FMBC 2024
Extended Abstracts of Lightning Talks

Bruno Bernardo Diego Marmsoler

April 7, 2024

Towards Formal Verification of DAG-Based Blockchain Consensus Protocols

Nathalie Bertrand
Univ Rennes, Inria, CNRS, IRISA

Pranav Ghorpade
The University of Sydney

Sasha Rubin
The University of Sydney

Bernhard Scholz
Fantom Research

Pavle Subotić
Fantom Research

Abstract

There is a trend in blockchains to switch to DAG-based consensus protocols to decrease their energy footprint and improve security. A DAG-based consensus protocol orders transactions for delivering blocks, and relies on built-in fault tolerance communications via Byzantine Atomic Broadcasts. The ubiquity and strategic importance of blockchains call for formal proof of their correctness. We formalize the DAG-based consensus protocol called DAG-Rider in TLA+ and prove its safety properties with the TLA+ proof system. The formalization requires a refinement approach for modelling the consensus. In an abstracted model, we first show the safety of DAG-based consensus on leaders and then further refine the specification to encompass all messages for all processes. The specification consists of 683 lines and the proof system verifies 1922 obligations in about 5 minutes.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Proof theory; Theory of computation → Distributed algorithms

Keywords and phrases Formal verification, Consensus, Blockchain, Theorem Proving, TLA+

Consensus and DAG-based protocols. Consensus is a fundamental problem in distributed computing. It aims to coordinate processes so that they agree on some value(s). Consensus algorithms have recently become an important topic in "proof-of-stake" blockchains that collaboratively build an order for submitted transactions. Of particular interest are consensus algorithms that assume little about the environment, namely, asynchronous communications with malicious processes, namely Byzantine Fault Tolerant (BFT) [15].

Early blockchain consensus protocols assume degrees of synchrony in the environment to ensure safety and liveness [16, 2, 11, 8]. Recently, a family of probabilistic asynchronous consensus protocols have been introduced that are based on Directed Acyclic Graphs (DAG-based protocols) [10, 1, 6]. These protocols report high performance while guaranteeing BFT, utilize processes fairly, and exhibit low communication complexity. Several leading blockchains thus have adopted DAG-based protocols as their main consensus mechanism [4, 5, 9].

DAG-Rider [10] is such a DAG-based protocol and has two main components: (1) a communication layer and (2) an offline ordering layer. The communication layer asynchronously exchanges messages between processes in rounds using *reliable broadcast*. Messages contain transaction proposals and metadata forming a DAG for each node. For a process, the DAG provides a local view of the order of blocks with respect to happened-before relation [12]. Due to the asynchronous nature of the network, processes do not necessarily have the same local DAGs at any point in time. However they are guaranteed to have same DAGs eventually. The ordering layer selects anchor points, guaranteeing consistent selection across all the processes. This allows the DAGs to be locally totally ordered while guaranteeing that all the processes agree on the same total order of messages.

Formal verification of DAG-based consensus. Blockchains provide mission-critical financial services and hence require rigour to show correctness. The verification challenges arise from the large number of possible interleaving in an asynchronous environment, the behaviours of Byzantine processes, and perhaps even more importantly the fact that correctness should hold for any number of participating processes.

We report here on our TLA+ [13] specification and proof –both publicly available at [7]– for a DAG-based consensus protocol using the TLA+ Proof System (TLA-PS) [3].

Procedural code is commonly modeled in TLA+ by a discrete transition system whose traces correspond to possible executions of the code. The naïve translation from the pseudo-code (by setting every variable from the protocol, including a variable for each process's current line number, to be a variable in the specification) into a TLA+ specification is not viable. While direct, this model is very fine-grained and renders the proofs extremely tedious.

