Rocq 上のスマートコントラクト検証フレームワーク ConCert における LTL 論理式による仕様記述と検証支援

諸冨 速人　西田 雄気

ブロックチェーン上に登録し動作するプログラムをスマートコントラクトと呼ぶ．本研究では, 時間に関する性質を記述可能な時相論理の一種である LTL（linear temporal logic）を用いて, コントラクト実行システムの Rocq 上での形式化ライブラリである ConCert [Annenkov et al. 2020] を拡張する．従来の ConCert は素朴な安全性（LTL 論理式の形で言えば単一コントラクトの状態に言及する述語 p を原子命題とした G p）の検証を支援する定義や補題を提供していた．本研究ではこの検証基盤を任意の LTL 論理式で記述された仕様を扱えるように拡張する．本研究は現在進行中のものであるが，G ΦのΦとして原子命題に限らず任意の LTL 論理式を置いた場合に対応できるよう支援補題を修正した他，U（Until）演算子の証明を容易にする補題については, 十分条件付きのランキング関数の存在証明に帰着させるアプローチを現在検討している．論文中では上記の成果に加え拡張における困難や選択についての議論やいくつかのコントラクトを用いた検証例を示す．

1　はじめに

ブロックチェーンシステムとは, 金融などの取引（以下トランザクション）の履歴を全参加者で共有する仕組みである．Ethereum [2] [18] に代表される多くのブロックチェーンプラットフォームで採用されているアカウントベースモデルでは, 全参加者はブロックチェーンシステム上にアカウントを所持し, その状態も共有される．

ブロックチェーン上に登録し動作するプログラムを**スマートコントラクト**（あるいは誤解のない場合単にコントラクト）と呼ぶ．これにより, 単純な送金よりも高度な操作をブロックチェーン上で行うことができる．コントラクトはそれぞれ固有の**内部状態**を持つ．例えば，クラウドファンディング・コントラクトならば寄付者のマップやファンディング期限などの情報を持つだろう．

ブロックチェーンシステムでは金融取引を行うことが多く，その障害・意図しない動作が大きな被害をもたらすため，事前の検証が重要である．特に，ブロックチェーンシステム特有の事情として一般に登録したスマートコントラクトは後から修正することが非常に難しい事実がある．これは特に公開型のブロックチェーンシステムにおいて支配的な所有者がいないプログラムという利点を損なわないためである．

プログラムの検証手法としては大きく分けると**テスト**によるものと**形式手法**によるものがある．テスト検証は実際にいくつかの入力を与えてみて期待された出力が返ってくるかを確かめる手法である．スマートコントラクトの文脈でも多くのテスト検証手法 [4] [6] [10] が提案されている．

形式手法を用いた検証は，対象を数学的に形式化した上で数学の道具立てを使ってプログラムの正しさを証明する方法である．本研究ではこちらのアプローチに基づいた検証を目的とし，定理証明支援系 Rocq [17] を用いたスマートコントラクト実行層の形式化ライブラリである ConCert [1] の拡張を行う．

検証フレームワークとして ConCert を利用する上で，ConCert では単一コントラクトの *safety* を証明するための便利な補題が提供されている [11]．直感的

Hayato Morotomi, Yuki Nishida, 東北大学情報科学研究科, Dept. of Graduate School of Information Science Science, Tohoku University.

には，safety は「常に，悪いことは起こらない」という性質である [7]．ある預金コントラクトの safety の例として，初期状態からこれまでの取引履歴を合算した全アカウントの残高と現在の内部状態に記録されている全アカウントの残高が常に一致している，などがある．提供された補題により，検証者はスマートコントラクト実行層の動作の詳細に触れることをせずに，主に仕様で言及している単一コントラクトの動作に関する検証条件だけを証明すればよい．

本研究では上記の検証アプローチにしたがって，ConCert を用いた単一コントラクトに対する safety 以外の一般的な時相的性質を検証することを目指す．Safety 以外の代表的な時相的性質として *liveness* がある．これは直感的には「いつか必ず望ましい結果が起こる」という性質である [7]．スマートコントラクトの文脈では，例えば「もしコントラクトにお金を預けたら，最終的にそれが返ってくる」などの livness を検証したい [14]．

既存の ConCert では safety 以外の時相的性質の検証は困難であった．特に liveness の関わる検証では以下のような問題点があった．

- コントラクト実行層の形式化である状態遷移系で，取引メッセージを処理できずに実質的に状態遷移ができない状態が存在した．具体的には，ユーザーからの取引が処理の途中で失敗する場合に行き詰まりが発生した．このような行き詰まり状態に望ましい事象が生じる前に到達する実行が存在すると，すべての実行においてその事象が最終的に生じることを要求する liveness を保証できない．実際のブロックチェーンプラットフォームでは，取引やブロック追加が処理され続ける．
- 1 つの取引の停止性が成立しない．これにより，実行可能なプロセスまたは遷移が永久に実行機会を奪われることを排除する *fairness* を仮定することが困難であった．liveness の証明には，この fairness のような仮定を入れる必要がある [7]．
- Liveness を記述する定義や証明のための補題が不足していた．補題なしで証明すると，仕様で言及しているコントラクトを直接は関係がないコントラクト実行層の動作の詳細に触れる必要があり，検証者の負担が大きい．

これらの問題点について，本研究では以下のアプローチで解決を試みた．

- 初期状態から到達可能な任意の状態からある状態に遷移できる遷移規則に変更した．ユーザーの取引が正常終了するか否かを確認して遷移するよう変更した．
- 取引を表す型に回数上限を表す型を加えることで，1 つの取引の停止性が成立するように変更した．Ethereum などのほとんどのブロックチェーンプラットフォームでは *gas* などの有限な実行資源を課す仕組みで 1 つの取引の停止性を保証しており [18]，これを抽象的に形式化した．
- 時相的性質を記述できる論理である *linear temporal logic* (LTL) [13] を ConCert 上に実装し，検証者が safety 以外の単一のコントラクトに関する時相的性質を記述できるようにした．また，LTL で記述した性質の検証を，言及している単一のコントラクトに関する検証条件に限定する補題をそれぞれ用意した．ただし，これらの検証条件を容易にするために，実装した LTL は時相演算子 X がなくす，原始命題の述語が言及できる情報に制限を加えるなどの点で標準的な LTL と異なる．

本研究の貢献をまとめると以下の通りである．

- ConCert のスマートコントラクト実行層の状態遷移系を liveness property の検証に適したものに変更した．
- LTL 論理式の形でブロックチェーンシステム上のある単一スマートコントラクトアカウントに対する時相的性質を記述できるようにした．
- LTL 論理式で記述された性質の証明について，論理式の構造に従って対象スマートコントラクトの動作だけに関する十分条件に帰着する補題を与えた．
- いくつかの具体例について，本研究の提案手法で検証した．

本論文の残りの構成は以下のとおりである．2 節では，スマートコントラクト実行層の動作と本研究の拡張対象である ConCert について概説する．3 節では，

本研究で行ったブロックチェーンシステムの状態遷移系の形式化の変更と Rocq 上での LTL の定義を示す．加えて，LTL で記述された仕様を検証する際に用いる証明補題について述べる．4 節では，具体的なコントラクトの検証例を述べる．5 節で，関連研究について議論し，6 節で本研究のまとめと今後の課題について述べる．

本研究における生成 AI の利用について

本研究では補題の証明において生成 AI（Chat GPT 5.6 Sol Pro 及び Gemini 3.1 Pro 拡張版思考モード）を支援に用いたほか，いくつかの補題については命題の候補出しにも用いた．

2 スマートコントラクト実行層と ConCert

本節では，まず，スマートコントラクト実行層の説明を通してスマートコントラクトの基本的な動作を示す．次に，スマートコントラクト実行層の Rocq での形式化ライブラリである ConCert について説明する．特に，スマートコントラクト実行層の状態遷移系での形式化を説明する．また，既存の ConCert を用いた具体的なコントラクトの safety の検証例も示す．

2.1 スマートコントラクト実行層

多くのブロックチェーンシステムは**スマートコントラクト実行層**と**コンセンサス層**という二つのレイヤーで構成されている．前者はユーザーからの**トランザクション**（より正確には以降で述べる**外部アクション**）を実際に処理して新しいブロックを生成する工程で，単一のサーバー内で完結する．後者は分散型ネットワークで構成されるブロックチェーンシステムにおいてサーバー間の同期を行うレイヤーである．新しく生成されたブロックの妥当性の検証やチェーンが分岐した場合の合意形成などはこちらのレイヤーで行われる．本研究では前者のスマートコントラクト実行層を取り扱う．

スマートコントラクト実行層の動作はブロックチェーンプラットフォームによって多様性があるが，以下では ConCert が主に想定している Tezos プラットフォーム [5] の動作に基づいた説明を行う．

