FMBC 2021: Pre-Proceedings

Bruno Bernardo Diego Marmsoler

July 18-19, 2021

Preface

The 3rd International Workshop on Formal Methods for Blockchains (FMBC) will take place virtually on July 18/19 2021 as part of CAV 2021, the 33rd International Conference on Computer-Aided Verification. Its 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 will be presented at the workshop as regular talks, as well as 3 extended abstracts that will be presented as lightning talks. Additionally, we are very pleased to have an invited keynote by David L. Dill (Novi/Facebook, USA).

This volume contains the papers selected for regular talks, the extended abstracts selected for lightning 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.

July 2021

Bruno Bernardo
Diego Marmsoler

nomadic labs

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	University of Luxembourg, Luxembourg
Dana Drachsler Cohen	Technion, Israel
Ansgar Fehnker	University of Twente, Netherlands
Maurice Herlihy	Brown University, USA
Lars Hupel	INNOQ, Germany
Florian Kammueller	Middlesex University London, UK
Igor Konnov	Informal, Austria
Andreas Lochbihler	Digital Asset, Switzerland
Diego Marmsoler	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

Supporting Reviewers

Yuteng Lu
Luis Arrojado da Horta

Papers

Keynote: Formal verification of Move programs for the Diem blockchain

Abstract

The Diem blockchain, which was initiated in 2018 by Facebook, includes a novel programming language called Move for implementing smart contracts. The correctness of Move programs is especially important because the blockchain will host large amounts of assets, those assets are managed by smart contracts, and because there is a history of large losses on other blockchains because of bugs in smart contracts. The Move language is designed to be as safe as we can make it, and it is accompanied by a formal specification and automatic verification tool, called the Move Prover. The Diem Framework is the Move code that makes up the core logic of the Diem blockchain, managing accounts, payments, etc. Extensive formal specifications have been written for the most important properties of the Framework, all of which can be formally verified by the Move Prover in less than 40 seconds per file. Indeed, the framework is re-verified automatically in continuous integration whenever new code is submitted to Github. The entire blockchain implementation, including the Move language, virtual machine, the Move Prover, and near-final various Move modules are available on https://github.com/diem/diem. This talk will be about the goals of the project and the most interesting insights we've had as of the time of the presentation.

Bio

David L. Dill is a Lead Researcher at Facebook, working on the Diem blockchain project. He is also Donald E. Knuth Professor, Emeritus, in the School of Engineering at Stanford University. He was on the faculty in the Department of Computer Science at Stanford from 1987 until becoming emeritus in 2017. Prof. Dill's research interests include formal verification of software, hardware, and protocols, with a focus on automated techniques, as well as voting technology and computational biology. For his research contributions, he has received a CAV award and Alonzo Church award. He is an IEEE Fellow, an ACM Fellow and a member of the National Academy of Engineering and the American Academy of Arts and Sciences. He also received an EFF Pioneer Award for his work in voting technology and is the founder of VerifiedVoting.org, an organization that champions trustworthy elections.

Part I.

Accepted Papers

Money Grows on (Proof-)Trees: Formalisation and Correctness of the FA1.2 Ledger Standard

Money grows on (proof-)trees: the formal FA1.2 ledger standard

Murdoch J. Gabbay
Heriot-Watt University, Edinburgh, UK and
Nomadic Labs, Paris, France

Arvid Jakobsson
Nomadic Labs, Paris, France

Kristina Sojakova[1]
INRIA, Paris, France

Abstract

Once you have invented money, you will find that you 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 entrypoints 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 Replace ccsdesc macro with valid one

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

Digital Object Identifier 10.4230/OASIcs.FMBC.2021.7

Acknowledgements [to be added in final version]

[1] Sojakova wrote the Coq code described in this paper

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 7; pp. 7:1–7:13
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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.[2] 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.[3] 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*);[4] 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.[5]

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

The **FA1.2 standard**[6] 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. An FA1.2-compliant smart contract must provide at a minimum the following five entrypoints and behaviours:[7]

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.[8]
3. `%getAllowance` expects an **owner**, a **spender**, and returns the approved transfer allowance for the spender, via a callback (see Remark 1 below).
4. `%getBalance` expects an **owner** and returns the owner's balance via a callback.

[2] https://en.wikipedia.org/wiki/Tezos

[3] 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 more precise, for the sake of space-efficiency, what is on the Tezos blockchain is not the protocol code but a hash of it (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 (in a sense of 'must' precisely as strong as 'our hash function is computationally infeasible to break') be the protocol code itself — is propagated across the nodes of the network for them to load and run. This low-level functionality is handled by a 'shell' (think: BIOS).

[4] We mention a few wrapped tokens at the start of Section 4.

[5] 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*.

[6] https://tzip.tezosagora.org/proposal/tzip-7/

[7] A smart contract could offer *more* than this, but if it offers *less* — then is not an FA1.2-compliant smart contract.

[8] When you use a debit card to pay a merchant for a purchase, you do not pay the merchant directly: you *authorise* the merchant to debit your account (occasionally, the merchant may even not do so; the authorisation is granted but the withdrawal does not take place). Likewise a direct debit is in fact an approval for a withdrawal. This is how things are done. Thus, `%approve` does not send tokens directly; it approves another smart contract to make a token withdrawal, up to a certain limit. Thus when you sell tokens for tez in an exchange like Dexter, you do not transfer tokens directly to Dexter yourself: you use `%approve` to give permission to Dexter to transfer tokens from your account to its own.

5. `%getTotalSupply` returns the total sum of all balances in the ledger, via a callback.

▶ Remark 1 (Some background). A call to an 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 that X will get passed as a parameter to the nominated callback entrypoint.

1.3 This is not enough

This is reasonable *per se*, but it is not enough, due to the following three points:

1. The FA1.2 standard is written in English. This means it *might* be incomplete or incoherent,[9] 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 now 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 thing them as it does to the other users with whom they might transact.

1.4 Our work in a nutshell

This paper reports on a verification effort undertaken at Nomadic Labs that we argue addresses all of 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 no fewer than three smart contract which claim to be FA1.2-compliant.

The reader should not expect novel maths in this work — indeed, in this context 'novel' = 'untested' and is likely to 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 all three of the points above, by providing: a formal specification of the standard, formal representations within the theorem-prover of the programs themselves,

[9] 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.

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 commercial problem of a particular (Coq-based) theorem-prover technology ecosystem.[10] We hope that the work reported on in this paper can serve as a model for future efforts.

▶ Remark 2. We may write *smart contract* and *implementation* synonymously in this paper. Note also that smart contracts may be written in high-level programming languages, but to run on Tezos they must always get compiled to a lower-level stack-based language called Michelson.[11] We may not always distinguish between the original program and its complied Michelson executable, but we will when this difference matters and it will always be clear what is meant.

2 Introducing: the formal FA1.2 standard

The verification files are written in Coq and structured into three files as follows, where the later items depend on the earlier ones:

1. `fa12_interface`: This specifies internal functions which the smart contract must support (see Figure 1), along with axioms on their behaviour.[12]
2. `fa12_specification`: This specifies entrypoint behaviour in terms of these functions. Files 1 and 2 render into precise Coq code the English of the FA1.2 standard, and also go beyond this by specifying internal functions which must be supported, rather than just entrypoints. Thus, the formal FA1.2 standard adds some *intensional* content, which the English FA1.2 standard lacks.
3. `fa12_verification`: This contains some useful lemmas and theorems, which are derived purely from postulates in the formal FA1.2 standard. Thus, these are properties of any FA1.2-compliant smart contract.

In this paper we will call the three files above the **formal FA1.2 standard**.

▶ **Example 3** (Component 1: The interface file). The code asserting functions required by `fa12_interface` is in Figure 1, and two example axioms are in Figure 2.

In Figures 1 and 2, `data` is a standard Mi-Cho-Coq [2] wrapper mapping (a Coq representation of) Michelson types to Coq types,[13] and `sto` is short for 'storage' and represents a state datum that is threaded through computations.[14] We trust that with this information, the functions and axioms should be self-explanatory.

[10] ...yet more proof, if proof were needed, that what starts in universities ends on the engineer's workbench.

[11] Think: the Tezos equivalent of bytecode or machine code, though Michelson is still quite high level.

[12] These are not entrypoints and cannot be called from outside the smart contract (see `fa12_specification`). Also, the functions may be, but need not be, explicit in the smart contract code — e.g. the smart contract might be in a low-level, non-functional language — so long as they *could* be defined on the underlying data. We might call the FA1.2 interface an **idealised implementation**, where 'idealised' is used in the sense of 'Platonic ideal' (rather than the sense of 'perfect').

[13] https://gitlab.com/nomadic-labs/mi-cho-coq/-/blob/kristina-fa12-verification-rebase/src/michocoq/semantics.v#L281

[14] ...containing information like e.g. ledger entries, address of admin, total of all balances, and so forth.

```
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
setAllowance   : data storage_ty -> data address -> data address ->
                                    data nat     -> data storage_ty
```

Figure 1 Types of key functions from the FA1.2 interface file

```
(* Set balance to amount and then read that balance: get that amount*)
Parameter getBalance_setBalance_eq : forall sto owner amount,
  getBalance (setBalance sto owner amount) owner = amount.

(* Set a balance and then read another balance: other balance unchanged*)
Parameter getBalance_setBalance_neq : forall sto owner owner' amount,
  owner <> owner' ->
  getBalance (setBalance sto owner amount) owner' =
  getBalance sto owner'.
```

Figure 2 Example axiom from the FA1.2 interface file

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

Figure 3 Example axiom from the FA1.2 specification file

```
Theorem sumOfAllBalances_constant
          (    env : @proto_env self_type  )
          (      p : data fa12_parameter_ty)
          (    sto : data storage_ty       )
          (ret_ops : data (list operation) )
          (ret_sto : data storage_ty       ) :
    main env p sto ret_ops ret_sto ->
    sumOfAllBalances ret_sto = sumOfAllBalances sto.
```

Figure 4 Main result of the FA1.2 verification file

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

Figure 5 Parsing to a Michelson code string into Mi-Cho-Coq's deep embedding of Michelson

▶ Remark 4 (Functions, not entrypoints). 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 down `fa12_interface` we have refined the 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, it is intensional, because it specifies certain *internal functions* which must be provided by an implementation.

▶ **Example 5** (Component 2: The specification file)**.** Code asserting the behaviour of entrypoints is given in `fa12_specification`; see Figure 3.[7] The example code 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. Let's spell this out (in small font):

```
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.  )
  env           (* the current environment *)
  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' *)
  balance       (* parameter: value sent to 'contr'. balance is thus a 'nat' *)
  (amount env)  (* amount of tez: using the function 'amount' applied to the environment, we return
                   the number of tez that was sent to this contract and that triggered this execution.
                   Hence, we just "pass the tez 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 *)
```

▶ **Example 6** (Component 3: The verification file)**.** This contains lemmas derived just from postulates in the interface and specification files, which therefore hold 'once and for all'; they are valid for any implementation for which these postulates hold.

