Verifying Safety of the CBFT Consensus Protocol in Coq

Elaine Li[1], Karl Palmskog[2], Mircea Sebe[1], and Grigore Roşu[1]

[1] Runtime Verification, Inc., Urbana, IL, USA
{elaine.li,mircea.sebe,grigore.rosu}@runtimeverification.com
[2] KTH Royal Institute of Technology, Stockholm, Sweden
palmskog@acm.org

Abstract

The Concurrent Byzantine Fault Tolerant (CBFT) consensus protocol is used to validate transactions and computations in the PlatON network. In this report, we describe a model of CBFT in the Coq proof assistant, and show how several key safety properties of the protocol are encoded and formally proved.

1 Introduction

The PlatON network provides a platform for distributed transactions and computations. The network relies on the Concurrent Byzantine Fault Tolerant (CBFT) consensus protocol to reach agreement among participating nodes on which sets of transactions (blocks) to add to a distributed ledger (blockchain) [1]. CBFT is a three-phase consensus protocol in the partially synchronous mesh communication model. The protocol is designed to tolerate that up to one third of all participating nodes are Byzantine, i.e., behave adversarially.

Due to the exponentially many possible protocol scenarios that are possible in a distributed setting with delays and Byzantine faults, traditional techniques for establishing protocol safety, such as testing, are not able to universally guarantee safety properties of CBFT.

In this report, we describe a model of CBFT in the Coq proof assistant [2], and how we use the model to state and formally verify several safety properties of CBFT, increasing the trustworthiness of the protocol. More specifically, we state and prove (1) prepare stage height injectivity, (2) precommit stage height injectivity, and (3) commit stage height injectivity. Thanks to the foundational type theory underpinning Coq, we are able to reason symbolically about all possible executions of CBFT, including those where nodes have adversarial behavior.

1.1 Scope of Formal Verification

Formal verification can only reduce the correctness properties of a model of a system to a set of underlying assumptions. As such, even when a verification process is successful and sound, it is possible that the model may be an incorrect representation of the system, and that the proven properties may be inadequate to capture system correctness. However, Coq and similar proof assistants based on a *small trusted proof-checking kernel* provide strong guarantees about proofs, meaning there is very little doubt in a successful verification process outcome. This is in contrast to less rigorous verification methods which require users to trust large programs and theories that cannot be manually inspected and validated in detail with reasonable effort.

1.2 Organization and Trusted Base

The report is organized as follows. Section 2 provides brief background on CBFT and contrasts our verification approach in Coq with others for similar protocols in the literature. Section 3 describes the basic data and assumptions (parameters and axioms) that comprise our model of CBFT. These definitions are the substrate for stating and proving the safety properties of the protocol, and must be accepted to trust any verification results about the model. Section 4 describes how the correctness properties are encoded and surveys how the properties were proved. Note that besides our custom parameters and axioms, we also use the *law of excluded middle* (`classic`) axiom from Coq's standard library in the formal proofs. Finally, Section 5 outlines some possible extensions to our Coq model and contextualizes the work.

To trust our Coq proofs, it is necessary to understand the property encodings and affirm that they capture the intentions behind the protocol. On the other hand, since Coq's kernel is trustworthy, there is little need to understand the details of Coq *proof scripts* (which build the formal proofs of properties when compiled by Coq) and are usually found between the keywords `Proof` and `Qed`. Our presentation here thus focuses on encodings of data and properties, and in providing *intuitions* behind the formal proofs; interested readers are encouraged to obtain the code, which is available online [3], and check it locally with Coq for additional certainty.

2 Background and Related Work

2.1 The CBFT protocol

A rigorous definition of CBFT is available in related work [4]. Here, we only summarize the key details relevant to formal specification and verification.

In CBFT, a block goes through three stages: Prepare, Precommit and Commit. The latter two block stages are defined in terms of its former: a block is in Precommit (respectively, Commit) stage if its parent block is in Prepare (respectively, Precommit) stage. Once a block reaches the Precommit and Commit stage, it is considered *final.* Therefore, consensus in CBFT means that all participating nodes agree on a unique block at each height at the Commit stage. This in turns requires that all participating nodes agree on a unique block at each height at the Precommit stage, which requires that all participating nodes agree on a unique block at each height at the Prepare stage in a single view. In this report, we focus on stating and proving these three safety properties in Coq.

