Multi: a Formal Playground for Multi-Smart Contract Interaction

Martín Ceresa
IMDEA Software Institute, Madrid
César Sánchez
IMDEA Software Institute, Madrid

Abstract

Blockchains are maintained by a network of participants, miner nodes, that run algorithms designed to maintain collectively a distributed machine tolerant to Byzantine attacks. From the point of view of users, blockchains provide the illusion of centralized computers that perform trustable verifiable computations, where all computations are deterministic and the results cannot be manipulated or undone.

Every blockchain is equipped with a crypto-currency. Programs running on blockchains are called smart-contracts and are written in a special-purpose programming language with deterministic semantics[1]. Each transaction begins with an invocation from an external user to a smart contract. Smart contracts have local storage and can call other contracts, and more importantly, they store, send and receive cryptocurrency.

Once installed in a blockchain, the code of the smart-contract cannot be modified. Therefore, it is very important to guarantee that contracts are correct before deployment. However, the resulting ecosystem makes it very difficult to reason about program correctness, since smart-contracts can be executed by malicious users or malicious smart-contracts can be designed to exploit other contracts that call them. Many attacks and bugs are caused by unexpected interactions between multiple contracts, the attacked contract and unknown code that performs the exploit.

Moreover, there is a very aggressive competition between different blockchains to expand their user base. Ideas are implemented fast and blockchains compete to offer and adopt new features quickly.

In this paper, we propose a *formal playground* that allows reasoning about multi-contract interactions and is extensible to incorporate new features, study their behaviour and ultimately prove properties before features are incorporated into the real blockchain. We implemented a model of computation that models the execution platform, abstracts the internal code of each individual contract and focuses on contract interactions. Even though our Coq implementation is still a work in progress, we show how many features, existing or proposed, can be used to reason about multi-contract interactions.

2012 ACM Subject Classification Theory of computation → Program reasoning; Theory of computation → Program constructs; Theory of computation → Abstract machines

Keywords and phrases blockchain, formal methods, theorem prover, smart-contracts

Digital Object Identifier 10.4230/OASIcs...

Funding This work was funded in part by the Madrid Regional Government under project “S2018/TCS-4339 (BLOQUES-CM)‘’ and by a research grant from Nomadic Labs and the Tezos Foundation.

1 Introduction

Smart-contract manipulate cryptocurrency, which has a corresponding value as money. Since smart-contracts cannot be modified once installed and their computations cannot be undone

---

[1] Although the behaviour of smart-contracts may depend on values to be known at runtime, i.e. block number; hashes; etc, their behaviour is deterministic.

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

("the contract is the law"), all interactions with the contract are considered valid. Therefore, there is an incentive for malicious users to take advantage from unexpected behaviors and interactions. Also, errors in contracts can result in losses and cryptocurrency being locked indefinitely, even when used but by well-intentioned users. We focus in this paper on the computational notion of correctness, and not on the real legal implications resulting from interactions in the blockchain or the use of smart-contracts to enforce legally binding contracts [8].

One important reason why it is very difficult to reason about smart contracts is that they live in an *open universe*. Even though the code of a given smart-contract C cannot be modified once installed, other contracts that call and are called from C can be programmed and deployed after malicious users study C. Therefore, programmers and auditors of contract C did not have to analyze all possible code that can invoke or be invoked from C.

At the same time, users demand blockchains to implement new features. Since there is a big competition between blockchains, this puts pressure on architects of blockchains on the time to market of new features. And each new feature potentially increases the attack surface of smart contracts.

There are different kinds of errors found in smart-contracts.

- *Logical problems* are related to errors in the logic of the smart-contract. Usually, attackers detect a corner case that can be exploited to generate an unwanted behaviour.
- *Low-level execution* problems that arise from a misunderstanding on details of the low-level execution platform. Examples include underflow, overflow or exploiting unexpected behavior after the stack limit is reached.
- Programmer can also employ *bad idioms* that they are familiar with from other areas of software applications, but which may be dangerous in interactive platforms like blockchains, where all data (including the state of the contracts) is public and verifiable.

Most bugs are related to multi-contract interactions. For example, the infamous DAO attack where malicious code *legally* exploited the machinery of the Ethereum blockchain creating unexpected re-entrant calls from remote contracts led to the loss of $60 million [14].

In this article, we present a formalization in Coq of a general blockchain model of computation that allows us to study new multi-contract interactions as well as new features. We aim to develop a formal and rigorous way to analyze the possible interactions between contracts and also to study how new features affect contracts before they are implemented and deployed. Our Coq library allows simulating the execution of smart-contracts, abstracting away the internal code of the contract. Our abstraction is based on the Tezos blockchain, but it is general enough to cover other blockchains like Ethereum. We model smart-contract (almost) as pure functions from the current storage and state of the blockchain into (possibly) a list of operations to do next plus changes in the storage.

