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

研 究 員：松尾　正裕（パナソニック ITS 株式会社）
主　査　：石川　冬樹（国立情報学研究所）
副 主 査：徳本　晋（富士通株式会社）
　　栗田　太郎（ソニー株式会社）

研究概要

本研究では，昨今のソフトウェア開発における規模の増大や短納期化に対応するため，品質担保の効率化に有効である,状態遷移に関する自動テスト生成手法について提案する．状態遷移不具合は，発生させてしまうと開発に与える影響が大きいため，自動テスト生成により不具合を発見した場合の効果は大きいと考えた．継続リリースなどを理由にドキュメントの内容を最小限に留める場合を仮定して，状態遷移モデルの一部が明示されていない場合の自動テスト生成手法について検討し，実験により有効性を検証した．

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ーニーズに合わせるために，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，高確率で別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，分業開発では，機能開発だけではなく，横串となる共通部分（フレームワークやライブラリ）の開発も行う．このような開発の特徴に合わせた手法を検討し，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 章では今回の研究課題と提案手法について述べる．その後，5 章，6 章では状態遷移テストに関する実験の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．

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，要求に合わせて各機能を修正する．特にユーザーに近い箇所は高頻度に，そして大幅に変更されるため，ドキュメントは最低限の内容に留めることが多い．一方，共通部分は修正による他機能への影響の大きさから，ユーザーに近い機能に比べると修正は限定的である．また，利用者の利便性から，使用方法や設計情報を記載して，ドキュメントの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ことを考える．修正量や修正頻度から状態遷移不

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たためである．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している．

今回の研究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と，それを利用するアプリを複数定義する．今回の仮定として，アプリの状態遷移モデルは明示されておらず，アプリフレームワークの状態遷移モデルは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，アプリ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

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，今回はアプリの状態遷移モデルが明示されていないため，通常の方法によりテストケースを生成することができない．従って，状態遷移テストに関する自動テスト生成を実施するためには，下記２点の課題を解決する必要がある．

3.1.　テストオラクル問題

状態遷移するためのイベントや関数（以下，コマンド）の情報は実装から取得できるため，コマンドを組み合わせることでテストケースを生成することができる．一方，遷移後の状態に関する情報が明示されていないため，期待値を機械的に決定することは難しい．そのため，状態遷移の期待値を知っている開発者がテストケースの期待値を埋めていく必要があるが，大量のテストケースに対して１つずつ期待値を記述することは現実的ではなく，また，自動化を妨げる要因にもなる．

3.2.　テストケースの妥当性・有効性

状態遷移は状態とコマンドの組み合わせにより決定されるため，ある状態においてコマンドを総当たりで入力することにより，状態遷移させることが可能である．ただし，このようにコマンドの全組み合わせによりテストケースを生成すると，状態遷移をしない無意味なテストケースを大量に生成してしまう懸念がある．その対策として，テストケースの生成手法についても新たな手法を検討する必要がある．

4.　提案手法

前章で挙げた２点の課題に対するそれぞれの解決策を述べ，その後，それらの解決策を反映した自動テスト生成手法について提示する．状態遷移モデルを仮定せずに自動テスト生成を実施した研究[3]があるが，本研究では部分的に状態遷移モデルを用いている点が異なるため，解決策も異なる．

4.1.　テストケース非依存の検査機構

アプリの状態遷移モデルが明示されていないため，各テストケースに対して期待値を記述することが難しい．そこで，テストケースに依存しない，常に成り立つべき検査項目[4]をアサーションとして記述する．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．今回は，明示されているアプリ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作成することで，複数アプリ共通のアサーションとして使用することができる．

モデル検査のように状態を全探索するような検査では，そのような検査項目の記述は従来からなされていて[4]，仕様記述言語や仕様パターン[5]が整備されている．なので，今回

はモデル検査で使用されている仕様記述言語のひとつとして LTL(Linear Temporal Logic: 線形時相論理）を用い，検査項目を記述することとした．この検査項目を予めモデル検査により満たしていることを確認した後，実行を監視してテスト成否を判断するような検査機構として実装することでテストケースの成否を判断する．例えば，「状態 A になったら，いつかは必ず状態 B になる」という LTL に対して，実際のテストでは状態 A になったらフラグを立て，状態 B になったらフラグを寝かすように実装する．テスト終了時にフラグの状態を確認し，LTL を満たしたかどうかを確認する．

