ブロックチェーンアプリケーション開発の実践と今後の課題

立石 孝彰[1,a)]　齋藤 新[1,b)]　岩間 太[1,c)]　天野 俊一[1,d)]　大澤 昇平[2,e)]　吉濱 佐知子[1,f)]

概要：我々は，これまでに複数のブロックチェーンアプリケーション開発に携わってきた．この中で，我々が主導した開発では，モデル駆動およびテスト駆動の開発方針を採用し，質の高いアプリケーションを短期間で開発してきた．これらの開発経験に基づいて，本論文では，ブロックチェーンアプリケーション (ブロックチェーン技術を用いたアプリケーション) 開発における我々の経験をまとめ，今後の研究課題・方針について述べる．

1. はじめに

ブロックチェーン技術を伴う開発・実行のためのプラットフォームとして，Hyperledger Fabric [*1] や Ethereum[*2] などが存在する．これらのブロックチェーンプラットフォーム上では，スマートコントラクトと呼ばれるイベント駆動のプログラムが動作し，クライアントアプリケーションから要求されるトランザクションを処理する．トランザクションは台帳と呼ばれるデータベースに蓄積されるため，過去のトランザクションを追跡することができ，監査が容易である．また，台帳は複数のサーバで分散して保持されるため，耐故障性に優れており，この観点から分散データベースとして見ることもできる．これら監査性と耐故障性の観点から，ブロックチェーン技術は金融や保険などの分野においても注目されている．

ブロックチェーン技術を用いたアプリケーションソフトウェア (以降ではブロックチェーンアプリケーションと呼ぶ) は，主にクライアントプログラムとスマートコントラクトから構成される．これまでに，我々は，Hyperledger Fabric を用いた複数のアプリケーション開発プロジェクトに携わってきた．いくつかのプロジェクトでは，ブロックチェーンの特徴であるスマートコントラクトの開発を主導した．これらスマートコントラクトの開発では，従来からあるリアクティブシステムに対するモデル駆動の開発方針が適当であると考え実践した．ここで用いたモデルは状態遷移モデルであり，このような状態遷移モデルに基づいたテストの自動化にも取り組んだ．この結果，高い品質のスマートコントラクトを短期間で作成することができた．また，これらの開発経験を通して，今後取り組むべき技術や研究課題の検討を行った．

本論文の構成は次の通りである．2 節において，Hyperledger Fabric を用いたシステムの一般的なアーキテクチャと，クライアントアプリケーションおよびスマートコントラクトの構成について述べる．3 節では，スマートコントラクトの開発事例と我々が実践した開発方針について説明する．4 節では，我々が利用・作成した開発ツールについて説明し，5 節において，今後必要となる開発技術について考察する．最後の 6 節で本論文のまとめを行う．

1 日本アイ・ビー・エム東京基礎研究所
2 東京大学
a) tate@jp.ibm.com
b) shinsa@jp.ibm.com
c) gamma@jp.ibm.com
d) e35063@jp.ibm.com
e) ohsawa@weblab.t.u-tokyo.ac.jp
f) sachikoy@jp.ibm.com

*1 Hyperledger Fabric は Linux Foundation のプロジェクトである．`https://www.hyperledger.org/projects/fabric`

*2 https://www.ethereum.org/

2. Hyperledger Fabric におけるアプリケーションアーキテクチャと開発モデル

Hyperledger Fabric[*3] は，分散台帳技術（以降では DLT:Distributed Ledger Technology）[18] を伴うリアクティブシステムのための実行基盤と見ることができる．台帳は，read-only のデータベースシステムと見ることができ，同じ内容の台帳が，複数のサーバ上で同期して保持される．トランザクションはブロックにまとめられ，ブロックのチェーンとしてデータベースに蓄積される．理論的には，台帳内のトランザクションを先頭から順に計算することによってデータベースの最新状態を取得することができる．台帳内のトランザクションから最新データを計算することは効率が悪いため，例えば Hyperledger Fabric にお

*3 本論文で扱う開発および事例は Hyperledger Fabric 0.6.1 に基づくものである．

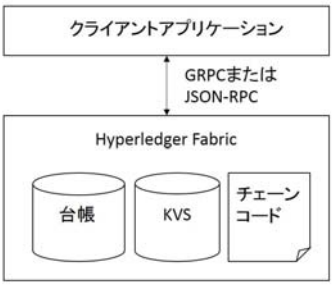


図 1　Hyperledger Fabric が想定するシステム構成例

いては，最新状態を保存するための専用の KVS 型のデータベース (以降では単純に KVS と呼ぶ) を利用できる．また，台帳内のトランザクションとその順序はハッシュ関数によって保護されており，改変等が行われないように設計されている．

2.1　Hyperledger Fabric の概要

Hyperledger Fabric が想定する一般的なシステム構成を図 1 に示す．ブロックチェーンアプリケーションは，主に Hyperledger Fabric 本体，チェーンコード (chaincode)，クライアントアプリケーションから構成される．ここで，Hyperledger Fabric は複数のサーバ上で動作し，それぞれのインスタンスをノードまたは peer と呼ぶ．また，チェーンコードは，Hyperledger Fabric におけるスマートコントラクトの呼び名である．チェーンコードは，特定の API を実装したリアクティブなプログラムであり，個々のチェーンコードには専用の KVS 型のデータベース (以降では単純に KVS と呼ぶ) が用意される．チェーンコードは Docker コンテナ内で実行され，その実装には，現時点において Go，Java を利用することができる．

