第　2　部　第2章

スマートコントラクトの安全性と公平性の保証

大矢野　潤

目　次

１．はじめに

インターネットはその発達過程において新聞やTV、友人関係など様々な「もの」を電子化しながら世界的な情報社会基盤としての地位を確立してきた。しかし、私達の日常生活に欠かせないものであるにもかかわらず、これまで電子化できていないものがあった。意外にも、それは「お金」である。

現実世界に存在する「もの」を電子化するためには、そのものの持つ性質をプログラムとして表現する必要がある。「お金」の持つ性質については、ここでは「流通する」「偽造できない」「匿名性が保たれる」「何度も使用できない」などとしておく。ビットコイン以前にも、偽造できないこと（正確には「しにくい」こと）や匿名性については暗号技術により実現されていたが、何度も使用できないこと、いわゆる二重支払いができないことを保証するためには、中央集権型の決済システムを仮定する必要があった。私たちが日常生活で使用している銀行のオンラインシステムや電子マネーなどは、お金に相当する電子データを適切な決済システムのもとで管理している。この決済システムの外にある主体への送金は、送受信に関与する主体間の信頼関係の構築から始める必要があり、結果的に高額な送金費用が発生することや､そもそも送金自体ができなくなることもある｡コンピュータ上の貨幣データはその複製が容易であるため、支払いの正当性を保証できない環境においては、一つの貨幣データを複数回の支払いに使用可能になってしまう。何度も使えるお金は通貨としての価値を保持できない。

非中央集権自律分散環境における二重支払い防止策は2009年のビットコインの誕生を待つ必要があった。ビットコインに代表される仮想通貨は、お金という商取引に欠かすことのできない決裁システムをインターネット上に構築するものであり、新たなビジネスの創出という重要な役割を担うことが期待されている。しかし、仮想通貨が保証する安全性はわれわれが考える安全性とは大きな隔たりがあることを認識しなくてはならない。例えば、あるPCに保存されている電子財布（ウォレット）やビジネスを具現化したプログラムがハッキングされた結果として保有するコインが他の主体に送金されたとしても、その送金手続きが妥当であれば、流出してしまったコインを回収することは事実上不可能である。

一般にコンピュータプログラムの不具合を見つけること、特に並列プログラムのデバッグは非常に困難であり、一般のユーザがその作業を担うことを期待することはできない。本報告では、形式検証技術をスマートコントラクトに応用した事例に基づき、そこで仮定

される計算科学的（形式的）正しさと日常生活における商取引における正しさとのギャップについて論じていく。具体的には、スマートコントラクト記述言語Solidityのサイトで紹介されている「Safe Remote Purchase」の実装例［13］をモデル記述言語Promelaに変換する。Promelaによる記述から構成されたモデルをモデル検査器SPINによって解析し、その過程において浮かび上がってくる形式的な正しさと商取引における正しさとの違いについて論じていく。

2．ブロックチェーンとスマートコントラクト

2－1　仮想通貨とブロックチェーン

一万円札には一万円の価値があることを「日本国」が保証しているが、インターネットは特定の国に属さない情報基盤であるため、特定の国家のようなオーソリティを仮定することができない。

2009年、Satoshi Nakamotoを名乗る人物によって論文“Bitcoin: A Peer-to-Peer Electronic Cash System”［8］が投稿され、非中央集権システム上でビットコイン（Bitcoin）とよばれる仮想通貨を実現するためのアイデアが提案された。ビットコインにおけるコインとは、電子コインが発行されてから現在までのコインの支払いの履歴そのものであるとされている。電子署名技術を用いたデータの改ざん防止のしくみはビットコイン以前にも実用化されていたが、それだけでは「お金」として十分と言えるものではなかった。ビットコインが解決した問題とは、コインの所有者が同一のコインを使用して複数の通信主体に対して支払いをする行為、いわゆる二重支払いをプルーフオブワーク（Proof Of Work）という仕組みを利用して防ぐことであった。支払い履歴（の集合）はプルーフオブワークによって繋げられ、ブロックチェーンを形成する。二重支払いの実行は、結果としてブロックチェーンの分岐を生成するが、分岐したチェーンの長い方をネットワークが正当なものとすることにより、どちらか一方の支払いだけが正当なものとなる。

「分散環境において、各々の通信主体が意図的であるかどうかにかかわらず間違った情報を伝達する可能性がある場合でも、ネットワーク全体として正しい合意を形成できるか?」という問題はビザンチン将軍問題と呼ばれ、1982年にLeslie Lamportが定式化した［6］。ブロックチェーンはビザンチン将軍問題に対する一つの現実解を提示しているということができる。

２－２　スマートコントラクト

スマートコントラクト（smart contract）とは、契約の締結や実行を自動的に行うコンピュータプログラムやトランザクションのこととされている。通貨の支払いはトランザクションとして自動的に実行され、そのトランザクションの履歴はブロックチェーンに追加されていく。前述の通り、ブロックチェーンに書き込まれたトランザクションログの正当性はネットワーク全体の合意事項となるため、ある主体から別の主体へのコインが送信されコイン所有者が移転したこと、同一のコインが他の支払いにも使われていないことが保証される。