各ブロックには，そのブロックの適用結果として得られる台帳が対応付けられる．この台帳状態とブロックのメタ情報（ブロックの長さなど）を含めてここでは**ブロック状態**と呼ぶ．一般に，コントラクトも自身のアカウントを所持し，残高とともにコードやプログラムの内部状態が記録される．

アカウントによる台帳の変化させる操作を，ここでは**アクション**と呼ぶ．アクションにはアカウント間の送金，コントラクトへの呼び出し，コントラクトの登録などがある．ユーザーアカウントからブロックチェーンにリクエストされる最上位のアクションを**外部アクション**と呼ぶ．ブロックにはブロック状態に加えてアクションの列が記録されている．

新しいブロックが追加されるとブロックのメタデータ（ブロックの長さなど）が更新され，ユーザーアカウントからリクエストされたいくつかのアクションがそこに記録される．そして直前のブロック状態からアクションが逐次的に実行され，ブロック状態が更新される．ブロックに含まれるすべての外部アクションの処理が完了すると，また新たなブロックが追加される．

コントラクトの呼び出しは記録されたコードの実行を行う．より有名な Ethereum と Tezos におけるもっとも大きな違いはこの部分にあり，Tezos においてコードの実行は与えられた入力（アクションと共に送られた実引数・現在の内部状態）から出力（新しい内部状態・追加のアクション）を返す純粋関数的な動作を行う．ここでは，このようにコントラクト実行によって生じるアクションを**内部アクション**と呼ぶ．内部アクションはコントラクトの呼び出しによって，さらに別の内部アクションを発生させることがあり，１つの外部アクションの処理はネストしたコントラクト実行を含み得る．内部アクションの実行中に失敗が生じた場合，起点となった外部アクションからの一連の状態変更が取り消される．

2.2 ConCert

ConCert は定理証明支援系 Rocq 上でスマートコントラクトを検証する枠組みである [1]．以下，実際

の実装コード †1 と Nielsen and Spitters [12] による形式的定義を用いて，ConCert におけるスマートコントラクトの形式化と，スマートコントラクトの実装例の説明を行う．

2.2.1 スマートコントラクトの形式化

ConCert では，スマートコントラクトは Rocq の純粋関数を用いて形式化されている．また，ブロックチェーン特有のデータ型やライブラリ関数が提供されている．コントラクトの内部状態の更新は，入力として事前状態，出力として事後状態を返すという状態渡しで形式化している．

具体的には ConCert のスマートコントラクトは，型 `Setup`，`State`，`Msg`，`Error` についてパラメータ化された以下の 2 つの関数で構成されたレコードである．

```
init : Chain → ContractCallContext →
    Setup → result State Error
receive : Chain → ContractCallContext
    → State → option Msg → result (
    State * list ActionBody) Error
```

`init` 関数はスマートコントラクトのアカウントがブロックチェーン上に登録された際に呼ばれ，初期状態を設定する．`receive` 関数はスマートコントラクトが呼ばれた時に実行される関数である．スマートコントラクトのエラーは，ConCert では result 型を用いて形式化している．

以下，各型とその役割について説明する．

- `Chain` はスマートコントラクトからアクセスできるブロックチェーンの属性を表す型である．具体的には例えば現在のチェーンの長さなどが含まれる．
- `ContractCallContext` は現在の呼び出しに関する情報（呼び出し元のアドレス，送金額など）に関する型である．
- `Setup` はスマートコントラクトの内部状態の初期パラメータである．
- `State` はスマートコントラクトの内部状態である．
- `Msg` はスマートコントラクト特有の呼び出し引数を表す．`Msg` 型の定義としてバリアント型などを用いることでエントリーポイントの機能を形式化できる．スマートコントラクト実行をせずに，送金のみ行う場合は `receive` の第 4 引数として `None` を渡す．
- `list ActionBody` はスマートコントラクトの実行によって発生した，追加の内部アクションを表す．アカウント間の送金，コントラクトへの呼び出し，新しいコントラクトの登録といったアクションの内容を表す．

Rocq 上でのスマートコントラクトの例を図 1 に示す．このスマートコントラクトは，特定の操作をある回数行うと，他の特定の操作が可能になるというコントラクトである．コントラクトはプログラムの状態として，`nat` 型の `count` と `bool` 型の `ended` の 2 つの内部状態をもつ (2 行目)．後者は終了フラグである．`CountCall`，`CountEnd` の 2 つのエントリーポイントをもち (3 行目)，前者がスマートコントラクト登録時に `setup_count` で設定された回数呼ばれた後でのみ，後者での呼び出しが成功するというスマートコントラクトである．

2.2.2 スマートコントラクト実行層の形式化

スマートコントラクト実行層は以下に示す `ChainState` 間の遷移関係で与えられる．

```
Record Environment := {
    env_chain :> Chain;
    env_account_balances : Address ->
    Amount;
    env_contracts : Address -> option
    WeakContract;
    env_contract_states : Address ->
    option SerializedValue; }

Record Action := {
  act_from : Address;
  act_body : ActionBody; }

Record ChainState := {
  chain_state_env :> Environment;
  chain_state_queue : list Action; }
```

`Environment` 型は，各ブロックの状態を表す．具体的には先ほど説明したブロックチェーンの属性を保持する `env_chain`，ブロックチェーンシステムに登録されている全てのアカウントの残高を保持する

†1 https://github.com/AU-COBRA/ConCert/tree/e86e71572d21f4951218351190698a4edd468665

```
Record Setup := { setup_count : nat }.
Record State := { count : nat; ended : bool }.
Inductive Msg := | CountCall | CountEnd.
Definition init (chain : Chain) (ctx : ContractCallContext) (setup : Setup)
    : result State Error :=
  Ok ({| count := setup.(setup_count); ended := false |}).
Definition receive (chain : Chain) (ctx : ContractCallContext) (st : State) (msg : option Msg)
    : result (State * list ActionBody) Error :=
  match msg with
  | Some CountCall => do _ <- throwIf st.(ended) default_error;
      Ok (st <| count := (st.(count) - 1)%nat |>, [])
  | Some CountEnd => do _ <- throwIf (0 <? st.(count))%nat default_error;
      Ok (st <| ended := true |>, [])
  | None => Err default_error
  end.
Definition contract := {| init := init; receive := receive |}.
```

図 1　ConCert でのコントラクト例：CountDown

$$\textsc{Step-Block}\quad \frac{b \text{ valid for } \sigma \qquad acts \text{ from users}}{(\sigma, [\,]) \;\to\; (\textbf{add_block}\; b\; \sigma,\; acts)}$$

$$\textsc{Step-Action}\quad \frac{\langle \sigma, a\rangle \Downarrow (\sigma', l)}{(\sigma, a :: l') \;\to\; (\sigma', l +\!\!+ l')}$$

$$\textsc{Step-Permute}\quad \frac{\mathrm{Perm}(l, l')}{(\sigma, l) \;\to\; (\sigma, l')}$$

$$\textsc{Step-Action-Invalid}\quad \frac{\neg(\exists \sigma', l'.\, \langle \sigma, a\rangle \Downarrow (\sigma', l')) \qquad a \text{ from a user}}{(\sigma, a :: l) \;\to\; (\sigma, l)}$$

図 2　スマートコントラクト実行層の遷移規則（Nielsen and Spitters [12] の p.6 に示された図に，Step-Action-Invalid 規則を本稿の著者が追加 †3）

`env_account_balances`，スマートコントラクトアカウントに登録されているスマートコントラクトを保持する `env_contracts`，スマートコントラクトアカウントの内部状態を保持する `env_contract_states` で構成される．†2`Action` 型は `ActionBody` に送信主の情報が付加されたものである．`ChainState` は各ブロックの状態と実行予定のアクションを組にしたものである．

`ChainState` 間の遷移関係を `ChainStep` として図 2 の規則によって定義する．図中では `ChainStep` 関係を $(\sigma, l) \to (\sigma', l')$ と表記し，「ブロック状態 σ においてアクション列 l が処理待ちのときにブロック状態とアクション列がそれぞれ σ' と l' に変化する」と読む．ここで σ は `Environment` の値を表し，l は `list Action` の値を表す．以下で，各規則について説明する．

Step-Block　新しい単一ブロックの追加を表す．全ての外部アクションが評価された後の空であるアクション列に新たにユーザーからリクエストされた外部アクションが追加される．ここで $acts$ を任意の外部アクション列とすることでユーザーからどういったアクションが送られてくるかという点を非決定的に扱っている．ブロック状態のチェーン長の増加やブロック作成者のアカウントの報酬獲得が行われる（`add_block` で表される）．「b valid for σ」の前提は，追加されるブロック b の高さが遷移前のよりも 1 段だけ高い，というような形式的な条件である．