クライアントアプリケーションは，Hyperledger Fabric と通信を行う．この通信には，gRPC[*4] または JSON-RPC[*5] を用いる．クライアントアプリケーションからトランザクションの依頼を受けた Hyperledger Fabric は，peer 間でコンセンサスをとった後にトランザクションを台帳に記録し，チェーンコードの API を呼び出す．ここで，コンセンサスとは，複数の peer 間で台帳の内容が同一となることを保証するために，peer 間でトランザクションの実行を同期させる仕組みである．

図 2 は，クライアントアプリケーション，Hyperledger Fabric，チェーンコード間の呼び出し関係を示したシーケンス図である．トランザクションには，チェーンコードのデプロイと KVS の初期化 (Deploy)，KVS の更新 (Invoke)，KVS への問合せ (Query) に対応する 3 種類が存在する．ここで，Deploy,Invoke トランザクションは，クライアントアプリケーションからは非同期的に呼ばれ，同じチェーンコードに対しては他の Deploy,Invoke トランザクションと並列実行されることはない．一方で，Query トランザクションは同期的であり，他のトランザクションと並列に実行され得る．

[*4] http://www.grpc.io/
[*5] http://www.jsonrpc.org/specification

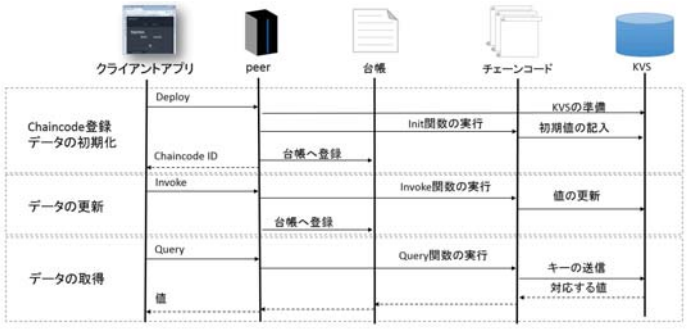


図 2　Deploy,Invoke,Query トランザクションのシーケンス図

```
1  function invoke(user) {
2     var invokeRequest = {
3        chaincodeID: chaincodeID,
4        fcn: "set_user",
5        args: ["a", "b", "1"]
6     };
7     var tx = user.invoke(invokeRequest);
8     tx.on('complete', function (results) {
9        console.log("invoke completed");
10       query(user);
11    });
12 }
```

図 3　HFC を用いた Invoke トランザクションの送信

2.2　クライアントアプリケーション

クライアントアプリケーションは，JSON-RPC または gRPC に基づくプロトコルによって，peer へトランザクションを依頼する．ここで，HFC (Hyperledger Fabric Client SDK) [*6] と呼ぶ，Node.js のための gRPC を利用したクライアント用ライブラリを用いることが多い．例えば，図 3 は，HFC を用いた Invoke トランザクションを行うためのコード例である．2–6 行目においてトランザクションに必要な情報を準備し，7 行目においてトランザクションを依頼している．ここで，3,4,5 行目では，トランザクションを依頼するチェーンコードの ID，トランザクション名，トランザクションの引数を設定している．このようにして送信されたトランザクション名と引数は，後述するチェーンコードの `Invoke` 関数に渡る．また，トランザクション結果を得るために，8 行目においてコールバックを設定している．Deploy および Query トランザクションの場合も同様に実装を行うことができる．

2.3　チェーンコード（スマートコントラクト）の構成

図 4 は，Go 言語で記述されたチェーンコードの例である．図 2 の通り，トランザクションの種類に応じて，`Init` 関数 (2 行目)，`Invoke` 関数 (7 行目)，`Query` 関数 (24 行目) が呼ばれる．各関数は，トランザクション名 `function` と，

[*6] https://www.npmjs.com/package/hfc

```
1  func (t *SimpleChaincode)
2  Init(stub shim.ChaincodeStubInterface,
3       function string, args []string) ([]byte, error) {
4      ...
5  }
6  func (t *SimpleChaincode)
7  Invoke(stub shim.ChaincodeStubInterface,
8         function string, args []string) ([]byte, error) {
9      ...
10     if function == "init" {
11         return t.Init(stub, "init", args)
12     } else if function == "set_user" {
13         marbleAsBytes, err := stub.GetState(args[0])
14         res := Marble{}
15         json.Unmarshal(marbleAsBytes, &res)
16         res.User = args[1]
17         jsonAsBytes, _ := json.Marshal(res)
18         err = stub.PutState(args[0], jsonAsBytes)
19         return nil, nil
20     }
21     return nil, errors.New("...")
22 }
23 func (t *SimpleChaincode)
24 Query(stub shim.ChaincodeStubInterface,
25       function string, args []string) ([]byte, error) {
26     if function == "read" {
27         name = args[0]
28         valAsbytes, err := stub.GetState(name)
29         return valAsbytes, nil
30     }
31     return nil, errors.New("...")
32 }
```

https://github.com/IBM-Blockchain/marbles/blob/v2.0/chaincode/marbles_chaincode.go より抜粋し，変数宣言やエラー処理等を省略したものである．