2 Motivation

After successful attacks like DAO [14] there is a growing interest in formal methods for smart-contracts. First, there is an interest in verifying that a contract satisfies a specification so certain properties can be guaranteed, e.g. the owner will be able to fetch all funds or that a bidder will either gain the bidding or recover the funds. Second, it is also important to formally study different mechanisms and features proposed for a given blockchain before they are offered so new attacks can be prevented. Some of these mechanisms are proposed to allow users to use more effective defensive programming idioms.

For example, by analyzing the DAO attack [9] proposed a property called *effectively* *callback free* which restricts the interactions within smart-contracts disabling these attacks. Later on, the Tezos blockchain [1] implements such property by construction: smart-contracts are functions that either fail or returning a list of operations to be executed plus a new storage. Therefore, the storage is updated before the operations are executed, which prevents attacks like the DAO using this programming style.

In order to prevent these attacks, the Tezos blockchain followed a conservative scheduling strategy. In Tezos, as is the general case, every transaction begins with a request by an external user indicating the smart-contract to invoke, method and arguments, and balance of the initial operation. Assume user *Alice* starts a transaction invoking method f of smart-contract C, and that, after executing $C.f$ we have a list of operations $[o_0, \ldots, o_n]$. To compute the result of the transaction, the blockchain will execute each operation o_i in order, until the gas is exhausted or the list of pending operations is empty. The order in which the operations are executed affects the outcome of the transaction. Two conventional strategies are: (1) to insert the new list of operations at the beginning of the list of pending operations (DFS) (2) to insert the new list of operations at the end (BFS). The first one, DFS, allows us to implement a call-and-return flow of computation and it is the more conventional in most blockchains. The second one, BFS, prevents call injection attacks by construction as one can guarantee that two operations are executed back-to-back and was used until version 8 of Tezos (Protocol Edo) [5]. In our example, assuming that executing o_1 generates bs operations, the result of the previous execution would be $[o_2, \ldots, o_n] \cdot bs$. While in DFS, the result would be $bs \cdot [o_2, \ldots, o_n]$, and thus, the instructions in bs will be executed before $o_2, \ldots, o_n$. However, BFS suffers from other classes of problems.

Assume a bank contract that holds money for a customer and the bank contract is willing to send money as long as the balance stays above threshold `threshold`. In a solidity like language, the contract could be as follows:

```
contract Bank {
  uint threshold;
  address owner;
  constructor(uint _threshold, address _owner) public {
    threshold = _threshold;
    owner = _owner;
  }
  function deposit() payable public{
    return([]);
  }
  function withdraw(uint ret) public {
    if (sender = owner) then
        if (balance - ret > threshold) then
            return ([transfer(owner.Receive, ret)])
        else
            fail("breaking invariant")
    else
        fail("not owner")
  }
}
```

Normal usage of a such a bank contract can be:

```
contract GoodClient{
```

```
    address bank;
    // ...
    function askMoney(uint m){  // Requests m from the vault
        return([bank.withdraw(m)]);
    }
}
```

On the other hand, the following is a simple attack exploiting the bank contract:

```
contract Bad{
   address bank;
   //...
   function rob(uint n, uint m){ // BFS attack to the vault!
       return(ntimes n [bank.withdraw(m)])
   }
}
```

The new method called `rob` generates a list of invocations to the vault. Assume the vault contract has a threshold of 9 and that is in a state in which it stores 15 units of cryptocurrency. A simple examination suggests that the vault will send money back to its owner whenever its balance is greater than 9, effectively allowing only one withdrawal. However, consider the following execution starting from `[rob(3,5))]`. After executing the operations, we would have the following pending queue:

```
[(Bad, vault.withdraw(5)), (Bad, vault.withdraw(5)), (Bad, vault.withdraw(5))]
```

Then the BFS sequence of executions leads to the following sequence of pending operations:

```
[(Bad, vault.withdraw(5)),  (Bad, vault.withdraw(5)),  (Bad, vault.withdraw(5))]  ⇝
[(Bad, vault.withdraw(5)),  (Bad, vault.withdraw(5)),  (Vault, Bad.Receive())]    ⇝
[(Bad, vault.withdraw(5)),  (Vault, Bad.Receive()),    (Vault, Bad.Receive())]    ⇝
[(Vault, Bad.Receive()),    (Vault, Bad.Receive()),    (Vault, Bad.Receive())]    ⇝
[(Vault, Bad.Receive()),    (Vault, Bad.Receive())]                               ⇝
[(Vault, Bad.Receive()) ]                                                         ⇝
[]
```

First, the operation sending the money back to contract `Bad` is added at the end, as dictated by BFS. Second, according to the semantics of feature "transfer" in the Tezos blockchain, funds are subtracted from the sending contract `Vault` after the transfer is executed. Therefore, the second `withdraw` request does not see the effect of attending the first one. The combined effect is that all three requests are attended resulting in a total extraction of 15 units leaving 0 in contract `Vault` *without noticing the attack*. The attack is based on the separation between the creation of a transfer and its execution. The lesson is that even though a BFS order prevents injection attacks, it allows attacks based on the delayed effect of emitted operation. The contract `Vault` can be easily fixed by encoding in a variable in the storage the balance that has been compromised with a future transfer. If necessary, `withdraw` can create two operations (1) the transfer, and (2) an invocation to a new private method in `Vault` whose purpose is to note that the compromised balance created by a withdraw has been effectively arrived.