To obtain a more succinct and tractable model, we employ several abstraction techniques: they remove unnecessary details and produce a specification that is amenable to proofs. First, we employ a *procedural abstraction* that ignores all states that are internal to a procedure and only represents the input/output behaviour of each procedure in the DAG-Rider protocol. For instance, in the *wave_ready* procedure of [10], the relevant variables are *decidedWave*, *deliveredVertices*, *leadersStack*, but not the loop variable w' or the auxiliary variable v'. Second, because we focus on safety properties, we remove component features that are only required for liveness and have no impact on the safety proof. For instance, random coin tosses can be replaced with deterministic ones. Third, we use memoization to efficiently compute the values taken by recursive functions, by introducing a fresh state variable that stores the needed information to evaluate recursive functions in a single step. Finally, we separate the concerns and break the safety property into two, namely (1) consistent communication and (2) consistent leader election. For (1) we model the DAG-construction and show that the causal histories agree for a same vertex in the DAG of two different processes [1]. For (2), we model the consensus protocol and prove that the same leaders are elected and in the same order. To obtain a complete yet simple model of the consensus protocol, we observe that it only needs reachability information associated with wave leader vertices to commit leaders and, therefore, abstract the content of DAG into the so-called *leaderReachablity* record. We combine consensus protocol specifications in DAG construction specifications to obtain one of the DAG-Rider protocols. This abstraction is not only interesting for DAG-Rider but could be helpful to generalize to other DAG-based protocols.

Given our faithful specification of DAG-Rider in TLA+, we prove its expected safety properties by identifying invariants and proving them within TLA-PS. When using TLA-PS and similar proof systems, the most challenging task is to come up with relevant inductive invariants (that hold initially and are preserved when taking transitions), see for instance [14]. For DAG-Rider, to prove the consistency of communication during the DAG construction we identified 6 new invariants, and to prove the consistency of leader election we identified 10 new invariants. We prove each one of the invariants hierarchically by induction.

Table 1 provides some metrics on our experiments, showing quite reasonable performances in terms of verification time. Most importantly, due to the modularity of our specification, we argue the effort to adapt proofs is minimal when making small changes to the specification.

Conclusion. Our work on DAG-Rider is an important and promising step towards a general library for specifying and verifying DAG-based consensus protocols. Beyond the

[1] The non equivocation of blocks is guaranteed by reliable broadcast abstraction

Table 1 Summary of experiments. An obligation is a condition that TLA-PS checks. The time to check is on a 2.10 GHz CPU with 8 GB of memory, running Windows 11 and TLA-PS v1.4.5.

Metric	DAG-Constr. Spec.	Consensus Spec.	DAG-Rider Spec.
Size of spec. (# loc)	460	250	710
Size of proof (# loc)	521	782	1303
Max level of proof tree nodes	10	9	10
Max degree of proof tree nodes	7	7	7
# obligations in TLA-PS	722	1205	1927
Time to check by TLA-PS (s)	224	87	311

specification of DAG-Rider, our specification reveals interesting insights into developing a modular and efficient TLA+ specification that is amenable to proofs in TLA-PS.

References

1 Leemon Baird and Atul Luykx. The hashgraph protocol: Efficient asynchronous BFT for high-throughput distributed ledgers. In *Proceedings of COINS 2020*, pages 1–7. IEEE, 2020. doi:10.1109/COINS49042.2020.9191430.

2 Vitalik Buterin. Ethereum white paper: A next generation smart contract & decentralized application platform, 2013. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

3 Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA + proofs. In *Proceedings of FM 2012*, volume 7436 of *Lecture Notes in Computer Science*, pages 147–154. Springer, 2012. doi:10.1007/978-3-642-32759-9_14.

4 Aptos Foundation. Understanding Aptos: A comprehensive overview, 2024. URL: https://messari.io/report/understanding-aptos-a-comprehensive-overview.

5 Fantom Foundation. Lachesis aBFT, 2024. URL: https://docs.fantom.foundation/technology/lachesis-abft.

6 Adam Gagol, Damian Lesniak, Damian Straszak, and Michal Swietek. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. In *Proceedings of AFT 2019*, pages 214–228. ACM, 2019. doi:10.1145/3318041.3355467.

7 Pranav Ghorpade. TLA+ specification and Proofs for DAG-Rider. https://github.com/pranavg5526/DAG-Rider, 2024. [Online; accessed 1-Feb-2024].

8 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of SOSP 2017*, pages 51–68. ACM, 2017. doi:10.1145/3132747.3132757.

9 Hedra. Streamlining consensus, 2024. URL: https://hedera.com/blog/streamlining-consensus-throughput-and-lower-latency-with-about-half-the-events.

10 Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In *Proceedings of PODC 2021*, pages 165–175. ACM, 2021. doi:10.1145/3465084.3467905.

