

# 3rd International Workshop on Formal Methods for Blockchains

FMBC 2021, July 18-19, 2021, Los Angeles, California, USA  
(Virtual Conference)

Edited by

Bruno Bernardo

Diego Marmosler



*Editors*

**Bruno Bernardo**

Nomadic Labs, Paris, France  
bruno@nomadic-labs.com

**Diego Marmsoler** 

University of Exeter, UK  
D.Marmsoler@exeter.ac.uk

*ACM Classification 2012*

Security and privacy → Logic and verification; Software and its engineering → Formal software verification;  
Security and privacy → Distributed systems security; Computer systems organization → Peer-to-peer architectures

**ISBN 978-3-95977-209-9**

*Published online and open access by*

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <https://www.dagstuhl.de/dagpub/978-3-95977-209-9>.

*Publication date*

November, 2021

*Bibliographic information published by the Deutsche Nationalbibliothek*

The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at <https://portal.dnb.de>.

*License*

This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0):  
<https://creativecommons.org/licenses/by/4.0/legalcode>.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/OASlcs.FMBC.2021.0

ISBN 978-3-95977-209-9

ISSN 1868-8969

<https://www.dagstuhl.de/oasics>

## OASlcs – OpenAccess Series in Informatics

OASlcs is a series of high-quality conference proceedings across all fields in informatics. OASlcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

### *Editorial Board*

- Daniel Cremers (TU München, Germany)
- Barbara Hammer (Universität Bielefeld, Germany)
- Marc Langheinrich (Università della Svizzera Italiana – Lugano, Switzerland)
- Dorothea Wagner (*Editor-in-Chief*, Karlsruher Institut für Technologie, Germany)

**ISSN 1868-8969**

**<https://www.dagstuhl.de/oasics>**



## ■ Contents

Preface	
<i>Bruno Bernardo and Diego Marmsoler</i> .....	0:vii
Program Committee	
.....	0:ix
Supporting Reviewers	
.....	0:xi

### Regular Papers

Towards Verified Price Oracles for Decentralized Exchange Protocols	
<i>Kinnari Dave, Vilhelm Sjöberg, and Xinyuan Sun</i> .....	1:1–1:14
Money Grows on (Proof-)Trees: The Formal FA1.2 Ledger Standard	
<i>Murdoch J. Gabbay, Arvid Jakobsson, and Kristina Sojakova</i> .....	2:1–2:14

### Short Papers

Using Coq to Enforce the Checks-Effects-Interactions Pattern in DeepSEA Smart Contracts	
<i>Daniel Britten, Vilhelm Sjöberg, and Steve Reeves</i> .....	3:1–3:8
Formally Documenting Tenderbake	
<i>Sylvain Conchon, Alexandrina Korneva, Çağdas Bozman, Mohamed Iguernlala, and Alain Mebsout</i> .....	4:1–4:9
Towards Contract Modules for the Tezos Blockchain	
<i>Thi Thu Ha Doan and Peter Thiemann</i> .....	5:1–5:9





## ■ Preface

The 3rd International Workshop on Formal Methods for Blockchains (FMBC) took place virtually on July 18/19 2021 as part of CAV 2021, the 33rd International Conference on Computer-Aided Verification. FMBC's purpose is to be a forum to identify theoretical and practical approaches applying formal methods to blockchain technology.

This third edition of FMBC attracted 15 submissions on topics such as verification of smart contracts or analysis of consensus protocols. Each paper was reviewed by at least three program committee members or appointed external reviewers. This led to a selection of 5 papers (2 long and 3 short) that were presented at the workshop as regular talks, as well as 3 extended abstracts that were presented as lightning talks. Additionally, we were very pleased to have an invited keynote by David L. Dill (Novi/Facebook, USA).

This volume contains the papers selected for regular talks as well as the abstract of the invited talk.

We thank all the authors that submitted a paper, as well as the program committee members and external reviewers for their immense work. We are grateful to Arie Gurfinkel, Workshop Chair of CAV 2021, for his guidance. Finally, we would like to express our gratitude to our sponsor Nomadic Labs for its generous support.

September 2021

Bruno Bernardo  
Diego Marmsoler



**nomadic labs**

3rd International Workshop on Formal Methods for Blockchains (FMBC 2021).

Editors: Bruno Bernardo and Diego Marmsoler



OpenAccess Series in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany





## ■ Program Committee

Wolfgang Ahrendt  
Chalmers University of Technology, Sweden

Lacramioara Astefanoei  
Nomadic Labs, France

Massimo Bartoletti  
University of Cagliari, Italy

Bruno Bernardo  
Nomadic Labs, France

Joachim Breitner  
Dfinity Foundation, Germany

Achim Brucker  
University of Exeter, UK

Zaynah Dargaye  
Nomadic Labs, France

Jérémie Decouchant  
TU Delft, Netherlands

Dana Drachler Cohen  
Technion, Israel

Ansgar Fehnker  
University of Twente, Netherlands

Maurice Herlihy  
Brown University, USA

Lars Hupel  
INNOQ, Germany

Florian Kammüller  
Middlesex University London, UK

Igor Konnov  
Informal Systems, Austria

Andreas Lochbihler  
Digital Asset, Switzerland

Diego Marmosler  
University of Exeter, UK

Simão Melo de Sousa  
Universidade da Beira Interior, Portugal

Karl Palmskog  
KTH, Sweden

Maria Potop-Butucaru  
Sorbonne Université, France

Andreas Rossberg  
Dfinity Foundation, Germany

Albert Rubio  
Complutense University of Madrid, Spain

César Sanchez  
Imdea, Spain

Clara Schneidewind  
TU Wien, Austria

Ilya Sergey  
Yale-NUS College/NUS, Singapore

Mark Staples  
CSIRO Data61, Australia


Meng Sun  
Peking University, China

Simon Thompson  
University of Kent, UK

Josef Widder  
Informal Systems, Austria

3rd International Workshop on Formal Methods for Blockchains (FMBC 2021).

Editors: Bruno Bernardo and Diego Marmosler

 OpenAccess Series in Informatics

**OASICS** Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## ■ Supporting Reviewers

Yuteng Lu

Luis Arrojado da Horta



# Towards Verified Price Oracles for Decentralized Exchange Protocols

Kinnari Dave ✉

CertiK, New York, NY, USA

Vilhelm Sjöberg ✉

CertiK, New York, NY, USA

Xinyuan Sun ✉

CertiK, New York, NY, USA

---

## Abstract

Various smart contracts have been designed and deployed on blockchain platforms to enable cryptocurrency trading, leading to an ever expanding user base of decentralized exchange platforms (DEXs). Automated Market Maker contracts enable token exchange without the need of third party book-keeping. These contracts also serve as price oracles for other contracts, by using a mathematical formula to calculate token exchange rates based on token reserves. However, the price oracle mechanism is vulnerable to attacks both from programming errors and from mistakes in the financial model, and so far their complexity makes it difficult to formally verify them. We present a verified AMM contract and validate its financial model by proving a theorem about a lower bound on the cost of manipulation of the token prices to the attacker. The contract is implemented using the DeepSEA system, which ensures that the theorem applies to the actual EVM bytecode of the contract. This theorem could be used as proof of correctness for other contracts using the AMM, so this is a step towards a verified DeFi landscape.

**2012 ACM Subject Classification** Software and its engineering → Software verification

**Keywords and phrases** Smart Contract Verification, Interactive Theorem Proving, Blockchain, Decentralized Finance

**Digital Object Identifier** 10.4230/OASICS.FMBC.2021.1

**Supplementary Material** *Software (Source Code)*: <https://github.com/certikfoundation/deepsea>

**Acknowledgements** The work was partially supported by a Conflux Ecosystem Grant (October 2020).

## 1 Introduction

The last two years have seen a rapidly increasing interest in using *decentralized finance* (DeFi) instead of traditional centralized exchanges in order to trade, lend, and borrow cryptocurrencies. DeFi puts the trading logic into a smart contract on the blockchain, which increases trust and transparency, and lets anyone compose financial applications “like lego pieces”. Smart contracts also enable completely new financial primitives, e.g. *flash loans* [22], risk-free lending of very large amounts which will be paid back within a single blockchain transaction. Estimates say DeFi total trading volume increased from \$0.67 billion in January 2020 to \$70 billion in January 2021, while DeFi investments reached 20.5 billion in January 2021 [16, 17].

However, protocols in decentralized finance are vulnerable to hacks. In 2020 there were at least 16 large DeFi hacks, with total losses of \$196 million. Some of these were due to mistakes in the financial model (we give an example below in Section 2.2), while in others the financial theory was sound but the contract itself was implemented incorrectly [19].

The high stakes of DeFi makes it crucial that the smart contracts executing these protocols come with formal guarantees. However, applying formal verification to them is challenging. Reasoning about the financial models often requires mathematics, e.g. real analysis, that



© Kinnari Dave, Vilhelm Sjöberg, and Xinyuan Sun;

licensed under Creative Commons License CC-BY 4.0

3rd International Workshop on Formal Methods for Blockchains (FMBC 2021).

Editors: Bruno Bernardo and Diego Marmosoler; Article No. 1; pp. 1:1–1:14

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

goes beyond the capabilities of non-interactive theorem provers such as SMT solvers. And even if we can prove theorems about the financial model, we must still show that the actual program code correctly implements the model. Existing tools either try to work at the model level, or they can prove quite shallow properties about code.

In this paper, we consider one of the most widely used DeFi protocols, a Uniswap-style automated market making (AMM) contract. Various attempts [4, 3, 7] have been made at studying the AMM model mathematically and reasoning about specific cases. However, these are paper proofs and do not directly reason about the program being executed on the virtual machine.

We make use of the DeepSEA system, which has been tailored to support rigorous formal verification. The DeepSEA compiler can automatically generate a high-level Coq model for a contract, so we know that the theorem we prove in Coq will apply to the actual executed code. Achieving this requires some care, because existing work deals with the AMM model in terms of real numbers, and we must lift that proof to give bounds for the integer variables in the actual program.

AMM contracts are a basic building block of more complex financial contracts. In the future, we envision that such contracts will also be formally verified, e.g. by using the result we prove here as one lemma.

**Contributions.** We make the following contributions in this paper:

1. We implement a Uniswap-style AMM contract in DeepSEA (Section 3).
2. We formalize in Coq a result, previously only proven on paper [4], which establishes a lower bound on the cost to an attacker of manipulating prices quoted by the AMM contract (Section 5.2). Our formalization makes use of existing third-party math developments (Section 5.1), which shows the benefit of working inside a general-purpose proof assistant.
3. We also establish the non-depletion property of the contract, i.e it is impossible to drain the contract of all its reserves by swapping any number of tokens. (Section 5.2). This proof requires exporting integer inequalities to reals and using ring homomorphism properties to transport them back to integers, as is evident in the *math\_lemma.v*<sup>1</sup> module.
4. We use the auto generated Coq functions by the DeepSEA compiler frontend to link the bytecode to the formalized proof, thus establishing important financial properties of the generated bytecode (Section 5.3).

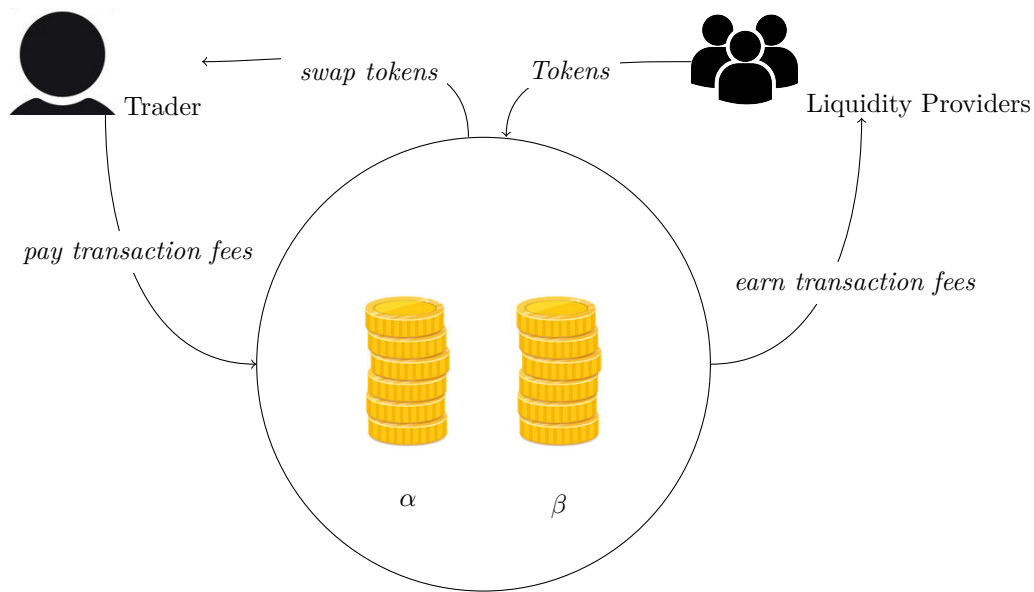
In the rest of the paper, we first explain the setting of the work (Section 2), then our specific contributions, and finally we discuss related work and conclude.

## 2 Background

“Market making” is the process of providing liquidity for various assets by continuously quoting of the price at which the market maker is willing to buy and the price at which the market maker is willing to sell their asset. Traditionally, these price quotes are listed in an order book, which records the current assets open for buying/selling. This requires trust in the central party managing the liquidity pools. The idea behind an Automated Market Maker protocol is to replace the central party with a smart contract that owns reserves of

---

<sup>1</sup> Available online at [https://github.com/certikfoundation/deepsea/blob/master/contracts/amm/math\\_lemma.v](https://github.com/certikfoundation/deepsea/blob/master/contracts/amm/math_lemma.v)



■ **Figure 1** AMM mechanism.

two Ethereum-hosted cryptocurrencies (a.k.a tokens), and trades them at a price determined using a mathematical formula. Once these smart contracts are deployed, they can also serve as price oracles for other smart contracts.

## 2.1 Automated Market Makers

Automated Market Makers are decentralized exchange protocols which facilitate trading of tokens on blockchain based platforms by providing liquidity pools without an order book mechanism. These protocols allow exchange between pairs of tokens, and each token pair has a corresponding smart contract which facilitates the exchange. The exchange rate is calculated using a mathematical formula which is a function of the token reserves in the contract. We focus on the constant product market makers. This type of automated market makers satisfy the invariant:

$$x_A \cdot y_B = k$$

where  $x_A, y_B$  are the reserves for token A and token B respectively. It follows from this formula that the marginal price of A with respect to B (the price offered by the contract for small trades) is the ratio of the token reserve of B to that of A. Since the Uniswap protocol [1] is one of the most popular implementations of the AMM mechanism, we briefly describe the functionalities it provides and its mechanism. The protocol is designed so that the smart contract implementing it interacts with two kinds of users: Liquidity Providers and Traders.

Liquidity Providers contribute to the pool of reserves of the token pair. This is enabled by issuing a liquidity token to the Liquidity Provider when they choose to contribute to the pool. The token dictates the share of the token reserves that the provider is entitled to. The provider can treat the liquidity token as an asset that can be traded. To incentivize the providers, they receive an interest proportionate to their shares, which is funded by charging traders a 0.3% transaction fee for each trade they make with the contract. The `mint()` method facilitates minting of liquidity tokens for the providers.

## 1:4 Towards Verified Price Oracles for Decentralized Exchange Protocols

At any point, the provider is free to withdraw liquidity from the token reserves by calling the `burn()` method from the smart contract. On doing so, they receive the tokens they had lent to the liquidity pool plus the interest they earned from the transaction fees.

The number of liquidity tokens minted for a particular liquidity provider is determined by their share in the token reserve. There are various formulae used to calculate this. The Uniswap protocol determines the number of tokens minted using the following formula:

$$s_{minted} = \frac{x_{deposited}}{x_{starting}} \cdot s_{starting}$$

Here,  $s_{minted}$  denotes the number of Liquidity Tokens minted for the amount  $x_{deposited}$  tokens that have been deposited to the pool containing  $x_{starting}$  tokens and  $s_{starting}$  liquidity tokens representing contributions to that liquidity pool.

In the event that liquidity is being deposited to the reserves for the first time, the above formula doesn't work (since  $x_{starting} = 0$ ). In this case, the number of liquidity tokens minted is given by:

$$s_{minted} = \sqrt{x_{deposited} \cdot y_{deposited}}$$

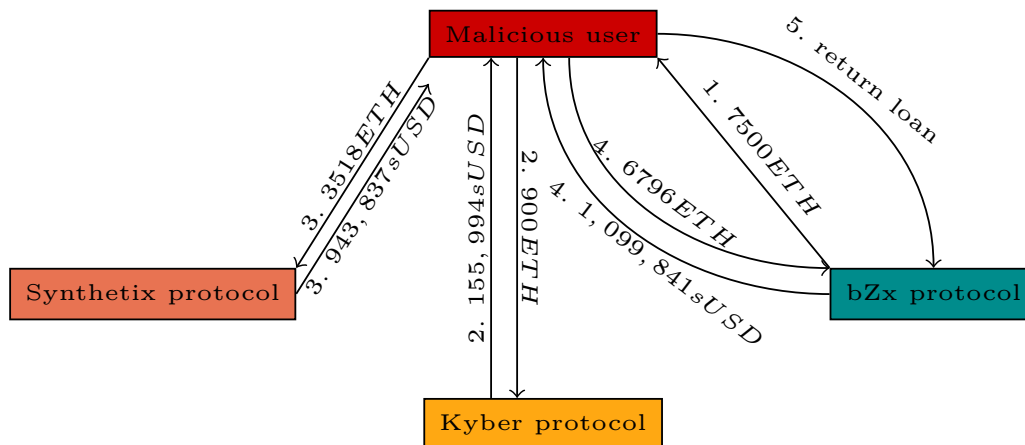
where  $x_{deposited}, y_{deposited}$  denotes the pair reserves that the depositor lends to the contract. Additionally, the Uniswap protocol charges a 0.05% protocol fee as a part of the net 0.3% transaction fee charged to traders. This fee is optional and is turned off for the DeepSEA implementation of the AMM contract.

## 2.2 Oracles

The AMM mechanism also supports a price oracle function. The first protocol designed by Uniswap supports an on-chain price oracle which computes prices using the constant product market maker formula and reports instantaneous prices when queried. Other contracts can e.g. issue loans of one token guaranteed by a collateral in another token, and use the reported price to calculate how much the collateral is worth. However, the mechanism of reporting instantaneous prices is highly susceptible to attacks, in particular in combination with flash loans. Let us consider an example of an oracle attack which happened on 18th February 2020 [5]:

► **Example 1.** The bZx protocol is a lending protocol which facilitates decentralized borrowing and lending of assets on Ethereum. The Kyber network on the other hand is an on-chain AMM protocol similar to Uniswap. An attacker flash-borrowed 7500 ETH from the bZx protocol, then called the Kyber protocol to swap a net amount of 900 ETH with 155,994 sUSD. This affects the reserves of ETH and sUSD in the Kyber protocol thus affecting the prices reported by it. The attacker later relies on bZx querying the faulty Kyber oracle to borrow ETH against sUSD at a cheap rate. To get the sUSD required to perform this exchange on bZx, the attacker buys sUSD from an unrelated contract at a normal rate. They used the Synthetix depot contract, which had larger reserves and therefore did not change price as much as Kyber. They used 3518 ETH from their borrowed ETH to get 943,837 sUSD. Now, they borrow 6796 ETH from bZx with a collateral of only 1,099,841 sUSD. They are able to do this because of the price manipulation on the Kyber oracle which bZx queries. Finally, they are able to transfer back the 7500 ETH borrowed from bZx to repay the flash loan. In effect, bZx lost \$600k in equity.





■ **Figure 2** bZx protocol attack.

How can such attacks be avoided? There are several partial solutions. The lending contract can try to avoid being called inside a flash-loan transaction (limiting the amount of funds available for oracle manipulations), or use a “slippage check”<sup>2</sup> to detect if a manipulation is in progress. The oracle can report a time-weighted average instead of the instantaneous price, to smooth out spikes (this approach is adopted by the Uniswap v2 protocol). We believe the ideal solution, which we build towards in this paper, is to *prove* that attacks are impossible by calculating the cost of such manipulation to the attacker as a function of various parameters of the contract such as token reserves. Once this is achieved, these parameters can be modified to make the cost of manipulation high enough that the attack can not be carried out using the funds available to the attacker.

