CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

Verification of Casper in the Coq Proof Assistant

Karl Palmskog Milos Gligoric
The University of Texas at Austin
{palmskog,gligoric}@utexas.edu

Brandon Moore
Runtime Verification, Inc.
brandon.moore@runtimeverification.com

Lucas Peña
Runtime Verification, Inc.
lucas.pena@runtimeverification.com
University of Illinois at Urbana-Champaign
lpena7@illinois.edu

Grigore Roşu
Runtime Verification, Inc.
grigore.rosu@runtimeverification.com
University of Illinois at Urbana-Champaign
grosu@illinois.edu

Abstract

This report describes our effort to model and verify the Casper blockchain finality system in the Coq proof assistant. We outline the salient details on blockchain systems using Casper, describe previous verification efforts we used as a starting point, and give an overview of the formal definitions and properties proved. The Coq source files are available at: https://github.com/runtimeverification/casper-proofs

1 Introduction

The Ethereum blockchain and platform [18] is increasingly used as a financial transaction mechanism, in particular by way of smart contracts. A desirable property of a transaction mechanism is *durability*—after a user has submitted a transaction and received initial confirmation, the transaction should not be rolled back. However, in a blockchain system, there may be several competing chains of blocks that agree on the transaction history from the initial *genesis block* only up to some point. These so-called *forks* may arise as a result of network delays or adversarial behavior by some nodes, and can lead to transactions disappearing as soon as one of the forks is preferred by most nodes. To address the problem of long-ranging blockchain revisions, Buterin and Griffith proposed Casper [11], a *finality* system that overlays another block chain such as Ethereum. When enough participants in the system are honest, Casper defends both against active attacks and catastrophic crashes.

For the greatest confidence, proofs about Casper should be formalized in a mechanical proof assistant, to ensure there are no unstated assumptions or invalid steps. This report describes our effort to model and verify Casper in the Coq proof assistant, both at the abstract protocol level and at the level of a distributed blockchain system. In the terminology of Appel et al. [7], the aim is to make the Casper specification *two-sided*: implementable in Ethereum nodes and provably beneficial to Ethereum users. A further goal is to lay the foundation for making the Casper specification *live*, i.e., enable verifying Ethereum nodes at the level of executable code, or even generating the code from the specification, correct-by-construction.

Project Report, November 15, 2018, Runtime Verification, Inc.
2018.

2 Background

This section explains blockchain and Casper terminology, provides pertinent Coq background, and describes the previous formalization and verification efforts we build upon.

2.1 Blockchain and Casper Terminology

Abstractly, the global state in a blockchain system is a *block forest*, with a unique genesis block that is the root of a special *block tree*. Trees not rooted in the genesis block may be possible but are typically disregarded. As new blocks arrive to the system, nodes in the system continually establish consensus on a canonical blockchain defined by one of the leaves in the special block tree. New blocks are minted through a *proposal* mechanism, which could be an underlying blockchain using proof-of-work [16] or proof-of-stake [9] to regulate block creation. Participating nodes use a local *fork choice rule* to decide where to construct a new block onto the current block tree. Due to, e.g., delays or adversarial behavior, there may be competing leaf blocks of similar tree height, defining different blockchain *forks*.

Casper overlays a blockchain system, and intuitively works by engaging a group of autonomous *validators* who attest to, by broadcasted votes, that certain blocks in the special tree belong to the designated canonical blockchain. To participate, validators must demonstrate that they have a stake in the blockchain system by locking up a *deposit* of the blockchain's cryptocurrency. The deposit will be *slashed* if the validator is verifiably reported by other validators to be behaving adversarially.

For verification, we focus on two properties of Casper that were proved informally in [11](i.e. without using a mechanical proof assistant): accountable safety and plausible liveness. Accountable safety intuitively states that conflicting blocks in different block tree forks cannot both be finalized if more than $\frac{2}{3}$ of validators (by deposit) behave honestly. Plausible liveness states that regardless of what has happened before, it is always possible to continue to finalize blocks when more than $\frac{2}{3}$ of validators follow the protocol.