2.2 Distributed systems verification in Coq

Verification of fault-tolerant distributed systems in Coq have been done in three principal ways.
Distributed separation logic: This approach is used in the Disel [5] and Aneris [6] frameworks to reason about the safety properties of two-phase commit protocols. However, there is to our knowledge no application of these framework for establishing Byzantine fault tolerance.
Lamport's happened-before relation: This approach is used by Rahli et al. [7] to verify an implementation of the Practical Byzantine Fault Tolerance (PBFT) protocol in the Velisarios framework in Coq, using arbitrary local actions to capture adversarial behavior.
Reasoning inductively on transition system relations: This approach is based on explicitly defining a transition system as a relation on global states and directly establishing properties of reachable states by induction. This is what we use here, and was also notably used in the verification of an implementation of the Raft distributed consensus protocol [8] in the Verdi framework in Coq [9]. However, Raft only provides tolerance to regular crash faults, not Byzantine node behavior.

Pîrlea and Sergey [10] modeled a generic blockchain system in Coq and prove agreement on a canonical blockchain for a system in quiescence, while abstracting from most consensus mechanism details. Closest to our work is the verified Coq model of the Algorand consensus protocol by Alturki et al. [11], which similarly abstracts from details on blocks and blockchains and captures adversarial behavior. However, the Algorand consensus protocol and CBFT use widely different assumptions on network behavior and message delivery, making the respective Coq models essentially incomparable.

In a similar setting to ours, Alturki et al. [12] proved correct in Coq a *finality* mechanism for the Ethereum blockchain. Most recently, Thomsen and Spitters [13] verified safety and liveness properties in Coq of a model of a Nakamoto-style blockchain based on proof-of-stake which abstracts from details of the consensus mechanism.

3 Protocol Model and Coq Encoding

In this section, we outline our model of CBFT and its Coq encoding. We designed our model mainly to illuminate the safety properties of CBFT. Therefore, we elide many features of a real-world implementation of CBFT. To explain the Coq encoding, we provide fragments of code in Coq's Gallina specification language that are adapted from the complete CBFT Coq code [3], but are (for presentation reasons) not always copied verbatim. For a full explanation of Gallina syntax, we recommend consulting the Coq manual [14] or other relevant literature [2].

3.1 Basic Datatypes and Assumptions

In this section, we present the basic datatypes of CBFT, which we call *parameters*, and their accompanying assumptions, which we call *axioms*. Declaration of global parameters and axioms (using the synonymous keywords `Parameter` and `Axiom`) is a common and convenient abstraction mechanism in Coq. We use this mechanism to avoid specifying unnecessary details that are best left to an implementation of CBFT. However, there is no general guarantee in Coq that parameters and axioms can be instantiated. Hence, we have chosen a minimal set which we are confident can be instantiated with only modest effort. We also highlight some derived datatypes, such as messages and node state.

3.1.1 Participating nodes

A static set of nodes participate in each round in CBFT, and perform actions such as proposing blocks, voting on blocks, broadcasting messages and requesting messages. Each node is associated with a unique identifier; we therefore represent participating nodes as a type equipped with decidable equality.

```
Parameters (node : Type) (node_eqb : node → node → bool).
Axiom node_eqb_correct : ∀ n1 n2, node_eqb n1 n2 = true ↔ n1 = n2.
```

We assume that the equality of all key CBFT parameters is decidable. Therefore, we omit future statements of decidable equality from all the parameters below.

We define a static set of participants in the protocol as a list of node identifiers.

```
Parameter participants : list node.
```

3.1.2 Quorums

In order to express the Byzantine assumption that no more than two thirds of participating nodes are malicious, we need to formalize the notion of quorums. CBFT's definition of quorum is a property over a set of messages, which informally states that "more than two thirds participating nodes have sent a certain kind of message for a certain block".

We define a quorum and its complement over sets of nodes as abstract functions over lists of nodes paired with a specification which captures the critical properties they must satisfy. Many concrete implementations of the notion of quorum can therefore be used; our approach does not commit us to any particular one.

```
Parameter has_at_least_two_thirdsb : list node → bool.
Definition has_at_least_two_thirds (l : list node) : Prop :=
 has_at_least_two_thirdsb l = true.
```

One important specification property for the functions above is that any superset of a quorum set must also be a quorum set.

```
Axiom quorum_subset : ∀ lm1 lm2,
 quorum lm1 → (∀ msg, In msg lm1 → In msg lm2) → quorum lm2.
```

Additionally, by the pigeonhole principle, any two quorum sets of nodes must have a common subset that contains at least one third of the nodes.

```
Axiom intersection_property : ∀ (l1 l2 : list node),
 has_at_least_two_thirds l1 → has_at_least_two_thirds l2 →
 ∃ (l : list node), has_at_least_one_third l ∧
  ∀ (n : node), In n l → In n l1 ∧ In n l2.
```

We then define message-based quorum in terms of node-based quorum as follows.

```
Definition quorum (lm : list message) : Prop :=
 has_at_least_two_thirds (map get_sender lm).
```

Given an abstract parameter representing node honesty, we can then express the assumption that no more than one third of participating nodes are Byzantine as follows.

```
Parameter honest_nodeb : node → bool.
Axiom evil_participants : ¬ has_at_least_one_third
 (filter (fun n => negb (honest_nodeb n)) participants).
```

We define honesty behaviorally using the state transition relation, i.e., a node is honest if it makes only valid, protocol-following transitions.