Another lesson is that relying on the balance of contracts is considered a bad smart-contract programming practice. Assume now that programmers would like the architects of the blockchain to implement not only `balance` but also `pending_balance`, which accounts for transfers sent but not executed. Moreover, assume also that the blockchain also implements the feature of *views*, an apparently innocent feature that simply returns information about the storage of a contract without any effect. We illustrate that these two features combined can lead to undesirable effects. For example, if we would like to maintain the invariant that at every moment the amount of combined funds between a collection of contracts is constant, the combination of `pending_balance` and views can break such an invariant.

For example, consider three smart-contract A, B, C, and the following pending queue of operations:

$$[\underbrace{A_1, \dots, A_o}_{A}, \underbrace{C_1, \dots, C_m}_{C}, \underbrace{B_1, \dots, B_n}_{B}]$$

where A sends money to B—in operations that are going to be executed after C but that update A pending balance. This leaves C in a difficult position. If C observes (using views) the balances of A and B there is going to be a mismatch with their real balances, because C will see the pending compromised balance but not the pending receives, which may induce bad behaviour in C. If C depends on $A.balance + B.balance$, for example, to buy some NFT it may incorrectly fail to take the right decision. A possible solution is to introduce yet another feature that captures pending receives.

In our line of work, we aim to build a *formal playground* where different features and mechanisms can be encoded and reasoned about easily and formally, while also simulating the execution of multiple contracts.

3 Previous Work

In our work, we follow a static verification approach where contracts and features are analyzed before deployment. The idea is to encode how blockchains are implemented and study the behavior of contracts and features by formally proving properties. Several approaches have been suggested for testing, model checking and functional and temporal verification of smart-contracts. We review the most relevant.

Mi-Cho-Coq. Mi-Cho-Coq is the first verification tool implemented in Coq for the Tezos blockchain ecosystem [4]. The main difference between Mi-Cho-Coq and our effort (Multi) is that Mi-Cho-Coq focuses on the analysis of the code of a single contract (or collection of calling contracts for which the code is available). We say that Mi-Cho-Coq implements small-step semantics to prove *functional properties*, which requires to have a concrete specification of a smart-contract and either its code or a higher level specification.

The main difference with Mi-Cho-Coq is that our goal is to prove properties *emerging* from interactions between smart-contracts. Our tool is a complementary effort to lower-level verification tools as Mi-Cho-Coq.

Concert. Concert [3] is another framework written in Coq to prove formal properties of smart-contracts, and in this case, they accept multi-contract interaction [11]. The fundamental idea of Concert is to model of smart-contracts as agents and computation as interaction (message passing) between these agents. They also implement specific mechanisms, for example, they implement delegation primitives in the Tezos blockchain. Moreover, Concert has an extraction mechanism to extract high-level smart-contracts written in Ligo [10].

Our main difference is that we implement a very flexible framework with the idea of encoding new potential blockchain features and prove properties of how different features interact with each other. Including BFS and DFS scheduling in the Tezos blockchain, but there may be other scheduling strategies.

Concert implements blockchains in a generic way using specific features of Coq (class system) and meta-programming features to easily embed blockchain smart-contract languages. Concert also builds proofs by inspecting the trace representing the evolution of the blockchain observed by a *small step relation*.

Implementing new blockchain features relating to how smart-contracts are executed is an important feature in our framework, and moreover, we want to be able to reason and prove properties about such features. For example, what would happen if smart-contracts can inspect *runtime* information as the stack call (what the next operations or pending operations are). Another difference is that (so far) we observe the state of the blockchain comparing just the state of the blockchain before a transaction begins and after a transaction ends. We are also able to inspect intermediate transition steps, but we are not exploiting that feature yet.

Scilla. Scilla is a smart-contract language embedded in Coq [16] that allows some temporal reasoning (see [17]). Scilla is an embedded domain-specific language in Coq which also abstracts smart contracts as functions returning a list of operations. The main difference between Multi and Scilla is that we do not present a language to write smart-contract but use Coq functions directly. We share the point where the effects of executing smart-contracts are simple a list of operations that are propagated by the executer. As Concert, we have a clean separation between the language of smart-contracts and the machinery required to execute smart-contracts. However, in our case, we decoupled the scheduler from the execution of single instructions, and thus, we can implement different scheduling strategies independently of the set of operations.

VerX. VerX is an automatic software verification tool that checks custom functional properties of smart-contract entrypoints. VerX works on a similar level to Mi-Cho-Coq, in the sense that they prove functional properties of smart-contracts, but it is built to be completely automatic and also to handle some multi-contract interactions. The interaction between smart-contracts comes from performing analysis on the possible onchain behaviours of a set of smart-contracts. VerX restricts the analysis to a set of smart-contracts, S, that have a condition called *effectively external callback free contracts*, which states that any behaviour generated by an interaction between smart-contracts in set S that has an external call is equivalent to a one without external calls [13]. This follows the lines of [9]. Because of that restriction, they can reason about smart-contract, proving PastLTL specifications, but it also restricts them to work in a **close universe**.