2.2 Casper Formalizations

Yoichi Hirai formalized and verified several earlier variants of Casper in the Isabelle/HOL proof assistant [15]. These formalizations are highly abstract, in the sense that they ignore most details of the structure of hashes, blocks, and validators. For example, the requirements on the fractions of honest validators is captured, via Isabelle's locale mechanism [8], by postulating abstract types $'q_1$ and $'q_2$ to represent collections of sets of validators of at least $\frac{2}{3}$ weight and sets of validators of at least $\frac{1}{3}$ weight respectively. Instead of any numerical details, the Isabelle formalization just assumes the key intersection property, that any two set of validators each of at least $\frac{2}{3}$ have a common subset of weight at least $\frac{1}{3}$. This is how that property was expressed in Isabelle:

$$\bigwedge q_1\, q_2\, .\, \exists q_3\, .\, \forall v\, .\, v \in_2 q_3 \longrightarrow v \in_1 q_1 \wedge v \in_1 q_2$$

Here, $\bigwedge$ is all-quantification, the subscripted operator $v \in_1 q$ means that validator v is a member of a set q which belongs to the type $'q_1$ that represents sets of weight at least $\frac{2}{3}$, and the subscripted operator $v \in_2 q$ means that validator v is a member of a set q which belongs to the type $'q_2$ that represents sets of weight at least $\frac{1}{3}$. In this proposition q_1 and q_2 have type $'q_1$ and q_3 has type q_2. This use of numbered "q"s as both types and variables is confusing, but exactly follows the Isabelle definition. While accountable safety is verified for the most recent Casper, plausible liveness is only proven for an earlier variant with different inter-validator messages. Moreover, these proofs were developed in an older version of Isabelle and already the proof of accountable safety cannot be checked with Isabelle2017.

2.3 Mathematical Components and Toychain

We employ several existing Coq libraries which already formalized the majority of the mathematics we need to define and reason about Casper. Mathematical Components [4] is a Coq library based on packaging mathematical structures and results in the form of Coq *canonical structures*, which can be reused and specialized when required [13]. The library was used by Gonthier et al. to capture finite group theory and prove fundamental results in abstract algebra [14]. In addition to structures from abstract algebra, the library also contains encodings of and results about many standard data structures, such as numbers, lists, and finite sets.

Toychain [6, 17] is a general formalization of blockchain systems in Coq using the Mathematical Components library. It defines blocks, forks, and distributed node state, but abstracts from specific block proposal mechanisms and procedures to let nodes decide between forks. Toychain represents a block tree as a finite map from hashes to blocks. Toychain describes the behavior of a blockchain system as a relation between global states, and establishes that absent adversarial interference, the canonical chain becomes known to all nodes in the steady state. For example, the Toychain global state is represented as a Coq record

```
Record World := mkW { localState : StateMap;
 inFlightMsgs : seq Packet; consumedMsgs : seq Packet; }.
```

where `localState` maps node names to their current block tree and other local data. We have extended and revised Toychain in collaboration with its authors to support capturing full realistic blockchain system specifications such as that for Bitcoin [2]. Our model of Casper incorporates definitions and lemmas from this extended version of Toychain.

3 Modeling and Verification Approach

This section outlines our approach to modeling and verifying Casper in Coq.

We decided to translate Hirai's Casper definitions and theorems [15] from Isabelle/HOL to Coq, and connect the resulting Casper definitions with key definitions from Toychain. At the same time, we leveraged the Toychain definitions and results to capture the behavior of nodes as found in the Casper-based beacon chain for Ethereum [1]. For tractability, we focused on translating and adapting Hirai's formal models with a *static* validator set, which are simpler (but less realistic) than the corresponding models with churn among validators.

We used concepts from the Mathematical Components library as far as possible. In particular, validators are identified by fixed-size keys so we represent validators as members of a *finite type* (having a finite number of members that can be enumerated), written `Validator : finType`. Using the library's `finType` simplifies forming and reasoning about sets of validators.

Since the formal model for establishing safety by Hirai mostly uses first-order reasoning, we were able to successfully leverage the CoqHammer extension [3, 12] to perform proofs in Coq that closely followed Isabelle/HOL proofs. At the same time, we reformulated Isabelle locale variables to Coq section variables.

In particular, the assumption on abstract set membership becomes:

```
Variables quorum_1 quorum_2 : {set {set Validator}}.
Hypothesis qs : ∀ q1 q2, q1 ∈ quorum_1 → q2 ∈ quorum_1 →
 ∃ q3, q3 ∈ quorum_2 ∧ q3 \subset q1 ∧ q3 \subset q2.
```

`quorum_1` abstracts the collection of sets of validators with combined deposits ("weight") of at least $\frac{2}{3}$ of the total. A link will be justified if every validator in some set in `quorum_1` has voted for the link. `quorum_2` abstracts the collection of sets of validators with combined deposits of at least $\frac{1}{3}$ of the total. Casper safety proofs generally show that an attack cannot succeed without a set of attacking validators of at least this size being slashed for it.