### 2.3 The DeepSEA system

DeepSEA (Deep Simulation of Executable Abstractions), is a programming language and system that links high-level specifications in Coq [20] to executable code. The original version [18] compiled programs into C, while a new version [10] compiles Ethereum contracts to Ethereum Virtual Machine (EVM) bytecode.

The DeepSEA compiler works in two steps. The front end parses and type-checks the input to create a typed intermediate representation. From the intermediate representation it then generates two things. First, a set of Coq Gallina functions that serves as a high-level model of the program, one function for each method in the contract. Since Gallina is a pure functional language, monads are used to capture effects. The end-user can load this model into their own Coq project, and prove theorems about the contract just as they would about any program written in Coq. Second, there is a backend similar to the CompCert compiler [12], which goes through a series of phases of intermediate representations and generates an EVM bytecode file. Crucially, there is a proof in Coq (although it is not yet complete) that this compilation is done correctly, which will give a high degree of certainty that results proven about the high-level specifications also hold for the bytecode.

<sup>2</sup> Slippage is defined to be the change in price in the time period between a trade order being placed and its execution. Hence causing traders to settle for a price different from the one they initially requested. An ongoing attack would cause such slippage.

Contracts written in DeepSEA are structured similarly to Solidity contracts, as a set of objects which contain state (storage) variables and methods which can modify the state. In DeepSEA the objects are further organized into “layers”, which can express the modular structure of large systems.

### **3** DeepSEA AMM

The smart contract written in DeepSEA to support AMMs<sup>3</sup> uses the Uniswap v2 protocol as a blueprint. Instead of dividing the functionality of the protocol into two basic types of smart contracts (as is done in the Uniswap protocol), the DeepSEA contract combines the functionality of the router contracts and that of the core contracts into a single contract with two sets of methods corresponding to the above classification.

In the DeepSEA setup, the entire contract is defined as a layer AMM on top of an underlay layer called the AMMLIB. The AMMLIB layer consists of three objects: two ERC20 tokens which are to be swapped and a liquidity token. The AMM layer acts as the interface for the contract. This layer consists of an object of type AMMInterface, which defines the methods that provide all the functionalities of the protocol. The methods in this object signature are given as follows:

- `simpleSwap0`: This method allows the transfer of one token to the contract to be exchanged for the other, and returns the amount of the second token to be received in return.
- `mint`: This method allows the transfer of liquidity to a liquidity pool for a liquidity provider.
- `burn`: This method allows a liquidity provider to withdraw liquidity from a pool.
- `sync`: This method is a recovery mechanism method to prevent the market for the given pair from being stuck in case of low reserves.
- `skim`: This method prevents any user from depositing more tokens in any reserve than the maximum limit, to prevent overflow.
- `k`: This method tracks the product of the reserves.
- `quote0`: This method returns the equivalent amount of the second token, given an amount of the first token and current reserves in the contract.
- `getAmountOut0`: This method returns the maximum possible amount of a token than can be gained in exchange for a particular input amount of the other token and that of the reserves.
- `getAmountIn0`: This method returns the amount of a given token that must be input in order to obtain the desired amount of the other token under the given reserves.

Compared to the Uniswap protocol, we have made a few simplifications. Unlike Uniswap, which offers the option of switching on/off the protocol fee, the DeepSEA contract does not model protocol fees. Moreover, instead of using the above mentioned square root formula to calculate the share of minted liquidity tokens for a liquidity provider, the DeepSEA contract uses the product and burns the first 1000 coins, as in Uniswap v2. The price oracle mechanism is based on Uniswap v1, and the DeepSEA contract does not support flash swaps (i.e., getting flash loans of the assets in the liquidity pool). In the future we may add these features, in order to make our contract completely ABI-compatible with the original. However, the DeepSEA AMM contract already offers all the core functionality offered by the Uniswap protocol, and it contains everything that is relevant to the specification that we are verifying. As such, our proof is an example of verifying a realistic contract.

---

<sup>3</sup> Available online at <https://github.com/certikfoundation/deepsea/blob/master/contracts/amm/amm.ds>

## 4 Mathematical Analysis of Automated Market Makers

While AMM contracts hold a great deal of promise for the future of Decentralized Finance, the stability of the markets generated using smart contracts remains a concern. To address such concerns and provide a rigorous comparison with Centralized Finance, Angeris et al. carried out a mathematical analysis [4]. They define the conditions on the Uniswap price in terms of the market price so that no arbitrage opportunities arise. Moreover, they show that it is impossible to drain the contract of all its liquidity reserves, and go on to model risk in the constant product market maker model. Here we give a brief description of their results, which we mechanize using the Coq proof assistant and connect to the DeepSEA AMM contract.

### 4.1 Manipulating Prices

Since the AMM contract calculates the prices of tokens based on liquidity reserves using a mathematical formula, an attacker can potentially trade with the contract to alter reserves in order to manipulate the prices, as illustrated in Section 2.2. Angeris et al. prove that the cost of such a manipulation is proportionate to the reserves, thus confirming the intuition that large liquidity reserves lead to stable prices.

Consider an AMM contract which facilitates trading of two tokens  $\alpha, \beta$ . Let the token reserves in this contract before the swap to be analysed is performed be given by  $R_\alpha, R_\beta$  respectively. Further assume that the swap involved a deposit of  $\Delta_\beta$   $\beta$  coins and a corresponding withdrawal of  $\Delta_\alpha$   $\alpha$  coins. We consider the cost of manipulating the market price in the event of the reference market price being infinitely liquid (i.e when  $\Delta_\beta = m_p \Delta_\alpha$ ).

Suppose that the attacker performs this swap to manipulate the reported price by a fraction  $\epsilon$ . This gives us the intended price  $m_u$  as a function of  $\epsilon$  and  $m_p$ :

$$m_u = (1 + \epsilon)m_p$$

Since the AMM under consideration uses the constant product market maker formula, an alternative computation of the new price is obtained to be the inverse ratio of the token reserves:

$$m_u = \frac{R_\beta + \Delta_\beta}{R_\alpha - \Delta_\alpha}$$

Thus, giving us the equation  $\frac{R_\beta + \Delta_\beta}{R_\alpha - \Delta_\alpha} = (1 + \epsilon)m_p$ .

This swap is made in order to achieve the new price as the reported price by the oracle when queried. Since it is performed by the attacker, Angeris et al. calculate its cost to the attacker and establish a lower bound on this cost. The attacker deposited  $\Delta_\beta$  amount of  $\beta$  tokens in the contract and received  $\Delta_\alpha$  amount of  $\alpha$  tokens from the contract. The price of the  $\alpha$  tokens relative to the  $\beta$  tokens before the manipulation is given by  $m_p$ . Thus, the net cost of this manipulation to the attacker is simply given by the expression:

$$\Delta_\beta - m_p \cdot \Delta_\alpha$$

After some algebraic simplifications, this cost can be calculated as a function of  $\epsilon$  and fixed contract parameters:

$$C(\epsilon) = \Delta_\beta - m_p \Delta_\alpha = R_\beta (\sqrt{1 + \epsilon} + (\sqrt{1 + \epsilon})^{-1} - 2)$$

The cost of manipulation theorem establishes that there is a minimal positive cost to the attacker which is proportional to the reserves of the token added.

► **Theorem 2** (Cost of manipulation). *The cost of manipulating the exchange rate of tokens by a fraction  $\epsilon$  ( $C(\epsilon)$ ) by performing a token swap is bounded below by a function of  $\epsilon$  depending on it's range:*

$$\blacksquare \quad 0 \leq \epsilon \leq 1 : C(\epsilon) \geq R_\beta \frac{\epsilon^2}{2} \text{ inf}_{0 \leq \epsilon' \leq 1} C'''(\epsilon') = \left(\frac{1}{32\sqrt{2}}\right) R_\beta \epsilon^2$$

$$\blacksquare \quad \epsilon \geq 1 : C(\epsilon) \geq \kappa R_\beta \sqrt{\epsilon}$$

where  $\kappa = 3/2 - \sqrt{2}$ .

## 4.2 Nondecreasing $k$ and no depletion

The cost of manipulation result is the most interesting result, because it has implications for the correctness of clients of the oracle and because it uses more advanced math. In addition, Angeris et al. also prove a simpler invariant which gives some additional confidence that the contract is implemented correct. We formalize this proof also.

In particular, they prove that it is impossible to drain the contract of all it's token reserves, irrespective of how many tokens the attacker can use for the swap. This is proved by showing that the net reserves of tokens in the contract are strictly bounded away from 0. The result is stated as follows:

► **Theorem 3** (Nondepletion). *At all times,  $R_\alpha + R_\beta > 0$ .*

The proof for this property follows by the AM-GM inequality, and the *increasing  $k$*  invariant. The increasing  $k$  invariant establishes that when the protocol charges trading fees, the product of the reserves of the tokens in the contract is strictly increasing over each swap operation.

These properties are related to the notion of *path independence* ([9]). For an arbitrary type of AMM, one could worry that it would be possible to exploit the liquidity providers by making repeated trades, e.g. selling token  $A$  when the price is low and then buying back when the price is high, and maybe eventually draining the entire reserve. The above invariant shows that constant-product market makers are immune to such attacks. If the product  $k$  was exactly constant, then the prices would only depend on the net amount of  $A$  traded, not on the exact “path” of buys and sells. In practice  $k$  is increasing because of trading fees, which means it is never advantageous to strategically split trades into multiple transactions.

## 5 Mechanizing results for the DeepSEA AMM contract

In this section we describe our mechanization of the properties stated in Section 4.

### 5.1 Importing third-party Coq libraries

In order to reason about various bounds on expressions used in the result established in Section 4.1, we chose to use the Coq-interval [15] library. The library supports a high degree of automation, to establish approximate preliminary bounds on certain standard functions like the square root function, polynomials, trigonometric functions, the exponential function and the logarithm. It uses Taylor models (as defined in [14]) to establish such bounds.

However, this library by itself doesn't prove to be sufficient, since it relies on Coquelicot [8] to prove certain results and doesn't provide automation to use them. In order to setup an environment compatible with these results, we use Coquelicot as well.

Additionally, we rely on the injections of natural numbers and integers into reals, (in particular, to establish the *increasing  $k$*  invariant from Section 4.2) and the proven ring homomorphism properties of these injection in the Coq standard library, to argue about integers in bytecode inside reals and then transport established inequalities over reals back to inequalities over integers.

## 5.2 Proof Outline

The formalization<sup>4</sup> of the above properties of the constant product market maker protocol in the Coq proof assistant requires the use of real analysis results.

To prove the lower bound in Theorem 2 in the first case, we use the Taylor series approximation for continuous and twice differentiable functions. We state and prove the Taylor series approximation for the function,  $\sqrt{1+\epsilon} + 1/\sqrt{1+\epsilon} - 2$  in the interval  $(0, 1]$ . The lemma is stated as follows:

```
Lemma taylor_m : 0 < eps <= 1 ->
exists eta,
(0 <> eps -> (0 < eta < eps \\/ eps < eta < 0)) /\
sqrt (1 + eps) + 1/sqrt(1 + eps) -2 =
(((2 - eta) / (8* ((1 + eta)^2) * sqrt (1 + eta))) * eps^2).
```

We use the general version of the Taylor Lagrange theorem formalized in the Coq-interval library to prove the above lemma. [14] The statement of the theorem is as follows:

```
Section TaylorLagrange.
Variables a b : R.
Variable n : nat.
Notation Cab x := (a <= x <= b) (only parsing).
Notation Oab x := (a < x < b) (only parsing).
Variable D : nat -> R -> R.
Notation Tcoeff n x0 := (D n x0 / (INR (fact n))) (only parsing).
Notation Tterm n x0 x := (Tcoeff n x0 * (x - x0)^n) (only parsing).
Notation Tsum n x0 x := (sum_f_R0 (fun i => Tterm i x0 x) n) (only parsing).
Section TL.

Hypothesis derivable_pt_lim_Dp :
forall k x, (k <= n)%nat -> Oab x ->
derivable_pt_lim (D k) x (D (S k) x).

Hypothesis continuity_pt_Dp :
forall k x, (k <= n)%nat -> Cab x ->
continuity_pt (D k) x.
Variables x0 x : R.
Theorem Taylor_Lagrange :
exists xi : R,
D 0 x - Tsum n x0 x =
Tcoeff (S n) xi * (x - x0)^(S n)
/\ (x0 <> x -> x0 < xi < x \\/ x < xi < x0).
End TL.
End TaylorLagrange.
```

In the above setting, the function  $D : \text{nat} \rightarrow \mathbb{R} \rightarrow \mathbb{R}$  represents the series of a function and it's  $n$ th derivative, where  $D0$  is the function itself and  $Dn$  is it's  $(n-1)$ th derivative.  $Tsum\ n\ x0\ x$  is the sum of the first  $n$  terms of the Taylor series of the function  $D0$ . Since, a necessary condition for the Taylor Lagrange theorem to hold is that if the approximation is of the  $n$ th order then each of the  $k$ th order derivatives of the function should be continuous and

<sup>4</sup> Available online at [https://github.com/certikfoundation/deepsea/blob/master/contracts/amm/cst\\_man.v](https://github.com/certikfoundation/deepsea/blob/master/contracts/amm/cst_man.v)

## 1:10 Towards Verified Price Oracles for Decentralized Exchange Protocols

differentiable for all  $k \leq n$ , the section in the Taylor.v module includes a hypothesis, which we must prove in order to apply the theorem. We define the following function and use it in place of the above function  $D$  for our application of the Taylor Lagrange theorem:

```

Definition T_f_1 (n : nat) :=
match n with
| 0%nat => (fun x => sqrt(1+x) + (1/sqrt (1 + x)))
| 1%nat => (fun x => 1/(2* sqrt(1+x)) - (1/ (2 * (1 + x) * (sqrt (1 + x)))))
| 2%nat => (fun x => (2 - x)/ (4 * ((1 + x)^2) * sqrt(1 +x)))
| _ => (fun x => 0%R)
end.

```

The required hypothesis are stated and proved as the following lemmas:

```

Lemma deriv_lim_T_f : forall (k : nat) (x : R),
(k <= 1)%nat ->
0 < x < 1 ->
derivable_pt_lim (T_f_1 k) x (T_f_1 (S k) x).

```

```

Lemma cont_lim_T_f : forall (k : nat) (x : R),
(k <= 1)%nat ->
0 <= x <= 1 -> continuity_pt (T_f_1 k) x.

```

Once, we have the Taylor approximation, we establish a lower bound on the remainder term using the powerful interval tactic. This can be done since we have a range in which  $\epsilon$  lies for the first lower bound on the cost of manipulation (i.e  $0 \leq \epsilon \leq 1$ ). The lemma for the lower bound on the remainder term is stated as follows:

```

Lemma lower_bnd : forall eta, 0 <= eta <= 1 ->
(2 - eta) / (8 * ((1 + eta)^2) * sqrt (1 + eta)) >= 1 / 48.

```

Note that, the lower bound in ([4]) is  $1/32\sqrt{2}$ . Since the interval tactic works based on approximations [15], it cannot be used to prove exact irrational bounds. Hence, we approximate  $\sqrt{2}$  with  $3/2$  to get a lower bound of  $1/48$ . This lower bound can be made closer to  $1/32\sqrt{2}$ , by using a finer approximation. This gives us the first part of the lower bound in Theorem 2.

To prove the second part, we use a different approach from the one used in [4]. The lower bound for the case when  $\epsilon \geq 1$  involves proving the following inequality:

$$x + x^{-1} - 2 \geq \kappa x$$

where  $x = \sqrt{1 + \epsilon}$ ,  $\kappa = 3/2 - \sqrt{2}$ . Here again we approximate  $\kappa$  by  $5/100$  to facilitate the use of the interval tactic. Instead of using the analysis of quadratic equations approach as in ([4]), we use a simpler way. After a small algebraic manipulation, and accounting for approximations the above inequality can be re-written as the following lemma:

```

Lemma eps_sq : eps >= 1 ->
(sqrt(1 + eps) - 1)^2 - ((5/100) * (1 + eps)) >= 0.

```

To show this, we use the fact that if the derivative of a continuous function defined on a connected domain is positive, then the function is increasing. This coupled with the observation that the value of the l.h.s of the above inequality evaluated at 1 is positive, gives us the proof. Thus we have the lower bound on the cost of manipulation for the second case :

```

Lemma cst_func_ge_1 : eps >= 1 ->
sqrt (1 + eps) + (1 / sqrt (1 + eps)) - 2 >= ((5/100) * sqrt (1 + eps)).

```

Combining them gives us the following mechanization of Theorem 2 :

```
Definition cost_of_manipulation_val := (IZR (reserve_beta s)) *
(sqrt (1 + eps) + (1/ sqrt (1 + eps)) - 2).
```

```
Theorem cost_of_manipulation_min : eps >= 0 ->
cost_of_manipulation_val >= (IZR (reserve_beta s)) * (5/100) * sqrt (eps) \ /
cost_of_manipulation_val >= (IZR (reserve_beta s)) * (1/48) * (eps^2).
```

We also mechanize the no-depletion property from Section 4.2. The formalized version of Theorem 3 holds for the AMM contract written in DeepSEA:

```
Theorem no_depletion_reserves : (IZR (reserve_beta s'')) +
(IZR (reserve_alpha s'')) > 0.
```

Proving this result requires establishing the increasing product invariant over integers. This requires some careful reasoning over reals before the result is transported back to integers. The increasing product invariant is stated as follows:

```
Lemma increasing_k : Z.lt (compute_k s) (compute_k s').
```

The above lemma states that the product of the reserves always increases with each swap operation. Thus, we establish two independent important algorithmic properties of the bytecode corresponding to the DeepSEA AMM contract.

### 5.3 Connection to the DeepSEA contract

The results we formalized in the Coq Proof assistant about the cost of manipulation of the market price are for the AMM contract written in DeepSEA. The `cost_man.v` module imports the Coq files generated by the DeepSEA compiler, so that computations can be made using the variables of the contract. We want to know how prices are affected by a single call to the `simpleSwap0` function, we do so by adding the following hypothesis to our file:

```
Hypothesis del_alp : runStateT (AutomatedMarketMaker_simpleSwap0_opt
toA (make_machine_env a)) s = Some (r' , s').
```

Here `AutomatedMarketMaker_simpleSwap0_opt` is an automatically generated Coq function which represents the behaviour of the contract method. The hypothesis says that a call to it, exchange of the token  $\alpha$  for the input token  $\beta$ , completed without reverting and left us in a new contract state  $s'$ .

This is done to calculate  $\Delta_\alpha$ . Once the values  $R_\alpha, R_\beta, \Delta_\alpha, \Delta_\beta$  are obtained from the contract, the fraction by which the exchange price can be manipulated (i.e  $\epsilon$ ) is computed using the formula:

$$\frac{R_\beta + \Delta_\beta}{R_\alpha - \Delta_\alpha} = (1 + \epsilon) \frac{\Delta_\beta}{\Delta_\alpha}$$

Now the results stated in 4.1 are formalized for the  $\epsilon$  obtained from above and its possible values.

## 6 Related Work

We are only aware of one mechanized proof applied to a DeFi contract: Park et al.'s verification of the original Uniswap AMM using the KEVM Framework [23]. They prove that the functions implemented by the contract bytecode conforms to a high-level specification

(the constant-product formulas), and they do not prove any financial correctness properties. In other words, the end result of the verification is a set of high-level functions similar to what DeepSEA generates automatically and we take as our starting point; however, this is done for the already existing and deployed contract, while DeepSEA requires you to re-implement the contract in the DeepSEA language.

As for paper proofs, we already discussed Angeris and Chitra’s theorem [4], which we mechanize in this paper. Angeris et al. [3] consider generalizations of the Uniswap formula and further desirable properties. Bartoletti et al set out to understand DeFi protocols by writing down (on paper) abstract models of AMMs [7] and lending pools [6] as transition systems, and then proving theorems such as demand-sensitivity and non-depletion about them. In future work, we aim to provide machine-checked proofs of many such properties, as we have already done for non-depletion.

An alternative to formal proof is to apply model checking [21, 19] or graph search with constraints [24] to find DeFi hacks. These works manually translate and simplify the set of contracts into a language the model checker can deal with, and can then make a best effort to find exploits. Because model checking is automatic (so it requires less user effort) we believe this can be a useful complement.

