nomadic labs

Formal verification of smart contracts

Table of contents

Introduction

Introduction

The Tezos blockchain was designed to ensure the required security and compliance of the code, particularly for high-value use cases such as the issue and management of Security Tokens[1] or stablecoins,[2] or even for the blocking of funds in a context such as liquidity pools.[3]

Michelson, the native smart contract language of the Tezos blockchain, facilitates **formal verification** by providing guarantees; this is a methodology currently used in critical environments such as the aerospace, nuclear and semi-conductor industries. This methodology consists in defining the specifications of a computer code and in proving that its execution indeed does what it was designed to do.

Any contract deployed on a public blockchain is almost impossible to rewrite or remove from the history. Consequently, for certain applications, unit tests are insufficient for ensuring that they have the desired operation. It is therefore essential to prove the correct operation of these contracts via formal verification.

In this document, we shall explain what formal verification of smart contracts is and illustrate it with Mi-Cho-Coq, the formal verification tool developed by Nomadic Labs.

[1] A security token is the representation of a financial security (for example, a company share or real estate asset) by a token.

[2] A stablecoin is a digital asset class the aim of which is to offer price stability compared to another asset, often a fiat currency (euro, dollar, etc.) such as, for example, the LUGH project. Different price stabilization methods are used depending on the project, such as sequestration or algorithmic stabilization.

[3] A Liquidity Pool (LP) is a smart contract in which tokens are blocked. A liquidity provider sends its funds to the pool in order to provide liquidity. From that point, it may no longer transfer its funds. Nevertheless, it receives LP tokens, which represent its part of the pool. It may, when it decides to do so, recover its funds. It also receives the exchange fees paid by the service users.

Definition of the main concepts

Definition of the main concepts 1/2

Smart Contract : Computer program that is deployed on the blockchain. On Tezos, these programs are written or (when they are written in a language other than Michelson) compiled in a language proprietary to Tezos: Michelson. This language was devised to facilitate formal verification.

Specifications of a code: Description of tasks that a computer code has to perform in a particular environment. They bring together all requirements to be met for this code, such as its behavior and the results expected when it is executed.

Unit tests : Process that, by a succession of tests, verifies the correct operation of a computer program. It compares the specification of a code with its performance, for a certain amount of data provided as input.

Definition of the main concepts 2/2

Formal logic: Deductive mathematical verification that calls on so-called mathematical logical reasoning, which uses a vocabulary and particular symbols as well as logical connectors.

Coq : Formal proof assistant that provides a formal language for writing mathematical definitions, executable algorithms and theorems with a semi-interactive environment for developing proof verification. A prominent tool that is well established in code verification, particularly in critical areas.

Mi-Cho-Coq : Formal verification library for smart contracts written in Michelson. It enables a Michelson script to be modeled in a mathematical model on which the reasoning that establishes its compliance with a specification is possible.

Motivations

Motivations 1/2

During the design of smart contracts, it is important to ensure that they are doing what they were designed to do.

Owing to the data that they handle and because, once deployed, you can no longer simply change the code, it is necessary to have guarantees for the security of the code: in other words, that its execution behaves as had been predicted by the specifications.

In fact, in certain areas where there is a low margin for error, a code that reacts unpredictably may be very costly and incur heavy human and financial consequences.

Ensuring that a smart contract is behaving correctly is not a trivial task. A standard good practice is to carry out unit tests. In practice, such tests only cover a finite number of cases. Consequently, how do we ensure that a program matches its specification and is mathematically correct?

This is where formal verification comes in. All methods and tools that address this issue constitute formal methods. It involves methods based on a mathematical representation of a program that allow reasoning about it. They are ranked according to the level of guarantees that they provide: from bug detection (tests, PBT testing, model-checking, etc.) to code correction and even the generation of correct code by construction.

Motivations 2/2

The idea then is to find a mean of **translating these smart contracts** into a language **that will be used in a proof assistant** (this is where Coq comes in). Before their deployment, certain smart contracts may be considered as critical, requiring a detailed audit. In this case, in addition to unit tests, they can be formally proven, using the Coq proof assistant.

It should be noted that there are other tools for verifying smart contracts on the Tezos blockchain. The main ones include:

- Why3: a program verification platform that consists in generating a series of proof obligations for each property of the specification. This tool is particularly used by the Archetype smart contract language.
- K-Michelson: Michelson interpreter developed by Runtime Verification, which develops a generic language specification tool, K-framework.

Some syntactical elements of the Coq language

Some syntactical elements of the Coq language 1/2

Le langage Coq :

– Boolean type definition:

```
Inductive bool : Type :=
  | true
  | false.
```

– Definition of a function that returns the denial of a Boolean (Fixpoint keyword to define a recursive function):

```
Definition negb (b:bool) : bool :=
  match b with
  | true => false
  | false => true
  end.
```

A few syntactical elements of the Coq language 2/2

Here is an example of proof that may be implemented:

```
Example test_orb1: (orb true false) = true.
Proof.
simpl. (* ===> true = true *)
reflexivity. (* ===> no more subgoals *)
Qed[1] .
```

In the example above, it involves showing that *true or false* $\Rightarrow$ *false*. The first line matches the one that we want to demonstrate. The lines between the `Proof` and `Qed` keywords constitute the proof.

All words between `Proof` and `Qed` are called **tactics**. These are functions that allow us to make our demonstration step by step.

All tactics are referenced at:
https://coq.inria.fr/refman/proof-engine/tactics.html#coq:tacv.destruct-eqn

[1] QED = quod erat demonstrandum; what was to be shown (in Latin).