図 4　チェーンコードの例

引数 args をパラメータに持つ．例えば，Invoke 関数においては，トランザクション名によって処理の振り分けを行う (10,12 行目)．トランザクション名が set_user の場合には，KVS の値の取得と更新のために，GetState 関数 (13 行目) と PutState 関数 (18 行目) をそれぞれ用いている．Init 関数および Query 関数の実装方法も Invoke 関数と同様である．

2.1 節で述べた通り，Query 関数は他のトランザクションと並列に実行されることがあるため，大域変数を利用する場合には注意が必要である．また，Init および Invoke 関数は，決定的な動作を行わなければならない．ここで，決定的な動作とは，KVS の状態と関数に渡る引数が同じである場合は，関数実行後の KVS の状態も同じになる動作である．非決定的な動作の例として，現在時刻をを KVS に書き込む動作がある．このような場合，peer 間で KVS の状態が異なる可能性がある．同様に，外部の Web サービスや，map オブジェクトを用いたループを行う場合などにも，動作の決定性に注意する必要がある．

3. 開発方針と主な事例

本節では，我々が開発を主導したある申請書の承認プロセス (3.2 節) と証券取引 (3.3 節) を扱うチェーンコードの例と，その開発方針について述べる．いずれも実証実験レベルのアプリケーションの開発であり，機能要求等を制限している．まず最初に，3.1 節において，ブロックチェーンアプリケーションの開発方針について説明する．

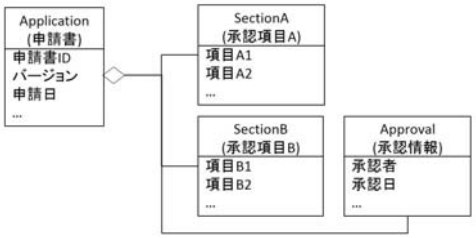
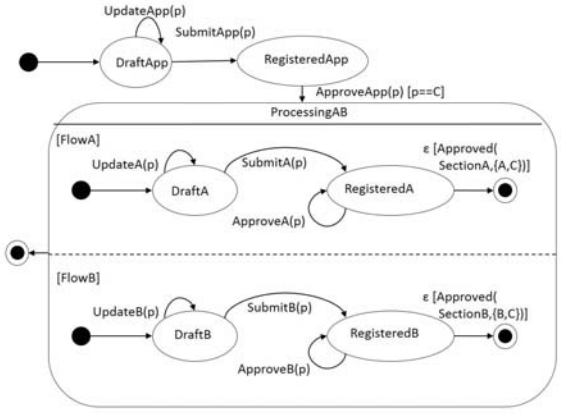


図 5　クラス図で表現した申請書のデータモデル



Processing,ProcessingAB は状態名であり，[] 内の,FlowMain,FlowA,FlowB はステートチャートに付けた名前とする．

図 6　申請書承認のステートチャート

3.1 開発方針

我々が行った開発では，主にデータモデルと状態遷移モデルの定義を行い，これらのモデルに基づいてチェーンコードの実装とテストを行った．このような開発方針を採った理由は次の通りである: (1) ブロックチェーンアプリケーションでは台帳へのトランザクションの記録が特徴であり，監査等において重要である (2) チェーンコード (より一般的にはスマートコントラクト) は，トランザクションに応じてリアクティブに動作するプログラムである．このため，トランザクションが扱うデータモデルと，状態毎に受付可能なトランザクションに着目した状態遷移モデルを構築することが適していると考えた．

3.2 申請書承認チェーンコードの開発

我々が扱った申請書承認プロセスでは，申請書の記入項目に応じて，異なる 3 者の承認が必要である．図 6 と図 5 は，この申請書承認プロセスの一部を簡略化したステートチャート [8] と，申請書のデータモデルである．申請書 (Application) は，セクション A,B(SectionA,SectionB) から構成されており，申請者は，申請書の概要を記入・提出 (UpdateApp, SubmitApp) し，承認者 C が承認 (ApproveApp) を行う．ここで，イベントに付随する属性 p は，トランザクションを実行したユーザを表し，ガード条件 [p==C] によって承認者が C であることを表す．セクション A とセクション B の承認は，それぞれ異なる承認

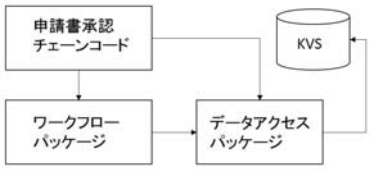


図 7　申請書承認チェーンコードのアーキテクチャ

者 ({A,C} と {B,C}) が行う．このため，これらのプロセスは平行して行われる．UpdateA,SubmitA,ApproveA は，セクション A に関する記入・提出・承認のイベントであり，同様に，UpdateB,SubmitB,ApproveB は，セクション B に関するイベントである．尚，図中のステートチャートではアクション (入力したデータの処理など) を省略している．

