状態遷移モデルの一部が明示されていない場合における自動テスト生成手法

Automatic test generation method for case where parts of the state transition model are not explicitly specified

パナソニック ITS 株式会社
Panasonic ITS Co., LTD
○松尾　正裕[1]
○Masahiro Matsuo

Abstract In a previous paper, we proposed an automatic test generation method for state transition testing that efficiently detects defects by inserting default commands before and after each command sequence that serves as a test case.

This method generates command sequences by covering all combinations, which causes combinatorial explosion when applied to complex state-transition models, making it difficult to apply the method to actual development.

In this paper, we define the coverage of combinations in a command subsequence and propose a new method to generate a small number of test cases that can find defects by optimizing for the coverage. We also confirm that the proposed method detects defects more efficiently than randomly generated tests.

1. はじめに

近年，様々な分野においてソフトウェア開発における開発費が増加しており，これはソフトウェアの大規模化を意味している．また，市場の変化やユーザのニーズに対応するために，ソフトウェアを継続して開発し続けることが必須となっており，度重なる仕様変更への対応や短い期間でリリースすることが求められている．このような環境において，品質保証のためのテスト自動化は必須であり，テストの実行を自動化することは広く議論され，現場でも導入・実施が進んでいる．一方，意図に合ったテストの生成を自動化することは，研究として非常に盛んに行われている[1]が，現場に行き渡っているとは言えない．

本研究では，実装の修正による影響を受けにくく，重大な不具合を発見することができる自動テスト生成手法を検討した．開発に与える影響が大きい不具合として，状態遷移不具合が挙げられる．状態遷移不具合は状態ロックや操作ロックに直結するため，高確率で他機能の動作にも影響を及ぼす．特に，分業開発を行う場合，機能単位で分業することが多く，不具合が他機能に影響を及ぼすことは他社の開発にも影響を及ぼすことを意味するため，問題はより深刻である．また，ソフトウェア開発には，機能開発だけではなく，横串となる共通機能（フレームワークやライブラリ）の開発も含まれる．このような開発対象の特徴に合わせた自動テスト生成手法を検討し，実験を行うことで状態遷移不具合に対する有効性を確認した．

本論文の構成は以下の通りである．2 章では前述の課題認識に関して，本研究の背景や動機についてまとめる．その点を踏まえて，3 章，4 章では従来の研究結果を踏まえた本研究における研究課題と提案手法について述べる．その後，5 章，6 章では提案手法を検証した実験内容と結果について報告する．最後に 7 章，8 章では実験結果に対する考察と今後の展望について議論する．

[1] パナソニック ITS 株式会社　Panasonic ITS Co., LTD
神奈川県横浜市都筑区佐江戸町 600 番 Tel:045-938-1810
600, Saedocho, Tsuzuki ward, Yokohama city, Kanagawa prefecture
【キーワード】自動テスト生成,テスト入力値生成,モデル検査,状態遷移テスト,ランダムテスト

2. 背景・動機

前述のように分業開発を行う場合，特に流用開発では，要求や仕様の変更に合わせて各機能を修正する．特に，ユーザが直接操作する機能は高頻度に，そして大幅に変更されるため，メンテナンスの観点から，設計ドキュメントへの記載は最低限の内容に留めることが多い．一方，共通機能は，修正による影響が複数の機能提供先に及ぶことから，ユーザに近い機能に比べると変更は限定的である．また，共通機能の利用者への情報提供の観点から，設計情報や使用方法を詳細に記載して，設計ドキュメントの記載内容を充実させていることが多い．

本研究では，頻繁に修正されるアプリケーション（以下，アプリ）の品質担保のため，自動テスト生成によりアプリ単体レベルの状態遷移テストを実施する方法を検討した．修正量や修正頻度の多さから状態遷移不具合を混入するリスクが高く，度重なる修正に対するテストのメンテナンスコストも考慮すると，自動テスト生成を導入する効果や効率の観点で有効性が高いと考えられる．ただし，開発者テストの置き換えではなく，追加して実施するテストを想定する．

