SMTソルバーを使ったカスタマイズ可能なブロックチェーン合意ルールの検証

河原 亮[1,a)]

概要：本稿は，ブロックチェーン基盤である Hyperledger Fabric を対象に，分散合意プロトコルの一部としてカスタマイズ可能な分散合意ルールの検証手法を提案した．近年，ブロックチェーンアプリケーションが多様化するにつれてその分散合意ルールがカスタマイズ可能なものが出てきたが，カスタマイズによってブロックチェーンが本来満たすべき性質を保障することが難しいのが課題であった．
本手法は (i) ブロックチェーンネットワークに特徴的なビザンチン障害耐性や組織間の信頼 (共謀への耐性) を要求として検証でき，それらを効率よく記述可能な DSL を定義した，(ii) ブロックチェーン基盤の共通の挙動を検証対象からはずし，さらに探索する障害の状態をワーストケースに限ることでモデルの状態数を削減した，(iii)DSL モデルを SMT ソルバーのスクリプトに変換し形式的に検証することにより，網羅的なテストに比べて高速である．という特徴を持つ．実際に，典型的なモデルを使って検証しモデルの問題点を指摘できることを示したほか，検証時間の比較やモデルの LOC を比較して有効性を確認した．

Verification of customizable blockchain consensus rule using SMT solver

1. 背景

ブロックチェーンは分散合意 (コンセンサス) プロトコルと暗号学的なテクニックを利用して，信頼関係がない個人や組織の間で単一の取引台帳の共有を実現する分散トランザクション処理システムであり，近年暗号通貨をはじめ，国際送金，証券，保険，サプライチェーンなど [1][2]，信頼関係がないためにデータの集中管理が適さず，台帳の正確性，正当性の確保のために多大なコストを必要とする各種の業務への適用が試みられている．このようなアプリケーションの開発には実装済みのブロックチェーン基盤ソフトウェアを用いることが多くなっている．

上記のような用途の多様化に応えるため，ブロックチェーン基盤にはいくつかの軸でバリエーションが存在する [3]．ひとつがメンバーシップの管理方式の違いである．ブロックチェーンネットワークはメンバー (参加主体である個人または組織) が所有するノードからなり，ノードは台帳をもつが，誰でも参加できるパブリック型 (パーミッションレス型) と，限られたメンバーのみが使用するプライベート型 (パーミッション型) のネットワークがある．またトランザクション処理において，一部のブロックチェーン基盤にはスマートコントラクトと呼ばれるプラグイン可能な台帳更新処理をサポートするものが出てきている．

更なる用途の多様化の一環として，分散合意プロトコルの一部をカスタマイズ可能にしたタイプのブロックチェーン基盤が開発されている．クライアントから受信したトランザクションをこれらのノードが処理して台帳を更新する際に，複数のノード間で同じように台帳を更新することを保障する手順が分散合意プロトコルである．従来ブロックチェーン基盤が用いる分散合意プロトコルは Proof of Work(PoW)[4] や Practical Byzantine Fault Tolerance(PBFT)[5] など，全てのノードを平等に扱う既存のプロトコルを用いることが一般的であった．

しかし，プライベート型ブロックチェーン基盤のひとつである Hyperledger Fabric [6] [7] では，分散合意の一部であるトランザクションごとの処理結果についての合意は，アプリケーションごとに定義されるルールに従って行われるため，どの組織のノードの主張をより重視するかといった柔軟な重み付けが可能となった．同様の仕組みはパブリック型ブロックチェーン基盤の分散合意アルゴリズム

[1] IBM Research - Tokyo
19-21, Nihonbashi Hakozaki-cho, Chuo-ku, Tokyo 103–8510, Japan
[a)] ryokawa@jp.ibm.com

である Ripple Protocol[8] や Stellar Consensus Protocol[9] でも見られる．これらにおいては同一のネットワーク内において，合意を取るべきノード集合をノードごとに指定できる仕組みになっている．

ここでブロックチェーンネットワークのような信頼関係のないメンバー間で使われる場合に重要なのが，不正目的の偽の情報による台帳の改竄の阻止である．いわゆるビザンチン将軍問題 [10] となるため，分散合意にはある程度のビザンチン障害耐性が必要で，PoW や PBFT はそのために用いられていた．

ここで，前述のカスタマイズ可能な分散合意ルールにはひとつの課題が挙げられる．すなわちカスタマイズされた分散合意ルールが，本来ブロックチェーンが満たすべき性質であるビザンチン障害耐性を持っていることを，アプリケーション開発者がアプリケーションごとに保障しなければいけない点である．現状では，Hyperledger Fabric や Stellar Consensus Protocol ではルール記述の設定がサポートするカスタマイズの範囲が広いため，うまく設計しなければ障害発生時の台帳の安全性や処理の継続性が犠牲になる可能性がある．これを防ぐためにはアプリケーション設計時にルールを検証する必要がある．一般的なモデル検査ツールを使ったり，シミュレータープログラムを開発することが考えられるが，アプリケーションごとに多様な故障モードおよびネットワーク構造をもつシステムのモデリングやシミュレーターの開発は技術的にも工数的にも負担になっていた．

本論文は，アプリケーション開発者を対象とした，上記のカスタマイズ可能な分散合意ルールの設計を検証するための手法を提案する．本手法は以下のような特徴を持つ．

- ブロックチェーンネットワークに特徴的なネットワーク構造およびビザンチン障害耐性などのブロックチェーン特有の要求，および合意ルールなど，アプリケーションごとに可変なパラメーターのみで記述できる DSL(Domain Specific Language[11]) を提供する．
- ブロックチェーン基盤の共通の挙動を検証対象からはずし，さらに探索する障害の状態をワーストケースに限ることでモデルの状態数を削減する．
- DSL モデルを SMT ソルバーのスクリプトに変換し形式的に検証することにより，網羅的なテストに比べて高速である．

