分散台帳の実装における安全性の形式検証

Formal Verification of Safety of a Blockchain Consensus Algorithm

齋藤 新 *

あらまし 本論文では分散台帳技術におけるコンセンサス・アルゴリズムの安全性を形式的に検証した．パーミッション型分散台帳である Hyperledger Fabric を対象とし，検証には形式仕様フレームワークである TLA$^+$ を用いた．仕様を 2 段階に詳細化するというアプローチでモデル化を行い，定理証明器を用いて検証を行った．その結果，再利用性が高い適切な抽象化が得られ，見通しのよい検証の道筋を立てることができた．

Summary. This paper formally verifies the safety of a consensus algorithm used in a DLT implementation. The target DLT is Hyperledger Fabric, one of the permissioned blockchains. We use a formal verification framework TLA$^+$. We describe the system as multi-level models and prove its refinement relation using its theorem prover. This gives us reusable abstraction model and good perspective for verification.

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．ところがプラットフォームの仕様と実装の些細な不具合を悪用し，スマートコントラクト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]．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による損失が報告されている．

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．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．

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．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

*Shin Saito, IBM Research–Tokyo

[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．

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．5 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 (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 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」 [5] 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が知られている．

2.2 Hyperledger Fabric

Hyperledger Fabric [6] (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．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．

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．

[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 形式検証系 TLA$^+$

本研究では Lamport らにより提案および開発された形式検証フレームワークである TLA$^+$ [7] を使用する．TLA$^+$ は集合論と時相論理に基づく言語である TLA 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たえるようになった．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．

3.1 仕様記述言語 TLA

TLA$^+$ で使用する仕様記述言語は TLA (Temporal Logic of Actions) と呼ばれる．詳細は [7] に譲るが，掲載されているサンプル Hour Clock (図 1) を参考にしたコードを用いて概要を説明する．なお，図では仕様が ASCII 文字列で表記されているが，本文ではそれらに対応する数学的な記法を用いる．

```
1  ---------------------------- MODULE HourClock ------------------------------
2  \* Hour Clock example from Lamport's book
3
4  EXTENDS Naturals
5  VARIABLES hr
6
7  Init == hr \in 1..12
8  Next == hr' = IF hr /= 12 THEN hr+1 ELSE 1
9
10 Spec == Init /\ [][Next]_hr
11
12 THEOREM TypeSafety == Spec => []Init
13 ==============================================================================
```

図 1　Hour Clock の TLA による記述例

TLA ではモジュールが 1 つの状態遷移系を記述する単位である．例ではモジュール *HourClock* を定義している．EXTENDS 宣言で他のモジュールを取り込むことができる．各種ライブラリもモジュールとして定義されており，ここでは自然数に関するライブラリである *Naturals* をインポートしている．モジュールの状態変数は VARIABLE(S) 宣言で，定数は CONSTANT(S) 宣言で定義する．これらはすべて集合であり，各々の自然数も定義された集合であるとみなされる．例では状態変数として *hr* を定義している．定数は定義していない．

Init の定義から始まるその後の行はすべて定義である．システムの記述に必要な値・集合・述語などはすべて定義として表現される．なお，定義の展開はマクロ的に行われる．

TLA において，状態遷移系は初期条件，および，状態遷移における事前・事後条件に関する論理式として定義される．*Init* は初期条件を定義する．ここでは *hr* は 1 から 12 までのいずれかであると定義している．このように非決定的な定義が可能であることが TLA の特徴の 1 つである．*Next* は状態遷移を定義している．これは事前条件と事後条件の論理積で表現される．その際に，*hr′* のように状態変数にプライム記号がついたものは「次の状態における変数の値」を表す．例では *hr* の値が 12

でなければ[3]，次の状態では $hr+1$ になり，12 であれば 1 になる，としている．なお例では事前条件が指定されていないので，すべての状態において遷移が可能である．

これをまとめてシステムの振る舞い *Spec* は時相論理式の定義 $Spec \triangleq Init \wedge \Box[Next]_{hr}$ で与えられる．ここで $[Next]_{hr}$ は $Next \vee$ UNCHANGED hr を表し，さらに UNCHANGED hr は $hr' = hr$ を表す．これは状態が変化しないステップ，stuttering を意味している．TLA のすべての時相論理式は stuttering に対して不変であることが求められており，実際に *Spec* の定義式もそうなっている．