複数の異なるビジネスロジックをシームレスに統合し、企業間のコラボレーションを促進しようとする動きは2000年台前半から盛んに行われてきた。代表的なものに、標準化団体 World Wide Web Consortium（W3C, https://www.w3.org/）による Web Services Choreography、OASIS（https://www.oasis-open.org/）による WS-BPEL などが知られている。例えば旅行業は企業間コラボレーションの代表例といってよい。旅行は宿泊施設、交通手段、観光施設など複数の「旅行素材」を組み合わせた国際的なサービスであり、インターネットを介してユーザからの予約を受け付ける業態（Online Travel Agency）はすでに広く利用されている。

今後、ブロックチェーンを基盤としたスマートコントラクト技術の確立と普及により、異業種間のビジネスの統合化、自動化が進むことによる新たなサービスの出現が期待される。しかし、悪意のあるユーザにビジネスロジックの欠陥をつかれ、業務妨害や第三者への不正な送金が発生した場合には大きな被害に発展する可能性がある。このため、具体的なビジネスをブロックチェーン上に実装する場合には、リリース前にプログラムを十分に検査する必要がある。

２－３　イーサリアム

ビットコインの誕生以降、さまざまな仮想通貨やブロックチェーンの亜種が発生している。これまでの議論からもわかるとおり、ブロックチェーンはプログラミング、暗号学、分散ネットワーク、セキュリティ、確率・統計学など様々な技術を応用して構築されており、すべての技術を理解した上でビジネスモデルを構築することは現実的ではない。適切な開発プラットフォームの選択は技術的なハードルを平準化させることを意味しており、プロジェクトの成功への近道でもある。

本研究では、開発プラットフォームとしてイーサリアム（Ethereum: https://ethereum.org/en/）を選択することにした。イーサリアムのもつスマートコントラクト開発環境、

コミュニティの大きさとそこから得られる開発関連情報が他のブロックチェーンに比較して充実していると判断したからである。イーサリアムの形式的意味は厳密に定義されており［14］、その意味論に則った開発言語と実装例が多数存在している。とりわけ Solidity というスマートコントラクト開発言語は Web プログラミングで利用されることの多い JavaScript に類似しており、Web サービス開発者にとって言語学習のハードルが非常に低い。開発者による教科書［1］も良書であり、他のプラットフォームと比較して十分なアドバンテージがあると判断した。

3.「正しさ」の検証

安心してスマートコントラクトを利用するためには、そのスマートコントラクトが「正しく」、すなわち設計者が意図した通りに振る舞わなければならない。前述の通り、仮想通貨やブロックチェーンは現代暗号技術によって支えられている。ここでは、用いられている暗号強度が十分であること、ブロックチェーンはビザンチン将軍問題、すなわち 51% ルールに従っている限り改ざんできないものとして議論を進めていく。スマートコントラクトはその利用手続き（プロトコル）に従って実行されるが、一般に分散環境におけるプロトコルの検証は非常に困難である。

3－1　プロトコルの正しさ

暗号を使用して他の主体と秘匿性を保った通信をする場合には、暗号化されたメッセージを受信する主体が予めそのメッセージを解読するための鍵を保持している必要がある。この鍵の受け渡しの手続きに脆弱性が潜んでおり、秘密にされるべき鍵が漏洩する可能性がある場合には、十分な強度の暗号を使用したとしてもそこでの通信が安全に行われるとは限らない。以下、プロトコルの弱点を狙われた代表的な例について紹介する。

3－1－1　Needham-Schroeder 公開鍵プロトコル

安全でないネットワーク上で二人の参加者が相互に認証するためのプロトコルとして、Roger Needham と Michael Schroeder によって発表された Needham-Schroeder 公開鍵プロトコルがある［9］。このプロトコルは発表以来 17 年間安全であると信じられていたが、1995 年に Gavin Lowe によってその脆弱性が指摘された［7］。二人の参加者のうち一人が「うっかり」侵略者に話しかけた結果、二者間の秘密が第三者の侵略者に漏洩してしまう可能性が指摘されたのである。このように、暗号システムの専門家が提案し、世界

中の検証者がその脆弱性を発見できないまま長期間使用し続けていたプロトコルの脆弱性がある日突然暴かれることがある。

３－１－２　The DAO 事件

The DAO とはイーサリアムブロックチェーン上で実現された、非中央集権分散自律組織の一つである。The DAO は、2016 年５月にトークンセールを通じてクラウドファンディングされた歴史上最大のクラウドファンディングキャンペーンであったが、2016 年６月にコードの脆弱性を悪用され、DAO の資金の３分の１（当時の相場で約 50 億円）を別のアカウントに吸い上げられてしまった［11,15］。これは、記述言語である Solidity の無名の支払い関数が DAO コード内で暗黙のうちに起動されるという、プログラムの振る舞いとビジネス設計者の意図した振る舞いとの差異を悪用されたものである。

これらの事例からも分かるように、それぞれのトランザクションが正しくてもコントラクト全体が正しく行われるとは限らず、その脆弱性をあらかじめ指摘することは非常に困難である。特に支払いを含むスマートコントラクトがその設計者や利用者の意図通りに動作することが保証されないということであれば、そのコントラクトをビジネスに用いることはできない。

３－２　スマートコントラクトの持つ「性質」

スマートコントラクトが意図した通りに振る舞うかということの判定は非常に困難であるため、スマートコントラクトの「コード」と、期待される性質に対する「記述」を入力すればその妥当性を自動的に判定し、結果を返してくれるような仕組みが望まれる。

