報酬構造を考慮したテストケース生成と信頼性評価の効率性

小澤 公貴
広島大学 大学院工学研究科
ozawa@s.rel.hiroshima-u.ac.jp

土肥 正
広島大学 大学院工学研究院
dohi@rel.hiroshima-u.ac.jp

要旨

本稿では, マルコフ使用モデル (Markov Usage Model) とシステムの運用プロファイル (Operational Profile) に基づいてテストケースを自動生成するための統計的方法について考察する. 特にソフトウェアの設計情報を報酬構造として導入することでマルコフ報酬過程を定義し, ソフトウェアテストの効率化とソフトウェアの信頼性評価尺度の算出を行う．最終的に, 実際のソフトウェアプログラムにフォールトを埋め込んでテスト実験を行った結果，ソフトウェアメトリクスを報酬として与えたテストケース生成アルゴリズムは, 従来の報酬なしのアルゴリズムと比較して数多くのフォールトを検出でき，しかも規定の信頼度に達する為に必要とされるテストケース数を効率的に削減できることが示される．

1. はじめに

ソフトウェア開発は主に,「要件定義」,「基本設計」,「詳細設計」「コーディング」,「テスト」の各工程から構成されている. ソフトウェアテストはテスト工程で行われ，ソフトウェア内に潜在するフォールトの発見・除去を通じて，要求される機能を満たしているかどうかを確認する．よって，テスト工程は製品の信頼性・品質を保証するために必要不可欠であり, 他の工程と比較して最も多くのコストがかかることが知られている. 特に, 大規模・複雑化するソフトウェア開発の現状において，より安価でかつ効果的なテストを実施することでテストの質を向上させることが要求されている.

一般に, プログラムを実際に実行するテストを動的テストと呼び，コードレビューのように実行を伴わないテストを静的テストと呼ぶ. 動的テストを行う際，何も判断基準を持たずにプログラムを実行するだけでは入出力の正誤判定が行えないため，ソフトウェアへの入力と正しい期待出力の組から構成されるテストケースを投入することで入出力の正誤判定を行う. 特別に大規模なソフトウェアでなくても, この入出力関係を規定する組合わせ数は膨大となり, ソフトウェアの全ての実行パスを網羅するテストケースを生成することは現実的に不可能である．よって動的テストでは，テストケースの効率的な自動生成が重要な問題となっている.

動的テストは，さらにホワイトボックステストとブラックボックステストに大別される．ホワイトボックステストとは，プログラムの内部構造を理解した上で，それら一つ一つが意図した通りに動作しているかを検証するテスト技法である．ホワイトボックステストにおけるテストケースは，命令の網羅性や判定条件の網羅性などの内部構造を注意深く検査しながら設計されるため, テストケースの生成には非常にコストがかかる．一方, ブラックボックステストはプログラムの内部構造には着目せず, 与えられた入力に対して正しい出力がなされたかどうかの確認のみを行うテスト技法である．ブラックボックステストに基づいたテストケースは，同値分割や限界値分析を通じて, 入力データを考慮しながら設計される.

ブラックボックステストの興味深い研究テーマとして, 設計情報をテストケース生成に活用するモデルベーステスト (Model-based Test; MBT) が挙げられる. これは, テスト対象となるソフトウェアの振る舞いや機能を抽象的モデルを用いて表現し，モデル上で表現されたソフトウェア・アーキテクチャの情報をテストケース生成に利用するものである．このような形式的モデルを用いることの利点として, 設計されたテストケースがモデル上で規定された各状態や遷移をどれだけ網羅できたのかを明確にすることが挙げられる．形式的モデルの例として,

UML (Unified Modeling Laungage) などの設計記述言語を用いるものや, 木構造や有限状態機械 [3],[5] などのグラフモデルを利用するもの, マルコフ連鎖 [16], [17] や確率オートマトンなどの確率モデルを用いるものが代表的である.

確率モデルを用いたテストケース生成について，今富ら [7] は，MBT の一種でありソフトウェアの運用プロファイル（Operational Profile）を用いてテストケースを生成する運用プロファイルベーステスト (Operational Profile-based Test; OPBT) に着目した．MBT がテスト網羅性のみを志向してテストケースを設計するのに対し，OPBT では運用プロファイルを用いることによりテスト網羅性とユーザの使用環境を同時に考慮してテストケースを設計する．本稿では，今富ら [7] の方法を拡張し，規定のソフトウェア信頼度に到達したらテストを打ち切る方策の下で，報酬構造を考慮したテストケース生成法の有効性を評価する．

2. 関連研究

テストケース生成法に関する従来研究として，Whittaker and Thomason [17] により提案され, 後に Avritzer and Weyuker [1], Gutjahr [6], Kallepalli and Tian [8], Poore ら [13], Whittaker ら [18] によって精緻化されたマルコフ連鎖に基づく統計的テストケース自動生成法がある．これは，ソフトウェアの機能や構造に着目した振る舞いを確率モデルによって表現し, それをシミュレートすることでテストケースの自動生成を可能にしている. プログラムの挙動をモデル化するための表現能力に関して, 有限状態機械 [3],[5] やマルコフ性を持たないより一般的な確率モデル [4] など他にも優れた方法が存在するが, パラメータの物理的性質の同定や統計的推論の簡便性から, マルコフ連鎖に基づく方法は最も汎用性が高いと言える.