The main result is that the FA1.2 entrypoints do not change the sum of all balances (the total number of tokens on the ledger, as also returned by `%getTotalSupply`): the statement of the result is given in Figure 4. This is a relevant result in and of itself, and it is a sanity check on the design of the interface and specification, that they specify enough of the behaviour of an implementation that this can be proved.

▶ Remark 7. So far we have sketched how the *FA1.2 standard* (a short English document) has been refined and formalised into the *formal FA1.2 standard*, which consists of three files:

1. the FA1.2 interface (specifies internal functions and axioms on those functions),
2. the FA1.2 specification (how external entrypoints are wired to internal functions), and
3. the FA1.2 verification file (some logical consequences of the interface and specification file; in particular that FA1.2-specified entrypoints do not change the total supply of tokens).[15]

Next we discuss the workflow of actually verifying a concrete implementation.

3 Per-implementation verification

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

1. an implementation by camlCase written in Morley[16],
2. an implementation by Edukera written in Archetype[17], and
3. a liquidity ledger that is part of the (at time of writing) prototype Dexter 2 smart contract by the LIGO lang team, written in CameLigo[18].

Verification follows the following steps, which for the sake of argument we consider for the camlCase smart contract:

[15] Nothing prevents an implementation from providing *additional* entrypoints to e.g. mint or burn tokens, and if it does it might still be FA1.2-compliant. It is just that the FA1.2-specified entrypoints must not do this.

[16] A Haskell eDSL for writing Michelson contracts: `https://hackage.haskell.org/package/morley`

[17] A language and toolchain for specifying, implementing and verifying Tezos smart contracts: `https://archetype-lang.org/`

[18] A language with ML-like syntax for implementing Tezos smart contracts: `https://ligolang.org/`

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 in a file `fa12_camlcase_string.v`.
   We do not work further with the original source code of the smart contract directly. We may use it for reference, but what gets validated is the Michelson file.[19]
2. This Michelson code string is parsed into a term of Mi-Cho-Coq's deep embedding of Michelson, in a file called `fa12_camlcase.v`. This is a one-line operation; see Figure 5.[20] Thus we now have (dynamically, in memory) a Coq datum `contract` representing the Michelson code string read from disk.[21] Properties of this Coq datum are asserted and proved.
3. It is in the file `fa12_camlcase_spec.v` that any peculiarities of the 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.
4. Finally, in a file `fa12_camlcase_verification.v` we prove that the (high-level description of the) implementation satisfies the formal FA1.2 specification.
   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 (three Coq files as discussed in Section 2) is not mathematically assured. This was a creative human step of taking the FA1.2 English description and fleshing it out to something formal in Coq that is more intensional, extensive, and precise than the English source.
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.[22] The Michelson code is what is actually gets executed, so this is no bad thing, but note that it 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. The transformation of the Michelson code string into Mi-Cho-Coq takes place in Coq but is just a mechanical transformation. We do have to trust that Mi-Cho-Coq does this correctly.
4. Everything else is completely rigorous, provided we trust the Coq kernel.

[19] 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 — so when we say "We validated a contract" what we *actually* mean is "We validated one particular compilation to Michelson of that contract".

[20] https://gitlab.com/nomadic-labs/mi-cho-coq/-/blob/kristina-fa12-verification-rebase/src/contracts_coq/fa12_camlcase.v#L239

[21] So when we say "We validated one particular compilation to Michelson of that contract" what we *actually* mean is "We validated a Coq datum representing one particular compilation to Michelson of that contract".

[22] Morley is more of a macro language for Michelson, but it still includes non-trivial transformation of the source code that are not yet proven semantics preserving.

On the one hand, this might seem complicated.[23]

On the other hand, we would argue that this is a sensible modular workflow, and may also be *the* best way to structure a verification of this type — especially if we plan to verify more than one FA1.2-compliant ledger. The workflow maximises modularity and reuse, minimises reinventing of the wheel, and accommodates both *a posteriori* and *a priori* validation workflows:

- *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 FA1.2 interface, 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 FA1.2 interface, 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 8.** We continue Example 6. 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 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.[24] 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.[25]

For scale, verification of the first property requires 60 lines of Coq code for the camlCase contract, 21 lines for the Edukera contract, and 61 for the Dexter 2 contract.

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 are all FA1.2-compliant — but clearly they are also different and special. Being FA1.2-compliant is just a property of a smart contract. In particular:

[23] In a sense, this is a kind of dual to *program extraction*, where we start from a high-level specification (e.g. in Coq) and extract from it an executable which then compiles to byte- or machine-code, which (if we trust our compilers) is correct by construction.

[24] 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.

[25] 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.

- 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:

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. comment 1 of Subsection 1.3).

4.2 Issue 2: passing tokens to a view entrypoint

As noted in Remark 1, 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`.[26]

All three implementations opted to discard such tokens, thus leaving the tokens with the sender. Thus, if we called `%getBalance` and sent it some tokens, the camlCase, Edukera, and Dexter 2 contracts would return the tokens untouched.

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 underlying 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.

[26] An OO programmer would call the view entrypoints *getters*.

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.

The ERC20-K semantics[27] 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,[28] 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 have 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.[29]

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.[30] 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.[31] Thus, the Edukera FA1.2 specification is a reflection of the FA1.2 standard into the Archetype toolstack, though as currently written it remains

[27] `https://runtimeverification.com/blog/erc20-k-formal-executable-specification-of-erc20/`

[28] No published academic work, unfortunately. The funniest is a linear logic representation by one Rainy McRainface `https://dapphub.github.io/LLsai/token`.

[29] `https://github.com/edukera/archetype-lang/blob/0a5ad0832709ac102a14534f22d4f94cb185866d/contracts/fa12.arl#L54`

[30] Details in an Agora post https://forum.tezosagora.org/t/a-verified-implementation-of-fa1-2/2264; search for the section on Verification.

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

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).[32]

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.

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.[33]

Property-based testing

We argued in Remark 9 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,[34] which will test your bytecode online for compliance with the ERC20 token standard while U wait [5], subject to significant restrictions on the code.[35]

[32] One could argue that the Archetype FA1.2 specification could be relaxed — and anyway, the formal FA1.2 standard is also 'just' a reflection of the FA1.2 standard into Coq. This is true, but it 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 other languages (Morley and CameLigo respectively). The key point here is not one of expressivity but of scope: Archetype's uniformity and power are available *inside the Archetype toolstack*, whereas to benefit from the formal FA1.2 standard you can use whatever toolstack you like — so long as you add an entry for a Coq wizard to your budget. This is not either/or, so much as two complementary approaches in a rich design space.

[33] https://gitlab.com/tzip/tzip/-/blob/master/proposals/tzip-12/tzip-12.md and https://tezos.b9lab.com/fa2

[34] https://erc20.fireflyblockchain.com

[35] Listed in https://erc20.fireflyblockchain.com/disclaimer.html. For instance: functions not in the ERC20 standard are ignored — which might sound innocuous but it is not, since without extra

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. More speculatively, one could imagine a general-purpose tool which inputs an arbitrary Coq specification like our formal FA1.2 standard, and outputs an online test-suite, thus combining the rigour of our approach 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.[36]

Several ledger implementations already exist, both live and deployed (ETHtz, USDtz, and tzBTC) and also prototypical and undeployed (camlCase, Edukera, and Dexter 2 by Nomadic Labs).

Smart contracts for ledgers are by design designed 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 blockchain.[37] 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 9. Before this research, there was an English standard called 'the FA1.2 standard', and multiple implementations whose correctness was unknown. If they were certified in some way (as is the case for the Edukera contract), then there was 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 (we call it the *formal FA1.2 standard* in this paper), and verified three implementations against this formal standard. Thus, not only are they now proven correct, but implicit in the framework in which those proofs of correctness are embedded is also a guarantee that they are 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 in the process, by the detection and elimination of some potentially dangerous corner cases. We think it can be considered a success.

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

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.

[36] 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.

[37] . . . which may find itself blamed even if the smart contract was created by a third party.

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. URL: https://rdcu.be/ck0WJ, 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. URL: https://doi.org/10.1007/978-3-030-54994-7_28, 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. URL: https://arxiv.org/pdf/1802.06038.pdf, 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.

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 entrypoint 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

Keywords and phrases contract API, modules, static checking

Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

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.

Contract modules provide a clean, language-integrated way to interact with a blockchain. 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 language.

Contract modules do not provide a fixed API, but rather generate a specialized interface for each contract. This interface is statically checked against the contract implementation to ensure type safety and exception safety (every failure condition arising is handled by proper error reporting).

Our work is situated in the context of the Tezos blockchain, which supports Michelson as its low-level contract language, and the language OCaml, which comes with an expressive

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:7
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
parameter (or (unit %close) (unit %bid));
storage (pair bool           # bidding allowed
         (pair address       #  contract owner
          address            # highest bidder's address
        ));
```

Listing 1 Simple auction contract (auction.tz)

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 [5]. 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 just nodes elected by the Tezos network according to their token balance.

Each Tezos contract owns an account as well as some storage. Contracts are pure functions of type parameter × storage → operation list × storage. 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 is a list of blockchain operations (contract deployments, token transfers, contract invocations) 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 contract.

Contracts are implemented in the language Michelson, a fully 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 (and many more) that can serve as types for storage and parameters.

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 entrypoints, `close` and `bid`, expressed by giving the single parameter a sum type. To call the entrypoint `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 also the highest bidder. On deployment the owner deposits an initial balance to indicate the minimum bid. Closing the contract transfers the balance to the owner. It 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, too.

```
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
```

Listing 2 Example contract module

To invoke this contract from an OCaml program, we'd like to generate an OCaml module, say `Auction`, from a specification of the contract. This module contains two functions `close` and `bid` corresponding to the entrypoints. The type of these entrypoints reflects further properties of these entrypoints as well as the ways in which an entrypoint 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 [?]. Our generated code handles failures in a suitable error monad that makes the failures explicit in a custom datatype.[1]

Listing 2 shows a contract module for the auction contract. It declares the entrypoint `bid` as **`paid`**, i.e., it needs to be invoked with a non-zero amount of tokens, it gives the pattern `()` for the input value of type `unit`, and it specifies two possible failure messages that we wish to deal with programmatically. The `close` entrypoint is similar, but requires no tokens.

Listing 3 contains an OCaml module signature as it could 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 entrypoint.

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 entrypoint), 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.

4 Simple Checking

We plan to check the contract by symbolic execution against its specification in the contract module. Here are some examples of checkable properties.

For each entrypoint, we collect the set of reachable instructions. For example, the `AMOUNT`

[1] Alternatively, this could be done using OCaml exceptions, but we chose to stay within the monadic framework that is already used by existing Tezos APIs.