最後に証明したいシステムの性質が時相論理式 *TypeSafety* として THEOREM 宣言で定義されている．例では不変条件，つねに *hr* の値が 1 から 12 までのいずれかであること，が表現されている．これは TLA$^+$ では type invariant と呼ばれる不変条件であり，安全性の一種である．一般に，不変条件の成立は $Spec \Rightarrow \Box Inv$ として表現される．なお，例では定理の証明が書かれていない．実際には仕様の記述者が証明支援系 TLAPS を使って構築していくこととなる．

4 TLA$^+$ による Hyperledger Fabric の安全性検証

ここでは Fabric のコンセンサスアルゴリズムの一部を単純化し，その安全性を TLA$^+$ の定理証明支援系 TLAPS により証明する．なお，仕様および証明の全体については `https://github.com/shinsa82/fabric-formal-model` から入手可能である．

4.1 Hyperledger Fabric のコンセンサス・アルゴリズム

Fabric のコンセンサス･アルゴリズムは処理性能の向上，台帳不整合の早期発見などを目的として，他の実装には見られない，MVCC (Multiversion Concurrency Control) をベースとしたものが用いられている．

例として，台帳をキーとして各ユーザ名，値としてその残高が記載された KVS であるとし，ユーザ A から B に送金する例を考える．ここで，KVS の各エントリはキーおよび値に加えてバージョンを保持している構造とする．このバージョンはエントリに書き込みがあるたびにインクリメントされる．

ある時点において，A の残高が 400 でそのバージョンは v2 (以下 $(A, 400, v2)$ と書く)，B は $(B, 300, v1)$ であるとする．A から B へ 100 を送金するトランザクションに対して，具体的なアルゴリズムは以下のようになる．ただし単純化のためトランザクションをブロックにまとめる処理をせず，トランザクション単位で処理を行うものとする．

1. クライアントは A から B への送金トランザクションを作成．各ノードにその仮実行 (simulation) を依頼する．
2. 各ノードは送金スマートコントラクトを仮実行し，その結果である read-write set (以下 RWSet) を返す．Read-set にスマートコントラクトが仮実行中に読み出したキーとそのバージョンが記録される．例では $\{(A, v2), (B, v1)\}$ である．Write-set には仮実行中に書き込んだキーとその値が記録される．例では $\{(A, 300), (B, 400)\}$ である．
3. クライアントは各ノードから返る RWSet たちを元のトランザクションにエンドースメントとして含め，オーダラー (順序付けサービス) と呼ばれる特殊ノードに送る．ここで，全ノードから RWSet が返るとは限らないし，(ビザンチン障害を仮定するので) その結果が正しいとは限らないことに注意する．
4. オーダラーは複数のクライアントから到達するトランザクションたちを一列に並べて，順にノードにブロードキャストする．
5. ノードはオーダラーからトランザクションを受け取ったらその検証 (MVCC 検

[3] # または /= で ≠ を表す．

証) を行う．
(a) エンドースメントが 3 個未満である場合には検証失敗とする．
(b) エンドースメントが 3 個以上ある場合，そのうち 2 つ以上の RWSet が一致する場合はそれを正として次のステップに進む．そうでない場合は検証失敗とする．
(c) 前ステップで正とした RWSet の read-set のすべてのエントリについて，そのバージョンが「現在のノードの対応エントリのバージョン」と等しい場合には検証成功とし，そうでない場合には検証失敗とする．

検証に失敗した場合はそのトランザクションに invalid フラグをつけて無視する．検証に成功した場合はそのトランザクションをブロックチェーンに追記し，RWSet を現在の状態に適用する．その結果，各ノードにおける KVS のエントリは $(A, 300, v3)$ および $(B, 400, v2)$ となる．

上記のアルゴリズムは高々1 台のノードがビザンチン障害にある場合については期待通りに動作するはずである．すなわち，その条件下において，ステップ 5b で正とされる RWSet は read-set: $\{(A, v2), (B, v1)\}$, write-set: $\{(A, 300), (B, 400)\}$ となるはずである．また MVCC 検証に成功した場合，送金トランザクションを実行して台帳の状態を更新する必要はなく，RWSet をコミットすることで同じ効果が得られるはずである．

