nomadic labs

La vérification formelle de smart contrats

Sommaire

Introduction

Introduction

La blockchain Tezos a été conçue pour assurer la sécurité et la conformité du code requises, notamment pour les cas d'utilisation de grande valeur comme l'émission et la gestion de Security Tokens [1], ou de stablecoins [2], ou encore pour le blocage de fonds dans un contexte tel que les pools de liquidité [3].
Michelson, le langage de smart contrat natif de la blockchain Tezos, facilite la **vérification formelle** en apportant des garanties : c'est une méthodologie couramment utilisée dans les environnements critiques tels que les industries aérospatiale, nucléaire et des semi-conducteurs. Cette méthodologie consiste à définir les spécifications d'un code informatique et à prouver que l'exécution de ce dernier fait bien ce pour quoi il a été conçu.

Tout contrat déployé sur une blockchain publique est quasiment impossible à réécrire ou retirer de l'historique. Par conséquent, pour certaines applications, les tests unitaires ne sont pas suffisants pour s'assurer qu'elles ont le fonctionnement désiré. Il est donc primordial de prouver le bon fonctionnement de ces contrats via la vérification formelle.

Dans ce document, nous allons expliquer ce qu'est la vérification formelle de smart contrats et l'illustrer avec Mi-Cho-Coq, l'outil de vérification formelle développé par Nomadic Labs.

---

1. Un security token est la représentation d'un titre financier (par exemple une action d'une entreprise ou un bien immobilier) par un token.
2. Un stablecoin est une classe d'actif numérique dont le but est d'offrir une stabilité de son prix par rapport à un autre actif, souvent une monnaie fiat (euro, dollar...) comme par exemple le projet LUGH. Différents moyens de stabilisation des prix sont utilisés selon le projet, notamment la mise sous séquestre ou la stabilisation algorithmique.
3. Un pool de liquidité ou Liquidity Pool (LP) est un smart contrat dans lequel sont bloqués des tokens. Un fournisseur de liquidités envoie ses fonds à la pool afin d'y apporter de la liquidité. Il ne peut plus transférer ses fonds à partir de ce moment. Néanmoins, il reçoit des LP tokens (Liquidity pool tokens) qui représentent sa part de la pool. Il peut, quand il le décide, récupérer ses fonds. Il reçoit également les frais de change payés par les utilisateurs du service.

nomadic labs

Définition des concepts principaux

Définition des concepts principaux 1/2

Smart Contrat : Programme informatique que l'on déploie sur la blockchain. Sur Tezos, ces programmes sont écrits ou compilés (lorsqu'ils sont écrits dans un autre langage que Michelson) en un langage propre à Tezos : Michelson. Ce langage a été pensé pour faciliter la vérification formelle.

Spécifications d'un code : Description des tâches que doit réaliser un code informatique dans un environnement particulier. Elles regroupent l'ensemble des exigences à satisfaire pour ce code, notamment son comportement et les résultats attendus lors de son exécution.

Tests unitaires : Procédé permettant, par une succession de tests, de vérifier le bon fonctionnement d'un programme informatique. Cela permet de comparer la spécification d'un code avec sa réalisation, pour un certain nombre de données qui lui sont données en entrée.

Définition des concepts principaux 2/2

Logique formelle : Vérification déductive mathématique faisant appel à des raisonnements dits de logique mathématique, qui utilisent un vocabulaire et des symboles particuliers ainsi que des connecteurs logiques.

Coq : Assistant de preuves formelles qui fournit un langage formel pour écrire des définitions mathématiques, des algorithmes exécutables et des théorèmes avec un environnement semi-interactif de développement de vérification de preuve. Outil figure de proue et bien installé dans la vérification de code notamment dans les domaines critiques.

Mi-Cho-Coq : Bibliothèque de vérification formelle pour des smart contrats écrits en Michelson. Elle permet de modéliser un script Michelson dans un modèle mathématique sur lequel le raisonnement établissant sa conformité à une spécification est possible.