3.1.3 Blocks

The safety properties of CBFT are stated in terms of blocks rather than block trees or blockchains. Therefore, the assumptions that we require about blocks are much weaker than in other blockchain formalizations, such as Toychain [10].

Block parameters. We model blocks as an arbitrary Coq type with a special *genesis block* as inhabitant. We equip all blocks with a height and an index.

```
Parameters (block : Type) (GenesisBlock : block).
Parameters (b_height : block → nat) (b_index : block → nat).
```

We also assume blocks have decidable equality.

```
Parameter block_eqb : block → block → bool.
Axiom block_eqb_correct : ∀ b1 b2, block_eqb b1 b2 = true ↔ b1 = b2.
```

Block ancestry. In practice, block hashes are used primarily to associate a block with its parent. We capture this using a primitive *parent block* relation on blocks that is defined by block generation, i.e., a block is the parent of any block that is generated based on it.

```
Parameters (generate_new_block : block → block) (parent_of : block → block).
Axiom generate_new_block_parent : ∀ b, parent_of (generate_new_block b) = b.
```

The parent block relation is also useful to determining block height: child block height is always parent block height plus one.

```
Axiom parent_block_height : ∀ b, S (b_height (parent_of b)) = b_height b.
```

In combination, these definitions allow for a block to have multiple children block at the same height, so that we can model malicious participating nodes potentially proposing multiple conflicting blocks at the same height.

Last blocks. A certain kind of block has special status in the CBFT, namely, the last block produced in a view. While the specific number of blocks produced per view may vary per round, there is always a unique last block for each view, and nodes are always able to recognize it when they see it. Therefore, we model the property of being the last block as an abstract parameter on blocks. We model last block uniqueness again using block generation, by defining a special block generation function for last blocks.

```
Parameters (b_last : block → bool) (generate_last_block : block → block).
```

We express the necessary properties of these functions using axioms:

```
Axiom about_generate_last_block : ∀ b, b_last (generate_last_block b) = true ∧
 b_height (generate_last_block b) = S (b_height b).
Axiom about_non_last_block : ∀ b, b_last (generate_new_block b) = false.
```

3.2 Messages

We use a Coq inductive type to the define the five CBFT message types.

```
Inductive message_type :=
| PrepareBlock | PrepareVote | ViewChange | PrepareQC | ViewChangeQC.
```

We then define a message as a Coq record:

```
Record message := mkMessage { get_message_type : message_type; get_view : nat;
 get_sender : node; get_block : block; get_piggyback_block : block }.
```

According to this definition, a CBFT message consists of a message type (`get_message_type`), a view as a natural number (`get_view`), the node identifier of the message sender (`get_sender`), a primary block (`get_block`) and a piggyback block (`get_piggyback_block`).

While all CBFT protocol messages contain at least one block, some contain additional information in the form of aggregated signatures of other messages. The signature effectively stands in for the messages themselves: if a node receives a message m containing an aggregated signature sig created using a set of message signatures $\{sig_{m_1}, sig_{m_2}, ..., sig_{m_n}\}$, the node behaves as though it has received the set of messages $\{m, m_1, m_2, ..., m_n\}$. We include an additional message field to model this "message carrying" behavior without explicitly resorting to, e.g., recursive message definitions to model messages containing other messages.

3.3 Local states

We define a record type `NState` that contains all the data available to a protocol participant during execution.

```
Record NState := mkNState { node_view : nat; node_id : node;
 in_messages : list message; counting_messages : list message;
 out_messages : list message; timeout : bool }.
```

In this definition, `node_view` is the current view number, `node_id` is the unique node identifier, `in_messages` is a message buffer containing all delivered messages that have been processed, `out_messages` is a message buffer containing all sent messages, and `timeout` is a flag indicating whether the current view has timed out. Given a node identifier `n`, we then define the initial state of a node:

```
Definition NState_init (n : node) : NState := mkNState 0 n [] [] [] false.
```

3.4 Local state transitions

The actions of a participating node in the protocol mainly consist of receiving and broadcasting messages. We model node behavior using local state transitions, and in turn define protocol-following node behavior as valid local state transitions that are a relation over 1) current state, 2) incoming message, 3) outgoing message(s), and 4) updated state. We refer to 1) and 4) as pre-state and post-state, respectively.