```
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
```

Listing 3 Generated signature

instruction obtains the amount of tokens sent with a contract invocation. It should not be possible to reach that instruction from an unpaid entrypoint like `close`.

For the `FAILWITH` instructions, we also collect their arguments. The symbolic interpreter needs to retain concrete values as much as possible to obtain precise results at this point. Each argument to `FAILWITH` should be accounted for by one **`raises`** clause.

5 Advanced Checking

As it is expensive to invoke a contract just to find out that it fails, we propose to extend entrypoint specifications with preconditions as shown in Listing 4. The idea is that the generated OCaml module tries to check the preconditions 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 the Tezos node without a fee! We discuss two of the preconditions to highlight the properties that need to be analyzed.

The precondition **`sender`** `= 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 entrypoint changes that component of the storage. To safely check this precondition, the analysis must determine that the owner component of the storage remains the same across all possible execution paths of the contract.

Moreover, the gathering of instructions must build path predicates, such that each `FAILWITH` instruction comes with a predicate that must be true to reach the instruction. In contract implementation, the path predicate is **`sender`** `!= owner`. As the conjunction of path predicate and precondition is unsatisfiable and because the `owner` component is constant, an off-chain test for **`sender`** `= owner` precisely predicts whether the failure condition arises.

The situation is slightly more complex at the `bid` entrypoint. The failure `"closed"` is guarded by `bidding`. As the `bidding` component of the state can change, 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.

```
contract type SaferAuction = sig
  storage (Pair (bidding : bool)
                (Pair (owner : address) (hi_bidder : address)))

  entrypoint close ()
  requires bidding raises "closed" (** auction closed*)
  ensures not bidding
  requires (sender = owner) raises "not␣owner"

  paid entrypoint bid ()
  requires bidding raises "closed" (** auction closed*)
  ensures bidding
  requires (amount > old.balance)
  raises "too␣low"                 (** bid too low *)
  ensures (balance >= old.balance)
  ensures (owner = old.owner)
end
```

Listing 4 Enhanced contract module

Hence the analysis should also record value transitions in the storage for non-failing executions. For `bidding` `bid` transitions from true to true and `close` transitions from true to false. Thus, if the off-chain check finds `bidding` = false, then we can precisely predict that `bid` would fail and trigger the corresponding error without invoking the contract proper.

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 > balance` fails, we can be sure that the contract invocation will also fail; either because some closed the auction or because the balance is at least as high as in the off-chain sample.

6 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 [10, 3] 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.[9] 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 [8, 4], 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 [7]. Software contracts have also been proposed for blockchain [2]. In our approach, the safe contract module in the OCaml language comes close to contracts in this sense. Several applications are based on JML [11]. 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 [6], 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.

7 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.

Status: we are currently working on a prototype of the symbolic interpreter.

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, 01 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 Giulio Caldarelli. Understanding the blockchain oracle problem: A call for action. *Information*, 11(11), 2020.

5 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.

6 L. Goodman. Tezos—a self-amending crypto-ledger, 2014. URL: https://www.tezos.com/static/papers/white-paper.pdf.

7 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, 10 2019.

8 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

9 Gary Leavens and Yoonsik Cheon. Design by contract with JML, 2006. URL: `https://www.cs.ucf.edu/~leavens/JML/index.shtml`.

10 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, 09 2020.

11 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.

12 Peter W. V. Tran-Jørgensen. Automated translation of VDM-SL to JML-annotated Java. *Technical Report Electronics and Computer Engineering*, 5(29), Mar 2017.

Towards Verified Price Oracles for Decentralized Exchange Protocols

Kinnari Dave
CertiK, USA

Vilhelm Sjöberg
CertiK, USA

Xinyuan Sun
CertiK, 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 engineering, Security and Privacy

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

Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

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* [21], 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.[15, 16]

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 are 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 [18].

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 goes beyond the capabilities of non-interactive theorem proves such as SMT solvers. And even if we can prove theorems about the financial model, we must still show that the actual

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:12
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 the the DeepSEA system, a language 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 the result establishing lower bounds on the cost of manipulating prices quoted by a contract implementing the AMM mechanism to an attacker (Section 4.2). This proof makes use existing third-party math developments (Section 4.1), 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 it's reserves by swapping any number of tokens. (Section 4.2). 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* [1] 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 4.3).

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 instantaneous 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 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.

[1] Available online at `https://github.com/certikfoundation/deepsea/blob/master/contracts/amm/math_lemma.v`

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. 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 pair contract. We focus on the constant product market makers. This type of automated market makers satisfy the invariant:

$$x_A . y_B = k$$

where x_A, y_B are the reserves for token A and token B respectively. The marginal price of A with respect to B is determined to be the ratio of the token reserve of A to that of B. 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 proportion of shares of 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.

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:

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

In the event that liquidity is being deposited to the reserves for the first time, the number of liquidity tokens minted is given by:

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

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 quering 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.

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" 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 [19] to executable code.The original version [17] compiled programs into C, while a new version [9] 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 [11], 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.

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[2] 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. Moroever, 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. 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.

[2] Available online at `https://github.com/certikfoundation/deepsea/blob/master/contracts/amm/amm.ds`

4 Manipulating Prices

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 it's liquidity reserves, and go on to model risk in the constant product market maker model. In this paper, we formalize two of their theorems inside the Coq proof assistant and connect them to the AMM contract written in DeepSEA.

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. 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 an attacker wants to manipulate the Uniswap price, s.t. $m_u = (1+\epsilon)m_p$. Hence we have, $\frac{R_\beta+\Delta_\beta}{R_\alpha-\Delta_\alpha} = (1+\epsilon)m_p$.

The theorem establishes that there is a minimal positive cost to the attacker which is proportional to the reserves of the token added. The cost of manipulation of the Uniswap price to the attacker is calculated as the difference between the amount of tokens the attacker needs to add to the liquidity pools and the value of the tokens they would receive as a result of the exchange. The function is calculated as follows:

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

The lower bounds on $C(\epsilon)$ are given as follows:

$$\forall 0 \le \epsilon \le 1, C(\epsilon) \ge R_\beta \frac{\epsilon^2}{2} inf_{0\le\epsilon'\le 1} C''(\epsilon') = (\frac{1}{32\sqrt{2})})R_\beta\epsilon^2$$

$$\forall \epsilon \ge 1, C(\epsilon) \ge \kappa R_\beta\sqrt{\epsilon}$$

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

We have formally proven these lower bounds in the Coq proof assistant.

Furthermore, we also prove the invariant property that it is impossible to drain the contract of all it's token reserves. The result is stated as follows:

$$R_\alpha + R_\beta > 0$$

Proving this requires establishing the invariant that the product of reserves is strictly increasing over each swap operation.

4.1 Importing third-party Coq libraries

In order to reason about various bounds on expressions used for the proof of the result, we chose to use the Coq-interval [14] 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 [13]) 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, 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.

4.2 Proof Outline

The formalization[3] 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 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. [13] 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 =
```

[3] Available online at https://github.com/certikfoundation/deepsea/blob/master/contracts/amm/cst_man.v

```
  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 $D : nat \to \mathbb{R} \to \mathbb{R}$ represents the series of a function and it's *nth* 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 *nth* order then each of the *kth* order derivatives of the function should be continuous and 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 ϵ 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 (cite) is $1/32\sqrt{2}$. Since the interval tactic works based on approximations [14], 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 on the cost of manipulation of the Uniswap price for exchange of tokens.

To prove the second part, we use a different approach from the one suggested 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 κ by $5/100$ to facilitate the use of the interval tactic. Instead of using the analysis of quadratic equations approach as suggested, 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 both the cases into one result, gives us the following lemma :

```
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 further prove another important property towards making the contract hacker resistant. We show that after performing an arbitrary swap, if the contract had positive balances of both the tokens, it is impossible to drain the contract of all it's reserves, irrespective of the amount of tokens swaped. The result 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.

4.3 Connection to the DeepSEA contract

The results we formalized in the Coq Proof assistant about the cost of manipulation of the Uniswap price are for the AMM contract written in DeepSEA. The *cst_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 α for the input token β ,completed without reverting and left us in a new contract state `s'`.

This is done to calculate Δ_α. 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 ϵ) 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 are formalized for the ϵ obtained from above and its possible values.

5 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 [22]. 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 similar properties.

An alternative to formal proof is to apply model checking [20, 18] or graph search with contraints [23] 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 aggregates data from off-chain sources and posts them on the blockchain. Analyses of the latter [10, 12] show that they, too, have problems with spurious data spikes and possible attacks.

6 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, 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. Theoretical results about such models however still remain to be established. Second, it will be interesting to prove the correctness of a *client* of this 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. *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 arXiv:1911.03380*, 2019.

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. URL: `https://arxiv.org/abs/2012.13230`, `arXiv:2012.13230`.

7 Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch-Lafuente. A theory of automated market makers in defi. *arXiv preprint arXiv:2102.11350*, 2021.

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 CertiK Foundation. DeepSEA. (Accessed on 05/23/2021). URL: `https://github.com/certikfoundation/deepsea`.

10 Wanyun Catherine Gu, Anika Raghuvanshi, and Dan Boneh. Empirical measurements on pricing oracles and decentralized governance for stablecoins. *Available at SSRN 3611231*, 2020.

11 Xavier Leroy. The CompCert verified compiler. `http://compcert.inria.fr/`, 2005–2021.

12 Bowen Liu, Pawel Szalachowski, and Jianying Zhou. A first look into defi oracles. *arXiv preprint arXiv:2005.04377*, 2020. URL: `https://arxiv.org/abs/2005.04377`.

13 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.

14 Guillaume Melquiond. Coq-interval. *Retrieved June*, 17:2017, 2011.

15 miscellanous. Cryptocurrency statistics. Blog post. `https://duneanalytics.com/queries/4494/8769`, 2020.

16 miscellanous. Defi statistics. Blog post. `https://cointelegraph.com/news/defi-hacks-and-exploits-total-285m-since-2019-messari-reports`, 2020.

17 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.

18 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.

19 The Coq Development Team. The Coq proof assistant. `https://coq.inria.fr/`. Accessed: 28/5/2019.

20 Palina Tolmach, Yi Li, Shang-Wei Lin, and Yang Liu. Formal analysis of composable defi protocols. *CoRR*, abs/2103.00540, 2021. URL: `https://arxiv.org/abs/2103.00540`, `arXiv:2103.00540`.

21 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. URL: `https://arxiv.org/abs/2010.12252`, `arXiv:2010.12252`.

22 Daejun Park Yi Zhang, Xiaohong Chen. Formal specification of constant product market maker model and implementation.

23 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 arXiv:2103.02228*, 2021.

Formally Documenting Tenderbake

Formally Documenting Tenderbake

Sylvain Conchon
Nomadic Labs, F-75013 Paris, France

Alexandrina Korneva
Université Paris-Saclay, CNRS, ENS Paris-Saclay, Laboratoire Méthodes Formelles, F-91190 Gif-sur-Yvette, France

Çagdas Bozman
Functori, F-75012 Paris, France

Mohamed Iguernlala
Functori, F-75012 Paris, France

Alain Mebsout
Functori, F-75012 Paris, France

Abstract

In this short 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 Author: Please fill in 1 or more \ccsdesc macro

Keywords and phrases Consensus algorithm, Tezos blockchain, TLA+

Digital Object Identifier 10.4230/OASIcs...