セキュリティの観点では，「承認者 A は，申請書のセクション A の内容のみを読むことができる」などのアクセス制御が必要であった．このアクセス制御は，申請書の項目 D，申請書の状態 Q，申請者および承認者のロール R，アクセス属性 C を引数とする述語として定義できた．ここで，申請書の項目 D は，図 5 のデータモデル中の各項目を一意に識別可能な名前の集合であるとする．ロールには，今回のプロジェクトでは簡単のため承認者名をそのまま利用した．アクセス属性 $C = \{\mathrm{r}, \mathrm{w}, \mathrm{a}\}$ であり，それぞれ，読込み，書き込み (更新)，承認を表す．申請書の状態集合 Q はステートチャートにおける状態名の集合である．このように，ロールベースアクセス制御に基づくが，申請書の状態 Q に応じてアクセス属性が変化する特徴がある．

申請書承認におけるチェーンコードの実装は，図 7 のアーキテクチャに従い，ワークフローパッケージとデータアクセスパッケージから構成される．DLT を用いたアプリケーションでは，今回のような申請書などのように，従来は紙のドキュメント等で管理されていた資産のライフサイクルを扱う要件が多くなるとの期待から，我々は KVS 上においてワークフロー管理と KVS へのデータアクセスを行うための再利用可能な Go パッケージを開発していた．本プロジェクトにおいてもこれらのパッケージを利用することによって，図 6 のステートチャートと，図 5 のデータモデルを直接的に定義した．このため，パッケージとモデルの実装に対するテストを明確に分離することができ，本プロジェクトではモデルに対応する実装のテストに多くの時間を割いた．このテストでは，ステートチャート上のすべての遷移を網羅するようなテストケースを 1 つだけ作成し，常にテストを行いながら遷移，ガード条件，アクションの実装を行っていった．ここでのテストケースとは，連続したトランザクション列のことであり，我々は，このようなテストケースを実行するプログラムを作成することによって，テストの自動化を行った．

3.3　証券取引チェーンコードの開発

証券取引では，各証券会社の顧客が証券の売り買いを行い，証券および資金の移動 (これを決済と呼ぶ) が行われる．図 8 と図 9 は，決済を含めた一連の証券取引プロセスを簡略化したステートチャートと，このプロセスが扱う関係データモデルである．取引 (Trade) は，売り手 (Seller) と買い手 (Buyer) の銀行が同じ場合には，直ちに決済可能な状態 (Settlement) に進む．ここで，決済方式には，いくつかの種類があり，我々が対象とした開発では，即時グロス決済 (isGross) と時点ネット決済 (isNet) の二つの方式を扱った．グロス決済では，振替処理が銀行に依頼され次第、一つ一つ即時に実行される．一方で，ネット決済では，振替処理が蓄積・遅延され (Waiting)，ある時点において金融機関同士で受払いする金額の差額のみを実際に振替処理 (doNetting) する．一方で，売り手と買い手の銀行が異なる場合は，相手からの許可 (ApproveTrade) を受けて取引・決済を進めることができ，また，取引の拒絶 (RejectTrade) とキャンセル (CancelTrade) も可能である．

