分散台帳技術の実装における安全性の形式検証

Formal Verification of Safety of a DLT Consensus Algorithm

齋藤 新 * その他 †

あらまし 本論文では分散台帳技術の安全性を形式仕様記述支援系である TLA+ で行った。定理証明を用いて検証。2 段階のリファインメントによりモデル化と検証を行った。TLA の使用で得た知見も紹介。

Summary. To appear.

1 序論

分散台帳技術は仮想通貨 (virtual currency) のプラットフォームとして提案・実装され，その非集中性・耐障害性・耐改竄性により注目を浴びた．例えば仮想通貨の代表例であるビットコイン [1] は 2009 年に運用が開始されてから現在まで，ネットワークを停止させたり，台帳に不整合を発生させたりするような攻撃が成功していない[1]．その成果を踏まえて，通貨にとどまらず様々なデジタル資産 (暗号資産; crypto-asset とよばれる) を扱えるプラットフォーム [2] が登場するなど，適用範囲が拡大している。

分散台帳技術はインセンティブ設計・暗号理論などを活用することにより，その安全性を担保している．ところが実装に求められる要件が増大するにつれそのアーキテクチャは複雑化の一途をたどっている．適切にアーキテクチャが設計されているか，実装がアーキテクチャに忠実に行われているかを確かめることは容易ではない．

例えば Ethereum とよばれる実装では，システム上の通貨を利用することによりプログラム (スマートコントラクト) を実行することができる．その例として，ユーザからの資産を集めて投資しその利益を配分する，DAO (Decentralized Autonomous Organization) と呼ばれるスマートコントラクトが存在する．ところがスマートコントラクト間で呼び出しが可能な Ethereum の機能と，DAO の実装の些細な不具合を利用することにより，DAO が管理する通貨をある限り引き出す攻撃が行われ，数十億円相当の損失となった [3]．また IOTA という実装においては暗号部分の設計に瑕疵があることが明らかになった ?[]．台帳側ではなくてクライアント側の実装においてもバグが発生しており，ユーザの保有する通貨が塩漬けになる，ネットワークに不正なトランザクションが放出される，といった例が報告されている．

このような事態を避けるためシステムが適切に設計・実装されていることを保証する方法として，形式検証を用いることが有効である．特に，一般に行われているテストベースの設計・開発と比べて次のような利点がある．まず，テストでは不可能な「設計の正しさ」を検証することができる．また，全状態を網羅する検証を行うことにより，分散システムにおいて重要であるコーナーケースにおける振る舞いについても確認することができる．さらに，検証により設計の前提条件などが明確となることにより，分析や改良をより確実・容易に行うことができるようになる．一般に，形式検証にかかるコストおよび必要知識はテストのそれに比べて大きく，人手を必要とする場合もあるが，そのメリットは不具合による損失・被害の大きさに見合うものであると思われる．

*Shin Saito, IBM Research–Tokyo

†Others, other

[1] ただし，そこからハードフォークにより分裂したビットコインゴールドに対しては 2018 年に取引結果を覆す 51%攻撃が成功している．これは小規模な通貨においては攻撃に必要な計算能力が少なくてよいためである．

本論文ではプライベート型分散台帳技術の一実装である Hyperledger Fabric (以下，Fabric) を対象とし，コンセンサスに関するコア部分を形式検証系である TLA$^+$ で検証する．Fabric は処理性能の追求などのため，一般的なブライベート型の分散台帳とは異なるアーキテクチャを採っている．そのためアーキテクチャが設計者の意図する分散台帳の性質を満たすかは自明ではない．検証は定理証明により行い，より抽象的なモデルからの詳細化 (refinement) を多段階で行うことによりその安全性を証明する．これにより証明を簡潔に行うことができ，モデルを拡張・再利用するための適切な抽象化が得られる．

本論文の構成は以下の通りである．2 節では分散台帳技術および Fabric について概説する．3 節では本論文で使用する形式仕様検証系である TLA$^+$ について紹介する．4 節では Fabric の TLA$^+$ による検証について詳細を説明する．**??**節・6 節ではそれぞれ，得られた知見および関連研究について述べる．

2 分散台帳技術

分散台帳技術はネットワーク上に散在するノード間で状態を同期するシステムおよびアーキテクチャの名称である．状態は任意のデータでよいが，キー・バリュー・ストア (KVS) であることが多い．ネットワーク上のノードに対して，クライアントはノードの状態を変更するためのリクエストであるトランザクションを送信できる．ほとんどの実装では受け取ったトランザクション列を同期することにより，その適用結果であるノード状態の同期を行う．

