of Hashgraph [4] in Coq. Losa and Dodds [28] describe formal verification of safety and liveness properties for the Stellar consensus protocol using Ivy and Isabelle/HOL. Alturki et al. [3] use Coq to formally verify properties for Gasper [11]—Ethereum 2.0's Proof of Stake consensus mechanism. Rather than assuming that any two quorums intersect on at least one honest node, they prove that, if (using our terminology) two conflicting blocks are committed, then there exist two quorums whose common members can have their stake slashed. This property would be satisfied if only the first offense results in slashing; presumably, a stronger property that ties the conflicting blocks to specific quorums related to those blocks could be proved.

There is also work model checking other BFT consensus protocols. For example, Tholoniat and Gramoli [38] have used ByMC [24] to model check RedBelly's consensus algorithm [16]; ByMC is a model checker designed to mitigate the state space blowup for algorithms in which processes wait for a threshold of messages. While basic HotStuff may fit this structure, chained HotStuff does not.

Braithwaite et al. [8] report on work in progress towards model checking Tendermint [9] using TLA+; so far, they have gained useful insight into the algorithm using very small configurations, and have found and fixed some specification bugs as a result. Nonetheless, their experience again highlights the challenges of model checking related to state space and execution time.

5 Concluding remarks and future work

We have presented a formal model of the essence of a Byzantine Fault Tolerant consensus protocol used in several existing implementations, and proved its safety properties—including one that enables non-participants to verify commits—for a single epoch, during which configuration does not change. Extending our proofs to accommodate epoch changes (reconfiguration) is future work.

Our contributions include precisely defining implementation assumptions and correctness conditions, and developing formal, machine-checked proofs of correctness properties for any implementation satisfying the assumptions. Our model, definitions, and proofs are all expressed in Agda, and are available in open source.

Our approach enables verifying implementations by proving only that honest peers obey the rules established by our abstract assumptions, without repeating the hard work of proving the underlying protocol correct each time.

Our *thmS5* property establishes correctness unless it can provide *evidence* of a *specific* injectivity failure between *Block*s that are *in the system*. Thus our proofs are independent of how specific implementations assign *Block* ids, and ensure that they hold unless and until an injectivity failure actually occurs. In this way, our abstract proofs support proving that implementations that use crypotgraphic hash functions to assign ids behave correctly, based on the standard assumption that a computationally bounded adversary cannot produce a hash collision.

In our broader project [7], we have defined a system model in which messages can be lost, duplicated and arbitrarily delayed, and dishonest peers are constrained only by their inability to forge signatures of honest peers. We have ported our Haskell implementation to Agda using a library we have developed [13], instantiated our system model with its types and handlers, and made substantial progress towards proving that it satisfies the required assumptions.

Beyond that, extending our system model to support proofs of liveness in the partial synchrony model [19] is future work. A pragmatic intermediate point is to prove within our existing system model that, from any reachable state that has *Block*s available to commit, there is some execution in which another *Block* is committed (called *plausible liveness* by Buterin and Griffith [10]). These liveness properties would pertain to a model of a specific *implementation*; liveness properties do not make sense for the abstract model presented in this paper.

Bibliography

[1] Agda 2.6.1.1 documentation (May 2021), agda.readthedocs.io/en/v2.6.1.1

[2] Alturki, M.A., Chen, J., Luchangco, V., Moore, B., Palmskog, K., Peña, L., Rosu, G.: Towards a verified model of the algorand consensus protocol in coq. Formal Methods. FM 2019 International Workshops p. 362–367 (2020), dx.doi.org/10.1007/978-3-030-54994-7_27

[3] Alturki, M.A., Li, E., Park, D., Moore, B., Palmskog, K., Peña, L., Rosu, G.: Verifying Gasper with dynamic validator sets in Coq (2020), github.com/runtimeverification/beacon-chain-verification/blob/master/casper/report/report.pdf

[4] Baird, L.: The Swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. Tech. Rep. SWIRLDS-TR-2016-01 (2016), www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf

[5] Baudet, M., Ching, A., Chursin, A., Danezis, G., Garillot, F., Li, Z., Malkhi, D., Naor, O., Perelman, D., Sonnino, A.: State machine replication in the Libra blockchain (2019), developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2019-06-28.pdf

[6] Bertot, Y., Castran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions. Springer Publishing Company, Incorporated, 1st edn. (2010)

[7] BFT consensus in Agda (December 2021), github.com/oracle/bft-consensus-agda/releases/tag/nasafm2022

[8] 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. 2nd Workshop on Formal Methods for Blockchains (2020), drops.dagstuhl.de/opus/volltexte/2020/13423/pdf/OASIcs-FMBC-2020-10.pdf

[9] Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on BFT consensus (2018). https://doi.org/10.48550/ARXIV.1807.04938, arxiv.org/abs/1807.04938

[10] Buterin, V., Griffith, V.: Casper the friendly finality gadget. CoRR **abs/1710.09437** (2017), arxiv.org/abs/1710.09437

[11] Buterin, V., Hernandez, D., Kamphefner, T., Pham, K., Qiao, Z., Ryan, D., Sin, J., Wang, Y., Zhang, Y.X.: Combining GHOST and Casper (2020). https://doi.org/10.48550/ARXIV.2003.03052, arxiv.org/abs/2003.03052

[12] Cachin, C., Vukolic, M.: Blockchain consensus protocols in the wild. CoRR **abs/1707.01873** (2017), arxiv.org/abs/1707.01873

