データベース・アプリケーションへの実行時モデル検査導入手法の提案

宮永 照二
(株) テクノネット
myngshj3@gmail.com

要旨

情報システムの急速なかつ広範囲な普及に伴って，情報システムの製品の品質とプロセスの品質の両方において，新たな適用方法が提案されている *[2][3]*.

情報システムは開発・運用プロセスにおいてライフサクルを持つが，このライフサイクル内の各工程間でシステムに関する知識が共有されていないために様々な問題を引き起こしている *[2]*. 特にシステムの重要な性質についてテストすら完全にできない状態が存在することも報告されている *[2]*.

このような状況でシステムの重要な性質を検証する技術として，実行時モデル検査 *(On-the-Fly Model Checking)* がある *[7] [9] [10]*.

本稿では，情報システムとくにデータベースアプリケーションについて，実行時モデル検査を効果的に適用する方法について提案する.

1. はじめに

1.1. 情報システムにおける品質保証の状況

情報システムは，研究目的の一時限りの開発 (One-Time) ではなく，ビジネス上の要求にもとづくシステム開発ではその品質とコストと適用効果が考慮され，ある一定のタイムスパンを設けてライフサイクルが設定される (システムに対してライフサイクルという言葉を用いていることについては，システム，ライフサイクル概念，システムライフサイクルステージ，システムライフサイクルの典型について，[4] にその概要が記載されている．). 特に長いライフサイクルをもつレガシーシステムでは，計算機資源やアプリケーションの拡張によりパフォーマンスやユーザビリティの改善をもたらす一方で，ライフサイルのステージ間で知識の共有がなされないために，さまざまな問題を引き起こすことが報告されている [2].

レガシーシステムだけではなく，新規開発のシステムについても，レガシーシステムと同様にステージ間の知識の共有がなされず，1st リリース経過後の 2nd リリース，3rd リリース等の後期リリースで同様の問題が発生することが懸念される.

これらの問題の多くは，大規模化複雑化するシステムに対して，十分な要求や設計を吟味する時間が短くなる傾向があり，そのために，時間が経過するにつれて，ドキュメントとソースコードのギャップ，ドキュメントと実際の実装のギャップやドキュメント自体の非存在などが発生することが原因となっており，多くの場合，テストによる品質の担保すら困難な状況が発生している [2].

これは情報システムの製品としての品質が保たれていないだけではなく，開発プロセスの品質が担保されていないために発生している問題である.

さらに，現在は情報システムが社会インフラの 1 つとして認識されており，その信頼性や安全性という概念は重要な意味をもつ. 単に設計通りに動作するシステムを構築するというだけにとどまらず，フォールトトレラント・フェールセーフなどの運用やメンテナンスのステージにおける信頼性や安全性を含んだトータルな品質が要求される [6].

1.2. モチベーション

このような問題に対して，厳密なプロセスを経て，正しいソフトウェアを作る方法としてフォーマルメソッドがある [1] (ここでは正しいソフトウェアを仕様や想定通りの動作をするソフトウェアという意味で用いている).

フォーマルメソッドは数理的な技法を駆使して厳密なプロセスのもとに正しいプログラムを作成するための総合的な技術であるが，その用途は単に正しいソフトウェアを系統的に作成するだけにとどまらず，システムとしての性質を検証するのにも適用できる [1]．(正確にはシステムの性質を検証するプロセスはソフトウェアの信頼性を保証するためのプロセスに組み込まれており，そのプロセスに数理的な技法を適用している．)

情報システムはそれが大規模化複雑化すればするほど，仕様全体を把握することが困難であり，そのライフサイクルの各ステージ間で同じ知識を共有することはなおさら困難である．

このような状況に対して，システムの安全性などの重要な性質を実行時に検証する方法として，モデル検査 (Model Checking) がある．モデル検査は，検査対象のモデルを記述し，そのモデル上でシステムの重要な性質，例えば起こり得ない状態や安全性を脅かしかねない状態，あるいは重要な問題が発生した状態などを報告することによって，システムの品質向上に役立てることができる．モデル検査は、システムがとりうる状態を網羅的にたどることにより、システムの重要な性質が満たされることを検証する手法であるが、検証対象のシステムの規模によっては大きな状態空間とそれら状態間のネットワークを構築する必要があるため、性能の問題が顕在化する．特にこの性能の問題に対して，On-the-Fly モデル検査 (On-the-Fly Model Checking) [9] [7] [8] が提案されており，現実のシステム開発へ適用する研究もされている [8]．

複雑なシステムをモデル検査するのに，On-the-Fly モデル検査はパワフルで有効な手法である [9]．On-the-Fly モデル検査は，状態空間とそのネットワークを構築しながら，性質の検証を実行する [9] [10]．設計の早期の検証ステージで大量のバグを検出することができるため，本質的に設計プロセスでの適用に向いている [9]．