対象のブロックチェーン基盤は Hyperledger Fabric であり，3 節で詳しく説明される．一般化については将来課題である．2 節では関連研究を，3 節では提案手法を説明する．4 節では評価を行い，その結果を 5 節で議論する．6 節は結論である．

2. 関連研究

分散フォールトトレラントシステムのプロトコルの検証に形式手法を用いる試みはこれまでにも多く行われてきた．理論的には，非同期の分散システムでは 1 ノードでも任意の障害が存在する場合には分散合意できないことが知られている [12]．そのため，実用的な分散合意プロトコルは，通信の同期性，障害の種類，整合性などに何らかの前提条件をおいて定義される．この前提条件が手法によってまちまちであるため，プロトコルも多様であり，それぞれ検証が必要になっている．

例えば HO モデル [13] では同期・非同期通信の区分，およびネットワーク障害とノード障害の区分を統一的に扱った上で，クラッシュ障害などの親切な障害 (benign faults) への耐性を議論している．ビザンチン障害に分類される値の間違いなどへは対応していない．PSync[14] も HO モデルを採用し，DSL を定義してモデル検査によるプロトコル検証を実現している点で本手法とアプローチが似ているが，同様にビザンチン障害への対応は未実装となっている．

値の誤りなどを含む悪意のある障害 (malicious fault) への HO モデルの拡張が提案され [15]，インタラクティブ定理証明器によるプロトコルの形式検証 [16] も行われている．障害の有無に加えて値を考慮する点では本論文と同様のモデリングとなっているが，分散合意プロトコルの検証に特化した DSL を用いているわけではない．

A. John らは分散フォールトトレラントシステムのプロトコルの多くが，プロセス間メッセージによる投票と定足数 (quorum) の形で記述されることに着目し，整数の制約を記述できる SMT(Satisfiability Modulo Theory) ソルバーとモデル検査器を用いてパラメトリックに検査する手法を提案している [17]．本手法でも同じ理由により SMT ソルバーを採用している．しかし彼らの研究では分散合意プロトコルの検証に特化した DSL を用いていない．

また上記の研究の手法はいずれも任意の分散合意プロトコルの検証を志向する一般的な手法であるのに対し，本手法は特定のブロックチェーン基盤用のプロトコルのカスタマイズ部分の検証に特化して，より記述の効率化と高速化を図っている点が異なる．さらに，3 節で述べるように，本研究の対象であるブロックチェーンネットワークでは，参加するメンバーが組織であって複数のノードを所有しうる．この場合，定足数や分散システムに対する要求が組織ごとに定義されるケースや，障害のパターンとして同一組織内のノードが共謀するケースを検証する必要があるが，これらの研究では言及されていなかった．

別のアプローチとしては，PRISM のような確率的モデル検査 [18] があり，ランダム性をもつ分散フォールトトレラントプロトコルや，システム中の素子がランダムなハードウェア故障を起こす場合の検証に用いられている．しかし本問題では，プロトコルやアルゴリズムは決定的でありランダム性があるのはノードやネットワークの障害であること，その障害にバグや意図的な値の改ざんも対象とする

ため，ノード間の障害発生が独立ではないこと，等の難しさがあり，その確率的なモデリングは自明ではない．このため本論文では採用しなかった．

3. 提案手法

3.1 Hyperledger Fabric のアーキテクチャー

Hyperledger Fabric はブロックチェーン基盤の実装であり，The Linux Foundation が主催する Hyperledger プロジェクト群の一つである．想定されている用途は企業コンソーシアムなど複数組織間にまたがる各種の業務である．プライベート型のブロックチェーン基盤に分類され，スマートコントラクトをサポートする [7]．ブロックチェーンネットワークの参加者の単位は組織であり，1 つの組織は複数のノードやユーザーを持つ．1 つのトランザクションは複数のノードに送信される．各ノードはスマートコントラクトの定義に基づいてトランザクションを同じように実行し実行結果を他のノードと比較する．実行結果についてノード間で合意が取れれば，結果を各ノードが持つ台帳にコミットする．

アーキテクチャー上の特徴として，分散合意における「実行」と「順序」の分離が挙げられる．バージョン 0.6 までは Hyperledger Fabric は分散合意プロトコルに PBFT を採用していたが，現行方式には以下の利点がある [6]．

- ノード利用効率の向上．トランザクションの順序についての合意に必要な正常ノードの割合 (2/3) よりトランザクションの実行結果についての合意に必要な正常ノードの割合 (1/2) は小さい．よって実行ノード数を節約できる余地がある [19]．
- ノード間の信頼関係モデルの柔軟性．PBFT では順序と実行結果の両方について全てのノードを均一に扱って合意を得る．一方，実行に関しての合意ルールは本来はアプリケーション依存であってよく[*1]，分離したことでそれが可能である．

現行方式では，トランザクションを実行し，台帳を更新，保管する一般のノードと，トランザクションの順序付けに特化したノードが分離している．順序付けノードは，それらどうしで専用のクラスターを組み，障害耐性のある順序付けサービスを提供する．順序の合意に関する分散合意プロトコルは既存のよく知られた実装からの選択式になっており，Kafka+Zookeeper(クラッシュ障害耐性をサポート) や SBFT (ビザンチン障害耐性をサポート[*2]) などがある．

一方，一般ノード同士で行われる実行結果についての合意は，トランザクションの順序が決まっているという前提の下で行われるため，個々のトランザクションごとに，実行結果がノード間で一致しているかを同期的に見るものである．よって，状態変化のような時間的な概念を考慮する必要がない．この際どれだけのノードが一致したら合意を得たとするかという基準をエンドースメントポリシーと呼ぶ．これはアプリケーション毎に定義する．詳細は次節で説明する．本論文では実行結果についての合意のステップにおいて，エンドースメントポリシーに焦点を当てる．

3.2 問題の規模

Hyperledger Fabric はプライベート型もしくはパーミッション型ブロックチェーンであるため，事前に決められた組織に所属するノードで構成される．典型的には企業などのコンソーシアムによる利用が想定されており，数 10 組織程度，ノード数にして 100 ノード程度が想定される．また性能の観点からも 100 ノード以上までのスケーラビリティがあることが主張されている [7]．