To represent local transitions abstractly, we define labels for them as a Coq inductive type:

```
Inductive NState_transition_type :=
| propose_block_init_type
| discard_view_invalid_type
| process_PrepareBlock_duplicate_type
| process_PrepareBlock_pending_vote_type
| process_PrepareBlock_vote_type
| process_PrepareVote_vote_type
| process_PrepareVote_wait_type
| process_PrepareQC_last_block_new_proposer_type
| process_PrepareQC_last_block_type
| process_PrepareQC_non_last_block_type
| process_ViewChange_quorum_new_proposer_type
| process_ViewChange_pre_quorum_type
| process_ViewChangeQC_single_type
| process_PrepareBlock_malicious_vote_type.
```

We then define one Coq predicate for each of the above local transition types. As an example, we show the full definition of the predicate corresponding to `process_PrepareBlock_vote_type`:

```
Definition process_PrepareBlock_vote (s s' : NState) (msg : message)
 (lm : list message) : Prop :=
  s' = record_plural (process s msg) (pending_PrepareVote s msg) ∧
  lm = pending_PrepareVote s msg ∧
  received s msg ∧
  honest_node (node_id s) ∧
  get_message_type msg = PrepareBlock ∧
```

```
8    view_valid s msg ∧
9    timeout s = false ∧
10   prepare_stage s (parent_of (get_block msg)).
```

This predicate defines a CBFT transition where a node votes for a block whose parent has reached Prepare stage.

- Line 3 defines the post-state: the node moves the incoming message from `in_messages` to `counting_messages`, and adds `pending_PrepareVote s msg` to its `out_messages` buffer to record as sent message history.
- Line 4 defines the messages to be broadcast on the global network.
- Line 5 is included in all valid CBFT transitions, and states that the message is indeed received by the transition-making.
- Lines 6 and 8 are included in all honest CBFT transitions, and state that the transition-making node is honest and the message was produced in the current view, and can be processed.
- Line 7 states that the message being processed is a `PrepareBlock` message, which ensures that suitable protocol actions are being taken.
- Line 9 is included in all PrepareBlock and PrepareVote-processing transitions – nodes cannot process these messages once a timeout has occurred.
- Line 10 states the condition for nodes to be able to vote for a block: its parent block must have reached Prepare stage in its local state.

3.5 Global states

We define a CBFT global state as a tuple containing (1) a mapping of node identifiers to their states, and (2) a list of all messages ever sent over the network:

```
Definition GState : Type := (node → NState) * list message.
```

We also define an *initial state* that serves as the starting point of any protocol execution:

```
Definition GState_init : GState := (fun (n : node) => NState_init n, []).
```

In the following, we use the Coq standard library functions `fst` and `snd` to obtain the state mapping and the message lists from global states, respectively. For example, `snd GState_init` yields the empty list, i.e., `[]`.

3.6 Global state transitions

We define the global transition relation as a binary relation on global states. There are two kinds of transitions: *process* steps and *timeout* steps. In a process step, a single participating node performs some process (which may change its state) and broadcasts the resulting messages. In a timeout step, the timeout flags of all participating nodes are flipped. We represent these two steps as clauses (constructors) in a Coq inductive predicate:

```
1  Inductive GState_step : GState → GState → Prop :=
2  | GState_step_process : ∀ (n : node) (process : NState_transition_type)
3    (msg : message) (lm : list message) (g g' : GState),
4     In n participants →
5     get_transition process (fst g n) msg (fst g' n) lm →
6     g' = broadcast_messages g (fst g n) (fst g' n) lm →
7     GState_step g g'
8  | GState_step_timeout : ∀ (g g' : GState),
9      g' = ((fun n => if is_member n participants then
10         flip_timeout (fst g n) else fst g n), snd g) →
11    GState_step g g'.
```

Intuitively, to take a process step according to this relation, the involved node must be a participant (line 4), the state and generated messages must be according to a local transition predicate (line 5), and the updated global state `g'` must have the updated state and broadcasted messages (line 6). To take a timeout step, all participants (line 9) must have their timeout flags flipped in the updated state (line 10).