11 Jae Kwon. Tendermint: Consensus without mining. https://tendermint.com/docs/tendermint.pdf, 2014.

12 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978. doi:10.1145/359545.359563.

13 Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. URL: http://research.microsoft.com/users/lamport/tla/book.html.

14 Leslie Lamport. Teaching concurrency, 2009. URL: https://lamport.azurewebsites.net/pubs/teaching-concurrency.pdf.

15 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, jul 1982. doi:10.1145/357172.357176.

16 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

Secure Smart Contracts with Isabelle/Solidity

Diego Marmsoler
University of Exeter, UK

Abstract

Smart contracts are programs stored on the blockchain. They are usually developed in a high-level programming language, the most popular of which being Solidity. Smart contracts are often used to automate financial transactions and thus, vulnerabilities in smart contracts can result in high financial losses. Therefore, it is important to guarantee their correctness which is best done using verification.

To this end, we developed Isabelle/Solidity, a deep embedding of Solidity in Isabelle. In the following, we describe our work on Isabelle/Solidity. We first describe the supported subset of the language and how it is evaluated. Then, we take a brief look at various applications of Isabelle/Solidity. Finally, we discuss ongoing and future work on Isabelle/Solidity.

2012 ACM Subject Classification Software and its engineering → Software verification; Software and its engineering → Formal software verification; Theory of computation → Logic and verification

Keywords and phrases Smart Contracts, Blockchain, Theorem Proving, Isabelle

Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

Category Extended Abstract

1 Introduction

Blockchain [11] is a novel technology to store data in a decentralised way, providing *transparency*, *security* and *trust*. Although the technology was originally invented to enable cryptocurrencies, it quickly found applications in several other domains, such as *finance* [5], *healthcare* [1], *land management* [2], and even *identity management* [14].

One important innovation which comes with blockchain are so-called *smart contracts*. These are digital contracts which are automatically executed once certain conditions are met and which are used to automate transactions on the blockchain. For instance, a payment for an item might be released instantly once the buyer and seller have met all specified parameters for a deal. Every day, hundreds of thousands of new contracts are deployed managing millions of dollars' worth of transactions [13].

Technically, a smart contract is code which is deployed to a blockchain and which can be executed by sending special transactions to it. Thus, as for every computer program, *smart contracts may contain bugs which can be exploited.* However, since smart contracts are often used to automate financial transactions, such exploits may result in huge economic losses. In general, it is estimated that since 2019, more than $5B was stolen due to vulnerabilities in smart contracts [3].

The high impact of vulnerabilities in smart contracts together with the fact that once deployed to the blockchain, they cannot be updated or removed easily, makes it important to "get them right" before they are deployed. In the following, we describe our work to address this problem.

2 Isabelle/Solidity

Smart contracts are usually developed in a high-level programming language, the most popular of which is *Solidity* [4]. Solidity is based on the Ethereum Virtual Machine (EVM) and thus it works on all EVM-based smart contract platforms, such as Ethereum, Avalanche, Moonbeam, Polygon, BSC, and more. As of today, 85% of all smart contracts are developed using Solidity [6].

Isabelle/Solidity is a deep embedding of Solidity in Isabelle/HOL [12]. A first version of the semantics is described in [7]. Since then the language was constantly extended and the current version is available in the archive of formal proofs [9]. It supports the following features of Solidity:

- Fixed-size integer types of various lengths and corresponding arithmetic.
- Domain-specific primitives to support transferring of funds or query balances.
- Different types of stores, such as storage, memory, calldata, and stack.
- Complex data types, such as hash-maps and arrays.
- Assignments with different semantics, depending on data types.
- An extendable gas model.
- External and internal method declarations and the ability to transfer funds with external method calls.
- Declaration of fallback methods which are executed with monetary transfers.
- Dynamic creation of new contract instances.

To ensure that the semantics complies with the reference implementation the semantics comes with a grammar based fuzzying framework to allow for automatic testing of compliance. The testing framework automatically generates random Solidity programs and corresponding states. The semantics is then used as an oracle to predict the modifications to the state which are then inserted as assertions to the original contract. Then, the contract is deployed to the Blockchain and executed. If the assertions are violated a deviation from the semantics is reported. The framework is described in more detail in [8] and it is used to guarantee compliance of the semantics to Solidity.