Motivations

Motivations 1/2

Lors de la conception de smart contrats, il est important de s'assurer qu'ils font ce pour quoi ils ont été conçus.

De par les données qu'ils manipulent et du fait qu'une fois déployé on ne peut plus simplement changer le code, il est nécessaire d'avoir des garanties de sûreté du code : c'est-à-dire que l'exécution se comporte comme le cahier des charges l'avait prédit.

En effet, dans certains domaines où la marge d'erreur est faible, un code réagissant de façon imprévisible peut être très coûteux et entraîner de lourdes conséquences humaines et financières.

S'assurer du bon comportement d'un smart contrat n'est pas une tâche triviale. Une bonne pratique courante est d'effectuer **des tests unitaires**. En pratique, ces tests ne couvrent qu'un nombre fini de cas. Par conséquent, comment s'assurer qu'un programme correspond bien à sa spécification, et est mathématiquement correct ?

C'est ici qu'intervient la vérification formelle. L'ensemble des méthodes et outils adressant cette problématique constitue les méthodes formelles. Il s'agit de méthodes basées sur une représentation mathématique d'un programme et permettant de raisonner à son sujet. Elles sont classées selon le niveau de garanties qu'elles apportent : de la détection de bugs (tests, PBT testing, model-checking...) à la correction de code voire à la génération de code correcte par construction.

Motivations 2/2

L'idée est alors de trouver un moyen de **traduire ces smart contrats** (ici intervient Mi-Cho-Coq) en un langage **qui sera utilisé dans un assistant de preuve** (ici intervient Coq).

Avant leur déploiement, certains smart contrats peuvent être considérés comme critiques, et nécessitant un audit poussé. Dans ce cas, en plus des tests unitaires, ils peuvent être prouvés formellement, à l'aide de l'assistant de preuve Coq.

À noter qu'il existe d'autres outils pour vérifier des smart contrats sur la blockchain Tezos. Parmi les principaux :

- Why3 : une plateforme de vérification de programmes consistant à générer pour chaque propriété de la spécification, une série d'obligations de preuve. Cet outil est notamment utilisé par le langage de smart contrats Archetype.
- K-Michelson : interpréteur de Michelson développé par Runtime Verification, qui développe un outil générique de spécification de langages, K-framework.

Quelques éléments syntaxiques du langage Coq

Quelques éléments syntaxiques du langage Coq 1/2

Le langage Coq :

– Définition du type booléen :

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

– Définition d'une fonction renvoyant la négation d'un booléen (mot clé Fixpoint pour définir une fonction récursive) :

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

Quelques éléments syntaxiques du langage Coq 2/2

Voici un exemple de preuve pouvant être implémentée :

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

Dans l'exemple ci-dessus, il s'agit de montrer que *vrai ou faux* $\Rightarrow$ *vrai*. La première ligne correspond à ce que l'on veut démontrer. Les lignes entre les mots clés `Proof` et Qed constituent la preuve.

Tous les mots situés entre `Proof` et Qed sont appelés des **tactiques**. Ce sont des fonctions permettant de faire pas à pas notre démonstration.

L'ensemble des tactiques est référencé à :
https://coq.inria.fr/refman/proof-engine/tactics.html#coq:tacv.destruct-eqn

1. QED = quod erat demonstrandum; ce qu'il fallait démontrer (en latin).

Création de preuves à l'aide de Mi-Cho-Coq

Création de preuves à l'aide de Mi-Cho-Coq

Coq étant un logiciel de preuve possédant sa propre syntaxe, la difficulté a été de trouver un moyen de traduire du code de Michelson vers Coq.