よって本提案手法においても 100 ノード程度までは実用的に使えることを目標とする．

3.3 エンドースメントポリシー

エンドースメントポリシーは，ノード間でのトランザクションの実行結果がどれほど一致すれば合意をとれたとみなすか，という基準を与える設定項目である．一般的にはノードは障害を起こす可能性があり，偶発的な故障やバグのほかスマートコントラクトや台帳に対する不正の結果，実行結果が返ってこなかったり，値が間違っている可能性がある．不一致の状態の値が台帳に書き込まれる (台帳の分岐) ことは防がなければならない．全数一致を要求すると安全ではあるが，障害があると処理を継続できなくなる．そこで通常は少数の不一致があっても一定数以上 (定足数, quorum) のノードで一致していれば合意したとみなす．合意した場合は台帳の更新をコミットする．

図 2 のようなノードと組織構造の場合に，次のようなエンドースメントポリシーがあるとする:

$$T(2, \text{org\_a.p1}, \text{org\_b.p1}, \text{org\_c.p1}) \qquad (1)$$

これは「指定された 3 ノード中 2 ノード以上で値が一致していること」という意味である．ここで $T(m, \text{node}_1, \cdots, \text{node}_n), m \in \mathbb{N}$ は閾値関数 (threshold function) とよび，引数で指定されたノード部分集合の実行結果による投票で，一致した結果の数が閾値 m を超えればポリシー成立でその結果の値を返し，超えなければ不成立で null を返すとする．引数の node_i 部分はノード以外に T も指定できる．本問題における定義は 3.5 節で行う．

3.4 提案手法の概要

提案手法の手順の概要を図 1 に示す．提案手法は入力として以下の 3 種類の要素を記述した DSL で記述されたモ

[*1] 著者注: 一部のアプリケーションにしか関与しない組織が存在しうるほか，組織ごとにノード数や運用基準が異なる場合がありうる．
[*2] v1.1 の時点では開発中

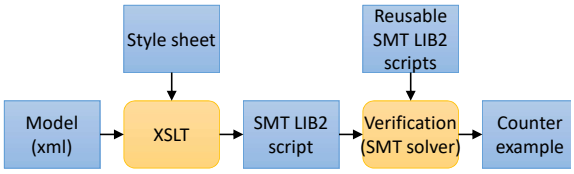

図 1　提案手法の概要

デルを受け取る．

- エンドースメントポリシー
- ブロックチェーンネットワーク構造
- 障害耐性に関する要求

入力はモデル変換により SMT ソルバーで検証可能な形式に変換される．検証の段階では SMT ソルバーによって必要な障害の状態が網羅的に探索され，与えられたエンドースメントポリシーの下で安全性，活性が成立するかが検証される．出力としては，検証結果と，仮に検証失敗の場合は具体的な反例を出力する．

実装としては，以下の技術を選択した．

- 入力モデルには XML 形式を採用した．
- モデル変換には XSLT (Extensible Stylesheet Language Transformation)[20] を用いた．変換エンジンには Apache Xalan-Java[21] を使用した．
- 検証用の SMT ソルバーには Microsoft Z3 Theorem Prover[22] を使用し，検証用モデルの記述には SMT Lib 2.0[23] の形式を採用した．

以下の節で各ステップについて詳細を説明する．

3.5　障害状態のモデリング

$\mathbb{B} = \{\text{true}, \text{false}\}$ を真偽値, $\mathbb{N}$ を自然数の集合とする．

$$s = (s.e, s.v) \in \mathbb{B} \times \mathbb{B} \tag{2}$$

はポリシー検証者の立場から見たノードの障害の状態を表す．ここで $s.e, s.v$ は状態 s の各成分である．

- $s.e \in \mathbb{B}$：トランザクションの実行結果が存在するか
- $s.v \in \mathbb{B}$：トランザクションの実行結果の値

よって $s = (\text{true}, \text{true})$ は正常状態，$(\text{false}, \text{false})$ はクラッシュ状態，$(\text{true}, \text{false})$ は値の誤り障害 (稼働しているがビザンチン障害) の状態を表す．実行結果が真偽値である理由は，一般的には実行結果は誤りも含めて多様な値をとりうるが，障害の有無という観点では正しい値 (true) か誤った値 (false) かだけが区別できればよく，また閾値関数における最悪ケースは誤った値が全て同一である場合であるからである．また，今後 s をノードの障害状態のように説明するが，HO モデル [13] と同様にネットワーク障害も区別せず含めることができる．そして 3.1 節のとおり，トランザクションごとの評価のため状態遷移も考慮しなくてよく，これらにより探索する状態空間を削減している．

閾値関数 $T : \mathbb{N} \times (\mathbb{B} \times \mathbb{B})^n \mapsto (\mathbb{B} \times \mathbb{B})$ はこのモデル化の範囲では以下のように定義される．

$$T(m, s_1, \cdots, s_n) = (T.e, T.v) \tag{3}$$

$$T.e = \left(\sum_i^n I(s_i.e \wedge s_i.v) \geq m\right) \vee \left(\sum_i^n I(s_i.e \wedge \neg s_i.v) \geq m\right) \tag{4}$$

$$T.v = \neg\left(\sum_i^n I(s_i.e \wedge \neg s_i.v) \geq m\right) \tag{5}$$

ここで m は閾値，$\{s_i\}$ は障害状態，$I : \mathbb{B} \mapsto \mathbb{N}$ は指示関数 (indicator function) で次のような変換である．

$$I(b) = \begin{cases} 1 & \text{if } b \text{ is true} \\ 0 & \text{if } b \text{ is false} \end{cases} \tag{6}$$