4.2 Refinement による安全性証明

Fabric のコンセンサス・アルゴリズム安全性を証明するにあたり，現在の実装モデル化し，それが安全性を満たすことを直接証明するのではなく，本研究では多段階の refinement により証明する方針を取る．

状態遷移系 $\mathbf{S_1}, \mathbf{S_2}$ において，$\mathbf{S_1}$ により許される任意の (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．このとき，$\mathbf{S_2}$ 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．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 と定義した．そしてその mapping が存在する場合には $\mathbf{S_1}$ は $\mathbf{S_2}$ の refinement になること (さらにある条件下ではその逆が成り立つこと) を示した [8]．Refinement の定義が状態の無限列の集合の包含関係に基づくことに対し，refinement mapping に関する条件はほとんどが状態間の対応関係を考えれば十分であり，証明が容易である．

Refinement の方針は次のようになる．まず最上位のモデルとして，分散台帳ネットワークを外部からブラックボックスとして見た時の振る舞いをモジュール *Ledger* としてモデル化する．これは単一の状態機械として振る舞い，クライアントからのトランザクションを整列して順に状態に適用していけばよい．次に，MVCC 検証を行う台帳をモジュール *MVCC_Ledger* として定義する．このモデルは依然として単一のノードとして振る舞うが MVCC 検証を導入している．トランザクションの投入時に仮実行結果を保存しておき，トランザクションの処理時には MVCC 検証を行い，成功したら RWSet をコミットすることにより状態の更新を行う．最後に，より低位のモデルとして，複数のノードからの RWSet を収集するモデルをモジュール *MVCC_Consensus_Ledger* として記述する．なお，前述のアルゴリズムに現れるオーダラーについては取り除いて単純化しているが，これを陽にモデルで表現して検証することも可能である．

4.3 台帳仕様 *Ledger*

最上位の台帳仕様である *Ledger* から一部抜粋したものを図 2 に示す．

このモジュールは内部状態として KVS の状態 *state*，ブロックチェーン *chain*，チェーンにおける未処理のトランザクションのインデックスである *index* を定義している (4–6 行目)．9–10 行目ではデータ型に相当する集合を定義している．なお *TX*，*NULL*

```
1  ------------------------------- MODULE Ledger --------------------------------
2  EXTENDS Sequences, Integers, TLAPS, Datatype
3
4  VARIABLES state, \* current state of the ledger state machine.
5            chain, \* blockchain, a list of received transactions.
6            index \* unprocessed TX index at the blockchain.
7  vars == <<state, chain, index>>
8
9  ChainEntry == [tx: TX, is_valid: BOOLEAN \union {NULL}]
10 Chain == Seq(ChainEntry)
11
12 Init ==
13    /\ state = InitState \* state is at the initial state, and
14    /\ index = 1
15    /\ chain = <<>> \* empty transaction queue.
16
17 SubmitTX(tx) ==
18    /\ chain' = Append(chain, [tx |-> tx, is_valid |-> NULL])
19    /\ UNCHANGED <<state, index>>
20
21 ProcessTX_OK ==
22    LET
23       f == chain[index].tx.f
24    IN
25       \* /\ Len(chain) >= index
26       /\ index \in DOMAIN chain
27       /\ chain' = [chain EXCEPT ![index].is_valid = TRUE] \* update validity flag
28       /\ index' = index + 1 \* increment the index.
29       /\ state' \in f[state] \* perform non-deterministic state transition by f.
30
31 ProcessTX_ERR ==
32    LET
33       f == chain[index].tx.f
34    IN
35       \* /\ Len(chain) >= index
36       /\ index \in DOMAIN chain
37       /\ chain' = [chain EXCEPT ![index].is_valid = FALSE] \* see above.
38       /\ index' = index + 1 \* see above.
39       /\ UNCHANGED state \* state does not change due to invalid TX.
40
41 Next == (\E tx \in TX: SubmitTX(tx)) \/ ProcessTX_OK \/ ProcessTX_ERR
42
43 Spec == Init /\ [][Next]_vars
44 =============================================================================
```

図 2　Ledger モジュール (抜粋)