To abstract a block forest, we assume a parent relationship on a type `Hash` (with member `genesis`).

An individual validator v must not publish two distinct votes,

$$\langle v, s_1, t_1, h(s_1), h(t_1)\rangle \quad \text{and} \quad \langle v, s_2, t_2, h(s_2), h(t_2)\rangle,$$

such that either:

I. $h(t_1) = h(t_2)$. Equivalently, a validator must not publish two distinct votes for the same target height.

II. $h(s_1) < h(s_2) < h(t_2) < h(t_1)$. Equivalently, a validator must not vote within the span of its other votes

Figure 1. Slashing Conditions of Casper

```
Variable hash_parent : rel Hash.
Hypothesis hash_at_most_one_parent : ∀ h1 h2 h3,
 hash_parent h2 h1 → hash_parent h3 h1 → h2 = h3.
```

Given the block forest we define the global state as a function which represents votes cast by validators:

```
Record State :=
 mkSt { vote_msg : Validator → Hash → nat → nat → bool }.
```

The arguments are the validator making the vote, the hash and height of the target block, and the height of the source block. Blocks have at most one parent so the source hash of any valid vote can be computed.

With votes defined we can then define the slashing conditions of Casper (shown in Figure 1) in terms of conflicting votes, and define a predicate saying a validator is slashed in a given global state.

```
Definition slashed_dbl_vote s n :=
 ∃ h1 h2, h1 ≠ h2 ∧ ∃ v s1 s2,
   vote_msg s n h1 v s1 ∧ vote_msg s n h2 v s2.
```

Condition I is violated if validator `n` has voted for two edges targeting different blocks `h1` and `h2` at height `v`.

```
Definition slashed_surround s n :=
  ∃ h1 h2 v1 v2 s1 s2, v1 > v2 ∧ s2 > s1
    vote_msg s n h1 v1 s1 ∧ vote_msg s n h2 v2 s2.
```

Condition 2 is violated if validator `n` has made two votes so the source height `s1` and target height `v1` of one strictly surround the source height `s2` to target height `v2` range of the other vote.

```
Definition slashed s n : Prop :=
 slashed_dbl_vote s n ∨ slashed_surround s n.
```

A validator is slashed if it has violated either slashing condition.

We can also define justified links between blocks based on the votes in a state:

```
Definition justified_link s q parent pre new now :=
  q ∈ quorum_1 ∧ (∀ n, n ∈ q → vote_msg s n new now pre) ∧
  now > pre ∧ nth_ancestor (now - pre) parent new.
```

The conjuncts respectively require that set `q` is large enough to justify a link, that every validator in `q` has voted for this link, that the claimed height `now` of the target is greater than the claimed height `pre` of the source, and that `parent` is actually the ancestor the appropriate number of levels above `new`.

This allows us to define justified blocks, starting from the genesis block, and furthermore finalized blocks, as in the Casper paper.

```
Inductive justified : State → Hash → nat → Prop :=
| orig : ∀ s, justified s genesis 0
| follow : ∀ s parent pre q new now,
    justified s parent pre →
    justified_link s q parent pre new now →
    justified s new now.

Definition finalized s q h v child := justified s h v
  ∧ h <~ child ∧ justified_link s q h v child v.+1.
```

4 Accountable Safety

This section describes the Coq formalization and verification of Casper's *accountable safety* property. A major part of the Casper design is ensuring that an attack cannot finalize blocks on both sides of a fork in the chain without the attacker losing a significant amount of money by having their validator deposits slashed. This is formalized as the accountable safety theorem, which says that if two blocks are finalized and neither is an ancestor of the other, then validators having at least $\frac{1}{3}$ of the deposits must have violated the slashing conditions. The difference between slashing violations and actually losing funds is addressed and handled with other countermeasures in the Casper paper [11].