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 aynchronous 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].

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

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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.

In order to guarantee the correctness of the consensus, Tenderbake implements several mechanisms to circumvent Byzantine attacks or asynchrony-related problems. 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$^+$ 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$^+$ automaton has proven useful for reasoning and exchanging with the developers of the actual implementation. The TLA$^+$ model is available at `https://www.lri.fr/~conchon/tenderbake/`.

2 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, called a *baker*, runs the same automaton. We explain how this automaton is implemented in TLA$^+$ in Section 3.

The automaton is given in Figure 1. It represents the evolution of a baker's state and the actions perfomed 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 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 (stricly) less than non-empty variables. We stick to conventional message passing notation where $m(x_1, \ldots, x_k)$? stands for the reception of a message m with parameters $x_1, \ldots, x_k$, and $m(v_1, \ldots, v_k)!$ is the asynchronous broadcast of m with $v_1, \ldots, v_k$ as arguments. Note that when a baker broadcasts a message, he does not send it to himself.

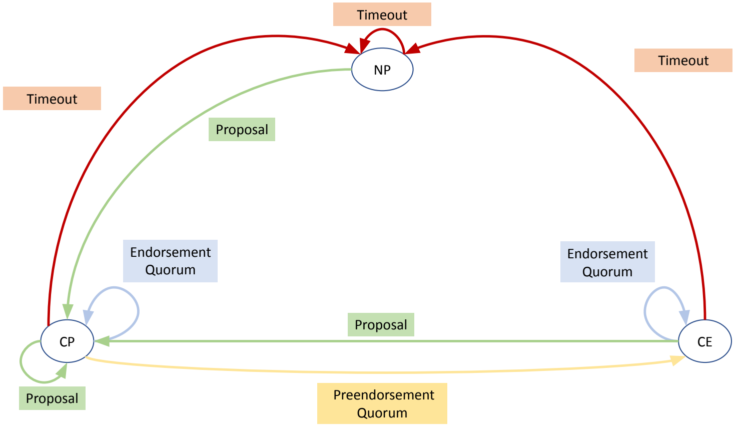

Figure 1 Tenderbake automaton

General architecture. Our modeling follows Tezos's general architecture and constraints. In this architecture, bakers don't communicate directly with each other, but through a *Node* of the network. A node itself is composed of a Peer-to-Peer layer, a validator (which uses the rules of the economic protocol to check blocks and operations), a distributed database, and a specific data structure for pending operations (transactions, votes, etc.), called the *Mempool*. All these components are involved in the consensus mechanism. To clarify the presentation, we equate all of them to the Mempool.

Bakers communicate with the Mempool using RPC (Remote Procedure Calls). To simplify our modeling, we approximate these communications through a shared memory mechanism.

Baker's state. As shown in Figure 1, 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.* $\mathsf{B}.r$). The role of each of these components is as follows:

ℓ	the level of the block in the blockchain, with the convention that the first block of the blockchain, *Genesis*, is at level 0;
r	the consensus round during which the block was proposed;
t	the timestamp of when the block was proposed;
p	the block's payload - i.e. its contents without the consensus operations;
pqc	the preendorsing majority (the first vote), if any, for the proposed block;
eqc	then endorsing majority (certificate for the second vote) for the previous block.

In addition to the two head blocks stored in CH_i, a baker maintains his current consensus round in RND_i. For safety and progress reasons, a baker must also keep track of the block he voted for and for which a preendorsement voting quorum was first observed. A record LOCKED_i of the form $\{\ b;\ q;\ \}$ stores the head block of CH_i (in b) and the list of participants who voted for (preendorsed) this block during the first voting phase (in q). In the same way, a record ELECT_i is used to store the first endorsement quorum for the head block. Finally, in order to speed up the convergence of the algorithm, a similar two-component record PQC_i is used to keep track of the proposed block with the latest round to have achieved a preendorsement quorum.

Time and clocks. Tenderbake runs on the notion of rounds and time. As mentioned in section 1, the ideal consensus scenario isn't 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 hasn't 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 2) 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 TO_i, for *timeout*, used by the oracle to communicate the end of a round to the baker, and an integer TICK_i to count the number of rounds elapsed since the blockchain was started. We also use a constant Δ 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 following guard/action command as soon as (1) the baker i has no timeout to handle (2) the differences between any two bakers' counted rounds doesn't exceed Δ, before and after execution of the transition. The command's action sets the *timout* variable of the baker i to `true` and

Trigger Timeout

$$\neg\mathsf{TO}_i\ \wedge\ (\forall j,k.j \neq k \implies |\mathsf{TICK}_j - \mathsf{TICK}_k| \leq \Delta)\ \wedge\ (\forall j.j \neq i \implies |\mathsf{TICK}_i + 1 - \mathsf{TICK}_j| \leq \Delta) \longrightarrow$$

$$\mathsf{TO}_i := \mathtt{true};$$
$$\mathsf{TICK}_i := \mathsf{TICK}_i + 1;$$

Figure 2 Trigger timeout oracle

increments its tick counter. This transition guarantees that no two bakers can drift for more than Δ 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 1, a baker is forced to move to state NP when the oracle resets his 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 $\mathsf{IsProposer}(i, \ell, r)$ which is true when baker i is the proposer at level ℓ and round r.

Figure 3 (see in the Appendix) contains the possible behaviors (or transitions) of a baker after a timeout. In (A) Not the Proposer, the baker first checks that he *is not* the proposer for the next round $\mathsf{RND}_i + 1$ of the current level $\mathsf{B}.\ell$. Then, either there is no block stored in ELECT_i (denoted by $\mathsf{ELECT}_i = \texttt{-}$), meaning the baker did not obtain a quorum for his head block, or the baker is not the proposer for the next level. In the latter case, instead of $\mathsf{IsProposer}(i, \mathsf{B}.\ell + 1, 0)$, the baker checks for the round $\mathsf{RND}_i - \mathsf{ELECT}_i.b.r$ of the next level $\mathsf{B}.\ell + 1$. This expression takes into account the difference between the baker's current round RND_i and the round during which the baker obtained a quorum for his head block (stored in the ELECT_i variable). Thus, for instance, if a baker obtains a quorum at round $\mathsf{RND}_i = r$, 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 $\mathsf{RND}_i - r = 0$ of the next level. The actions associated to this transition consist only of resetting the TO_i variable and incrementing the counters TICK_i and RND_i.

In (B) Proposer of next round, the baker communicates a proposal $\mathsf{Propose}(i, (b, P))$ for the next round to the Mempool through the variable P_i. The block b is built using the content of the head block B with new timestamp and round information. The payload of this new proposal is either a fresh value (denoted by ε) or the payload of the block stored in the baker's 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 (C) Proposer of next level, the baker must have a block stored in ELECT_i and he must also be the proposer of the round $\mathsf{RND}_i - \mathsf{ELECT}_i.b.r$ in the next level $\mathsf{B}.\ell + 1$. The new proposal contains a fresh payload, an endorsement quorum for its block predecessor taken from $\mathsf{ELECT}_i.q$ 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 ($\mathsf{PreEndorse}$ or $\mathsf{Endorse}$ 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

$(\mathsf{H}.\ell, \mathsf{H}.pqc.r, -\mathsf{PRE}.r, \mathsf{H}.r) < (l, pqc.r, -pre.r, r)$, where H and PRE are the first two head blocks of NodeCH_i, while h and pre are the blocks received in a $\mathsf{Propose}(j, (h, pre))$ message. Moreover, in addition to fitness, the Mempool ensures the information contained in the *eqc* and *pqc* fields is valid. Last, if this better proposal has been received through a shared variable, the Mempool broadcasts it to the other participants. Figure 4 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[1].

Proposal transition. As seen in Figure 1, a baker can handle a new proposal in any state. We give in Figure 5 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 p0 at some round r0, but the current proposal's payload is equal to p0, or the current proposal got a PQC at some round r1 > r0.

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 6. 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 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's about to endorse, and communicate an $\mathsf{Endorse}(i, \mathsf{B}.\ell, \mathsf{B}.r, \mathsf{B}.p)$ 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.

[1] 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.

Initial state. Initially the chain is composed of two genesis blocks (to simplify checks in the automaton). Initial states for both the baker i and the associated Mempool are given below:

Initial state for Baker i	**Initial state for Mempool**
$\mathsf{CH}_i = (Genesis, Genesis)$	$\mathsf{NodeCH}_i = (Genesis, Genesis)$
$\mathsf{RND}_i = 0$	$\mathsf{M}_i = \emptyset$
$\mathsf{TICK}_i = 0$	$\mathsf{P}_i = -$
$\mathsf{LOCKED}_i = Genesis$	$\mathsf{E}_i = -$
$\mathsf{PQC}_i = \{b = Genesis; q = \emptyset\}$	$\mathsf{PE}_i = -$
$\mathsf{ELECT}_i = \{b = Genesis; q = \emptyset\}$	
$\mathsf{TO}_i = \texttt{true}$	

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

Bakers are locked on and have elected the *Genesis* block (with empty quorums) in order to force the progression to go through proposals at level 1.

3 TLA$^+$

In this section we discuss how we go from the previous automaton to its TLA$^+$ implementation. The automaton makes it fairly straightforward to convert to TLA$^+$ 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 2), represented as a record structure. We stray from the types in section 2 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 ⟨ ⟩, and *Genesis* is the genesis block:

$$\begin{aligned} InitialState \triangleq\ & [state \mapsto \texttt{"np"}, pqc \mapsto \langle\rangle, ch \mapsto \langle Genesis, Genesis\rangle, rnd \mapsto 0,\\ & locked \mapsto \langle\rangle, elect \mapsto \langle Genesis, \{\}\rangle, timeout \mapsto \textsc{true}, tick \mapsto 0] \end{aligned} \tag{Initial BakerState}$$

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 2, we introduce an oracle transition which allows bakers to progress individually with timeouts (TO_i) while maintaining synchronization, *i.e.* by being at most Δ rounds apart. We do the same thing in our TLA$^+$ 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$^+$[41]. For example, the Endorsement Quorum step in Figure. 1, enabled in CP or CE, is defined as follows:

$$\begin{aligned}
EndQuorum(i) \triangleq\ & \land\ BakerState[i].timeout = \text{FALSE} \\
& \land\ BakerState[i].elect = \langle\rangle \\
& \land\ BakerState[i].state = \text{"cp"} \lor BakerState[i].state = \text{"ce"} \\
& \land\ CollectEnd(i) \\
& \land\ BakerState' = [BakerState \text{ EXCEPT} \\
& \qquad\qquad !\,[i].elect = \langle BakerState[i].chain[1].round, \\
& \qquad\qquad\qquad\qquad BakerState[i].chain[1].contents, \\
& \qquad\qquad\qquad\qquad BakerState[i].chain[1].time\rangle, \\
& \qquad\qquad !\,[i].state = \ BakerState[i].state] \\
& \land\ \text{UNCHANGED } Mempool
\end{aligned} \tag{Endorsement Quorum}$$

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 that's 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 1 is defined in a similar way.