計算科学の分野では、プログラムの満たす性質を表現するための論理体系として時相論理（Temporal Logic）が古くから研究されている。通常、数学や論理学は静的な世界、すなわち、一旦ある命題が真であるということが判定されればその真偽値が変化することがない世界を仮定しており、そこでの論証には命題論理や述語論理などが用いられることが多い。しかし、手続き型言語によって記述されたプログラムの実行は、複数の変数の値の集合（状態）を更新し続けることでプログラムの目的を達成する。例えば、ある変数 X によって「預金の残高」が表現されていることを考えてみよう。この時、アクションの前後における状態の集合（世界）の変化によって預金に対する操作が表現されることになる。その結果、「X の残高が十分にある」などといった命題は、引き出しの前の世界と後の世界において同じ真偽値を持つとは限らない。このような、アクションの実行の「前後」に

おける状態の変化を考慮に入れた論理体系の一つに時相論理がある。

3－3　安全性と活性、公平性

時相論理での性質は、安全性（safety property）と活性（liveness property）の組み合わせで記述できることが知られている。非形式的には、安全性とは「悪いことが起こらない（“bad things” do not happen）」こと、活性とは「良いことがいつか起こる（“good things” do happen）」とされている［5］。安全性の証明は不変性（invariance）を示すことで、活性のプログラムの前後の世界の関係が整礎（well-founded）であるということを示すことで得られる。時相論理における性質が安全性と活性の組み合わせで記述できるのであれば、その証明手続きは不変性の証明と整礎性の証明アルゴリズムの合成により実現可能である。

Alpern と Schneider は論文 “Defining liveness”［2］において、活性の形式的な定義を提案するとともに、安全性と活性に対する位相的な特徴づけ、すなわち、安全性がちょうど閉集合（closed set）となる位相空間が存在し、その位相空間において活性とは稠密集合（dense set）となることを示している。さらに、任意の位相に関する性質は安全性と活性の積集合で表現可能であることを証明している。

時相論理については、その表現力の差により LTL（Linear Time Logic）や CTL（Computational Tree Logic）、CTL*、様相 μ 計算などが知られている。通常、論理体系の持つ表現力が強くなればその証明手続きは複雑になるため、目的に応じた論理体系の選択が必要である。

本論文で利用したモデル検査器 SPIN（http://spinroot.com/spin/whatispin.html）はプロセスの性質を記述する論理体系として LTL を採用している。厳密には、ユーザが与えた LTL を等価な Büchi オートマトンに変換して使用している。LTL について詳細を述べるのは本報告書の範囲を超えているが、大雑把に言うと「□ P」で「いつも（always）P」を「◇ P」で「いつか（eventually）P」を表している。この□と◇を組み合わせることにより「□◇ P: infinitely often P」、「◇□ P: eventually forever P」などの豊かな性質を表現することが可能になる。その他にも「□◇ P →□◇ Q：強公平性（strong fairness）」や「◇□ P → □◇ Q：弱公平性（weak fairness）」など、本報告におけるキーワードである「公平性」を表現することができる。

3－4　モデル検査

ここでは、「プログラムが、ユーザが意図したとおりに動作すること」ということをも

う少し形式的に述べてみよう。まず、「ユーザの意図」をΦ、「プログラムとその状態集合」をプログラムのモデルM、「モデルMがΦを充足する」ということを「M ⊨ Φ」と記述することにする。モデルMと性質Φを入力とし自動的にM ⊨ Φかどうかを判定するコンピュータプログラム、すなわちモデル検査器に関する研究が世界中でおこなわれ、急速に普及している。モデル検査器にはさまざまな種類が存在するが、本研究では、SPINを用いることとする。モデル検査全般については代表的な教科書“Model Checking”［３］を参照されたい。

SPINはG. Holzmann博士によって開発され、2001年にACM Software System Awardを受賞するなど評価の高いソフトウェアである。世界中に多くの利用者が存在しており、関連研究の論文やマニュアル、教科書［4,16］、事例集など学習に必要な教材を比較的容易に入手することができる。本論文で参照した公開鍵認証プロトコルに対する攻撃［７］に関するSPINを用いた追試の論文［10］も公開されている。さらに、SPINの入力を記述する言語Promelaの文法は本研究で解析するSolidityと類似しており、Solidityのプログラムを解析し、Promelaの記述に変換する作業が容易になることが期待できる。Promelaについての簡潔な説明は文献［12］で得ることができる。

SPINは性質Φとして時相論理式LTL、モデルMとしてPromelaで記述されたプログラムからBüchiオートマトンと呼ばれる、アルファベットの無限列からなる言語を受理する有限状態機械に変換する。Φ、Mに対応するオートマトンが受理する言語をそれぞれL（Φ）、L（M）とする時、L（M）⊆L（Φ）であればMの振る舞いが性質Φをもつ振る舞いに含まれる、すなわちMが性質Φをもつと言える。集合論的にL（M）⊆L（Φ）であることは∀ t ∈ L（M）→ t ∈ L（Φ）、つまり、L（M）のすべての要素はL（Φ）の要素になっていることを示すことで証明されるが、一般にL（M）は無限集合となるため虱潰し的に調べることができない。このため、実際には、「性質Φをもたない」ということを意味する論理式「¬ Φ」に対応するオートマトンを作成しL（M）∩L（¬ Φ）が空集合であるかどうかを判定する。この集合が空でない場合には、そのなかから適当に選んだ要素が性質Φを満たさないモデルMの具体的な振る舞い例、すなわち反例（counter example）として採用される。