Let `hash_ancestor` be the reflexive-transitive closure of the `hash_parent` relation. Building on the definitions of validator quorums and justification described above, we define a fork in a state as two conflicting finalized blocks (hashes):

```
Definition fork s := ∃ h1 h2 q1 q2 v1 v2 c1 c2,
 finalized s q1 h1 v1 c1 ∧ finalized s q2 h2 v2 c2 ∧
 ~ hash_ancestor h2 h1 ∧ ~ hash_ancestor h1 h2 ∧ h1 ≠ h2.
```

With `slashed s v` capturing that the slashing conditions in Figure 1 are true in state `s` for validator `v`, we define the condition of $\frac{1}{3}$ of validators by weight being slashed:

```
Definition quorum_slashed s :=
 ∃ q, q ∈ quorum_2 ∧ ∀ v, v ∈ q → slashed s v.
```

This allows us to define and prove accountable safety:

```
Theorem accountable_safety : ∀ s, fork s → quorum_slashed s.
```

The proof roughly follows the informal argument [11]. If the two finalized blocks were at the same height then at least $\frac{1}{3}$ weight (or a `quorum_2` set) of validators violated slashing condition I. Otherwise, consider the path of justified links which justifies the higher finalized block. If any justified block along this path has the same height as the other justified block or its justified immediate child then we again have a sufficiently large set of validators violating slashing condition I. Otherwise find the justified link along that path

with a source lower than the other justified block and a target higher than its child. Taking this link along with the justified link between the lower finalized its child we have a set of validators violating slashing condition II.

5 Plausible Liveness

The liveness goal for Casper is that as long as at least $\frac{2}{3}$ of validators are following the protocol and block proposals continue, then further checkpoints can continue to be finalized regardless of the behavior of the less the $\frac{1}{3}$ of misbehaving validators, without needing any of the honest validators to violate a slashing condition and sacrifice their deposit to allow the chain to live.

Our Coq proof is based on the argument given in [11], and also Yoichi Hirai's Isabelle/HOL proof of plausible liveness for an older variant of the Casper protocol [15].

5.1 Context

The proof works with the same model of checkpoint blocks as the accountable safety proof. We are given a set of checkpoint block hashes, a parent relationship giving each hash at most one parent, and a hash of the genesis block. The changing part of the system states is only the set of votes, because we do not model the block proposal process or dynamic validator sets.

Finalization is an essential part of the Casper design for dynamic validator sets, which uses finalized blocks as the boundary of the "epochs" where the validator set can change. Because of this essential role we state and prove liveness in terms of a hash and height of the most recent previously-finalized block, even though we are not yet proving liveness in the presence of dynamic validator sets.

```
Variable epoch_start : Hash.
Variable epoch_height : nat.
Hypothesis epoch_ancestry :
  nth_ancestor epoch_height genesis epoch_start.
```

Here `epoch_start` is the hash of that block, `epoch_height` is its height, and `epoch_ancestry` is a proof that the epoch start block is actually at the claimed height over the genesis block.

To demonstrate a further block can be finalized we only need to consider justification of blocks descending from the start of this epoch. We define justification within the epoch as having a path of super-majority links from the epoch start up to a block.

```
Inductive justified_this_epoch (st:State)
  : Hash → nat → Prop :=
| epoch_justified :
    justified_this_epoch st epoch_start epoch_height
| justfied_link : ∀ s s_h t t_h,
    justified_this_epoch st s s_h →
    hash_ancestor s t →
    supermajority_link st s t s_h t_h →
    justified_this_epoch st t t_h.
```

This definition defines justification inductively, from two cases. A block is justified if it is the epoch start block, or if it has a justified ancestor and a supermajority of validators have voted for a link to this block from that ancestor.

Besides the parameterization over the epoch start block, we do not actually consider the history of any past epochs. In particular, we assume that the well-behaved nodes have not made any votes with sources outside the current epoch. We could work without that assumption, but we would need to repeat part of the accountable safety proof to argue that an unslashed validator's votes from past epochs cannot prevent them from safely making the votes chosen in out proof.

5.2 Theorem

To formalize a statement of plausible liveness, we follow our examples and abstract away from any implementation details of "following the protocol", and simply require that our $\frac{2}{3}$ good validators have not made certain sorts of bad votes. For the conclusion we also ignore the details of how validators decide to make votes, and simply show that there is some set of votes which the good validators could make that would finalize a further block and not violate slashing conditions.

In Hirai's proof for the previous Casper design it was sufficient to simply require that the good validators were unslashed. The current Casper design intentionally removed any slashing conditions that required knowing the state of the chain when the votes were made. We found that *one of these conditions was essential to the proof*, so our definition of good behavior for the current Casper protocol requires that a validator has not made votes with an unjustified source in addition to requiring that a good validator had not been slashed.