ブロックチェーンとは分散台帳のアーキテクチャのひとつである．複数のトランザクションをまとめてブロックに格納し，ブロック単位でノード間のデータ共有を行う．ブロックをチェーン上に並べそれらの同期を取る．各ブロックには直前のブロックのハッシュ値を格納することによりブロックの改竄を防止する．なお，ブロックチェーンの同期を取ることをコンセンサスとよぶ．

分散台帳の実装は参加者[2]の制限の有無により，2 つの種類に大別される．パブリック型 (public/permissionless blockchain) は参加者 (ノードおよびクライアント) の資格・数に制限がない．そのため，コンセンサスには計算量などをもとにしたインセンティブベースのアルゴリズムが用いられる．例えばビットコインのコンセンサスにはハッシュの計算量をもとにしたプルーフ・オブ・ワーク (PoW) が使われている．一方，パーミッション型[3] (permissioned blockchain) においては認証ノードに許可された参加者しか加われない．また，ある時点においては参加ノードの数が固定されている．これは 1980 年代から研究されていた分散合意の問題設定であり，コンセンサスには多数決ベースのアルゴリズムが使用される．例えば Hyperledger Fabric では，バージョン 0 において PBFT アルゴリズム [4] が使用されていた．

2.1 ビザンチン耐性

分散台帳技術においてはノードやネットワークの障害に対して耐性がある，すなわち状態の同期が正しく行われることが要求される．ここでいうノードの障害は故障などによる停止のほか，ノードで動作するプログラムにバグがある，ノードに悪意がある，などの理由によりコンセンサス・プロトコルに従わない状況を含む．このような障害はビザンチン将軍問題 **?**[] にちなんでビザンチン障害とよばれる．ビザンチン障害に対して耐性を持つことをビザンチン耐性 (Byzantine Fault Tolerance; BFT) とよぶ．

既知の結果として，参加ノードのうち高々 f 台がビザンチン障害にある場合において，ネットワークが耐性を持つためには最低 $3f+1$ 台のノードが必要であること

[2] ここでいう参加者とは，ネットワークに参加するノード，および，クライアントがトランザクションを発行する際に使用するユーザアカウントのことである．

[3] プライベート型，コンソーシアム型などともよばれる．

が知られている **?**[].

2.2 Hyperledger Fabric

Hyperledger Fabric [5] (以下，Fabric) はパーミッション型ブロックチェーンの 1 実装であり，Hyperledger コンソーシアムの 1 プロジェクトである．ビジネス用途向けに様々なアーキテクチャの工夫がなされている．Fabric はコンソーシアム型とよばれるアーキテクチャを採用する．ネットワークは複数の組織からなり，各組織の認証機関が参加を許可するノードおよびユーザを指定する．組織内の参加者は互いに信用するモデルである．

Fabric では各ノードを Docker コンテナとして動作させることにより，OS を問わずに稼働させることができる．また，スマートコントラクトも独立したコンテナとすることにより，複数の言語[4]がサポートされている．

処理性能向上のため，Fabric バージョン 1 以降では一般に見られる分散台帳の実装とは異なり，トランザクションを整列する専用のノードを持つこと，トランザクションの提出に先立ってスマートコントラクトを仮実行すること，を特徴とするアーキテクチャが採用されている．本論文の目的はこの安全性を検証することである．コンセンサス・アルゴリズムの詳細については 4 節で説明する．

3 形式検証系 TLA$^+$

形式検証系は対象とするシステムを数学的に記述し，それが満たす性質を機械的に検証するためのシステムである．分散台帳技術の検証においては並列システムおよびその性質を記述できる検証系が求められる．

本研究では Lamport らにより提案および開発された TLA$^+$ を使用する．TLA$^+$ は集合論と時相論理に基づく言語，TLA (Temporal Logic of Actions) で記述されたシステムを検証するツール群である．検証器としてモデル検査器 TLC が提供されていたが，後に定理証明システム TLAPS が追加され，より大規模なシステムの検証に耐えるようになった．さらに近年，手続き型言語 PlusCal および TLA への変換器が提供されたこともあり，ソフトウェアエンジニアなどにも使用しやすくなった．その結果，企業でのソフトウェア開発への適用例が増えている．上記各ツールに加えて Eclipse ベースの IDE である TLA Toolbox が提供されている．本論文では TLA Toolbox のバージョン x.y.z および TLAPS のバージョン x.y.z を使用した．