モデル検査のような数理的な技法による検証プロセスを実際に導入する場合，コストが余計にかかるというデメリットが存在する [3] モデル検査を適用することによる品質の向上という大きなメリットを生かすためには，通常のプロセスに自然と組み込まれているように適用されるのが好ましいと考えられる．たとえば，検証のための追加の労力を極力課さずに設計検証を行う方法として UML により記述される設計をモデル検査する方法の研究があるが [5][8]，これらも検証のための設計と製造のための設計を分離しない，つまりに二度手間を排除し効率的にモデル検査を適用しようとする事例である．

UML の使用の有無に限らず，モデル検査のための設計プロセスは，追加の労力を課さず，なおかつ専門的な知識の習得を極力排除して導入することが望ましいが，一般的なデータベース管理システムの機能を利用すると，比較的容易にシステムの動的な検証が可能である．そこで，データベースアプリケーションにフォーカスを当て，その構成や性質から，検証用のコードを追加する労力を極力抑えて，動的な検証，すなわち実行時モデル検査を実施する方法を考案した．

本稿では，その実現方式について述べる．

2. データベース・アプリケーションの特徴

2.1. リレーショナル・データベースとデータ構造

データベース・管理システムの４つの重要な特性，すなわち，ACID 特性 (Atomicity, Consistency, Isolation, Durability) はデータベース管理システムが提供する最も重要な性質である．これらの性質は，いくつかの構造を保全した情報システムの開発を容易にしている．

データベース管理システムをアプリケーションの中心に位置づける大きな要因は，大きく分けて次の点に分類されると考えている．

1. データについて構造を保全した記録と抽出が可能
2. 大量データの扱いについて，性能のチューニングが容易
3. バックアップ・リストア・レプリケーション等の管理が容易

1. は，リレーショナル・データベースのように何かしら存在する実体（エンティティ）間の関係を保存し，満たすべき関係を破壊する操作をキャンセルできる．この性質は重要な点である．既約なエンティティとそれらの関係からなるオブジェクトを統一的に記述できるアドバンテージは大きい．この構造は通常，静的な構造・リレーションを基本とする値の制約だけでなく，アプリケーションレベルの制約を与えることで構造を破壊するための起こるアプリケーションエラーを回避できる．

2. は，大量データを扱うアプリケーションに対して，特に個別のアルゴリズムの開発を行うことなく，性能を

改善することができる．実際には，その用途に応じて，キャッシュやインデックス等の構成を設定したり，検索のオプションの記述を追加することで，並列化等のアルゴリズムを用いて高速化をはかることができる．昨今のデータベースは大量のデータを保存し再利用する用途が多くなっており，それに対してデータ量に応じたアルゴリズムの切り替えはアプリケーションの仕様に影響を及ぼすため実施しがたい．データ構造やアプリケーションのアルゴリズムに影響を与えず，高速化をはかるためのチューニングの機能は重要である．

3. は，管理面の効率化に寄与する．データベースをバックアップする主な理由は，障害発生時の復旧にある．障害発生時にはバックアップ記録から，システムを安全裏に回復するための機能が必要である．

アプリケーションの品質にもっとも大きく影響するのは，やはり，1. データ構造の保全機能である．リレーショナル・データベースは前述の通り，エンティティ間の関係を保持し，エンティティ間の参照関係や依存関係を記述することができる．さらに，その使い方次第では，参照数の制限等の制約の追加などにより，より高い表現力を与えることができる．

例えば，グラフの構造を考える．グラフは構造 $G = (V, E)$ で与えられるが，最もシンプルな構造，すなわち頂点と辺の定義のみに基づく構造のみでは，任意のノード間に双方向の参照をもつグラフが記述される．しかし一般に，アプリケーションの持つデータ構造は，例えばそれは木構造であったり，サイクルであったり，またループを持たないものであったりする．そこでは，単なるグラフ G に対して，いくつかの制約を与えなければならない．しかもそれは，アプリケーションの実行のタイミングで，部分グラフ $G_1 \subset G$ が，組み込まれた別の構造を持つことがある．単なる静的な制約では実現できないこれらの構造の保全機能は，シンプルなプログラムを組み込むことで事前に実装可能となる．組み込み可能な構造についていくつか例を以下に挙げる．

1. 連結グラフ : 任意の 2 つのノードを結ぶパスが存在する
2. 二分木 : 連結グラフであり，かつ，すべての頂点 v に対して，v を根とする辺は存在しないか，または存在すればその数は 2 である
3. サイクル : ある頂点 v から同じ頂点 v へのパスが存在する