テストケースの自動生成とは離れて, マルコフ連鎖に基づく統計的方法では, テストケース投入後のフォールト検出に関する実績データからソフトウェアの信頼性を定量的に評価できることが特徴として挙げられる. Avritzer and Weyuker [1], Gutjahr [6], Kallepalli and Tian [8], Prowell and Poore [14], Weyuker [15], Whittaker ら [18] はプログラムの振る舞いを表すマルコフ連鎖の状態推移図に故障状態を導入し, テストケース実行中にフォールトが検出される確率を推定し, さらにプログラムを任意回数実行中に障害に至る確率を推定する方法について議論している. これより, プログラムの実行中にソフトウェア障害が発生しない確率であるソフトウェア信頼度を, テストケースの投入履歴から統計的に推論することができる.

以上の統計的テストケース自動生成法を踏まえ，今富ら [7] は，従来までの多くの研究ではプログラムの振る舞いを表現するためにモデルの表現能力の向上だけに注力しており, ソフトウェアの設計情報や各モジュールを構成するソースコードの定量的性質に着目していないことを指摘した．そこで，パフォーマビリティ [12] もしくは報酬過程の概念を導入することで, より集中してテストを行うべきモジュールをソフトウェアメトリクスによって特徴づけた上で，テストケースの効率的な生成方法を提案している．最近 Avritzer ら [2] は, この報酬過程の概念を導入し，実在するミッションクリティカルなシステムに対するテストケースの自動生成問題を取扱い，ソフトウェア信頼度の推定問題を考察している.

一方で, Avritzer ら [2] の論文は連続時間マルコフ連鎖 (Continuous-time Markov Chain; CTMC) の過渡解析の結果を用いており, 連続時間で駆動するシステムに対するテストケース生成と信頼度推定に主眼を置いている．今富ら [7] はテスト工程においてより取扱いの簡便な離散時間マルコフ連鎖 (Discrete-time Markov Chain; DTMC) を仮定し, さらに Avritzer ら [2] が論文中に明示的に述べていなかった報酬の決め方について実証的なアプローチを展開している．すなわち，ソフトウェアの各モジュールを構成するプログラムを, ソフトウェア設計メトリクスを用いて報酬の形で与えることを提案している．これにより, プログラムの振る舞いを表す形式モデルは離散時間マルコフ報酬過程 (Discrete-time Markov Reward Process; DTMRP) として表現される.

本稿では今富ら [7] の手法をもとに，単にテストケースを生成するだけでなく，ソフトウェア信頼度の定量的な推定も併せて行う．最終的に, 実際のソフトウェアプログラムにフォールトを埋め込んでテスト実験を行った結果, ソフトウェアメトリクスを報酬として与えたテストケース生成アルゴリズムは，従来の報酬なしのアルゴリズムと比較して数多くのフォールトを検出でき，しかも規定の信頼度に達するために必要とされるテストケース数を効率的に削減できることを示す.

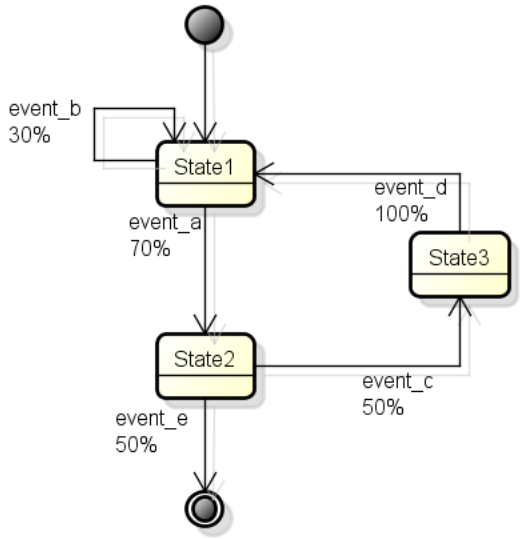


図 1. 有限状態機械によるモデル表現の一例.

3. 運用プロファイルベーステスト

運用プロファイルとは, 開発対象であるソフトウェアが実際の運用環境においてどのように使用されるかを規定した仕様であり, ソフトウェアの各機能や命令を使用する頻度を予め想定することでユーザの運用環境を予測するために用いられる. Musa [10], [11] は, ソフトウェアの各機能の使用確率を要求仕様段階において仮定することで, ユーザの運用環境を考慮した設計, テスト, 検証を行うべきであることを主張している. 運用プロファイルの定量的な取扱いでは, ソフトウェアにおける各機能の使用確率 (Usage Probability) を過去のバージョンの使用履歴や類似ソフトウェアの開発データから推定したり，ユーザの使用振る舞いを開発者が主観的に見積もることにより割当てられる．この使用確率は有限状態機械の推移確率に対応しており, 運用プロファイルを規定することで形式的モデルの振る舞いが定量的に表現できる. 運用プロファイルを MBT に適用するテストである OPBT は, 従来の MBT で作成したモデルに対して, 運用プロファイルから各状態の遷移確率を付加することでユーザの運用環境を記述した動的モデルであると解釈できる.