Proof creation using Mi-Cho-Coq

Creation of proofs using Mi-Cho-Coq

As Coq is proof software that has its own syntax; the difficulty has been in finding a way to translate the Michelson code into Coq.

Mi-Cho-Coq is the library that allows you to do this. When you wish to prove a smart contract written in Michelson, you must use this library and import the code that you wish to formally prove.
The syntax to be used at the start of the file containing the proof is as follows:

```
Require Import Michocoq.macros.
Import syntax.
Require Import Michocoq.semantics.
Require Import Michocoq.util.
Import error.
```

For a more technical understanding of the operation of Mi-Cho-Coq, this tutorial can be viewed: https://gitlab.com/nomadic-labs/mi-cho-coq/

Execution of a simple proof using Coq software

Proof execution with the Coq software 1/3

We shall execute the proof of: Example test_orb2 (Example keyword below) using the Coq proof assistant.

```
Require Import Michocoq.macros.
Import syntax.
Import comparable.
Require Import NArith.
Require Import Michocoq.semantics.
Require Import Michocoq.util.
Import error.
Require List.

Inductive bool : Type :=
  | true
  | false.

Definition negb (b:bool) : bool :=
  match b with
  | true => false
  | false => true
  end.

Definition andb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => b2
  | false => false
  end.

Definition orb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => true
  | false => b2
  end.

Example test_orb2: (orb false false) = false.
Proof.
simpl.
reflexivity.
Qed.
```

```
1 subgoal                                        (1/1)
orb false false = false
```

Proof : false or false = false

Messages ↗ Errors ↗ Jobs ↗

Proof execution with the Coq software 2/3

```
Require Import Michocoq.macros.
Import syntax.
Import comparable.
Require Import NArith.
Require Import Michocoq.semantics.
Require Import Michocoq.util.
Import error.
Require List.

Inductive bool : Type :=
  | true
  | false.

Definition negb (b:bool) : bool :=
  match b with
  | true => false
  | false => true
  end.

Definition andb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => b2
  | false => false
  end.
Definition orb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => true
  | false => b2
  end.

Example test_orb2: (orb false false) = false.
Proof.
simpl.
reflexivity.
```

```
1 subgoal
                                        (1/1)
false = false
```

Messages ↗ Errors ↗ Jobs ↗

Simplification step for the left part of the expression using *orb* definition

Proof execution with the Coq software 3/3

```
Require Import Michocoq.macros.
Import syntax.
Import comparable.
Require Import NArith.
Require Import Michocoq.semantics.
Require Import Michocoq.util.
Import error.
Require List.

Inductive bool : Type :=
  | true
  | false.

Definition negb (b:bool) : bool :=
  match b with
  | true => false
  | false => true
  end.

Definition andb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => b2
  | false => false
  end.
Definition orb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => true
  | false => b2
  end.

Example test_orb2: (orb false false) = false.
Proof.
simpl.
reflexivity.
Qed.
```

```
No more subgoals.

Messages ↗   Errors ↗   Jobs
```

Reflexivity step compares both terms of the previous egality (false=false) and ends the proof

End of proof.

Limitations

Limitations

Here are the main limitations of the formal logic.

Limitation of a general nature:

- It is **difficult to formally define the expected behavior of a contract**. However, a proof depends on the quality of the formal definitions associated with it. Consequently, a poorly defined contract may, even if it has been formally proven, have undesirable behaviors.

Limitations specific to Mi-Cho-Coq:

- A formal proof of a smart contract **does not take account of problems related to its Gas consumption.**[1]. Consequently, an evaluation dedicated to the cost of a smart contract is relevant to complement the formal logic tests.
- Mi-Cho-Coq allows you to show a contract's calls but **not on a series of several independent calls**. However, other tools allow you to do this; this is, for example, the case of Concert.[2]

[1] Unit of measurement of the cost of executing a smart contract in time and resources. A price in ꜩ will correspond to a quantity of Gas.
[2] Certification tool for smart contracts in Coq.

Conclusion

Conclusion

Mi-Cho-Coq is a tool that allows you to obtain certain guarantees with regard to the execution of a smart contract deployed on the Tezos blockchain. In fact, when it involves creating and deploying smart contracts, it is essential to verify that the code meets the specification that characterizes it.
Nowadays, many smart contracts deployed with prior audit and unit tests have shown serious flaws, particularly in the context of DeFi[1] (Decentralized Finance), which increases the risks of financial loss if a flaw is detected.
During the creation of smart contracts, as well as conducting unit tests, formally verifying the program created using proof software helps to reduce considerably the possibility of vulnerabilities and undesirable behaviors.
The Tezos blockchain and its code were designed and developed in order to obtain a high level of code security guarantees, either through technological choices or its architecture. In particular, its native Michelson smart contract language facilitates formal verification. This is why it is generally easier to formally prove contracts on the Tezos blockchain, compared to other blockchains whose smart contract languages do not facilitate formal verification.

Mi-Cho-Coq has in particular helped to formally verify the standard of FA1.2 tokens on Tezos, Dexter 2 (version 2 of the decentralized Dexter exchange platform), Liquidity Baking (a single-asset decentralized exchange offered by the Granada amendment), or even the multisignature contract used by the Tezos client.

[1] All financial applications based on smart contracts, some of these application cases being similar to so-called traditional finance such as loans.

nomadic labs

Let’s keep the conversation going

https://tezos.com
https://developers.tezos.com