2.2. アプリケーションの仕様と有限オートマトン

データベース・アプリケーションはその名のとおり，そのデータをデータベースに保存するアプリケーションである．アプリケーションはデータベースの ACID 特性の恩恵により，アプリケーションを効率的に開発することができる．特に構造を保全する機能を導入することにより，重要なデータ構造の破壊をもたらす機会を除去することができる．

一方で，アプリケーションは有限状態機械とみることができる．例えば，一般的な Web アプリケーションは複数の画面と画面の背後にある，GUI と分離された機能からなり，ユーザは画面に対して操作し画面を切り替えながら操作する．ある 1 つの画面とその背後にある意味のある機能の組を 1 つの単位として，なになに画面のなになに状態，などと名付けることによって，アプリケーションの状態と関連付けることができる．

アプリケーションを有限状態機械とみなしたとき，それをデータベース内にグラフ構造を保存する機能を用いて実装することはあまりないと考えられる．それは実現可能であり，そのようなニーズも存在するかもしれないが，一般的にはアプリケーションの取りうる状態の全体をそのままデータベースに保存してなにかしらの機能を実現するために用いているとは考えづらい．

その一方で，有限状態機械を採用すれば，システムにその性質に対する高い表現力を与えることができる．ここで，有限状態機械の例として決定性有限オートマトンを挙げる．決定性有限オートマトンの定義は以下のとおりである．

$A = (Q, \Sigma, \delta, q_0, F)$．ここで，$Q$ は状態の集合，Σ は文字の集合，δ は遷移関数，q_0 は初期状態，F は受理状態である．

いま，このオートマトン A に対応するラベル関数 L を以下のように定義する．

$L : Q \cup \Sigma \to S$．ここで S は文の集合である．

すると，このオートマトンは図 1 のような概観をもつ．

いま，関数 L はオートマトンの取りうる状態や受け付ける文字から文への対応を与えるため，文 $\phi, \psi \in S$ にある意味を与えることにより，プログラムのモデルであるこのオートマトンに重要な意味を与えることができる．

$S = L(Q) \cup L(\Sigma)$ とする．$L(s)$ はある状態 $s \in Q$ に対応する文であり，この文が状態 s で常に成り立つ式

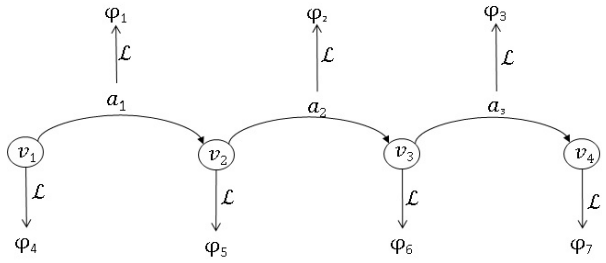

図 1. ラベル付きオートマトンの概念図

であるとする．また，$L(a)$ をある文字 $a \in \Sigma$ に対応する文であり，この文が文字 a に対応するプログラム (命令) であるとすると，オートマトン A はある命令 $L(a)$ を入力として状態遷移する状態遷移システムを表すとみなすことができる．さらに，$L(a)$ を正規言語に限定し，オートマトン A を非決定性オートマトンに拡張すると，このオートマトンは一般化非決定性有限オートマトン (Generalized Non-deterministic Finite Automaton: GNFA) を構成するとみなせる．

そこでいま，この GNFA を構成するための構造をリレーショナル・データベースに与えるために，次のようなテーブルを定義する．

表 1. 状態テーブル (States)

s	$label$
s_1	ϕ_1
s_2	ϕ_2
.	...
.	...

表 2. 文字テーブル (Alphabets)

a	$label$
a_1	ψ_1
a_2	ψ_2
.	...
.	...

このとき GNFA が矛盾なく構成される構造は以下の性質を満たす．

・状態のユニーク性: $\forall s, t \in States \circ s \neq t \rightarrow s.s \neq t.s$

表 3. 遷移テーブル (Transitions)

u	a	v	$remarks$
s_1	$a_{1,2}$	s_2	transition from v_1 to v_2
s_2	$a_{2,3}$	s_3	transition from v_2 to v_3
.	.	.	...
.	.	.	...

表 4. 現在状態テーブル (CurrentState)

s	$remarks$
s_x	$state - x$

・遷移状態の参照制約: $\forall t \in Transitions \circ t.u, t.v \in States \wedge t.a \in Alphabets$