3.1 TLA$^+$

blank

4 Hyperledger Fabric の安全性検証

状態遷移系 $\mathbf{S_1}, \mathbf{S_2}$ において、$\mathbf{S_1}$ の (観測可能な) 状態の遷移列が $\mathbf{S_2}$ のそれでもあるとき、$\mathbf{S_1}$ は $\mathbf{S_2}$ の refinement であるとよぶ。$\mathbf{S_1}$ が $\mathbf{S_2}$ の refinement であり $\mathbf{S_2}$ が安全性を満たす場合には $\mathbf{S_1}$ も満たすことが示される。

本論文ではこれを用いて MVCC 台帳が台帳仕様の正しい実装であることを示す。

4.1 Refinement Mapping を用いた安全性証明

Abadi らは、状態遷移系 $\mathbf{S_1}, \mathbf{S_2}$ において、$\mathbf{S_1}$ の状態を $\mathbf{S_2}$ の状態に写す関数で、ある条件を満たすものを refinement mapping と定義し、mapping が存在する場合には $\mathbf{S_1}$ は $\mathbf{S_2}$ の refinement になること (およびある条件下ではその逆が成り立つこと) を示した [Abadi and Lamport '88]。

ここではより限定された形のモデルを扱う。低位のモデル (= 実装)、高位のモデ

[4] v1.4 の時点では Go, Node.js, Java をサポートする．

ル (= 要求仕様) をそれぞれ $\mathbf{S_1}$、$\mathbf{S_2}$ とし、さらに $\mathbf{S_2}$ は内部状態を持たないとする、つまり $\Sigma_2 = \Sigma_E$ とみなす。またどちらのモデルも自明な liveness property を持つ、つまり $L_i = \Sigma_i^\omega$ とする。このとき refinement mapping f とは以下の条件を満たす Σ_1 から Σ_2 への写像である。

1. すべての $s \in \Sigma_1$ について、$f(s) = \Pi_E(s)$
2. $f(F_1) \subseteq F_2$
3. $\langle s,t\rangle \in N_1$ ならば $\langle f(s), f(t)\rangle \in N_2$ または $f(s) = f(t)$ である

このとき条件 1 より $f(\langle e,y\rangle) = e$、つまり f は Σ_E への射影関数であることがわかる。これに従い前記条件 2 および 3 を書き直すと以下のようになる。

2' $\{e \mid \langle e,y\rangle \in F_1\} \in F_2$
3' $\langle (e,y),(e',y')\rangle \in N_1$ ならば $\langle e,e'\rangle \in N_2$ または $e = e'$ である

さらに $\mathbf{S_1}$ において、外部状態 e は内部状態 y の関数 π であり、状態機械 $\mathbf{S_1}$ は内部状態のみに依存して定義されていると仮定する。このとき、前記条件はさらに以下のように変形できる。

2" $y \in F_1 \Rightarrow \pi(y) \in F_2$
3" $\langle y,y'\rangle \in N_1$ ならば $\langle \pi(y),\pi(y')\rangle \in N_2$ または $\pi(y) = \pi(y')$ である

これは TLA$^+$ においては $e \triangleq \pi(y)$ とおいたうえで $L(y)!Spec \Rightarrow H(e)!Spec$ を示すことに等しい。これは前述の 2" および 3" を示すことにより TLA$^+$ で証明できる。

$$\land\ L(y)!Init \Rightarrow H(e)!Init \tag{1}$$

$$\land\ Inv \land L(y)!Next \Rightarrow Inv' \land\ \lor H(e)!Next \lor \text{UNCHANGED}\langle a,b,c\rangle \tag{2}$$