などは共通モジュール *Datatype* で定義されている．12 行目ではモジュールの初期状態を定義している[4]．17–39 行目では 3 種類の状態遷移に関する論理式を定義している．TLA ではこれらをアクションとよび，TLA の名称の由来になっている．例えばアクション *ProcessTX_OK* (21–29 行目) はトランザクションの処理が成功した場合の状態遷移を定義している．トランザクションに格納されている関数，つまりスマートコントラクト (23 行目の f) を現在の状態に適用し更新する．ここでは f は非決定的であることを許容している．処理したトランザクションには valid であるというフラグを付与し，インデックスを 1 つ進めている．なお，*ProcessTX_ERR* は不要に見えるが，後に refinement mapping を定義するためにこのようになっている．

次に，このモデルに対する安全性検証の例を図 3 に示す．不変条件としては type invariant (省略) に加えて，ブロックチェーンに含まれるトランザクションが前から順に処理されていること (*ChainInv*) を含めている．階層化された証明 (14–24 行目) が TLAPS の記法に従って与えられている．これにより *Ledgder* が台帳として期待する性質を満たしていることが形式的に確認できる．

```
1  ------------------------------- MODULE Ledger --------------------------------
2  ...
3  Spec == Init /\ [][Next]_vars
4
5  ---
6  \* Invariant (safety) on the blockchain
7  ChainInv ==
8      \* chain = (processed part) + (unprocessed part)
9      /\ \A i \in 1 .. index-1: chain[i].is_valid \in BOOLEAN
10     /\ \A i \in {i \in Nat: index <= i} \cap DOMAIN chain: chain[i].is_valid = NULL
11
12 Inv == TypeInv /\ ChainInv
13
14 (* Invariant (safety) on the high-level Ledger *)
15 THEOREM LedgerInv == Spec => []Inv
16 PROOF
17     <1>1 Init => Inv
18         BY InitStateAxiom DEF Init, Inv, TypeInv, ChainInv, Chain
19     <1>2 Inv /\ [Next]_vars => Inv'
20         <2>1 SUFFICES ASSUME TypeInv, ChainInv, [Next]_vars PROVE Inv' BY DEF Inv
21         <2>2 CASE Next
22             <3> USE DEF Inv, Next
23 ...
24     <1> QED BY PTL, <1>1, <1>2 DEF Spec
25
26 ===============================================================================
```

図 3　Ledger モジュールの安全性 (抜粋)

[4]配列のインデックスは 1 から始まる．また，<<a,b,c>> は数学的記法では $\langle a,b,c \rangle$ と書き，タプルおよび配列を表す．当然であるがこれらもすべて集合である．

4.4 MVCC 台帳仕様 *MVCC_Ledger*

図 4 に *MVCC_Ledger* モジュールを抜粋したものを示す．ここでは MVCC 検証を導入するために，ブロックチェーンにエンドースメントを格納する箇所を設ける (3, 5 行目)．このモデルでは，トランザクションの投入時にエンドースメントを計算し格納するように変更されている (7–12 行目)．トランザクションの処理時には read-set に関する条件が成功した場合の限り，RWSet をコミットすることにより *state* の変更を行う (14–24 行目)．

```
1  ------------------------------ MODULE MVCC_Ledger ------------------------------
2  ...
3  Endorsement == RWSet
4  \* each entry of blockchain now has a RWSet.
5  ChainEntry == [tx: TX, endorsement: Endorsement, is_valid: BOOLEAN \union {NULL}]
6  ...
7  SubmitTX(tx) ==
8      LET
9          end == endorsement(tx)
10     IN
11         /\ chain' = Append(chain, [tx |-> tx, endorsement |-> end, is_valid |-> NULL])
12         /\ UNCHANGED <<state, index>>
13 ...
14 ProcessTX_OK ==
15     LET
16         f == chain[index].tx.f
17         rwset == chain[index].endorsement
18     IN
19         \* /\ Len(chain) >= index
20         /\ index \in DOMAIN chain
21         /\ SameOnRSet(state, rwset)
22         /\ chain' = [chain EXCEPT ![index].is_valid = TRUE] \* update validity flag
23         /\ index' = index + 1 \* increment the index.
24         /\ state' = Commit(state, rwset) \* perform non-deterministic state transition by rwset.
25 ...
26 h_state == state
27 h_chain == Proj(chain)
28 h_index == index
29 ...
30 HSpec ==
31     INSTANCE Ledger WITH state <- h_state, chain <- h_chain, index <- h_index
32
33 THEOREM Refinement == Spec => HSpec!Spec
34 ...
35
36 =============================================================================
```