[13] Carr, H., Jenkins, C., Moir, M., Miraldo, V.C., Silva, L.: An approach to translating Haskell programs to Agda and reasoning about them (2022). https://doi.org/10.48550/ARXIV.2205.08718, arxiv.org/abs/2205.08718

[14] Carr, H., Jenkins, C., Moir, M., Miraldo, V.C., Silva, L.: Towards formal verification of HotStuff-based byzantine fault tolerant consensus in Agda: Extended version (2022), arxiv.org/abs/2203.14711

[15] Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Proceedings of the Third Symposium on Operating Systems Design and Implementation. p. 173–186. OSDI '99, USENIX Association, USA (1999)

[16] Crain, T., Gramoli, V., Larrea, M., Raynal, M.: DBFT: Efficient leaderless byzantine consensus and its application to blockchains. In: 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA). pp. 1–8 (2018), doi.org/10.1109/NCA.2018.8548057

[17] Crary, K.: Verifying the hashgraph consensus algorithm (2021), arxiv.org/abs/2102.01167

[18] diem.com: Diem github repository (2021), github.com/diem/diem

[19] Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (Apr 1988), doi.org/10.1145/42282.42283

[20] Garillot, F., Siles, V.: Librachain (May 2020), github.com/novifinancial/LibraChain

[21] Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. In: Proceedings of the 26th Symposium on Operating Systems Principles. p. 51–68. SOSP '17, Association for Computing Machinery, New York, NY, USA (2017), doi.org/10.1145/3132747.3132757

[22] Jehl, L.: Formal verification of HotStuff. In: Peters, K., Willemse, T.A.C. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 41st IFIP WG 6.1 International Conference, FORTE 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12719, pp. 197–204. Springer (2021), doi.org/10.1007/978-3-030-78089-0_13

[23] King, S., Nadal, S.: PPCoin: Peer-to-peer crypto-currency with proof-of-stake (2012), web.archive.org/web/20171211072318/peercoin.net/assets/paper/peercoin-paper.pdf

[24] Konnov, I., Widder, J.: Bymc: Byzantine model checker. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11246, pp. 327–342. Springer (2018), doi.org/10.1007/978-3-030-03424-5_22

[25] Kukharenko, Vladimir, Ziborov, Kirill, Sadykov, Rafael, Rezin, Ruslan: Verification of HotStuff BFT consensus protocol with TLA+/TLC in an industrial setting. SHS Web Conf. **93**, 01006 (2021), doi.org/10.1051/shsconf/20219301006

[26] Kuppe, M.A., Lamport, L., Ricketts, D.: The TLA+ toolbox. Electronic Proceedings in Theoretical Computer Science **310**, 50–62 (Dec 2019), dx.doi.org/10.4204/EPTCS.310.6

[27] Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (May 1998), doi.org/10.1145/279227.279229

[28] Losa, G., Dodds, M.: On the formal verification of the Stellar consensus protocol. 2nd Workshop on Formal Methods for Blockchains (2020), drops.dagstuhl.de/opus/volltexte/2020/13422/pdf/OASIcs-FMBC-2020-9.pdf

[29] Mark Bickford, Robert L. Constable, V.R.: Logic of events, a framework to reason about distributed systems. In: 2012 Languages for Distributed Algorithms Workshop. Philadelphia, PA (2012), www.nuprl.org/documents/Bickford/LOE-LADA2012.html

[30] Menezes, A.J., Vanstone, S.A., Oorschot, P.C.V.: Handbook of Applied Cryptography. CRC Press, Inc., USA, 1st edn. (1996)

[31] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2009), www.bitcoin.org/bitcoin.pdf

[32] Norell, U.: Dependently typed programming in Agda. In: Proceedings of the 6th International Conference on Advanced Functional Programming. pp. 230–266. AFP'08, Springer-Verlag, Berlin, Heidelberg (2009), doi.org/10.1007/978-3-642-04652-0_5

[33] Pîrlea, G.: Toychain formally verified blockchain consensus (April 2020), pirlea.net/papers/toychain-thesis.pdf

[34] Pîrlea, G., Sergey, I.: Mechanising blockchain consensus. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 78–90. CPP 2018, Association for Computing Machinery, New York, NY, USA (2018), doi.org/10.1145/3167086

[35] Rahli, V., Vukotic, I., Völp, M., Esteves-Verissimo, P.: Velisarios: Byzantine fault-tolerant protocols powered by Coq. In: Ahmed, A. (ed.) Programming Languages and Systems. pp. 619–650. Springer International Publishing, Cham (2018)

[36] Spreitzer, M.J., Theimer, M.M., Petersen, K., Demers, A.J., Terry, D.B.: Dealing with server corruption in weakly consistent, replicated data systems. In: Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking. pp. 234–240. MobiCom '97, ACM, New York, NY, USA (1997), doi.acm.org/10.1145/262116.262151

[37] The LibraBFT Team: State machine replication in the Libra blockchain (May 2020), developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2020-05-26.pdf

[38] Tholoniat, P., Gramoli, V.: Formal verification of blockchain byzantine fault tolerance (2019). https://doi.org/10.48550/ARXIV.1909.07453, arxiv.org/abs/1909.07453

[39] Wadler, P., Kokke, W.: Programming language foundations in Agda (2009), doi.org/10.1016/j.scico.2020.102440

[40] Wikipedia: Universe (mathematics) (February 2021), en.wikipedia.org/wiki/Universe_(mathematics)

[41] Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: HotStuff: BFT consensus with linearity and responsiveness. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. p. 347–356. PODC '19, Association for Computing Machinery, New York, NY, USA (2019), doi.org/10.1145/3293611.3331591