・現在状態のシングルトン性: $\|CurrenState\| = 1$

以上で，GFNA の静的構造は完成する．このオートマトンが入力により状態遷移するには，状態遷移を発生させるための次の機能の記述が必要である．

・遷移実行処理: $doTansition(a \in Alphabets)$

2.3. アプリケーションを実行時に検証する仕組み

前項で説明した構造をもつ GNFA は，ある状態 $s \in S$ のときに成り立つ文 $L(s)$ を s で成り立つ式とみなす．また，ある文字 $a \in \Sigma$ のときに成り立つ文 $L(a)$ を状態遷移の契機となる命令とみなす．この構造を利用すると，図 1 であらわされるオートマトンの実行は，$\psi_1, \phi_1, \psi_2, \phi_2, \psi_3$ という文のシーケンスを表示する．言い換えれば，一連の文 $\psi_1, \phi_1, \psi_2, \phi_2, \psi_3$ はアプリケーションの動作シナリオとして定義される仕様の一つであり，このシナリオに基づいたアプリケーションの動作を検証することができる．

この例では，状態 s_1 にあるときに文 ϕ_1 を与えると，状態 s_2 に遷移し，さらにそのときに文 ϕ_2 を与えると，状態 s_3 に遷移するというシナリオを与えることができる．

これはちょうど，状態 s_1, s_2, s_3 をそれぞれ画面 1, 画面 2, 画面 3 に対応させ，文 ϕ_1, ϕ_2 をそれぞれイベント 1, イベント 2 に対応させると，前述のシナリオは,「画面 1 のときイベント 1 を発生させると画面 2 に遷移し，さらにそのときイベント 2 を発生させると画面 3 に遷移する」と言い換えることができ，操作仕様と対応付けられ

た仕様として記述することができる．この仕様は，オペレーションベースでかつ検証可能なため，先述したシステムライフサイクルのステージ間における知識の共有を促進する．

アプリケーションを実行時に検証する仕組みとしては，状態 s に対応する文 $L(s)$ を状態 s に遷移完了したタイミングで評価し，それが真となるか偽となるかを判定する方法が考えられる．より具体的には，文 $L(s)$ が表す SQL 文を実行し，その結果によって判定する方法が考えられる．

いま，Java や C#のようなオブジェクト指向言語で採用されている try〜catch ブロックによる例外捕捉のメカニズムの記法を導入し，状態遷移に伴う処理を以下のように記述したとする．

$\psi_1, try\{\phi_1, \psi_2\} catch\{\phi_2, \psi_3\}$

この処理のシナリオは ψ_1, ϕ_1, ψ_2 と $\psi_1, \phi_1, \phi_2, \psi_3$ の 2 つである．

2.4. 一階述語論理と時相論理による性質の記述

アプリケーションは一般的に，複数の異なる状態を取るが，各状態においてアプリケーションのもつデータがある性質を示している場合を考えると，

$s \rightarrow \forall o \in Objects \circ o.p$

のような，ちょうど状態 s にあるとき，$Objects$ のインスタンス o の性質 p を満たすというような一階述語論理の記述ができる．

さらに，アプリケーションの性質の意味論としてに有限オートマトンによる定義を与えると，例えば,「ある状態 s から，入力 a によってクリティカルな状態 s_c に到達する関係が存在する」という意味の到達可能関係 $s \overset{a}{\rightsquigarrow} s_c$ を反例とする，時相論理式

$\Box\neg\forall s \forall a \circ s \overset{a}{\rightsquigarrow} s_c$

を，有限オートマトンのもつ構造から検証することができる．この時相論理式の直感的な意味は,「どんな状態にあっても，どんな入力を与えても，状態 s_c に到達することはない」である．

2.5. コンカレントシステムの検証

マルチスレッドやマルチプロセスのシステムでは，複数のタスクが非同期に稼働する．分散処理においては，ひとつのデータベースに複数のタスクが同時に書き込みを行うが，書き込みの衝突を回避するために，同じ領域への書き込みを同時に行わないような制御がなされる．そのために，同一領域への書き込みリクエストを同時に処理しないような方式を採用する．

いま 2 つのプロセスを状態遷移機械で表現し，2 つのプロセスが同時に同じデータへの書き込みをしないかどうかを検証しようとする場合を考える．

プロセス 1 とプロセス 2 の計算モデルをそれぞれ，$A_1 = (Q_1, \Sigma_1, \delta_1, q_{1,0}, F_1)$, $A_2 = (Q_2, \Sigma_2, \delta_2, q_{2,0}, F_2)$ とする．

これらの並行プロセスのモデルをオートマトンの積 $A = \prod_i A_i$ を定義することであらわすことにする．ここで，

$\prod_i A_i = (\prod_i Q_i, \prod_i (\Sigma_i \cup \{\epsilon\}), \prod_i \delta_i, \prod_i q_{i,0}, \prod_i F_i)$,

$\prod_i \delta_i = \prod_i Q_i \times \prod_i (\Sigma_i \cup \{\epsilon\}) \rightarrow \prod_i Q_i$

である．

いま，2 つのオートマトンがそれぞれの状態遷移のシーケンス，s_0, s_1, s_2, s_3, s_0 と q_0, q_1, q_2, q_0 を取るとして，図 2 のようなコンカレントな状態をとる並行システムを考える．