3.3 節で前述のように，引数 s_i にはノードの状態ないし別の $T(\cdots)$ を指定できる．例えば $T(2, T(1, s_1, s_2), s_3)$ である．エンドースメントポリシー EP は閾値関数 T の木構造として記述され，そのルート要素の各成分を値に持つ．

$$(EP.e, EP.v) = (T(\cdots).e, T(\cdots).v) \tag{7}$$

すなわち，$EP.e = \text{true}$ のとき，ポリシー成立とし，$EP.v$ の値で台帳が更新される．

3.6　要求のモデリング

ブロックチェーンシステムが実現すべき性質としては，障害耐性がある．すなわち，ネットワーク全体で $f \in \mathbb{N}$ 台のノードまで障害を起こしても，残りの正常ノードは分散合意が可能であることを要求する．このうち，障害ノードの台数についての制約は以下のようにかける．

$$FT_{\mathrm{G}}(f) = \left(\sum_i^n I(S_{\mathrm{F}}(s_i)) \leq f\right) \tag{8}$$

ここで，$S_{\mathrm{F}} : (\mathbb{B} \times \mathbb{B}) \mapsto \mathbb{B}$ はノードの状態 s が障害に相当するかを表す．

$$S_{\mathrm{F}}(s) = \neg(s.e \wedge s.v) \tag{9}$$

また，障害ノードの台数はネットワーク全体ではなく，各参加組織ごとに定義することもできる．

$$FT_{\mathrm{O}}(f_1, \cdots, f_{|Orgs|}) = \bigwedge_k^{Orgs} \left(\sum_i^{org(k)} I(S_{\mathrm{F}}(s_i)) \leq f_k\right) \tag{10}$$

ここで，$Orgs$ は組織の集合，$org(k)$ は組織 k のノード集合，f_k は組織 k に許容される障害ノードの数である．この範囲内であれば，システムは障害に耐えられるだけでなく，メンテナンスのために組織ごとに独立にノードを停止するといったこともできる．

次に，このような障害耐性の要求がある場合に検証すべき命題は以下のように安全性 (safety) と活性 (liveness) に分けられる．安全性は (想定される障害の範囲内では) エンドースメントポリシーが満たされるならば，正常なノード同士は正しい実行結果の値で合意されるということで，以下の条件が恒等的に成り立つことである．

$$FT \wedge EP.e \Rightarrow EP.v \tag{11}$$

ここで FT は FT_{O} または FT_{G} である．

活性は (想定される障害の範囲内では) エンドースメントポリシーがいつかは必ず成立するという要請である．よって以下の式が恒等的に成立することである．

$$FT \Rightarrow EP.e \tag{12}$$

ここで，現在考えているプロトコルではこのステップは同期的に行われているので「いつかは必ず」という部分については考慮する必要がなく，各トランザクションについて必ずおきなければならない．

次の要求として，組織間の信頼 (trust) が挙げられる．信頼のない参加者間では取引の不正を目的とした台帳の改竄の可能性があり，一種のビザンチン障害なので，あるノード数 f の障害までは上記の障害耐性つきの安全性により台帳を保護する．ところが今のケースでは，参加者は組織で複数ノードを所持しうるため，同一組織内のノードは共謀して f 台を上回るビザンチン障害を起こす可能性がある．したがって要求は，組織的なノードの共謀が起きた場合でも，間違った値で分散合意をしないということである．ある組織 j の共謀を考慮して，障害の範囲についての制約は以下のように修正される．

$$FT_{\mathrm{O}\setminus org(j)}(f_1,\cdots,f_{|Orgs|}) = \bigwedge_{k\neq j}^{Orgs}\left(\sum_{i}^{org(k)} I(S_{\mathrm{F}}(s_i)) \leq f_k\right) \tag{13}$$

この場合，安全性についての要求 (式 (11)) は以下のように修正される．

$$FT_{\mathrm{O}\setminus org(j)}(f_1,\cdots,f_{|Orgs|}) \wedge EP.e \Rightarrow EP.v \tag{14}$$

ここでは j は外部条件 (束縛変数) としている．すなわち問題で与えられた全ての組織 $\forall j$ の共謀を個別に検証する．

3.7 検証

安全性，活性の条件式 (11)，(12) および組織間信頼の条件式 (14) はいずれも以下のような形をしている．

$$A(x) \Rightarrow B(x) \tag{15}$$

ここで，x は問題の状態を記述する変数のベクトルであり，今回の場合はノードの障害状態 $s_1, s_2, \cdots$ に相当する．これが成立するかは以下の充足可能性の問題と等価である．

$$A(x) \wedge \neg B(x) \quad \text{unsatisfiable by } \forall x \tag{16}$$

よって式 (16) の形のアサーションを SMT ソルバーに入力として与え，充足可能性を検査させることにより，安全性，活性の条件式 (11)，(12) および組織間信頼の条件式 (14) が成立しているかの検査がそれぞれ可能である．

SMT ソルバーは与えられた問題が充足不可能である場合，unsat という表示をして終了する．この場合は検証対象の式が成立することが示せたので，本手法では検証は成功であるという表示を行う．問題が充足可能である場合，SMT ソルバーは充足可能な例を生成する．この例は，本手法では検証対象の式に対する反例となる．この反例は設計を見直す際に参考となるため，ユーザーに表示する．反例には具体的なノードの障害の状態 $s_1, s_2, \cdots$ が含まれる．

3.8 DSL とモデル変換

以下に入力モデルの DSL の XML スキーマの概略を示す．検証モデルを記述するより工数を減らすため，アプリケーションごとの可変部分のみをモデリングしている．

- endorsement policy：エンドースメントポリシーの構文をそのまま XML にマップし，閾値関数 T の合成を伴うエンドースメントポリシーを記述できる．
- network：組織 (organization) およびノード (peer) のタグを定義し，組織とその中のノードという階層構造のあるネットワークを表現する．
- requirement：許容する障害の範囲の種別 (FT_{G}, FT_{O}, or $FT_{\mathrm{O}\setminus org(j)}$) を指定する．いずれもパラメーターは f (許容される障害ノード数) である．