3.7 Protocol traces

To reason about executions of CBFT, we need to consider sequences of global states where each adjacent pair follows the global transition relation. We first define traces as functions from natural numbers to global states:

```
Definition GTrace : Type := nat → GState.
```

Then, we define protocol traces that start from the initial global state and take steps from one natural number to its successor according to the transition relation:

```
Definition protocol_trace (tr : GTrace) : Prop :=
  tr 0 = GState_init ∧ ∀ n : nat, GState_step (tr n) (tr (S n)).
```

This definition is sufficient to allow us to establish protocol state invariants (i.e., safety properties) by induction on natural numbers.

4 Safety Properties and Proofs

In this section, we define the three key safety properties of CBFT sketched in Section 2.1 in Coq and outline their formal proofs. For more details on the the proofs and the helper lemmas they depend on, we refer the reader to the Coq code [3] and in particular to its generated HTML documentation, which makes cross-references easily accessible.

4.1 Prepare stage height injectivity

A block is considered to be at the Prepare stage for some node if it has received quorum `PrepareVote` messages or a `PrepareQC` message in the current view or some previous view. We first define the Prepare stage for some block for a particular view.

```
Definition vote_quorum_in_view (s : NState) (view : nat) (b : block) : Prop :=
  quorum (processed_PrepareVote_in_view_about_block s view b).
```

```
Definition PrepareQC_in_view (s : NState) (view : nat) (b : block) : Prop :=
  ∃ msg : message, In msg (counting_messages s) ∧
    get_view msg = view ∧
    get_block msg = b ∧
    get_message_type msg = PrepareQC.

Definition prepare_stage_in_view (s : NState) (view : nat) (b : block) : Prop :=
  vote_quorum_in_view s view b ∨ PrepareQC_in_view s view b.
```

We then define the Prepare stage as an existential proposition over view numbers:

```
Definition prepare_stage (s : NState) (b : block) :=
  ∃ v', v' ≤ node_view s ∧ prepare_stage_in_view s v' b.
```

4.1.1 Property statement

The first safety property states that no two blocks of the same height can be at Prepare stage in the same view, i.e., that prepare stage block height is injective in the same view. This property differs from the following two in that it contains a view restriction as a premise. This premise is important because it is possible for multiple blocks at the same height to reach Prepare stage across different views, as we will see later in the case of abnormal view changes.

```
Definition prepare_stage_same_view_height_injective_statement :=
  ∀ (tr : GTrace), protocol_trace tr →
    ∀ (i : nat) (n m : node) (b1 b2 : block) (p : nat),
      In n participants →
      In m participants →
      b1 ≠ b2 →
      prepare_stage_in_view (fst (tr i) n) p b1 →
      prepare_stage_in_view (fst (tr i) m) p b2 →
      b_height b1 = b_height b2 →
      ⊥.
```

4.1.2 Proof outline

Intuitively, the proof of this property follows directly from the definition of non-Byzantine/honest voting behavior: honest nodes cannot vote for two conflicting blocks during the same view. If two conflicting blocks reach prepare stage in the same view, then two quorums of at least 2/3 validators voted for each block. By the pigeonhole principle, there must exist a set of at least 1/3 validators who voted for both conflicting blocks and are therefore Byzantine/dishonest. By assumption, there are no more than 1/3 Byzantine/dishonest blocks. Therefore, we reach a contradiction (written ⊥ or `False` in Coq).

We formalize the connection between more than 1/3 dishonest nodes and node-local premises using a series of intermediate facts. Because our fault model does not include the behavior of sending two `PrepareVote` messages for different blocks of the same height in the same view, we cannot make use of the premises directly. Instead, we make use of the fact that every node must have processed a `PrepareBlock` message in order to send a `PrepareVote` message.

```
Lemma sent_PrepareVote_means_received_PrepareBlock :
  ∀ (tr : GTrace), protocol_trace tr →
```

```
    ∀ (n : node) (i : nat) (msg : message),
      In n participants →
      In msg (out_messages (fst (tr i) n)) →
      get_message_type msg = PrepareVote →
      ∃ (msg' : message),
        In msg' (counting_messages (fst (tr i) n)) ∧
        get_message_type msg' = PrepareBlock ∧
        get_block msg'= get_block msg ∧
        get_view msg' = get_view msg.
```