本研究の対象とするソフトウェア構成を図 1 に示す．アプリ共通のアプリフレームワーク（以下，フレームワーク）と，それを利用するアプリが複数存在する構成とする．本研究における仮定として，アプリの状態遷移モデルは設計情報として存在しているが，設計ドキュメントには明示されておらず，フレームワークの状態遷移モデルは設計ドキュメントに記載されているものとする．このような条件下において，フレームワークまで含めた状態遷移テストを自動生成する手法を検討する．

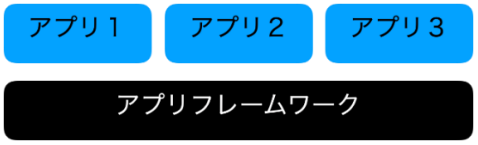

図 1 今回の研究対象とするソフトウェア構成

3. 研究課題・従来研究

3.1. 研究課題

通常の状態遷移テストでは，状態遷移図や状態遷移マトリクスに従ってテストケースを生成する[2]．しかし，本研究では，一部の状態遷移モデルが明示されていないため，状態遷移モデルに従ってテストケースを生成することや，動作の期待値を用いてテスト結果を検証することができない．そこで，従来の我々の研究では，テスト結果の検証方法と妥当なテストケースの生成方法について検討し，解決策を提案した[3]．ただし，従来の提案手法では，関数やイベント（以下，コマンド）の組み合わせを網羅してテストケースを生成するため，実際のソフトウェア開発に適用すると，状態数やコマンドの種類が多いことにより，組み合わせ爆発が起こる懸念がある．そのため，実際のソフトウェア開発に提案手法を導入するためには，生成するテストケースを現実的に実施できる程度まで絞り込む必要がある．

3.2. 従来研究

従来の我々の研究で提案した，テストの成否を判断する方法と有効性の低いテストケースを排除する方法，また，その実験結果[3]について述べる．まず，従来の研究で用いたフレームワークとアプリの状態遷移モデルを図 2，図 3 に示す．ただし，アプリの状態遷移モデルは設計ドキュメントに記載がなく，明示されていないものとして扱う．

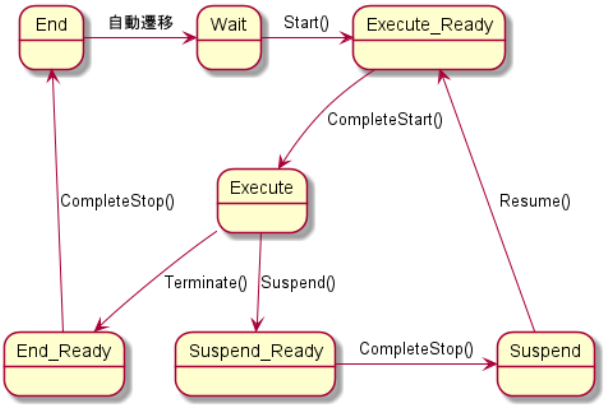

図 2　フレームワークの状態遷移モデル

(1)　テストの成否を判断する方法

状態遷移モデルが明示されていない場合，各テストケースに対して期待値を機械的に記述することは難しい．そこで，常に成り立つべき性質[4]を全テストケース共通のアサーションとして記述する．状態遷移における時系列を記述するために，共通のアサーションの記述には，仕様記述言語のひとつである LTL(Linear Temporal Logic:線形時相論理)を用いる．共通のアサーションを用いて各テストケースを検査することで，正常な状態遷移から逸脱した場合に不具合として検出することができる．従来の研究では，明示されているフレームワークの状態遷移モデルを用いて共通のアサーションを記述した．

また，共通のアサーションに対して予めモデル検査を実施することで，共通のアサーションが

状態遷移モデルにおいて常に成り立つべき性質であることを検証する．ただし，モデル検査では状態遷移が無限に実施されることを想定して検査を行うが，ソフトウェアテストにおける状態遷移は有限である．従って，状態遷移が無限に続くことを前提としているモデル検査における検査手法をそのままソフトウェアテストに適用すると，状態遷移が途中で終了することに起因する，不具合の誤検出が発生する．その対策として，検査に影響のない状態でテストを終了させるために，テストケースの最後にコマンドを追加する．