図 1 に, 有限状態機械によって表現されたソフトウェアの振る舞い図の例を示す．ここで, 図中のノードはソフトウェア機能の種類を表す状態を意味し，アークは状態間の遷移をそれぞれ表している．各遷移に付加された event は遷移の発火条件を表し，黒丸は実行開始状態，二重丸は実行終了状態をそれぞれ表す．図中のアーク上に割当てられた数値は状態推移確率を表しており, 例えば “State2” に対応するソフトウェア機能を実行した後, 確率 50% で “event_e” が生じるとプロセスの実行は終了し, 確率 50% で “event_c” が生じると状態は “State3” に移行し, さらに 100% の確率で状態は “State1” に移る. テストケースの生成は, この状態機械を実行（シミュレート）することで実現される. すなわち, 実行開始状態から実行終了状態に至るサンプルパスをモンテカルロ・シミュレーションを通じて生成し, 各テストケースを構成する入出力値を定める.

4. マルコフ報酬過程

4.1. 離散時間マルコフ連鎖

本論文では, 有限状態機械のサブクラスである DTMC を用いる. 状態の表現能力に関してはオートマトンなど他にも有力な有限状態機械が存在するが, マルコフ連鎖を用いる利点としてモデルパラメータの物理的意味が明確であり，かつ実際のデータからパラメータを統計的に推定することが容易である点が挙げられる．また, ソフトウェア信頼度に代表される信頼性評価尺度を算出するためには, 必然的に解析的取扱いが可能な確率過程を適用する必要がある.

今, 離散時刻 $n = 0, 1, 2, \cdots$ において非負値状態空間上で値をとる確率過程 $\{X(n), n = 0, 1, 2, \cdots\}$ を考える．任意の状態 i, j, i_n に対して，

$$\begin{aligned} P\{X(n+1) = j \mid X(0) = i_0, X(1) = i_1, \cdots, \\ X(n-1) = i_{n-1}, X(n) = i\} \\ = P\{X(n+1) = j \mid X(n) = i\} = p_{ij} \end{aligned}$$

のような関係が成立するならば, $X(n)$ はマルコフ性を有し, DTMC に従うと言う．ここで状態 i から状態 j に推移する条件付き確率 $p_{i,j}$ は推移率と呼ばれる．図 1 の実行開始状態を除いた 4 つの状態の内，実行終了状態のようにこれ以上推移が起こらない状態を吸収状態, 以降も推移可能な状態を過渡状態と呼ぶ. 過渡状態の数が

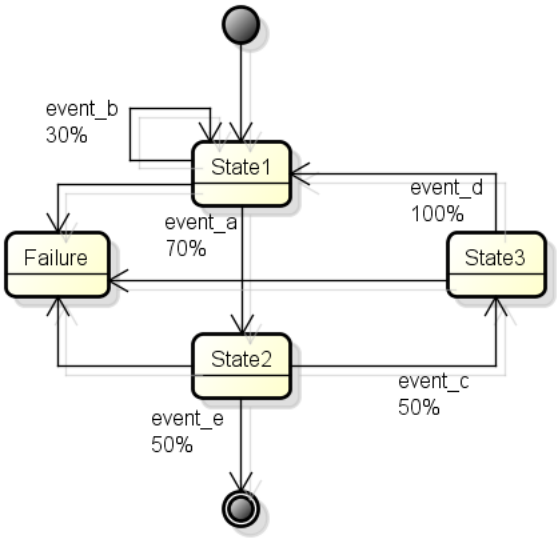


図 2. 障害状態を追加したモデル表現.

m で吸収状態の数が l の DTMC において, 推移確率行列を

$$\mathbf{P} = \left(\begin{array}{c|c} \mathbf{Q} & \mathbf{C} \\ \hline \mathbf{O} & \mathbf{I} \end{array} \right) \tag{1}$$

によって定義する. ここで, $\mathbf{Q}$, $\mathbf{C}$ は, 過渡状態から過渡状態への推移と, 過渡状態から吸収状態への推移を表す部分行列であり, $\mathbf{O}$, $\mathbf{I}$ はそれぞれ零行列と単位行列を表している. よく知られた DTMC の結果より, DTMC の n ステップ推移確率行列は

$$\mathbf{P}^{(n)} = \mathbf{P}^n = \begin{pmatrix} \mathbf{Q}^n & \mathbf{C}_n \\ \mathbf{O} & \mathbf{I} \end{pmatrix} \tag{2}$$

によって与えられる. ここで部分行列 C_n は

$$\mathbf{C}_n = (\mathbf{I} + \mathbf{Q} + \cdots + \mathbf{Q}^{n-1})\mathbf{C} \tag{3}$$

となる. $\mathbf{Q}$ は列ベクトルの総和が 1 となる非負値行列であり,

$$\lim_{n\to\infty} \{\mathbf{I} + \mathbf{Q} + \mathbf{Q}^2 + \cdots + \mathbf{Q}^{n-1}\} = (\mathbf{I} - \mathbf{Q})^{-1} \tag{4}$$