Step-Action　先頭のアクション 1 つが `ActionEvaluation` 関係 $\Downarrow$ によって評価され，

†2　`WeakContract` や `SerializedValue` はスマートコントラクトによってばらつきのある内部状態の型を統一的に扱うためのものである．

†3　Step-Action-Invalid 規則は，ConCert 実装リポジトリに基づき，本稿の著者が推論規則として書き下したものである．`https://github.com/AU-COBRA/ConCert`（2026 年 8 月 6 日参照）．

ブロック状態とアクション列が更新される．$\langle \sigma, a \rangle \Downarrow (\sigma', l)$ は「ブロック状態 σ においてアクション a が評価されると，ブロック状態が σ' に変化し，内部アクションの列として l が出力される」と読む．追加された l はアクション列の先頭から追加される．

STEP-PERMUTE　アクション列を任意の順番に並び替える．この規則によって，任意の内部呼び出しの評価の順序に対応している．例えば，深さ優先実行や幅優先実行である．Nielsen and Spitters [12] らが幅優先実行として挙げていた Tezos，Scilla は，現在では内部アクションが深さ優先実行で処理される [3] [20]．

STEP-ACTION-INVALID　先頭のアクションが失敗した際にブロック状態を変化させずにそのアクションを除去する．内部アクション（コントラクトからのアクション）が失敗した際は，最上位の外部アクションの実行前に戻る必要があるため，この形式化では外部アクションのみがこの規則で処理できる．したがって，コントラクトからのアクション（内部アクション）が失敗する際は状態遷移が行き詰まる（正確には，その後 STEP-PERMUTE でのみ遷移し続ける）．

実行層の動作は `ChainTrace start end` という先頭と最後が `start` と `end` である `ChainState` のリストで隣接する要素が，`ChainStep` 関係にあるものとして形式化する．この形式化により 1 回のアクションの実行だけでなく，ブロックチェーンシステムの状態遷移列に関しての性質を記述，証明ができる．

2.2.3　単一スマートコントラクトに対する Safety Property の検証

前節の最後に述べた通り ConCert の形式化を用いると Rocq の能力の許す限りの検証作業を行うことができるが，検証者はスマートコントラクト実行層の動作の詳細まで把握する必要がある．この負担を軽減するために，ConCert では単一スマートコントラクトの safety に限った場合の証明をスマートコントラクト自身の動作（具体的には `init` 関数と `receive` 関数の定義）のみに依存した検証条件に帰着する `contract_induction` という補題を提供している．

```
1  forall (st : ChainState) (caddr : Address) (
       trace : ChainTrace empty_state st),
2  env_contracts st caddr = Some (contract :
       WeakContract) ->
3  exists dep_info cstate inc_calls,
4    deployment_info Setup trace caddr = Some
       dep /\
5    contract_state st caddr = Some cstate /\
6    incoming_calls Msg trace caddr = Some
       inc_calls /\
7    P (chain_height st)
8      (current_slot st)
9      (finalized_height st)
10     caddr
11     dep_info
12     cstate
13     (env_account_balances st caddr)
14     (outgoing_acts st caddr)
15     inc_calls
16     (outgoing_txs trace caddr).
```

図 3　Safety Property の記述フォーマット

`contract_induction` が適用できるかどうかは，証明したい性質の形に依存している．具体的には図 3 のような形である必要があり，`contract_induction` の結論部分がこの論理式に対応する．この中で `contract`，`Setup`，`Msg` は検証対象としているスマートコントラクトの具体的な定義に束縛された変数である．P には検証したい safety で成り立ち続けてほしい対象スマートコントラクトの仕様が入る．直感的には，初期状態から到達可能な任意のシステム状態で (1 行目)，コントラクトが登録されているならば (2 行目)，P が成り立つ (7-16 行目)，という命題である．

P の各引数について，以下で説明する．

chain_height, current_slot, finalized_height　ブロックの高さなどのブロックチェーンの属性．

caddr　コントラクトが登録されているアドレス

dep_info　Setup の値と登録者のアドレスと登録時に預け入れたお金の金額

cstate　現在の内部状態

env_account_balances st caddr　現在のコントラクトが紐付けられているアカウントの残高

outgoing_acts st caddr　実行予定のコントラクトからのアクション

inc_calls　これまでのこのコントラクトへの呼び出

```
1 Lemma ended_correct (st : ChainState) caddr
2   (trace : ChainTrace empty_state st) :
3   env_contracts st caddr = contract ->
4   exists (cstate : State),
5   contract_state st caddr = Some cstate /\
6   (0 < cstate.(count) -> cstate.(ended) =
      false).
```

図 4　Safety Property の例

しの情報

outgoing_txs trace caddr　これまでのこのコントラクトからのアクションの情報

例えば図 1 で定義した CountDown に対する Safety Property の一例である図 4 の記述は，`contract_induction` を用いて検証できる形にマッチする．図 3 の P で言えば，*cstate* : State のみに言及している．システムの初期状態 (empty_state) から到達可能な任意の状態 st に関して (2 行目)，コントラクトがアドレス caddr に登録されているならば (3 行目)，内部状態の count フィールドの値が 0 ならば，ended フィールドの値は false である (6 行目)，という仕様である．

図 4 の証明は実行列 (2 行目) に関する帰納法でも証明できるが，前述した通り煩雑な作業を伴う．ConCert で提供されている contract_induction 補題によって，遷移規則での場合分けのうち，仕様で言及しているスマートコントラクトに関連した場合で不変条件が成立することのみを検証者は証明すればよい [11]．P がシステム状態 **ChainState** の遷移列に関する述語ではなく，1 つのコントラクトに関する情報に制限された述語にすることで，このような簡略化を可能にしている．具体的には，主に以下の場合分けである．

- ブロックチェーンへの単一のブロックの追加の場合
- このコントラクトがブロックチェーン上に登録される場合
- このコントラクトが他のアカウントへの送金・呼び出しを行う場合
- このコントラクトが呼び出される場合：receive 関数で呼び出しが処理され，内部状態が前後で変化する

3　遷移規則の変更と LTL の実装

本節では，本研究が提案する，ConCert のコントラクト実行層の形式化の変更と LTL の実装を示す．このコントラクト実行層の形式化の変更により liveness の検証が可能になる．本研究では，検証者は本節で示される Rocq 上に実装された LTL でコントラクトの仕様を記述し検証することを想定している．検証を容易にするために，LTL は原始命題が主に単一コントラクトに関する情報に関する述語に制限されている．時相演算子 G, U それぞれに対応する証明補題を示す．

3.1　スマートコントラクト実行層形式化の修正

既存の ConCert では liveness の検証は困難であった．内部アクションが失敗した場合の遷移規則が存在しない（Step-Action-Invalid では失敗するアクションが外部アクションであることが要求されている）ため，すべての実行においてその事象が最終的に生じることを要求する liveness を保証できなかった．加えて，1 つの外部アクションから派生する内部アクションの処理が有限ステップで終了することが保証されていない．そのため，一つの外部アクションの内部処理が無限に継続し，後続する外部アクションやブロックの処理に到達しない実行を排除できなかった．

主に 2 点を変更した．1 点目は，外部アクションを評価する際に，それに伴う内部アクションも含めて正常終了するか否かをみて評価を進めるか全体を棄却するかを決めるように変更した．これにより，初期状態から到達可能な状態における内部アクション（コントラクトからのアクション）が必ず成功するため，状態遷移の行き詰まりが解消された．つまり，初期状態から到達可能な任意の状態から無限遷移列を構成できる．2 点目は，以下に示すように `Action` 型に回数上限を表す自然数型 `act_limit` を加えることで，1 つの取引の停止性が成立するように変更した．gas などの有限な実行資源による 1 つの取引の停止性の保証を抽象的に形式化した．内部アクションが増える度に `act_limit` の値が減っていき，0 でアクション実行が失敗する．

$$\frac{b \text{ valid for } \sigma \qquad acts \text{ from users}}{(\sigma, [\,]) \;\to\; (\textbf{add_block}\; b\, \sigma, acts)}\ \textsc{Step-Block}$$

$$\frac{\langle \sigma, a\rangle \;\Downarrow\; (\sigma', l)}{(\sigma, a :: l') \;\rightsquigarrow\; (\sigma', l +\!\!+ l')}\ \textsc{Inner-Step}$$

$$\frac{(\sigma, a :: l') \rightsquigarrow^* (\sigma_{finish}, l') \qquad \langle \sigma, a\rangle \;\Downarrow\; (\sigma', l) \qquad a \text{ from a user}}{(\sigma, a :: l') \;\to\; (\sigma', l +\!\!+ l')}\ \textsc{Step-Action-User}$$

$$\frac{\neg\big(\exists \sigma'.\, (\sigma, a :: l) \rightsquigarrow^* (\sigma', l)\big) \qquad a \text{ from a user}}{(\sigma, a :: l) \;\to\; (\sigma, l)}\ \textsc{Step-Action-Invalid}$$

$$\frac{\langle \sigma, a\rangle \;\Downarrow\; (\sigma', l) \qquad a \text{ from a contract}}{(\sigma, a :: l') \;\to\; (\sigma', l +\!\!+ l')}\ \textsc{Step-Action-Contract}$$