詳細な定義は A.2 節に，モデルの例を A.1 に示した．

エンドースメントポリシー中の閾値関数 T の引数 (子要素) としては別の閾値関数 T またはノード名を指定できる．現在，組織名は指定できないが指定できるように拡張することは可能である．また現状の Hyperledger Fabric の仕様に合わせて一つのエンドースメントポリシー内に同一のノード名が複数回登場することを許しており，エンドースメントポリシーは構文的には木構造であるものの，サブツリー間の値には依存関係がある．

network タグにおいては，組織は排他的でありノードを葉とする木構造となる．組織の多階層化は現在は Hyperledger Fabric が対応していないが将来的にはありうる．

次に入力モデルを検証用の SMT スクリプトに変換するルール (XSLT スタイルシート) の概要を示す．

- network：ノードの ID を名前にもつ Endorsement 型変数の宣言に変換する．これは 2 つの真偽値変数のタプルであり障害の状態を表す．例えば `<peer id="org_a.p1"/>` というタグは SMT Lib 2.0 スクリプトのヘッダー部分を含め以下のようになる．

```
(declare-datatypes (T1 T2) (
```

```
 (Pair (mk-pair (first T1) (second T2))) ))
(define-sort Endorsement () (Pair Bool Bool))
...
(declare-const org_a.p1 Endorsement)
```

- requirement： 要求の記述を式 (8), (10) で定義される式に展開する．これらの式はノードの集合におけるある特定の障害状態のノードの数に関する制約条件となり `assert` 命令の引数に変換される．この際，特定の状態のノードの数など，真偽値変数の集合 $b_1, \cdots, b_N$ 中の true 値の数え上げが必要になるが，これは `(add (I b1) (add (I b2) (add ... )))` のように足し算関数 `(add X Y)` の多段の呼び出しに変換される．ここで `(I b)` は指示関数 (式 6) である．
- endorsement policy： エンドースメントポリシーの記述を式 (4), (5) で定義される閾値関数の組み合わせに展開する．これらの式の和の計算 $\sum$ は，前述の requirement の場合と同様に足し算関数に展開される．

最後に，検証する式 (11), (12), (14) のうちの 1 つを式 (16) の形のアサーションに変換する．

4. 評価

4.1 典型的なケースでの検証例

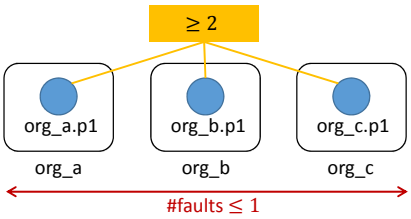

図 2 モデル M1 の組織構成とエンドースメントポリシー．

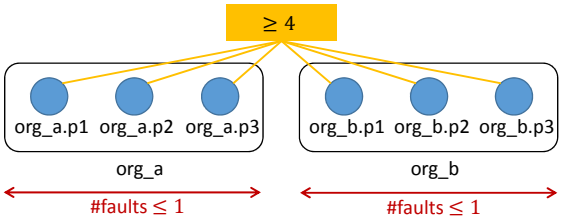

図 3 モデル M2 の組織構成とエンドースメントポリシー．

結果のわかっている小規模なケースに適用し，実際に検証ができることを示す．使用したネットワーク構造およびエンドースメントポリシーは以下の 4 種類である．

(1) M1 (図 2): 正常なポリシーのケース．
(2) M2 (図 3): 組織間信頼が保障されない．
(3) M3 (図 4): HA 構成だがビザンチン障害耐性がない．
(4) M4 (図 5): 正常ポリシーのケース．

なお，図の青い丸はノード，丸角の長方形の囲みは組織，その上の黄色い四角はエンドースメントポリシーのひとつ

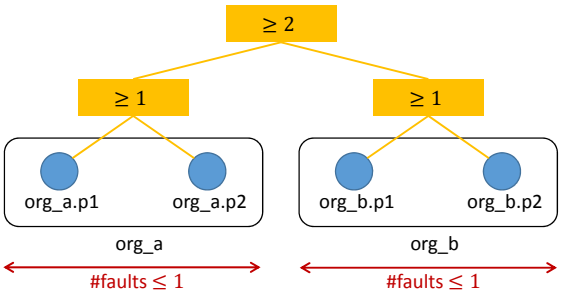

図 4 モデル M3 の組織構成とエンドースメントポリシー．

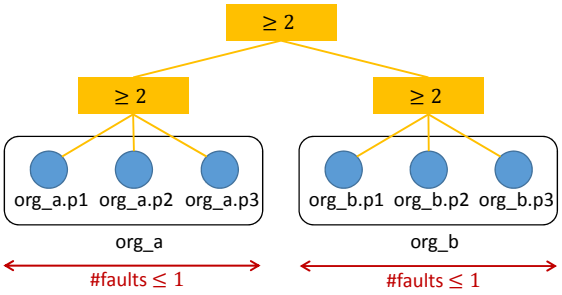

図 5 モデル M4 の組織構成とエンドースメントポリシー．

の閾値関数 T をあらわし，黄色い線は閾値関数の引数になっていることを表す．組織の下の#faults は許容されるべき障害ノード数である．

本手法による検証結果を表 1 に示す．

表 1 検証結果

Model	Safety	Liveness	Trust
M1	OK	OK	OK
M2	OK	OK	NG
M3	NG	NG	NG
M4	OK	OK	OK

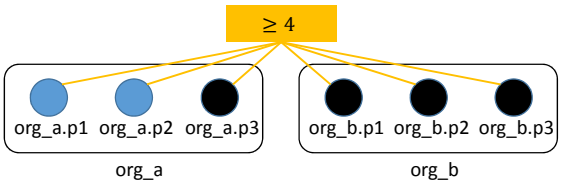

図 6 モデル M2 の組織間信頼の検証で得られた反例

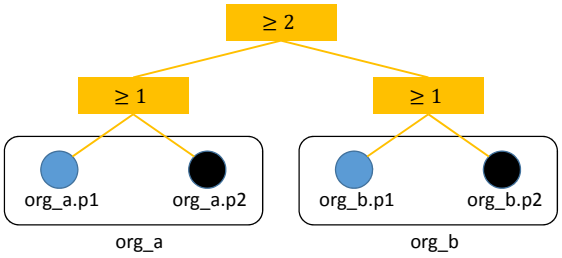

図 7 モデル M3 の安全性検証で得られた反例

検証が NG となったケースでは，SMT ソルバーにより反例 (制約充足の例) が示される．これを図示したものを図 6 および図 7 に示す．

- 図 6．モデル M2 の組織間信頼 (Trust) 検証の反例.
- 図 7．モデル M3 の安全性検証 (Safety) の反例.