Test scenarios. While the automaton made writing our TLA$^+$ specification easier, the spec itself has, in return, proven extremely useful in debugging the automaton. Sometimes a deadlock would be reached when it shouldn't 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.

4 Conclusion

In this paper we proposed a TLA$^+$ 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, Ilina 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*, 2014.

6 Jae Kwon and Ethan Buchman. Cosmos whitepaper, 2019.

A Automaton Transitions

Not Proposer

```
$\mathsf{CH}_i = (\mathsf{B}, \_)$

$\mathsf{TO}_i \wedge$
$\neg\ \mathit{isProposer}(i, \mathsf{B}.\ell, \mathsf{RND}_i + 1) \wedge$
$(\mathsf{ELECT}_i = - \vee\ \neg\ \mathit{isProposer}(i, \mathsf{B}.\ell + 1, \mathsf{RND}_i - \mathsf{ELECT}_i.b.r)) \longrightarrow$
    $\mathsf{TO}_i := \texttt{false}$
    $\mathsf{RND}_i := \mathsf{RND}_i + 1$
```

(a) Not the proposer

Proposer Next Round

```
$\mathsf{CH}_i = (\mathsf{B}, \mathsf{P})$

$\mathsf{TO}_i \wedge$
$\mathit{isProposer}(i, \mathsf{B}.\ell, \mathsf{RND}_i + 1) \wedge$
$(\mathsf{ELECT}_i = -\ \vee\ \neg\ \mathit{isProposer}(i, \mathsf{B}.\ell + 1, \mathsf{RND}_i - \mathsf{ELECT}_i.b.r)) \longrightarrow$
    $\mathsf{TO}_i := \texttt{false}$
    $\mathsf{RND}_i := \mathsf{RND}_i + 1$
    $\textbf{let}\ p, pqc = \textbf{if}\ (\mathsf{PQC}_i = -)\ \textbf{then}\ (\varepsilon, -)\textbf{else}\ (\mathsf{B}.p, \mathsf{PQC}_i)\ \textbf{in}$
    $\textbf{let}\ \mathsf{b} = \{\mathsf{B}\ \textbf{with}\ t = \mathsf{TICK}_i; r = \mathsf{RND}_i; p; pqc\}\ \textbf{in}$
    $\mathsf{P}_i := \mathsf{Propose}(i, (\mathsf{b}, \mathsf{P}))$
```

(b) Proposer of next round

Proposer Next Level

```
$\mathsf{CH}_i = (\mathsf{B}, \_)$

$\mathsf{TO}_i \wedge$
$\mathsf{ELECT}_i \neq - \wedge$
$\mathit{isProposer}(i, \mathsf{B}.\ell + 1, \mathsf{RND}_i - \mathsf{ELECT}_i.b.r) \longrightarrow$
    $\mathsf{TO}_i := \texttt{false}$
    $\textbf{let}\ \mathsf{b} = \{\ell = \mathsf{B}.\ell + 1;\ t = \mathsf{TICK}_i;\ p = \varepsilon;\ r = \mathsf{RND}_i - \mathsf{ELECT}_i.b.r;$
            $eqc = \mathsf{ELECT}_i.q;\ pqc = -\}\ \textbf{in}$
    $\mathsf{RND}_i := \mathsf{RND}_i + 1$
    $\mathsf{P}_i := \mathsf{Propose}(i, (\mathsf{b}, \mathsf{ELECT}_i.b))$
```

(c) Proposer of next level

Figure 3 A baker's possible actions once `timeout` has been reset

```
Mempool

$\mathsf{NodeCH}_i = (\mathsf{H}, \mathsf{PRE})$

$\mathsf{PreEndorse}(j, \ell, r, p)? \vee \mathsf{PE}_i = \mathsf{PreEndorse}(j, \ell, r, p) \longrightarrow$
    $\mathsf{M}_i := \mathsf{M}_i + \mathsf{PreEndorse}(j, \ell, r, p)$
    $\mathsf{PE}_i \neq - \longrightarrow$
        $\mathsf{PreEndorse}(i, \ell, r, p)!$
        $\mathsf{PE}_i := -$

$\mathsf{Endorse}(j, \ell, r, p)? \vee \mathsf{E}_i = \mathsf{Endorse}(j, \ell, r, p) \longrightarrow$
    $\mathsf{M}_i := \mathsf{M}_i + \mathsf{Endorse}(j, \ell, r, p)$
    $\mathsf{E}_i \neq - \longrightarrow$
        $\mathsf{Endorse}(i, \ell, r, p)!$
        $\mathsf{E}_i := -$

$\mathsf{Propose}(j, (h, pre))? \vee \mathsf{P}_i = \mathsf{Propose}(j, (h, pre)) \longrightarrow$
    $\textbf{let } \{\ell; r; pqc; eqc\} = h \textbf{ in}$
    $isProposer(j, \ell, r) \ \wedge \ (\mathsf{H}.\ell, \mathsf{H}.pqc.r, -\mathsf{PRE}.r, \mathsf{H}.r) < (\ell, pqc.r, -pre.r, r) \ \wedge$
    $valid\_eqc(eqc, pre) \ \wedge \ (pqc = - \vee valid\_pqc(b)) \longrightarrow$
        $\mathsf{NodeCHi} := (h, pre)$
        $\mathsf{P}_i \neq - \longrightarrow$
            $\mathsf{Propose}(i, (h, pre))!$
            $\mathsf{P}_i := -$
```

where the predicate *valid_eqc* ensures the *eqc* contained in the proposal is valid with respect to the previous block *pre*, and the predicate *valid_pqc* ensures the *pqc* of the proposed block (only in the case of a re-proposal) is valid, *i.e.* is for the current payload and on a previous round of the same level. We give here their definitions:

$$quorum(x) \triangleq |x| > \frac{2 \times |\mathsf{BAKERS}|}{3}$$

$$\begin{aligned} valid\_eqc(eqc, pre) \triangleq \ & quorum(eqc.q) \ \wedge \\ & \forall\ \mathsf{Endorse}(i, \ell, r, p) \in eqc.q. \\ & \quad p = pre.p \ \wedge \ \ell = pre.\ell \ \wedge \ r = pre.r \end{aligned}$$

$$\begin{aligned} valid\_pqc(b) \triangleq \ & quorum(b.pqc.q) \ \wedge \\ & \forall\ \mathsf{PreEndorse}(i, \ell, r, p) \in b.pqc.q. \\ & \quad p = b.p \ \wedge \ \ell = b.\ell \ \wedge \ r < b.r \end{aligned}$$

Proposers are chosen in a round robin with the formula:

$$isProposer(i, \ell, r) \triangleq ((\ell + r) \mod |\mathsf{BAKERS}|) + 1 = i$$

Figure 4 Mempool transitions

```
Proposal

$\mathsf{CH}_i = (\mathsf{B}, \_)$
$\mathsf{NodeCH}_i = (\mathsf{H}, \mathsf{PRE})$

$\neg\mathsf{TO}_i \;\wedge\; \mathsf{B} \neq \mathsf{H} \longrightarrow$
    $\mathbf{let}\{\ell; r; p; pqc; eqc\} = \mathsf{H}\ \mathbf{in}$

    $\mathsf{B}.\ell < \ell \longrightarrow$
        $\mathsf{RND}_i := \mathsf{TICK}_i - (\mathsf{PRE}.t + 1)$
        $\mathsf{LOCKED}_i := -; \mathsf{PQC}_i := -; \mathsf{ELECT}_i := -$
        $\mathsf{RND}_i = r \longrightarrow \mathsf{PE}_i := \mathsf{PreEndorse}(i, \ell, r, b)$
        $\mathsf{CH}_i := (\mathsf{H}, \mathsf{PRE})$
        $\mathsf{PQC}_i.r^? < pqc.r^? \longrightarrow \mathsf{PQC}_i := pqc$

    $\ell = \mathsf{B}.\ell \;\wedge\; r < \mathsf{RND}_i \wedge \mathsf{B}.r < r \longrightarrow$
        $\mathsf{CH}_i := (\mathsf{H}, \mathsf{PRE})$
        $\mathsf{PQC}_i.r^? < pqc.r^? \longrightarrow \mathsf{PQC}_i := pqc$

    $\ell = \mathsf{B}.\ell \;\wedge\; r = \mathsf{RND}_i \;\wedge\; (r \neq \mathsf{B}.r \;\vee\; p = \mathsf{B}.p) \longrightarrow$
        $\mathsf{CH}_i := (\mathsf{H}, \mathsf{PRE})$
        $\mathsf{PQC}_i.r^? < pqc.r^? \longrightarrow \mathsf{PQC}_i := pqc$
        $\mathsf{LOCKED}_i = - \;\vee\; \mathsf{LOCKED}_i.p = \mathsf{B}.p \;\vee$
        $(pqc.r \neq - \;\wedge\; \mathsf{LOCKED}_i.r \leq pqc.r) \longrightarrow$
            $\mathsf{PE}_i := \mathsf{PreEndorse}(i, \ell, r, p)$
```

Figure 5 Receiving a proposal

```
Preendorsement Quorum

$\mathsf{CH}_i = (\mathsf{B}, \_)$

let $q = \{m \in \mathsf{M}_i \mid m = \mathsf{PreEndorse}(\_, \ell, r, p) \;\wedge\; \ell = \mathsf{B}.\ell \;\wedge\; p = \mathsf{B}.p \;\wedge\; r = \mathsf{RND}_i = \mathsf{B}.r\}$ in
$\neg\mathsf{TO}_i \;\wedge\; quorum(q) \;\wedge\; \mathsf{LOCKED}_i \neq \mathsf{B} \longrightarrow$
    $\mathsf{PQC}_i := \{b = \mathsf{B}; q = q\}$
    $\mathsf{LOCKED}_i := \mathsf{B}$
    $\mathsf{E}_i := \mathsf{Endorse}(i, \mathsf{B}.\ell, \mathsf{B}.r, \mathsf{B}.p)$
```

```
Endorsement Quorum

$\mathsf{CH}_i = (\mathsf{B}, \_)$

$\mathbf{let}\ q = \{m \in \mathsf{M}_i \mid m = \mathsf{Endorse}(\_, \ell, r, p) \;\wedge\; \ell = \mathsf{B}.\ell \;\wedge\; p = \mathsf{B}.p \;\wedge\; r = \mathsf{RND}_i = \mathsf{B}.r\}\ \mathbf{in}$
$\neg\mathsf{TO}_i \;\wedge\; quorum(q) \;\wedge\; \mathsf{ELECT}_i = - \longrightarrow$
    $\mathsf{ELECT}_i := \{b = \mathsf{B}; q = q\}$
```

Figure 6 Preendorsement and endorsement quorums

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

Daniel Britten
The University of Waikato, New Zealand

Vilhelm Sjöberg
CertiK, USA

Steve Reeves
The University of Waikato, 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. 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.CVIT.2016.23

Supplementary Material `https://github.com/Coda-Coda/deepsea-1/tree/fmbc-2021`

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"[9] and others[2, 3, 4]) 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 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.