図 5　修正したスマートコントラクト実行層の遷移規則

```
Record Action := {
  act_limit : nat;
  act_from : Address;
  act_body : ActionBody;
}
```

図 5 に，本研究で用いる修正後の遷移規則を示す．以下, 変更前（図 2）と変更後の相違点を説明する．

- Inner-Step は ChainState 上の二項関係 $\rightsquigarrow$ の唯一の遷移規則である．内容は変更前の Step-Action と同じである．状態のアクション列の先頭のアクションを評価する規則である．
- 変更後の Step-Action-User と Step-Action-Invalid は, 外部ユーザーアカウントからのアクション（つまりトランザクション）が, それが引き起こす内部呼び出しも含めて, 正常終了するか否かを判断して遷移する．正常終了する Step-Action-User の場合，先頭のアクション 1 つを評価して遷移する．これは仕様の記述・証明の際に, 1 つの遷移で 1 つのアクションが評価されるような状態遷移系を扱いたかったためである．
- Step-Action-Contract は, 変更前の Step-Action に，コントラクトからの Action（つまり内部呼び出し）であるという条件を追加したものである．Step-Action-User の前提で正常終了することを確認すること（$(\sigma, a :: l') \rightsquigarrow^* (\sigma_{finish}, l')$）と，後述する `ActionEvaluation`($\Downarrow$) の決定性によって，初期状態 `empty_state` から到達可能なブロック状態では，内部アクションは失敗せず，必ずこの規則で遷移する．
- Step-Permute を削除した．したがって, 変更後の遷移規則は変更前とは異なり, 内部呼び出しの深さ優先実行にのみ対応している．

変更後は, 以下のアクションの停止性補題（補題 1）と状態遷移の継続性補題（補題 3）が成立する．$\textbf{empty_state} \to^* (\sigma, l)$ はシステムの状態 (σ, l) が初期状態 `empty_state` から到達可能であるという意味である．

補題 1 (アクションの停止性)**.**

$$\forall \sigma, a, l.\ \textbf{empty_state} \to^* (\sigma, a :: l) \implies \exists \sigma'.\ (\sigma, a :: l) \to^* (\sigma', l)$$

補題 2 (アクションの成否の決定可能性)**.**

$$\forall \sigma, a, l.\ (\exists \sigma'.\ (\sigma, a :: l) \rightsquigarrow^* (\sigma', l)) \vee \neg\,(\exists \sigma'.\ (\sigma, a :: l) \rightsquigarrow^* (\sigma', l))$$

コントラクト登録時のアドレスの決定性を仮定することで，ActionEvaluation($\Downarrow$) の決定性が成り立つ．これと補題 2 から補題 3 が導かれる．

補題 3 (状態遷移の継続性)**.**

$$\forall \sigma, l.\ \textbf{empty_state} \to^* (\sigma, l) \implies \exists \sigma', l'.\ (\sigma, l) \;\to\; (\sigma', l')$$

3.2　local-LTL

単一コントラクトに対する時相的性質の記述として本研究では原子命題を特定コントラクトの状態に関する述語に限定した *local-LTL* を提案する．

local-LTL の文法は以下の BNF で定義し，Rocq 上では図 6 のように定義する．

$$\varphi ::= p \mid \top \mid \bot \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \to \varphi \mid \mathbf{G}\varphi \mid \varphi\, \mathbf{U}\, \varphi.$$

後述する時相演算子 G の証明補題を適用後の検証条件を少なくするために, 時相演算子 X (Next) は除いた．コントラクトが動作する前後の仕様は, コントラクトを構成する関数 (**init**, **receive**) に関する性質

```
1  Inductive formula :=
2  | l_atom (P :forall
3    (chain_height : nat)
4    (current_slot : nat)
5    (finalized_height : nat)
6    (caddr : Address)
7    (deployment_info : DeploymentInfo Setup)
8    (cstate : State)
9    (balance : Amount)
10   (outgoing_actions : list ActionBody)
11   (incoming_calls_seen : list (
       ContractCallInfo Msg))
12   (outgoing_txs_seen : list Tx), Prop)
13 | l_true
14 | l_false
15 | l_not (f : formula)
16 | l_and (x y : formula)
17 | l_or (x y : formula)
18 | l_impl (x y : formula)
19 | G (f : formula)
20 | U (x : formula) (y : formula)
```

図 6　LTL 論理式の構文

として LTL を用いずに ConCert で検証できる．

各コンストラクタについて，LTL との対応や直感的な意味などを示す．

l_true　真

l_false　偽

l_not　否定

l_and　論理積

l_or　論理和

l_impl　含意

G x　すべての地点で x が成り立つ

x U y　いずれ y が成り立ち，その直前まではずっと x が成り立つ

原始命題 P はシステム状態遷移列のうち，コントラクトから参照できる情報に関する述語である．P の型は，`contract_induction` で求められる P と等しい．この制限によって，検証条件が仕様で言及している単一コントラクトに絞られる．

local-LTL 論理式の充足関係として「無限遷移列 `str` の `i` 番目以降において `caddr` に登録される `contract` に対して `f` が成り立つ」という事実を図 7 の `satisfies` として定義した．`str` と `contract` と `caddr` が文脈から明らかな場合 **satisfies** *contract caddr str i f* を $i \models f$ と記述することにする．

各コンストラクトの解釈は前述した直感的意味に沿ったものであるが，原子命題に関する解釈について少し補足する．ConCert の safety(図 3) に似て，P を現在までのコントラクトの情報に適用する．7-11 行目は図 3 の safety の記述と同様にコントラクトが登録されている仮定とコントラクトに関する情報である．特に，コントラクトが登録されている仮定（7 行目）で *contract* と *caddr* が紐付いている．当該コントラクトが登録されていない状態では，任意の原始命題 P で真になる．

また，仕様記述で便利な派生形式として LTL 論理式 `l_dep` を以下のように定義する．

```
Definition l_dep := l_not (l_atom (fun _
    _ _ _ _ _ _ _ _ _ => False))
```

$i \models$ `l_dep` は状態 i でコントラクトが登録されていることと同値である．

```
Lemma satisfies_l_dep_iff contract caddr
    str i :
satisfies contract caddr str i l_dep <->
env_contracts (CStr_nth i str) caddr =
    Some (contract : WeakContract).
```

3.3　論理式の構造に応じた証明補題

3.3.1　時相演算子 G の証明補題

時相演算子 G のための証明補題の概観を図 8 に示す．時相演算子 G の補題と contract_induction の相違点を以下に示す．

- `contract_induction` が単一コントラクトの状態に言及できる述語 P を原始命題とした G P にのみ適用できたことに対して，ϕ を任意の local-LTL 論理式とした G ϕ に適用できる．したがって，検証条件では j を i 以上の任意の自然数として，$S\ j \models \phi$ の結論に対して，帰納法の仮定は $j \models \phi$ である．
- $i \models f$ を示す検証条件が追加されている．図 3 のように，contract_induction が適用できる命題は，LTL で表現するならば原始命題 P に対して $0 \models$ G P で表現される．P はコントラクトが登録されていることは仮定されており (図 3 の 2 行

```
Fixpoint satisfies contract (caddr : Address)
    (str : ChainStream) (i : nat) (f : formula) : Prop :=
match f with
| l_atom P =>
  let st := CStr_nth i str in
  let trace := CStr_clist str i in
  env_contracts bstate caddr = Some (contract : WeakContract) ->
  exists dep cstate inc_calls,
    deployment_info Setup trace caddr = Some dep /\
    contract_state st caddr = Some cstate /\
    incoming_calls Msg trace caddr = Some inc_calls /\
  P st.(chain_height)
    st.(current_slot)
    st.(finalized_height)
    caddr
    dep
    cstate
    (env_account_balances st caddr)
    (outgoing_acts st caddr)
    inc_calls
    (outgoing_txs trace caddr)
| l_true => True
| l_false => False
| l_not x => satisfies contract caddr str i x -> False
| l_and x y => satisfies contract caddr str i x /\ satisfies contract caddr str i y
| l_or x y => satisfies contract caddr str i x \/ satisfies contract caddr str i y
| l_impl x y => satisfies contract caddr str i x -> satisfies contract caddr str i y
| G x => forall j : nat, (i <= j)%nat -> satisfies contract caddr str j x
| U x y => exists j, (i <= j)%nat /\
    (forall k, (i <= k)%nat -> (k < j)%nat -> satisfies contract caddr str k x) /\
    satisfies contract caddr str j y
end.
```

図 7　充足関係の定義

```
Lemma solve_G (contract : Contract
    Setup Msg State Error) (caddr :
    Address) (str : ChainStream) (
    start : CStr_nth 0 str =
    empty_state) (f : formula) :
forall i,
(* 検証に必要な十分条件 *)
... ->
... ->
(* 結論 *)
satisfies contract caddr str i (G f)
    .
```

図 8　時相演算子 G の証明補題：solve_G

目), 0 番目の状態が全てのコントラクトが未登録であるシステムの初期状態であると固定されていたため, $0 \models P$ が自明に真であった.