SmartPulse. SmartPulse [18] is another automatic verification tool for smart-contracts. The main goal is to verify temporal properties including some simple liveness properties. This tool is similar to *VerX* but it is focused on proving liveness properties of a single contract in a closed universe. They do not support multi-contract interaction.

4 Model of Computations

Blockchain Model. We ignore the internals of the infrastructure of blockchain implementations (like cryptographic primitives, consensus algorithms or mempools) and focus exclusively on the model of computation that blockchains offer to external users. The blockchain is then abstracted by a partial map from addresses to smart-contracts. Smart-contracts are programs with some structure:

- Storage: a segment of memory that can only be modified by the smart-contract.

- Balance: an attribute of contracts that indicates the amount of cryptocurrency stored in the contract.
- The program code: a well-formed program that represents the implementation of the smart-contract.

The state of a smart-contract is a proper value of its storage plus the balance its stores. The model of computation consists of the sequential execution of transactions, each of which is started by the invocation of an operation. In the current version, we ignore how gas or fees are paid or how new currency is created during the evolution of the blockchain to pay the bakers. Smart-contracts can be executed upon request from an external user that initiates a transaction or by the invocation from a running contract. Upon invocation, the blockchain evaluates the result of executing the smart-contracts program following a given semantics producing effects on the blockchain (further invocations) and changes on the smart-contracts' storage or they may fail.

Open Universe. We introduce now the concept of *universe of computation*. Once a smart-contract has been installed on a blockchain, every other entity in the blockchain can interact with it. The smart-contract itself can invoke or be invoked by older or newer contracts. The case of smart-contracts invoking just older and well known contracts can be useful sometimes but in general smart-contracts may not know a priori who they are going to interact with. This differs from conventional software where components are built from well-known trustable components and the surface of interaction with potentially malicious usage is small and well defined. The classical way of programming exposes the internals of complex software and leaves open attack vectors. For example, to guarantee certain behaviour high-level smart-contracts invoke low-level smart-contracts following a protocol to logically guarantee a result. However, malicious software **may not** follow such protocols possibly breaking or leaving low-level smart-contracts in an incorrect state. This open universe model of computation forces smart-contracts to implement defensive mechanisms to prevent undesired executions.

Most verification techniques and frameworks mentioned previously do not take into care such assumption. They operate under the idea that smart-contracts behave the way they are supposed to, in the sense, that either they avoid external call invocations by removing interactions or by assuming they are interacting with good smart-contracts. However, this is not the case, the blockchain is an aggressive environment, a so called *dark forest* [15]. In this paper, we study this problem attempting to formalize properties of smart-contracts operating under a more realistic (and pessimistic) view of the world and also to develop new mechanisms or features to explicitly guarantee that we are working under a safe environment. Such mechanisms could be implemented inside smart-contracts, but not every mechanism can be implemented using current blockchain technologies, like transaction monitors [6, 7].

5 Formalization

In this section, we describe the building blocks of our Coq library implementation that allows us to reason about different blockchain execution mechanisms. Our goal is to study how smart-contracts interact with other smart-contracts, and thus, we abstract away the internal execution of the instructions of the smart-contract. Moreover, we need a framework flexible enough to implement new features (i.e. different execution models, scheduling strategies, etc) and, additionally, a formal system to prove and verify properties of interactions between smart-contracts implementing and using such features. In short, we implemented a *formal playground* simulating the model of computation of blockchains.

We abstracted blockchains following the model described in Section 4 in the proof-assistant Coq. We interpret smart-contracts as pure functions in the host language Coq and every additional feature is implemented on top of pure functions.

Smart-contracts are implemented as a structure with three fields (Listing 1): a storage, a balance, and a pure function implementing the smart-contracts code.

```
Structure SmartContract (Ctx Param Storage Error Result : Type) : Type :=
  mkSmartContract {
      (* Storage *) _Sst : Storage ;
      (* Balance *) _Sbalance : ℕ ;
      (* Computation that result in an element of type Result *)
      _Sbody : Ctx → Param → Storage → Error + (Result * Storage)
    }.
```

Listing 1 Smart contract Definition

Note that structure `SmartContract` is highly parametric:

- Parameter `Ctx` represents what smart-contracts can observe about the blockchain and the execution model as: current block level, the total balance of the transaction, who the sender and source are, etc.
- Parameter `Param` represents the parameters the body of the smart-contract expects to receive; using `Param` we model the different entrypoints of a contract.
- Parameter `Storage` represents the storage of the smart-contract.
- Parameter `Error` represents the type of errors that can result from the execution of the smart-contract.
- Parameter `Result` represents the resulting type of smart contracts, which in the Tezos model is a list of further operations.

The type `SmartContract` represents the most basic structure of a smart-contract. It is simply a structure with some storage, balance and a body.