の関係を満足する. この $\mathbf{M} = (\mathbf{I} - \mathbf{Q})^{-1}$ は基本行列 (fundamental matrix) と呼ばれる. テスト対象とするソフトウェアには m 個の機能が備わっており, フォールトが全く含まれていない場合には, 1 回のテスト実行で状態 1 から任意の過渡状態を経由して実行終了状態である吸収状態に至る. もし各機能を実現するモジュールにフォールトが潜在的に含まれるならば, 各状態からもうひとつの吸収状態である “Failure” 状態に推移が発生する. 図 2 は $m = 3, l = 2$ の場合の DTMC の推移図の例を表している. OPBT においてテストケースを生成する場合, 図 1 のようにフォールトの検出を想定することなく, 実行開始状態から実行終了状態までの経路をシミュレートし, 各機能が順に実行される遷移系列を求める. 求められた遷移系列の入力値と期待出力値を予め用意することで, 当該テストケースによる正誤判定を行うことができる. 十分なテストケースが投入された後, 図 2 の過渡状態 j への累積訪問回数を s_j $(j = 1, 2, \cdots, m)$, 各過渡状態においてフォールトが検出された回数を f_j とすれば, モジュール j の信頼度は

$$R_j = 1 - (f_j/s_j) \tag{5}$$

によって求められる. よって, 任意回数のテスト実行でフォールトが検出されることなく終了する確率であるソフトウェア信頼度を, 基本行列の要素から容易に算出することができる. これにより, DTMC を用いてテストケースの自動生成とソフトウェア信頼度の推定を同時に行うことが可能となる.

4.2. マルコフ報酬過程とソフトウェアメトリクス

先に述べた DTMC に基づいた OPBT の問題点として, 仕様の段階で類推される運用プロファイル（DTMC における過渡状態間の推移確率）を予め正確に知ることが困難である点が挙げられる. 一般に, 不特定多数のユーザがどの機能をどのような頻度で使用するかを正確に予測することは容易でなく, 運用プロファイルを利用した定量化技術にも自ずから限界がある. また, 運用プロファイルはあくまでユーザの動作環境を想定したものであり, ユーザによるソフトウェアの使用環境を正確にモデル化できたとしても, 必ずしも潜在するフォールトが多数検出されるテストケースを生成できる保証はない. 換言すれば, 頻繁に使用される機能を実現したモジュールに常に多くのフォールトが含まれているわけではないため, なるべく多くのフォールトを検出したいというテスト効率の観点から通常の OPBT が適しているとは言えない.

ここでは, DTMC の各状態を訪問する度に r_i $(i = 1, 2, \cdots, m)$ の報酬が与えられるものとし, DTMC の拡張である DTMRP を導入する．マルコフ報酬過程はパフォーマビリティの概念と共に Meyer [12] によって導入され, 最近 Avritzer ら [2] によってテストケースの自動生成に応用されている．テストにおいて探索すべき（フォールトプローンな）モジュールが存在し, その探索優先順位が予め定められているならば, 優先順位ごとに大きくなる報酬値を割振り, 1 回のテストケースの実行が終了するまでの単位ステップ当たりの累積報酬を求め, その値が大きいテストケースから順に実行すればよい．しかしながら, 問題は各モジュールをテストする際に割与えられる報酬値をどのように決定するかにある．最も簡便な方法は, 単体テストにおいてフォールトプローンモジュールの相対順位を決定し, 優先順位の低い（フォールトがより多く含まれない傾向にある）モジュールから, $1, 2, \cdots$ のような整数値を割振ることであろう．しかし, 単体テストの成果として類推されたフォールトプローンモジュールの相対順位さえも, 必ずしもシステムテストにおいて重点的に探索されるべきモジュールであるかどうか定かではない．

本稿では, ソフトウェアの各モジュールを特徴付ける報酬値として，ソフトウェアの複雑性メトリクスの相対順位を用いることを提案する．複雑性メトリクスとは, プログラムを構成するソースコードから計測される形式的構造の複雑性を表す評価尺度である．本研究では単純なコード行数， Halstead によるソフトウェアサイエンス理論 (例えば [19] を参照) に基づいたボリュームメトリクス，および McCabe [9] によるサイクロマチック数を採用した．コード行数は, 開発規模の見積もり方法として最も計測が簡単な尺度である．一方, ソフトウェアサイエンス理論とは, 複雑性の尺度をオペレータおよびオペランドの数による関数によって定義する複雑性メトリクスの枠組みである．ここで, オペレータとは算術演算子, 比較演算子, 理論演算子, 関数名, コマンド名, 句読点, 区切り符号などで構成され，オペランドとは変数, 定数, ラベルなどを含む．オペレータとオペランドの種類および総出現回数を計測することにより，プログラム開発に要する時間やプログラムがメモリ上に占める大きさなどの指標が計算できる．

Halstead によるソフトウェアサイエンス理論における基本的尺度は以下の 4 つである．

- n_1 : プログラム内のオペレータの種類数
- n_2 : プログラム内のオペランドの種類数
- N_1 : プログラム内でのオペレータの総出現数
- N_2 : プログラム内でのオペランドの総出現数

これらはプログラムのソースコードから容易に計測可能であり, これら 4 つの基本的尺度を用いてプログラムのボリュームメトリクスを算出する．

報酬値として用いる評価尺度の例を表 1 に示す．表中のプログラムの大きさ V^* は予測値であり，正確な値が求められないため, 本稿では

- プログラムの長さ N
- プログラムの大きさ V
- プログラムの水準の推定値 $\hat{L}$
- プログラミングの困難さの推定値 $\hat{D}$
- プログラミング労力の推定値 $\hat{E}$
- プログラミング言語水準の推定値 $\hat{\lambda}$