３－４－１　モデル検査の利点と欠点

本報告書で紹介したモデル検査以外にも、証明論に基づくシステム（証明系）を用いてプログラムを調べる方法もある。証明系の利用は通常の数学の証明手続きに沿った形で行われるため、数学の無限に関する言明をプロセスの無限の振る舞いに適用することができ

るが、解析者に求められる数学的・証明論的ハードルは高くなる傾向がある。これに対してモデル検査の場合は、モデル構築作業がプログラミングの作業に近いこと、モデル構築後の検証がほぼ自動的に行われるなどの利点がある。さらに、プログラムから得られたモデルに誤りが潜んでいないことが証明できるということだけでなく、誤りが検知されたときに報告される反例の解析が設計者のプロトコルに対する理解を深めることに役立つことも強調しておく。

通常のプログラムはチューリング等価であり、チューリング等価なプログラムの性質は一般に決定不能であることから、解析プロセスの完全な自動化は原理的に不可能である。モデル検査は有限のモデルを構築し網羅的に調査する手法であるが、一般にプログラムは無限の状態をもつため、有限化の段階でエラーが混入する可能性を否定できない。また、非常に簡単な有限プロセスであったとしても、そのモデルの状態数は爆発的に増加する傾向がある。状態数爆発は実行コンピュータに深刻なメモリ不足を引き起こし、また、モデル探査が現実的な時間内に終了しないなどの直接的な原因となる。これらの理由から、状態数爆発を効率的に回避することがモデル検査の主たる課題とされている。本調査でも状態数爆発を回避するためにもとのプログラムに制限を加えている。有限化手続きと同様に、状態数を抑えるために加えた操作がモデル検査の信頼性に著しい影響を与える可能性がある。

4．モデル検査の実例

本研究では、Solidity によって記述された安全な遠隔購入手続き（Safe Remote Purchase）に対しモデル検査器 SPIN を用いて解析した。

Solidity で記述されたプログラムを Promela の記述に自動的に変換し、モデル検査をする方法はいくつか提案されている。しかし、プログラマがモデル検査によって得られる反例を通じてプログラムの脆弱性に関する知見を深めていくことはモデル検査の重要な意義の一つである。プロトコルに対する安全性の検証が失敗し反例が示された場合、検証者はその反例を分析する。反例は、プロトコル設計者が意図していなかった具体的な攻撃手順であり、この情報をもとにプロトコルが解決すべき問題を明確化し、改善作業に役立てていくことができるからである。さらに、モデル検査では解析する状態数が爆発的に増加することが頻繁に起こる。自動生成された Promela ファイルを SPIN で解析する際に発生した状態数爆発に、ソースファイルの Solidity プログラムを変更して対応することは困難であるとともに本末転倒でもある。

以下、Solidity プログラムとして記述された遠隔購入手続きを「手で」Promela 言語に

翻訳した作業とそれを通じて得られた知見について報告する。

４－１　遠隔購入手続き

まず、Solidity ドキュメントに紹介されている遠隔購入手続き（Safe Remote Purchase）について簡単に説明する。

商品を遠隔購入するということは、買い手（Buyer）は売り手（Seller）から商品を受け取り、売り手はその見返りに代金を受け取ることで成立する。ここで、購入手続きに参加する売り手（Seller）と買い手間に信頼関係が成立してなくてはならない。なぜなら、信用できない買い手に対して先に商品を発送してしまった場合には代金が回収できず商品を騙し取られる可能性があり、逆に買い手が先に代金を送ってしまった場合には商品を受け取れないまま代金だけを騙し取られる可能性が発生する。結果的に、売り手も買い手も先にアクションを起こすことはなく、全体の遠隔購入手続きは双方が動けない、ロックがかかった状況になる。

この問題に対して、Ethereum ネットワークに対して保証金を預託する、いわゆる第三者預託取引（Escrow）によって解決を図るアルゴリズムが公開されている。このアルゴリズムを、通信主体と送金金額を具体化して説明しよう。

まず、A に 500 円の商品を販売する意思があり、B にはその商品を同じく 500 円で購入する意思があるものとする。この時、A、B は直接代金のやりとりをするのではなく、「仲介者」を介して売買手続きを行う。ただし、ここでの仲介者とは、Ethereum ブロックチェーンに登録された改ざんできないプロセスであると仮定すると、売買のプロセスは以下のように実行される。

１．A（売り手）が価格の２倍の 1000 円を仲介者に送金する［預託金＝ 1000 円］
　ⅰ．A が仲介者に取引中止を知らせる［預託金＝ 1000 円］
　ⅱ．仲介者は A に預託金を返金し取引を終了させる［預託金＝ 0 円］
２．B（買い手）が同じく 1000 円を仲介者に送金し、購買の意思を示す［預託金 =2000 円］
３．A は B に品物を送付する［預託金 =2000 円］
４．B は品物を受け取ったことを仲介者に知らせる［預託金 =2000 円］
５．仲介者は預託金から B に 500 円を返金する［預託金 =1500 円］
６．仲介者は預託金から A に 1500 円を返金する［預託金 =0 円］