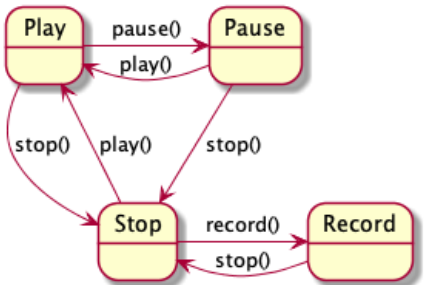

図 3　アプリの状態遷移モデル

(2)　有効性の低いテストケースを排除する方法

実装などからコマンドを抽出し，テスト対象への入力とすることは可能である．抽出したコマンドの組み合わせを網羅してテストケースを生成する場合，実際に状態遷移するコマンド入力は僅かであり，状態遷移しないコマンド入力を大量に実施することになる．そのため，状態遷移することで不具合を検出するという狙いに対しては，非効率なテストを実施することになるという課題がある．

通常の状態遷移テストでは，状態遷移モデルを用いて，テスト開始時の状態と入力するコマンドの組み合わせにより遷移先の状態を決定する．従って，コマンド入力時の状態に着目することで，この課題を解決できると考えられる．そこで，明示されているフレームワークの状態遷移モデルを活用する．例えば，アプリが起動していない状態では，どんなアプリのコマンドを入力しても状態遷移しない．そのため，アプリを起動させた後に，アプリのコマンドを入力する必要がある．従って，この例では，テストケースの初めにアプリを起動させるコマンドを入力することで，状態遷移しない有効性の低いテストケースを排除することができる．

前述の例を参考にして，網羅的かつ効率的なテストケース生成方法は次の通りである．まず，アプリとフレームワークにおける全コマンドの組み合わせを網羅してコマンド列を生成する．この際，コマンド列の長さはあらかじめ決めておく．次に，生成した各コマンド列の前後に，高確率で状態遷移するコマンドを挿入してテストケースとする．挿入するコマンドはフレームワークの状態遷移モデルを活用して決定する．

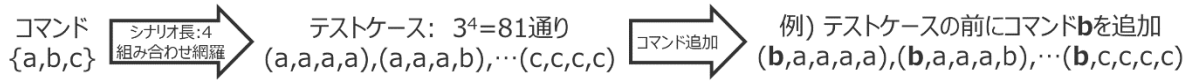

図 4　網羅的なテストケース生成の例

(3)　自動テスト生成の実験とその結果

(1)(2)で述べた手法の有効性を確認するために，アプリとフレームワークの状態遷移モデルを用いて，自動テスト生成に関する実験を行った．この実験では，下記 2 点について検証した．

(1) 生成したテストケースが混入した不具合を検出する回数
(2) アサーションが不具合を検出する精度

上記 2 点を検証するために，不具合を 1 件混入し，コマンド長が 5 のテストケースを，以下の 3 つの方法で生成・実行することで，不具合の検出状況を確認した．実験結果を表 1 に示す．

テスト 1：関数 8 個を 5 回呼び出す全てのテストケース
テスト 2：テスト 1 の前に Start，後ろに Terminate を追加したテストケース
テスト 3：テスト 1 の前に Start，後ろに Suspend を追加したテストケース

表 1 中の各メトリクス項目について説明する．不具合顕在化数はアサーションによる検出とは関係なく，実際に不具合現象（状態ロック）が発生したテストケース数である．不具合密度はテストケース数に対する不具合顕在化数の比である．アサーション精度はアサーションで失敗になったテストケースのうち，実際に不具合が発生したテストケース数の割合である．混入した不具合の性質上，アサーションによる不具合の見逃しはなかったが，不具合の誤検出が発生した．

この実験により検証する 2 つの内容を順番に確認する．テスト 1 とテスト 2，テスト 3 を比較すると，最初と最後にコマンドを追加することにより，不具合顕在化数が増加したことがわかる．また，不具合顕在化数も増加していることから，提案手法を導入することで効率的なテスト実施