の 6 つを用いた．McCabe によるサイクロマチック数は，プログラムの複雑度をソースコードから測定する手法である．ソースコードから制御フローを有向グラフで表現することで，グラフを網羅するパスの数がサイクロマチック数として算出される．制御フローグラフが e 個の枝，n 個の節点を持つとき，サイクロマチック数 M は

$$M = e - n + 1 \tag{6}$$

と求めることができる．各モジュールごとにこれら 6 つのボリュームメトリクスもしくは複雑性メトリクスを算出し, 優先順位の低い (メトリクスの値が小さい) 順に $1, 2, \cdots$ の整数値を割振った相対順位を報酬として与える．DTMC に基づいて生成された各テストケースごとに単位ステップ当たりの累積報酬を計算し, その値が大きいものから優先的にテストに使用する．

5. 実験

ここでは実際のプログラムに予めフォールトを挿入し, 生成されたテストケースに基づいてフォールトを発見・除去する実験を行うことで報酬に基づいたテストケース

表 1. ソフトウェアサイエンス理論における評価尺度 (文献 [19] から抜粋).

分類	尺度	公式
プログラムの長さ	プログラムの長さ	$N = N_1 + N_2$
	プログラムの長さ N の推定値	$\hat{N} = N_1 + N_2$
プログラムの大きさ	プログラムを実現したときの大きさを記述に必要とされるビット数により表したもの	$V = N\log_2(n_1 + n_2)$
	アルゴリズムを表現したときの最小の大きさ	$V^* = (2n_2^*)\log_2(2 + n_2^*)$ $(n_2^*) =$ アルゴリズムの入出力パラメータ数
プログラムの水準	プログラムの水準を実現された大きさ V と最も簡潔な大きさ V^* との比で表したもの	$L = V^*/V$
	プログラムの水準 L の推定値	$\hat{L} = 2n_2/(n_1N_1)$
プログラミングの困難さ	プログラミングをするのに伴う困難さ	$D = 1/L$
	プログラミングの困難さ D の推定値	$\hat{D} = 1/\hat{L} = 2n_2/(n_1N_1)$
プログラミング労力	アルゴリズムを実現するときの理解に必要とされる労力	$E = V/L = V^2/V^*$
	プログラミング労力 E の推定値	$\hat{E} = V/\hat{L}$
プログラミング言語水準	使用するプログラミング言語の言語水準	$\lambda = LV^*$
	言語水準 λ の推定値	$\hat{\lambda} = \hat{L}^2V$

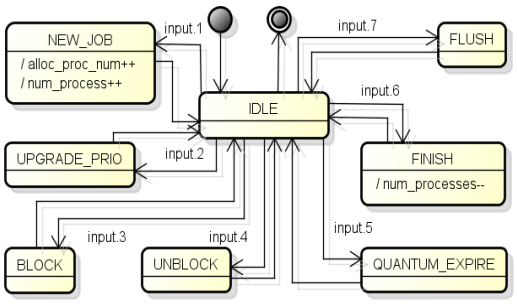


図 3. ‘schedule’ の状態遷移図.

生成法の性能評価を行う. 実験対象とするソフトウェアは ‘schedule’ と呼ばれるプロセス管理ソフトウェア [1] であり, C 言語で記述され, コード行数は約 400 行, 18 の関数から構成されているプログラムである. このプログラムを起動するとまず待機状態になり, 次に入力を受け付ける. 入力された値によって各機能を実行し，機能実行が終了すると再び待機状態となる. 終了命令が入力されるまで上述の動作を繰り返す. テスト対象ソフトウェアの振る舞いを有限状態機械で表したものを図 3 に示す. プロセスの開始, 終了, 優先度の変更など 7 つの機能によって状態が構成されていることが分かる. このソフトウェアに実際のテスト工程で発見された 7 つのフォールトを事前に埋め込み，用意されたテストケースを実行することで実際のソフトウェアテスト環境を再現した. ここでフォールトはプログラム中の 7 箇所に対応しており，1 つの故障が複数箇所の原因から引き起こされた場合，原因箇所それぞれが発見されたフォールトとしてカウントされる. 本稿では以下のように運用プロファイルを想定し，テスト実験を行った.

- DTMC における状態推移確率を確率の公理を満たすようランダム (一様乱数) に設定:
$$\begin{pmatrix} p_{12} = 0.13 & p_{13} = 0.14 & p_{14} = 0.15 & p_{15} = 0.11 \\ p_{16} = 0.13 & p_{17} = 0.15 & p_{18} = 0.10 & p_{1end} = 0.09 \end{pmatrix}$$

ここで, 各モジュール IDLE, NEW_JOB, UPGRADE_PRIO, BLOCK, UNBLOCK, QUANTUM_EXPIRE, FINISH, FLUSH にそれぞれ 1 ∼ 8 までの状態番号を割当て, 推移率 $p_{1,j}$ $(j = 2, \cdots, 8)$ と $p_{1,end}$ を与えている. また, $p_{j,1}$ $(j = 2, \cdots, 8)$ および