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:8
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 are 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[6] 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, though there are some false negatives 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[1] proposition formalising the notion of a smart contract function following the *Checks-Effects-Interactions Pattern*. This is discussed in Subsection 2.4.
- Automatic proofs related to the previous contribution as well as related automated proofs that the lists of transfers 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[6] is an up and coming framework and smart contract language that promises to provably link high-level specifications in Coq[1] 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.

Listing 3 ‘Safe’ function in different representations with similarities highlighted.

```
DeepSEA smart contract source code (not Coq):
let safeExample() =
  transferSuccessful := true ;
  transferEth(msg_sender, 0u42)
DeepSEA Intermediate Level Language in Coq:
(CCsequence
(CCstore
  (LCvar Contract_transferSuccessful := true_var)
  (ECconst_int256 tint_bool true Int256.one))
(CCtransfer
  (@ECbuiltin0 _ _  _ builtin0_caller_impl)
  (ECconst_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)
```

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 [11]. 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 *Checks-Effects-Interactions Pattern*.

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 *Checks-Effects-Interactions Pattern*. 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 *Checks-Effects-Interactions Pattern*. The 🟢 icon indicates that the contract cannot in any way have triggered reentrancy yet and the 🟠 icon indicates that reentrancy may have been

Figure 1 Selected typing rules for a command that adheres to the *Checks-Effects-Interactions Pattern*. See the appendix for more.

$$\frac{\{\rho_1\}\ c_1\{\rho_2\} \qquad \{\rho_2\}\ c_2\ \{\rho_3\}}{\{\rho_1\}\ \texttt{let}\ x = c_1\ \texttt{in}\ c_2\ \{\rho_3\}} \qquad \frac{}{\{\text{🟢}\}\ \texttt{load}\ \{\text{🟢}\}} \qquad \frac{}{\{\text{🟢}\}\ e_1 := e_2\ \{\text{🟢}\}}$$

$$\frac{\{\rho_1\}\ c_1\{\rho_2\} \qquad \{\rho_2\}\ c_2\{\rho_3\}}{\{\rho_1\}\ c_1\ ;\ c_2\ \{\rho_3\}} \qquad \frac{\{\text{🟢}\}\ c_{true}\{\text{🟢}\} \qquad \{\text{🟢}\}\ c_{false}\{\text{🟠}\}}{\{\text{🟢}\}\ \texttt{if}\ e_1\ \texttt{then}\ c_{true}\ \texttt{else}\ c_{false}\ \{\text{🟠}\}}$$

$$\frac{\{\rho\}\ c\ \{\rho\}}{\{\rho\}\ \texttt{for}\ e_1\ \texttt{to}\ e_2\ \texttt{do}\ c\ \{\rho\}} \qquad \frac{}{\{\text{🟢}\}\ \texttt{transferEth}(e_1, e_2)\ \{\text{🟠}\}}$$

Listing 4 Defining *Checks-Effects-Interactions Pattern* 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. *)
```

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 allowable 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 ρ (either 🟢 or 🟠) then the for loop as a whole is also defined to stay at state ρ.

The remaining definitions are available in the GitHub repository[1]. This defines what it means for a DeepSEA smart contract function to follow the *Checks-Effects-Interactions Pattern*. To be precise, if *cmd_constr_CEI_pattern_prf* can be proven for a given function then that function follows the *Checks-Effects-Interactions Pattern*.

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 *Checks-Effects-Interactions Pattern* was indeed followed, this formulation would not allow those functions to be proved to be safe. This does however simplify proof automation.

Another drawback is that other techniques to manage reentrancy issues such as locks are not considered to be safe by this method, 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 the case when no reentrancy at all occurs.

[1] `https://github.com/Coda-Coda/deepsea-1/tree/fmbc-2021` - See README for the specific files relevant to this paper.

Listing 5 Defining *Checks-Effects-Interactions Pattern* 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 *Checks-Effects-Interactions Pattern*.

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

3 Automatically proving the absence of reentrancy situations

Now that we have defined the notion of a smart contract following the *Checks-Effects-Interactions Pattern* the goal is to automatically prove this for every function that does indeed follow the *Checks-Effects-Interactions Pattern* (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[2] 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 *Checks-Effects-Interactions Pattern* was not followed).

4 Simplifying the modelling of Ether transfer

Since the *Checks-Effects-Interactions Pattern* has been followed we can now prove in Coq that no more than one transfer occurs (since that would violate the *Checks-Effects-Interactions Pattern*). Using these proofs we can simplify the modelling of Ether transfer as we now only need to consider the case that at most one transfer has occurred.

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 occurred 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

[2] 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.
```

related to multiple transfers. If we did not have the proof we would be stuck 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). It would also be possible in theory to fully model reentrant calls but clearly this would not simplify the model.

The relevant proofs that each smart contract function makes at most one transfer are similar to the proofs about the *Checks-Effects-Interactions Pattern* 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 *Checks-Effects-Interactions Pattern*.

5 Related Work

A number of other tools aim to tackle the problem of reentrancy, such as [7, 8, 5] and [10]. This work is unique in that it explicitly makes use of proofs related to the *Checks-Effects-Interactions Pattern* 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 *Checks-Effects-Interactions Pattern* (code available on GitHub[3]). This is demonstrated by defining an inductive proposition in Coq that states that a particular smart contract function follows the *Checks-Effects-Interactions Pattern*. 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.

[3] `https://github.com/Coda-Coda/deepsea-1/tree/fmbc-2021` - See README for the specific files relevant to this paper.

References

1 The Coq proof assistant. `https://coq.inria.fr/`. (Accessed on 05/23/2021).

2 DeFi platform dForce hacked for $25m - ERC777 reentrancy attack. `https://defirate.com/dforce-hack/`. (Accessed on 05/23/2021).

3 Living in a lego house: The imBTC DeFi hack explained. `https://www.zengo.com/imbtc-defi-hack-explained/`. (Accessed on 05/23/2021).

4 The reentrancy strikes again — the case of Lendf.Me. `https://valid.network/post/the-reentrancy-strikes-again-the-case-of-lendf-me`. (Accessed on 05/23/2021).

5 Remix - Ethereum IDE. `http://remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.1+commit.df193b15.js`. (Accessed on 05/23/2021).

6 CertiK Foundation. DeepSEA. (Accessed on 05/23/2021). URL: `https://github.com/certikfoundation/deepsea`.

7 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.

8 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.

9 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, January 2019.

10 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.

11 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.

A Appendix

Figure 2 Typing rule for a command that adheres to the *Checks-Effects-Interactions Pattern*, corresponding to the Coq inductive proposition `cmd_constr_CEI_pattern_prf`.
(Some rarely used rules have been omitted).

$$\frac{}{\{\rho\}\ \texttt{skip}\ \{\rho\}} \qquad \frac{\{\rho_1\}\ c_1\{\rho_2\} \qquad \{\rho_2\}\ c_2\ \{\rho_3\}}{\{\rho_1\}\ \texttt{let}\ x = c_1\ \texttt{in}\ c_2\ \{\rho_3\}} \qquad \frac{}{\{\text{🟢}\}\ \texttt{load}\ \{\text{🟢}\}} \qquad \frac{}{\{\text{🟢}\}\ e_1 := e_2\ \{\text{🟢}\}}$$

$$\frac{\{\rho_1\}\ c_1\{\rho_2\} \qquad \{\rho_2\}\ c_2\{\rho_3\}}{\{\rho_1\}\ c_1\ ;\ c_2\ \{\rho_3\}} \qquad \frac{\{\text{🟢}\}\ c_{true}\{\text{🟢}\} \qquad \{\text{🟢}\}\ c_{false}\{\text{🟢}\}}{\{\text{🟢}\}\ \texttt{if}\ e_1\ \texttt{then}\ c_{true}\ \texttt{else}\ c_{false}\ \{\text{🟢}\}}$$

$$\frac{\{\text{🟢}\}\ c_{true}\{\text{🟠}\} \qquad \{\text{🟢}\}\ c_{false}\{\text{🟢}\}}{\{\text{🟢}\}\ \texttt{if}\ e_1\ \texttt{then}\ c_{true}\ \texttt{else}\ c_{false}\ \{\text{🟠}\}} \qquad \frac{\{\text{🟢}\}\ c_{true}\{\text{🟢}\} \qquad \{\text{🟢}\}\ c_{false}\{\text{🟠}\}}{\{\text{🟢}\}\ \texttt{if}\ e_1\ \texttt{then}\ c_{true}\ \texttt{else}\ c_{false}\ \{\text{🟠}\}}$$

$$\frac{\{\text{🟢}\}\ c_{true}\{\text{🟠}\} \qquad \{\text{🟢}\}\ c_{false}\{\text{🟠}\}}{\{\text{🟢}\}\ \texttt{if}\ e_1\ \texttt{then}\ c_{true}\ \texttt{else}\ c_{false}\ \{\text{🟠}\}} \qquad \frac{\{\text{🟠}\}\ c_{true}\{\text{🟠}\} \qquad \{\text{🟠}\}\ c_{false}\{\text{🟠}\}}{\{\text{🟠}\}\ \texttt{if}\ e_1\ \texttt{then}\ c_{true}\ \texttt{else}\ c_{false}\ \{\text{🟠}\}}$$

$$\frac{\{\rho\}\ c\ \{\rho\}}{\{\rho\}\ \texttt{for}\ e_1\ \texttt{to}\ e_2\ \texttt{do}\ c\ \{\rho\}} \qquad \frac{\{\rho_1\}\ \textit{function}\ \{\rho_2\}}{\{\rho_1\}\ \textit{function call}\ \{\rho_2\}}$$

$$\frac{}{\{\text{🟢}\}\ \texttt{transferEth}(e_1, e_2)\ \{\text{🟠}\}} \qquad \frac{\{\rho\}\ c\ \{\rho\}}{\{\rho\}\ \texttt{assert}\ c\ \{\rho\}} \qquad \frac{\{\rho\}\ c\ \{\rho\}}{\{\rho\}\ \texttt{deny}\ c\ \{\rho\}}$$

Part II.

Accepted Extended Abstracts

Towards a Theory of Decentralized Finance

Massimo Bartoletti[1], James Hsin-yu Chiang[2], Alberto Lluch Lafuente[2]

[1] Università degli Studi di Cagliari, Cagliari, Italy
[2] Technical University of Denmark, DTU Compute, Copenhagen, Denmark

Abstract. Decentralized Finance (DeFi) has brought about decentralized applications which allow untrusted users to lend, borrow and exchange crypto-assets or crypto-derivatives. Many of such applications fulfill the role of markets or market makers, and thus feature complex, highly parametric incentive mechanisms to equilibrate interest rates or prices. This complexity makes the behaviour of DeFi archetypes difficult to understand and to predict: indeed, ineffective incentives and attacks could potentially lead to emergent unwanted behaviours. Reasoning about DeFi applications is made even harder by the lack of executable models of their behaviour: to precisely understand how users interact with DeFi instances, eventually one has to inspect their different implementations, where the incentive mechanisms are intertwined with low-level implementation details. To this end, we are developing new executable specifications of the DeFi archetypes lending pools and automatic market makers, allowing us to prove universal properties and precisely describe their interactions, vulnerabilities and attacks. In this presentation, we introduce this new line of research which aims to bridge the research communities of economic analysis and formal methods, paving the way for the development of formally secure, domain specific languages for DeFi.