このとき、売り手と買い手の双方が状況を解決しなければ、彼らが前もって預託した代金が仲介者により永久にロックされてしまう。売り手が商品を発送しなければその倍の金額がネットワークにより凍結され、逆に買い手が商品をだまし取ろうとして商品の到着をネットワークに告げなければ、商品の倍の金額が凍結されてしまう。すなわち、いずれかが代金や商品をだまし取ろうとしても、お互いに定価の倍の金額を預託しているため詐欺行為自体が割に合わない状況を作りだすことで遠隔取引を安全にすることができるとしている。

4－2　Safe Remote Purchase: Solidity

あらためて Solidity について簡単に説明する。Solidity とはブロックチェーンプラットフォーム上にスマートコントラクトを実装するためのオブジェクト指向言語である。2014 年に Gavin Woo により提案され、Christian Reitwiessner、Alex Beregszaszi らによって実装された。利用者は急速に拡大し、現在では Ethereum 上で最も使用されている言語であるとされている。本報告書執筆時における Solidity のバージョンは Release v0.8.0 となっている。Solidity 言語の詳細については公式ドキュメント（https://solidity-jp.readthedocs.io/ja/latest/）や教科書等［1］を参考にしてほしい。

次に、Solidity によって記述された Safe Remote Purchase のコードを紹介する。コードはすべてドキュメントサイトからの引用であるが、オリジナルのコメントは削除しており、代わりに簡単な説明と遠隔購入手続きの説明に付け加えた参照番号を付記している。

```
pragma solidity >=0.7.0 <0.9.0;
contract Purchase {
    uint public value;
    address payable public seller;
    address payable public buyer;

    // オートマトン状態集合の定義
    enum State { Created, Locked, Release, Inactive }
    State public state;

    // 関数が呼び出されるための条件（ガード）の定義
    modifier condition(bool _condition) {
        require(_condition);
```

```
    _;
}
modifier onlyBuyer() {
    require(
        msg.sender == buyer, // 関数呼び出しを買い手に制限
        "Only buyer can call this."
    );
    _;
}
modifier onlySeller() {
    require(
        msg.sender == seller, // 関数呼び出しを売り手に制限
        "Only seller can call this."
    );
    _;
}
modifier inState(State _state) {
    require(
        state == _state,// 指定された状態においてのみ実行可能
        "Invalid state."
    );
    _;
}

// イベントの宣言
event Aborted();
event PurchaseConfirmed();
event ItemReceived();
event SellerRefunded();

// (1) 売り手からの預託金受付け。取引生成
constructor()
```

```
    payable // 入金可能（payable）
{
    seller = payable（msg.sender）;
    value = msg.value / 2;
    require（（2 * value） == msg.value, "Value has to be even."）;
    // 入金金額（= 預託金額）の半額が代金となるため、入金金額は偶数
}

// (i) 預託取引の中止
function abort（）
    public
    onlySeller // 取引中止は売り手に限定
    inState（State.Created）// タイミングは取引開始前
{
    emit Aborted（）;//Aborted イベントの発出
    state = State.Inactive;//（ii）取引終了
    seller.transfer（address（this）.balance）;
    // 返金手続き
}

//（2）買い手からの預託金受付。購入の意思確認
function confirmPurchase（）
    public
    inState（State.Created）// タイミングは取引開始前
    condition（msg.value ==（2 * value））// 預託金額は代金の2倍
    payable
{
    emit PurchaseConfirmed（）;//PurchaseConfirmed イベントの発出
    buyer = payable（msg.sender）;
    state = State.Locked;// 品物が到着するまで預託金を凍結
}
```

```
    //（4）品物の到着確認、（5）買い手へ返金
    function confirmReceived()
        public
        onlyBuyer// 買い手だけが到着確認可能
        inState(State.Locked) // タイミングは預託金凍結時
    {
        emit ItemReceived();//ItemReceived イベントの発出
        state = State.Release;// 預託金の凍結解除の状態に遷移
        buyer.transfer(value);// 買い手に預託金の半額（= 代金）を返金
    }

    //（6）売り手へ返金
    function refundSeller()
        public
        onlySeller// 返金要求は買い手に限定
        inState(State.Release) // タイミングは預託金凍結解除時
    {
        emit SellerRefunded();//SellerRefunded イベントの発出
        // 取引が終了した状態に遷移した場合には、refundSeller() が
        // 複数回呼ばれる可能性があるので注意
        state = State.Inactive;// 取引終了の状態に遷移
        seller.transfer(3 * value);// 預託金＋代金を返金
  }
}
```

４－３　Safe Remote Purchase: Promela

４－３－１　Promela

Promela はその文法が C 言語や JavaScript、Solidity によく似ているため、その逐次的な意味については理解が容易であろう。ここでは、並列プロセス記述に関する特徴についてのみ簡単に説明することにする。

まず、プロセスは proctype で宣言され、複数のプロセスを任意のタイミングでの並列実行が可能である。特に、Promela 起動時にデフォルトで起動される init と呼ばれる特殊

なプロセスによりモデルの初期化、他のプロセスの起動を行うことが多い。

次に、プロセス間の通信として、入出力チャネルを装備している。「入力チャネル？変数の並び」で入力チャネルを通してメッセージを受け取り、受け取ったメッセージを変数に格納することができる。ここで、変数の代わりに定数を用いた場合は、その場所に入力されたデータが指定された定数と等しいとする制約として解釈される。これに対し、「出力チャネル！データの並び」で出力チャネルを通してメッセージを出力することができる。また、チャネル自身もデータとして送受信可能であるため、動的に通信相手を変更することもできる。