また，モデル検査では無限の実行を想定して検査を行うが，ソフトウェアテストにおけるテストシナリオは有限である．従って，状態遷移を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入力を追加する．

4.2.　明示されている状態遷移モデルを活用したテスト生成

無意味なテストケースが大量に生成されてしまう懸念への対策として，明示されているアプリフレームワークの状態遷移モデルを活用する．まず，アプリとアプリフレームワークの全コマンドを用いて,全ての組み合わせを網羅したテストケースを生成する.この時，コマンド長は予め決めておく．次に，無意味なテストケースを排除するために，確実もしくは高確率で状態遷移するコマンドを，生成した各テストケースの前後に挿入する．例えば，アプリは起動しないと状態遷移しないため，起動コマンドを各テストケースの前に挿入する．このようなテストケース生成手法により，全く状態遷移しないテストケースを排除することができるため，有効性の高いテストケースのみに絞り込むことが可能になる．

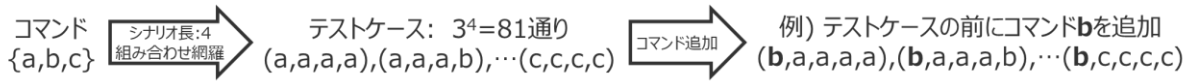

図 2　網羅的なテストケース生成の例

生成するテストケースにおけるシナリオ長について，シナリオが長すぎると不具合原因の解析に時間を要する[6]．また，短すぎても状態遷移が十分に行われないため，不具合が顕在化しにくい．従って，適切なシナリオ長にする必要がある．本研究の実験では，検証のため生成したテストケースを全て実行できる程度の短いシナリオ長を選択した．

4.3.　自動テスト生成手法

4 章で述べた解決策を用いた自動テスト生成手法について述べる．本手法におけるフローの全体像を図 3 に示す．自動生成の対象はテストケースとテストコードである．そのため，テストケースのシナリオ長やアサーションの検査内容，テストケースの前後に追加するコマンドなどは予め決めておく必要がある．

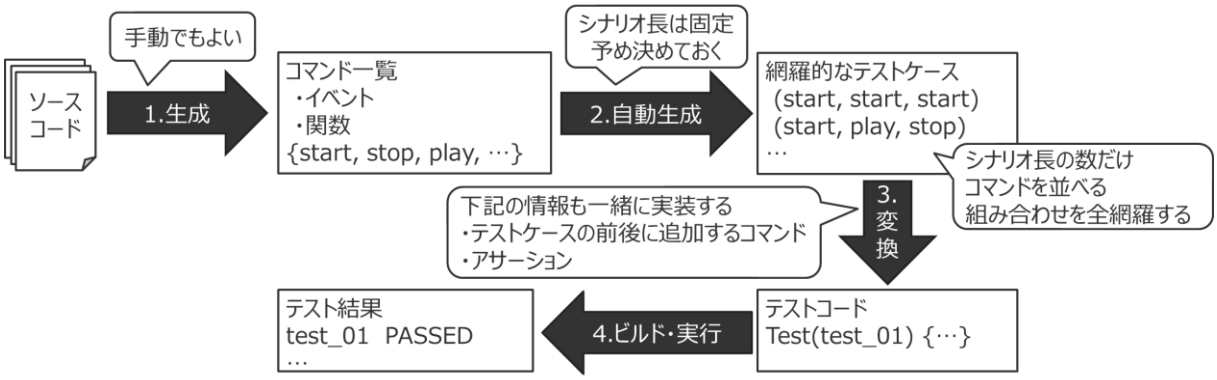

図 3　自動テスト生成の全体像

まず，「1.生成」では，実装から全コマンドを取得し，コマンド一覧を作成する．このコマンド一覧は手動で作成しても問題ない．作成したコマンド一覧を用いて，「2.自動生成」

ではコマンドを全て組み合わせて機械的にテストケースを自動生成する．テストケースを生成する手法は 4.2.に提示した通りであるが，コマンドの追加は行わない．

次に，「3.変換」では，生成したテストケースをもとにテストコードを自動生成する．このとき，テストケースの前後に追加するコマンドも実装する．また，常に満たすべき性質の検査機構をアサーションとして実装する．具体的には，コマンドを入力する度にアプリフレームワークの状態を確認し，必要に応じてフラグの状態を変更する処理を実装する．