The kind of oracle we consider provides pricing information based directly on on-chain trades. This should be distinguished from oracles that aggregate data from off-chain sources and posts them on the blockchain. Analyses of the latter [11, 13] show that they, too, have problems with spurious data spikes and possible attacks.

## **7** Conclusions and Future Work

We take the first step in formally verifying financial properties of the contract at the algorithmic level. This is enabled by the Coq functional model that is automatically generated by the DeepSEA compiler. Not only is the compilation to bytecode verified, but we also have a formalization of the desirable properties of the Constant Product Market Maker model which is directly tied to the DeepSEA AMM contract. Hence we have a verified specification and verified code.

In the future, we want to extend this line of work in two directions. First, we want to consider theorems about more advanced kinds of oracles. The non-manipulation result we formalized is still the state of the art when it comes to mathematical analysis of oracles, but more recent developments have made it partially obsolete. As we explained in Section 5.2, the assumption is that the attacker makes a loss because he bought tokens at a too-high price. But in a scenario involving flash-loans this is not necessarily the case, because in a single transaction the attacker can carry out the manipulation and then immediately sell back the tokens again. There is not enough time for other market participants to exploit the incorrect price through arbitrage. Similar considerations apply if the attacker is colluding with a miner to control the order of transactions in a block. Newer oracles such as Uniswap v2 [1] and Uniswap v3 [2] are more robust and hacker-resistant than the simple instantaneous-price oracle that we consider, because they average the price over a longer time period. Theoretical results about such models however still remain to be established.

Second, it will be interesting to prove the correctness of a *client* of an oracle, e.g. to give bounds on when a lending protocol can become undercollateralized. Just like the DeFi applications themselves are built from “money lego”, we hope that they can be verified by composing together theorems about the individual components.



---

**References**


---

- 1 Hayden Adams, Noah Zinsmeister, and Dan Robinson. Uniswap v2 core. 2020, 2020. URL: <https://uniswap.org/whitepaper.pdf>.
- 2 Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 core, 2021.
- 3 Guillermo Angeris and Tarun Chitra. Improved price oracles: Constant function market makers. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 80–91, 2020.
- 4 Guillermo Angeris, Hsien-Tang Kao, Rei Chiang, Charlie Noyes, and Tarun Chitra. An analysis of uniswap markets. *arXiv preprint*, 2019. [arXiv:1911.03380](https://arxiv.org/abs/1911.03380).
- 5 Korantin Auguste. The bzx attacks explained. Blog post. <https://www.palkeo.com/en/projects/ethereum/bzx.html#second-transaction>, 2020. (Accessed on 05/23/2021).
- 6 Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch-Lafuente. Sok: Lending pools in decentralized finance. *CoRR*, abs/2012.13230, 2020. [arXiv:2012.13230](https://arxiv.org/abs/2012.13230).
- 7 Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch-Lafuente. A theory of automated market makers in defi. *arXiv preprint*, 2021. [arXiv:2102.11350](https://arxiv.org/abs/2102.11350).
- 8 Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for coq. *Mathematics in Computer Science*, 9(1):41–62, 2015.
- 9 Vitalik Buterin. On path independence. Blog post, 2017. URL: <https://vitalik.ca/general/2017/06/22/marketmakers.html>.
- 10 CertiK Foundation. DeepSEA. (Accessed on 05/23/2021). URL: <https://github.com/certikfoundation/deepsea>.
- 11 Wanyun Catherine Gu, Anika Raghuvanshi, and Dan Boneh. Empirical measurements on pricing oracles and decentralized governance for stablecoins. *Available at SSRN 3611231*, 2020.
- 12 Xavier Leroy. The CompCert verified compiler. <http://compcert.inria.fr/>, 2005–2021.
- 13 Bowen Liu, Pawel Szalachowski, and Jianying Zhou. A first look into defi oracles. *arXiv preprint*, 2020. [arXiv:2005.04377](https://arxiv.org/abs/2005.04377).
- 14 Érik Martin-Dorel, Laurence Rideau, Laurent Théry, Micaela Mayero, and Ioana Pasca. Certified, efficient and sharp univariate taylor models in coq. In *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 193–200. IEEE, 2013.
- 15 Guillaume Melquiond. Coq-interval. *Retrieved June, 17:2017*, 2011.
- 16 miscellaneous. Cryptocurrency statistics. Blog post, 2020. URL: <https://duneanalytics.com/queries/4494/8769>.
- 17 miscellaneous. Defi statistics. Blog post, 2020. URL: <https://cointelegraph.com/news/defi-hacks-and-exploits-total-285m-since-2019-messari-reports>.
- 18 Vilhelm Sjöberg, Yuyang Sang, Shu-chun Weng, and Zhong Shao. DeepSEA: a language for certified system software. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.
- 19 Xinyuan Sun, Shaokai Lin, Vilhelm Sjöberg, and Jay Jie. How to exploit a defi project (extended talk abstract). Talk at the 1st Workshop on Decentralized Finance (DeFi), colocated with Financial Cryptography and Data Security 2021 (fc21), March 2021.
- 20 The Coq Development Team. The Coq proof assistant. <https://coq.inria.fr/>. Accessed: 28/5/2019.
- 21 Palina Tolmach, Yi Li, Shang-Wei Lin, and Yang Liu. Formal analysis of composable defi protocols. *CoRR*, abs/2103.00540, 2021. [arXiv:2103.00540](https://arxiv.org/abs/2103.00540).
- 22 Dabao Wang, Siwei Wu, Ziling Lin, Lei Wu, Xingliang Yuan, Yajin Zhou, Haoyu Wang, and Kui Ren. Towards understanding flash loan and its applications in defi ecosystem. *CoRR*, abs/2010.12252, 2020. [arXiv:2010.12252](https://arxiv.org/abs/2010.12252).

## 1:14 Towards Verified Price Oracles for Decentralized Exchange Protocols

- 23 Daejun Park Yi Zhang, Xiaohong Chen. Formal specification of constant product market maker model and implementation, 2018. URL: <https://github.com/runtimeverification/verified-smart-contracts/blob/uniswap/uniswap/x-y-k.pdf>.
- 24 Liyi Zhou, Kaihua Qin, Antoine Cully, Benjamin Livshits, and Arthur Gervais. On the just-in-time discovery of profit-generating transactions in defi protocols. *arXiv preprint*, 2021. [arXiv:2103.02228](https://arxiv.org/abs/2103.02228).

# Money Grows on (Proof-)Trees: The Formal FA1.2 Ledger Standard

Murdoch J. Gabbay 

Heriot-Watt University, Edinburgh, UK

Nomadic Labs, Paris, France

Arvid Jakobsson 

Nomadic Labs, Paris, France

Kristina Sojakova<sup>1</sup> 

INRIA, Paris, France

---

## Abstract

---

Once you have invented digital money, you may need a ledger to track who owns what – along with an interface to that ledger so that users of your money can transact. On the Tezos blockchain this implies: a smart contract (distributed program), storing in its state a ledger to map owner addresses to token quantities; along with standardised endpoints to query and transact on accounts.

A bank does a similar job – it maps account numbers to account quantities and permits users to transact – but in return the bank demands trust, it incurs expense to maintain a centralised server and staff, it uses a proprietary interface . . . and it may speculate using your money and/or display rent-seeking behaviour. A blockchain ledger is by design decentralised, inexpensive, open, and it won't just decide to bet your tokens on risky derivatives (unless you want it to).

The FA1.2 standard is an open standard for ledger-keeping smart contracts on the Tezos blockchain. Several FA1.2 implementations already exist.

Or do they? Is the standard sensible and complete? Are the implementations correct? And what are they implementations *of*? The FA1.2 standard is written in English, a specification language favoured by wet human brains but notorious for its incompleteness and ambiguity when rendered into dry and unforgiving code.

In this paper we report on a formalisation of the FA1.2 standard as a Coq specification, and on a formal verification of three FA1.2-compliant smart contracts with respect to that specification. Errors were found and ambiguities were resolved; but also, there now exists a *mathematically precise* and battle-tested specification of the FA1.2 ledger standard.

We will describe FA1.2 itself, outline the structure of the Coq theories – which in itself captures some non-trivial and novel design decisions of the development – and review the detailed verification of the implementations.

**2012 ACM Subject Classification** Software and its engineering → Formal software verification; Theory of computation → Program verification; Social and professional topics → Quality assurance

**Keywords and phrases** Distributed ledger, smart contracts, Coq, formal verification, blockchain

**Digital Object Identifier** 10.4230/OASICS.FMBC.2021.2

**Supplementary Material** *Software (Source Code)*: [https://gitlab.com/nomadic-labs/mi-cho-coq/-/tree/kristina@fa12-verification-v2/src/contracts\\_coq](https://gitlab.com/nomadic-labs/mi-cho-coq/-/tree/kristina@fa12-verification-v2/src/contracts_coq)

archived at `swh:1:dir:d27f5d0a1c97463a68274d4006103dcc032a401b`

**Acknowledgements** The authors acknowledge the support of Nomadic Labs and the detailed and very helpful comments of three anonymous referees. We also thank Benoît Rognier, the Edukera team, and Tom Jack for their feedback and support.

---

<sup>1</sup> Sojakova wrote the the Coq code described in this paper.



## 1 Introduction

### 1.1 Tezos: a universal, modular blockchain

The Tezos blockchain was outlined in a 2015 whitepaper [3] and went live in September 2018. It is an accounts-based proof-of-stake blockchain system with the unique feature that it is a *universal blockchain* in the sense that the protocol for running Tezos is itself data on the Tezos blockchain, and this data is subject to regular upgrade by stake-weighted community vote.<sup>2</sup> Universality favours a healthy modularity at every level of the system’s design, since almost anything in the running system can be and is subject to update.

Tezos has *just one* native token: the *tez*. Further tokens can be created in a modular fashion, using smart contracts.

Thus we can represent Ethereum and Bitcoin on Tezos (using so-called *wrapped tokens*);<sup>3</sup> we can represent NFTs (non-fungible tokens representing unique assets); likewise for stablecoins and so forth. All these things can be and have been represented as Tezos smart contracts. Given this freedom, we need *interoperability standards* for our tokens to adhere to. After all, a token on its own is useless; its value comes from how we might *transact* with it.<sup>4</sup>

### 1.2 The FA1.2 standard: five entrypoints, in English

The **FA1.2 standard** is an English document specifying a minimal API for a ledger-like smart contract. Compliance with FA1.2 ensures some degree of interoperability across multiple smart contracts and tools on the Tezos blockchain.

The FA1.2 standard asserts that a given smart contract should provide the following five entrypoints and behaviours:

1. `%transfer` expects a **from** account, a **to** account, and an **amount** of tokens to be transferred, and updates the ledger accordingly.
2. `%approve` expects an **owner**, a **spender**, and a **new allowance** for the spender, and updates the transfer approvals accordingly.<sup>5</sup>
3. `%getAllowance` expects an **owner**, a **spender**, and returns the approved transfer allowance for the spender, via a callback (see Remark 2 below).
4. `%getBalance` expects an **owner** and returns the owner’s balance via a callback.
5. `%getTotalSupply` returns the total sum of all balances in the ledger, via a callback.

► Remark 1. The list above is nearly a complete summary of the FA1.2 standard, which is just a couple of pages long and clearly intended to be as straightforward as possible (which is a good thing). A few words may be helpful on what this standard *leaves out*:

<sup>2</sup> As the programs of the “universal” Turing machine are themselves data on its memory. The “regular upgrade” property is called a *self-amendment* in the Tezos literature.

To be precise: for space-efficiency, the Tezos blockchain holds not the protocol code but its hash – it is a standard trick to store large datastructures off-chain and retain an on-chain hash. When the protocol self-amends the hash gets updated and code matching that hash – which must be the protocol code (where “must” = “our hash function is computationally infeasible to break”) – propagates across the network for nodes to load and run. This low-level functionality is handled by a “shell” (think: BIOS).

<sup>3</sup> We mention a few wrapped tokens at the start of Section 4.

<sup>4</sup> Like money in the bank is only useful because you could use it to perform transactions. You don’t *have* to – at least not all at once – but that’s not the point: what matters is that you *could*.

<sup>5</sup> When you use a debit card you *authorise* a debit. The merchant could in principle not do this; thus the authorisation is granted but the withdrawal does not take place. Likewise a direct debit is an approval for a withdrawal. Similarly `%approve` does not send tokens; it approves another smart contract to make a token withdrawal, up to some limit. E.g. when you sell tokens for *tez* in Dexter, you give permission using `%approve` for Dexter to transfer tokens from your account.

1. The FA1.2 standard *does not* exclude entrypoints that are not mentioned in the list above: an implementation may offer additional entrypoints.
2. The FA1.2 standard *does not* constrain the behaviour of additional entrypoints, if present in an implementation. These entrypoints could change balances or allowances (such as some kind of *admin* entrypoint) or the total number of tokens in circulation (often called *mint* or *burn* entrypoints).
3. The FA1.2 standard *does not* exclude additional preconditions on the entrypoints that it mentions: an implementation may impose additional preconditions.

For example, consider a simple standard for doors that insists on just one entrypoint, `%openDoor`, with two conditions: if the call to `%openDoor` succeeds then the door will be open afterwards; and the call fails with error “alreadyOpen” if the door is already open. Hopefully your home’s front door complies with this standard, which is why it is called *a door*, but also: it has an additional implementation-specific entrypoint `%lockDoor`; and an additional implementation-specific precondition on its `%openDoor` entrypoint that it will fail – even if the door is not open! – if the door is locked; and finally the door has an administrative override entrypoint `%fireDepartment`, to be invoked only by people with authorisation and in special circumstances.<sup>6</sup> Yet none of this implementation-specific structure makes your front door any less of *a door*. More on this is in Section 4 and Remark 8.

► **Remark 2 (Callbacks).** A call to any entrypoint of a smart contract in Tezos takes some parameters, some (possibly zero) quantity of tez, and a continuation address of another entrypoint, called a *callback*, to which flow of control will continue. Thus “returns X via a callback” above means X is passed as a parameter to the nominated callback entrypoint.

### 1.3 This is not enough

The English FA1.2 standard is reasonable *per se*, but it is not enough:

1. The FA1.2 standard is written in English. This means it *might* be incomplete or incoherent,<sup>7</sup> and it *can’t* be directly manipulated using verification tools.
2. Just because a smart contract claims to be FA1.2-compliant does not mean that it is: perhaps it is buggy; perhaps it is hostile; perhaps the implementors just interpreted the English specification differently than the standard’s authors intended.
3. The FA1.2 standard is not itself a standard for verifying compatibility with the FA1.2 standard! That is: given two verifications of two implementations (or even of the same implementation), it is not *a priori* guaranteed that they are verifying *the same properties* – and the FA1.2 standard, which is written in English, cannot help resolve this.

To state the obvious: ledgers are safety-critical. This is real money – for a certain 21st century definition of “real” – that our smart contracts could be manipulating [4, 1].

Saying “trust us, we’re experts” is problematic not just because we might be wrong, but also because an open permissionless blockchain should not demand such trust: users should be able to check correctness, or trust that somebody independent of a central “expert” authority has checked or could check this, and (since this is an open system) they should best also trust that whatever “correctness” means, it means nearly, and preferably precisely, the same to them as to the other users with whom they might transact.

<sup>6</sup> True story. The first author has seen this entrypoint invoked.

<sup>7</sup> In fact there’s no “might” about it: a quick scan of the standard reveals points which a suitably naïve, bloody-minded, or hostile reader could interpret in spectacularly different ways, in spite of the authors’ efforts to be precise. Thus multiple implementations could exist, doing radically different things and all claiming plausibly to be “true” to “the” FA1.2 standard. This is not a criticism of FA1.2 or its authors: it is in the nature of the English language itself.

## 1.4 Our work in a nutshell

This paper reports on a verification effort undertaken at Nomadic Labs that we argue addresses the points above. That is:

- we place the FA1.2 standard on a precise mathematical footing that can be both trusted and verified, and
- we check correctness of three smart contracts which claim to be FA1.2-compliant.

The reader should not expect novel maths in this work – indeed, in this context “novel” = “untested” and may best be avoided where possible. However, there are other types of innovation to this work:

1. To our knowledge, this is the only full formalisation of a blockchain ledger standard and of multiple implementations against it, in a theorem-prover.

This addresses the three points above, by providing: a formal specification of the standard, formal representations within the theorem-prover of the programs themselves, proofs of compliance for the latter with respect to the former – and also a gold standard for comparing and operating on all of these proofs, since they are all proof-objects within the theorem-prover itself.

2. Also relevant is the theory files’ structure, which is new as we discuss below.

Having secure and reliable ledgers on Tezos is an existential issue for the blockchain ecosystem, so the fact that this could be nailed down, as we have done, has both practical and theoretical importance. Thus, this work exemplifies the application to a tangible industrial problem of a particular (Coq-based) theorem-prover technology ecosystem.

► **Remark 3.** We may write *smart contract* and *implementation* synonymously. Also, smart contracts may be written in high-level languages, but to run on Tezos they must get compiled to a lower-level stack-based language called Michelson.<sup>8</sup> We may not always distinguish between the original program and its compiled Michelson executable, but we will when this difference matters and it will always be clear what is meant.

► **Remark 4.** The formal FA1.2 standard does not replace the English FA1.2 standard: to be fully proficient a reader would have to know Coq *and also* understand something about how Tezos contracts are embedded into it. However, the formal FA1.2 standard (or an evolution of it) *could* serve as a standard reference within certain expert communities, and even outside such communities a reader with good reason to try could parse the Coq code, if they have some knowledge of dependent types and perhaps have read the English standard and looked at this paper. Thus, the formal and the English FA1.2 standards are embedded in what one might call a larger “space of understanding”, within which they complement and enrich one another such that each is made stronger and more rigorous by the existence of the other.

## 2 Introducing: the formal FA1.2 standard

The formal FA1.2 standard is written in Coq and structured into a small number of modules:

1. `FA12StorageAccess`, `FA12StorageDefinitions`, and `FA12StorageAxioms`: These specify internal functions which the smart contract must support (see Figure 1), along with axioms on their behaviour. These functions are not entrypoints and cannot be called from outside the smart contract. They may be explicit in the code of a concrete smart contract implementation, but not necessarily – e.g. the smart contract might be in a low-level, non-functional language – so long as they *could* be defined on the contract’s underlying data structures.

---

<sup>8</sup> Think: the Tezos equivalent of bytecode or machine code, though Michelson is still quite high level.

We might call this part of the standard an **idealised implementation**, where “idealised” is used in the sense of “Platonic ideal”, rather than in the sense of “perfect”.

2. **FA12Standard:** This specifies entrypoint behaviour in terms of the functions above and renders into precise Coq code the English of the FA1.2 standard (whence the module name). Note however that the formal FA1.2 standard goes beyond the English standard by specifying internal functions rather than just entrypoints as per the previous item, thus it adds some *intensional* content which the English FA1.2 standard lacks.
3. **FA12SumOfBalances:** This contains results derived from postulates in the formal FA1.2 standard, so which are guaranteed properties of any FA1.2-compliant smart contract.

```
getAllowance  : data storage_ty -> data address -> data address -> data nat
getBalance    : data storage_ty -> data address -> data nat
getTotalSupply : data storage_ty -> data nat
setBalance    : data storage_ty -> data address -> data nat
              : data storage_ty -> data storage_ty
setAllowance  : data storage_ty -> data address -> data address ->
              : data nat -> data storage_ty
```

■ **Figure 1** Types of key functions from the formal FA1.2 standard.

```
(** Asking for the balance of an owner we just set yields the new value. *)
Axiom getBalance_setBalance_eq : forall sto owner balance',
  getBalance (setBalance sto owner balance') owner = balance'.

(** Setting a balance leaves everyone else's balances unchanged. *)
Axiom getBalance_setBalance_neq : forall sto owner balance' owner',
  owner <> owner' ->
  getBalance (setBalance sto owner balance') owner' = getBalance sto owner'.
```

■ **Figure 2** Example axioms: *ledger entries are abstract arrays*.

```
(** Entry point: ep_getBalance *)
Definition ep_getBalance
  ( p : data ep_getBalance_type
  ( sto : data storage_type )
  ( ret_ops : data (list operation) )
  ( ret_sto : data storage_type ) ) :=
  let '(owner, contr) := p in
  let balance := getBalance sto owner in
  let op := transfer_tokens nat I balance tokens contr in
  ret_sto = sto /\ ret_ops = [op].
```

■ **Figure 3** Specification for %getBalance entrypoint.

```
(** In the case when the sender is withdrawing from someone else's account,
  they must be authorized to transfer at least the specified amount. *)
(sender <> from -> amount <= getAllowance sto from sender)%N
```

■ **Figure 4** Translation into Coq of a line from the English standard.

```

Theorem ep_getBalance_sumOfAllBalances
  ( env : @proto_env self_type )
  ( p : data ep_getBalance_type)
  ( sto : data storage_type )
  (ret_ops : data (list operation) )
  (ret_sto : data storage_type ) :
  ep_getBalance env p sto ret_ops ret_sto ->
  sumOfAllBalances ret_sto = sumOfAllBalances sto.
Proof.
  destruct p as [owner contr]; cbn.
  intros [H_ret_sto _].
  subst; auto.
Qed.