Next, we show that if there is evidence of two conflicting sent PrepareVote messages, there must also be evidence of two conflicting processed PrepareBlock messages.

```
Definition equivocating_in_state (s : NState) : Prop :=
  ∃ (msg1 msg2 : message),
    In msg1 (out_messages s) ∧
    In msg2 (out_messages s) ∧
    get_view msg1 = get_view msg2 ∧
    get_message_type msg1 = PrepareVote ∧
    get_message_type msg2 = PrepareVote ∧
    get_block msg1 ≠ get_block msg2 ∧
    b_height (get_block msg1) = b_height (get_block msg2).

Definition pre_equivocating_in_state (s : NState) : Prop :=
  ∃ (msg1 msg2 : message),
    In msg1 (counting_messages s) ∧
    In msg2 (counting_messages s) ∧
    get_view msg1 = get_view msg2 ∧
    get_message_type msg1 = PrepareBlock ∧
    get_message_type msg2 = PrepareBlock ∧
    get_block msg1 ≠ get_block msg2 ∧
    b_height (get_block msg1) = b_height (get_block msg2).

Lemma pre_local_means_local_evidence_of_equivocation :
  ∀ (tr : GTrace), protocol_trace tr →
    ∀ (n : node) (i : nat),
      In n participants →
      equivocating_in_state (fst (tr i) n) →
      pre_equivocating_in_state (fst (tr i) n).
```

Because our fault model explicitly includes the behavior of processing two conflicting PrepareBlock messages, we can directly derive dishonesty from pre_equivocating_in_state.

```
Lemma pre_local_evidence_of_equivocation :
  ∀ (tr : GTrace), protocol_trace tr →
    ∀ (n : node) (i : nat),
      In n participants →
      pre_equivocating_in_state (fst (tr i) n) →
      ~ honest_node n.
```

We can then weaken it to derive dishonesty from equivocating_in_state.

```
Lemma local_evidence_of_equivocation :
  ∀ (tr : GTrace), protocol_trace tr →
    ∀ (n : node) (i : nat),
    In n participants →
    equivocating_in_state (fst (tr i) n) →
    ¬ honest_node n.
```

Finally, the safety theorem is stated in terms of local views of two different nodes, whereas dishonesty is stated in terms of the behavior of one node. To bridge this gap, we leverage the global out message buffer, which can be seen as a ghost variable, to derive global evidence of dishonesty from local evidence of dishonesty, to simplify the proof.

```
Lemma global_equivocating_local :
  ∀ (tr : GTrace), protocol_trace tr →
    ∀ (n : node) (i : nat),
      In n participants →
      equivocating_in_global_state (tr i) n →
      equivocating_in_state (fst (tr i) n).
```

4.2 Precommit stage height injectivity

A block is said to be at Precommit stage for some node if it is in Prepare stage and has a child block in Prepare stage.

```
Definition precommit_stage (tr : GTrace) (i : nat) (n : node) (b : block) :=
  ∃ b_child, parent_of b_child = b ∧
    prepare_stage (fst (tr i) n) b ∧
    prepare_stage (fst (tr i) n) b_child.
```

Note that this definition does not imply that the parent and child block received enough votes, i.e. *reached* Prepare stage, in the same view. Neither does it imply that the parent or child block reached Prepare stage in node `n`'s current view. It only states that there exist two views, `v` and `v'`, that are less than or equal to the current view of node `n` in protocol state `i`, such that `b` and its child reached Prepare stage in `v` and `v'` respectively. From the fact that child blocks necessarily reach Prepare stage later than their parent blocks, we know that `v` must be less than or equal to `v'`. Therefore, denoting the current view of `n` as `v0`, the relation `v` ≤ `v'` ≤ `v0` always holds.

4.2.1 Property statement

The second safety property states that no two blocks of the same height can be at Precommit stage, i.e., Precommit stage block height is injective.

```
Definition precommit_stage_height_injective_statement :=
  ∀ (tr : GTrace), protocol_trace tr →
    ∀ (i : nat) (n m : node) (b1 b2 : block),
      In n participants →
      In m participants →
      b1 ≠ b2 →
      precommit_stage tr i n b1 →
      precommit_stage tr i m b2 →
```

```
     b_height b1 = b_height b2 →
     ⊥.
```

Precommit stage safety is significantly more complex than Prepare stage safety in the same view because while the latter only reasons about messages sent in one single view `p`, the former reasons about six potentially different views:

- Node `n`'s current view `v_n = get_view (fst (tr i) n)`.
- The view during which `b1` received enough votes and reached Prepare stage `v1`.
- The view during which `b1_child` received enough votes and reached Prepare stage `v1_child`.
- Node `m`'s current view `v_m = get_view (fst (tr i) m)`.
- The view during which `b2` received enough votes and reached Prepare stage `v2`.
- The view during which `b2_child` received enough votes and reached Prepare stage `v2_child`.