We define the condition that a super-majority of validators is good given a set of votes:

```
Definition two_thirds_good (st : state) :=
  ∃ good_validators, good_validators ∈ quorum_1
  ∧ ∀ v, v ∈ good_validators
     → (~ slashed st v ∧ sources_justified st v).
```

The property `sources_justified st v` says the source block of any vote that validator `v` has made in state `st` is justified within the current epoch[1]

```
Definition sources_justified st v :=
  ∀ s t s_h t_h,
   vote_msg st v s t s_h t_h →
   hash_ancestor epoch_start s ∧ justified_this_epoch st s s_h.
```

The other explicit hypothesis of our plausible liveness theorem captures the assumption that block proposals continue. We only need to assume that blocks can be found above the highest justified block, but we may need to find a block at arbitrary height. These definitions capture the properties

[1] This enforces the assumption above that good nodes only have votes with sources within the current epoch

of being the highest justified block (which will be unique by accountable safety), and having descendants at arbitrary heights.

```
Definition blocks_exist_high_over (base : Hash) : Prop :=
  ∀ n, ∃ block, nth_ancestor n base block.
Definition highest_justified st b b_h : Prop :=
  ∀ b' b_h', b_h' >= b_h
  → justified_this_epoch st b' b_h'
  → b' = b ∧ b_h' = b_h.
```

With these ingredients, this is the statement of plausible liveness that we prove:

```
Lemma plausible_liveness :
  ∀ st, two_thirds_good st →
  (∀ b b_h, highest_justified st b b_h
     → blocks_exist_high_over b) →
  ∃ st', unslashed_can_extend st st'
   ∧ no_new_slashed st st'
   ∧ ∃ (new_finalized new_final_child:Hash) new_height,
       justified_this_epoch st' new_finalized new_height
        ∧ epoch_height < new_height
        ∧ new_finalized <~ new_final_child
        ∧ supermajority_link st' new_finalized new_final_child
                                   new_height new_height.+1.
```

The desired conclusion of the theorem is expressed by stating that there exists a set of votes `st'`, such that any votes which were not already in `st` were made by unslashed validators, that no validators that were unslashed in `st` are slashed in `st'`, and that there is a block `new_finalized` which is finalized under votes `st'` and is higher than the previous finalized block.

The proof proceeds as outlined in [11]: We take the highest justified checkpoint J which is a descendant of the most recently finalized block, consider the maximum height of the target of any existing vote, and use the assumption that block proposals continue to find a descendant A of J at a greater height than that, and an immediate child B of A. Then all good validators will vote for the links J⟶A and A⟶B. This is easily shown to finalize A and require votes only from good validators. It remains only to prove that this does not violate any slashing condition. Recall the definitions of the slashing conditions from Figure 1.

- Condition I is not violated because a good validator's two new votes have targets at different heights, and any previous votes have a target at a lower height than A, by choice of A.
- Condition II is not violated because the two new votes of good validators do not nest, a previous vote cannot have a range surrounding a new vote because all previous votes have targets below A by choice of A, and a previous vote cannot nest within the vote for J⟶A because it would have a source above J, we assume existing votes from good validators have justified sources, and J is the highest justified block.

5.3 Unslashed is not enough

One might wonder if a different voting strategy could prove plausible liveness while only needing to require that the good validators be unslashed. To show that there cannot be such a proof, we give an example where all validators are unslashed but it is impossible to justify any further blocks (let alone finalize a further block). Every validator in this example has made one vote with an unjustified source.

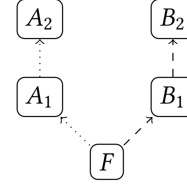

Figure 2. Unslashed Progress Counterexample

The example is shown in Figure 2. F is the most recent finalized block, A_1 and B_1 immediate children of F, and A_2 and B_2 immediate children (respectively) of A_1 and A_2. Half of the validators have voted for the edges $F \longrightarrow A_1$ and $A_1 \longrightarrow A_2$ (dotted arrows) and the other half of the validators have voted for $F \longrightarrow B_1$ and $B_1 \longrightarrow B_2$ (dashed arrows). It is impossible to make a super-majority link from F to any other block without some validators being slashed. A new edge from F to a block one or two levels above F would violate slashing Condition I, because every validator already has votes with a target at that level. An edge from F to a block at any higher level would violate Condition II, with the inner vote being that validator's vote for $A_1 \longrightarrow A_2$ or $B_1 \longrightarrow B_2$.