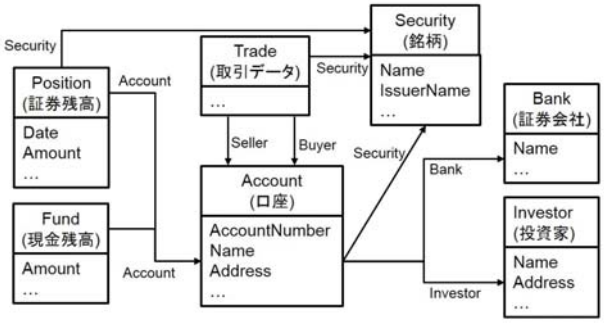


図 8　証券取引におけるデータモデル

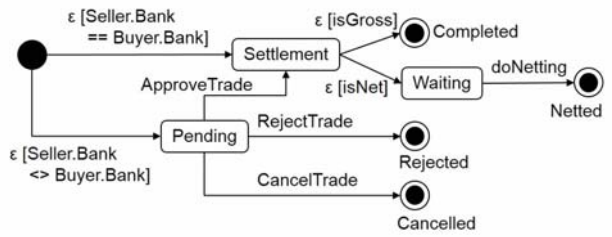


図 9　証券取引におけるステートチャート

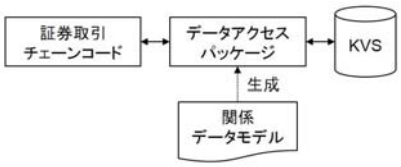


図 10　証券取引チェーンコードのアーキテクチャ

実装は，図 10 のアーキテクチャに従う．前節のようなステートチャート実行器を用いずに，チェーンコード中で状態変数を直接管理することによって状態遷移を扱った．また，チェーンコード中のオブジェクトを KVS を用いて永続化するために，後述する 4.3 節のツールを用いた．このツールは，関係データモデルからデータアクセスパッケー

```
1  flowA := ...
2  flowB := ...
3  mainFlow := Workflow{
4      InitState:   "DraftApp",
5      FinalStates: []State{"Completed"},
6      Transitions: Transitions{
7          {"DraftApp", "DraftApp", "UpdateApp",
8              nil, ActionUpdateApp, PostActionUpdateApp},
9          {"RegisteredApp", "ProcessingAB", "ApproveApp",
10             ConditionApprovedByC, nil, nil},
11         {"ProcessingAB", "Completed", EPSILON, nil, nil, nil},
12         ...
13     },
14     Subflows: Subflows{
15         "UwDetails": Workflows{
16             "FlowA":         &flowA,
17             "FlowB":         &flowB,
18         },
19     },
20 }
21
22 func (t *SimpleChaincode)
23 Invoke(stub shim.ChaincodeStubInterface,
24        function string, args []string) ([]byte, error) {
25     ...
26     appId := args[0]
27     return mainFlow.receive(appId, function, args[1:])
28 }
```

図 11　ステートチャートの定義と使用

ジの生成を行うものであり，ガード条件やアクションの実装のために利用される．チェーンコードのテストでは，ステートチャートの全遷移を網羅するテストと，各トランザクション毎の単体テストを行った．この単体テストは，トランザクションのパラメータなど，そのトランザクションに関与する属性 (テスト因子) に対して，とり得る値が正常な場合と異常の場合をテストした．この結果，単体テストケース数は，146 ケースであった．

4. 開発ツールの作成と利用

3 節で説明した開発方針に基づき，開発効率およびソフトウェア品質の向上のために我々が作成または利用した開発ツールまたはライブラリについて紹介する．

4.1 ステートチャート実行器

3.2 節で述べた通り，ブロックチェーンアプリケーションの動作の論理モデルとして，我々はステートチャートを用いた．このような論理モデルを直接定義および実行するために，Go 言語用のパッケージ (ライブラリ) を作成した．このステートチャートパッケージは，図 11 の通り，ステートチャートの定義を Go のオブジェクトを作成することによって行い，その定義に従って与えられたイベントに応じて状態遷移を行うものである．

我々のステートチャートパッケージは，(1) ステートチャートの特徴でもある階層化，空遷移，アクション，ガード条件をサポートし，また，(2) 特定のデータモデルやアクション言語とは非依存，かつ，(3) ステートチャートの定義を将来的には SCXML[*7] などの DSL(Domain Specific Language) として分離して定義することを想定して設計・

[*7] https://www.w3.org/TR/scxml/

```
1  type DAOReadInterface interface {
2      GetRoot() dao.JsonEntity
3      Query(string, ...bool) ([]dao.JsonEntity, error)
4      ...
5  }
6  type DAOInterface interface {
7      DAOReadInterface
8      Create(q string, jsonData string) error
9      Update(q string, jsonData string) error
10     Delete(q string) error
11     ...
12 }
```

図 12　DAO のインターフェイス定義

実装した．尚，有限状態オートマトンやステートチャートをシミュレートする既存の Go 言語用パッケージがいくつか存在し，https://github.com/hhkbp2/go-hsm は (1),(2) の特徴をカバーしている．

図 11 では，3-20 行目において図 6 で示したステートチャートの定義を行っている．ここで，子ステートチャート FlowA,FlowB も，同様に 1,2 行目で変数 flowA,flowB として定義されているものする．このように定義された子ステートチャートは，16,17 行目において子ステートチャートして登録される．ステートチャートにおける状態はその状態名を表す文字列によって識別しており，初期状態，終了状態，遷移を，構造体 Workflow のフィールド InitState,FinalStates,Transitions によってそれぞれ指定している．各遷移は，{src,dst,e,cond,preAction,postAction} の組によって定義され，遷移元状態 src において指定されたイベント e を受け取り，ガード条件 cond が満たされた場合，状態 src において事前アクション preAction を実行し，その後現在の状態を dst に変更し事後アクション postAction を実行する．ここで，事前/事後アクション，ガード条件は Go 言語の関数として実装され，その引数には 4.2 節で説明するデータアクセス用のオブジェクトに加えて，トランザクション実行者のユーザ情報やトランザクションへの引数が渡されるものとする．空遷移はイベントに EPSILON を用いることによって表している．このように定義したステートチャートは，27 行目のようにチェーンコードの Invoke 関数の中で利用することを想定している．この行によって，変数 appId で識別される申請書の状態が，イベント function とその引数 args[1:] によって変化する．args[1:] は，配列 args の 2 番目以降の要素の配列である．

4.2 KVS に対するクエリー言語およびアクセス制御

図 5 のクラス図は，単純な木構造を持ち，JSON オブジェクトとしてエンコードできる．例えば，JSON オブジェクト [{"a": 1, "b": {"c": 3}}] は，key-value のペアの集合 {("obj0",["obj1"]), ("obj1",["a","b"]), ("obj1.a",1), ("obj1.b", "obj2"), ("obj2",["c"]), ("obj1.c", 3) } として

```
1  type Position struct {
2      Id         string
3      AccountId  string //@PK
4      Account    *Account
5      SecurityId string //@PK
6      Security   *Security
7      Date       string //@PK @index
8      Amount     int
9  }
```