1 DeFi Archetypes and their Formalization

The emergence of permissionless, public blockchains has given birth to an entire ecosystem of *crypto-tokens* representing digital assets and derivatives. Facilitated and accelerated by smart contracts and standardized token interfaces [1], these so-called *decentralized finance* (DeFi) applications promise an open alternative to the traditional financial system. Prior foundational research in the domain of DeFi has been thoroughly summarized in [11].

To study properties emerging from the interaction between users and DeFi applications, we have initiated our line of research towards a theory of DeFi by focusing on the identification of *archetypal* DeFi applications and on the development of *executable specifications* for them, based on manual inspection of the underlying implementations of mainstream implementations. Our formal specifications encompass (abstractions of) the underlying economic incentive mechanisms [6, 9, 10] and pave the way towards a generalized theory of DeFi archetypes and their interactions, which may be intractable from analysis at the implementation level alone. These executable semantics represents a first step towards *domain-specific languages* for decentralized finance, where DeFi contracts are composed from formally specified primitives and thus exhibit well-defined, analyzable behaviour inferred from the language semantics. The main archetypes we have considered so far are *Lending Pools* (LPs) [8] and *Automatic Market Makers* (AMMs) [7].

As of May 2021, the two leading LP platforms hold \$18B [3] and \$14B [2] worth of tokens in their smart contracts, whereas the two leading AMM platforms hold \$8.3B and \$7.7B worth of tokens, and process \$1.4B and \$270M worth of transactions daily [4, 5].

Lending Pools Lending pools are decentralized applications which allow mutually untrusted users to lend and borrow crypto-assets. In [8], we formalize all interactions between users and LPs, thereby providing a complete specification for the economic functionality of LPs. Our model allows to formally state and specify fundamental properties of LPs, like e.g. correct accounting of *minted* tokens and preservation of the supply of *deposited* tokens, which are crucial to ensure consistency in exchange and distribution of tokens enabled by LPs. Furthermore, our model allows one to reason about rational agents, which are incentivized to liquidate loans in return for discounted collateral or perform deposits immediately prior to interest accrual. We also provide solid arguments for the design of incentives of LPs, for example by formally proving that depositors can potentially redeem more tokens than they deposited, and by identifying the conditions under which redeems are not possible. In this regard, we formalize notions of *utilization safety,* which represents a utility trade-off between borrow and redeem actions, moderated by a dynamic interest rate. In LPs, loans are secured by collateral: yet, there exist LP states in which the borrower is no longer incentivized to return loan should the agent's collateralization drop below a certain threshold. We formally characterize such *collateral-safe* states. Finally, we exploit both notions of safety to illustrate attacks on utilization and collateralization, aimed at undermining the incentive mechanisms of LPs.

Automatic Market Makers Automatic market makers allow users to exchange units of different types of crypto-assets, without the need to find a counter-party. In [7], we develop a theory for AMMs, specifying their possible interactions and their economic mechanisms. One of the results we provide is a concurrency theory for AMM actions. In particular, we show that sequences of *deposit* and *redeem* actions can be ordered interchangeably, resulting in observationally equivalent AMM states. We prove fundamental preservation properties for our AMM specification, like e.g. the preservation of deposited token supplies, and *token liquidity,* which ensures that deposited tokens cannot be *frozen* in an AMM application. Furthermore, we introduce a formal notion of *incentive-consistency*: AMM's rely on a dynamic exchange rate governed by a so-called *trading invariant.* Notably, we formalize the key incentive mechanism, the arbitrage game, for all trading invariants which are *incentive-consistent*, thus facilitating formal analysis of AMM behaviour which can be generalized beyond the mainstream constant-product AMMs.

2 Proposed talking points

If accepted, this presentation will provide an example-driven introduction to our formal models of lending pools [8] and automatic market makers [7], which precisely describe their interactions as transitions of a state machine. Our model exhibits the typical user interactions with such archetypes, and all the main economic features. More specifically, our proposed presentation will include:

1. selected fundamental behavioural properties of our formal models of LP's and AMM's, some of which were informally stated in literature, and are expected to be satisfied by any implementation;
2. key aspects of the formalization of LP and AMM incentive mechanisms, and a discussion of their properties, vulnerabilities and attacks;
3. a preliminary discussion on the interplay between LP's, AMM's and other DeFi archetypes, like stable coins and margin trading contracts;
4. the opportunities of bridging economic analysis and formal methods research in decentralized finance.

References

1. ERC-20 token standard
2. Aave markets website (2021), https://app.aave.com/markets
3. Compound markets website (2021), https://compound.finance/markets
4. Curve statistics (2021), https://www.curve.fi/dailystats
5. Uniswap v2 statistics (2021), https://v2.info.uniswap.org
6. Angeris, G., Evans, A., Chitra, T.: When does the tail wag the dog? Curvature and market making. arXiv preprint arXiv:2012.08040 (2020), https://arxiv.org/pdf/2012.08040
7. Bartoletti, M., Chiang, J.H., Lluch-Lafuente, A.: A theory of Automated Market Makers in DeFi. In: COORDINATION (2021), (to appear) https://arxiv.org/abs/2102.11350
8. Bartoletti, M., Chiang, J.H., Lluch-Lafuente, A.: Sok: Lending pools in decentralized finance. In: 5thWorkshop on Trusted Smart Contracts (2021), (to appear) https://arxiv.org/abs/2012.13230
9. Evans, A., Angeris, G., Chitra, T.: Optimal Fees for Geometric Mean Market Makers (2021), https://web.stanford.edu/~guillean/papers/g3m-optimal-fee.pdf
10. Gudgeon, L., Werner, S., Perez, D., Knottenbelt, W.J.: Defi protocols for loanable funds: Interest rates, liquidity and market efficiency. In: ACM Conference on Advances in Financial Technologies. pp. 92–112 (2020). https://doi.org/10.1145/3419614.3423254
11. Werner, S.M., Perez, D., Gudgeon, L., Klages-Mundt, A., Harz, D., Knottenbelt, W.J.: Sok: Decentralized finance (defi) (2021)

A formal model of Algorand smart contracts (extended abstract)

Massimo Bartoletti[1], Andrea Bracciali[2], Cristian Lepore[2],
Alceste Scalas[3], Roberto Zunino[4]

[1] Università degli Studi di Cagliari, Cagliari, Italy
[2] Stirling University, Stirling, UK
[3] Technical University of Denmark, Lyngby, Denmark
[4] Università degli Studi di Trento, Trento, Italy

Abstract. Algorand is a late-generation blockchain that features several interesting features, including high-scalability and a no-forking consensus protocol based on Proof-of-Stake. Besides this, Algorand features a complex transaction mechanisms, which supports both stateless and stateful smart contracts. Although this mechanism is already used in practice to develop industrial use cases, it is still largely unexplored by the research community. In this extended abstract we summarize our ongoing work on formal modelling of Algorand smart contracts.

1 Introduction

Smart contracts are agreements between two or more parties that are automatically enforced without trusted intermediaries. Blockchain technologies reinvented the idea of smart contracts, providing trustless environments where they are incarnated as computer programs. However, writing secure smart contracts is difficult, as witnessed by the multitude of attacks on smart contracts platforms (notably, Ethereum) — and since smart contracts control assets, their bugs may directly lead to financial losses.

Algorand [10] is a late-generation blockchain that features a set of interesting features, including high-scalability and a no-forking consensus protocol based on Proof-of-Stake [6]. Its smart contract layer (ASC1) aims to mitigate smart contract risks, and adopts a non-Turing-complete programming model, natively supporting atomic sets of transactions and user-defined assets. These features make it an intriguing smart contract platform to study.

The official specification and documentation of ASC1 consists of English prose and a set of templates to assist programmers in designing their contracts [1,3]. This conforms to standard industry practices, but there are two drawbacks:

1. Algorand lacks a mathematical model of contracts and transactions suitable for formal reasoning on their behaviour, and for the verification of their properties. Such a model is needed to develop techniques and tools to ensure that contracts are correct and secure;

2. furthermore, even preliminary informal reasoning on non-trivial smart contracts can be challenging, as it may require, in some corner cases, to resort to experiments, or direct inspection of the platform source code, in the lack of a more abstract model.

Given these drawbacks, in [8] we have introduced a formal model of Algorand smart contracts that:

o1. is high-level enough to simplify the design of Algorand smart contracts and enable formal reasoning about their security properties;
o2. expresses Algorand contracts in a simple declarative language, similar to PyTeal (the official Python binding for Algorand smart contracts) [5];
o3. provides a basis for the automatic verification of Algorand smart contracts.

More in detail, our formal framework features:

- an *executable semantics of stateless ASC1* providing a solid theoretical foundation to Algorand smart contracts. Such a model formalises both Algorand accounts and transactions, and stateless smart contracts;
- a validation of our model through experiments [4] on the Algorand platform;
- the formalisation and proof of some *fundamental properties of the Algorand state machine*: no double spending, determinism, value preservation;
- an *analysis of Algorand contract design patterns*, based on several non-trivial contracts (covering both standard use cases, and novel ones). Quite surprisingly, we have shown that stateless contracts are expressive enough to encode finite state machines;
- the proof of relevant *security properties of smart contracts* in our model;
- a *prototype tool* that compiles smart contracts (written in our formal declarative language) into TEAL, the bytecode language interpreted by Algorand nodes.

Ongoing work and future directions Perhaps, the most interesting current effort is to adapt the formal model to the ongoing evolution of the Algorand framework. A notable development is that, in mid August 2020, Algorand has introduced *stateful* ASC1 contracts [2], which provide the contracts execution environment with a persistent key-value store, accessible and modifiable through a new kind of transaction (which can use an extended set of TEAL opcodes). This extension appears to increase the expressivity of Algorand smart contracts in a way that poses new challenges for the formalisation and verification of their security.

The stateful paradigm leads to a further research direction: investigating declarative languages for stateful Algorand smart contracts, and their associated verification techniques.

Another relevant research direction is the mechanization of our formal model, e.g. through verification techniques such as proof assistants or model checking. this would allow for machine-checking the proofs developed by pencil-and-paper in [7, §D]. Similar work has been done, e.g., for Bitcoin [11] and for Tezos [9].