```

■ **Figure 5** Example result: %getBalance does not affect total supply.

```

Definition contract
:= Eval cbv in extract (contract_file_M fa12_camlcase_string.contract 500) I.

```

■ **Figure 6** Parsing a Michelson code string into Mi-Cho-Coq’s deep embedding of Michelson.

► **Example 5.** Code asserting functions of the idealised implementation is in Figure 1, and two of its example axioms are in Figure 2 (together, these axioms assert that the ledger is an *abstract array*). In these figures, `data` is a standard Mi-Cho-Coq [2] wrapper mapping (a Coq representation of) Michelson types to Coq types, and `sto` is short for “storage” and represents a state datum (ledger entries, address of admin, total of all balances, and so forth) that is threaded through computations.

► **Remark 6.** We continue the discussion of the *idealised implementation* above: The functions in Figure 1 are building blocks with which we can specify the behaviour of the entrypoints listed in Subsection 1.2. In this respect, our verification has done something that looks deceptively simple but is not. By writing these functions down we have refined the English FA1.2 standard – which speaks only about entrypoints and thus is in some sense purely extensional – to a specification which is not just more precise (since it is written in Coq); but also adds intensional structure (i.e. not having purely to do with entrypoint behaviour) in that it describes an idealised implementation which a concrete implementation must resemble in a sense made quite formal by the standard itself.

► **Example 7.** Code asserting entrypoint behaviour is in `FA12Specification`. For instance:

1. Figure 3 includes code which specifies that a call to the %getBalance entrypoint should get the balance (this is the `balance := getBalance sto owner` part, which is passed to the callback in the operation `op`) and any tokens attached to the call just get passed through untouched, as per a line from the standard that “*getBalance ... returns [the] balance of the given address, or zero if no such address is registered.*”. We spell this out (in small font) in Figure 7: see also code for `getBalance` and returning zero if no address is registered.
2. Figure 4 reflects in the formal FA1.2 standard a condition from the English FA1.2 standard that “*the transaction sender must be previously authorized to transfer at least the requested number of tokens from the “from” account using the approve entrypoint.*”



```

let balance := getBalance sto owner in (* 'owner' balance retrieved from 'sto' and put in 'balance' *)
let op := transfer_tokens
  (* 'op' is a transfer_token operation, which will act as a callback to the contract 'contr', sending
  it the value 'balance'. tez transfers and smart contract calls in Tezos are the same thing! *)
  (* Each transfer_token operation has a recipient contract (+ optional entrypoint), an amount of tez,
  and a parameter. *)
  nat      (* parameter type: each contract (+entrypoint) has a parameter type.
            In this case, recipient parameter type is 'nat', as it is to receive the 'balance',
            which is also 'nat' *)
  I        (* trivial proof by construction that 'nat' is *passable* (technical requirement) *)
  balance  (* parameter: value sent to 'contr'. balance is thus a 'nat' *)
  tokens   (* amount of tokens: using the notation 'tokens', we return
            the number of tez that was sent to this contract and that triggered this execution.
            Hence, we just "pass the tokens along". *)
  contr    (* recipient: the contract 'contr' will be the receiver of this call.
            Note that 'contr' comes from the parameter sent to 'getBalance'. Thus we have
            a "callback" pattern: the value requested is not "returned" to the caller,
            instead call back 'contr' (which may be the caller but not necessarily) with
            the requested value *)
in ret_sto = sto /\ ret_ops = [op]. (* require 'op' to be the only returned operation *)

```

■ **Figure 7** Closer look at some code from Figure 3.

► **Remark 8.**

1. Continuing Remark 1, we do not assert that an entrypoint call must succeed, even if all of the conditions described in the FA1.2 standard are met, since entrypoints can fail for implementation-specific reasons.
2. Furthermore, `FA12Specification` is a *ledger standard*, concerned with the conditions for entrypoint calls to succeed, and what happens when they do. Thus we *do* assert that an entrypoint must fail if conditions for a successful execution are unmet, but we *do not* assert what error it should return and we omit clauses from the English standard of the form “*This entrypoint can fail with the following errors*”.

This is not because they do not matter (they do, of course) but because from the point of view of the maths of maintaining a ledger, these clauses concern standards for diagnostics and debugging rather than standards for *being a ledger*. An analogy: it matters if a computation of  $100!$  reports `NatOverflow` when its 32-bit integers overflow, but not to a mathematical specification of what it is to be *the factorial function*.<sup>9</sup>

► **Example 9.** The module `FA12SumOfBalances` contains results valid for any implementation compliant with the formal FA1.2 standard, because they are derived just from postulates of the standard. Figure 5 illustrates one such result, which states that querying a balance does not change the total number of tokens on the ledger, as also returned by `%getTotalSupply`. This is a relevant result and is also a sanity check on the design, that it postulates enough that this can be proved.

<sup>9</sup> Error-reporting is also a bit of a rabbit-hole which the English standard skirts but a Coq standard could not. For example suppose *two* preconditions fail: which associated error should be returned? The English standard does not say (because it does not care) but from our point of view, disambiguating such issues is a distraction which could also restrict generality, and in ways which would not add value to the standard itself. One could even argue that the English FA1.2 standard is in fact two standards intertwined: a ledger standard, and an error-reporting standard, and we have formalised the former.

The Edukera FA1.2 verification specifies error messages as per the English standard. Is this wrong? No, just different: their verification follows more in the *verified abstraction of code* style of Remark 11 (see also Subsection 5.2) than in the *spec as a logical theory* style of this paper, so for example their specification could just state that the error returned is whatever error *actually is* returned by the implementation they are abstracting. As always, what we view as a feature depends on what we wish to achieve.

► Remark 10. We have sketched how the *FA1.2 standard* was refined and formalised into the *formal FA1.2 standard*, which conceptually splits in three parts:

1. an *idealised implementation*,
  2. a *behavioural specification* (how external entrypoints are wired to the functions), and
  3. a small *logical theory* of the idealised implementation and its behaviour, stating in particular that FA1.2-specified entrypoints do not change the total supply of tokens.
- Next we discuss the workflow of verifying a concrete implementation against the standard.

### 3 Per-implementation verification

We have verified three implementations as FA1.2 compliant (see below for what that means):

1. an implementation by camlCase in Morley (a Haskell eDSL for Michelson contracts),
2. an implementation by Edukera in Archetype (an integrated toolchain for specifying, implementing and verifying Tezos smart contracts), and
3. a liquidity ledger from the prototype Dexter 2 smart contract by the LIGO lang team, in CameLIGO (a language with ML-like syntax for Tezos smart contracts).

Concretely, verification proceeds as follows (we consider the camlCase contract):

1. The smart contract is compiled from some high-level smart contract language (Morley, Archetype, CameLIGO...), to a Michelson codestring – Michelson is the low-level stack-based native smart contracts language of the Tezos blockchain – and stored as a Coq string (see `fa12_camlcase_string.v`).
2. This Michelson code string is parsed to a term of Mi-Cho-Coq’s deep embedding of Michelson (see `fa12_camlcase.v`). This is a one-line operation; see Figure 6.
3. Details of the concrete implementation – how data is stored, any additional entrypoints and their behaviour – are packaged up, abstracted, and proved as high-level descriptions in Coq of behaviour (see `fa12_camlcase.v`).
4. Finally we prove that the high-level description of the implementation satisfies the formal FA1.2 specification (see `fa12_camlcase_verif.v`).

And thus we conclude that the contract is *FA1.2-compliant*.

Let’s unpack that. The sentence “*And thus we conclude that the contract is FA1.2-compliant*” is shorthand for a fuller statement that:

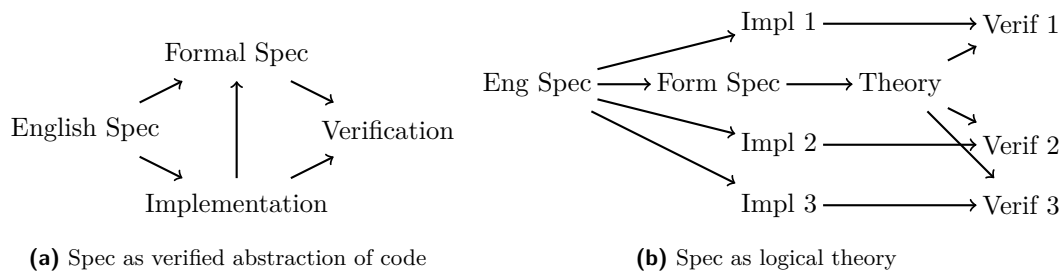
*A high-level Coq description of a Mi-Cho-Coq datum representing a Michelson code compilation of the original smart contract, satisfies a Coq formalisation of a refinement of the FA1.2 standard.*

Let’s unpack that further to spell out what parts of this are mathematically assured:

1. Refining the English FA1.2 standard to the formal FA1.2 standard is not mathematically assured. This was a creative human step of taking the FA1.2 English description and obtaining from it something formal in Coq that is more intensional, extensive, and precise than the English source, yet which we can still judge to be in some sense faithful to it.

<sup>10</sup>This is good, in the sense that the Michelson code is what gets executed on-chain. But note that the Michelson code may be compiler-dependent. Therefore when we say “We validated a contract” this *actually* means “We validated a Michelson compilation of that contract”.

<sup>11</sup>So “We validated one particular compilation to Michelson of that contract” actually means “We validated a Coq datum representing one particular compilation to Michelson of that contract”.



■ **Figure 8** Two workflow architectures.

2. The compilation of the smart contract from its original source code to Michelson is not assured, unless the compiler is verified in some way – which currently isn’t the case for Morley, Archetype and LIGO.<sup>12</sup> Note also that the Michelson code is what gets executed, which localises any subsequent validation to *that* compilation, and not some other compilation e.g. using a different compiler or a different version of that compiler.
3. We have to trust correctness of the transformation of the Michelson code string into Mi-Cho-Coq’s representation of Michelson code; and that Mi-Cho-Coq itself correctly captures the intended semantics of Michelson.
4. Everything else is rigorous, provided we trust the Coq kernel.

► **Remark 11.** There are (at least) two ways to use logic: to communicate meaning about specific objects (“this chair I’m sitting on, is black”), or to reason about (possibly empty) classes of things (“black chairs”; “the King of France”). Likewise there are two ways to view a formal specification: as a higher-level description of properties of a specific piece of code, or as a specification of properties which pieces of code in general may or may not have. This distinction matters for how we write and evaluate the usefulness of our verifications:

1. Figure 8a illustrates one verification workflow. A programmer reads a specification in English and writes an implementation (bottom left arrow). Then for additional assurance a formal spec is designed – in the light of the informal English spec and implementation (top left and middle arrows) – and the implementation is verified (two right-hand arrows). Here, *the formal spec is a verified abstraction of the code*.
2. Figure 8b illustrates our workflow. An English specification is refined to some Coq code (the *formal specification*) which entails via its definitions and axioms a collection of properties (*a theory*) against which implementations can be verified.<sup>13</sup> Here implementations are viewed as *models of the spec as a logical theory*. If the verifications fail that’s an error, and the specification, the implementation, or the theory are modified until they succeed.

In Figure 8a we see implementational choices and may even *want* to represent them in the abstract description (a concrete example is in Footnote 9, second paragraph). In Figure 8b we cannot see implementational choices when we design the abstract description and we do *not want* to. We will see how this *lack* of access will help us to detect ambiguities in the source English standard in Section 4.

<sup>12</sup>Morley is more of a macro language for Michelson, but it includes non-trivial transformations of the source code that are not yet proven to preserve semantics.

<sup>13</sup>This is a kind of dual to *program extraction*, where we start from a specification and extract an executable which then compiles to byte- or machine-code, which (if we trust our compilers) is correct by construction.

► **Remark 12.** We would argue that our workflow in Figure 8b is a natural way to structure verification against a standard, especially if we plan to verify more than one ledger. The specification-as-a-theory maximises modularity and reuse, minimises reinventing of the wheel, and accommodates both *a posteriori* and *a priori* reasoning:

- *A posteriori.* Write your smart contract in whatever language you prefer. Compile it to Michelson code as you would have to anyway; then (guided by the original source code) *rebuild* a certified correct high-level description of your contract in Coq, prove that the certified high-level description satisfies the formal standard, and that (the representation in Coq of) the compiled Michelson respects this description.
- *A priori.* Express a high-level design in Coq (or translate one into Coq). Prove it satisfies the formal standard, thus validating your design. Then implement this design in your language of choice, and verify that it respects the high-level description.

We would submit to the reader that this is reasonable and that most software development follows some mix of the two patterns above.

► **Example 13.** We continue Example 9. Two typical results in the per-implementation files, which exemplify the kind of results they contain, are that:

- Validity of storage is preserved by all entrypoints. This is a key sanity property which must include the five entrypoints mentioned in the FA1.2 standard (as listed in Subsection 1.2) but must also include any other operations offered by the smart contract.
- The total supply of tokens is correctly preserved (or updated, if tokens were minted or burned), and in particular that `%getTotalSupply` really does return the total supply. This is not entirely trivial because, for computational efficiency, most smart contracts track the total number of tokens separately from the tokens themselves.<sup>14</sup> Thus checking that `%getTotalSupply` returns the total supply requires us to write a predicate that computes the total supply, and verify that this “real” total supply is correctly tracked by whatever computationally efficient tally the smart contract is keeping.<sup>15</sup>

For scale, verification of the first property requires 115 lines of Coq code for the `camlCase` contract, 115 lines for the `Edukera` contract, and 139 for the `Dexter 2` contract (roughly half of which is boilerplate code).

## 4 Refining the FA1.2 standard

FA1.2 is underspecified by design, and often constructively so. For instance, `ETHtz`, `USDtz`, and `tzBTC` are all Tezos tokens (wrapping Ether, US Dollars, and Bitcoin respectively), and they may be FA1.2-compliant (or maybe not) – but clearly they are also different and special. Being FA1.2-compliant is just a property of a smart contract. In particular:

- The standard does not restrict the operations returned by the `%transfer` and `%approve` entrypoints. For instance, a contract may call another contract to access its ledger, e.g. if the ledger data is stored remotely.
- A contract may have more entrypoints than are mentioned in the standard, e.g. to mint and burn tokens.

However, it is also possible for FA1.2 to be underspecified in undesirable ways, and our verification effort uncovered two such issues, which were updated and corrected:

<sup>14</sup>An analogy: the Bank of England may keep track of how much cash is in circulation, but it would be computationally prohibitive to actually go out and count all the cash in the country.

<sup>15</sup>Another analogy: if the reader has ever lost money down the back of a sofa and then struggled (and perhaps failed) to find it again, they may appreciate that making sure that *absolutely no* tokens slip through *any* cracks, may require careful discipline. Somewhere in the first author’s childhood home there may still be a cheque for fifty pounds from his grandfather.

## 4.1 Issue 1: Self-transfer

When the from and to accounts in the `%transfer` entrypoint coincide, the operation can be treated either as a NOOP, or as a regular transfer (affecting allowances). The `camlCase` implementation originally chose the former; the Edukera and Dexter 2 implementations choose the latter.

It was agreed that this underspecification is undesirable and the FA1.2 standard was updated to require that this case be treated as a regular transfer. The `camlCase` implementation of the `%transfer` entrypoint was updated accordingly.

Note how this was noticed because we checked *more than one* ledger implementation against the *same* formal standard (cf. Remark 11 and comment 1 of Subsection 1.3).

## 4.2 Issue 2: passing tokens to a view entrypoint

As noted in Remark 2, when we call an entrypoint in Michelson we must pass it some (possibly zero) number of tez tokens. What should an entrypoint do if it gets passed tokens and does not need them? For instance, the entrypoint could be one of the so-called **view** entrypoints of FA1.2, `%getAllowance`, `%getBalance`, and `%getTotalSupply`.<sup>16</sup>

The `camlCase` and Edukera implementations opted to keep such tokens. Thus, if we called `camlCase` or Edukera implementation of `%getBalance` with some tokens, the contract would simply keep the tokens in its balance. We contacted the creators of the FA1.2 standard and they said this was undesirable: such tokens should be forwarded to the entrypoint's callback – i.e. a *passthrough*. The standard was updated to include this condition, and the implementations updated accordingly.

## 4.3 Summary of refinements

Thanks to this verification work the FA1.2 standard could be updated to eliminate two missed corner cases. The implementations were also updated as required.

Notably, the architecture of our verification (as discussed in Section 3) had a subtle but powerful effect on the errors that we could detect: because of how we factorised our verification files, and because (thanks to this factoring) we could consider *multiple* implementations uniformly against the *same* formal standard, it was easier to see where different implementations had made substantively divergent design decisions and to trace these decisions back to undesirable underspecifications in the core standard.

## 5 Related and future work

So far as we know there is nothing else in the literature quite like the FA1.2 formal standard and verifications reported on in this work. There have however been some other formalisation efforts in this field, notably: the ERC20 standard and its executable semantics in K; and a formalisation and verification of FA1.2 in Archetype by Edukera. We discuss each in turn:

### 5.1 ERC20-K

ERC20 is to Ethereum as FA1.2 is to Tezos (in fact, ERC20 came first and FA1.2 follows its example). ERC20 is a quite detailed API specification, but just like the FA1.2 standard, it is written in English, which is neither formal nor executable.

<sup>16</sup>An OO programmer would call the view entrypoints *getters*.

The ERC20-K semantics formalises ERC20 in K and annotates it with unit tests, with a particular focus on corner cases. As per the description:

ERC20-K is ... a formal executable semantics of a refinement of ... ERC20 [in] the K framework. ERC20-K clarifies [the precise meaning of] ERC20 functions [and] the corner cases that the ERC20 standard omits ... such as transfers from yourself to yourself or transfers that result in arithmetic overflows, [and we] manually ... produced ... a test-suite [of] tests which we believe cover all the corner cases.

In other words, ERC20-K turns the English API specification into a executable API specification in K, and provides a detailed test suite of sixty unit tests.

The ERC20-K homepage contains references to other work,<sup>17</sup> and the broad thrust of its argument is, just like ours, that a standard needs written in a *formal* language.

## 5.2 Archetype FA1.2 implementation and verification by Edukera

The company Edukera has a smart contracts language *Archetype*, in which they wrote a (short and succinct) implementation of an FA1.2-compliant smart contract. Included with the Archetype source code is a specification which asserts compliance with the FA1.2 standard.

In common with our work and with ERC20-K, the development argues for the need for a formal specification against which implementations can be checked.

The verification itself uses a Why3 library for Archetype that implements and specifies Archetype-specific abstractions. Half of this library is currently verified, which includes the parts that correspond directly to the FA1.2 smart contract, but not all of the libraries on which it depends.<sup>18</sup> Verification of the rest is a work in progress.

Archetype is an expressive environment in which a user can employ a single set of convenient high-level abstractions to specify and implement a contract, within a uniform and well-automated workflow.<sup>19</sup> Thus, the Edukera FA1.2 specification is a reflection of the FA1.2 standard into the Archetype toolstack, though as currently written it remains closely-tailored to the sole FA1.2 implementation which it has to talk about, namely the Edukera FA1.2 implementation in Archetype (e.g. if an implementation has additional mint or burn entrypoints, like the Dexter 2 contract, then it will not satisfy the Archetype specification's condition that the total supply is unchanged after each entrypoint).<sup>20</sup>

By design our work exists at a distance from any specific implementation and indeed from any specific source language, and it can be applied to any contract that can be compiled to Michelson, following a formal standard that does not require the smart contract programmer to buy in to any particular ecosystem except for Tezos itself. The correctness guarantee provided by compliance with our formal FA1.2 standard is correspondingly flexible and high-level, and our three verifications (including of the Archetype contract's compilation to Michelson) illustrate how this guarantee can be obtained as part of a practical workflow.