The Smart-contracts body is modeled as a pure functions from the current state of the blockchain and its storage to a sequence of operations. In this way, we abstract away concrete blockchain programming languages or implementations. Even though our formalization is based on the semantics of method invocations in the Tezos blockchain, different programming language can be modeled in this paradigm using standard compiler techniques (essentially dividing a complex function with effects into its basic blocks that are pure functions as modeled here).

5.1 Execution

The execution of a smart-contracts, aside from changes in the storage, also produces a sequence of operations to be executed. Therefore, we have to take care of two things: how to execute these operations, and how to order the execution. We split the execution model into two main pieces: a scheduler and an executor.

Scheduler. The scheduler is in charge of the order of execution, adding new operations the pending queue (either at the beginning or the end, etc). The scheduler is also in charge of creating new contexts. Finally, it is in charge of building the graph/tree of transactions, every information that descendants of an operation may share is kept and organized by the scheduler.

Executer. The executer is in charge of executing an operation in a given context, and it is the same independently of the evaluation order. The most basic operation of an executor is smart-contract invocation, which requires that the executor collects and builds the environment in which such invocation should be executed. The context is the blockchain state from the point of view of the contract execution. Another operation is smart-contract creation, which in this case it is going to generate a modification to the blockchain, and communicate it to the scheduler.

Operations. We assume the blockchain has a simple set of operations. We start from a minimal set of operations that is simple enough to enable smart-contracts interaction, and later add new operations as needed afterward.

We begin our implementation with two operations: `Transfer` and `Create_Contract`.

- Operation `Transfer` performs an invocation to a given address while also sending money.
- Operation `Create_Contract` installs a new smart-contract at an indicated address with an initial amount of balance and storage.

```
Inductive EnvOps : Type :=
| Transfer : forall (T : Mich_Type),
    (* Parameter *) (Type_Interpret T) →
    (* Amount to transfer *) Mutez →
    (* Contract address to invoke *) (Type_Interpret (ContractT T)) →
    EnvOps
| Create_Contract : forall (PTy StTy : Mich_Type),
    (* Pre-computed Address *) Address →
    (* Initial amount *) Mutez →
    (* Initial Storage *) (Type_Interpret StTy) →
    (* Body *) MichBodyTy PTy StTy (list EnvOps) →
    EnvOps.
```

Where `Mich_Type` is an enumeration type of the different data structures supported by the blockchain, i.e. natural numbers, strings, etc. In our case, since we are working close to the implementation of the Tezos blockchain, we implement most of its data structures, and we represent them as an inductive type `Mich_Type`. Using the previous operations, we can define smart-contracts simply as the following structure:

```
Structure MichContract : Type := mkMich {
    (* Contract parameter type *) _Param : Mich_Type ;
    (* Storage type *) _Storage : Mich_Type;
    (* Contract body*)
    _Soul : SmartContract
                (TzCtxt _Param)
                (Type_Interpret _Param)
                (Type_Interpret _Storage)
                OError
                WritingContext;
    }.
```

Essentially, we capture smart-contracts as their body plus information about their types. Hiding away the type information forces us to implement a lot of type matching clauses when it comes to the execution of smart-contracts. However, it enables us to represent the state of the blockchain simply as a (partial) map of addresses to smart-contract.

```
Definition TezosEnvironment := string → option MichContract.
```

Given an operation, the executer is in charge of building the required information to execute. In the case of an invocation to an address `addr`, the executer looks up the address `addr` into the current environment to see if there is a smart-contract matching the expected type at that address, and in that case, executes its body to obtain either a new storage and further operations or a fail. In the case of a smart-contract creation operation, the executer is in charge of checking that the address is actually free and updating the environment adding such smart-contract. Finally, the executer is also in charge of checking that smart-contracts have enough balance to perform transactions and update the current environment with the new balances.

We can characterize our executer as follows:

```
Definition ExecuterTy : Type :=
    (* Input context information *) (ctx : ExecutionContext)
→ (* Operation to execute*) (o : EnvOps)
→ (* Current state *) (env : BCEnvironment)
→ MFail (* possibly returning: *)
        (option(
          (* Address emitting new operations, next sender *) Address *
          (* Effects generated (new operations) *) WritingContext)
          * (* Updates to the environment *)
          (list (Address * MichContract))).
```

Different executers exercising type `ExecuterTy` can interpret operations in different ways. Executers receive two arguments, `ctx` and `env`, representing the execution context and the environment of the blockchain, respectively, and in return, provide the modifications to the environment and possibly a list of new operations. Note that `ExecuterTy` leaves some proofs obligations if we want to simulate current blockchains, i.e. we need to show that `ExecuterTy` does not modifies or upgrades exiting smart-contracts' code (see Section 5.2).