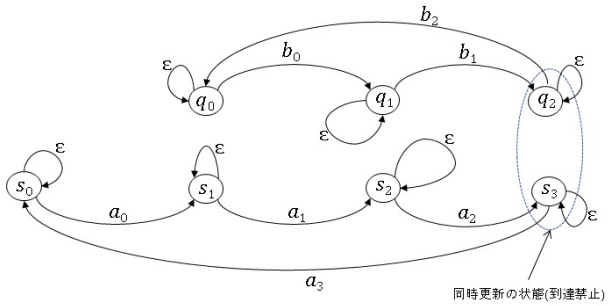

図 2. コンカレントシステムの状態図の概要

いま，A_1 と A_2 の積をとった時，現れる状態の一つ (s_3, q_2) は同時に書き込みが行われる状態であるとすると，この状態に到達しないこと，すなわち $G\neg(s_3, q_2)$ を検証するのが，実行時モデル検査の役割の一つである．

ユーザ $user$ と状態 $state$ の組 $(user, state)$ を保存し，集合 $\{(user, state)\}$ に $state = s_3$ なるエンティティと $state = q_2$ なるエンティティが同時に存在しないこと，すなわち一階述語論理式

$\neg(\exists u \circ u.state = s_3 \wedge \exists v \circ v.state = q_2)$

を満たすかどうかで検証できるし，このような性質を保全するために，状態遷移が確定する直前に，この一階

述語論理式が満たされない場合に，遷移をキャンセルするといった処理を組み込むことで，安全にコンカレントシステムの競合回避機能を実装することができる．

3. ケーススタディ

3.1. 電卓 GUI アプリケーション

トイ問題として，電卓 GUI アプリケーションを例に挙げる．

このアプリケーションの仕様は以下のようである．

画面仕様 : このプログラムは，1 つのスクリーンからなり，数字 : 0,1,..,9 に対応するボタンと演算子:+,-,*,/ に対応するボタンとクリアキー:C に対応するボタンとエンターキー:E に対応するボタンが配置されている．さらに，計算される式を画面に表示するための Widget と計算結果を画面に表示するための Widget からなる．

分析結果 : 電卓は一般的に，複数のキーを組み合わせて連続でキーインすることによって，計算したい式を指定する．さらに計算用のキーをキーインすることによって，計算を実行し，その計算結果を表示する．

計算式を指定するときには，計算式が適切な式になるように，キーインを受け付けるかどうかの制御を行う．例えば，すべてのキーインをすなおに受け付けてしまうと，計算式が例えば「3 + − 2」となるような，算出不可能な式を指定してしまう可能性があるために，入力状態に応じてキーインを受け付けるかどうかの制御を行う．

この分析結果を反映した結果，設計は以下のようになった．まず，このアプリケーションは以下の 3 つの状態を持ち，それぞれの状態からそれ自身または他の状態へ遷移する．

1. 初期状態 s_0 は，アプリケーションが立ち上がって最初の状態．初期状態 s_0 は，ユーザがなんのアクションをしなくても，初期化処理の後に次の状態 s_1 に遷移する．

2. 状態 1 s_1 は，数字キーイン待ち状態．状態 1 s_1 は，ユーザが最初に数字を入力するモードである．ここで数字を入力するとモード変更し s_2 に遷移する．演算キーインは受け付けない．C キーを入力すると s_0 に遷移する．また，このモードに滞在するときは，Enter キーを受け付けない．

3. 状態 2 s_2 は，Enter キーイン待ち状態．状態 2 s_2 は，ユーザが計算を実行可能な状態である．このモードに滞在するときは，演算キーインがあった場合は s_1 へ遷移し，クリアキーが入力されたときには s_0 に遷移する．Enter キーインがあった場合は計算を実行し，s_0 へ遷移する．

このアプリケーションの状態遷移システムは図 3 のようになる．

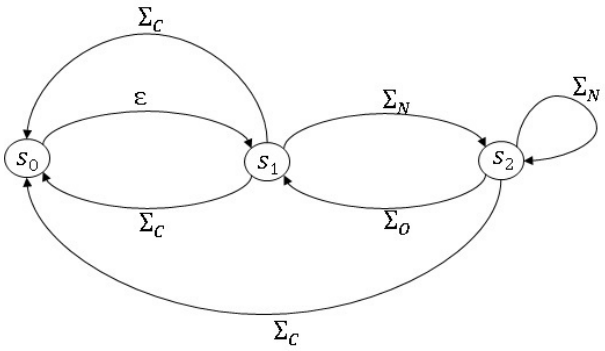

図 3. 電卓アプリの状態図

設計にならって有限オートマトンを構成すると，以下のようになる．

$Q = \{s_0, s_1, s_2\}$

$\Sigma_N = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$\Sigma_O = \{+, -, *, /\}$