[1] http://sir.unl.edu/content/sir.php

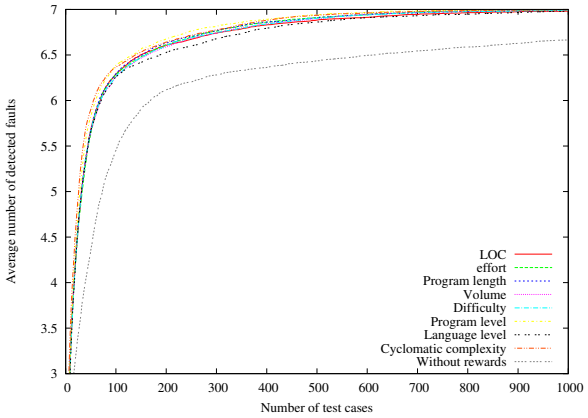
$p_{end,1}$ に関しては

$$\begin{pmatrix} p_{21}=1 & p_{31}=1 & p_{41}=1 & p_{51}=1 \\ p_{61}=1 & p_{71}=1 & p_{81}=1 & p_{end1}=0 \end{pmatrix}$$

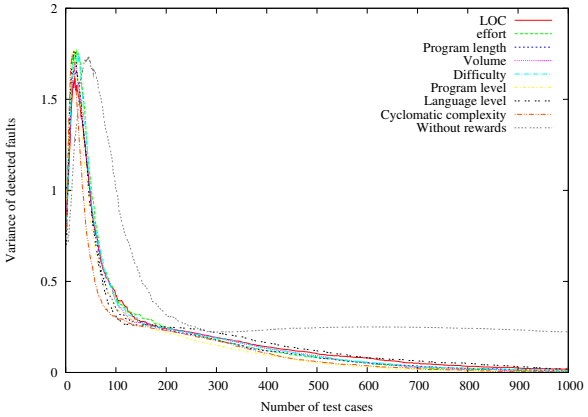
のように設定した．一旦運用プロファイルが定まると，次にソフトウェア設計メトリクスに基づいて報酬としての相対順位を決定する．表 1 において，プログラムの水準およびプログラミング言語水準に関しては，それらのメトリクス値が大きいものほどフォールトは検出されにくいと考えられる。そのため各機能のメトリクス値を算出したのち逆数をとったものを用いることで，フォールトが潜在すると想定されるモジュールをテストするためのテストケース生成を行った。各テストにおいて，実行開始状態から実行終了状態に至るステップ数（推移の回数）と，その間に訪問するモジュールのメトリクス値に対する相対順位の累積値をカウントする．次に相対順位の累積値をステップ数で割ることで, 単位ステップ当りの報酬を求める．DTMRP を 10,000 回シミュレートすることで 10,000 本の状態遷移列をテストケースとして生成し，全てのテストケースの中で単位ステップ当たりの報酬値が高い順に並び換えた 1,000 本のテストケースを，1 つのテストスイートとする．この実験では上述のテストスイートを独立に 1,000 本用意し，それらを順次実行することで，1,000 本のフォールト検出過程を観測する．各テストケース実行中にフォールトが検出されると, それを修正した後に同じテストケースが再度実行され, 終了するまで繰り返される．ここで，フォールトの修正によってメトリクス値が変化することが考えられるが，本実験では修正による報酬値の変化を考慮しないものとする．

図 4 は，想定した運用プロファイルを仮定した 1,000 本のテストスイートに対して，発見フォールト数の平均値と分散を消化テストケース数ごとにプロットしたものである．発見フォールト数の平均値はテストケースが消化されるにつれて指数関数的に増加し，潜在フォールト数の値である 7 に近づく傾向がある．しかしながら，報酬を伴わない方法では，報酬を伴う場合と比較して平均発見フォールト数が若干少なくなっている．分散に関しては，テスト初期段階で大きくなり，その後で徐々に減少する傾向を読み取ることができる．報酬を伴わない場合の分散は他と比較して大きくなり，その結果かなりのバラツキが見える．これより，何らかの報酬に基づいてテストケースを自動生成する方が，多くのフォールトを

図 4. 各テストケース生成アルゴリズムに基づいて算出された累積発見フォールト数の比較.



(i) 平均値.



(ii) 分散.

早い段階で安定して検出することが可能であることが分かった．

次に, より詳細に報酬値の種類に応じたテスト効率の比較を行う．表 2 はテスト効率の比較結果を示しており，1,000 本のテストスイートの中ですべてのフォールトを発見・除去することができたテストスイート数と，各テストスイート内で 7 つすべてのフォールトを発見・除去するまでに要したテストケース数の最小値, 平均, 分散をそれぞれ求めたものである．これより，報酬を伴わない方法で生成したテストスイートでは，全体の 67% しかすべてのフォールトを発見・除去することができなかった．一方，報酬を伴った方法で生成したテストスイートではすべて 98% 以上となり，報酬としてプログラム水準を導入した場合はすべてのフォールトを検出できたテ

表 2. 報酬値の種類に応じたテスト効率の比較.