なお，図中の青丸は正常なノード，黒丸はビザンチン障害ノード $s = (\text{true}, \text{false})$ である．

4.2 記述量の比較

定義した DSL が効率よく検証対象をモデリングできるかを評価するため，同一のモデルを検証するのに必要なコードを複数の言語で記述して行数 (LOC, lines-of-code) を比較した．比較対象は以下のとおりである．

- 本手法の DSL.
- SMT Lib 2.0 script．SMT ソルバー用のスクリプトを直接作成した場合であり，本手法のモデル変換 (3.8 節) による出力と同等である．
- Python シミュレーター．Python 言語により，全ての障害状態を網羅的にテストするシミュレーターを手作業で実装した．障害状態のモデル化方法は本手法と同じにした．できるだけコードが短くなるよう，エンドースメントポリシーなどはハードコーディングとし，余計な一般化をせず，I/O などの本質的でない部分を含めないようにした．

実装したモデルは図 2 である．また LOC の計測の際に，空行とコメント行は除外してある．比較結果を表 2 に示す．

表 2 各言語による検証モデルのコード行数 (lines-of-code, LOC)

手法	LOC
本手法	24
SMT Lib 2.0 script	119
Python シミュレーター	60

4.3 検証時間の比較

形式手法の一般的な課題として状態数爆発により検証時間が増加し，実用的な時間で計算が終了しないことが挙げられる．この点を評価するため，4.2 節で作成した Python シミュレーターと，本手法の検証時間を比較した．モデルとして図 2 を拡張した次のようなモデルを使用した．

- 組織の数は 3 で固定.
- 1 組織あたりのノード数を $n = 1, 2, 3, \cdots$ (全ノード数を $N = 3n = 3, 6, 9, \cdots$) のように増加させる.
- エンドースメントポリシーは全ノードの過半数 $T(\lfloor N/2 \rfloor + 1, \cdots)$
- 障害は全体で $f = N - (\lfloor N/2 \rfloor + 1)$ まで許す.

これらのモデルは全て安全性および活性の要求を満たし，検証は合格することが事前にわかっている．

表 3 測定環境

CPU	Intel®Core™i5-5300U, 2.30GHz
メモリー	8GB
OS	Windows™7 - 64 bit
Python	version 3.6.1 - 64 bit
Z3	version 4.5.0 - 64 bit

測定結果を図 8 に示す．グラフの Y 軸は片対数プロットとなっている．測定環境は表 3 のとおりである．

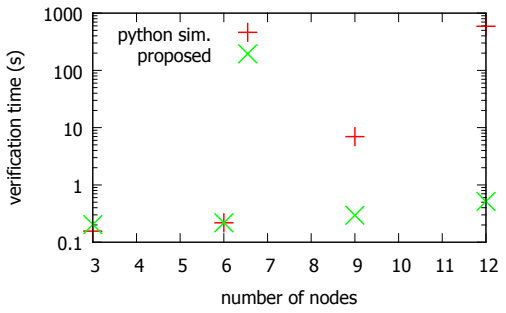

図 8 検証時間の比較.

5. 議論

5.1 実験結果の解釈

表 1 に示すように，4.1 節のモデルの検証結果は全て想定どおりとなり，本手法を用いて安全性，活性，および組織間信頼の検証が可能であることがわかった．

安全性，活性について (図 7) は，モデル M3 のエンドースメントポリシーの 1 段目 (図 3 参照) にあるように，n 個中 1 個の一致を要求するような閾値関数のルールは，クラッシュ障害の場合には障害耐性を与えることができる (いわゆる冗長構成)．しかしビザンチン障害には対応せず，このルールは応答はしているが値が間違っているノードの結果を採用してしまう可能性がある．この結果，2 つの組織間で実行結果が異なればエンドースメントポリシーのルート要素の閾値関数である n 個中 n 個の一致のルールにより，ポリシーを満たすことが出来ず活性を失う．また確率は低いものの両方の組織で誤った値が採用されれば，誤った値のまま合意する可能性があり，安全性も保障されない．

組織間の信頼について (図 6)，モデル M2 を M1 と比較すると，ノード数，エンドースメントポリシー，および障害耐性の設定がそれぞれ 2 倍になっただけであるため，安全性，活性ともに満足されている．にもかかわらず，組織構成の違いにより，モデル M2 においては組織 org_b の同一組織内のノードによる共謀と組織 org_a の単一の (偶然の) 障害が重なる時には，間違った値での合意が行われる可能性があることを例示している．

また効率性を見るために，4.2 節でモデルのサイズを計測した．モデルサイズはシミュレーターや SMT ソルバー

のスクリプトを直接書く場合に比べて小さくなっており，より効率的にモデリングができると考えられる．比較にLOC を用いたが言語やコーディングスタイルでばらつきが生じるため，定量的な比較のためにはトークン数などを用いる必要がある．またサイズだけでなく，設計変更に伴うモデルの変更量なども評価対象になりうる．

DSL の表現力については，3.8 節で述べた通り，エンドースメントポリシー (endorsementPolicy タグ) およびネットワーク構造 (network タグ) については既存の Hyperledger Fabric の仕様を十分に再現している．ただし今後，組織構造の階層化などの拡張に追従する必要はあるだろう．要求 (requirement タグ) においては，組織ごとの障害耐性や共謀などプライベート型ブロックチェーン固有の要求に対応した．一方で過去の経験や想定ユースケースから，以下のような要素のサポートは将来課題である．