3 Applications

Being a deep embedding, Isabelle/Solidity can be used not only to reason about individual contracts but also about tools and calculi for Solidity. Thus, so far, Isabelle/Solidity was used for the following use cases:

- The verification of the correctness of a Gas-optimization tool for Solidity.
- The verification of the soundness of a methodology to verify invariants for Solidity contracts.
- The verification of the functional correctness of a simple Solidity token.

In particular, in [7] we describe the implementation of a constant folding tool for Solidity. The tool replaces constant expressions with their corresponding evaluation. For example, the expression `int16(250) + uint8(500)` would be replaced with the expression `int16(494)` which costs 12 Gas less. However, since the optimizer is modifying the source code of a program it is important to guarantee that it does not modify its semantics. Thus, we implemented the optimizer in Isabelle and verified it using Isabelle/Solidity.

In another example, Isabelle/Solidity was used to verify the correctness of a calculus for Solidity which is described in [10]. In essence, the calculus allows for the verification of invariants for Solidity contracts. To this end it requires a user to specify the following components.

- An invariant: A predicate over the contract's member variables and the contract's balance.
- Preconditions for internal methods: Predicates over the method's formal parameters, the contract's member variables, and the contract's balance.

- Postconditions for internal methods: Predicates over the contract's member variables and the contract's balance.
- A precondition and postcondition for the contract's fallback method: A predicate over the contract's member variables and the contract's balance.

The method then requires a user to verify postconditions for internal methods and the fallback method as well as that external methods preserve the invariant. To ensure that the proof obligations indeed guarantee that the invariant is not violated we formalized the calculus in Isabelle and verified its soundness using Isabelle/Solidity.

Finally, in [7, 10], we used Isabelle/Solidity to verify a basic version of an Ethereum token. In particular, we formalized the contract in Isabelle and verified that the sum of all balances corresponds to the internal balance of the token contract.

4 Ongoing and Future Work

We are currently working on several topics related to Isabelle/Solidity. First, we are constantly adding new features to our semantics as for example inheritance between contracts. In addition, we are currently verifying type safety of the language. To simplify the verification of individual contracts we are currently also working on an alternative, shallow embedding of Solidity in Isabelle. Finally, we are working on a verified compiler for Isabelle/Solidity to EVM bytecode.

References

1 Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. Medrec: Using blockchain for medical data access and permission management. In *2016 2nd international conference on open and big data (OBD)*, pages 25–30. IEEE, 2016.

2 Gertrude Chavez-Dreyfuss. Sweden tests blockchain technology for land registry, 2016.

3 CipherTrace. Cryptocurrency crime and anti-money laundering report. Technical report, 2021.

4 Ethereum. https://docs.soliditylang.org/.

5 Jemima Kelly. Banks adopting blockchain 'dramatically faster' than expected: IBM, 2016.

6 Defi Llama. Tvl breakdown by smart contract language, 2022.

7 Diego Marmsoler and Achim D. Brucker. A denotational semantics of solidity in isabelle/hol. In Radu Calinescu and Corina S. Păsăreanu, editors, *Software Engineering and Formal Methods*, pages 403–422, Cham, 2021. Springer International Publishing.

8 Diego Marmsoler and Achim D. Brucker. Conformance testing of formal semantics using grammar-based fuzzing. In Laura Kovács and Karl Meinke, editors, *Tests and Proofs*, pages 106–125, Cham, 2022. Springer International Publishing.

9 Diego Marmsoler and Achim D. Brucker. Isabelle/solidity: A deep embedding of solidity in isabelle/hol. *Archive of Formal Proofs*, July 2022. `https://isa-afp.org/entries/Solidity.html`, Formal proof development.

10 Diego Marmsoler and Billy Thornton. Sscalc: A calculus for solidity smart contracts. In Carla Ferreira and Tim A. C. Willemse, editors, *Software Engineering and Formal Methods*, pages 184–204, Cham, 2023. Springer Nature Switzerland.

11 Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008.

12 Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. 2002.

13 YCharts.com. Ethereum transactions per day, 2022.

14 Bryan Yurcan. How blockchain fits into the future of digital identity, 2016.