<sup>17</sup> Sadly no published academic work, but see a linear logic representation by one Rainy McRainface.

<sup>18</sup> Details in an Agora post; search for the section on Verification.

<sup>19</sup> As per the Archetype README, it provides a *single language to describe [a] business logic ... from which the different operational versions may be derived*.

<sup>20</sup> One could argue that the Archetype FA1.2 specification could be relaxed and the formal FA1.2 standard is also "just" a reflection of the FA1.2 standard into Coq. This is true but misses the point: it wasn't, because there was not any need, because the camlCase and Dexter 2 contracts do not exist in the Archetype implementation/specification ecosystem, because they are written in Morley and CameLIGO respectively. The point is not expressivity but scope: Archetype's uniformity and power are available *inside the Archetype toolstack*, whereas you can benefit from the formal FA1.2 standard using any toolstack – provided you add an entry for a Coq wizard to your budget. This is not either/or, but two complementary approaches in a rich design space.

## 5.3 Future work

### Extending to FA2

The third author is currently extending the development here to the FA2 standard, which is an update and extension of FA1.2 to allow, amongst other things, multiple token types.<sup>21</sup>

### Property-based testing

We argued in Remark 14 above, and in point 3 of Subsection 1.3, that proofs of FA1.2-compliance using our methodology are by construction comparable, because they are all Coq proofs of the same properties – namely, those stated in the formal FA1.2 standard.

This is true, but not the whole story: what if you have a program and you want to test it? Here, our development is of little direct help.

Contrast with the Edukera specification and ERC-20K, which come bundled with unit tests which are visibly more portable (we are not aware of the Edukera tests having been made available as a separate portable entity, but the test suite could presumably be ported).

It would be helpful for future work to extend the formal FA1.2 standard to a library of unit tests, or property-based testing properties, against which a prototype smart contract could be plugged, before going to the trouble of running the workflow described in Section 3.

### Accessibility

For sheer accessibility, the work in this paper falls far short of a tool like the ERC20 token verifier, which will test your bytecode online for compliance with the ERC20 token standard while U wait [5], subject to significant restrictions on the code.<sup>22</sup>

To the extent that these restrictions map from ERC20 to FA1.2, they do not apply to the work reported in this paper, and we see here the usual trade-off between ease-of-use and power (i.e. between price and performance). Which we prefer depends on our use case.

We could certainly envisage future work in which such a tool is created for FA1.2, based on a test-suite automatically derived from our Coq development. One could even imagine a tool which inputs a Coq specification (like the formal FA1.2 standard) and outputs an online test-suite, thus combining the rigour of Coq with the accessibility of an online testing suite. It is early days in this technology and there is much scope for innovation.

## 6 Conclusion

Having dependable token ledgers is absolutely necessary for the Tezos blockchain. Because of the blockchain's modular and updatable architecture, such ledgers are not primitive to the blockchain kernel, and therefore must be coded as smart contracts.<sup>23</sup>

Several ledger implementations exist, both live and deployed (ETHtz, USDtz, and tzBTC) and prototypical (camlCase, Edukera, and Dexter 2 by the LIGO lang team).

Smart contracts for ledgers are by design intended to handle real value – and once deployed they may be impossible to change or update. Users may lose money if mistakes are made, and also any failures may be perceived as reflecting poorly on the parent Tezos

<sup>21</sup> See Tezos Improvement Proposal 12 (TZIP-12) and the Tezos Developer Portal FA2 documentation.

<sup>22</sup> For instance: functions not in the ERC20 standard are ignored – which might sound innocuous but it is not, since without extra functions we might as well use a well-tested smart contract off-the-shelf. Similarly, the tool does not support external function calls or loops.

<sup>23</sup> This is just one small facet of the general fact that innovation in financial technology would benefit from any and all techniques to produce scalable, reliable smart contracts.

blockchain.<sup>24</sup> Therefore, the standards for safety and correctness for this class of program are exceedingly high, not only in the sense that the programs should be right, but also that what “being right” means should be described with clarity and precision.

In particular, it is in the blockchain’s best interests that validation of ledger implementations be made as modular as possible, so that proofs and proof-architectures can be reused and presented uniformly and reliably.

► **Remark 14.** Before this research, there was an English standard called “the FA1.2 standard”, and multiple implementations whose correctness was unknown. If they were verified (as is the case for the Edukera contract) there was still no way to systematically say what passing that verification meant compared e.g. against another verification by another team working to another interpretation of the English standard.

After this research, we have refined FA1.2 to a precise software artefact in Coq, and verified three implementations against this. Thus they are proven *correct*, in *the same way*, with respect to *the same notion of correctness*.

This development is visibly modular and systematic. Furthermore, the implementations and the standard have both been refined through the detection and elimination of some potentially dangerous corner cases. We think it can be considered a success.

We hope the ideas in this paper may serve as a model for future research and development.

---

## References

- 1 Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A Survey of Attacks on Ethereum Smart Contracts (SoK). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 164–186, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. doi:10.1007/978-3-662-54455-6\_8.
- 2 Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. Mi-Cho-Coq, a Framework for Certifying Tezos Smart Contracts. In *Formal Methods. FM 2019 International Workshops – Porto, Portugal, October 7–11, 2019, Revised Selected Papers, Part I*, volume 12232 of *Lecture Notes in Computer Science*, pages 368–379. Springer, 2019. doi:10.1007/978-3-030-54994-7\_28.
- 3 L.M. Goodman. Tezos – a self-amending crypto-ledger, 2014. URL: <https://tezos.com/whitepaper.pdf>.
- 4 Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC '18*, page 653–663, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3274694.3274743.
- 5 Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. A Formal Verification Tool for Ethereum VM Bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018*, page 912–915, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3236024.3264591.

---

<sup>24</sup> ... which may find itself blamed even if the smart contract was created by a third party.



# Using Coq to Enforce the Checks-Effects-Interactions Pattern in DeepSEA Smart Contracts

Daniel Britten ✉ 

The University of Waikato, Hamilton, New Zealand

Vilhelm Sjöberg ✉

CertiK, New York, NY, USA

Steve Reeves ✉ 

The University of Waikato, Hamilton, New Zealand

---

## Abstract

Using the DeepSEA system for smart contract proofs, this paper investigates how to use the Coq theorem prover to enforce that smart contracts follow the *Checks-Effects-Interactions Pattern*. This pattern is widely understood to mitigate the risks associated with reentrancy. The infamous “The DAO” exploit is an example of the risks of not following the *Checks-Effects-Interactions Pattern*. It resulted in the loss of over 50 million USD and involved reentrancy – the exploit used would not have been possible if the *Checks-Effects-Interactions Pattern* had been followed.

Remix IDE, for example, already has a tool to check that the *Checks-Effects-Interactions Pattern* has been followed as part of the Solidity Static Analysis module which is available as a plugin. However, aside from simply replicating the Remix IDE feature, implementing a *Checks-Effects-Interactions Pattern* checker in the proof assistant Coq also allows us to use the proofs, which are generated in the process, in other proofs related to the smart contract.

As an example of this, we will demonstrate an idea for how the modelling of Ether transfer can be simplified by using automatically generated proofs of the property that each smart contract function will call the Ether transfer method at most once (excluding any calls related to invoking other smart contracts). This property is a consequence of following a strict version of the *Checks-Effects-Interactions Pattern* as given in this paper.

**2012 ACM Subject Classification** Security and privacy → Logic and verification; Computer systems organization → Distributed architectures

**Keywords and phrases** smart contracts, formal methods, blockchain

**Digital Object Identifier** 10.4230/OASICS.FMBC.2021.3

**Category** Short Paper

**Supplementary Material** *Software (Source Code)*: <https://github.com/Coda-Coda/deepsea-1/tree/fmbc-2021>; archived at `swh:1:dir:85ea91f0b51380b40bf760195c03a5564d195993`

**Acknowledgements** I (Daniel Britten) want to thank the University of Auckland and Associate Professor Jing Sun for kindly hosting me during this research.

## 1 Introduction

The importance of smart contracts being correct has been voiced many times, most obviously because of the high financial risk associated with a smart contract being incorrect and exploited (such as “The DAO” [8] and others [1, 7, 9]) which all involved the use of what we will refer to as *malicious reentrancy*.

Reentrancy involves a smart contract  $C$  that triggers the execution of code of another smart contract  $D$  which then calls a function in the original smart contract  $C$  before the original execution of  $C$  has completed. However, when not handled properly, reentrancy can



© Daniel Britten, Vilhelm Sjöberg, and Steve Reeves;  
licensed under Creative Commons License CC-BY 4.0

3rd International Workshop on Formal Methods for Blockchains (FMBC 2021).

Editors: Bruno Bernardo and Diego Marmosoler; Article No. 3; pp. 3:1–3:8

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

### 3:2 Enforcing Checks-Effects-Interactions in DeepSEA

cause a smart contract to behave incorrectly and be exploited. This happens with malicious reentrancy, which maliciously exploits the situation that the original execution of  $C$  has not completed.

This issue can be mitigated by following the *Checks-Effects-Interactions Pattern* which suggests that a smart contract should first do the relevant *Checks*, then make the relevant internal changes to its state (*Effects*), and only then interact with other smart contracts which may well be malicious. When following the *Checks-Effects-Interactions Pattern* a reentrant call is essentially no different to a call that is initiated after the first call is finished so no additional risk from malicious reentrant calls is possible.

On the Ethereum blockchain, interacting with a malicious smart contract is even possible when transferring Ether. This is because if the recipient is a smart contract then it has the opportunity to run some code on receiving funds.

The problem with all this is that the modelling of smart contract execution when there is the possibility of reentrancy is difficult and the related correctness proofs would be complex as well. Even modelling the humble Ether transfer needs to take the possibility of reentrancy into account.

Using the DeepSEA [2] system for proofs about smart contract correctness, a method of enforcing the *Checks-Effects-Interactions Pattern* has been developed. Enforcing the *Checks-Effects-Interactions Pattern* greatly simplifies the modelling of any action that might involve external calls (including Ether transfers).

Tangibly, enforcing the *Checks-Effects-Interactions Pattern* means that the DeepSEA code for a smart contract function shown on the left (Listing 1) should not be permitted and the code shown on the right (Listing 2) should be allowed.

■ **Listing 1** “Unsafe” function.

```
let unsafeExample() =
  transferEth(msg_sender, 0u42);
  transferSuccessful := true
```

■ **Listing 2** “Safe” function.

```
let safeExample() =
  transferSuccessful := true;
  transferEth(msg_sender, 0u42)
```

The end result of the work in this paper is a system which automatically proves that the *Checks-Effects-Interactions Pattern* has been followed for most cases when it indeed has been, although there are some cases where the *Checks-Effects-Interactions Pattern* has been followed but this system cannot prove it, as a compromise for automation. A related result is then used to demonstrate an idea for simplifying the modelling of Ether transfers.

The main contributions of this paper are as follows:

- A Coq [3] proposition formalising the notion of a smart contract function following the *Checks-Effects-Interactions Pattern*. This is discussed in Subsection 2.4.
- Automated proofs related to the previous contribution as well as related automated proofs that the lists of transfers (that are directly generated by the smart contract) after function calls are of length at most one. See Section 3 and Section 4 respectively.
- A demonstration of an idea for simplifying the modelling of what states are reachable by a smart contract by making use of some of these automated proofs (Section 4).

## 2 Representing the absence of reentrancy situations as a proof goal

### 2.1 The DeepSEA system

All the modelling and proofs in the paper make use of the DeepSEA system for smart contract proofs. DeepSEA [2] is an up and coming framework and smart contract language that promises to provably link high-level specifications in Coq [3] to Ethereum Virtual Machine (EVM) bytecode. This will give a high degree of certainty that results proven about the high-level specifications also hold for the bytecode. The DeepSEA compiler is based upon the CompCert verified compiler [5].

### 2.2 The *Checks-Effects-Interactions Pattern*

The *Checks-Effects-Interactions Pattern* suggests that a smart contract should follow a pattern in which calls to external contracts are always the last step [12].

When following the *CEIP*, nested calls are equivalent to calls invoked one after another as nested calls cannot influence the outcome of the original call (excluding considering gas). This enables a simpler model than completely modelling reentrancy with co-recursive functions. The simpler model is considered to be equivalent to a complete model in terms of modelling what states are reachable and we rely on an informal knowledge for this. Ideally, we would like to model reentrancy foundationally making use of the EVM semantics and then prove that the simple model is equivalent to the more complex model in the case where the *CEIP* is followed.



In this paper, a stricter version of the *Checks-Effects-Interactions Pattern* is used where only one *Interaction* is permitted. This eliminates modelling complications in the situations where two external calls are done but the first one turns out to throw an error. It is virtually impossible to know, when modelling, whether an arbitrary external call will throw an error, particularly due to the possibility of gas being exhausted.

This strict version of the *Checks-Effects-Interactions Pattern* will now simply be referred to as the *CEIP*.

### 2.3 Relevant aspects of the DeepSEA system

Listing 3 shows the same DeepSEA smart contract function in different representations. The intermediate level and high level representation are both generated automatically from the DeepSEA source. First, the intermediate level abstract syntax tree in Coq is generated from the source. The denotational semantics of the AST gives the high level representation (by the `synth_stmt_spec_opt` Coq function as a part of the DeepSEA system). The AST for each function contains the relevant information required to formulate the notion of whether the function adheres to the *CEIP*. The inductive proposition described in the next section makes use of the intermediate level AST representation.


### 2.4 Coq Inductive Proposition: `cmd_constr_CEI_pattern_prf`



The typing rule (Figure 1) corresponds to the definition of `cmd_constr_CEI_pattern_prf` which is an inductive proposition in Coq capturing the notion of a function following the *CEIP*. The typing rule is based upon the syntax of the smart contract as represented in the DeepSEA intermediate level language. It uses the assumption that reentrancy is only possible when certain syntax, such as `Ctransfer` is encountered. `Ctransfer` is the intermediate level language construct corresponding to a `transferEth` call. The  icon indicates that the contract cannot in any way have triggered reentrancy yet and the  icon indicates that



### 3:4 Enforcing Checks-Effects-Interactions in DeepSEA

■ **Listing 3** “Safe” function in different representations with similarities highlighted.

```
DeepSEA smart contract source code (not Coq):
let safeExample() =
  transferSuccessful := true ;
  transferEth(msg_sender, Ou 42)
DeepSEA intermediate level language in Coq:
(CC sequence
(CCstore
  (LCvar Contract_transferSuccessful := true_var)
  (Econst_int256 tint_bool true Int256.one))
(CC transfer
  (@ECbuiltin0 _ _ builtin0_caller_impl)
  (Econst_int256 tint_U (Int256.repr 42_var))
  (Int256.repr 42))))
DeepSEA high level language in Coq:
(get;;
MonadState.modify (update_Contract_transferSuccessful true)) ;;
d <- get;;
(let (success, d') :=
me_transfer me (me_caller me) (Int256.repr 42) d in
if Int256.eq success Int256.one then put d' else mzero)
```

reentrancy may have been triggered by that point (and so no unsafe commands such as writing to storage should be allowed after that point). The  icon would indicate a contract that is vulnerable to malicious reentrancy but does not occur in the typing rule as the rule defines what is safe.

The transfer related rule in Coq is shown in Listing 4. The notion that at most one external call is allowed is captured by the fact that the proof requires the state `Safe_no_reentrancy` () beforehand. Due to the transfer the contract is then in a state where reentrancy may have occurred and this is captured by the state `Safe_with_potential_reentrancy` (.

In Listing 5 we define that if the body of a for loop stays at state  $\rho$  (either  or ) then the for loop as a whole is also defined to stay at state  $\rho$ .

The remaining definitions are available in the GitHub repository<sup>1</sup>. This defines what it means for a DeepSEA smart contract function to follow the *CEIP*. To be precise, if `cmd_constr_CEI_pattern_prf` can be proven for a given function then that function follows the *CEIP*.

A drawback of this formulation is that interrelated if statements are not able to be reasoned about. If the logical content of interrelated if statements made it possible to know the *CEIP* was indeed followed, this formulation would not allow those functions to be proved to be safe. This does however simplify proof automation. An alternative approach which made use of the high level representation of the smart contract was also explored. This “instrumented semantics” approach added reentrancy state tracking to the semantics of DeepSEA, and as a result *is* able to reason about interrelated if statements. This alternative approach still assumes specific syntactic elements correspond to the possibility of causing reentrancy.

<sup>1</sup> <https://github.com/Coda-Coda/deepsea-1/tree/fmbc-2021> – See README for the specific files relevant to this paper.

■ **Figure 1** Typing rule for a command that adheres to the *CEIP*, corresponding to the Coq inductive proposition `cmd_constr_CEI_pattern_prf`. (Some rarely used rules have been omitted).  $\rho_x \in \{\bullet, \circ\}$ .

$$\begin{array}{c}
\frac{}{\{\rho\} \text{ skip } \{\rho\}} \quad \frac{\{\rho_1\} c_1 \{\rho_2\} \quad \{\rho_2\} c_2 \{\rho_3\}}{\{\rho_1\} \text{ let } x = c_1 \text{ in } c_2 \{\rho_3\}} \quad \frac{}{\{\bullet\} \text{ load } \{\bullet\}} \quad \frac{}{\{\bullet\} e_1 := e_2 \{\bullet\}} \\
\\
\frac{\{\rho_1\} c_1 \{\rho_2\} \quad \{\rho_2\} c_2 \{\rho_3\}}{\{\rho_1\} c_1 ; c_2 \{\rho_3\}} \quad \frac{\{\bullet\} c_{true} \{\bullet\} \quad \{\bullet\} c_{false} \{\bullet\}}{\{\bullet\} \text{ if } e_1 \text{ then } c_{true} \text{ else } c_{false} \{\bullet\}} \\
\\
\frac{\{\bullet\} c_{true} \{\circ\} \quad \{\bullet\} c_{false} \{\bullet\}}{\{\bullet\} \text{ if } e_1 \text{ then } c_{true} \text{ else } c_{false} \{\circ\}} \quad \frac{\{\bullet\} c_{true} \{\bullet\} \quad \{\bullet\} c_{false} \{\circ\}}{\{\bullet\} \text{ if } e_1 \text{ then } c_{true} \text{ else } c_{false} \{\circ\}} \\
\\
\frac{\{\bullet\} c_{true} \{\circ\} \quad \{\bullet\} c_{false} \{\circ\}}{\{\bullet\} \text{ if } e_1 \text{ then } c_{true} \text{ else } c_{false} \{\circ\}} \quad \frac{\{\circ\} c_{true} \{\circ\} \quad \{\circ\} c_{false} \{\circ\}}{\{\circ\} \text{ if } e_1 \text{ then } c_{true} \text{ else } c_{false} \{\circ\}} \\
\\
\frac{\{\rho\} c \{\rho\}}{\{\rho\} \text{ for } e_1 \text{ to } e_2 \text{ do } c \{\rho\}} \quad \frac{\{\rho_1\} \text{ function } \{\rho_2\}}{\{\rho_1\} \text{ function call } \{\rho_2\}} \\
\\
\frac{}{\{\bullet\} \text{ transferEth}(e_1, e_2) \{\circ\}} \quad \frac{\{\rho\} c \{\rho\}}{\{\rho\} \text{ assert } c \{\rho\}} \quad \frac{\{\rho\} c \{\rho\}}{\{\rho\} \text{ deny } c \{\rho\}}
\end{array}$$

■ **Listing 4** Defining *CEIP* adherence for `CCTransfer`.

```

| CCCEIPtransfer :
forall e1 e2,
cmd_constr_CEI_pattern_prf
_ (* Infer the return type *)
Safe_no_reentrancy (* ● *)
(CCtransfer e1 e2) (* Typically related to a 'transferEth' call. *)
Safe_with_potential_reentrancy (* ○ *)
(* After, the possibility of reentrancy is noted. *)

```

■ **Listing 5** Defining *CEIP* adherence for `CCFor`.

```

| CCCEIPfor :
forall {ρ} id_it id_end e1 e2 c,
cmd_constr_CEI_pattern_prf _ ρ c ρ
(* Given a command that stays at state ρ *)
-> cmd_constr_CEI_pattern_prf _ ρ (CCfor id_it id_end e1 e2 c) ρ
(* Then the for loop as a whole stays at state ρ *)

```

■ **Listing 6** Coq tactic to prove adherence to the *CEIP*.

```
Ltac CEI_auto :=
  repeat (
    reflexivity
  + typeclasses eauto
  + eapply CCCEIPskip + eapply CCCEIPlet + eapply CCCEIPload
  + eapply CCCEIPfor + eapply CCCEIPtransfer + ... ).
```

Another drawback (with both approaches) is that other techniques to manage reentrancy issues such as locks are not considered to be safe by these methods, even when they may have been used in a way which is safe. On the other hand, this does simplify modelling by only needing to consider cases equivalent to when no reentrancy occurs.