制御構造としては、「逐次実行」「選択実行」「繰り返し」が用意されている。

逐次実行

```
statement1; statement2
```

は文 statement1 の次に文 statement2 を実行するということを表している。ここで、statement1 が bool 値をとる場合にはその値が true になるまで statement2 は実行されない。このように、statement1 を statement2 のガードとして使用する場合には、

```
statement1 -> statement2
```

と記す場合もある。

選択実行

選択実行は次のように表現される。

```
if
  :: statementA -> statement1;
  :: statementB -> statement2;
fi
```

このブロックは、statementA、statementB の評価値によって statement1、statement2 のいずれかが選択され、実行されることを表現している。ここで、statementA、statementB 双方が true と解釈された場合には statement1、statement2 の選択は非決定的に行われる。逆に両方が false の場合には if 文全体の実行がブロックされる。

反復実行

反復実行は次のように表現される。

```
do
  :: statementA -> statement1;
  :: statementB -> statement2;
od
```

基本的な動作は選択実行と同様であるが、statement1 もしくは statement2 が終了したあとに、再度この文全体が実行される。このため、意味は次の文と同様である。

```
some_label:
if
  :: statementA -> statement1; goto some_label
  :: statementB -> statement2; goto some_label
fi
```

アトミック宣言

全体の状態数を抑制するために atomic を用いることがある。たとえば、プロセス A が文A1; A2から、プロセス B が文B1からなる時、全体のシステムの振る舞いとしては、「A1, A2, B 1」、「A1, B1, A2」、「B1, A1, A2」の３パターンを考慮する必要がある。ここで、プロセス A をatomic{A1; A2}と宣言することにより、A1 と A2 が「かたまり」で実行されることを主張する。その結果、B 1 は A 1 と A 2 の間に実行されることはなくなるため、振る舞いのパターンは「A1, A2, B1」、「B1, A1, A2」の２種類のみ考慮すればよい。複数のプロセスが複数の文からなるとき、その振る舞いのパターンは組み合わせ論的に増大するため、他のプロセスの影響を受けない文は atomic 宣言でひとまとめにすることにより、モデル検査の効率を向上させることが期待される。

Promela のこの他の機能ついては、SPIN 公式サイトや教科書を参照されたい。

4－3－2　Solidity から Promela への変換

Solidity で記述された遠隔購入取引の大まかな振る舞いは、オートマトンの一種である Mealy 機械として解釈することができる。大まかにいうと、Mealy 機械とは入出力付きオートマトンである。本論文では、状態遷移がA $\overset{a/b}{\rightarrow}$ Bのように記述された場合、「機械が状態 A にあるときにアルファベット a が入力された時場合、b を出力して B の状態に遷移する可能性がある」ことを表現することにする。

コントラクト Purchase のメンバシップ関数 input() の呼び出しとイベント output（emit output）の発出のペアを「入力 input、出力 put の遷移」と解釈することにより、Solidity の記述を Mealy 機械として解釈すると Figure1 のような図式を得ることができる。

