Formalizing Correct-by-Construction Casper in Coq

Elaine Li*, Traian Şerbănuţă*, Denisa Diaconescu*, Vlad Zamfir†, and Grigore Roşu*‡
*Runtime Verification
†Ethereum Research
‡University of Illinois at Urbana-Champaign
*{elaine.li, traian.serbanuta, denisa.diaconescu}@runtimeverification.com
†vlzmfr@gmail.com
‡grosu@illinois.edu

***Abstract*—Correct-by-Construction Casper (CBC Casper) is an Ethereum candidate consensus protocol undergoing active design and development. We present a formalization of CBC Casper using the Coq proof assistant that includes a model of the consensus protocol and proofs of safety and non-triviality protocol properties. We leverage Coq's type classes to model CBC Casper at various levels of abstraction. In doing so, we 1) illuminate the assumptions that each protocol property depends on, and 2) reformulate the protocol in general, mathematical terms. We highlight two advantages of our approach: 1) from a proof engineering perspective, it enables a clean separation of concerns between theory and implementation; 2) from a protocol engineering perspective, it provides a rigorous, foundational understanding of the protocol conducive to finding and proving stronger properties. We detail one such new property: strong non-triviality.**

***Index Terms*—consensus protocols, blockchain, Ethereum, Coq, formal verification**

I. Correct-by-Construction Casper

Correct-by-Construction (CBC) Casper [1] is a partial specification for a family of consensus protocols. Briefly, CBC Casper is instantiated with five framework parameters to define concrete protocols: 1) participating nodes, or validators, 2) an assignment of positive real numbers to each validator, or validator weights, 3) a fault tolerance threshold, or the sum of permissible faulty validator weights, 4) consensus values, and 5) an estimator function that gives acceptable consensus values for each protocol state. These five framework parameters in turn define protocol states and messages. CBC Casper protocols are Byzantine fault-tolerant with respect to *equivocation*: nodes sending two *equivocating* messages that could not have been produced by a single execution of the protocol. CBC Casper protocols can be instantiated to define concrete protocols, such as Casper the Friendly Finality Gadget [2], that share the same proofs of desired protocol properties: safety and non-triviality. Safety states that with not too many equivocating nodes, all participating nodes decide on the same consensus value. Non-triviality states that it is always possible for participating nodes to make inconsistent decisions on consensus values.

978-1-7281-6680-3/20/$31.00 © 2020 IEEE

We contribute to CBC Casper's design and "correct-by-construction" namesake by contributing the following:

- a formal specification of CBC Casper, instantiated with a full node and light node version of the protocol,
- an abstraction hierarchy refining CBC Casper in terms of i) partial order, ii) a partial order with non-local confluence, and iii) a global state transition system,
- formal proofs of safety and weak non-triviality,
- a novel proof of strong non-triviality.

Our Coq development is publicly available at:

https://github.com/runtimeverification/casper-cbc-proofs

II. Coq Formalization

We formalize CBC Casper at three different levels of abstraction to illuminate the assumptions required to prove safety and non-triviality properties. Our first abstraction removes the protocol-specific features of messages and validator weights, and is sufficient to show safety, namely that nodes "decide" on the same consensus value given not too many Byzantine nodes. We further abstract this as a bottomed partial order, and show that it remains sufficient to show safety. In order to obtain non-triviality properties, we add non-local confluence to the partial order, to form our third abstraction. We use Coq's type classes to define our abstraction hierarchy, leveraging the relationship between a typeclass and its instances to reflect the relationship between CBC Casper and its protocol family members. We summarize the relationship between the three abstractions and the protocol properties they give rise to in the diagram below. Boxes represent type classes, and ovals represent properties. Solid arrows between type classes represent "instantiate", and dotted arrows between type classes and properties represent "derive". `FullNode` and `LightNode` are two concrete instances of CBC Casper, with different framework parameters and respectively, different protocol states and messages. All arrows are transitive, and the proof of this diagram can be found in `protocol.v` of our development.

Next, we detail two benefits of our formalization approach, from a proof engineering and protocol engineering perspective respectively.

Authorized licensed use limited to: New York University. Downloaded on December 14,2023 at 20:18:40 UTC from IEEE Xplore. Restrictions apply.

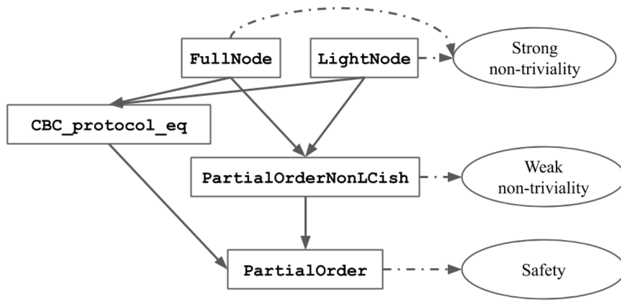

Fig. 1. Type class hierarchy

A. Mutual recursion

One central feature of CBC Casper is its mutually recursive definition of protocol states and messages. CBC Casper protocol states are defined as sets of messages, where each message is a triple (c, v, j):

- c is a (proposed) consensus value;
- v identifies the message sender;
- j, the justification, is the *protocol state* seen by the sender at the time of message sending.