By objectives o1–o3, our formal model strives at being high-level and simple to understand, while faithfully describing the most commonly used primitives and mechanisms of Algorand, and supporting the specification and verification of non-trivial smart contracts. To achieve these objectives, we have introduced some high-level abstractions over low-level details: e.g., since TEAL code has the purpose of accepting or rejecting transactions, we model it using expressions that evaluate to *true* or *false*. Hence, another possible direction for future research is to cover all the possible TEAL contracts with bytecode-level accuracy, as well as all the low-level details of Algorand transactions.

References

1. Algorand developer docs (2020), https://developer.algorand.org/docs/
2. Algorand developer docs: stateful smart contracts (2020), https://developer.algorand.org/docs/features/asc1/stateful/
3. Algorand developer docs: Transaction Execution Approval Language (TEAL) (2020), https://developer.algorand.org/docs/reference/teal
4. ASC1 coherence-checking experiments (2020), https://github.com/blockchain-unica/asc1-experiments
5. PyTeal: Algorand smart contracts in Python (2020), https://github.com/algorand/pyteal
6. Alturki, M.A., Chen, J., Luchangco, V., Moore, B.M., Palmskog, K., Peña, L., Rosu, G.: Towards a verified model of the Algorand consensus protocol in Coq. In: Formal Methods Workshops. Lecture Notes in Computer Science, vol. 12232, pp. 362–367. Springer (2019). https://doi.org/10.1007/978-3-030-54994-7_27
7. Bartoletti, M., Bracciali, A., Lepore, C., Scalas, A., Zunino, R.: A formal model of Algorand smart contracts. CoRR **abs/2009.12140v3** (2020), https://arxiv.org/abs/2009.12140v3
8. Bartoletti, M., Bracciali, A., Lepore, C., Scalas, A., Zunino, R.: A formal model of Algorand smart contracts. In: Financial Cryptography (2021), (to appear) https://arxiv.org/abs/2009.12140
9. Bernardo, B., Cauderlier, R., Hu, Z., Pesin, B., Tesson, J.: Mi-Cho-Coq, a framework for certifying Tezos smart contracts. In: Sekerinski, E., Moreira, N., Oliveira, J.N., Ratiu, D., Guidotti, R., Farrell, M., Luckcuck, M., Marmsoler, D., Campos, J., Astarte, T., Gonnord, L., Cerone, A., Couto, L., Dongol, B., Kutrib, M., Monteiro, P., Delmas, D. (eds.) Workshop on Formal Methods for Blockchains. LNCS, vol. 12232, pp. 368–379. Springer (2019). https://doi.org/10.1007/978-3-030-54994-7_28
10. Chen, J., Micali, S.: Algorand: A secure and efficient distributed ledger. Theoretical Computer Science **777**, 155–183 (2019). https://doi.org/10.1016/j.tcs.2019.02.001
11. Rupić, K., Rozic, L., Derek, A.: Mechanized formal model of Bitcoin's blockchain validation procedures. In: Workshop on Formal Methods for Blockchains (FMBC@CAV). OASIcs, vol. 84, pp. 7:1–7:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/OASIcs.FMBC.2020.7

A Technique For Analysing Permissionless Blockchain Protocols

Stefano Bistarelli
University of Perugia, Perugia, Italy

Rocco De Nicola
IMT School for Advanced Studies, Lucca, Italy

Letterio Galletta
IMT School for Advanced Studies, Lucca, Italy

Cosimo Laneve
University of Bologna, Bologna, Italy

Ivan Mercanti
IMT School for Advanced Studies, Lucca, Italy

Adele Veschetti
University of Bologna, Bologna, Italy

Abstract

In this work we present our research with the main goal of formally model and analyze the main blockchain consensus protocols. We have analyzed the protocol of the Bitcoin blockchain by using the PRISM probabilistic model checker. The probabilistic analysis of the model highlights how forks happen and how they depend on specific parameters of the protocol, such as the difficulty of the cryptopuzzles and the network communication delays. We also study the behaviour of networks with churn miners, which may leave the network and rejoin afterwards, and with different topologies.

2012 ACM Subject Classification Theory of computation → Formalisms

Keywords and phrases Bitcoin, blockchain, security, CTMCs, PRISM, fork, Casper

Digital Object Identifier 10.4230/OASIcs...

1 Introduction

The main goal of our research is to formally model and analyze the main blockchain consensus protocols. In particular, we adopt the approach of [10, 6], and we model the network of nodes as the *parallel composition* of processes and the time needed to mine a block and to broadcast a message as an *exponential distribution*. As done in [1, 5], we use Continuous Time Markov Chains (CTMCs) for providing a probabilistic model of our processes, where each process action is associated with a *rate* representing its duration.

In [2], we apply this methodology to analyze the probabilistic behaviour of the Bitcoin protocol by using the probabilistic model checker PRISM [7]. In our model, we consider every implementation detail of Bitcoin that influences the probability of reaching a state of fork, and ignore every detail that does not impact on our analysis, e.g., the verification of digital signature and the actual computation of the hashes.

The formal definition of the abstract model of Bitcoin is described using the language provided by PRISM, which is a process calculus with a probabilistic and stochastic semantics. Actually, in order to deal with the complex data types used by Bitcoin, e.g., blocks and ledger, and with the operations upon them, we extend this language, and introduce the richer variant, called PRISM+, that provides these concepts natively, thus simplifying the definition of the model. We remark that PRISM+ is not strictly tied to Bitcoin but can be used to model other blockchain systems. In this work, we briefly provide an overview of our

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Bitcoin model in PRISM+ and of the results of the different analyses we performed. Finally, we briefly discuss about our current and future works.

2 The Bitcoin Model

We model the Bitcoin protocol as the parallel composition of n Miner processes, n *Hasher* processes and a process called *Network*. *Hasher* processes model miners' attempts to solve the cryptopuzzle, while the *Network* process models the communication infrastructure the miners rely on. To abstract out the solution of the cryptopuzzle and the emission of new blocks, we use rates. It turns out that every process can be modelled as a CTMC because:

- the time spent by a miner m_i to mine a block can be described by an exponential distribution $1 - e^{-\lambda_{m_i} t}$, where the parameter λ_{m_i} depends on the miner hashing power and the difficulty level of the crytopuzzle (see [9]);
- the communication delay across the Bitcoin network can be also approximated by an exponential distribution [4].

The behaviour of processes can be summarized as follow:

- The `Hasher_i` processes represent the PoW algorithm performed by miners: the `Miner_i` who wants to solve the cryptopuzzle synchronizes with the `Hasher_i` which "answers" telling the him if he succeeded or not.
- A `Miner_i` may win the cryptopuzzle and create a new block or lose and wait for new blocks. In our setting, mining a block amounts to synchronizing with the Hasher. This synchronization has a rate `hR_i`, with $0 < \texttt{hR\_i} < 1$, indicating the computational power of the node. When a block is mined, it is added to the local ledger of the miner and it is forwarded to all the other, by synchronizing with the Network process. When the miner loses the cryptopuzzle, the miner may receive a block from the network or may try to add blocks stored in his local set.
- The `Network` process contains a set of blocks for each `Miner_i` that represents the messages to be delivered to the miner.

3 Probabilistic Analyses

We performed four probabilistic analyses covering different aspects. The first analysis assessed the coherence of our model with the real protocol, in a simple scenario, i.e. we assumed that miners all communicate with each other, so the Network process implements the broadcast of messages. To achieve that, we verified that the probability of mining a new block within a given amount of time, and that of reaching a fork correspond to those of Bitcoin, and coincide with the values available in the literature. Those results are consistent with the ones of Laneve and Veschetti [8], which provide a formal proof of the probability that Bitcoin ledgers may devolve into inconsistent states, also in presence of a double spending attack.

The second analysis concerned the trade-off between security and the difficulty of the cryptopuzzle. Our results show that a slight decrease of the difficulty level of the cryptopuzzle leads to a significant increase of the speed of mining at the cost of an almost irrelevant increase of the probability of a fork.

We also modeled and analyzed a network with churning nodes, i.e., nodes that can leave and rejoin the network, which provide a more realistic account of the behaviour of this complex platform. Our results show that churn nodes have a strong impact on the way the mining intervals vary with time: indeed, when a node leaves the network frequently, there is an immediate consequence on how the hashing power is distributed in the network.

Finally, we modeled and analyzed the Bitcoin protocol taking into account different kinds of network topologies (daisy topology, round topology, tree topology and fully connected topology). The driving question of our analyses was checking whether the considered alternative topologies have a resistance to forks equal to or greater than the original one of Bitcoin. The results of our simulations clearly pointed out that the less the nodes are connected, the higher is the probability of reaching a state of fork. Moreover, our results made evident that the dynamism of nodes affects how blocks propagate across the network. Indeed, when a node disconnects, the network connectivity is reduced and this leads to an increase of the mean number of hops required for a block to reach all miners. So this delay in propagation increases the probability of forks.

4 Future Works

So far we assumed that miners are honest and work to extend the blockchain. As a future work, we would like to weaken this assumption and analyze also Bitcoin security issues, in particular, considering scenarios with malicious miners.

Moreover, the methodology we adopted to model Bitcoin and the language PRISM+ are generic and can be used for other consensus algorithms. We are currently working on the analysis of the Casper protocol, the future PoS protocol of Ethereum [3]. Casper introduces the notions of *justification* and *finalization* of blocks, and the main chain is the one whose justified and finalized blocks received the most of the votes by validators, i.e., the nodes taking part to the consensus. The aim of our work is (*i*) to obtain a faithful representation of what the system will be; (*ii*) to study the probability of creating a new block within a certain amount of time, the probability of forks, and of justification and finalization of blocks; and (*iii*) to analyse its resilience to attacks. In particular, we are focusing on the Eclipse attack and the 51% attack. The first can be modeled by changing the Network process, whereas the second one can be analyzed by introducing malicious Miner processes.

References

1 Bruno Biais, Christophe Bisiere, Matthieu Bouvard, and Catherine Casamatta. The blockchain folk theorem. *The Review of Financial Studies*, 32(5):1662–1715, 2019.

2 Stefano Bistarelli, Rocco De Nicola, Letterio Galletta, Cosimo Laneve, Ivan Mercanti, and Adele Veschetti. Stochastic modelling and analysis of the bitcoin protocol. Under review, 2021.

3 Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, 1:22–23, 2013.

4 Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *P2P*, pages 1–10. IEEE, 2013.

5 Ittay Eyal and Emin Gün Sirer. Majority is not enough: bitcoin mining is vulnerable. *Commun. ACM*, 61(7):95–102, 2018. `doi:10.1145/3212998`.

6 Vincent Gramoli. From blockchain consensus back to byzantine consensus. *Future Gener. Comput. Syst.*, 107:760–769, 2020. `doi:10.1016/j.future.2017.09.023`.

7 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

8 Cosimo Laneve and Adele Veschetti. A formal analysis of the bitcoin protocol. In *Gabbrielli's Festschrift*, volume 86 of *OASIcs*, pages 2:1–2:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

9 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008.

10 Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *EUROCRYPT (2)*, volume 10211 of *LNCS*, pages 643–673, 2017.