- プライバシーや負荷分散の理由により特定のノードや組織にトランザクションを送信しないケース．
- 同一組織内のノードは嘘をつかないと仮定してクラッシュ障害耐性までに限定する．
- 複数組織による共謀．

4.3 節で検証時間を計測した．図 8 に見られるように，シミュレーターによる全パターンテストの検証時間に対し，本手法の検証時間は大幅に小さくなっており，今回試した範囲では全て 1 秒以内であった．これは使用している SMT ソルバーが充足可能性の判定に不要な状態の探索を枝狩りしているためであると考えられる．また現状では要求は比較的単純なため自作シミュレーターには高速化余地があるが，前述の新しい要求に対応すると反例が複雑化すると考えられ，SMT ソルバーが有利であると考えられる．

しかし両手法において徐々に計算時間は増大していく傾向が見られており，計算量は指数オーダーであると考えられる．よって数十ノードを超えるモデルでのスケーラビリティについては今後の課題である．計算量低減の方法としては，現在自由度の高い記述を許しているエンドースメントポリシーに対して実用的な制限 (例えば組織単位での記述のみ許す，など) をかけることが考えられる．

5.2 他のブロックチェーン基盤への適用可能性について

今回，Hyperledger Fabric というブロックチェーン基盤の固有のカスタマイズ可能な合意ルールであるエンドースメントポリシーの検証に特化して検証手法を開発した．実装には XSLT によるモデル変換と SMT ソルバーによる検査を用いており，合意ルール言語の文法などはある程度カスタマイズ可能である．しかしながら，他のブロックチェーン基盤において同様の分散合意ルールのカスタマイズが可能なケースは筆者の知る限り存在しない．

一方で，送金用のパブリック型ブロックチェーン基盤のプロトコルである Ripple Protocol[8] や Stellar Consensus Protocol[9] の場合，PBFT と同様の投票によってビザンチン障害耐性を実現するが，スケーラビリティを向上させるため，全ノード同士ではメッセージ交換を行わず，ノードのサブセットのみで投票を行って局所的な合意の”パッチ”をつくり，これを全体にわたって継ぎ合わせることでグローバルな合意を形成する仕組みである．よってブロックチェーンネットワークに参加するノードは，分散合意においてどのノードからの投票メッセージを受け入れるかというリストを各自で定義することになっている．Stellar consensus protocol の場合，これは quorum slice と呼ばれているが，このリストの定義の仕方によっては合意内容がネットワーク内で分裂する可能性があり，quorum slice が満たすべき性質について議論されている [9]．これはちょうど Hyperledger Fabric におけるエンドースメントポリシーの定義と同様の課題と考えられ，本手法の適用可能性の検討は今後の課題である．

6. 結論

本論文では，ブロックチェーン基盤である Hyperledger Fabric を対象に，分散合意プロトコルの一部としてアプリケーション開発者にカスタマイズ可能な分散合意ルールの検証手法を提案した．本手法は (i) ブロックチェーンネットワークに特徴的なビザンチン障害耐性や組織間の信頼 (共謀への耐性) を要求として検証でき，それらを効率よく記述可能な DSL を定義した，(ii) ブロックチェーン基盤の共通の挙動を検証対象からはずし，さらに探索する障害の状態をワーストケースに限ることでモデルの状態数を削減した，(iii)DSL モデルを SMT ソルバーのスクリプトに変換し形式的に検証することにより，網羅的なテストに比べて高速である．という特徴を持つ．実際に，典型的なモデルを使って検証し問題点を指摘できることを示した．検証時間の比較やモデルの LOC を比較して有効性を確認した．

今後の課題としては反例の可視化や，分散合意ルールの安全な設計パターンの導出がある．またノード数が増加したときのスケーラビリティや Hyperledger Fabric 以外のブロックチェーン基盤への適用可能性も検討したい．

謝辞 IBM Research - Tokyo Blockchain Technology チームでのさまざまな議論に感謝する．

参考文献

[1] Catallini, C.: How blockchain applications will move beyond finance, *Harvard Business Rev*, Vol. 2 (online), available from ⟨https://hbr.org/2017/03/how-

blockchain-applications-will-move-beyond-finance⟩ (2017).

[2] Crosby, M., Pattanayak, P., Verma, S. and Kalyanaraman, V.: Blockchain technology: Beyond bitcoin, *Applied Innovation*, Vol. 2, pp. 6–10 (2016).

[3] Cachin, C. and Vukolic, M.: Blockchain Consensus Protocols in the Wild (Keynote Talk), *31st International Symposium on Distributed Computing (DISC 2017)* (Richa, A. W., ed.), Leibniz International Proceedings in Informatics (LIPIcs), Vol. 91, Dagstuhl, Germany, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 1:1–1:16 (online), DOI: 10.4230/LIPIcs.DISC.2017.1 (2017).

[4] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system, (online), available from ⟨https://bitcoin.org/bitcoin.pdf⟩ (2008).

[5] Castro, M. and Liskov, B.: Practical Byzantine Fault Tolerance and Proactive Recovery, *ACM Trans. Comput. Syst.*, Vol. 20, No. 4, pp. 398–461 (online), DOI: 10.1145/571637.571640 (2002).

[6] Vukolić, M.: Rethinking Permissioned Blockchains, *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, BCC '17, New York, NY, USA, ACM, pp. 3–7 (online), DOI: 10.1145/3055518.3055526 (2017).

[7] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S. W. and Yellick, J.: Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains, *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, ACM, pp. 30:1–30:15 (online), DOI: 10.1145/3190508.3190538 (2018).

[8] Schwartz, D., Youngs, N., Britto, A. et al.: The Ripple protocol consensus algorithm, *Ripple Labs Inc White Paper*, Vol. 5 (2014).

[9] Mazieres, D.: The stellar consensus protocol: A federated model for internet-level consensus, *Stellar Development Foundation* (2015).