Schedulers are in charge of gluing together the effects generated by the execution of operations in the current blockchain. We model them in Coq as a type listed in Listing 2 where `SchedulingStrategy` implements the execution order to follow. In other words, schedulers keep track of the evolution of the state of the blockchain while managing the pending queue of operations. Schedulers take the first operation on the pending queue, build the information required by the executor, and pass everything to an executer. When executers return, schedulers take the resulting operations and updates to the current state of the blockchain.

```
Definition Scheduler : Type
  := (* Strategy *) SchedulingStrategy
  → (* External user *) Address
  → (* Executor *) ExecuterTy
  → (* Current environment *) BCEnvironment
  → (* Time *) Timestamp
  → (* Pending Execution list *) list (list EnvOps * ExecutionContext)
  → (MFail BCEnvironment * Timestamp).
```

■ **Listing 2** Schedulers type

The computation of a transaction begins with an external user (outside the blockchain) posting one or more operations to be executed, defined in Listing 3. The initial transaction

```
Structure SignedTrans : Type := mkSignedTrans {
      _author : Address; _trans : list EnvOps
    }.
```

Listing 3 Signed transactions definition.

is given to the scheduler, which also receives a scheduler strategy, an executer, and a context to compute the transaction and its descendants operations. The result is a pair composed of a possible new environment and the next timestamp. We need timestamps to represent the passage of time, and thus, time progresses even in the case that a transaction is reverted. In practice, the scheduler strategy is fixed for a given blockchain.

Since blocks in the blockchain are just sequences of signed transactions, `SignedTrans`, we can generate arbitrary traces with systems like QuickChick [12]. Given a logical program (reflected in a set of smart-contracts), we can codify the possible logical operations in an inductive type in Coq. Therefore, we can generate a sequence of actions translating the logical steps into transactions in the blockchain and verify that the smart-contracts do not reach an invalid state.

5.2 Proof of Correctness

We can define a specification of how a proper blockchain should behave and check that our implementation follows the specification. For example, a basic property is *no-double spending* which states that transfers (remote contract invocations) are paid once, i.e. the sender is not charged twice for the same operation. We can go even further and prove that executing a transfer does exactly what it is supposed to do (Listing 4), i.e. invokes another smart-contract, executes its code, deduce the expected amount from the sender's account, and adds it to the destination's account, or fail (in which case the transfer has no effect).

```
Lemma SimpleTransferCheck :
  forall callerContract calleeContract parameterTy BCCtxt send
    (parameter : Type_Interpret parameterTy)
    (storage storage' : Type_Interpret (_Storage calleeContract))
    (contractContext : TzCtxt parameterTy),
    successWith (ops, (caller', callee'))
                (SimpleTransfer callerContract send calleeContract)
    → _st calleeContract ≡ storage
    → successWith (ops, storage') (exec calleeContract BCCtxt parameter)
    ∧ ((_balance callerContract) - send) ≡ _balance caller'
    ∧ ((_balance calleeContract) + send) ≡ _balance callee'
    ∧  _st callee' ≡ storage'.
```

Listing 4 Transfer is correct.

An alternative approach would be to define a small step inductive relation defining how blockchains should behave and prove that the scheduler follows it step by step. The framework Concert [3] follows that approach.

5.3 Multi-Contract Interaction Proofs

The most important part of our framework is that we can simulate executions of smart-contracts and inspect the effects generated by smart-contract interactions. In other words, we have a big-step semantics of blockchain operations where we can study how smart-contracts using different mechanisms (i.e. BFS/DFS, etc) interact with each other. We can build proofs either by observing the evolution of the transaction execution operation by operation, or analyzing its final state after the transaction terminates. In other words, we have a definition of observational equivalence of smart contracts modulo the particular blockchain employed as evaluator.

This is extremely useful because we can abstract away entire smart-contracts and event simulate the more realistic scenario: a demonic environment. Either we know the code of smart-contract and we can predicate over these code during the proof, or we do not have these code, which requires reasoning with universal quantification over all possible smart-contracts. In other words, to prove that smart-contracts are prepared to operate properly in the open universe of the blockchain requires to reason about the interactions with all possible contracts.

We can model angelic computations by expanding our known universe of smart-contracts simply by implementing smart-contract on our framework and having them installed in the blockchain inside a simulation.

6 Conclusion

In this paper, we present Multi, a formal playground to reason about smart multi-contract interaction and to study features of the blockchain before deployment. Additional features and mechanisms are described in Appendix C and Appendix B where we introduce the idea of *Bundles* of operations: semantic restrictions on the execution of a sequence of operations. Our framework, based on the Tezos blockchain, is very general and allows us to reason about different execution orders, abstracting away each operation on a contract by a pure function whose output is either a failure or the changes in the local storage plus further operations.

Future work includes:

- Examples and study cases: implement and study complex use cases.
- Integrate Multi to the Tezos formal ecosystem and study interactions with Concert and Mi-Cho-Coq.
- Implement additional features, e.g. transaction monitors, views, etc, and study how they interact between each other.
- Design and implement a DSL to easily encode specific smart-contracts easing the translation from existing languages into Coq functions.
- Write more expressive smart contract types following the steps of Concert since Coq functions are more general than the contracts accepted by most blockchains (like Tezos).
- Implement complex features as *tickets*/NFT using some mechanisms (like monads) to better capture the space of functions that represent smart-contracts.