3.3.2　時相演算子 U の証明補題

時相演算子 U の証明補題を図 9 に示す. 本補題は liveness を証明する原理として一般的なランキング関数を用いるものに沿って構成した [13].

$\varphi \mathrm{U} \phi$ の証明として, 検証者はまず, 引数として単一コントラクトの内部状態の情報を受け取り整礎な型の値を返す部分関数としてランキング関数を定義する. 多くのコントラクトは, コントラクトと紐付いたアカウントの残高よりも, ブロックチェーンの長さかコントラクトの内部状態のあるパラメータを閾値として動作が変わる. これらの値は, 前者は単一ブロックの追加, 後者はアクション評価かつ当該コントラクトが呼び出された場合のみ変わる. したがって, ランキング関数は引数として, LTL の原始命題の引数 (図 6 の 2-12 行目) から, ブロックの長さと内部状態に関する

```
1   Lemma solve_U {A : Type} (R : A -> A -> Prop) (wfR : well_founded R) (bot : A) (eq_decA :
      forall x y : A, {x = y} + {x <> y}) (contract : Contract Setup Msg State Error) (caddr :
      Address) (str : ChainStream) (start : CStr_nth 0 str = empty_state) (x y : formula)
2   (rank_fun : Address -> Chain -> DeploymentInfo Setup -> State -> list (ContractCallInfo Msg)
      -> option A) :
3   forall i, let bstate_i := CStr_nth i str in let trace := CStr_clist str i in
4   (* Condition 1: start rank some *)
5   (exists v,
6     let Rank_Some := fun ch cs fh caddr dep_info cstate _ _ inc_calls _ =>
7       rank_fun caddr (build_chain ch cs fh) dep_info cstate inc_calls = Some v
8     in
9     satisfies contract caddr str i (l_dep l_and (l_atom Rank_Some))) ->
10  (* Condition 2: ランクが極小限でないならば，が成り立つ x *)
11  (let Not_bot := fun ch cs fh caddr dep_info cstate _ _ inc_calls _ =>
12      forall v1,
13      rank_fun caddr (build_chain ch cs fh) dep_info cstate inc_calls = Some v1 ->
14      v1 <> bot in
15    satisfies contract caddr str i (G ((l_atom Not_bot) l_impl x))) ->
16  (* Condition 3: 極小限でないランクはいずれ必ず減少する *)
17  (forall v1,
18    let Rank_Some := fun ch cs fh caddr dep_info cstate _ _ inc_calls _ =>
19      rank_fun caddr (build_chain ch cs fh) dep_info cstate inc_calls = Some v1 /\
20      v1 <> bot in
21    let Decrease := fun ch cs fh caddr dep_info cstate _ _ inc_calls _ =>
22      exists v2,
23      rank_fun caddr (build_chain ch cs fh) dep_info cstate inc_calls = Some v2 /\
24      R v2 v1 in
25    satisfies contract caddr str i
26      (G ((l_dep l_and (ltl_atom Rank_Some)) l_impl (l_true U (l_atom Decrease))))) ->
27  (* Condition 4: ランクが極小限ならば y が成り立つ *)
28  (let Rank_Bot := fun ch cs fh caddr dep_info cstate _ _ inc_calls _ =>
29      rank_fun caddr (build_chain ch cs fh) dep_info cstate inc_calls = Some bot in
30    satisfies contract caddr str i (G ((l_atom Rank_Bot) l_impl y))) ->
31  (* Conclusion *)
32  satisfies contract caddr str i (x U y).
```

図 9　時相演算子 U の証明補題

ものにさらに限定されたものを受け取るようにした (図 9 の 2 行目). 状態遷移系には当該コントラクトが登録されていない状態も含むため, 利便性のためにランキング関数は部分関数とした. 遷移列の i 番目時点でのランキング関数が定義されているときそのランクを，$\mathrm{rank}(i)$ と表記する.

そして, 以下の十分条件を証明する.

1. 今の状態 i のランクが定義される.
2. 任意の $j \geq i$ において，$\mathrm{rank}(j)$ が極小元でないならば，$j \models \varphi$.
3. 任意の $j \geq i$ において，$\mathrm{rank}(j)$ が極小元でないならば，ある $k > j$ が存在して，$\mathrm{rank}(k) < \mathrm{rank}(j)$. この状態遷移系では仕様で言及しているコントラクトと直接は関係のない遷移も含まれるため，任意の遷移でランクが減少することは一般に示せない. したがって，このように条件を緩めた.
4. 任意の $j \geq i$ において，$\mathrm{rank}(j)$ が極小元ならば，$j \models \phi$.

$\varphi \mathrm{U} \phi$ の証明が主に 1 状態に関する検証条件 (1, 2, 4) に分解される. φ, ϕ が G, U を含む LTL 論理式であれば複数状態にまたがる条件ではあるが，その場合はまた対応する補題を適用して検証を進める.

```
1  Record CallSpec := { origin_addr_spec : Address -> Prop;
2                       amount_spec      : Amount  -> Prop;
3                       msg_spec         : option Msg -> Prop; }.
4  Definition call_spec_queued (spec : option CallSpec) caddr (st : ChainState) : Prop :=
5    match spec with
6    | Some s => forall n, exists limit origin_addr amount msg,
7        (n <= limit)%nat /\ (* enough gas limit *)
8        origin_addr_spec s origin_addr /\
9        amount_spec s amount /\
10       msg_spec s msg /\
11       List.In (build_act limit origin_addr origin_addr
12                 (match msg with
13                 | None => act_transfer caddr amount
14                 | Some m => act_call caddr amount ((@serialize Msg _) m) end))
15               st.(chain_state_queue)
16   | None => True end.
17 Definition fairness (spec : option CallSpec) (caddr : Address) (str : ChainStream) : Prop :=
18   forall n, exists m, call_spec_queued spec caddr (CStr_nth (n + m) str).
```

図 10　fairness の仮定の定義

3.3.3　Fairness 仮定と補助補題

前述の条件 3 は以前として liveness の形を取っており直接証明することが依然として難しい．さらに言えば，コントラクトの内部状態はそのコントラクトが呼び出されることでしか変化しないため，多くの場合条件 3 が無条件に成立することは少ない．この点に対処するため「当該コントラクトが定期的に呼ばれる」という条件を `fairness` という条件として定義し，その条件が仮定されているもとで条件 3 をさらに分解する補助定理 `solve_U_eventually_decrease_spec` を提供する．

`fairness` の定義は図 10 の通りである．検証者は，CallSpec 型の値によって指定された呼び出しが無限回発生する無限遷移列を仮定して，仕様を記述できる．ランクが減少するために何らかの特定の操作が必要なコントラクトの liveness を検証する際に用いる．

`fairness` の仮定を持っている場合，補助定理を用いて条件 3 を以下の 1 ステップの状態遷移間の十分条件の証明に帰着できる．

3-1. 任意のエントリーポイントでの呼び出しが正常終了した前後で，ランクは増加しない．
3-2. ブロック追加の前後で，ランクは増加しない．
3-3. CallSpec（仕様に記述）を満たす任意の呼び出しは，内部呼び出しも含めて正常終了するかつランクが減少する．

4　検証例

本節では，2.2.1 節で紹介した CountDown コントラクトを用いて，LTL での仕様記述と時相演算子に対する補題による証明を示す．

時相演算子 G と U 両方を用いた仕様の例を図 11 に示す．「CountCall エントリーポイントでこのコントラクトが呼ばれ続けるならば，いつか必ず，それ以降 CountEnd エントリーポイントでの呼び出しが成功する状態になる」という仕様である．コントラクトの実装 (図 1) をみると，CountCall の処理で内部状態の count フィールドでがデクリメントされ (10-11 行目)，CountEnd は count フィールドが 0 の場合に成功することから，この仕様は成立する．これを形式的に保証する．

仮定などを整理すると，横線の上側を仮定，下側を結論として以下のようになる（LTL の Notation を使用）

$$\frac{i \models \mathrm{l_dep} \quad \mathrm{fairness}\ spec\ caddr\ str}{i \models \top\ \mathsf{U}\ (\mathsf{G}\, P_{\mathrm{CanEnd}})}$$