### 3 Automatically proving the absence of reentrancy situations

Now that we have defined the notion of a smart contract following the *CEIP* the goal is to automatically prove this for every function that does indeed follow the *CEIP* (or at least, most). The automation will be carried out by Coq tactics.

The tactic, partially shown in Listing 6, will repeatedly apply the constructors from the `cmd_constr_CEI_pattern_prf` definition along with resolving certain typeclass goals automatically. The `+` used to combine the tactics is critical to ensure the tactic backtracks as necessary because sometimes it is not the first matching constructor that is relevant.

See GitHub<sup>2</sup> for the full definitions of all the tactics involved. The proofs are done automatically and provide the user with an error if they fail (which would likely indicate the *CEIP* was not followed).

### 4 Simplifying the modelling of Ether transfer

The fact that we are following the *CEIP* simplifies the modelling of Ether transfer due to the fact that nested calls can be considered to be called one after another as no nested calls can influence the outcome of the original call (excluding gas considerations), as discussed in Section 2.2. This means that when considering what states are reachable it is sound to treat the transfer as only affecting Ether balances and ignore any other potential state changes. Also, since we are following the strict version of the *CEIP* we know that there is at most one call to `transferEth` which further simplifies the modelling.

When modelling Ether transfer in DeepSEA, at the end of a smart contract function call a list of transfers is produced and the modelled overall balances need to be updated based upon that list. If the list contains more than one element, how the balances should be updated is unclear due to the possibility of reentrancy having occurred. This is where a proof that only one transfer at most was directly generated is particularly useful. Coq allows us to pass this proof as an argument to our definition and use it to discharge the case where the list is longer than one element, as shown in Listing 7. This is greatly useful for simplifying the modelling by allowing us to demonstrate to Coq that we do not need to model reentrancy related to multiple transfers. If we did not have the proof we would be stuck

<sup>2</sup> <https://github.com/Coda-Coda/deepsea-1/tree/fmbc-2021> – See README for the specific files relevant to this paper.

■ **Listing 7** Updating balances for a list of length at most one.

```

Program Definition update_balances_from_transfer_list transfers
  ( length_evidence : length transfers <= 1 ) previous_balances a :=
match transfers with
| [] => previous_balances a
| [t] => update_balances_from_single_transfer contract_address
      (recipient t) (amount t) previous_balances a
| (h :: i :: t) as l => _ (* Coq allows us to discharge this case. *)
end.
Next Obligation. (* This is the case where transfers = h :: i :: t *)
intros.
exfalso. (* There is an impossible situation. *)
rewrite <- Heq_transfers in length_evidence. simpl in length_evidence. lia.
Defined.

```

with either truncating the list (which would be inaccurate) or assuming all the transfers took place with no reentrancy (which would also be inaccurate and leave the supposedly proven correct contract open to potential malicious reentrancy).

The relevant proofs that each smart contract function directly generates at most one transfer are similar to the proofs about the *CEIP* being followed in the sense that the DeepSEA `inv_runStateT_branching` tactic considers all branches of code execution like done by the `CEI_auto` tactic (Listing 6).

This technique simplifies the modelling of Ether transfer without leaving the door open for malicious reentrancy. The proofs are automated, only requiring the DeepSEA smart contract programmer to follow the strict version of the *CEIP*.

## 5 Related Work

A number of other tools aim to tackle the problem of reentrancy, such as [4, 6, 10] and [11]. This work is unique in that it explicitly makes use of proofs related to the *CEIP* in simplifying modelling smart contracts. It also is a step towards a smart contract proof system that uniquely targets the EVM as well as allowing proofs to be done on a high-level representation of the smart contract with strong guarantees that the properties proven about the high-level representation will also apply to the EVM bytecode.

## 6 Conclusion

This paper discusses an approach for representing and automatically proving that DeepSEA smart contracts follow the *CEIP* (code available on GitHub<sup>3</sup>). This is demonstrated by defining an inductive proposition in Coq that states that a particular smart contract function follows the *CEIP*. A proof that each smart contract function calls the Ether transfer function at most once is also discussed. An application of these proofs to simplify the modelling of Ether transfer is then discussed.

<sup>3</sup> <https://github.com/Coda-Coda/deepsea-1/tree/fmbc-2021> – See README for the specific files relevant to this paper.

---

**References**

---

- 1 Lucas Campbell. DeFi platform dForce hacked for \$25m - ERC777 reentrancy attack. <https://defirate.com/dforce-hack/>. (Accessed on 23 May 2021).
- 2 CertiK Foundation. DeepSEA. <https://github.com/certikfoundation/deepsea>. (Accessed on 23 May 2021).
- 3 The Coq Development Team. The Coq proof assistant. <https://coq.inria.fr/>. (Accessed on 23 May 2021).
- 4 Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE, 2019.
- 5 Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert – a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016. URL: <https://hal.inria.fr/hal-01238879>.
- 6 Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.
- 7 Alex Manuskin. Living in a lego house: The imBTC DeFi hack explained. <https://www.zengo.com/imbtc-defi-hack-explained/>. (Accessed on 23 May 2021).
- 8 Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M. Kim, and Marek Laskowski. Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack. *Journal of Cases on Information Technology*, 21(1):19–32, 2019.
- 9 David Oz Kashi. The reentrancy strikes again – The case of Lendf.Me. <https://valid.network/post/the-reentrancy-strikes-again-the-case-of-lendf-me>. (Accessed on 23 May 2021).
- 10 Remix. Remix – Ethereum IDE. <https://remix.ethereum.org/>. (Accessed on 23 May 2021).
- 11 Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with Scilla. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.
- 12 Maximilian Wohrer and Uwe Zdun. Smart contracts: security patterns in the Ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8. IEEE, 2018.



# Formally Documenting Tenderbake

**Sylvain Conchon**

Nomadic Labs, Paris, France

**Alexandrina Korneva**

Université Paris-Saclay, ENS Paris-Saclay, CNRS, LMF, 91190, Gif-sur-Yvette, France

**Çagdas Bozman**

Functori, Paris, France

**Mohamed Iguernlala**

Functori, Paris, France

**Alain Mebsout**

Functori, Paris, France

---

## Abstract

In this paper, we propose a formal documentation of Tenderbake, the new Tezos consensus algorithm, slated to replace the current Emmy family algorithms. The algorithm is broken down to its essentials and represented as an automaton. The automaton models the various aspects of the algorithm: (i) the individual participant, referred to as a baker, (ii) how bakers communicate over the network (the mempool) and (iii) the overall network the bakers operate in. We also present a TLA+ implementation, which has proven to be useful for reasoning about this automaton and refining our documentation. The main goal of this work is to serve as a formal foundation for extracting intricate test scenarios and verifying invariants that Tenderbake's implementation should satisfy.

**2012 ACM Subject Classification** Software and its engineering → Formal methods

**Keywords and phrases** Consensus algorithm, Tezos blockchain, TLA+

**Digital Object Identifier** 10.4230/OASICS.FMBC.2021.4

**Category** Short Paper

**Supplementary Material** *Model (Source Code)*: <https://www.lri.fr/~conchon/tenderbake/Tenderbake.tla>

## 1 Introduction

Tenderbake is a new consensus algorithm designed by Nomadic Labs for the Tezos blockchain [5]. Tenderbake participates in the blockchain protocol to ensure that all peers reach agreement on the state of the distributed ledger. Essentially, the algorithm ensures that all participants record the same blocks, in the same order, in their local copy of the blockchain.

Like Tezos's current Emmy family protocols, Tenderbake is a Byzantine Fault-Tolerant (BFT) algorithm that can tolerate (a limited number of) malicious machine failures on an asynchronous network. The main advantage of Tenderbake is related to block finality, *i.e.*, the point at which the parties involved can consider the consensus on adding a block to be complete. More precisely, this is the moment when it becomes impossible to go back or modify a block that has been added to the blockchain. Unlike the probabilistic finality of Emmy algorithms, where the probability that a block will eventually belong to the blockchain increases with the number of blocks added in front of it, Tenderbake allows for an almost immediate finality: a block is considered to belong to the chain when only two blocks are added after it. This new consensus algorithm technology is inspired by PBFT (practical Byzantine Fault-Tolerant) protocols [4] like Tendermint [1, 3] in the Cosmos project [6].



© Sylvain Conchon, Alexandrina Korneva, Çagdas Bozman, Mohamed Iguernlala, and Alain Mebsout; licensed under Creative Commons License CC-BY 4.0

3rd International Workshop on Formal Methods for Blockchains (FMBC 2021).

Editors: Bruno Bernardo and Diego Marmosoler; Article No. 4; pp. 4:1–4:9

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

To achieve such a finality result, Tenderbake implements a three-phase PBFT protocol: a *proposal* phase where a single participant (called *baker*) proposes a new block, and two successive *voting* phases (called *preendorsement* and *endorsement*) at the end of which a quorum of votes must be reached on the proposed block. If a consensus is reached, each participant adds the proposed block locally to their blockchain and a new instance of the algorithm can then start for the next block (referred to as the *next level* in Tezos). However, this idyllic scenario can fail for many reasons. For example, Byzantine participants can inject fake blocks or fake votes. The consensus can also fail even in the absence of participant failure because blocks and votes, which are sent as messages, can be arbitrarily delayed or lost by the network. In this case, a new round of proposals/votes is launched, possibly with a new block issued by another participant.

Tenderbake implements several mechanisms to avoid Byzantine attacks or asynchrony-related problems to guarantee the correctness of the consensus. For instance, a synchronization mechanism is required for each participant to decide that a round of proposals/votes is over. For this purpose, Tenderbake implements a partially synchronous system, where participants synchronize without exchanging messages, by exploiting their internal clocks and the information stored in the blockchain. As another example, cryptographic certificates about the (pre)endorsing majority are injected into blocks to prevent Byzantine attacks.

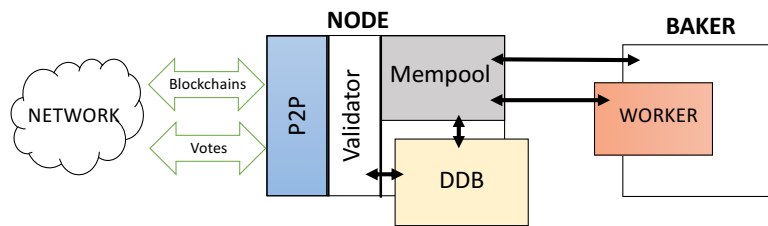
Designing and implementing a consensus algorithm like Tenderbake is notoriously challenging. While a very precise proof-and-paper description of this algorithm has been given in [2], we propose in this paper a TLA<sup>+</sup> modeling of Tenderbake. To do this, we break down the algorithm to its essentials and represent bakers' roles as an automaton. We also abstract the notion of time, but retain a synchronization mechanism that allows the drift of participants' clocks to be simulated. We do not sacrifice any of the more subtle features of Tenderbake's implementation, like how the protocol is handled by both the mempool (a more sophisticated gossip layer) and the bakers themselves.

The main goal of our work is to provide a formal executable documentation of Tenderbake that will serve as a basis for extracting complex test scenarios and invariants that the Tenderbake implementation must satisfy. So far, our TLA<sup>+</sup> automaton has proven useful for reasoning and exchanging with the developers of the actual implementation. The TLA<sup>+</sup> model is available at <https://www.lri.fr/~conchon/tenderbake/>.

## 2 Tezos Architecture

Tezos forms a Peer-to-peer network in which peers, called *nodes*, are interconnected and communicate by message passing. Nodes implement the core algorithms and data structures of the blockchain. They are composed of a Peer-to-peer layer (P2P), validators (which use the rules of the economic protocol to check blocks and operations), a distributed database (DDB), and a specific data structure for pending operations, called the *Mempool*.

Nodes continuously run a gossip protocol to communicate and exchange blockchains (complete or just head blocks) with each other. Each node maintains in the Mempool the best version of the blockchain that it has received. Nodes do not communicate plain messages directly, but only a hash value of them. When a node receives a hash, it checks if this value is already stored in its DDB before saving it. The role of the DDB is to maintain a correspondance between hash keys and the plain values associated with them. For that, as a parallel task, the DDB fetches data (of which only the hash is known) from the node's peers, and, conversely, responds to similar peers' requests by providing them with the requested data. When the DDB gets a response, it transmits to the Mempool the plaintext values that correspond to blocks, transactions, or votes.



■ **Figure 1** Tezos general architecture.

In this architecture, shown in Figure 1, bakers are not directly visible on the network. For security reasons, they only communicate with each other through the nodes they are connected to (which we refer as *the node of the baker*). The role of a baker is to produce proposal blocks and to vote for the head blocks of the blockchain stored in its node’s Mempool. For that, a baker gets the first two blocks of the blockchain from the Mempool (via a Remote Procedure Call mechanism – RPC) and it implements the consensus rounds of Tenderbake to decide whether to vote on the current head or not. A baker is also composed of a worker running in parallel, whose role consists of getting the votes from the Mempool (via RPC) and checking for potential quorums.

This modular, secure and highly parallel architecture raises several issues when implementing a PBFT algorithm like Tenderbake. First, while a Baker is voting on a specific blockchain head, the Mempool can receive a new proposal and decide to change its head. This means that everything needs to be resynchronized for the baker and the worker to vote or get a quorum on the current head. Secondly, Tezos has been designed to be agnostic to the consensus algorithm used to produce blocks. As a consequence, the rules of the Tenderbake algorithm are abstract, so it is important to make sure that the Mempool has access to all necessary information needed to choose the best blockchain. Last, bakers combine timestamp information stored in the blocks and their current clock to know how long before a round timeout is triggered. Since each baker has their own clock, this can lead to clock drift, to which the protocol must be resistant.

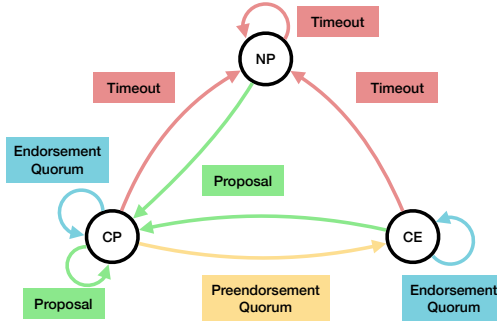
Finally, the communication mechanism between components involves RPC (Worker/Mempool and Baker/Mempool) and streams of events (Worker/Baker). To simplify our modeling, we approximate these communications through a shared memory mechanism and leave the modeling of a communication layer closer to the implementation to future work.

### 3 Tenderbake Automaton

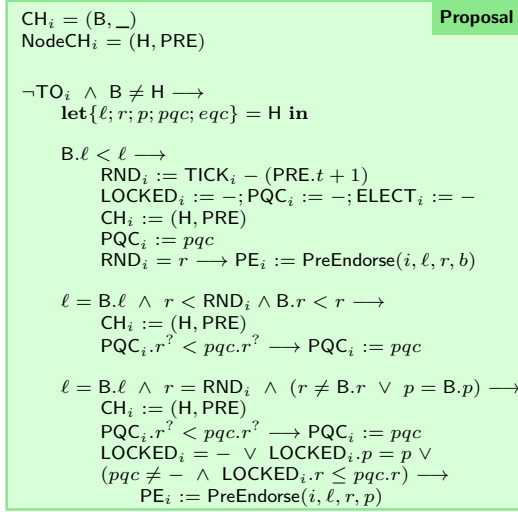
In this section, we describe the Tenderbake consensus formally, for a set of participants BAKERS. Contrary to the implementation in Tezos, where participants change at each level, we assume that this set is fixed. Each individual participant (baker) runs the same automaton. We explain how this automaton is implemented in TLA<sup>+</sup> in Section 4.

The automaton is given in Figure 2. It represents the evolution of a baker’s state and the actions performed by this baker in the three possible consensus phases. In the rest of this section, we give a description of the local state maintained by an arbitrary baker  $i$  and we detail the transitions of this automaton using a rudimentary guarded command language.

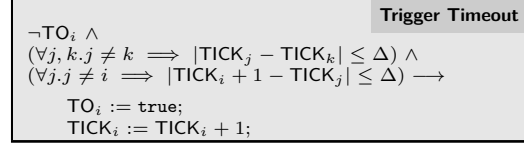
**Notations.** By convention, the internal variables of the baker  $i$  are denoted by capital letters associated with an index  $i$ . Thus,  $X_i$  represents the internal variable  $X$  of  $i$ . We use lowercase letters for parameters. Certain variables are *option variables*, meaning that



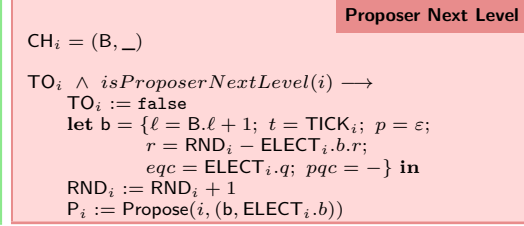
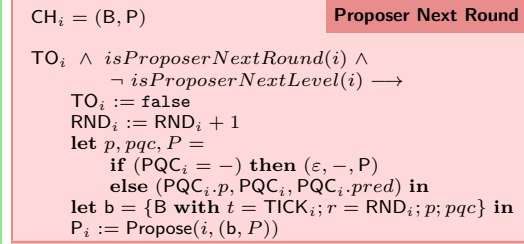
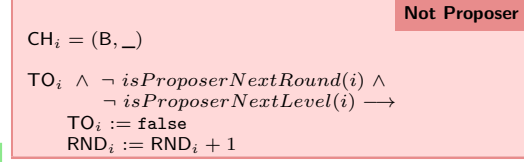
■ Figure 2 Tenderbake automaton.



■ Figure 3 Receiving a proposal.



■ Figure 4 Trigger timeout oracle.



■ Figure 5 A baker's possible actions once timeout has been reset.

they can have a value or not. Not having a value is denoted by the symbol  $-$ . When comparing variables,  $X^?$  means that  $X$  is an option variable and can therefore be empty. By convention, empty variables are (strictly) less than non-empty variables. We stick to conventional message passing notation where  $m(x_1, \dots, x_k)^?$  stands for the reception of a message  $m$  with parameters  $x_1, \dots, x_k$ , and  $m(v_1, \dots, v_k)!$  is the asynchronous broadcast of  $m$  with  $v_1, \dots, v_k$  as arguments. Note that when a baker broadcasts a message, he does not send it to himself.

**Baker's state.** As shown in Figure 2, our automaton has three distinct states, which correspond to the possible phases of the consensus algorithm: NP for *Non Proposer*, CP for *Collecting Preendorsements*, and CE for *Collecting Endorsements*. In addition to this control flow information, a baker  $i$  maintains a copy of the blockchain in a variable  $CH_i$ . Since only the two head blocks of the blockchain are needed for the consensus algorithm,  $CH_i$  contains a pair of blocks  $(B, P)$ , where  $B$  is the head block of the blockchain and  $P$  its predecessor. A block is represented by a record  $\{\ell; r; t; p; eqc; pqc\}$ , where each component is accessible via the standard record access notation (e.g.  $B.r$ ). The role of each of these components is summarized in Figure 6.

In addition to the two head blocks stored in  $CH_i$ , a baker maintains his current consensus round in  $RND_i$ . For safety reasons, a baker must also keep track of the block he voted for, in variable  $LOCKED_i$  and for which a preendorsement voting quorum was observed.

$\ell$  level of the block in the blockchain;  
 $r$  consensus round during which the block was proposed;  
 $t$  timestamp of when the block was proposed;  
 $p$  block's payload - i.e. contents without consensus operations;  
 $pqc$  preendorsing majority certificate with the round when it was observed;  
 $eqc$  endorsing majority certificate for the previous block.

■ **Figure 6** Block structure.