This definition presents us with several mechanization difficulties. Constructing the mutually recursive definition requires first constructing a generic state type and in turn defining protocol states as inductive predicates on these states. Additionally, we need to define a notion of state equality. Defining an equivalence relation which disregards message ordering is possible but non-trivial: the definition would itself need to be recursive, as it requires an equivalence on messages, which in turn is defined in terms of the same state equivalence. To circumvent the hindrance of working with mutually recursive state equality, we instead use canonical, sorted representatives for states, on which we can then express state equality as *syntactic* equality. These difficulties add complexity to proving properties about protocol states and messages defined in this way.

Our formalization approach reduces this complexity overhead through the observation that safety and non-triviality properties do not depend on protocol states and messages being mutually recursive: a weaker property about states and reachability is sufficient. Specifically, we can represent adding messages to states as a reflexive, transitive reachability relation on states, thus abstracting away the mechanics of how messages are added, and how they relate to the protocol states they are added to.

B. Strong non-triviality

Another observation in our efforts to generalize the protocol is that non-triviality, which states the existence of a protocol state that can reach two future protocol states which do not share a common future, mirrors the notion of non-local confluence in abstract rewriting systems. On the definition of protocol states as sets of messages, future (or reachability) is defined as set inclusion, and the common future of two protocol states must contain the union of both their messages. Naturally, the question of whether the existential quantifier could be strengthened to a universal quantifier arose: is every state a non-locally confluent state? We found a positive answer to this question, and further found that the reachability relation could be strengthened to atomic reachability. We call the stronger property strong non-triviality, which states that for every protocol state `s1`, there exists a protocol state `s2` reachable from `s1` in one step and a protocol state `s3` reachable from `s1` in an arbitrary number of steps, such that `s1` and `s3` share a common future, but `s2` and `s3` do not. Further, we give a constructive method for finding `s2` and `s3` for any protocol state `s1`. In the following section we sketch the constructive proof of this theorem.

1) Pivotal senders: By definition, protocol states are only valid iff they do not admit too many equivocating nodes, or too much "fault weight". Since fault weights are discrete, participating nodes can continue to equivocate, increasing the present state's fault weight, until a state that is on the verge of exceeding the fault weight is reached, i.e. a state that cannot admit any more equivocation. These states are non-locally confluent: if one more node sends a pair of equivocating messages, the union of the resulting two protocol states would exceed the fault weight threshold. We call such states *heavy* states, and we call the additional equivocating node a *pivotal* node. While a heavy state need not have a unique pivotal node; for the purposes of proving strong non-triviality it suffices to show the existence of one. We prove that every state has such a pivotal sender.

2) No common futures: Given an arbitrary protocol state `s1`, the property above allows us to obtain a set of nodes `vs` and a single pivotal node `v`. We first construct a pair of equivocating messages for `v`, and send one half from `s1` to obtain `s1'` and `s2` respectively, such that `v` is equivocating in the union of `s1'` and `s2`. We can then incrementally construct a future state of `s1'` by iterating through `vs` and adding pairs of equivocating messages from each node to obtain `s3`. We must now show that 1) `s1` and `s3` have a common future, but 2) `s2` and `s3` do not. 1) is trivial by definition. 2) proceeds via contradiction: assume we have a protocol state `s` that is a future state of both `s2` and `s3`, and therefore contains all messages in both `s2` and `s3`, respectively all equivocating nodes in both `s2` and `s3`, respectively all fault weight in both `s2` and `s3`. It must then contain the fault weight of all the nodes in `vs` in addition to the fault weight of `v`. However, by definition of `v`'s pivotal property, the combined fault weights of `vs` and `v` exceed the fault tolerance threshold, therefore `s` cannot be a valid protocol state. We find a contradiction.

These two key ingredients respectively identifies a pivotal node, and describes how to construct equivocations for the remainder set of nodes and reach a future state that is heavier by exactly their weights. The extent to which these ingredients can be generalized across different concrete protocols is a topic of ongoing investigation.

Authorized licensed use limited to: New York University. Downloaded on December 14,2023 at 20:18:40 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] V. Zamfir, N. Rush, A. Asgaonkar, and G. Piliouras, “Introducing the ‘Minimal CBC Casper’ Family of Consensus Protocols,” https://github.com/cbc-casper/cbc-casper-paper.

[2] V. Buterin and V. Griffith, “Casper the Friendly Finality Gadget,” https://arxiv.org/abs/1710.09437.

[3] The Coq Development Team, “The Coq Proof Assistant,” https://coq.inria.fr/.

Authorized licensed use limited to: New York University. Downloaded on December 14,2023 at 20:18:40 UTC from IEEE Xplore. Restrictions apply.