図 13　関係データモデルの定義

扱うことができる．ここで，obj0,obj1,obj2 は，対応する値が JSON オブジェクトを表す場合のキー名であり，実際には，他のプロパティ名と衝突することがない名前を用いる．特に，obj0 は，KVS 全体を表す JSON オブジェクトの識別名である．

以上のエンコード方法に基づき，KVS を JSON オブジェクトと見なして KVS へのデータアクセスを行うための DAO(Data Access Object) パッケージを利用した．DAO パッケージ中では，図 12 の通りの CRUD 操作を行うためのインターフェイスとその実装となる構造体が定義されている．JSON オブジェクトへのアクセスは，XPath 式のようなパス表現を用いて行う．例えば，申請書のデータモデルにおいて，すべての申請書の SectionA を表すパスは /Application/SectionA であり，これらの値を取得するためには，dao.Query("/Application/SectionA") を用いる．

また，我々の DAO パッケージでは，アクセス制御リストに基づいて，データアクセスの可否の判定も同時に行う．表 1 は，3.2 節で用いたアクセス制御の例である．ここで，承認権限については，対応する承認データ (/Application/Approval) への読み書きの可否として記載している．また，階層化された状態を扱っていることから，パス表記を用いて状態を識別している．例えば，/ProcessingAB/FlowA.DraftA は，状態 ProcessingAB 中の子ステートチャート FlowA の状態 DraftA を表す．また，任意の状態を表すために，*を用いている．

4.3　オブジェクト永続化ツール

チェーンコードでは，KVS を用いたプログラムを開発しなければならない．4.2 節では，このような KVS へのデータアクセスをクエリー言語によって簡単に行うものであったが，データモデルの定義は木構造に基づくものでなければならないという制約があった．そのため，図 8 の関係データモデルを KVS 上で扱うためには，データの重複などを許して木構造にエンコードする，または，参照を扱う特殊な値を用意するなど工夫が必要がある．しかし，このような工夫によって木構造へエンコードを行うと，KVS を用いた実装においては，元の関係データモデルではなくエンコード後の木構造を考慮してプログラムを書かなくてはならない．

このような問題点を解消するために，関係データモデルの定義から KVS を扱うプログラムを自動生成するツールを利用した．ここで，関係データモデルは，Go 言語の構造体として定義する．例えば，図 13 の通りの Go 言語の構造体が関係データモデルを表す．@PK と@index は，該当するフィールドが主キーとインデックスであることを表すアノテーションである．[*8]

```
1  func (this *HDLS)
2  listPositionsByDate(date string, user *User)
3    (*Positions, error) {
4    ...
5  }
```

図 14　関係データモデルに対して生成される関数

ツールの出力は，この構造体に対する関数 (KVS を扱うプログラム) である．例えば，アノテーション@index を付けたフィールド Date に対しては，図 14 中の 1-4 行目の関数 listPositionByDate を生成する．この関数を利用するこによって，KVS 内のデータ構造を気に掛けることなく，特定の日付を含むような Position レコードの集合を取得することができる．このようなレコード集合の計算は，KVS 上に Date フィールドの値による索引を作成することによって効率的に行われる．また，@PK を付けたフィールドの値は，KVS におけるキーとして利用され，CRUD 操作を行う関数に利用される．

5.　今後の研究課題

我々は状態遷移モデルに基づく開発を行っており，このような開発アプローチを対象とした従来の技術や方法論を適用できる．例えば，モデルベーステスト [6], [17]，モデル駆動アーキテクチャ [13]，モデル検査 [1], [9] がある．以降では，前節までに述べた開発の経験に基づき，モデル検証，プログラム合成 (チェーンコードの自動生成)，プログラム解析，ソフトウェアテストの観点から今後有用と考える研究課題について考察する．

5.1　モデル検証

本論文で扱った 3 節のプロジェクトでは，要件定義，ステートチャートとデータモデルの作成を我々自身が行い，モデルの正しさについては目視で確認した．例えば，3.2 節のモデルにおいては，「承認プロセス完了のためには，承認者 A,B,C の承認が必要である」などの要件が満たされていることの確認を行う．このような確認作業を計算機で自動的に行うために，ステートチャートとデータモデルに対する到達可能性の形式検証が必要と考える．ここで，ステートチャートは，各データ (例えば申請書) のライフサイクルを表していることに注意されたい．このような状態遷

[*8] 現在の実装の都合上，必ず Id フィールドを持つものとする．この Id は，参照を行う場合に利用するものであり，例えば，AccountId フィールドが利用する．これら，Id フィールドや AccountId フィールドは自動的に生成することが可能であるが，現在の実装では，明示的に定義している．

表 1　アクセス制御リストの例

データ項目へのパス	状態	承認者 A	承認者 B	承認者 C
/Application/ApplicationId	*	r	r	r
/Application/Version	*	r	r	r
/Application/SectionA	/ProcessingAB/FlowA.DraftA	rw	r	r
/Application/Approval/SectionA	/ProcessingAB/FlowA.RegisteredA	rw	rw	rw