Initial state for Baker $i$	Init. state for Mempool
$CH_i = (G, G)$	$NodeCH_i = (G, G)$
$RND_i = 0$	$M_i = \emptyset$
$TICK_i = 1$	$P_i = -$
$LOCKED_i = Genesis$	$E_i = -$
$PQC_i = \{p = []; q = \emptyset; r = 0\}$	$PE_i = -$
$ELECT_i = \{b = G; q = \emptyset\}$	
$TO_i = true$	

where  $G = \{\ell = 0; r = 0; t = 0; p = []; pqc = -; eqc = \emptyset\}$

■ **Figure 7** Initial states.

NodeCH $_i = (H, PRE)$	Mempool
$PreEndorse(j, \ell, r, p)? \vee PE_i = PreEndorse(j, \ell, r, p) \longrightarrow$ $M_i := M_i \cup \{PreEndorse(j, \ell, r, p)\}$ $PE_i \neq - \longrightarrow$ $PreEndorse(i, \ell, r, p)!$ $PE_i := -$	
$Endorse(j, \ell, r, p)? \vee E_i = Endorse(j, \ell, r, p) \longrightarrow$ $M_i := M_i \cup \{Endorse(j, \ell, r, p)\}$ $E_i \neq - \longrightarrow$ $Endorse(i, \ell, r, p)!$ $E_i := -$	
$Propose(j, (h, pre))? \vee P_i = Propose(j, (h, pre)) \longrightarrow$ <b>let</b> $\{\ell; r; pqc; eqc\} = h$ <b>in</b> $isProposer(j, \ell, r) \wedge$ $(H.\ell, H.pqc.r, -PRE.r, H.r) < (\ell, pqc.r, -pre.r, r) \wedge$ $valid\_eqc(eqc, pre) \wedge$ $(pqc = - \vee valid\_pqc(h)) \longrightarrow$ $NodeCH_i := (h, pre)$ $P_i \neq - \longrightarrow$ $Propose(i, (h, pre))!$ $P_i := -$	

■ **Figure 9** Mempool transitions.

$CH_i = (B, \_)$	Preendorsement Quorum
<b>let</b> $q = \{m \in M_i \mid m = PreEndorse(\_, \ell, r, p) \wedge$ $\ell = B.\ell \wedge p = B.p \wedge$ $r = RND_i = B.r\}$ <b>in</b> $\neg TO_i \wedge quorum(q) \wedge LOCKED_i \neq B \longrightarrow$ $PQC_i := \{p = B.p; r = B.r; q = q\}$ $LOCKED_i := B$ $E_i := Endorse(i, B.\ell, B.r, B.p)$	
$CH_i = (B, \_)$	Endorsement Quorum
<b>let</b> $q = \{m \in M_i \mid m = Endorse(\_, \ell, r, p) \wedge$ $\ell = B.\ell \wedge p = B.p \wedge$ $r = RND_i = B.r\}$ <b>in</b> $\neg TO_i \wedge quorum(q) \wedge ELECT_i = - \longrightarrow$ $ELECT_i := \{b = B; q = q\}$	

■ **Figure 8** Preendorsement and endorsement quorums.

$quorum(x) \triangleq  x  > \frac{2 \times  BAKERS }{3}$
$valid\_eqc(eqc, pre) \triangleq quorum(eqc) \wedge$ $\forall Endorse(i, \ell, r, p) \in eqc, (\ell, r, p) = pre.(\ell, r, p)$
$valid\_pqc(b) \triangleq quorum(b.pqc.q) \wedge$ $\forall PreEndorse(i, \ell, r, p) \in b.pqc.q, (\ell, p) = b.(\ell, p) \wedge r < b.r$
$isProposer(i, \ell, r) \triangleq ((\ell + r) \bmod  BAKERS ) + 1 = i$
$isProposerNextRound(i) \triangleq \text{let } (B, P) = CH_i \text{ in}$ $isProposer(i, B.\ell, RND_i + P.r - PQC_i.pred.r + 1)$
$isProposerNextLevel(i) \triangleq ELECT_i \neq - \wedge$ $isProposer(i, B.\ell + 1, RND_i - ELECT_i.b.r)$

■ **Figure 10** Definitions of predicates.

To guarantee progression, a record  $ELECT_i$  of the form  $\{b; q;\}$  is used to store the first observed endorsement quorum (in  $q$ ) for the head block (in  $b$ ). Finally, in order to speed up the convergence of the algorithm, a record  $PQC_i$  of the form  $\{p; r; q;\}$  is used to keep track of the preendorsement quorum  $q$  with the highest round  $r$ , associated to the block payload  $p$ . The initial state for a baker is given in Figure 7. Bakers are locked on and have elected the genesis block  $G$  in order to force the progression to go through proposals at level 1.

**Time and clocks.** Tenderbake runs on the notion of rounds and time. As mentioned in Section 1, the ideal consensus scenario is not always attainable. This is where the concept of rounds comes in. Bakers have a predefined number of seconds to decide on a block. Once that time is up, and if an agreement has not been reached, a timeout event is triggered, and the bakers have to drop what they were doing and start a new round. In Tenderbake, this is achieved with clocks and real-time. By combining timestamp information stored in the blocks and their current clock, bakers can calculate both their current round in the consensus and the time remaining before a timeout is triggered. The protocol is also resistant (to some extent) to a possible clock drift between bakers.

Our model accounts for this clock/real-time mechanism in an abstract way. To do this, we first simplify the problem by considering that all rounds have the same duration. Then, we get rid of local clocks by replacing them with local counters that contain the number of timeouts a baker has received. Finally, we use a global mechanism (the *oracle*, depicted in Figure 4) to notify a baker when a round ends. Although it may seem too simplistic, our mechanism allows us to account for the problems related to time in Tenderbake, in particular the one related to clock drift.

To implement our abstract synchronization mechanism, we assign two local variables to each baker: a boolean  $\text{TO}_i$ , for *timeout*, used by the oracle to communicate the end of a round to the baker, and an integer  $\text{TICK}_i$  to count the number of rounds elapsed since the blockchain was started. We also use a constant  $\Delta$  to set the maximum offset on the number of ticks (*i.e.* rounds) between bakers.

To start a new round for a baker  $i$ , our oracle executes *non-deterministically* the guard/action command in Figure 4 as soon as (1) the baker  $i$  has no timeout to handle (2) the differences between any two bakers' counted rounds does not exceed  $\Delta$ , before and after execution of the transition.

The command's action sets the *timeout* variable of the baker  $i$  to **true** and increments its tick counter. This transition guarantees that no two bakers can drift for more than  $\Delta$  rounds but allows each one to proceed independently. After this transition, the baker must handle its timeout and move according to one of the three cases described in the next paragraph. In Tenderbake, we use  $\Delta = 1$ , which means that internal clocks of the machines on which bakers run are only allowed to drift by an amount that would result in a difference of at most one round.

**Timeout transitions.** As shown in Figure 2, a baker is forced to move to state NP when the oracle resets his  $\text{TO}_i$  variable. This is when the baker can start a new round if no consensus was reached during the current round, or a new level, if the baker has collected a quorum of endorsements for his current head block. The actions bakers are allowed to perform on timeouts depend on their right to propose a new block for the next round (in the same level), or for the earliest possible round of the next level in which the baker can propose. We abstract this authorization with a predicate  $\text{IsProposer}(i, \ell, r)$  which is true when baker  $i$  is the proposer at level  $\ell$  and round  $r$ .

Figure 5 contains the possible behaviors (or transitions) of a baker after a timeout. In NOT THE PROPOSER, the baker first checks that he *is not* the proposer for the next round  $\text{RND}_i + 1$  of the current level  $\text{B.l}$  (see Def. of *isProposerNextRound*). Then, either there is no block stored in  $\text{ELECT}_i$  (denoted by  $\text{ELECT}_i = -$ ), meaning the baker did not obtain a quorum for his head block, or the baker is not the proposer for the next level (see Def. of *isProposerNextLevel*). In the latter case, instead of  $\text{IsProposer}(i, \text{B.l} + 1, 0)$ , the baker checks for the round  $\text{RND}_i - \text{ELECT}_i.b.r$  of the next level  $\text{B.l} + 1$ . This expression takes into account the difference between the baker's current round  $\text{RND}_i$  and the round during which the baker obtained a quorum for his head block (stored in the  $\text{ELECT}_i$  variable). Thus, for instance, if a baker obtains a quorum at round  $\text{RND}_i = r$ , and if he is the proposer for the next level at the end of that round  $r$ , then the baker checks indeed the first round  $\text{RND}_i - r = 0$  of the next level. The actions associated to this transition consist only of resetting the  $\text{TO}_i$  variable and incrementing the counters  $\text{TICK}_i$  and  $\text{RND}_i$ . In PROPOSER OF NEXT ROUND, the baker communicates a proposal  $\text{Propose}(i, (b, P))$  for the next round to the Mempool through the variable  $\text{P}_i$ . The block  $b$  is built using the content of the head block  $\text{B}$  with new timestamp and round information. The payload of this new proposal is either a fresh value (denoted by  $\varepsilon$ ) or the payload of the block stored in the baker's  $\text{PQC}_i$

variable, if it exists. The preendorsement quorum certificate of this new block is either empty or the one stored in  $PQC_i$ . In **PROPOSER OF NEXT LEVEL**, the baker must have a block stored in  $ELECT_i$  and he must also be the proposer of the round  $RND_i - ELECT_i.b.r$  in the next level  $B.l + 1$ . The new proposal contains a fresh payload, an endorsement quorum for its block predecessor taken from  $ELECT_i.g$  and a timestamp equal to  $TICK_i$ .

**The Mempool.** While a Mempool typically serves as a gossip layer, simply passing on messages between bakers, Tenderbake’s Mempool is more sophisticated. For instance, the Mempool keeps a local variable  $NodeCH_i$ , its own copy of the blockchain, the most up-to-date version that it has “seen” come through. Since the consensus in Tenderbake depends on the last two blocks,  $NodeCH_i$  contains only the head of the blockchain and its predecessor in our model. In addition to these blocks, the Mempool also maintains a set  $M_i$  of all of the votes (PreEndorse or Endorse messages) that it receives from all bakers.

Furthermore, when the Mempool receives a proposal, either through a message or a shared variable, it first verifies that the proposed block is actually *better* than its current head. If it is indeed better, the Mempool simply updates its version of the blockchain. Otherwise, it is ignored. The notion of a *better chain* is an important part of a consensus algorithm, corresponding to a total ordering between blocks. In Tezos, this ordering is based on a notion of *fitness*, which amounts to comparing, in a lexicographic order, the following quadruples  $(H.l, H.pqc.r, -PRE.r, H.r) < (l, pqc.r, -pre.r, r)$ , where  $H$  and  $PRE$  are the first two head blocks of  $NodeCH_i$ , while  $h$  and  $pre$  are the blocks received in a  $Propose(j, (h, pre))$  message. Moreover, in addition to fitness, the Mempool ensures the information contained in the *eqc* and *pgc* fields is valid. Last, if this better proposal has been received through a shared variable, the Mempool broadcasts it to the other participants. Figure 9 shows transitions of the Mempool that handle PreEndorse and Endorse votes (received either by messages or through the shared variables  $PE_i$  and  $E_i$ ). These messages are simply stored in  $M_i$ <sup>1</sup>.

**Proposal transition.** As seen in Figure 2, a baker can handle a new proposal in any state. We give in Figure 3 the **Proposal** transition that a baker can execute as soon as he is running a new round and when the head block  $B$  in  $CH_i$  is different from the one in the Mempool. In that case, a baker determines if he can vote (preendorse) for the new head stored in the Mempool. There are only two possibilities for a baker to preendorse a proposal:

1. The chain stored in the Mempool is *strictly* longer than the one stored in the baker.
2. Both chains have the same length and the proposal’s round is equal to the current baker’s round  $RND_i$ . The baker also checks that he is not about to vote twice in the same round, except for the same payload. Moreover, the baker only preendorses in this case if:
  - a. he has never endorsed (locked) a previous proposal in the same level, or
  - b. he is locked to some block payload  $p_0$  at some round  $r_0$ , but the current proposal’s payload is equal to  $p_0$ , or the current proposal got a PQC at some round  $r_1 > r_0$ .

In (1), a baker synchronizes the value of its current round  $RND_i$  in the new level. It also checks, before preendorsing, that the block  $H$ , while at a higher level, does not correspond to an old proposal.

**Quorums.** The last two transitions are described in Figure 8. As mentioned above, the Mempool keeps a set  $M_i$  of all the messages it has received. If the number of preendorse messages for the head block  $B$  stored in  $CH_i$  is enough for a quorum, then a baker can

<sup>1</sup> Although we could wipe the contents of  $M_i$  at each new round startup, we decided not to do it explicitly to be able to explore different mempool cleaning strategies in practice.

execute the Preendorsement Quorum transition to update  $PQC_i$  with his current head and the calculated quorum, change  $LOCKED_i$  to  $B$ , since this is the block he is about to endorse, and communicate an  $Endorse(i, B.l, B.r, B.p)$  message to the Mempool. An endorsement quorum transition is possible in states  $CE$  and  $CP$ . The baker observes endorsement quorums only when his  $ELECT_i$  variable is not set. In that case, if enough endorsement messages exist in the Mempool for his head block, the baker records that block and its quorum in  $ELECT_i$ .

## 4 TLA<sup>+</sup>

In this section we discuss how we go from the previous automaton to its TLA<sup>+</sup> implementation. The automaton makes it fairly straightforward to convert to TLA<sup>+</sup> by simply representing the baker, the Mempool, the possible actions, and the synchronization mechanism.

**The Baker and the Mempool.** We define a constant set  $BAKERS$  of all bakers in the network. A variable  $BakerState$  maps each baker to their state (i.e. the internal variables from Section 3), represented as a record structure. We stray from the types in Section 3 by using  $n$ -tuples instead of records to represent  $LOCKED_i$ ,  $ELECT_i$ , and  $PQC_i$ .  $BakerState[i]$  represents the state of baker  $i$ . To model the phases of the algorithm, we add an internal variable  $STATE_i$  for each baker. Initially, each baker starts off in the following state, where sequences are delimited by  $\langle \rangle$ , and  $Genesis$  is the genesis block:

$$InitialState \triangleq [state \mapsto "np", pqc \mapsto \langle \rangle, ch \mapsto \langle Genesis, Genesis \rangle, rnd \mapsto 0, \\ locked \mapsto \langle \rangle, elect \mapsto \langle Genesis, \{ \} \rangle, timeout \mapsto TRUE, tick \mapsto 0]$$

The Mempool is a record with the fields -  $nodeCH$ , for its local blockchain (the first two blocks),  $msgs$ , the set of  $Endorse$  and  $PreEndorse$  messages it has received, and the fields  $propose$ ,  $endorse$ ,  $preendorse$  for the variables  $P_i$ ,  $E_i$ ,  $PE_i$ . It starts off with an empty set of  $msgs$  and two  $Genesis$  blocks.

**Synchronization.** As mentioned in Section 3, we introduce an oracle transition which allows bakers to progress individually with timeouts ( $TO_i$ ) while maintaining synchronization, *i.e.* by being at most  $\Delta$  rounds apart. We do the same thing in our TLA<sup>+</sup> implementation:  $TO_i$  is the first enabling condition of each timeout step definition.

**Actions.** Bakers and the Mempool are impacted by the various actions on the network. Each of these are defined individually in TLA<sup>+</sup>. For example, the **Endorsement Quorum** step in Figure 2, enabled in  $CP$  or  $CE$ , is defined as follows:

$$EndQuorum(i) \triangleq \wedge BakerState[i].timeout = FALSE \\ \wedge BakerState[i].elect = \langle \rangle \\ \wedge BakerState[i].state = "cp" \vee BakerState[i].state = "ce" \\ \wedge CollectEnd(i) \\ \wedge BakerState' = [BakerState \text{ EXCEPT} \\ \quad ![i].elect = \langle BakerState[i].chain[1].round, \\ \quad \quad BakerState[i].chain[1].contents, \\ \quad \quad BakerState[i].chain[1].time \rangle, \\ \quad ![i].state = BakerState[i].state] \\ \wedge \text{UNCHANGED } Mempool$$

Baker  $i$  can execute this step iff (i) he is synchronized, (ii) he is in state  $cp$  or  $ce$ , and (iii)  $CollectEnd(i)$  is true.  $CollectEnd$  (for “collecting endorsements”) counts all of the  $Endorse$  messages for  $i$ ’s current head in  $Mempool.msgs$  and checks whether it is enough for a quorum. If these three conditions are satisfied, baker  $i$  modifies  $ELECT_i$  and transitions to phase  $NP$  of the algorithm. Every other transition in Figure 2 is defined in a similar way.



**Test scenarios.** While the automaton made writing our TLA<sup>+</sup> specification easier, the spec itself has, in return, proven extremely useful in debugging the automaton. Sometimes a deadlock would be reached when it should not have been, leading us to review Tenderbake's code, and fixing things we overlooked in our model. The main advantage is, however, being able to run various test scenarios. We can easily modify our spec to account for various clock drifts or Byzantine bakers.

## 5 Conclusion

In this paper we proposed a TLA<sup>+</sup> model of Tenderbake, along with an automaton detailing the key parts of Tenderbake. This method simplifies the problem by abstracting the notion of time, while retaining Tenderbake's more nuanced features, such as its more elaborate Mempool. Our method gives us a formalized and executable Tenderbake documentation. This serves as the foundation for running specific test scenarios and verifying properties Tenderbake needs to satisfy. An immediate line of future work is to define those properties and check them with the TLC model checker.

---

### References

- 1 Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Correctness of tendermint-core blockchains. In *22nd International Conference on Principles of Distributed Systems, OPODIS 2018, December 17–19, 2018, Hong Kong, China*, volume 125 of *LIPICs*, pages 16:1–16:16, 2018.
- 2 Lăcrămioara Astefănoaei, Pierre Chambart, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci Piergiovanni, and Eugen Zalinescu. Tenderbake – A solution to dynamic repeated consensus for blockchains. In *Fourth International Symposium on Foundations and Applications of Blockchain*, 2021.
- 3 Sean Braithwaite, Ethan Buchman, Igor Konnov, Zarko Milosevic, Iliana Stoilkovska, Josef Widder, and Anca Zamfir. Formal specification and model checking of the tendermint blockchain synchronization protocol (short paper). In *2nd Workshop on Formal Methods for Blockchains, FMBC@CAV 2020, July 20–21, 2020, Los Angeles, California, USA (Virtual Conference)*, volume 84 of *OASICs*, pages 10:1–10:8, 2020.
- 4 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.
- 5 LM Goodman. Tezos – A self-amending crypto-ledger white paper. URL: [https://www.tezos.com/static/papers/white\\_paper.pdf](https://www.tezos.com/static/papers/white_paper.pdf), 2014.
- 6 Jae Kwon and Ethan Buchman. Cosmos whitepaper, 2019. URL: <https://cosmos.network/resources/whitepaper>.



# Towards Contract Modules for the Tezos Blockchain

Thi Thu Ha Doan ✉

University of Freiburg, Germany

Peter Thiemann ✉ 

University of Freiburg, Germany

---

## Abstract

Programmatic interaction with a blockchain is often clumsy. Many interfaces handle only loosely structured data, often in JSON format, that is inconvenient to handle and offers few guarantees.

Contract modules provide a statically checked interface to interact with contracts on the Tezos blockchain. A module specification provides all types as well as information about potential failure conditions of the contract. The specification is checked against the contract implementation using symbolic execution. An OCaml module is generated that contains a function for each entry point of the contract. The types of these functions fully describe the interface including the failure conditions and guarantee type-safe and sometimes fail-safe invocation of the contract on the blockchain.

**2012 ACM Subject Classification** Software and its engineering → Specification languages

**Keywords and phrases** contract API, modules, static checking

**Digital Object Identifier** 10.4230/OASICS.FMBC.2021.5

**Category** Short Paper

**Supplementary Material** *Software (Source Code)*: <https://github.com/proglang/tezos-project/tree/master/papers/contract-modules/code>

**Funding** *Thi Thu Ha Doan*: supported by a grant from the Tezos foundation.

## 1 Introduction

Contracts on the blockchain rarely run in isolation. To be useful beyond shuffling tokens between user accounts, they need to interact with the outside world. On the other hand, the outside world also needs to interact by initiating transactions and starting contracts that feed information into the blockchain. One direction is addressed by oracles that watch certain events on the blockchain, create a response by calculation or gathering data, and then invoke a callback contract to inject this response into the chain. Trust is an essential aspect for an oracle.