A proof of Precommit stage safety involves case analysis on these six views, restricted by `v1 ≤v1_child ≤v_n` and `v2 ≤v2_child ≤v_m`.

4.2.2 Proof outline

We state the lemmas in the CBFT specification [4], and show that they are strong enough to establish all cases of Precommit stage safety. Similar to our Prepare stage proofs, we often leverage the global out message buffer as a ghost variable, and use global versions of local property statements, i.e., that depend on the global out message buffer rather than a specific node's local message buffers. We also favor the view-explicit Prepare stage definition over the general one, thus eliminating the need for existential quantifier elimination each time.

Lemma 1 in the specification [4] states that every `ViewChangeQC` message must contain either the highest or second highest Prepare stage block in its view. We first define maximum and minimum height Prepare stage block heights in a view. These definitions are global in that they look at all local Prepare stage blocks for all participating nodes, and therefore do not contain a node identifier `n` as an argument.

```
Definition is_max_prepare_height_in_view_global (tr : GTrace) (i h p : nat) :=
  (∃ b, prepare_stage_in_view_global tr i b p ∧ b_height b = h) ∧
  (∀ b, prepare_stage_in_view_global tr i b p → b_height b ≤ h).

Definition is_min_prepare_height_in_view_global (tr : GTrace) (i h p : nat) :=
  (∃ b, prepare_stage_in_view_global tr i b p ∧ b_height b = h) ∧
  (∀ b, prepare_stage_in_view_global tr i b p → b_height b ≥ h).
```

We rephrase Lemma 1 to state the first key property: for every lowest Prepare stage block in a view, its parent block must be either the highest or second-highest Prepare stage block of a past view. This property addresses not only abnormal view change cases in which the block contained in a `ViewChangeQC` message is produced upon in the new view, but also normal view change cases in which the last and highest block of the previous block is produced upon in the new view.

```
Definition prepare_stage_first_parent_highest_or_second_global :=
  ∀ tr i, protocol_trace tr →
```

```
   ∀ b1 v1, is_min_prepare_height_in_view_global tr i (b_height b1) v1 →
    ∃ v', v' ≤ v1 ∧
  (is_max_prepare_height_in_view_global tr i (b_height (parent_of b1)) v' ∨
   is_max_prepare_height_in_view_global tr i (S (b_height (parent_of b1))) v').
```

The second key property, corresponding to Lemmas 2 and 3, states that Prepare stage block height increases as view number increases. Intuitively, this is because new blocks are proposed upon a block carried over from the previous view, either through normal or abnormal view change. The carryover block is at least the second highest block in the previous view, and therefore the first block of the next view is at least the height of the highest block in the previous view.

```
Definition prepare_stage_height_view_morphism_global :=
  ∀ tr i, protocol_trace tr →
    ∀ b1 b2 v1 v2,
      prepare_stage_in_view_global tr i b1 v1 →
      prepare_stage_in_view_global tr i b2 v2 →
      v1 ≤ v2 →
      b_height b1 ≤ b_height b2.
```

If we consider only non-last Prepare stage blocks in `v1`, we can strengthen the conclusion of `prepare_stage_height_view_morphism_global` from ≤ to <.

```
Definition prepare_stage_height_view_morphism_global_non_last :=
  ∀ tr i, protocol_trace tr →
    ∀ b1 b2 v1 v2,
      prepare_stage_in_view_global tr i b1 v1 →
      prepare_stage_in_view_global tr i b2 v2 →
      ¬ last_block b1 →
      v1 ≤ v2 →
      b_height b1 < b_height b2.
```

Finally, a block that has a child block in Prepare stage in the same view cannot be the last block of the view. Intuitively, this is true by construction.

```
Definition prepare_stage_child_non_last :=
  ∀ tr i n v b, protocol_trace tr →
    In n participants →
    prepare_stage_in_view (fst (tr i) n) v b →
    (∃ b_child,
      parent_of b_child = b ∧
      prepare_stage_in_view (fst (tr i) n) v b_child) →
    ¬ last_block b.
```

We then use these four key properties to prove all cases of Precommit stage safety.

```
Lemma precommit_height_injective_symmetric :
  prepare_stage_first_parent_highest_or_second_global →
  prepare_stage_height_view_morphism_global →
  prepare_stage_height_view_morphism_global_non_last →
  prepare_stage_child_non_last →
  ∀ (tr : GTrace), protocol_trace tr →
    ∀ (i : nat) (n m : node),
      In n participants →
      In m participants →
      ∀ (b1 b2 : block),
        b1 ≠ b2 →
        precommit_stage_now tr i n b1 →
        precommit_stage_now tr i m b2 →
        b_height b1 = b_height b2 →
        ⊥.
```

Without loss of generality, assume that `b_1` reached Prepare stage before `b_2`. We proceed by contradiction on the existence of a child block for `b_1`. Non-last Prepare stage block heights strictly increase as view increases. For there to exist another block in a different view of the same height as `b_1`, it must be the case that `b_1` is the highest block of its view, and that view change occurred via timeout, whereby the next block was produced based on `b_1`'s parent block. Consequently, `b_1` will never have a child block. By assumption, `b_1` has a child block (that is in Prepare stage), which gives us a contradiction.