が可能になるといえる．次に，テスト 1 とテスト 2，テスト 3 のアサーション精度を比較すると，テスト 2，テスト 3 ではアサーション精度は向上し，1.0 となっている．従って，テストケースの前後にコマンドを挿入することによる，テストケースを絞り込むことの効果として，アサーションによる不具合の誤検出が無くなったことがわかる．

表 1　従来研究における実験結果（抜粋）

メトリクス	テスト 1	テスト 2	テスト 3
テストケース数	32768	32768	32768
不具合顕在化数	684	2133	5851
不具合密度	0.021	0.065	0.179
アサーション精度	0.065	1.000	1.000

3.3.　本研究における研究課題

本研究では，従来研究の実験で定義した状態遷移モデルを使用して，コマンド長を固定して，網羅的に生成した全テストケース数と比べて，十分少ない数のテストケースを実施することで，不具合を検出することを考える．表 1 のテスト 1 における不具合密度が 2.1%なので，一様乱数を用いてコマンド長 5 のテストケースを生成した場合，50 件程度のテストケースを生成すれば，不具合を 1 件検出することが期待される．本研究ではランダムテストよりも効率的に不具合を検出できる，つまり，50 件よりも少ないテストケースを生成・実施して，1 件の不具合を検出することが期待できるような方法を検討する．また，実際の開発では複数の不具合が潜在していることを考え，本研究では，検出できる不具合の数だけではなく，不具合の種類についても着目する．

4.　提案手法

3 章で述べた通り，ランダム生成よりも少ない数のテストケースで不具合を検出することが期待される，コマンドの組み合わせ最適化を用いた自動テスト生成手法を提案する．状態遷移モデルが明示されていない場合，テスト生成に使用できる情報は全コマンドの情報程度しかないため，テスト対象に入力するコマンドの組み合わせを最大化（最適化）することにより，ランダム生成よりも不具合を効率的に発見するための自動テスト生成手法を検討する．

4.1.　コマンドの組み合わせ

通常の状態遷移テストでは，状態遷移モデルを参考にして，遷移する状態や状態間の遷移イベントを網羅するようにテストを実施する．テスト対象に含まれる不具合の数や種類，場所はわからないため，網羅的にテストを実施することで，より多くの数・種類の不具合を検出することができる．本研究においても同様に，状態遷移テストを網羅的に実施することで，より多くの数・種類の不具合を検出することができると考えられる．ただし，本研究における提案手法では，テストケース数を少数に絞るため，十分な網羅性を求めることはできない．しかし，コマンドの組み合わせにおける重複を減らすことにより，状態遷移に関する網羅性を高めることが期待できると考えられる．以下に，部分列に着目したコマンドの組み合わせの数え方を示す．

コマンド長 N（N は自然数）のテストケースに対して，連続する 2 個以上 N 個以下のコマンドの部分列に着目する．コマンドの組み合わせを最大化するためには，コマンドの部分列の種類を最大化することを考えればよい．ここでは，次節の最適化計算で使用する，任意の i 個のコマンドの部分列に対して，1 つのテストケースの中で取り得る最大個数と，全コマンドを用いて網羅した個数について考える．具体例として，コマンドが $1, 2, \dots, c$ の c（c は $N < c$ の自然数）種類ある場合のコマンド列 $1, 2, \dots, N$ について，任意の i 個のコマンドの部分列の組み合わせの種類は以下のように計算することができる．以下では，コマンドの部分列を $[a, b, c]$ のように記述する．

- 2 個のコマンドの部分列：$[1, 2]$，$[2, 3]$，…，$[N-1, N]$ の $N-1$ 個の組み合わせがある．また，全コマンドを用いた，2 個のコマンドの部分列の組み合わせは最大で c^2 個になる．
- i 個のコマンドの部分列：$[1, 2, \dots, i]$，$[2, 3, \dots, i, i+1]$，…，$[N-i+1, N-i+2, \dots, N]$ の $N-i+1$ 個の組み合わせがある．また，全コマンドを用いた，i 個のコマンドの部分列の組み合わせは最大で c^i 個になる．