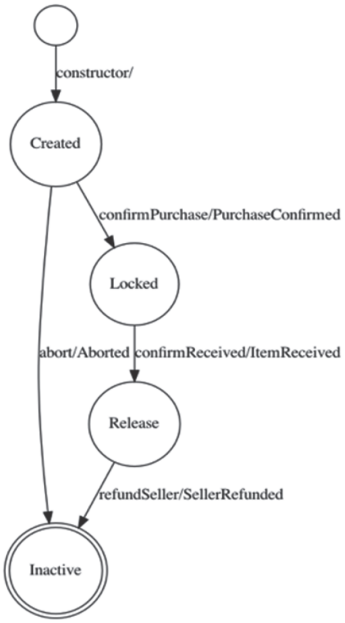

Figure 1 : 預託プロセス

この図式を元に作成した遠隔取引の Promela 記述は次のようになった。

```
// Solidity と Promela の型の違いを吸収
#define address chan
#define uint byte
#define State mtype

#define NULL 0   // 意味のないメッセージを表現
#define SYNCH 12 // 通信チャネルのバッファ数（要調整）
#define N_PRO 7  // 検証プログラム数（要調整）

// 状態の集合の定義
mtype {Created, Locked, Release, Inactive}
// Purchase の入力アルファベット集合
mtype {constructor, abort, confirmPurchase, confirmReceived,
refundSeller}
```

```
// Purchase の出力アルファベット集合
mtype {Aborted, PurchaseConfirmed, ItemReceived, SellerRefunded}

// ブロックチェーン。単に広域変数として定義
typedef Ledger {uint balance, value; bool done=true}
Ledger ledger[N_PRO+1];// 台帳
// 預託プロセスの通信チャネルを公開
chan purchase = [SYNCH] of  {mtype, address, uint};

// Solidity の modifier の言い換え
#define onlyBuyer eval(buyer)
#define onlySeller eval(seller)
#define inState(_state) (state == _state)

inline payable(from, to, value){
  // Solidity の payable を「from から to へ金額 value を送金」として表現
  atomic{
    ledger[from].balance = ledger[from].balance - msg_value;
    ledger[to].balance = ledger[to].balance + msg_value;
  }
}
// 預託プロセス本体
proctype Purchase() {
  address msg_sender; uint msg_value; // 入力メッセージ格納バッファ
  address seller; address buyer; // 出力メッセージ格納バッファ
  uint value;
  State state;
  atomic{
    ledger[purchase].balance = 0;
    ledger[purchase].value = 0;
  }
```

```
progress_await_seller:
  do
    :: purchase??constructor(msg_sender, msg_value)
       ->  value = msg_value/2
       if
           :: (msg_value != 2*value) -> goto progress_await_seller
           // 預託金額は代金の2倍
           :: else -> atomic{
              state = Created;
              payable(msg_sender, purchase, msg_value);
              seller = msg_sender; // 売り手は msg_sender に決定
              break
            }
       fi
  od

  do
    :: inState(Created) ->
progress_await_buyer:
       if
           :: purchase??abort(onlySeller, _)  ->
               atomic{
                state = Inactive;
                seller!Aborted(purchase, ledger[purchase].balance);
                goto end_purchase
              }
           :: purchase??confirmPurchase(msg_sender, msg_value) ->
              if
               :: (msg_value != (2 * value))  ->
                  goto progress_await_buyer
               :: else -> atomic{
                  state = Locked;
```

```
                payable(msg_sender, purchase, msg_value);
                buyer = msg_sender;
                buyer!PurchaseConfirmed(purchase,NULL)
                // 買い手が msg_sender に決定
               }
            fi
       fi
    :: inState(Locked) ->
       if
         :: purchase??confirmReceived(onlyBuyer, _) ->
            atomic{
             state = Release;
             buyer!ItemReceived(purchase, value);
             // 品物が受領された
         }
       fi
     :: inState(Release) ->
       if
              :: purchase??refundSeller(onlySeller, _) ->
                  atomic{
                    state = Inactive;
                    seller!SellerRefunded(purchase, 3 * value);
                    // 返金された
                  }
       fi
       goto end_purchase
  od

end_purchase:
  // 取引終了
}
```

遠隔購入取引をシミュレートする場合、預託者 Purchase だけでなく、売り手 Seller、買い手 Buyer も実装する必要があった。紙面の都合上それらのソースコードは割愛するが、それぞれに対応する Mealy マシンは Figure2,3 のようになっている。なお、図を簡潔にするため、Timeout などのイベントはその記載を省略している。なお、Seller、Buyer では状態数をコントロールするために明示的な状態宣言はしていない。このため、図にある状態名はあくまで Purchase との対応をとるために記したものであり、実際にはプログラムカウンタのように単なるプログラムの中の位置を表現しているにすぎない。

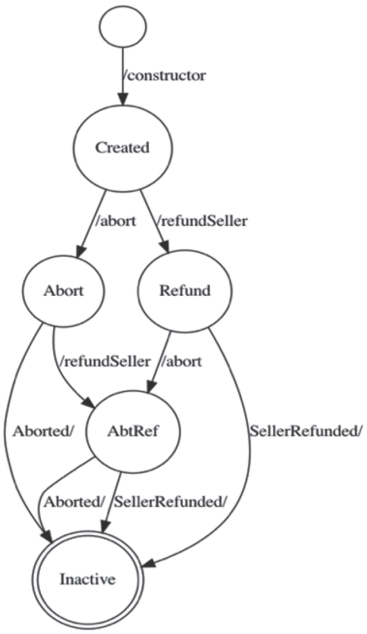

Figure 2: 売り手プロセス

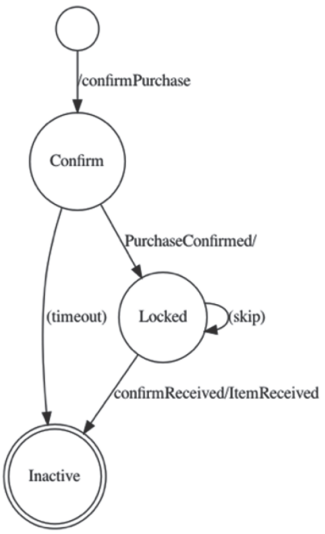

Figure 3: 買い手プロセス

ここで構築した Purchase、Seller、Buyer のモデルに対して SPIN を用いて検証した結果デッドロックフリー、すなわち取引が進まない状態がないことが確認できた。

4－4　商取引における「公平性」

ここで、買い手の Mealy 機械の図に

$$Locked \overrightarrow{(skip)} Locked$$

というループ状の状態遷移があることに注目したい。本プログラムでは単に以下のように実装されている。

```
do
  :: purchase! confirmReceived（buyer, NULL）-> break
  :: true ->
progress:
     skip
od
```

このskipは意味のあることは「何もしない」ことを表す手続きである。doブロック全体として、買い手が購入意思を示した後、品物の受け取りまでの時間をアイドリングしながら消費し、商品が到着すれば「品物の受け取りを確認した」というメッセージを送るという振る舞いを表現している。このプロセスに対し、ラベル「progress:」を外してモデル検査を実行すると、「non-progress execution cycles」すなわち進展のない無限ループが存在することを報告する。これは、無限回「purchase! confirmReceived（buyer, NULL)」が実行可能であるにもかかわらず実行されない可能性、すなわち（弱）公平性が満たされていないということを意味している。