5 考察

5.1 TLA$^+$ におけるモデル化と証明の難易度

TLA$^+$ に

5.2 モデルと実装

形式検証の課題の 1 つとして，モデルと実装[5]の乖離がある．

まず，先に要求仕様としてのモデルを作成・検証し，それをもとにコードを実装する場合を考える．分散プロトコルを一からデザインする場合にはこちらのアプローチが望ましい．このとき，モデルが正しいことを検証できたとしても，実装へ抽出または変換する際に誤りがあると形式検証を行った意味が失われる．この主要な原因としてはモデルの記述力が低いことなどに起因してモデルが単純化されすぎている場合と，逆に記述力が高すぎるため，プログラミング言語で実装可能なコードに変換する際に誤りが混入する場合があると考えられる．TLA$^+$ の場合は後者にあたる可能性が大きい．TLA$^+$ における operation はマクロ的な働きをする上に，様々なデータ型や操作が集合およびその上の演算として表現されている．集合には内包表記が使用可能で，さらに非決定的代入も存在する．これらは TLA$^+$ で記述する際には頻出するが，例えば Go 言語のような低レベル言語で直接に実装するのは難しい．この場合には，上記のモデルをいったん実装に近い形に詳細化して，それをプログラムに変換するのが望ましい．

一方，実装がすでに存在し，その性質を検証する場合にはモデルが実装の振る舞いを正しく反映しているかが問題となる．例えば TLA$^+$ では PlusCal という手続き型言語から TLA への変換器が提供されており，これと同様に実装から自動的にモデルを抽出することが望ましい．Hyperledger Fabric の場合はすでに実装があり，バージョンアップが頻繁であるため，現在のところはこのアプローチが現実的であると思われる．しかし実際に動作するレベルのコードから変換して検証器が耐えうる程

[5] ここでいう実装は，プログラミング言語による実装を指す．

度の規模のモデルに収まるかについては未知数である．

6 関連研究

これまでに様々な形式検証系が研究されている．

関数型言語をベースとしたものでは Coq, Agda, Isabelle/HOL が著名である．これらは帰納データ型とタクティクに基づく豊富な証明能力があるが，デフォルトでは時相論理を扱えないため今回は採用しなかった．なお，Coq に Logic of Events と呼ばれる理論を導入し，PBFT アルゴリズムの検証を行った研究がある **?**[]．Coq には証明からプログラムを抽出する機能があり，形式検証で問題になるモデルと実装の乖離を防ぐ意味で有用な手法であると思われる．

TLA$^+$ では Paxos，Raft を始めとする分散合意プロトコルが検証されている．

状態遷移系 (Kripke 構造) をベースとするものには TLA$^+$ の他に Event-B や NuSMV が挙げられる．前者は定理証明ベースであり後者はモデル検査器を使用する．これらの中では TLA$^+$ がもっとも記述の自由度が高い．たとえば Event-B はモジュールの概念がないため，大規模なシステムを記述することにはあまり向かない．また，基本的なデータ型しか存在しないために実装に近い記述をすることには困難が伴う．時相論理で性質を記述することができないことも欠点である．NuSMV は時相論理で性質を記述することができ，モジュールの概念がある．ただし複数のモジュールが非同期的に動作するようなモデルを容易には書きにくい[6]．またモデル検査の都合上有限の状態遷移しか許されない．

また，並行プロセス計算である π 計算に基づくものとして ProVerif がある．これはセキュリティプロトコルの検証に特化しており,「イベント A が起きたときには必ず過去にイベント B が起きている」といった他の検証器にはない，後ろ向きの性質が記述できるが，一方で記述できない時相論理式が多い．そのため今回のような一般の分散システム検証には不向きである．

7 まとめ

本論文ではパーミッション型の分散台帳技術である Hyperledger Fabric の安全性に関する形式検証を行った．Fabric のコンセンサスアルゴリズムの主要部分を取り出したものについて，低レベルのモデル (実装) が高レベルのモデル (仕様) を詳細化していることを証明することによりその安全性を検証した．形式検証系には集合論および時相論理をベースとする TLA$^+$ を用いた．豊富なデータ型と SMT ソルバなどを用いた定理証明器により，シンプルに証明を構築することができた．現在の TLA$^+$ のモデル検査器の性能が向上すれば，定理証明とモデル検査を組み合わせることにより，さらに効率の良い検証が可能になると思われる．今後の研究は Fabric の独自機能をさらに盛り込むこと，プログラミング言語による実装とモデルの対応を正しく取ることについて取り組んでいきたい．

謝辞 test あれば書く。

参考文献

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`, 2008.
[2] Ethereum Project. `https://www.ethereum.org/`.
[3] Deconstructing theDAO Attack: A Brief Code Tour. `http://vessenes.com/deconstructing-thedao-attack-a-brief-code-tour`.
[4] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pp. 173–186, 1999.
[5] Hyperledger fabric – hyperledger. `https://hyperledger.org/projects/fabric`.

[6] 厳密に言うと現在はその機能があるが削除予定である．