	全フォールトを発見したテストスイート数	全フォールトを発見できなかったテストスイート数	全フォールトを発見するまでのテストケース数の統計量		
			最小	平均	分散
報酬なし	668	332	2	408.8	73343.3
コード行数	982	18	8	215.0	38800.7
プログラミング労力	994	6	3	205.2	33069.0
プログラムの長さ	994	6	6	204.2	32846.2
プログラムの大きさ	995	5	5	216.9	34648.8
プログラムの水準	998	2	4	179.0	26386.6
プログラミングの困難さ	990	10	4	212.2	34879.7
プログラミング言語水準	980	20	7	244.0	43767.7
サイクロマチック数	997	3	4	192.3	29035.1

ストスイート数は 98.8% となった．さらに，この傾向は 7 つすべてのフォールトを発見・除去するまでに要したテストケース数に対しても同様に観測され，特にプログラム水準をメトリクスとして用いた場合にテストケース数が最小となっている．以上の結果より，どのメトリクスを報酬値として用いた場合でも，報酬値を用いないテストケースよりフォールト検出力は格段に向上することが分かった．

図 5 は，生成したテストケースを用いてソフトウェアテストを実行する際に算出されるソフトウェア信頼度の平均値と分散を消化テストケースごとにプロットしたものである．信頼度の平均値はテストケース消化と共にテストの初期段階から大きく増加し，上限である 100 に近づいてゆくことがわかる．しかし，報酬を伴わない方法では報酬を伴うテストケース生成法と比較して，平均ソフトウェア信頼度の値が小さいことが示される．ソフトウェア信頼度の分散では，テスト開始時点では分散値は極端に大きいが，テストの進捗が進むにつれて 0 に収束する様子を確認できる．分散に関する結果から，報酬を用いないテストケース生成法は，報酬を用いた場合と比較して，バラツキが収束するまでにより多くのテストケース数が必要であることが示されている．この結果より，いずれかの報酬を用いてテストケースを生成することにより，テストの早期からフォールトを効率的に検出することが可能であり，ソフトウェア信頼度の値が早い段階で安定することが分かる．

次に，ソフトウェア信頼度が規定の値に達した時点でソフトウェアテストを打ち切る状況を考える．表 3 は目標信頼度を 99%とした場合に，目標を達成するのに要したテストケース数の平均と分散を比較したものであり，目標を達成した時点での発見フォールト数の平均と分散をそれぞれ求めた結果である．ソフトウェア信頼度が 99%に達するまでに必要なテストケース数について，いずれかのメトリクスを報酬として用いた場合の方が，用いない場合と比較して平均と分散が共に小さいことが確認された．中でも，サイクロマチック数を報酬として用いた場合，平均と分散が共に最小となり，目標信頼度に到達するためにはサイクロマチック数を報酬として与えることでテストケースを生成すれば良いという結論を得た．一方，ソフトウェア信頼度が 99%に達成した時点で最も多くのフォールトが発見できているメトリクスはプログラムの水準であった．しかし，99%の目標水準に達した時点での平均発見フォールト数は，どのメトリクスを用いたとしてもほぼ全てのフォールトを検出できているため，メトリクスの種類による違いはほとんどないと言ってよい．また，報酬なしの場合のテストケース生成法で発見された平均発見フォールト数は，報酬を用いたものと比較すると若干小さいことが分かる．本稿で提案した信頼度基準に基づいてテストの打ち切りを行う場合，どのメトリクスを報酬として用いたとしてもフォールト検出に有効なテストケースを生成できることが分かる．

詳細な実験結果は割愛するが，DTMC における状態推移確率を変更した実験において，全ての場合において

表 3. 信頼度基準に基づいたテスト効率の比較.

	信頼度が 99% を超えるまでにに要したテストケース数		信頼度が 99% を超えた時点での発見フォールト数	
	平均	分散	平均	分散
報酬なし	645.4	3146.3	6.5	0.2
コード行数	344.0	641.1	6.8	0.2
プログラミング労力	343.1	464.7	6.8	0.2
プログラムの長さ	375.9	517.2	6.8	0.1
プログラムの大きさ	344.6	628.7	6.8	0.2
プログラムの水準	402.6	542.9	6.8	0.2
プログラミングの困難さ	342.2	492.0	6.9	0.1
プログラミング言語水準	352.7	970.6	6.7	0.2
サイクロマチック数	286.1	368.9	6.7	0.2

報酬なしと比較した結果，報酬を与えたテストケース生成が良好な結果をもたらすことが示された．その中でも，プログラム水準とサイクロマチック数を報酬として用いた場合が，概ね良い結果を示していることが観測できた．

6. おわりに

本稿では OPBT に報酬構造を導入することで，より効率的なテストケースの自動生成が可能であることを示した．特に，報酬として与えたボリュームメトリクスおよび複雑性メトリクスの中でも，プログラム水準とサイクロマチック数の意味で，複雑度の高いモジュールを優先的にテストするようなテストケースを生成することが効率性の観点から好ましいことが分かった．今回得られた結果は一つのプログラムに対してのみ実験を行ったものであるため，実験結果が他の種類のソフトウェアに対しても成立するか調べる必要がある．よって，どのような報酬構造がテストケースの効率性に寄与するかを包括的に調べるためには，今後さらに他の種類のプログラムを用いたテスト実験を行う必要がある．また，テストケースの自動生成を通じてソフトウェア信頼度を推定することは，テスト技術者がテスト進捗状況を逐次把握するためには有益な情報に繋がるものと考えられる．その際，ソフトウェア信頼性だけでなく，期待報酬に基づいた信頼性評価尺度を求めることにより，異なる視点からテスト終了後の品質評価を行うことは興味深いものと思われる．

参考文献

[1] A. Avritzer and E. J. Weyuker, The automatic generation of load test suites and the assessment of the resulting software, *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp.705–716 (1995).