時相演算子 U の証明補題（図 9）を適用する．ランキング関数 f は以下のように定義する．内部状態の count フィールドを返す．

```
Definition rankf (_ : Address) (_ :
```

```
1  Lemma can_end caddr str (start : CStr_nth 0 str = empty_state) :
2  let spec := Some
3   {| origin_addr_spec := fun addr => address_is_contract addr = false;
4      amount_spec := fun amount => (amount = 0)%Z;
5      msg_spec := fun msg => msg = Some BombCall; |} in
6  fairness spec caddr str ->
7  let P := fun ch cs fh (caddr : Address)
8        (_ : option (DeploymentInfo Setup)) (cs : option State) (_ : Amount)
9        (_ : list ActionBody) (_ : option (list (ContractCallInfo Msg))) (_ : list Tx) =>
10     exists cstate, cs = Some cstate /\
11       exists ctx new_cstate,
12         receive (build_chain ch cs fh) ctx cstate (Some BombEnd) = Ok (new_cstate, []) in
13 satisfies contract caddr str 0  (G (l_dep l_impl (l_true U (G (l_atom P))))).
```

図 11　時相演算子 U を用いた具体例

```
    Chain) (_ : DeploymentInfo Setup) (
    cstate : State) (_ : list (
    ContractCallInfo Msg))
  : option nat := Some cstate.(count).
```

時相演算子 U の証明補題の十分条件について, U の第 1 引数が $\top$ であるため, 条件 2 は自明である. 条件 1 は $i \models$ l_dep より自明である. 今回のランキング関数はブロックチェーンの長さ (第 2 引数) を参照しないため条件 3-1 は自明である.

条件 3-1（任意のエントリーポイントでの呼び出しが正常終了した前後で，ランクは増加しない) に関して, `CountCall`, `CountEnd` それぞれでの呼び出しが正常終了した計算結果から `count` の値が増加しないことを示す.

条件 3-3（CountCall が正常終了）に関して，CountCall での呼び出しは `ended` $= \mathit{false}$ ならば成功する．ランク（`count`）が極小限でないことと，補題 `ended_conrrect`（図 4）により示される.

条件 4（i 以降の任意の j でのランクが極小元ならば $j \models \mathsf{G}\, P_{\mathrm{CanEnd}}$) に関して，$P_{\mathrm{CanEnd}}$ が真，つまり `CountEnd` での呼び出しが正常終了可能であるためには，`cstate.(count)` $= 0$ であればよい．内部状態の `count` が 0 であることを表す原始命題を $P_{\mathrm{CountZero}}$ とすると，Rocq 上では，$P_{\mathrm{CountZero}}$ は次のように記述される.

```
Definition  P_CountZero := l_atom (
fun _ _ _ _ _ cs _ _ _ _ => cs.(
count) = 0).
```

これを用いて，検証条件は以下に帰着される.

$$\frac{j \models P_{\mathrm{CountZero}}}{j \models \mathsf{G}\, P_{\mathrm{CountZero}}}$$

時相演算子 G の証明補題 `solve_G` を適用する．全てのエントリーポイントでの呼び出しで `count` $= 0$ が保存されることから示される.

5　関連研究

ConCert では単一コントラクトを対象とした検証を組み合わせて，複数のコントラクトで構成されている分散型金融取引所アプリケーションの safety の検証を行っている [11]．まず，2 節で触れたように単一コントラクトの safety を証明する．その後，コントラクト間の呼び出しの履歴が一致していることを利用して複数コントラクトの性質を証明している.

SmartPulse [16] はコントラクトの性質を LTL を拡張した SmartLTL で記述し，safety，liveness などの時相的な性質をモデル検査できる．具体例として，オークションコントラクトにおいて，ある参加者が入札した後に敗者となったならば，最終的にその参加者へ入札額が返金される，という liveness 仕様が挙げられる．敗者自身が掛金を引き戻すエントリーポイント（`withdraw`）を呼び出すことを fairness で仮定することで成立する．Schiffl and Beckert [14] は，このように実際に返金が行われることを要求するよりも，敗者が返金を実現するための操作を実行可能であること，すなわち返金を実現する能力（`enabledness`）を検証する方がスマートコントラクトの性質として適切であると主張している．本研究で検証した仕様 (図 11

の P) も，将来ある事象が実際に発生することそのものではなく，ある状態において目的とする操作が実行可能であることを表す点で，後者の `enabledness` に近い．

VeriSolid [8] [9] は，スマートコントラクトを高水準の状態遷移系としてモデル化し，モデル検査によって性質を検証した後，検証済みモデルから Solidity コードを自動生成する，設計段階で正しさを保証する開発手法である．これは相互作用する複数のスマートコントラクトの検証にも対応している．一方，論文中で示されている liveness の具体的な検証例は，主として単一の取引における処理の進行を対象としており，複数の独立した取引におけるコントラクトの呼び出しを跨ぐ liveness の検証例は示されていない．

コントラクトを状態遷移系としてモデル化する開発手法として SCAR [15] もある．これに伴い，LTL にコントラクト特有のプリミティブを追加した仕様記述言語を提案している [14]．この言語には，X 演算子がないという制限と，Negation Normal Form で *Until* が存在しないという SafetyLTL [19] と呼ばれる制限が加えられている．前者は，スマートコントラクト・ネットワークの文脈において次にどのトランザクションが実行されるかが予測できず，X 演算子による仕様が意味をなさないことから正当化されている．本研究では証明補題の与え方のために X 演算子を導入しなかったが，こうした観点から X 演算子の意味論について考察の余地がある．

SmartPulse，VeriSolid，そして SCAR と ConCert を比べると，モデル検査法と定理自動証明法，仕様で言及しているコントラクトのみに関する状態遷移系とブロックチェーンシステム全体の状態遷移系を扱う，などの相違点がある．

6 まとめと今後の課題

本研究は ConCert を用いて safety 以外の一般的な時相的性質を検証できるよう ConCert を拡張した．遷移規則の変更により，1 つの取引の停止性と状態遷移の継続性が成立することで liveness の検証に適したものになった．検証者が単一コントラクトに関する一般的な時相的性質を記述するために利用する local-LTL を Rocq 上に用意した．原始命題の述語が単一コントラクトに関する情報にのみ言及できることから，時相演算子 G，U などで記述された複雑な時相的性質を，単一コントラクトに関する遷移の前後の条件に帰着できる．具体的なコントラクトで safety と liveness を組み合わせた仕様を検証した．

6.1 今後の課題

ActionEvaluation ($\Downarrow$) の決定性のために，コントラクト登録時のアドレスの決定性を素朴に仮定したため吟味したい．fairness（図 10）は原始命題の引数の情報（図 6 の 11 行目）を利用すれば LTL 論理式として記述できるかもしれない．ConCert を用いて複数コントラクトの safety が検証されている [11] ため，本研究での local-LTL でも複数コントラクトの性質を検証できるように拡張したい．Weak Until ($\varphi \mathrm{WU} \phi := \mathrm{G}\varphi \vee \varphi \mathrm{U} \phi$) は「1 回だけ○○できる」というような仕様を記述する際に有用である．この証明は論理和を分解し時相演算子 G，U それぞれの証明を試みても一般には行き詰まってしまうため，WU のための補題が必要である．

謝辞 本研究は JSPS 科研費 JP26K21193 の助成を受けたものです．

原稿執筆における生成 AI および AI 支援技術の宣言 本原稿執筆において，ChatGPT 5.6 Sol Pro 及び Gemini 3.1 Pro 拡張版思考モードを 2 節のコントラクト実行層の説明と 5 節の各関連研究の草案及び図のキャプション案，そして補題 1-3 と遷移規則の図 25 の LaTeX 表現を生成する目的で利用した．得られた内容については精査及び編集の上，公表されるすべての内容について責任を負うものとする．

参 考 文 献

[1] Annenkov, D., Nielsen, J. B., and Spitters, B.: ConCert: a smart contract certification framework in Coq, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, (2020).

[2] Buterin, V.: Ethereum: A next-generation smart contract and decentralized application platform, `https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/EthereumWhitepaper-Buterin2014.pdf`.

[3] Contributors to the Octez & Tezos protocol documentation: Michelson: the Language of Smart Contracts in Tezos, https://octez.tezos.com/docs/active/michelson.html. Accessed: 2026-08-09.

[4] Gan, J., Su, J., Lin, K., and Zheng, Z.: FinanceFuzz: fuzzing smart contracts with financial properties, *Blockchain: Research and Applications*, Vol. 6, No. 4(2025), pp. 100301.

[5] Goodman, L.: Tezos – a self-amending crypto-ledger, https://tezos.com/whitepaper.pdf.