図 4　MVCC_Ledger モジュール (抜粋)

26–28 行目では refinement mapping を定義している．エンドースメントに関するフィールドを削除する以外，変換はほぼ不要である．30 行目では mapping された変

数を用いて上位モジュールである *Leader* をインスタンス化している．これにより，このモジュールが *Ledger* の refinement となることは，39 行目の命題 *Refinement* で表現できる．詳細は省略するが，これは refinement mapping の定義 [8] において，$\mathbf{S_2}$ が内部状態を持たず，$\mathbf{S_1}$ の外部状態が内部状態の関数として定義される場合に相当する．

4.5 MVCC コンセンサス台帳仕様 *MVCC_Consensus_Ledger*

このモデルにおいては複数のノードからの RWSet をエンドースメントとして収集する．したがって，*Endorsement* の定義を *RWSet* から *Seq*(*RWSet*) に変更する必要がある．さらに，トランザクションを投入する際にネットワーク環境を反映した適切なエンドースメントを生成する必要がある．例えば 4 台のノード中高々1 台がビザンチン障害にあるという設定であれば，エンドースメントの数は 0-4 個であり，そのうち高々1 つが障害ノードからの RWSet であるとする．そして障害ノードからの RWSet の中身は任意のデータであり，その他の RWSet は正しい仮実行結果である，とすればよい．TLA の非決定的代入はこのようなモデル化に極めて有用である．

あとはこのモジュールが *MVCC_Ledger* の refinement であることを示せばよく，これは命題 *Spec* ⇒ *MVCC_Ledger*!*Spec* を証明すればよい．紙面の都合上，詳細は省略する．

5 考察

5.1 TLA$^+$ によるモデル化と検証の容易性

前述したように TLA$^+$ にはデータ型が存在せず，すべて集合として扱われる．例えば自然数は自然数型 `Nat` を持つ値，ではなくて自然数の集合 *Nat* の元であるとみなされる．

型を扱わないことにはメリットとデメリットがある．TLA$^+$ で記述される仕様は検証対象のシステムと外部の環境をまとめて扱う closed-system specification である．その際，環境から与えられる値に関しては型を制限しないことが現実のモデル化という観点からは望ましい．一方で，型システムが導入されていれば自明であるにもかかわらず，TLA$^+$ では値がどの集合に属するか証明する必要がある場合が多く効率的でない．現実的にはシステムの内部で扱うデータは統制されていることがほとんどであるから，部分的に型を指定・推論できる仕組みがあると便利である．

TLA$^+$ の証明支援系である TLAPS は自然演繹に基づく前向き推論を採用している．Coq などに見られる後ろ向きの推論とは異なり，現在のゴールが自動的に構築されるわけではないので，試行錯誤が必要になるケース多いが，IDE のサポートが十分であるとは言い難い．他の証明支援系でも見られるように SMT ソルバなどを用いた自動証明は利便性が高いが，ゴールに対して適用する定理をピンポイントで指定する形式ではないので，意図した証明にたどり着くのに苦労するケースも見られる．

なお，TLA$^+$ にはモデル検査器が付属しているが，少し複雑な仕様に対しては状態爆発により現実的な時間では検証が完了せず，今回は使用を見送った．

5.2 モデルと実装の整合性

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

[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．

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．この場合，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 [9]．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．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．

[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．

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] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, Vol. 4, No. 3, pp. 382–401, July 1982.
[6] Hyperledger fabric – hyperledger. https://hyperledger.org/projects/fabric.
[7] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
[8] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, Vol. 82, No. 2, pp. 253–284, May 1991.
[9] Vincent Rahli, Ivana Vukotic, MarcusV¥"olp, Paulo Esteves-Verissimo. Velisarios: Byzantine fault-tolerant protocols powered by coq. In Amal Ahmed, editor, *Programming Languages and Systems*, pp. 619–650, Cham, 2018. Springer International Publishing.