Finally, we aspire to implement a richer specification language using ATL [2] to describe the interaction between smart-contracts and fully verify their specification in Coq. The idea consists in describing programs as interactions between agents (i.e. smart-contracts) where agents cooperatively guarantee certain properties or exercise certain rights. At the semantic level, we would connect the evaluation of smart-contracts in a blockchain with their semantic given by ATL and concurrent games. In other words, with Multi, we can interact between a rich specification language of smart-contracts and their behaviour defined by the execution of blokchains.

References

1 Victor Allombert, Mathias Bourgoin, and Julien Tesson. Introduction to the tezos blockchain, 2019. URL: `https://arxiv.org/abs/1909.08458`, `doi:10.48550/ARXIV.1909.08458`.

2 Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, sep 2002. `doi:10.1145/585265.585270`.

3 Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. Concert: a smart contract certification framework in coq. *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, Jan 2020. URL: `http://dx.doi.org/10.1145/3372885.3373829`, `doi:10.1145/3372885.3373829`.

4 Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. Mi-cho-coq, a framework for certifying tezos smart contracts. *CoRR*, abs/1909.08671, 2019. URL: `http://arxiv.org/abs/1909.08671`, `arXiv:1909.08671`.

5 Tezos Blockchain. Tezos agora: Florence, no ba (psflorena), 2021-12-24. URL: `https://agora.tezos.com/period/46`.

6 Margarita Capretto, Martin Ceresa, and Cesar Sanchez. Transaction monitoring of smart contracts, 2022. URL: `https://arxiv.org/abs/2207.02517`, `doi:10.48550/ARXIV.2207.02517`.

7 Alberto Cuesta Cañada, Fiona Kobayashi, fubuloubu, and Austin Williams. Eip-3156: Flash loans. URL: `https://eips.ethereum.org/EIPS/eip-3156`.

8 Joshua Ellul, Jonathan Galea, Max Ganado, Stephen Mccarthy, and Gordon J. Pace. Regulating blockchain, dlt and smart contracts: a technology regulator's perspective. *ERA Forum*, 21(2):209–220, Oct 2020. `doi:10.1007/s12027-020-00617-7`.

9 Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017. `doi:10.1145/3158136`.

10 Ligo. Ligo: A friendly smart contract language for tezos, 2022. URL: `https://ligolang.org/`.

11 Jakob Botsch Nielsen and Bas Spitters. Smart contract interactions in coq. In *FM Workshops (1)*, volume 12232 of *Lecture Notes in Computer Science*, pages 380–391. Springer, 2019.

12 Zoe Paraskevopoulou, Cătălin HriŢcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving*, pages 325–343, Cham, 2015. Springer International Publishing.

13 Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1661–1677, 2020. `doi:10.1109/SP40000.2020.00024`.

14 Daian Phil. Analysis of the dao exploit, 2016. URL: `https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`.

15 Dan Robinson and Georgios Konstantopoulos. Ethereum is a dark forest, 2020. URL: `https://www.paradigm.xyz/2020/08/ethereum-is-a-dark-forest`.

16 Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level language, 2018. URL: `https://arxiv.org/abs/1801.00687`, `doi:10.48550/ARXIV.1801.00687`.

17 Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Temporal properties of smart contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 323–338, Cham, 2018. Springer International Publishing.

18 Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. Smartpulse: Automated checking of temporal properties in smart contracts. In *42nd IEEE Symposium on Security and Privacy*. IEEE, May 2021. URL: `https://www.microsoft.com/en-us/research/publication/smartpulse-automated-checking-of-temporal-properties-in-smart-contracts/`.

A Angelic/Demonic

Given the open universe nature of blockchains, smart-contracts are forced to identify who are they interacting with. Programmers when they are designing complex software do not think that they are in a dangerous and aggressive environment, as it is now, and simply think that smart-contracts will interact with good pieces of software doing what they are supposed to do. However, as we saw before, this may not be true.

In this section, we present a new characterization when it comes to classifying the interaction between multiple smart-contracts. We call this characterization *Angelic/Demonic* where we mark smart-contracts as *angelic* when they do what they are supposed to do, or as *demonic* when we cannot assume anything about their behaviour, and thus, we cannot predict nor predicate about their behaviour. Note that this is not a property enforced by blockchains, but it is more of a mindset at the moment of designing complex software that is going to run on the blockchain.

There are essentially two basic models to reason about multi-contract interaction:

Closed World Assumption: every smart-contract knows and trusts the smart-contracts that it is invoking (directly and transitively). In particular, every smart-contract C only invokes contracts that are older than C and whose properties are known.

Open World Assumption: every contract C runs in an adversarial environment and smart-contracts should protect against possible *evil* smart-contracts.

A closed world assumption is feasible on many occasions because of the public and *immutable*[2] character of the blockchain. Since everything is public and smart-contracts do not change, as smart-contract developers, we can observe the state and code of smart-contracts that we are going to interact with and decide if they are *angelic*, i.e. if they do what they are supposed to do.