移モデルの検証を行うツールとして，spin[*9]，MONA[*10]，Event-B[*11] などが利用できると考えている．

データベース分野において，状態の遷移列とデータを扱ったシステムを data-centric system 等と呼び，そのようなシステム M に対して性質 P が成り立つこと $M \models P$ を検証する手法 [4], [5] が多く提案されている．ここで，性質 P を記述するためには，状態とデータの両方を言及可能な論理が必要であり，例えば文献 [4] では，LTL (Linear Temporal Logic) を FO (First-order Logic) によって拡張した LTL-FO を用いることを提案している．前述の「承認プロセス完了のためには，承認者 A,B,C の承認が必要である」ということを LTL-FO を用いて記述すると次の通りである: $\Box(\forall d.\mathrm{Completed}(d) \Rightarrow \mathrm{Approved}(d, \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}))$．ここで，$d$ は申請書を表し，$\mathrm{Completed}(d)$ と $\mathrm{Approved}(d, \{\mathtt{A}, \mathtt{B}, \mathtt{C}\})$ は「申請書 d の承認プロセス完了」と「申請書 d が承認者 A,B,C によって承認済み」をそれぞれ表す述語である．一般的に，LTL-FO のように，FO を含むような論理は，その充足可能性が決定不能 (undecidable) である．そのため，LTL-FO　の記述力を制限することによって，検証を自動化する試みが行われている．

一方で，現実的な契約では，義務 (obligation) と権利 (permission) に関する条項から構成される．このような義務と権利を様相として扱った論理を一般的に Deontic Logic[11] と呼ぶ．また，このような義務と権利は，Dynamic Logic[3], [12] として見ることができる．そこで，契約書から Deontic Logic または Dynamic Logic の式 P を抽出し，システム M がその性質 P を満たすことの検証手法が有用であると考えている．

5.2　プログラムの自動生成

5.2.1　仕様に基づくプログラム合成

5.1 節では，仕様記述言語として Deontic Logic または Dynamic Logic を用いた検証について述べた．一方で，仕様からプログラムを自動生成するのと同様に，契約書から，その契約内容を実行するための data-centric system を自動生成する研究もまた有用であると考える．

5.2.2　datalog クエリーからチェーンコードの生成

3.3 節で述べた通り，実際の開発では関係データモデルを想定した設計を行うことが多い．関係データモデルで

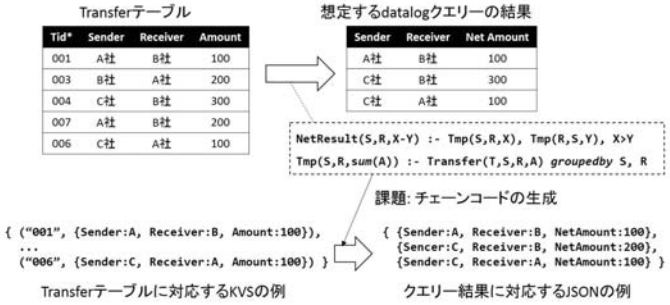


図 15　datalog クエリーからチェーンコードへの変換

```
1  import "time"
2  func (t *Sample)
3  Invoke(stub shim.ChaincodeStubInterface,
4         function string, args []string) ([]byte, error) {
5      var m map[string]int
6      ...
7      keys := ""
8      for k, _ := range m {
9          keys = keys + "," + k
10     }
11     stub.PutState("result", keys)
12     ...
13     date := time.Now()
14     if checkDate(date) {
15         stub.PutState("onTime", "yes")
16     }
17     ...
18 }
```

図 16　非決定的なトランザクション

は，datalog を用いてデータ操作を宣言的かつ簡潔に定義することができる．一方で，実装としてのチェーンコードでは，関係データモデルではなく KVS や JSON のように key-value 型のデータモデルを前提とするプログラムである．このため，実行効率の観点から，想定され得るデータ操作に適した構造を持つように KVS を設計する必要がある．このような手間を削減するために，datalog クエリーをチェーンコードに変換する技術が有用である．一般的に，datalog から手続き型プログラムへのコンパイル [10] や，クエリーとデータの変換は，データベース分野で古くから研究されており，チェーンコードの生成へ応用できるものと考えている．

例えば，図 15 は，3.3 節で扱った相殺決済の処理を datalog クエリー `NetResult` を用いて定義し，それをチェーンコードへ変換する様子を示したものである．ここで，Transfer テーブルは，送金処理データを持ち，これらのデータに基づいて相殺決済の処理を行うものとする．また，SQL と同様の集約関数 sum と group by を用いるものとする．

[*9] http://spinroot.com/
[*10] http://www.brics.dk/mona/index.html
[*11] http://www.event-b.org/

5.3 プログラム解析

2.3 節で述べた通り，チェーンコードの Invoke 関数と Init 関数は決定的でなければならい．例えば，図 16 は，非決定的な Invoke 関数の例である．5-11 行目は，map オブジェクトのキーとなる文字列を連結して，その結果を KVS に保存している．Go 言語では，map オブジェクトの要素での繰り返しでは，取り出される要素の順序が決定的ではない．このため，連結して作られた文字列 (変数 `keys` が持つ値) も実行毎に異なり，KVS に保存される値も実行毎に異なる．このようなプログラムを決定的にするためには，取り出した要素をソートしてから連結する必要がある．また，13-16 行目は，現在の時間を `checkDate` 関数で検査し，条件を満たせばキー名`"onTime"`で文字列`"yes"`を KVS に保存する．このようなプログラムも，実行毎に値が変わるため非決定的な動作の要因となる．決定的なプログラムにするためには，チェーンコード内で時刻を取得するのではなく，クライアント側で時刻を取得し，トランザクションを呼び出すときにその時刻を引数に与えて呼び出す必要がある．