$\Sigma_C = \{C, E\}$

$\Sigma = \Sigma_N \cup \Sigma_O \cup \Sigma_C \cup \{\epsilon\}$

$\delta = \delta_{s_0 \to s_0} \cup \delta_{s_0 \to s_1} \cup \delta_{s_1 \to s_0} \cup \delta_{s_1 \to s_2} \cup \delta_{s_2 \to s_0}$

$q_0 = s_0$

$F = \{s_2\}$

このとき検証したい性質は，Enter キーが受け付けられる場合には，計算式が適切であり計算可能な式を表している必要がある．いま計算式を μ とすると，μ は次のような正規表現のクラスに属する必要がある．

$\mu \in ((\Sigma_N^+ \Sigma_O^1)^* \Sigma_N)$

すなわち，このアプリケーションのもつ性質である，計算が正しく行われるための性質は，

$s_2 \to \mu \in ((\Sigma_N^+ \Sigma_O^1)^* \Sigma_N)$

となる．実行時にこのような性質を検証することによって，仕様の検証を行うことができる．

3.2. 既存仕様の記述:個人情報更新処理

トイ問題としてネットバンキングの個人情報更新機能を考えてみる．この処理では図 4 のようなステートチャートが想定される．

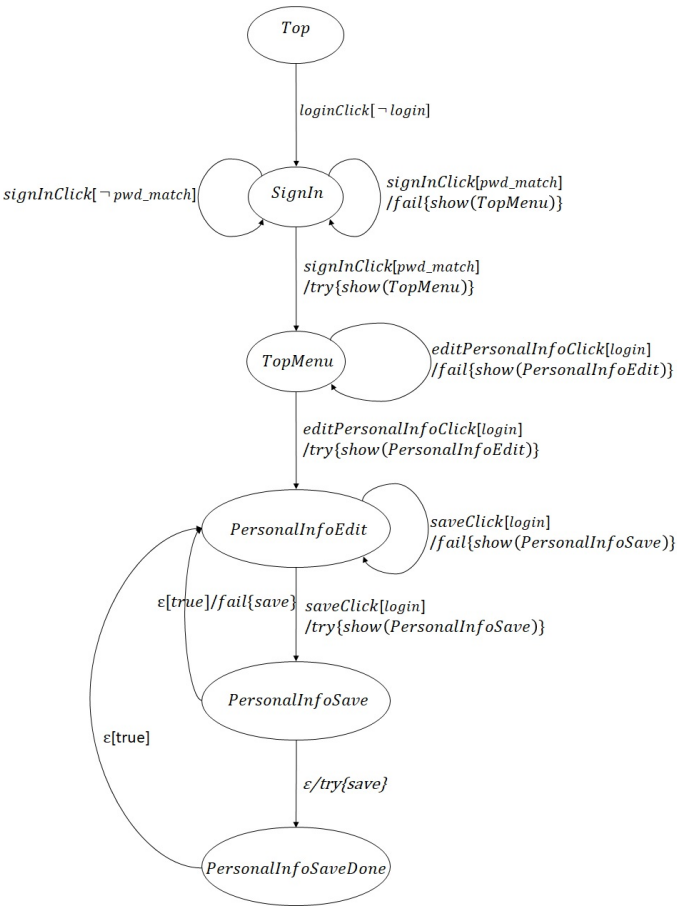

図 4. ネットバンキングのステートチャートの部分図

このログイン (サインイン) に関する性質には以下の性質があげられる．

1. 有効なログイン状態である $\equiv login$
2. 有効なログイン状態にない $\equiv \neg login$
3. Credential がマッチする $\equiv pwd\_match$
4. Credential がマッチしない $\equiv \neg pwd\_match$

このように定め，複雑な性質の記述を回避することにする．次にオートマトンを次のように構成する．

1. Step-1:ステートチャートから導き出される，静的に決定するオートマトンを構成する
2. Step-2:ガード条件のすべての性質を CNF または DNF に変換する
3. Step-3:時相論理に基づいて到達可能性判定を行う
4. Step-4:静的なオートマトンに対して，状態間の新たな遷移を追加したオートマトンを動的に構成する
5. Step-5:新しいオートマトンに対して性質を検証する

Step-4 と Step-5 を繰り返すことによって，On-the-Fly にオートマトンの構成とその上の性質の検証を同時に行う．

複雑なアプリケーションになればなるほど，静的な構造はそれを適切にモデル化することが難しく，実際のプログラムと乖離したモデルを作成してしまうことも往々に存在する．

実行時に，実際に実装された構造を構築するメリットは，本来ある仕様記述の不備や，想定されない振る舞いを検証する機会を与える．

この場合，あるときに成立していた性質が，その後のあるときには成り立たないことが起こりうる（到達可能な経路が現れる）．そのため，オートマトンが構成されたタイミングで検証を行う．