・N個のコマンドの部分列：$[1, 2, \ldots, N]$の 1 個になる．また，全コマンドを用いた，N個のコマンドの部分列の組み合わせは最大で c^N 個になる．

4.2. 最適化する対象と目的関数

(1) コマンドの組み合わせカバレッジ

まず，4.1.で述べたコマンドの部分列の組み合わせの数を使用して，任意の i 個のコマンドの部分列の組み合わせに対するカバレッジを定義する．今回のテスト生成では，生成するテストケースを少数に絞るため，全コマンドを用いて生成可能なコマンドの部分列を網羅することはできない．そのため，生成する全テストケースに対して，任意の i 個のコマンドの部分列が，組み合わせの種類を最大限に網羅していることをカバレッジにより確認できれば良い．従って，カバレッジは，全テストケースにおける任意の i 個のコマンドの部分列が取り得る組み合わせの最大値に対する，実際のコマンドの部分列の組み合わせ数の比で表すことにする．任意の i 個のコマンドの部分列のカバレッジを(1)式の通りに定義する．

$$i\text{個のコマンドの部分列のカバレッジ} = \frac{\left(i\text{個のコマンドの組み合わせの数}\right)}{\min\left\{c^i,\ (N-i+1)\times\left(\text{テストケース数}\right)\right\}} \tag{1}$$

分母について，i が 2 に近い時には全テストケースにおける組み合わせの最大値と全コマンドを用いて網羅した組み合わせの数は等しく，c^i になる．一方，i が N に近くなると，全テストケースにおける組み合わせの最大値のほうが全コマンドを用いて網羅した組み合わせの数よりも小さくなるため，（$N-i+1$）×(テストケース数)となる．

次に，(1)式で定義したカバレッジを用いて，2 個から N 個までの全てのコマンドの部分列に対するカバレッジを足したもの（以下，全カバレッジ）を目的関数として(2)式の通りに定義する．

$$\text{全カバレッジ} = \sum_{i=2}^{N}(i\text{個のコマンドの部分列のカバレッジ}) \tag{2}$$

(2) コマンドの使用回数

全カバレッジのみを最大化するように最適化を行った場合，局所最適解に陥ってしまい，既に使用しているコマンドの組み合わせを優先的に用いることで最適化が行われるため，作りやすい組み合わせを使ってカバレッジを稼ぐことがある．この場合，特定のコマンドを高頻度に使用してテストケースを生成する，また，特定のコマンドを一切使わずにテストケースを生成することがある．生成するテストケースの集合における各コマンドの使用回数に大きな差がある場合，特定のコマンドに偏ったテストを実施することになる．一方，不具合を見つけるためには，特定のコマンドを集中的にテストするよりも，全コマンドを満遍なくテストしたほうが探索効率は良いと考えられる．そのため，全テストケースにおける各コマンド使用回数に対する標準偏差を目的関数として最小化することにより，コマンド間における使用回数の差を小さくする．

4.3. 最適化手法

今回の最適化では，全カバレッジの最大化と，コマンドの使用回数における標準偏差の最小化の 2 つの最適化を行うので，多目的最適化を行うことになる．前述の通り，全カバレッジを最大化する場合，特定のコマンドの使用回数が増えることにより，コマンドの使用回数における標準偏差は大きくなることがある．これら 2 つの目的関数はトレードオフの関係があることから，それぞれを目的関数として個別に最大化・最小化を行い，テストケースを最適化する必要がある．

5. 実験内容

4 章にて提案した最適化手法を用いた自動テスト生成方法と一様乱数を用いたテスト生成手法の 2 種類のテスト生成手法を比較することにより，提案手法の有効性を確認する．

5.1. 実験により検証する内容

まず，実験により検証する項目について述べる．本研究の実験では，提案手法がランダムなテスト生成よりも有効であることを確認するために，より多くの件数・種類の不具合を検出できることを確認する．検証する内容について，Research Question(以下 RQ)として以下に記載する．