The other direction is about automatizing certain processes in connection with the blockchain. For example, opening or closing an auction according to a schedule, programming a strategy for an auction, or creating an NFT. To this end, an interface is needed to invoke contracts safely. Existing interfaces are lacking because they are essentially untyped (string-based or JSON-based) and often low level because they require dealing directly with RPC interfaces. Trust is not needed because the process runs on behalf of a certain user.

We propose contract modules that provide a clean, language-integrated way to interact with a blockchain from a host language (OCaml in our case). They abstract over underlying string-based interfaces and details like fee handling. They provide a high-level typed interface which reduces a contract invocation to a function call in the host language.

The contract modules approach does not provide a fixed API, but rather generates a specific interface for each contract with one function for each entry point of the contract. This interface is statically checked against the contract implementation to ensure type safety and exception safety. That is, values passed to an interface function do not lead to type mismatches when invoking the underlying contract. Moreover, every failure condition arising during contract execution is handled by proper error reporting according to the interface.



© Thi Thu Ha Doan and Peter Thiemann;

licensed under Creative Commons License CC-BY 4.0

3rd International Workshop on Formal Methods for Blockchains (FMBC 2021).

Editors: Bruno Bernardo and Diego Marmosler; Article No. 5; pp. 5:1–5:9

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

■ **Listing 1** Simple auction contract (auction.tz).

```
parameter (or (unit %close) (unit %bid));
storage (pair (bool %bidding)
             (pair (address %owner)
                  (address %hi_bidder)));
```

Our work is situated in the context of the Tezos blockchain, which supports Michelson as its low-level contract language, and the host language OCaml, which comes with an expressive polymorphic type system as well as a powerful module system that we enhance with contract modules.

## 2 Context

Tezos is a third generation, account-based, self amendable blockchain [8]. It employs a proof-of-stake consensus protocol, which includes ways to evolve the protocol itself. The consensus protocol is executed by so-called bakers and their proposed blocks are checked by validators. They receive some compensation in the form of tokens (Tezzies) for their work. According to proof-of-stake, bakers and validators are nodes elected by the Tezos network according to their token balance.

Each Tezos contract is associated with an account as well as some storage. Contracts are pure functions of type  $parameter \times storage \rightarrow operation\ list \times storage$ , where the types *parameter* and *storage* are depend on the specific contract while the type *operation* is fixed by the Tezos system. When a contract is invoked with a parameter, the blockchain provides the current storage and updates it with the second, storage component of its return value. The first component of the return value is a list of blockchain operations (contract deployments, token transfers, contract invocations, and delegation of baking rights) that are executed transactionally after the first invocation terminates. Each invocation may be accompanied with an amount of tokens that are added to the current account balance of the callee contract.

Contracts are implemented in the language Michelson, a statically typed stack-based language. Each contract has fixed types for its parameter and for its storage. The storage is initialized when the contract is deployed. Besides primitive types like unit, int, bool, address, and string, there are pairs, sums, functions, lists, and maps along with a range of domain-specific types (operation, key, signature, timestamp, key\_hash, contract, mutez – for tokens, and so on) most of which can serve as types for storage and parameters.

A Michelson contract has a single default entry point. However, the *parameter* type is typically a sum type and each component of the sum can serve as a subsidiary entry point.

## 3 An auction contract

As a concrete example, we consider a simple auction contract with the header shown in Listing 1. This contract has two entry points, `close` and `bid`, expressed by giving the single parameter a sum type. To call the entry point `close` we invoke the contract with parameter `Left ()` otherwise we use `Right ()`, where `()` is the sole value of type `unit`. The contract's storage is a nested pair which contains a boolean flag and two addresses.

The contract works as follows. It is deployed with storage `(true, (owner, owner))` which indicates that bidding is allowed and the contract owner is currently the highest bidder. On deployment the owner deposits an initial balance to indicate the minimum bid. Closing the

■ **Listing 2** Example contract module.

```

contract type Auction = sig
  paid entrypoint bid ()
  raises "closed"                (** auction closed *)
    | "too_low"                  (** bid too low   *)

  entrypoint close ()
  raises "closed"                (** auction closed *)
    | "not_owner"              (** caller cannot close *)
end

```

contract transfers the balance to the owner. Closing is restricted to the owner. Closing as well as bidding fails if the auction is closed. If bidding is open and the amount of tokens accompanying the bid exceeds the current highest bid, the current bidder replaces the previous highest bidder and the previous highest bidder is reimbursed. Otherwise, bidding fails.

To invoke this contract from an OCaml program, we generate an OCaml module, say `Auction`, from a specification of the contract. This module contains two functions `close` and `bid` corresponding to the entry points. The type of these entry points reflects further properties of these entry points as well as the ways in which an entry point might fail.

Besides the obvious, technology induced ways that a contract invocation might fail (insufficient gas price offered, insufficient gas to complete, timeout due to lack of connectivity, etc) a Michelson contract can fail due to a programmer induced condition caused by the instruction `FAILWITH`. It terminates contract execution with an error message which is reported back to the caller. This error message includes the top value on the stack.

We consider the technological failures like Java's unchecked exceptions, but we wish to deal with the explicit failures like checked exceptions [2]. Our generated code handles failures in a suitable error monad that makes the failures explicit in a custom datatype.<sup>1</sup>

Listing 2 shows a contract module for the auction contract. It declares the entry point `bid` as `paid`, i.e., it should be invoked with a non-zero amount of tokens, it states the pattern `()` for the input value of type `unit`, and it specifies two failure messages that we wish to deal with programmatically. The `close` entry point is similar, but it is not `paid`. This contract reflects the understanding of the programmer with the intention that the `raises` clauses cover all failures that can arise during execution of the respective entry point.

Listing 3 contains an OCaml module signature as it would be generated from the contract module. The module `Tezos` supposedly contains types and other low-level Tezos-specific definitions. The type `pukh` for public key hashes identifies contracts, the type `mutez` stands for Tezos tokens, the type `status` reflects the internal return status, and `monad` is an internal monad type. The signature declares a function and an error type for each entry point.

The error types mirror the raises clauses. The first argument of each function is the address of the contract, then an optional argument for the transaction fee, an argument for passing an amount of tokens (only for a paid entry point), the next argument would be for the parameter; it is omitted here because its type is `unit`. The return type refers to the specific error type.

<sup>1</sup> Alternatively, errors could be modeled using OCaml exceptions, but we choose to stay within the monadic framework that is used by existing Tezos APIs.

■ Listing 3 Generated signature.

```

type bid_errors =
  | bid_closed      (** auction closed *)
  | bid_too_low    (** bid received is too low *)

val bid
  : Tezos.pukh -> ?fee:Tezos.mutez -> amount:Tezos.mutez
  -> (Tezos.status, bid_errors) Tezos.monad

type close_errors =
  | close_closed    (** auction closed *)
  | close_not_owner (** caller cannot close the auction *)

val close
  : Tezos.pukh -> ?fee:Tezos.mutez
  -> (Tezos.status, close_errors) Tezos.monad

```

#### 4 Simple Checking

We check the contract by symbolic execution against its specification in the contract module. Symbolic execution proceeds by calculating a symbolic stack at each transition from one Michelson instruction to the next. The symbolic interpreter is fully typed and rejects ill-typed Michelson programs, if the ill-typed part is reachable from the default entry point. The initial stack is calculated from the storage type, the parameter type, and the entry point. As we aim to keep symbolic values as concrete as possible, some instructions may turn out to be unreachable or only reachable under certain conditions. This way, we obtain, for each entry point, a set of final symbolic stacks along with a path condition indicating when this final state is reachable. Moreover, for each `FAILWITH` instruction we obtain a symbolic value for the reported message and a path condition indicating reachability of this instruction. We employ the SMT solver Z3 [7, 4] to check the feasibility of a path condition.

Here are some simple examples of checkable properties.

For each entry point, we collect the set of reachable instructions. For example, the `AMOUNT` instruction obtains the amount of tokens sent with a contract invocation. It should not be possible to reach that instruction from an unpaid entry point like `close`. This property is straightforward to check from the path condition generated for the `AMOUNT` instruction.

For each entry point, we collect the set of reachable `FAILWITH` instructions along with their path condition and their arguments. In most cases the symbolic interpreter finds a concrete argument for each `FAILWITH` instruction because the typical usage pattern is to push a concrete (string) value on the stack immediately preceding the `FAILWITH`. It remains to check that each argument to `FAILWITH` should be accounted for by one `raises` clause.

Checking the simple auction contract in this way already flags an omission in the contract module (Listing 2). The problem is that bidding returns the previous highest bid as part of the transaction where the highest bidder is stored as a value of type `address`. However, to receive a token transfer, this address must be cast (by the Michelson implementation of the contract) to an implicit contract of Michelson type (`contract unit`). This cast is

■ **Listing 4** Enhanced contract module.

```

1  contract type SaferAuction = sig
2    storage (Pair (bidding : bool)
3              (Pair (owner : address) (hi_bidder : address)))
4
5    entrypoint close ()
6    requires bidding raises "closed"          (** auction closed*)
7    requires (SOURCE = owner) raises "not_owner"
8    ensures not bidding
9    ensures (post.BALANCE = 0)
10   ensures (TRANSFER_TOKENS unit BALANCE hi_bidder)
11
12   paid entrypoint bid ()
13   requires bidding raises "closed"          (** auction closed*)
14   requires (AMOUNT > pre.BALANCE) raises "too_low" (** low bid *)
15   ensures bidding
16   ensures (post.BALANCE = AMOUNT)
17   ensures (post.hi_bidder = SOURCE)
18   ensures (TRANSFER_TOKENS unit pre.BALANCE hi_bidder)
19
20   invariant (post.owner = owner)
21   invariant (post.bidding => bidding)
22   invariant (post.hi_bidder = hi_bidder or post.hi_bidder = SOURCE)
23 end

```

unavoidable as a value of contract type cannot be stored. However, the cast may fail and its failure leads to an error condition that is reported with a `FAILWITH` instruction that is not covered by the contract module.<sup>2</sup> The entry point for closing has the same issue with the address of the owner, who is scheduled to receive the balance of the contract.

The best way to address this issue would be to assert that the addresses stored for the owner and the highest bidder always cast successfully into the `(contract unit)` type so that the stated module type can be retained. Unfortunately, doing so requires control over the initialization of the storage, which is part of the deployment of the contract. At present, our design for contract modules does not include support for the deployment of a contract. Moreover, it is not clear whether it makes sense to support it as there is no guarantee that a contract will be deployed using the API generated from a contract module.

The quick fix is to add `raises` clauses for the `No entrypointing ...` message. With this fix, we successfully check the module description against the Michelson implementation of its contract. Another fix would be to rewrite the contract to enable the dynamic creation of auctions.

## 5 Advanced Checking

As it is expensive to invoke a contract just to find out that it fails, we propose to extend entry point specifications with preconditions along with some global invariants as shown in Listing 4. The idea is that the generated OCaml module tries to check the preconditions

<sup>2</sup> We generated the Michelson code for this example using the Liquidity compiler (<https://www.liquidity-lang.org/>). While the failures for `closed` and `too low` are explicit in the source program, the compiler inserts the casts into the contract type automatically using the error message `No entrypoint default with parameter type unit`.

## 5:6 Towards Contract Modules for the Tezos Blockchain

off-chain before invoking the contract. To this end, the off-chain code needs to obtain properties like balance, storage etc of the contract, but this information is available from a Tezos node without a fee! Of course, such an off-chain check is prone to race conditions as concurrent contract invocations from other parties may interfere and change the data before our module gets a chance to invoke the contract.

Generally, a specification can refer to the values available to the contract. For example, the instructions `SOURCE`<sup>3</sup>, `BALANCE`, and `AMOUNT` refer to the respective values. As the current `BALANCE` includes the `AMOUNT` sent with the transaction, we write `pre.BALANCE` (= `BALANCE` – `AMOUNT`) for the balance before the transaction starts and `post.BALANCE` for the balance after the transaction finishes. The latter is calculated by subtracting the amounts transferred from the current `BALANCE`. The existence of a token transfer in the returned operation list is indicated by the respective `TRANSFER_TOKENS` instruction. The components of the storage are referred to by name. We distinguish the outgoing storage by prepending `post.` as in `post.owner`.

For each clause `requires  $\Phi$  raises  $s$` , we take the path condition  $\Theta$  for a `FAILWITH` instruction with argument matching the string  $s$  and check that  $\Phi \wedge \Theta$  is not satisfiable in the context given by the initial stack for the entry point.

From all `requires  $\Phi_i$`  and `ensures  $\Psi_j$`  clauses, we check that  $\neg(\bigwedge \Phi_i \rightarrow \bigwedge \Psi_j)$  is unsatisfiable in the context given by the initial stack for the entry point and its corresponding final stack and path condition.

We discuss two of the preconditions to highlight the properties that need to be analyzed and where race conditions may interfere.

The precondition `SOURCE = owner` of `close` can be checked off-chain because the owner's address is part of the storage. However, it is in general unsound to perform such a test off-chain because the owner's address could change if an entry point changes that component of the storage. To safely check this precondition, all other entry points must preserve the `owner`'s address, which is indeed the case by the postcondition in line 17. This postcondition is verified as outlined above.

The situation is slightly more complex at the `bid` entry point. The failure "`closed`" is guarded by `bidding`. As the `bidding` component of the state can change, a precise prediction is not possible. A closer look reveals some subtlety. If `bidding` is true, then the flag may have changed by some interleaved call to `close`. However, if `bidding` is false, then there is no point in invoking the contract because `bidding` will never be reset to true. We address this situation by verifying the global invariant `post.bidding => bidding`.

For the failure "`to_low`", the analysis is very similar: we need to know that there is no successful execution of `bid` after an execution of `close`. Moreover, each invocation of `bid` raises the balance of the contract monotonically. Thus, if the off-chain check `AMOUNT > pre.BALANCE` fails, we can be sure that the contract invocation will also fail; either because someone closed the auction or because the balance is at least as high as in the off-chain sample. Checking `pre.BALANCE` off-chain is particularly simple, because it is the current balance of the account.

---

<sup>3</sup> The execution of a Michelson contract is part of a transaction, which can encompass several contract executions. The `SOURCE` of a Michelson contract is the originator of the entire transaction.



## 6 Symbolic interpretation of Michelson

Michelsym, our symbolic interpreter for Michelson, works in two stages. In the first stage, it calculates symbolic stacks between each pair of reachable instructions. The underlying symbolic domain comprises all concrete values, supports the type system, and generates a term representation for symbolic values. Michelsym collects a path predicate that is extended at each conditional, but which remains uninterpreted. If symbolic execution reaches certain instructions (most notably `FAILWITH`), Michelsym records the argument value and the path condition.

Presently, Michelsym works on Michelson files which result from compiling the examples provided with the Liquidity compiler. It generates human-readable output as well as output in the SMTlib format suitable for SMT-solvers like Z3. This output needs to be weaved together manually with the formulas generated from a contract module.

We plan to revise Michelsym so that it directly communicates with Z3 to directly check for unsatisfiable path conditions and to be better integrated with the contract module frontend.

## 7 Related work

Smart contract-based applications often require interaction between a smart contract on the blockchain and the outside world. However, smart contracts cannot connect to external sources on their own. This is where oracles [13, 5] come into play. Oracles act as a bridge between smart contracts and external sources. Namely, they collect and verify external information and make it available to smart contracts on the blockchain. Several research works have been conducted to provide oracle solutions for the Blockchain. Adler et al. [12] proposed a framework to provide developers with a guide for incorporating oracles into blockchain-based applications. Oracles may need to observe the state of the chain to determine what information to send. In addition, oracles transmit data from external sources to the blockchain. Therefore, they would need to have a programmatic interface to interact with the blockchain.

The basic idea of our advanced checking, namely precondition checking, is inspired by JML, the Java modeling language [11, 6], in which the behavior of program components is described as a contract between Java program and its clients. This contract specifies preconditions that must be satisfied by clients and postconditions that are guaranteed by the program. A precondition supplied with a client call must be verified before a function defined by the program is called, and the program guarantees that the postconditions are satisfied in return after the call. The original idea of using preconditions and postconditions dates back to Hoare's paper [10]. Software contracts have also been proposed for blockchain [3]. In our approach, the safe contract module in the OCaml language comes close to contracts in this sense. Several applications are based on JML [14]. Ahrendt et al. [1] propose the KeY framework for deductive software verification.

Our contract module specifies preconditions and then off-chain checks whether a user call satisfies those preconditions. Symbolic execution plays an important role in the preconditions checking in our method. A smart contract is verified against its specification in the contract module by symbolic execution. In a paper on symbolic execution [9], Hentschel et al. proposed the symbolic execution debugger (SED) platform, which is based on the KeY framework. The platform SED has a static symbolic execution engine for sequential programs.

## 8 Conclusion

Current blockchains often provide low-level interfaces to interact with smart contracts. These interfaces work with loosely structured without static guarantees. This paper presents ongoing research on the programmatic interaction with smart contracts on the Tezos blockchain that could benefit developers of mixed applications and oracles comprised of on-chain and off-chain parts. The approach does not provide a general API, but targets each individual smart contract by generating a specialized contract module that provides a typed high-level interface from a contract specification. In doing so, errors from contract calls are explicitly specified in a user-defined data type. A contract call is wrapped in a fully typed and integrated OCaml function. In addition, the wrapper can check preconditions before the actual call to reduce the waste of gas of a failed call.

While our conceptual approach is applicable and would be beneficial for other blockchains, the actual implementation is very much tied to the Tezos blockchain. The key asset here is the symbolic interpreter which is hardcoded for Michelson and adapted to the peculiarities of the Tezos blockchain. By targeting Michelson, our work is applicable to all languages running on Tezos, but a similar tool would have to be developed from scratch for another blockchain.

---

## References

- 1 Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter Schmitt, and Mattias Ulbrich. *Deductive Software Verification – The KeY Book: From Theory to Practice*, volume 10001. Springer-Verlag, January 2016. doi:10.1007/978-3-319-49812-6.
- 2 Davide Ancona, Giovanni Lagorio, and Elena Zucca. A core calculus for Java exceptions. In Linda M. Northrop and John M. Vlissides, editors, *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14–18, 2001*, pages 16–30. ACM, 2001. doi:10.1145/504282.504284.
- 3 Massimo Bartoletti. Smart contracts contracts. *Frontiers Blockchain*, 3:27, 2020. doi:10.3389/fbloc.2020.00027.
- 4 Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph M. Wintersteiger. Programming Z3. In Jonathan P. Bowen, Zhiming Liu, and Zili Zhang, editors, *Engineering Trustworthy Software Systems – 4th International School, SETSS 2018, Chongqing, China, April 7-12, 2018, Tutorial Lectures*, volume 11430 of *Lecture Notes in Computer Science*, pages 148–201. Springer, 2018. doi:10.1007/978-3-030-17601-3\_4.
- 5 Giulio Caldarelli. Understanding the blockchain oracle problem: A call for action. *Information*, 11(11), 2020.
- 6 Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects, FMCO'05*, page 342–363, Berlin, Heidelberg, 2005. Springer-Verlag.
- 7 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3\_24.
- 8 L. Goodman. Tezos – A self-amending crypto-ledger, 2014. URL: <https://www.tezos.com/static/papers/white-paper.pdf>.

- 9 Martin Hentschel, Richard Bubel, and Reiner Hähnle. The symbolic execution debugger (SED): a platform for interactive symbolic execution, debugging, verification and more. *International Journal on Software Tools for Technology Transfer*, 21, October 2019.
- 10 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- 11 Gary Leavens and Yoonsik Cheon. Design by contract with JML, 2006. URL: <https://www.cs.ucf.edu/~leavens/JML/index.shtml>.
- 12 Kamran Mammadzada, Mubashar Iqbal, Fredrik Milani, Luciano García-Bañuelos, and Raimundas Matulevičius. *Blockchain Oracles: A Framework for Blockchain-Based Applications*, pages 19–34. Springer Verlag, September 2020.
- 13 Roman Mühlberger, Stefan Bachhofner, Eduardo Castelló Ferrer, Claudio Di Ciccio, Ingo Weber, Maximilian Wöhrer, and Uwe Zdun. Foundational oracle patterns: Connecting blockchain to the off-chain world. *Business Process Management: Blockchain and Robotic Process Automation Forum*, page 35–51, 2020.
- 14 Peter W. V. Tran-Jørgensen. Automated translation of VDM-SL to JML-annotated Java. *Technical Report Electronics and Computer Engineering*, 5(29), March 2017.