4.3 Commit stage height injectivity

A block is in commit stage in some local state iff it and its child block are both in Precommit stage.

```
Definition commit_stage (tr : GTrace) (i : nat) (n : node) (b : block) :=
  ∃ b_child, parent_of b_child = b ∧
    precommit_stage tr i n b ∧
    precommit_stage tr i n b_child.
```

The final proof of Commit stage safety is straightforward: we can derive Precommit stage from Commit stage by definition, and directly apply Precommit stage safety.

```
Definition commit_height_injective_statement :=
  ∀ (tr : GTrace), protocol_trace tr →
    ∀ (i : nat) (n m : node),
      In n participants →
      In m participants →
      ∀ (b1 b2 : block),
        b1 ≠ b2 →
        commit_stage tr i n b1 →
        commit_stage tr i m b2 →
        b_height b1 = b_height b2 →
        ⊥.
```

5 Conclusion

We presented an abstract model of the CBFT protocol and its encoding into the Coq proof assistant, along with formal proofs of key safety properties of CBFT. We believe our model accurately captures the *safety* properties of CBFT, and can serve as a guide for implementation and testing of CBFT. However, additional work is required to model and prove *liveness* properties of CBFT. Deriving a verified implementation by refining our model to executable code is another possible avenue of future work, and would also increase confidence in the adequacy of our model of CBFT.

Our formalization of CBFT consists of around 1800 lines of specifications in Coq's Gallina language and around 3400 lines of proof scripts, and is compatible with Coq version 8.11 [15]. The code is publicly available [3].

References

[1] PlatON Network, "PlatON consensus solution," 2020. https://devdocs.platon.network/docs/en/PlatON_Solution/.

[2] Y. Bertot and P. Casteran, *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Berlin, Heidelberg: Springer, 2004.

[3] E. Li, K. Palmskog, and M. Sebe, "CBFT model Coq code," 2020. https://github.com/runtimeverification/platon-network-verification/.

[4] E. Li, K. Palmskog, M. Sebe, and G. Roşu, "Specification of the CBFT consensus protocol," 2020. Technical report.

[5] I. Sergey, J. R. Wilcox, and Z. Tatlock, "Programming and proving with distributed protocols," *PACMPL*, vol. 2, no. POPL, pp. 28:1–28:30, 2018.

[6] M. Krogh-Jespersen, A. Timany, M. E. Ohlenbusch, S. O. Gregersen, and L. Birkedal, "Aneris: A mechanised logic for modular reasoning about distributed systems," in *Programming Languages and Systems*, pp. 336–365, 2020.

[7] V. Rahli, I. Vukotic, M. Völp, and P. Esteves-Verissimo, "Velisarios: Byzantine fault-tolerant protocols powered by Coq," in *Programming Languages and Systems*, pp. 619–650, 2018.

[8] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson, "Planning for change in a formal verification of the Raft consensus protocol," in *Certified Programs and Proofs*, pp. 154–165, 2016.

[9] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: A framework for implementing and formally verifying distributed systems," in *Conference on Programming Language Design and Implementation*, pp. 357–368, 2015.

[10] G. Pîrlea and I. Sergey, "Mechanising blockchain consensus," in *Certified Programs and Proofs*, pp. 78–90, 2018.

[11] M. A. Alturki, J. Chen, V. Luchangco, B. Moore, K. Palmskog, L. Peña, and G. Roşu, "Towards a verified model of the Algorand consensus protocol in Coq," in *Formal Methods 2019 International Workshops*, pp. 362–367, 2020.

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

[13] S. E. Thomsen and B. Spitters, "Formalizing Nakamoto-style proof of stake," 2020. https://arxiv.org/abs/2007.12105.

[14] Coq Development Team, "The Gallina specification language," 2020. https://coq.inria.fr/distrib/V8.11.2/refman/language/gallina-specification-language.

[15] The Coq Development Team, "The Coq proof assistant, version 8.11.0," Jan. 2020. https://doi.org/10.5281/zenodo.3744225.