RQ1. 提案手法は，ランダムなテスト生成よりも多くの数の不具合を検出できること
RQ2. 提案手法は，ランダムなテスト生成よりも多くの種類の不具合を検出できること

5.2. 実験内容

従来研究の実験で定義した状態遷移モデル（図 2，図 3）をテスト対象として，ランダムなテスト生成と提案手法の 2 つの方法における不具合検出状況を確認する．従来研究の実験結果より，50 件のテストケースを生成すれば不具合を検出することが期待できることを踏まえ，コマンド長は 5 に固定して，生成するテストケースの数は，10 件から 50 件までを 10 件間隔で生成する．

(1) 前提条件

状態遷移モデルに関する前提条件として，フレームワークの状態遷移モデルのみ明示されており，一方，アプリの状態遷移モデルは明示されておらず，状態遷移に関係するコマンドのみわかっているものとする．

(2) テスト対象・不具合の混入

テスト対象とする状態遷移モデルは，従来研究の実験で定義した状態遷移モデル（図 2，図 3）を使用する．本研究の実験では，検出する不具合の種類についても検証を行うため，不具合を 2 つ混入する．1 つは従来研究の実験と同じ不具合を混入し，もう 1 つは別の不具合を混入する．いずれの不具合も同じアサーションで検出できるように考慮する．従って，異なる原因ではあるが，フレームワークレベルでは同じ状態ロックの振る舞いをする不具合を混入することになる．

(3) 不具合の検出方法

本研究の実験では，発生した不具合の数・種類を正確に検出するため，また，検証内容にアサーションの精度を含まないため，不具合の検出には，従来研究で提案したアサーションは用いない．その代わりに，ログを用いて局所的に不具合が顕在化したことを確認することで，確実に不具合の検出を行う．混入した 2 つの不具合が顕在化した際に区別できるようにログを出力する．また，今回発生する不具合は状態ロックであるから，2 つの不具合が 1 つのテストケースの中で連続して起こることはなく,1 つのテストケースにおいて起こり得る不具合は最大 1 種類である．

(4) テストケースの生成手法

ランダム生成では，一様乱数を用いて乱数を用いてコマンド列を生成する．最適化による生成では，4 章に記載した最適化手法を用いてコマンド列を生成する．最適化を行うために，進化計算アルゴリズムの 1 つである，遺伝的アルゴリズムを用いる．遺伝的アルゴリズムのパラメータについては，交叉率は 1.0，突然変異率は全テストケース中のコマンド延べ数の逆数，個体群サイズは 100 とした．ランダム生成，最適化による生成，いずれの方法についても，テスト生成の度に検出する不具合の数・種類には一定程度の幅がある．今回の 2 つの手法により生成したテストによる不具合の数・種類の検出数は正規分布に従うものと考えられる．従って，テスト生成を 1000 回実施して平均を取ることで，その手法による期待値とする．

6. 実験結果

6.1. 1000 回実施した結果の平均

ランダム生成と最適化による生成の両手法による実験結果について，それぞれ表 2，表 3 に示す.今回の実験では 5.2.に記載した通り,平均値をそのテスト生成方法の期待値としてまとめた.検出した不具合の数や種類，そして，参考情報として全組み合わせ数の期待値を記載している.

表 2　1000 回実施したテストケース生成結果(ランダム生成)

#	テストケース数	不具合件数の平均	不具合の種類の平均	全組み合わせ数の平均
1	10	0.434	0.394	88.930
2	20	0.844	0.702	162.153
3	30	1.283	0.947	226.555
4	40	1.681	1.119	284.942
5	50	2.059	1.311	340.243