Mi-Cho-Coq est la librairie permettant de faire cela. Lorsque l'on souhaite prouver un smart contrat écrit en Michelson, il faut donc utiliser cette librairie et importer le code que l'on souhaite prouver formellement.
La syntaxe à utiliser au début du fichier contenant la preuve est la suivante :

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

Pour une compréhension plus technique du fonctionnement de Mi-Cho-Coq, ce tutoriel pourra être consulté :
https://gitlab.com/nomadic-labs/mi-cho-coq/

Exécution d'une preuve simple à l'aide du logiciel Coq

Exécution de preuve avec le logiciel Coq 1/3

Nous allons exécuter la preuve de : *Example test_orb2* (ci-dessous après le mot clé *Example*) à l'aide de l'assistant de preuve Coq.

```
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
```

Preuve à effectuer (faux ou faux = faux)

Messages ↗ Errors ↗ Jobs ↗

Exécution de preuve avec le logiciel Coq 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
```

Étape de simplification de la partie gauche de l'égalité à l'aide de la définition orb

Messages ↗ Errors ↗ Jobs ↗

Exécution de preuve avec le logiciel Coq 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

Étape de réflexivité, compare les deux terme de l'égalité précédente (false=false) et termine la preuve

Fin de la preuve.

Limitations

Limitations

Voici les principales limitations de la logique formelle :

Limitation d'ordre général :

- Il est **difficile de définir formellement le comportement attendu d'un contrat**. Or, une preuve dépend de la qualité des définitions formelles qui lui sont associées. Par conséquent, un contrat mal défini peut, même s'il a été prouvé formellement, avoir des comportements indésirables.

Limitations spécifiques à Mi-Cho-Coq :

- Une preuve formelle d'un smart contrat **ne tient pas compte des problèmes liés à sa consommation de Gas** [1]. Par conséquent, une évaluation dédiée du coût d'un smart contrat est donc pertinente comme complément des tests de logique formelle.
- Mi-cho-coq permet de montrer des propriétés sur les appels d'un contrat, mais **pas sur une série de plusieurs appels indépendants**. Cependant d'autres outils permettent de le faire, c'est par exemple le cas de Concert [2].

1. Unité de mesure du coût d'exécution d'un smart contrat en temps et en ressources. À une quantité de Gas va correspondre un prix en ꜩ.
2. Outil de certification de smart contrats en Coq.

Conclusion

Conclusion

Mi-Cho-Coq est un outil permettant d'obtenir certaines garanties quant à l'exécution d'un smart contrat déployé sur la blockchain Tezos. En effet, lorsqu'il s'agit de créer et de déployer des smart contrats, il est primordial de vérifier que le code respecte la spécification qui le caractérise.
Aujourd'hui, de nombreux smart contrats déployés avec audit préalable et tests unitaires ont montré des failles graves, notamment dans le contexte de la DeFi [1] (Finance décentralisée), ce qui accroit les risques de perte financière si une faille est décelée.
Lors de la création de smart contrats, en plus de faire des tests unitaires, vérifier formellement le programme créé à l'aide de logiciels de preuve permet de réduire fortement la possibilité de vulnérabilités et de comportements indésirables.
La blockchain Tezos et son code ont été conçus et développés dans un souci d'obtention d'un fort niveau de garanties de sûreté de code que ce soit au travers des choix technologiques ou de son architecture. En particulier, son langage natif de smart contrats Michelson facilite la vérification formelle. C'est pourquoi il est généralement plus facile de prouver formellement des contrats sur la blockchain Tezos, comparé à d'autres blockchains dont les langages de smart contrats ne facilitent pas la vérification formelle.

Mi-Cho-Coq a notamment permis de vérifier formellement le standard de tokens FA1.2 sur Tezos, Dexter 2 (la version 2 de la plateforme d'échanges décentralisés Dexter), Liquidity Baking (un échange décentralisé mono-asset proposé par l'amendement Granada), ou encore le contrat multisignature utilisé par le client Tezos.

1. Ensemble des applications financières reposant sur des smart contrats, certains de ces cas d'application étant similaires à la finance dite traditionnelle comme les prêts.

nomadic labs

Continuons cet échange

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