[6] Jiang, B., Liu, Y., and Chan, W. K.: ContractFuzzer: fuzzing smart contracts for vulnerability detection, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, New York, NY, USA, Association for Computing Machinery, 2018, pp. 259–269.

[7] Lamport, L.: Proving the Correctness of Multiprocess Programs, *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 2(1977), pp. 125–143.

[8] Mavridou, A., Laszka, A., Stachtiari, E., and Dubey, A.: VeriSolid: Correct-by-Design Smart Contracts for Ethereum, *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers*, Berlin, Heidelberg, Springer-Verlag, 2019, pp. 446–465.

[9] Nelaturu, K., Mavridou, A., Stachtiari, E., Veneris, A., and Laszka, A.: Correct-by-Design Interacting Smart Contracts and a Systematic Approach for Verifying ERC20 and ERC721 Contracts With VeriSolid, *IEEE Trans. Dependable Secur. Comput.*, Vol. 20, No. 4(2023), pp. 3110–3127.

[10] Nguyen, T. D., Pham, L. H., Sun, J., Lin, Y., and Minh, Q. T.: sFuzz: an efficient adaptive fuzzer for solidity smart contracts, *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, New York, NY, USA, Association for Computing Machinery, 2020, pp. 778–788.

[11] Nielsen, E. H., Annenkov, D., and Spitters, B.: Formalising Decentralised Exchanges in Coq, *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023, New York, NY, USA, Association for Computing Machinery, 2023, pp. 290–302.

[12] Nielsen, J. B. and Spitters, B.: Smart Contract Interactions in Coq, *FM Workshops (1)*, Lecture Notes in Computer Science, Vol. 12232, Springer, 2019, pp. 380–391.

[13] Pnueli, A.: The temporal logic of programs, *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, USA, IEEE Computer Society, 1977, pp. 46–57.

[14] Schiffl, J. and Beckert, B.: A Practical Notion of Liveness in Smart Contract Applications, *5th International Workshop on Formal Methods for Blockchains (FMBC 2024)*, Bernardo, B. and Marmsoler, D.(eds.), Open Access Series in Informatics (OASIcs), Vol. 118, Dagstuhl, Germany, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, pp. 8:1–8:13.

[15] Schiffl, J. and Beckert, B.: Scar: Verification-Based Development of Smart Contracts (Tool Paper), *6th International Workshop on Formal Methods for Blockchains (FMBC 2025)*, Marmsoler, D. and Xu, M.(eds.), Open Access Series in Informatics (OASIcs), Vol. 129, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (LZI), 2025, pp. 14:1–14:13. 46.23.01; LK 01.

[16] Stephens, J., Ferles, K., Mariano, B., Lahiri, S., and Dillig, I.: SmartPulse: Automated Checking of Temporal Properties in Smart Contracts, *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 555–571.

[17] The Rocq Development Team: The Rocq Reference Manual, https://rocq-prover.org/docs/, 2024.

[18] Wood, G.: Ethereum: A Secure Decentralised Generalised Transaction Ledger, https://ethereum.github.io/yellowpaper/paper.pdf, 2025. Ethereum Yellow Paper, Shanghai version.

[19] Zhu, S., Tabajara, L. M., Li, J., Pu, G., and Vardi, M. Y.: A Symbolic Approach to Safety ltl Synthesis, *Hardware and Software: Verification and Testing*, Strichman, O. and Tzoref-Brill, R.(eds.), Cham, Springer International Publishing, 2017, pp. 147–162.

[20] Zilliqa: Scilla in Depth, https://scilla.readthedocs.io/en/latest/scilla-in-depth.html. Accessed: 2026-08-09.

A 付録

```
Lemma solve_G (contract : Contract Setup Msg State Error) (caddr : Address) (str : ChainStream)
    (start : CStr_nth 0 str = empty_state) (f : formula) :
forall i,
(* iで fが成立 *)
satisfies contract caddr str i f ->
(* AddBlock *)
(forall (j : nat) (Hij : (i <= j)%nat) (header : BlockHeader)
    (queue_nil : chain_state_queue (CStr_nth j str) = [])
    (is_valid : IsValidNextBlock header (CStr_nth j str))
    (from_accs : Forall act_is_from_account (chain_state_queue (CStr_nth (S j) str)))
    (origin_acts : Forall act_origin_is_eq_from (chain_state_queue (CStr_nth (S j) str)))
    (env_eq : EnvironmentEquiv (CStr_nth (S j) str) (add_new_block_to_env header (CStr_nth j str
    )))
    (dep_info_eq : deployment_info Setup (CStr_clist str (S j)) caddr = deployment_info Setup (
    CStr_clist str j) caddr)
    (inc_calls_eq : incoming_calls Msg (CStr_clist str (S j)) caddr = incoming_calls Msg (
    CStr_clist str j) caddr)
    (out_txs_eq : outgoing_txs (CStr_clist str (S j)) caddr = outgoing_txs (CStr_clist str j)
    caddr)
    (IH : satisfies contract caddr str j f) (tag : TagAddBlock),
satisfies contract caddr str (S j) f) ->
(* Deploy *)
(forall (j : nat) (Hij : (i <= j)%nat) ctx setup state
    (ctx_addr : ctx.(ctx_contract_address) = caddr)
    (undeployed : env_contracts (CStr_nth j str) caddr = None)
    (init_some : init contract (transfer_balance ctx.(ctx_from) caddr ctx.(ctx_amount) (CStr_nth
     j str)) ctx setup = Ok state)
    (env_eq : EnvironmentEquiv (CStr_nth (S j) str) (set_contract_state caddr (serialize state)
    (add_contract caddr contract (transfer_balance ctx.(ctx_from) caddr ctx.(ctx_amount) (
    CStr_nth j str)))))
    (dep_info_eq : deployment_info Setup (CStr_clist str (S j)) caddr = Some (
    build_deployment_info ctx.(ctx_origin) ctx.(ctx_from) ctx.(ctx_amount) setup))
    (inc_calls_eq : incoming_calls Msg (CStr_clist str (S j)) caddr = incoming_calls Msg (
    CStr_clist str j) caddr)
    (out_txs_eq : outgoing_txs (CStr_clist str (S j)) caddr = outgoing_txs (CStr_clist str j)
    caddr)
    (inc_calla_nil : incoming_calls Msg (CStr_clist str j) caddr = Some [])
    (out_txs_nil : outgoing_txs (CStr_clist str j) caddr = [])
    (IH : satisfies contract caddr str j f) (tag : TagDeployment),
satisfies contract caddr str (S j) f) ->
```

図 12　時相演算子 G の証明補題 (1)

```
(* Outgoing *)
(forall (j : nat) (Hij : (i <= j)%nat) (out_act : ActionBody) (tx : Tx)
    (tx_from_caddr : tx_from tx = caddr)
    (tx_amount_eq : tx_amount tx = act_body_amount out_act)
    (tx_act_match : match out_act with
       | act_transfer to amount => tx_to tx = to /\ tx_amount tx = amount /\ (tx_body tx =
     tx_empty \/ tx_body tx = tx_call None)
       | act_deploy amount wc setup => tx_amount tx = amount /\ tx_body tx = tx_deploy wc setup
       | act_call to amount msg => tx_to tx = to /\ tx_amount tx = amount /\ tx_body tx =
     tx_call (Some msg) end)
    (chain_eq : (CStr_nth (S j) str : Chain) = (CStr_nth j str : Chain))
    (out_acts_eq : outgoing_acts (CStr_nth j str) caddr = out_act :: outgoing_acts (CStr_nth (S
     j) str) caddr)
    (dep_info_eq : deployment_info Setup (CStr_clist str (S j)) caddr = deployment_info Setup (
     CStr_clist str j) caddr)
    (inc_calls_eq : incoming_calls Msg (CStr_clist str (S j)) caddr = incoming_calls Msg (
     CStr_clist str j) caddr)
    (out_txs_eq : outgoing_txs (CStr_clist str (S j)) caddr =
        tx :: outgoing_txs (CStr_clist str j) caddr)
    (deployed_prev : env_contracts (CStr_nth j str) caddr = Some (contract : WeakContract))
    (cstate : State)
    (deployed_cstate_prev : contract_state (CStr_nth j str) caddr = Some cstate)
    (deployed_cstate_next : contract_state (CStr_nth (S j) str) caddr = Some cstate)
    (IH : satisfies contract caddr str j f) (tag : TagOutgoingAct),
satisfies contract caddr str (S j) f) ->
(* Nonrecursive Call *)
(forall (j : nat) (Hij : (i <= j)%nat) ctx prev_state msg prev_inc new_state new_acts
    (ctx_addr : ctx.(ctx_contract_address) = caddr)
    (from_other : ctx.(ctx_from) <> ctx.(ctx_contract_address))
    (deployed_cstate : contract_state (CStr_nth j str) caddr = Some prev_state)
    (receive_some : receive contract (transfer_balance ctx.(ctx_from) ctx.(ctx_contract_address)
      ctx.(ctx_amount) (CStr_nth j str)) ctx prev_state msg = Ok (new_state, new_acts))
    (env_eq : EnvironmentEquiv (CStr_nth (S j) str) (set_contract_state caddr (serialize
     new_state) (transfer_balance ctx.(ctx_from) ctx.(ctx_contract_address) ctx.(ctx_amount) (
     CStr_nth j str))))
    (dep_info_eq : deployment_info Setup (CStr_clist str (S j)) caddr = deployment_info Setup (
     CStr_clist str j) caddr)
    (prev_inc_calls_some :  incoming_calls Msg (CStr_clist str j) caddr = Some prev_inc)
    (inc_calls_eq : incoming_calls Msg (CStr_clist str (S j)) caddr
       = Some (build_call_info ctx.(ctx_limit) ctx.(ctx_origin) ctx.(ctx_from) ctx.(ctx_amount)
     msg :: prev_inc))
    (out_txs_eq : outgoing_txs (CStr_clist str (S j)) caddr = outgoing_txs (CStr_clist str j)
     caddr)
    (IH : satisfies contract caddr str j f)
    (tag : TagNonrecursiveCall),
satisfies contract caddr str (S j) f) ->
```