このことを打開するためには、無限にskipを続けることをPurchaseの「環境」が打開することを示す「progress:」ラベルをつける必要がある。すなわち、ここで仮定している公平性とは「自分以外のプロセスによってもたらされる実行機会が公平である」ということを意味しており、通常のシステム記述においては、システムのスケジューラなどがこの役目を行う。しかし、遠隔購入取引においてこの無限ループを打開する（progress）インセンティブは買い手がおさめた預託金の返還であったことを思い出そう。例えば中古車など定価が存在せず、支払い後の代金の金額に折り合いがつかない場合、もしくは、特に買い手が粗悪品の中古車を掴まされたためにconfirmReceivedを発行したくない場合はどのように解決するのであろうか。売り手の資金力が買い手の資金力を上回る状況では、売り手は持久戦に持ち込むことができる。その状況において買い手が無限のskipを打開するということは、泣き寝入りを意味しないであろうか。

このような状況を解決するためには、この預託取引の他に、取引相手の「信頼度」や「返品」などの仕組みを付け加える解法もいくつか提案されている。しかし、通常、そのような追加の機能を付け加えたプロトコルの複雑さは向上し、極端に解析が難しくなることが予想される。一般の利用者が安心してスマートコントラクトを利用できるようにするため

には、世界中の技術者によってその性質が検証され、安全性が確認されたものだけを適切に選択できるような仕組み、プロトコルスイートの整備が必要である。

5．おわりに

本論文では、仮想通貨はブロックチェーンの発明により実現されたこと、そして、ブロックチェーンはスマートコントラクトのプラットフォームとして有力であることについて論じた。

しかし、非中央集権型のネットワーク上で展開されるスマートコントラクトには脆弱性が潜んでいる可能性があり、そこでの安全性の担保のためには、高度な技術が必要となる。特に、Solidity で記述されたスマートコントラクトをモデル記述言語 Promela に変換し、モデル検査を実行していく過程で、形式的な公平さと商取引における公平さには隔たりがあることがわかった。本論文では記載を省略したが、遠隔取引 Purchase を検証するためには売り手のプロセス Seller、買い手のプロセス Buyer の作り込みも必要であった。プログラムの設計者の主な仕事は不具合の検証ではなく、典型的な取引シナリオを実装することであり、「正しく」振り込みをしたプロセスが商品を受け取れないことは当然あってはならない。プロトコルの不具合が Seller と Buyer の実装の段階で混入する可能性も排除する必要がある。これらのギャップを埋めるためには、安全性の保証された完全なプロトコルスイートの整備が重要である。

本研究は、いままで筆者が抱いていた「非形式的な不公平感」が形式的な仕様のどの部分から発生しているかを示した。今後は、これまで提案されているスマートコントラクトテンプレートの形式的解析を進めるとともに、一般的な解析手法の確立、安全性の保証されたプロトコルスイートに基づくビジネステンプレートの実現に向けて努力していく。

参考文献

［1］Andreas M. Antonopoulos, Gavin Wood. (2019). "マスタリング・イーサリアム ―スマートコントラクトと DApp の構築（日本語）". オライリージャパン .ISBN-10:4873118964.

［2］Bowen Alpern, Fred B. Schneider.（1986）. "Recognizing Safety and Liveness". Distributed Computing, vol 2, p. 117--126.

［3］E. M. Clarke, O. Grumberg and D. Kroening, D. A. Peled, H. Veith.（2018）. Model

Checking Second Edition, The MIT Press.

［４］G. J. Holzmann.（2004）.The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley,

［５］Lamport L.（1977）. Proving the correctness of multiprocess programs. IEEE Trans Software Eng SE-3, ２:125-143

［６］Lamport, L.; Shostak, R.; Pease, M.（1982）. "The Byzantine Generals Problem". ACM Transactions on Programming Languages and Systems　4（３）:382–401. doi:10.1145/357172.357176.

［７］Lowe, Gavin.（1995）. "An attack on the Needham-Schroeder public key authentication protocol". Information Processing Letters. 56（３）: 131–136. doi:10.1016/0020-0190（95）00144-2.

［８］Nakamoto, Satoshi.（2009）. 2021-01-10 閲覧."Bitcoin: A Peer-to-Peer Electronic Cash System", https://bitcoin.org/bitcoin.pdf

［９］Needham, Roger; Schroeder, Michael.（1978）. "Using encryption for authentication in large networks of computers". Communications of the ACM. 21（12）: 993–999. doi:10.1145/359657.359659.

［10］Paolo Maggi, Riccardo Sisto.（2002）. "Using SPIN to Verify Security Properties of Cryptographic Protocols". Springer-Verlag, LNCS, p.187-204

［11］Popper, Nathaniel.（2016）.2021-01-10 閲覧."Paper Points Up Flaws in VentureFund Based on Virtual Money". The New York Times. ISSN 0362-4331.

［12］Rob Gerth.（1997）. 2021-01-10 閲覧. "Concise Promela Reference". http://spinroot.com/spin/Man/Quick.html.

［13］Solidity. 2021-01-10 閲　覧."Safe Remote Purchase". https://solidity.readthedocs.io/en/latest/solidity-by-example.html#safe-remote-purchase

［14］Vitalik Buteri.（2013）. 2021-01-10 閲覧. Ethereum Whitepaper. https://ethereum.org/en/whitepaper/.

［15］知念 祐一郎, 芦澤 奈実, 矢内 直人, クルーズ ジェイソン ポール.（2020）." スマートコントラクト――ブロックチェーンからなるプログラミングプラットホーム――". 電子情報通信学会　通信ソサイエティマガジン, vol. 14, no. 1, p. 26-33

［16］中島震. 2008."SPIN モデル検査―検証モデリング技法 ". 近代科学社, ISBN-10 : 4764903539