6 Instantiating Hypotheses and Parameters

The definitions and proven properties in the abstract models of Casper are quite far from those in the Casper paper [11], partly due to proof engineering concerns described above. To bring our results closer to the paper, and thus to actual implementations, we instantiate the key abstractions. In particular, we assume an arbitrary function which maps validators to their deposits:

```
Variable deposit : Validator → nat.
```

Using the support for big operators in Mathematical Components [10], we then define the total deposits of all validators:

```
Definition deposits := \sum_(v : Validator) (deposit v).
```

We also define a function from a natural number `n` to the set of sets of validators which have total deposit at least `n`:

```
Definition gdset n := [set x in powerset [set: Validator] |
 \sum_(v in x) (deposit v) >= n].
```

Writing `m %/ d` for the quotient of the Euclidean division of `m` by `d`, we define two sets:

```
Definition deposit_bot := gdset (deposits %/ 3).+1.
Definition deposit_top := gdset ((2 * deposits) %/ 3).+1.
```

which allows us to prove:

```
Lemma deposit_validator_intersection : ∀ q1 q2,
 q1 ∈ deposit_top → q2 ∈ deposit_top →
 ∃ q3, q3 ∈ deposit_bot ∧ q3 \subset q1 ∧ q3 \subset q2.
```

using the arithmetical lemma

```
Lemma thirds : ∀ n, (n %/ 3).+1 + n <= 2 * (2 * n %/ 3).+1.
```

This intersection lemma shows that we can instantiate the abstract `quorum_1` and `quorum_2` used in our proofs with specific sets `deposit_top` and `deposit_bot`.

6.1 Block Trees and Transition System

Working from the other end of the abstraction spectrum, we define functions and datatypes in Coq resembling those in the beacon chain implementation [1], to be used when defining global system state and node-local behavior. In particular, following Toychain, we abstracted a block to a Coq record type containing, most notably, a hash of the previous block and a collection of attestations by validators:

```
Record Attestation := mkAR { distance_src : nat;
 attester : Validator; }.

Record Block := mkB { parent_hash : Hash;
  attestations : seq Attestation;
  distance : nat; }.
```

Using notions from the FCSL PCM library [5], we then define block forests as finite maps from hashes to blocks:

```
Definition Blockforest := union_map Hash Block.
```

This allows us to instantiate a suitably concrete parent relation over hashes:

```
Definition hash_parent_bf (bf : Blockforest) : rel Hash :=
[rel x y | (x ∈ dom bf) && (y ∈ dom bf) &&
 if find y bf is Some b then parent_hash b == x else false].
```

for which we can instantiate the desired hypothesis:

```
Lemma hash_parent_bf_eq : ∀ f h1 h2 h3,
 hash_parent_bf f h2 h1 → hash_parent_bf f h3 h1 → h2 = h3.
```

We can now consider a block forest as a concrete global state, and define a mapping from from block forests to a `vote_msg` function:

```
Definition vote_msg_bf bf v h distance distance_src : bool :=
 if find h bf is Some b then
  (distance b == distance) &&
  ((mkAR distance_src v) ∈ attestations b)
 else false.
```

The initial state of the system is simply the finite map taking the genesis block hash to the genesis block:

```
mkSt (vote_msg_bf (#GenesisBlock \\→ GenesisBlock))
```

Finally, we define a transition system that given a block `b`, updates the global state `bf` with `b`:

```
mkSt (vote_msg_bf (#b \\→ b \+ bf))
```

Accountable safety holds for every state in this transition system, and we believe it can serve as a specification for the Casper beacon chain implementation [1], and is closer to the level of detail needed generate a correct-by-construction implementation from the specification.

6.2 Beacon Node

The state of the beacon chain [1] is split into two main parts, an active state and a crystallized state. The active state is updated every block, while the crystallized state is only updated according to a cycle length parameter of the protocol, which is set to 64 slots in the beacon chain implementation [1].