これらのプログラムの非決定性の問題を検出するためのプログラム解析が必要であり，情報フロー解析 [14] やプログラム依存グラフ [7] を応用することによって実現できるものと考えている．例えば，図 16 中の 5-11 行目の問題は，情報フロー解析における explicit flow を解析するのみで検出可能でき，13-16 行目の問題では，implicit flow を解析する必要がある．なお，5-11 行目の問題に対しては，ソート結果を KVS に保存すると問題はなくなるが，このようなソート操作は情報フロー解析における declassification[15] と見ることができる．

5.4 ソフトウェアテスト

3 節で扱った開発プロジェクトでは，ステートチャートの遷移を網羅するようにテストを行い，各アクションや条件分岐に相当するトランザクション部分の単体テストについては，各開発者の裁量に委ねた．しかしながら，3.3 節で扱った証券取引のチェーンコードでは，トランザクション内で複雑な計算を行っている．このような場合，トランザクションに与える引数の組み合わせ (テストデータ) の求め方について，pairwise 法 [2], [16] など，系統的な手法を適用することによって，より高いテスト網羅性を達成しながらテスト工数を削減できたと考えている．さらに，トランザクション単体のテストだけではなく，状態遷移モデルに基づいたテストシナリオ (一連のトランザクション列) も考慮したテスト手法が必要と考えている．

6. まとめ

本論文では，状態遷移モデルに基づくブロックチェーンアプリケーション (特にチェーンコード) の開発事例と利用したツールを紹介することによって，ブロックチェーンアプリケーション開発の特徴を説明した．そして，これらの開発から得た経験に基づいて，チェーンコードに特有の今後必要となる以下の研究課題・方針について述べた．

- 契約書に対する data-centric system の自動検証手法
- datalog クエリーからチェーンコードの自動生成
- チェーンコードにおける非決定的動作の検出
- ステートチャートからのテストケース自動生成

参考文献

[1] Clarke, E. M., Grumberg, O. and Peled, D.: *Model checking*, MIT press (1999).
[2] Czerwonka, J.: Pairwise Testing in the Real World: Practical Extensions to Test-Case Scenarios, `https://msdn.microsoft.com/en-us/library/cc150619.aspx` (2008).
[3] De Giacomo, G. and Vardi, M. Y.: Linear temporal logic and linear dynamic logic on finite traces, *IJCAI* (2013).
[4] Deutsch, A., Hull, R. and Vianu, V.: Automatic verification of database-centric systems, *ACM SIGMOD Record* (2014).
[5] Deutsch, A., Li, Y. and Vianu, V.: Verification of hierarchical artifact systems, *PODS* (2016).
[6] Dias Neto, A. C., Subramanyan, R., Vieira, M. and Travassos, G. H.: A Survey on Model-based Testing Approaches: A Systematic Review, *WEASELTech* (2007).
[7] Ferrante, J., Ottenstein, K. J. and Warren, J. D.: The program dependence graph and its use in optimization, *ACM TOPLAS* (1987).
[8] Harel, D.: Statecharts: A Visual Formalism for Complex Systems, *Sci. Comput. Program.* (1987).
[9] Jhala, R. and Majumdar, R.: Software model checking, *ACM Computing Surveys (CSUR)* (2009).
[10] Liu, Y. A. and Stoller, S. D.: From datalog rules to efficient programs with time and space guarantees, *ACM TOPLAS* (2009).
[11] McNamara, P.: Deontic Logic, *The Stanford Encyclopedia of Philosophy* (Zalta, E. N., ed.) (2014).
[12] Meyer, J.-J. C. et al.: A different approach to deontic logic: deontic logic viewed as a variant of dynamic logic., *Notre dame journal of formal logic* (1988).
[13] Object Management Group, I.: OMG Model Driven Architecture, `http://www.omg.org/mda/`.
[14] Sabelfeld, A. and Myers, A. C.: Language-based information-flow security, *IEEE Journal on selected areas in communications* (2003).
[15] Sabelfeld, A. and Sands, D.: Declassification: Dimensions and Principles, *Journal of Computer Security* (2009).
[16] Segall, I., Tzoref-Brill, R. and Farchi, E.: Using binary decision diagrams for combinatorial test design, *ISSTA* (2011).
[17] Seifert, D.: Conformance Testing Based on UML State Machines, *ICFEM* (2008).
[18] 山藤 敦史，箕輪 郁雄，保坂 豪，早川 聡，近藤 真史，一木 信吾，金子裕紀：金融市場インフラに対する分散型台帳技術の適用可能性について，`http://www.jpx.co.jp/corporate/research-study/working-paper/tvdivq0000008q5y-att/JPX_working_paper_No15.pdf`.