[10] Lamport, L., Shostak, R. and Pease, M.: The Byzantine Generals Problem, *ACM Trans. Program. Lang. Syst.*, Vol. 4, No. 3, pp. 382–401 (online), DOI: 10.1145/357172.357176 (1982).

[11] Mernik, M., Heering, J. and Sloane, A. M.: When and How to Develop Domain-specific Languages, *ACM Comput. Surv.*, Vol. 37, No. 4, pp. 316–344 (online), DOI: 10.1145/1118890.1118892 (2005).

[12] Fischer, M. J., Lynch, N. A. and Paterson, M. S.: Impossibility of Distributed Consensus with One Faulty Process, *J. ACM*, Vol. 32, No. 2, pp. 374–382 (online), DOI: 10.1145/3149.214121 (1985).

[13] Charron-Bost, B. and Schiper, A.: The Heard-Of model: computing in distributed systems with benign faults, *Distributed Computing*, Vol. 22, No. 1, pp. 49–71 (online), DOI: 10.1007/s00446-009-0084-6 (2009).

[14] Drăgoi, C., Henzinger, T. A. and Zufferey, D.: PSync: A Partially Synchronous Language for Fault-tolerant Distributed Algorithms, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA, ACM, pp. 400–415 (online), DOI: 10.1145/2837614.2837650 (2016).

[15] Biely, M., Widder, J., Charron-Bost, B., Gaillard, A., Hutle, M. and Schiper, A.: Tolerating Corrupted Communication, *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, New York, NY, USA, ACM, pp. 244–253 (online), DOI: 10.1145/1281100.1281136 (2007).

[16] Charron-Bost, B., Debrat, H. and Merz, S.: Formal Verification of Consensus Algorithms Tolerating Malicious Faults, *Stabilization, Safety, and Security of Distributed Systems* (Défago, X., Petit, F. and Villain, V., eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 120–134 (online), available from ⟨https://link.springer.com/chapter/10.1007%2F978-3-642-24550-3_11⟩ (2011).

[17] John, A., Konnov, I., Schmid, U., Veith, H. and Widder, J.: Parameterized model checking of fault-tolerant distributed algorithms by abstraction, *2013 Formal Methods in Computer-Aided Design*, pp. 201–209 (online), DOI: 10.1109/FMCAD.2013.6679411 (2013).

[18] Kwiatkowska, M., Norman, G. and Parker, D.: Probabilistic Model Checking in Practice: Case Studies with PRISM, *SIGMETRICS Perform. Eval. Rev.*, Vol. 32, No. 4, pp. 16–21 (online), DOI: 10.1145/1059816.1059820 (2005).

[19] Yin, J., Martin, J.-P., Venkataramani, A., Alvisi, L. and Dahlin, M.: Separating Agreement from Execution for Byzantine Fault Tolerant Services, *SIGOPS Oper. Syst. Rev.*, Vol. 37, No. 5, pp. 253–267 (online), DOI: 10.1145/1165389.945470 (2003).

[20] W3C: XSL Transformations (XSLT) Version 2.0, (online), available from ⟨https://www.w3.org/TR/xslt20/⟩ (2007).

[21] The Apache Software Foundation: Xalan-Java, (online), available from ⟨http://xml.apache.org/xalan-j/⟩ (2006).

[22] Microsoft Corporation: Z3 Theorem Prover, (online), available from ⟨https://github.com/Z3Prover/z3⟩ (2018).

[23] Barrett, C., Stump, A. and Tinelli, C.: The SMT-LIB Standard: Version 2.0 (2012).

[24] W3C: W3C XML Schema Definition Language (XSD) (2012).

付　　録

A.1 モデルの例

DSL を用いたモデル (図 4) の例を以下に示す.

```
<?xml version="1.0" encoding="UTF-8" ?>
<ep-checker>
<endorsementPolicy>
 <t threshold="2">
  <t threshold="1" >
   <peer ref="org_a.p1" />
   <peer ref="org_a.p2" />
  </t>
  <t threshold="1" >
   <peer ref="org_b.p1" />
   <peer ref="org_b.p2" />
  </t>
 </t>
</endorsementPolicy>
```

```
<network>
  <org id="org_a">
    <peer id="org_a.p1"/>
    <peer id="org_a.p2"/>
  </org>
  <org id="org_b">
    <peer id="org_b.p1"/>
    <peer id="org_b.p2"/>
  </org>
</network>
<requirement>
  <faultTolerance>
    <org ref="org_a" num="1" />
    <org ref="org_b" num="1" />
  </faultTolerance>
</requirement>
</ep-checker>
```

A.2 DSL の定義

今回定義した DSL を XML Schema[24] で定義する．紙面の都合上要求中の一部タグは省略した．

```
<?xml version="1.0"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="ep-checker" >
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="endorsementPolicy"/>
    <xsd:element ref="network"/>
    <xsd:element ref="requirement"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="endorsementPolicy">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="t">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:choice maxOccurs="unbounded">
        <xsd:element ref="t" />
        <xsd:element ref="peer" />
       </xsd:choice>
      </xsd:sequence>
      <xsd:attribute
        name="threshold" type="xsd:int"/>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="network" >
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="org" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="requirement">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="faultTolerance"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="peer">
  <xsd:complexType>
   <xsd:choice>
    <xsd:attribute name="ref" type="xsd:IDREF"/>
    <xsd:attribute name="id" type="xsd:ID"/>
   </xsd:choice>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="org" >
  <xsd:complexType>
   <xsd:choice>
    <xsd:group>
     <xsd:attribute name="id" type="xsd:ID"/>
     <xsd:sequence>
      <xsd:element ref="peer"
        maxOccurs="unbounded"/>
     </xsd:sequence>
    </xsd:group>
    <xsd:group>
     <xsd:attribute name="ref" type="xsd:IDREF"/>
     <xsd:attribute name="num" type="xsd:int"/>
    </xsd:group>
   </xsd:choice>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="faultTolerance">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="org" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```