In our Coq formulation, we define the state of a node in the beacon chain as a record containing the blockforest and this beacon chain state. It also includes other parameters of the system, such as the cycle length mentioned above.

```
Record State {Hash : ordType} :=
  Node {
    id : NodeId;
    peers : peers_t;
    blocks : Blockforest;
    cstate : @CrystallizedState Hash;
    astate : @ActiveState Hash;
    (* set to 1024 shards, see beacon chain implementation *)
    shardCount : nat;
    (* set to 64 slots, see beacon chain implementation *)
    cycleLength : nat;
  }.
```

When a block arrives on a node, the crystallized and active states are updated via the state transition function specified in [1]. The update of the crystallized state is where most of the Casper functionality occurs. Should the slot number of the block be at least 64 past the last recalculation slot, the crystallized state is updated by applying the rewards and penalties for the validators, as well as managing the justification and finalization of blocks. For example, the Coq code for the state update has functions such as `applyRewardsAndPenalties` below, which calculates and updates the balances for all active validators given the current state and block.

```
Definition applyRewardsAndPenalties
          (crystallizedState :
            @CrystallizedState [ordType of Hash])
          (activeState : @ActiveState [ordType of Hash])
          (blk : block)
          (cycleLength : nat) :=
  (* ... omitted ... *)
```

In addition to updating the crystallized and active states, the arrived block is also appended to the block forest, and the new state is returned, along with a message from the node that received the block to all of its peers.

```
Definition procInt (st : @State [ordType of Hash])
          (tr : InternalTransition) (ts : Timestamp) :=
```

```
let: Node n prs bf cst ast cl := st in
match tr with
| BlockT b ⇒
  let: parentBlock := get_block bf (parent_hash b) in
  let: (crystallizedState, activeState) :=
    computeStateTransition cst ast parentBlock b cl in
  let: newBf := bfExtend bf b in
  pair (Node n prs newBf crystallizedState activeState cl)
      (emitBroadcast n prs (BlockMsg b))
end.
```

These definitions tighten the correspondence between our Coq code and the Beacon chain implementation in [1]. Along with our abstract safety and liveness proofs, these definitions will allow us to perform verification at this low level, providing as much assurance as possible for the correctness of the Casper protocol.

7 Conclusion

We presented a formalization of Casper in Coq, and proofs of accountable safety and plausible liveness. These proofs clarified the assumptions needed for the properties stated in the Casper paper [11], especially what assumptions on validator behavior are needed for plausible liveness. Our formalization work paves the way for transferring accountable safety and plausible liveness from Hirai's abstract models to hold for steps over the Toychain global state according to a step relation and node state definition matching those in the Ethereum reference beacon chain node implementation [1].

References

[1] 2018. Beacon Chain. https://github.com/ethereum/beacon_chain/
[2] 2018. Bitoychain. https://github.com/palmskog/bitoychain
[3] 2018. CoqHammer. https://github.com/lukaszcz/coqhammer
[4] 2018. Mathematical Components Project. https://math-comp.github.io/math-comp/
[5] 2018. PCM library. https://github.com/imdea-software/fcsl-pcm
[6] 2018. Toychain. https://github.com/certichain/toychain
[7] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017).
[8] Clemens Ballarin. 2014. Locales: A Module System for Mathematical Theories. *Journal of Automated Reasoning* 52, 2 (01 Feb 2014), 123–153.
[9] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. 2016. Cryptocurrencies Without Proof of Work. In *Financial Cryptography and Data Security*, Jeremy Clark, Sarah Meiklejohn, Peter Y.A. Ryan, Dan Wallach, Michael Brenner, and Kurt Rohloff (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 142–157.
[10] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. 2008. Canonical Big Operators. In *Theorem Proving in Higher Order Logics*. 86–101. https://doi.org/10.1007/978-3-540-71067-7_11
[11] Vitalik Buterin and Virgil Griffith. 2017. Casper the Friendly Finality Gadget. *CoRR* abs/1710.09437 (2017).
[12] Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *Journal of Automated Reasoning* 61, 1 (2018), 423–453.
[13] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. 2009. Packaging Mathematical Structures. In *Theorem Proving in Higher Order Logics*. 327–342.
[14] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. 2013. A Machine-Checked Proof of the Odd Order Theorem. In *Interactive Theorem Proving*. 163–179.
[15] Yoichi Hirai. 2018. A repository for PoS related formal methods. https://github.com/palmskog/pos
[16] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. http://www.bitcoin.org/bitcoin.pdf
[17] George Pîrlea and Ilya Sergey. 2018. Mechanising blockchain consensus. In *Certified Programs and Proofs*. 78–90.
[18] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. (2014). http://gavwood.com/paper.pdf