[2] A. Avritzer, E. de Souza e Silva, R. M. M. Leao and E. J. Weyuker, Automated generation of test cases using a performability model, *IET Software*, vol. 5, no. 2, pp.113–119 (2011).

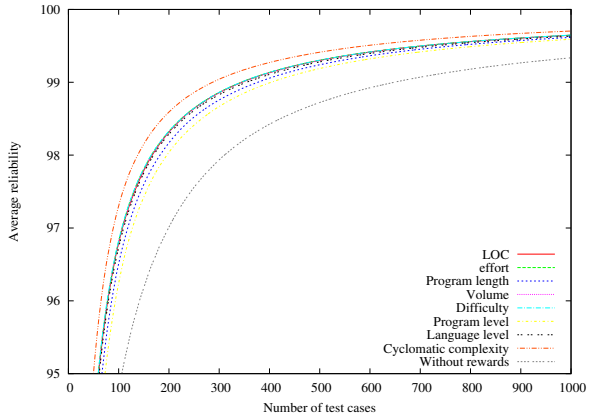
[3] T. S. Chow, Testing design modeled by finite-state machines, *IEEE Transactions on Software Engineering*, vol. 4, no. 3, pp.178–187 (1978).

[4] K. Doerner and W. J. Gutjahr, Representation and optimization of software usage models with non-Markovian state transitions, *Information and Software Technology*, vol. 42, no. 12, pp. 815–824 (2000).

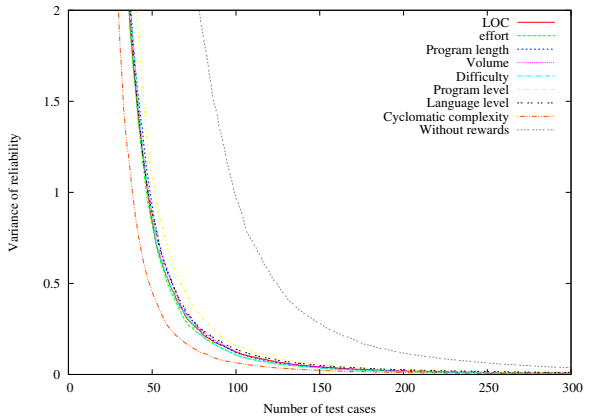
[5] S. Fujiwara, G. V. Bochmann, F. Khendek, M. Amalou and A. Chedamsi, Test selection based on finite state models, *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp.591–603 (1991).

[6] W. J. Gutjahr, Software dependability evaluation based on Markov usage models, *Performance Evaluation*, vol. 402, pp. 199–222 (2000).

図 5. 報酬値の種類に応じた信頼度のモーメント比較.



(i) 平均値.



(ii) 分散.

[7] 今富政喜，土肥正,“運用プロファイルと報酬構造を考慮したテストケース生成法,” ソフトウェアシンポジウム ’14 論文集, ソフトウェア技術者協会, 10 pages, 秋田 (June 9–11, 2014).

[8] C. Kallepalli and J. Tian, Measuring and modeling usage and reliability for statistical web testing, *IEEE Transactions on Software Engineering*, vol. 27, no. 11, pp. 1023–1036 (2001).

[9] T. J. McCabe, A complexity measure, *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308-320 (1976).

[10] J. D. Musa, Operational profiles in software-reliability engineering, *IEEE Software*, vol. 10, no. 2, pp. 14–32 (1993).

[11] J. D. Musa, Software-reliability-engineered testing, *IEEE Computer*, vol. 29, no. 11, pp. 61–68 (1996).

[12] J. Meyer, On evaluating the performability of degradable computing systems, *IEEE Transactions on Computer*, vol. C-29, no. 8, pp. 720–731 (1980).

[13] J. H. Poore, G. H. Walton and J. A. Whittaker, A constraint-based approach to the representation of software usage models, *Information and Software Technology*, vol. 42, pp. 825–833 (2000).

[14] S. J. Prowell and J. H. Poore, Computing system relaibility uising Markov chain usage models, *Journal of Systems and Software*, vol. 73, pp. 219–225 (2004).

[15] E. J. Weyuker, Using failure cost information for testing and reliability assessment, *ACM Transactions on Software Engineering and Methodology*, vol. 5, pp. 87–98 (1996).

[16] J. A. Whittaker and J. H. Poore, Markov analysis of software specifications, *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 1, pp. 93–106 (1993).

[17] J. A. Whittaker and M. G. Thomason, A Markov chain model for statistical software testing, *IEEE Transactions on Software Engineering*, vol. 20, no. 10, pp. 812–824 (1994).

[18] J. A. Whittaker, K. Rekab and M. G. Thomason, A Markov chain model for predicting the reliability of multi-build software, *Information and Software Technology*, vol. 42, pp. 889–894 (2000).

[19] 山田茂，高橋宗雄，ソフトウェアマネジメントモデル入門－ソフトウェア品質の可視化と評価法－，共立出版，東京，1993.