Note that "the angelic state" is fragile and it may change. For example, assume we invoke a smart-contract B that in turn invokes another smart-contract whose address $addr$ is stored in B's storage. As we are about to submit our smart-contracts to the blockchain, we can explore and decide that B and the current $addr$ are angelic. However, eventually, B may change it to another smart-contract $addr'$ that may also be angelic to B, or B is protected towards possible attacks from $addr'$, but it may open an attack on our smart-contract.

The second option, an open world assumption, is a more real situation and sometimes the only possible case for certain smart-contracts. One of the most prominent cases is exchange houses: let Dex be a smart-contract that is always willing to exchange token A for token B for a certain fee in behave of a set of investors. In this case, the smart-contract Dex is doomed to interact with unknown addresses.

Another example is that we can implement a call-and-return model using *continuation passing style* between smart-contracts in BFS blockchains. However, implementing such interactions between smart-contracts requires to assume that *every smart-contracts is going to behave accordingly*, and thus, we are under an angelic assumption. Therefore, we need a framework that can handle angelic and demonic assumptions.

---

[2] Although it is possible to implement mutable and upgradable smart-contracts, this is not the general case, and even if the nature of the smart-contract was to mutate this would be known by the invoker.

B Bundles of Operations

In this section, we introduce the concept of *bundles of operations* high level restrictions on how we want a sequence of operations to be executed. For example, we can abstract away what is important about a scheduler following a BFS strategy: atomicity of a sequence of operations. In other words, the operations generated by a smart-contract are going to be executed one after another without other smart-contracts injecting operations between them.

A *bundle* is a semantic condition (or restriction) on the execution of a sequence of operations. Instead of forcing *the whole blockchain* to use a particular execution order, we theorize on having a domain-specific language (DSL) describing how we would want to execute a set of operations. In other words, we would like to predicate on how operations are to be executed explicitly, either by assuming a BFS/DFS or other mechanisms.

B.1 Atomic Sequence

Given a sequence of operations $\langle s_0, s_1, \ldots, s_n \rangle$, we want them to be executed atomically without interleaving operations independently of the execution order followed by the scheduler. BFS schedulers respect such bundle by definition, while DFS schedulers should check that the effects generated by each s_i with $i \leq n$ does not affect the rest of the smart-contracts.

B.2 Contexts

The call and return pattern enables us to reason about units of functionality, in the sense, that when we invoke a method in a smart-contract is because we expect a result independently of how many other functions that method is invoking. When we program smart-contracts under the demonic assumption, where giving control to other (possibly unknown) smart-contract may result in an attack, we want to encapsulate their behaviour while still interacting with them to obtain some functionality.

Independently of the execution order, we can devise an encapsulation mechanism enabling us to reason about the functionality of external invocations in a *context*. The general idea is to encapsulate the execution of smart-contracts and all of its descendant operations in a *context*. Instead of having a pending queue of operations, we would have a sequence of pending queues, each one representing an encapsulated context. Operationally, each context is completely executed before passing to the next. Contexts give us the ability to invoke functions and execute them as if they were the only procedures being executed in the machine, i.e. in a completely isolated context.

▶ **Example 1.** Let A and B be two smart-contracts such that the result of executing A is two operations $[A_1, A_2]$, while the result of executing B is just $[B_1]$. Moreover, operations A_1, A_2 do not generate new operations.

Assuming we have a pending queue formed by a context invocation to A followed by a normal invocation to B, we will have the following execution sequence:

$$[[A], B] \rightsquigarrow [[A_1, A_2], B] \rightsquigarrow [[A_2], B] \rightsquigarrow [[], B] \equiv [B] \rightsquigarrow \ldots$$

Implementing contexts is easy and very useful to encapsulate functionality. However, this brings some questions: how are contexts created? who creates them? From the point of view of defense programming, we have two possible answers:

Caller contextual call: upon invoking a remote procedure, the caller can specify the execution to be encapsulated in a context. This mechanism protects the callee since the new procedure cannot inject operations interleaving the ones already on the pending queue (as a DFS blockchain would do).

Callee contextual call: when invoked, the callee internally decides if its functions are to be executed in a context. This mechanism enables the function being called to assume that the pending execution queue is empty and nothing is going to modify it aside from itself or the invoked smart-contracts.

C Restricting Smart-Contracts Interaction

We implemented two kinds of restrictions: one where the blockchain enters into a mode where the smart-contract interactions are not allowed, and another where we can reduce the set of addresses that can be invoked.

End of Interactions. The executor only accepts transactions from and to the same smart-contract.

Address Universe. We can dynamically restrict the universe of addresses that smart-contracts (and their descendants) can invoke, either by restricting the known universe of addresses or by specifying addresses that cannot be invoked. In other words, we would have two sets of addresses:

Allow addresses: the set of addresses that can be invoked during execution. Invoking an address outside this set will force the transaction to fail.

Block addresses: the set of addresses that cannot be invoked during execution. Invoking one of these addresses will force the transaction to fail.

Both mechanisms suggest the addition of a shared state between a smart-contract and its descendants during the execution of smart-contracts. If we see transaction executions as trees, we can add restrictions to such tree. Moreover, we can analyze *transaction trees* to restrict or predict the behaviour of smart-contracts.