本提案手法の，新規開発ではなく，リプレースや機能追加・改変への有効性・メリットは以下のことがあげられる．

1. プログラムのある状態で取りうる性質が変わることへ対応できる．
2. ステートチャートと状態遷移表で直感的オペレーション仕様は推定できる．
3. 状態を性質ベースで記述するため，プログラマレベルで認識しているステートとユーザレベル認識ステートの乖離を減少させられる．

3.3. 機能追加:多重ログイン許可機能

前述のバンキングシステムではログインユーザ 1 人に対するステートチャートを考慮していた．しかし実際には，複数のユーザが同時に操作する場合があり，場合によっては同時ログインしての作業を許す場合もある．

そこで，多重ログインを許可するデータ構造を導入し，その際に想定される満たすべき性質を検証する例を考える．

現行のログインユーザテーブルを仮定し，そこにユーザ ID とは別のクライアント ID を設けるように拡張し，

以下のような構造を与える．

表 5. ログインユーザテーブル (LoginUsers)

userid	*clientid*	*login_time*	*lastevent_time*
id_1	$client_1$	*login time*	*last event time*
.	.	.	...
.	.	.	...

このような構造を導入しないと，次のようなことが起こる．端末 A で操作しているときに，タイムアウトとなる時間が経過している．端末 B でログインして操作すると，タイムスタンプが更新される．端末 A での操作はタイムアウトのはずなのに，更新がかかっているために引き続き作業ができてしまう．

これを検証するには，以下の式を検証する．

ログインユーザのテーブル $LoginUsers$ があって，ある $userid = id_1$ のユーザが，$clientid = client_1$ で実行している場合を考える．その際のテーブルの構造として，$(userid, clientid))$ は主キーとなるように制約を与えられているとする．

$C = CurrentStates = \{(userid, clientid, state)\}$ とすると，ある $userid$ に対して個人情報の安全な書き込みが満たされる条件は，現在状態がクリティカルセクション $S_c = PersonalInfoSave$ に同時にないことであり，

$\forall u \in LoginUsers \circ$

$\|\{v \in C; v.userid = u.userid \wedge v.state = S_c\}\| \leq 1$

を満たすことである．これが満たされている場合，同時にクリティカルセクション $PersonalInfoSave$ にないために，同時に書き込みに入ることはない．したがって同時保存の状態を回避することができる．万が一クリティカルセクションに同時に入ってしまった場合，実際に入ってしまったことを ASSERTION で確認することもできる．

4. ディスカッションと結論

4.1. ディスカッション

本稿では，データベース・アプリケーションが状態遷移機械とみなせる場合のシステムについて，実行時に時相論理や一階述語論理に基づくモデル検査を行い，システムの安全性などの重要な性質が満たされるかどうかを検証する方法を提案した．

そのなかで，実際のケーススタディとして，電卓アプリケーションとコンカレントな操作が可能なネットバンキングの例を説明し、さらに実際の開発の現場で実施される機能拡張に該当する同時ログイン機能追加を例にして，モデル検査の方法を説明した．いずれにしても，本稿では極めてシンプルで状態の数や遷移パターンが人手で数え上げられるほどであり，現実のシーンではもっと多くの状態や遷移パターンが現れることは必至である．

さらに，検証すべき性質を満たすかどうかの式は一般に，一階述語論理式と時相論理式で表されるが，開発者は，例えば単体テストを行うときには，SELECT 文で必要なデータを抽出し，プログラムの正しい動作を検証するということを行うのが通常であろう．本稿の冒頭で述べたように，アプリケーションのもつ性質についての知識がステージ間で共有されていない現実に対応するためには，適切に入力したり，問題をアラートするためのフロント機能の提供が必要と考える．

4.2. 提案手法のアドバンテージ

本稿で述べた方式による利点は以下のものがあげられる．

1. アプリケーションへの実装の影響を極力抑えて，検証を可能とする
2. アプリケーションの実行時にシステム検証を行うことを可能とする

既に稼働しているシステムや，モデル検査等による設計検証を実施せずに実装しているシステムに対して検証を行う方式であり，まだまだ新しい技術であるモデル検査がコストを要する現状においては，本稿記載の方式はアドバンテージがあるといえる．

また，実装方式としては，静的な状態空間を作成せず，オンデマンドに状態空間とネットワークを生成する

On-the-Fly 方式を採用することにより，アプリケーションの実行と同期的にモデル検査を実施するという方式を提供できる点もアドバンテージがあると考えられる．

この点については，今後試験実装を進めることにより，本稿提案の方式の実用性を検証する必要がある．

本提案手法で懸念される問題は次項以降で説明する。

4.3. 性能面の問題

データベース管理機能を検証のプラットフォームとして利用するというアイデアは，実装の容易性や機能としての有効性とは相反する性能の低下を招くおそれがある．性能面の課題は以下のようなものがある．