表 2，表 3 における各項目について説明する．テストケース数は 1 回のテストケース生成にお

いて生成したテストケース数である．不具合件数の平均は，生成したテストケースにおける不具合を検出したテストケースの数を平均したものである．不具合の種類の平均は，混入した 2 種類の不具合のうち，検出できた不具合の数を平均したものである．不具合件数の平均では同じ不具合を検出した場合も複数回カウントするが，不具合の種類の平均では同じ不具合は 1 件としてカウントする．従って，不具合の種類の平均を，生成したテストケースが実際に検出できた不具合の数として考えることもできる．全組み合わせ数の平均は，生成したテストケースにおける全てのコマンドの部分列の組み合わせの種類を実施回ごとに合計し，平均したものである．

表 3 1000 回実施したテストケース生成結果(最適化による生成)

#	テストケース数	不具合件数の平均	不具合の種類の平均	全組み合わせ数の平均
1	10	0.484	0.449	99.897
2	20	0.938	0.808	182.534
3	30	1.247	0.983	243.951
4	40	1.690	1.198	303.991
5	50	2.139	1.402	363.994

6.2. 実験の結果

表 2，表 3 より実験の結果について確認する．比較しやすいように，不具合件数の平均と不具合の種類の平均について，棒グラフを図 5，図 6 に示す．不具合件数の平均については，多少の例外はあるが，概ね最適化のほうが高い値を示している．また，不具合の種類の平均については，全体的に最適化によるテスト生成のほうが高い値を示している．従って，実験結果から下記の通り RQ に関してそれぞれ確認することができた．

RQ1. 最適化によるテストケース生成のほうが，ランダムな生成よりも不具合件数の平均も高いことから，提案手法はランダムな生成よりも不具合を検出できたといえる．

RQ2. 最適化によるテストケース生成のほうが，ランダムな生成よりも不具合の種類の平均も高いことから，提案手法はランダムな生成よりも多くの種類の不具合を検出できる．

RQ1，RQ2 の結果から，最適化によるテストケース生成のほうが，ランダムな生成よりも効率的に不具合を検出できると考えることができる．

また，今回の検証項目には含めていないが，表 2 と表 3 を比較すると，全組み合わせ数の平均については提案手法のほうがランダム生成よりも高い値となっている．このことから，最適化の効果により，ランダム生成よりも様々なコマンドの組み合わせに関するテストを実施できていることがわかる．

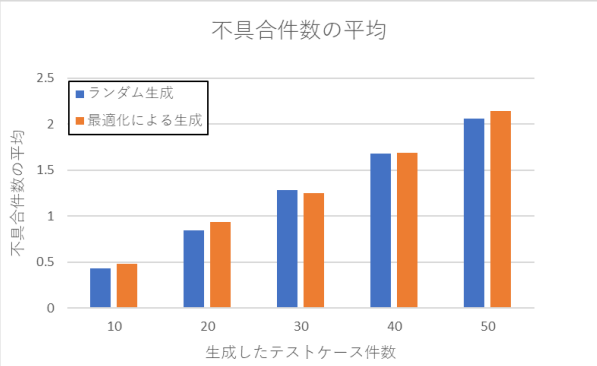

図 5 不具合件数の平均

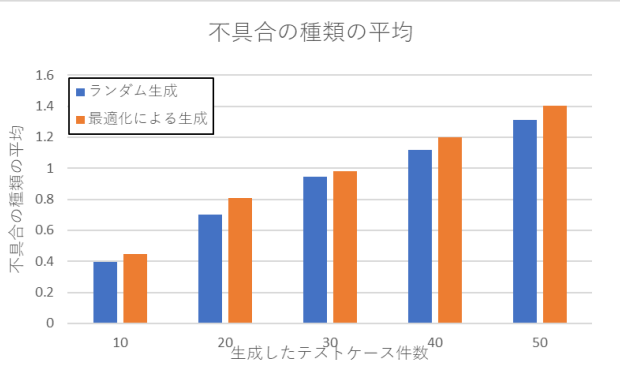

図 6 不具合の種類の平均

7. 考察

実験結果の考察として，今回のテスト生成手法によるテスト絞り込みの効果に関する考察と最適化手法の優位性について考察する．

7.1. 今回のテスト手法によるテスト絞り込みの効果

表 2 より，全件(32768 件)をテストしなくても，30 件程度のテストを実施すれば不具合を 1 件検出できることが期待される．また，全体の 0.1%未満のテストケース数のテスト実施で不具合を検出できることが期待されるので，全体に比べて十分に少ない数のテストケースで不具合を検出