図 13　時相演算子 G の証明補題 (2)

```
(* Recursive Call *)
(forall (j : nat) (Hij : (i <= j)%nat) (ctx : ContractCallContext) (prev_state : State) (msg : option Msg)
    head prev_inc
    (new_state : State) (new_acts : list ActionBody)
    (ctx_addr : ctx.(ctx_contract_address) = caddr)
    (from_self : ctx.(ctx_from) = ctx.(ctx_contract_address))
    (action_facts :
       match head with
       | act_transfer to amount => to = ctx_contract_address ctx /\ amount = ctx_amount ctx /\ msg = None
       | act_call to amount msg_ser => to = ctx_contract_address ctx /\ amount = ctx_amount ctx /\ msg <> None /\ deserialize msg_ser = msg
       | _ => False
        end)
    (deployed_cstate : contract_state (CStr_nth j str) caddr = Some prev_state)
    (receive_some : receive contract (transfer_balance ctx.(ctx_from) ctx.(ctx_contract_address) ctx.(ctx_amount) (CStr_nth j str)) ctx prev_state msg = Ok (new_state, new_acts))
    (env_eq : EnvironmentEquiv (CStr_nth (S j) str) (set_contract_state caddr (serialize new_state) (CStr_nth j str)))
    (dep_info_eq : deployment_info Setup (CStr_clist str (S j)) caddr = deployment_info Setup (CStr_clist str j) caddr)
    (prev_inc_calls_some : incoming_calls Msg (CStr_clist str j) caddr = Some prev_inc)
    (inc_calls_eq : incoming_calls Msg (CStr_clist str (S j)) caddr  = Some (build_call_info ctx.(ctx_limit) ctx.(ctx_origin) ctx.(ctx_from) ctx.(ctx_amount) msg :: prev_inc))
    (out_txs_eq : outgoing_txs (CStr_clist str (S j)) caddr  = build_tx ctx.(ctx_limit) ctx.(ctx_origin) ctx.(ctx_from) ctx.(ctx_contract_address) ctx.(ctx_amount)
          (tx_call (match head with
                    | act_call _ _ msg => Some msg
                    | _ => None
                    end)) ::  outgoing_txs (CStr_clist str j) caddr)
    (IH : satisfies contract caddr str j f)
    (tag : TagRecursiveCall),
satisfies contract caddr str (S j) f) ->
(* conclusion *)
satisfies contract caddr str i (G f).
```

図 14　時相演算子 G の証明補題 (3)

```
Lemma solve_U_eventually_decrease_spec (contract : Contract Setup Msg State Error) {A : Type} (R
      : A -> A -> Prop) (wfR : well_founded R) (transR : transitive A R) (bot : A) (s : CallSpec
     ) (caddr : Address) (str : ChainStream) (start : CStr_nth 0 str = empty_state) (i : nat) (
     initial_dep_info : DeploymentInfo Setup) (eq_decA : forall x y : A, {x = y} + {x <> y})
  (rank_fun : Address -> Chain -> DeploymentInfo Setup -> State -> list (ContractCallInfo Msg)
     -> option A) :
  fairness (Some s) caddr str ->
  env_contracts (CStr_nth i str) caddr = Some (contract : WeakContract) ->
  deployment_info Setup (CStr_clist str i) caddr = Some initial_dep_info ->
(* i でランクが存在する．条件 1と等しい *)
(exists v,
  let Rank_Some := fun ch cs fh caddr dep_info cstate _ _ inc_calls _ =>
    rank_fun caddr (build_chain ch cs fh) dep_info cstate inc_calls = Some v in
  satisfies contract caddr str i (ltl_atom Rank_Some)) ->
(** Condition 3-1: contract call はランクを増加させない． *)
(forall v1 (chain : Chain) ctx dep_info prev_state msg prev_inc_calls new_state new_acts
    (receive_some : receive contract chain ctx prev_state msg = Ok (new_state, new_acts))
    (prev_rank : rank_fun caddr chain dep_info prev_state prev_inc_calls = Some v1),
  exists v2,
    rank_fun caddr chain dep_info new_state (build_call_info ctx.(ctx_limit) ctx.(ctx_origin)
     ctx.(ctx_from) ctx.(ctx_amount) msg :: prev_inc_calls) = Some v2 /\
    (R v2 v1 \/ v1 = v2)) ->
(** Condition 3-2: block addition はランクを増加させない． *)
(forall v1 old_chain header caddr dep_info cstate inc_calls
      (is_valid : IsValidNextBlock header old_chain)
      (prev_rank : rank_fun caddr old_chain dep_info cstate inc_calls = Some v1),
  exists v2,
    rank_fun caddr (build_chain header.(block_height) header.(block_slot) header.(
     block_finalized_height)) dep_info cstate inc_calls = Some v2 /\
    (R v2 v1 \/ v1 = v2)) ->
(**  Condition 3-3: CallSpecを満たす callは，極小限ではないランクを狭義に減少させる． *)
(exists (lower_bound : nat),
forall v1 v2 (j : nat) (Hij : i <= j) limit origin_addr amount msg acts prev_cstate inc_calls
  (deployed_prev : env_contracts (CStr_nth j str) caddr = Some (contract : WeakContract))
  (deployed_cstate_prev : contract_state (CStr_nth j str) caddr = Some prev_cstate)
  (Hinc_calls : incoming_calls Msg (CStr_clist str j) caddr = Some inc_calls)
  (queue_prev : (CStr_nth j str).(chain_state_queue) = build_act limit origin_addr origin_addr
    (match msg with | Some m => act_call caddr amount (serialize m) | None => act_transfer caddr
      amount end) :: acts)
  (enough_limit : (lower_bound <= limit)%nat)
  (spec_origin_addr : origin_addr_spec s origin_addr)
  (spec_amount : amount_spec s amount)
  (spec_msg : msg_spec s msg)
  (prev_rank : rank_fun caddr (CStr_nth j str) initial_dep_info prev_cstate inc_calls = Some v1)
  (not_bottom : v1 <> bot),
  exists finish
    (action_trace : ActionChainTrace (CStr_nth j str) finish),
    finish.(chain_state_queue) = acts /\
    forall (finish_cstate : State) finish_inc_calls,
      contract_state finish caddr = Some finish_cstate ->
      incoming_calls_action_trace Msg action_trace caddr = Some finish_inc_calls ->
      rank_fun caddr (CStr_nth j str) initial_dep_info
        finish_cstate (finish_inc_calls ++ inc_calls) = Some v2 ->
      R v2 v1) ->
```

図 15　Condition3 の証明を容易にする補題 (1)

```
(* Conclusiton : Condition 3 *)
forall v1,
  let Rank_Some := fun ch cs fh caddr dep_info cstate _ _ inc_calls _ =>
    rank_fun caddr (build_chain ch cs fh) dep_info cstate inc_calls = Some v1 /\
    v1 <> bot in
  let Decrease := fun ch cs fh caddr dep_info cstate _ _ inc_calls _ =>
    exists v2,
    rank_fun caddr (build_chain ch cs fh) dep_info cstate inc_calls = Some v2 /\
    R v2 v1 in
  satisfies contract caddr str i
    (G ((l_dep l_and (ltl_atom Rank_Some)) l_impl (l_true U (l_atom Decrease))))
    .
```

図 16　Condition3 の証明を容易にする補題 (2)