1. 状態を記録するオーバーヘッドにどう対処するか
2. COMMIT によるブロックの発生による影響をどう抑えるか
3. ネットワークトラフィックの増加にどう対処するか

状態を記録するためにおこるオーバーヘッド，状態を確実に記録させるための記述量の増加とそのために発生する処理が懸念材料である．また，COMMIT によるブロックの発生頻度も考慮しなければならない．多数のユーザが利用するデータベースアプリケーションでは，ネットワークトラフィックの増加による検証への影響も考慮しなければならない。これらの性能低下を和らげる方策の一つとして，On-the-Fly モデル検査を採用した．On-the-Fly モデル検査は、検証が必要なときに必要な状態空間を生成するため，実行時のコストが低い．また状態遷移のタイミングで検証を行う必要があるが，複雑なアプリケーションや多数のユーザが協業するコンカレントシステムだと，これらのオーバーヘッドは避けられない．仕様理解やテスト等のステージでアプリケーションを Run させることにより性質の検証を網羅的に行い，性能が要求されるステージでは特に重要な性質のみに絞って検証するようにすれば，性能の問題を回避できると考えている．

検証のタイミングでブロックが発生することに関しては、例えば，集合演算を用いたデータベース管理システムの諸機能を，単一エンティティとそれに関連するエンティティに限定して操作するメカニズム，つまり

$\forall r \in Record \circ r.p$

という性質ではなく，

$\forall r \in Record \circ r.id = id \rightarrow r.p$

なる性質の検証を基本的に行うようなメカニズムを提供するようにすれば，性能が向上すると考えられる．

4.4. 性質の認識と記述

一般的にモデル検査は設計を検証するために行われており，既に製造された，あるいは製造中のシステムに対して検証を行うには，別途検証用のモデルを作成する必要がある．本稿で説明したような方式は，運用の点で以下のような課題があげられる．

1. 性質をどこまで適切に記述できるのか
2. 仕様としての正しさと想定外の振る舞いに対する対応をどうするか

性質をどこまで適切に記述できるのかという問題は，モデル検査で検証する性質の表現力の問題である．プログラマ目線で記述される検証式は、データ構造を熟知した状態の表現であり、単体テストで実行する検証式を流用し、ステージ間で共有することを想定している．より汎用で一般的な性質の表現の可能性を探る必要がある．

仕様としての正しさと想定外の振る舞いに対する対応は，完全な検証モデルの記述とトレードオフである．本稿では時相論理式と一階述語論理式で与えらえる検証式を対象とする方針を示したが，時相論理で対象にするオペレータは用途が広く，フルスペックの検証器を独自に実装するには実現するためのコストがかかる．さらに効率よくモデル検査を実装する方法し、高い表現力を誇る方式を検討している．

5. 謝辞

本稿執筆にあたって，アドバイスをいただいたソフトバンク株式会社の瀧本和弘氏，関係部署に調整いただいた株式会社テクノネットの原嶋麻衣子氏，および本稿を査読いただいた先生方にはこの場を借りてお礼申し上げます．

参考文献

[1] 中島 震, “ソフトウェア工学のツールとしての形式手法”, *NII Technical Report*, 2007.

[2] 情報処理推進機構, “システム再構築を成功に導くユーザガイド”, *SEC Books*, 2016

[3] 池田 信之, 今村 紀子, 高田 沙都子, “実用化に向けたモデル検査適用手法の開発”, 東芝レビュー, 2007

[4] IPA/SEC, “システムライフサイクルプロセス説明(案)”, システムズエンジニアリング推進 *WG* 参考 *2(17-SEWG-2)*

[5] 岸 知二, “モデル検査技術にょる UML 設計検証”, 情報処理 *2006* 年 *5* 月

[6] 向殿 政男, “信頼性と安全性”, *SEC journal Vol.10 No.3*

[7] Jean-Michel Couvreur, “On-the-fly Verification of Linear Temporal Logic”, *FM’99, Vol. I, LNCS 1708, pp. 253–271*, 1999.

[8] Stefania Gnesi, Franco Mazzanti, “On the fly model checking of communicating UML State Machines”, *http://fmt.isti.cnr.it/WEBPAPER/onthefly-SERA04.pdf*

[9] Ilan Beer, Shoham Ben-David, Avner Landver, “On-The-Fly Model Checking of RCTL Formulas”, *Lecture Notes in Computer Science*, Decembere 1999.

[10] R.Gerth, D.Peled, M.Y.Vardi, P.Wolper, “Simple On-the-fly Automatic Verification of Linear Temporal Logic”, *Proceedings of the 6th Symposium on Logic in Computer Science*