できるといえる．従来研究における提案手法の場合，コマンドを追加する方法により 1/64 にテストケースを絞り込むことができたが，今回の提案手法では 1/1000 にテストケースを絞り込むことができた．従来手法では同じ不具合を複数回検出していることや，確実に不具合を検出することができるため，単純比較はできないが，絞り込みの効果としては従来よりも効果が高いと考えられる．混入した不具合を 2 件とも見つけるためには，不具合の種類の平均が 1.5 を超えればよいと考えられるので，表 2，表 3 の結果を見ると最適化によるテスト生成でも 50 件では十分とは言えないので，60 件かそれ以上の数のテストケースを生成・実施する必要があると予想される．

7.2. 最適化手法の優位性

コマンドの使用回数についての標準偏差を最小化することにより，テストケース全体におけるコマンドの使用回数を均すことができた．状態遷移モデルがブラックボックスで，状態遷移に関する情報が何もない場合，探索方法としては全体を満遍なく探索することが良いということがわかる．一様乱数によるテストケース生成の場合，乱数によってコマンドの使用回数はある程度同じになることを考えると，コマンド使用回数の最適化によりランダム生成のようにコマンドの使用回数における偏りを少なくすることができ，また，全体カバレッジが大きいことから，より多くのコマンドの組み合わせに対してテストしているといえる．

8. 今後の展望

提案手法を実際の開発に導入するためには，解決すべき課題が複数ある．自動テスト生成手法を実際に導入するために解決すべき課題を今後の展望として挙げる．

8.1. より複雑な状態遷移モデルを用いた不具合検出検証

今回の実験で使用した状態遷移モデルはアプリの状態数が 4，コマンド数が 4 という，とても小さい状態遷移モデルに対して，不具合を 2 件混入して検証を行った．実際の開発におけるテスト対象は，状態遷移モデルは複雑で，状態数やコマンドが今回のモデルよりも多い．従って，状態数やコマンドを増やした時に，ランダム生成と比較して，提案手法が十分に有効であることを確認する必要がある．

8.2. 最適化する対象の検討

今回の提案手法では，カバレッジとコマンドの使用回数に対する標準偏差を最適化することで，ランダム生成よりも良い結果を得ることができた．さらに最適化する対象，つまり，目的関数やコスト関数，制約条件を追加することで，今回の提案手法の最適化よりもさらに良い結果を得られると考えられる．例えば，状態遷移を起こすためには異なるコマンドを入力したほうが有効であることが多いと考えられるので，連続して同じコマンドを入力する回数を最小化するように目的関数を定義することが挙げられる．この場合，同じコマンドを連続で入力することはカバレッジを上げるのに効果があるため，カバレッジに関する目的関数とトレードオフの関係になるのでカバレッジとは異なる目的関数とする必要がある．

9. 謝辞

本研究を行うにあたり， 2021 年度　SQiP 研究会 研究コース 5 主査の石川冬樹氏，副主査の徳本晋氏，栗田太郎氏には，本研究を遂行する上で様々なご指導や助言をいただきました．深く感謝申し上げます．また，本研究会への参加の機会をいただき，ソフトウェア開発における課題についての情報・助言をいただいたパナソニック ITS(株)の皆様にも感謝致します．

参考文献

[1]丹野治門， et al.「テスト入力値生成技術の研究動向」コンピュータソフトウェア， Vol34, No.3, p.3_121-3_147, 2017
[2] Boris Beizer「ソフトウェアテスト技法」 日経 BP 出版センター, 1990.
[3] SQiP 研究会　人工知能とソフトウェア品質分科会，状態遷移モデルの一部が明示されていない場合における自動テスト 生成手法の提案，2021
[4] 吉岡信和 「SPIN による設計モデル検証」 近代科学社，2008