テストコードを生成したら，「4.ビルド・実行」においてテスト対象を必要に応じてビルドしてから，テストコードをビルドする．一通りビルドが完了したらテストを実行する．

5.　実験

前章にて提案した手法の有効性を確認するために，2 つの実験を実施した．1 つ目の実験（以下，実験 1）ではアプリフレームワークの状態遷移モデルのみを用いて，混入させた不具合の検出とテストケース絞り込みの有効性について確認した．2 つ目の実験（以下，実験 2）では，アプリの状態遷移も含めた状態遷移モデルにおいて，実験 1 と同じ条件のアサーションにて不具合を検出できること，また，テストケースを絞り込むことの有効性について確認した．2 つの実験を通じて検証したい内容と，それぞれの実験における状態遷移モデルや前提条件，テストケースの生成方法について述べる．

5.1.　2 つの実験を通じて検証する内容

まず，実験で確認する項目について述べる．今回の実験では，提案手法により 3 章で掲げた 2 つの研究課題が解決可能であることを確認する．下記に検証する内容について，Research Question（以下，RQ.）として記載する．

RQ.1　仕様記述を用いて作成したアサーションにより不具合を検出できること
RQ.2-1 網羅的に生成したテストケースにより，不具合が顕在化すること
RQ.2-2 状態遷移モデルを活用したテストケース絞り込みにより，不具合が顕在化しやすくなること

5.2.　実験 1

状態遷移モデル

実験 1 ではアプリフレームワークの状態遷移モデルを用いたテストを行った.アプリフレームワークの状態遷移モデルと呼んでいるが，実際はアプリの起動，終了といったアプリのライフサイクルに関する状態遷移モデルであり,アプリフレームワークはこの状態遷移を管理する役割を担っている．なお，アプリフレームワークからアプリにコマンドが通知されることにより状態遷移するが，名称の後半に「_Ready」と付いている状態から次の状態に遷移するためには，アプリがアプリフレームワークに応答を返す必要がある．

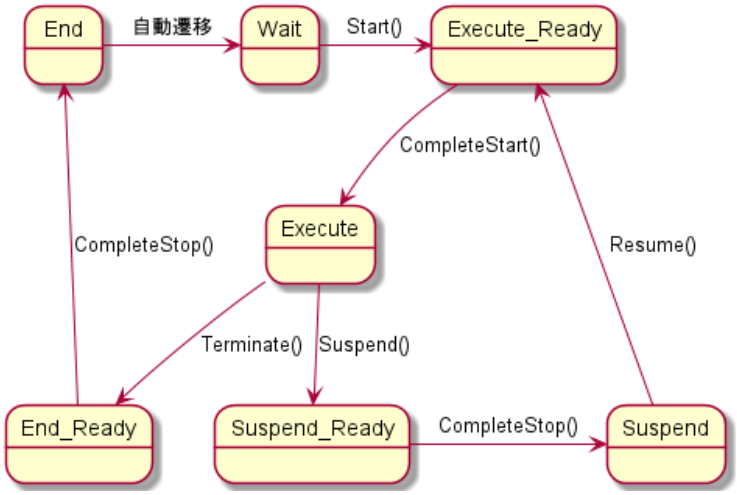

図 4 アプリフレームワークの状態遷移図

前提条件

状態遷移モデルに関する前提条件として，前節で提示した状態遷移モデルは完全であると仮定する．現実的にはドキュメントに記載されている状態遷移モデルが完全であるとは限らないが，今回の実験では，アプリフレームワークの状態遷移モデルに不足している情報があっても検証内容に反映できないためである．

次に，テストケース生成で使用する全コマンドは，図 4 に記載されている．この情報からコマンド一覧を作成し，テストケースを生成する．これは，4.3.で示したコマンド一覧

を手動で作成したことに相当する．

使用するアサーション

今回，下記の通りアサーションを 2 つ作成した．

Assert.1: Execute 状態になったら，いつかは Suspend 状態になる

Assert.2: Execute 状態になったら，いつかは Suspend 状態または Wait 状態になる

Assert.1 は図 4 の状態遷移モデルにおいて公平性を前提としている．つまり，Execute 状態において，Terminate と Suspend はある程度均等に実行される[4]という前提の下で成り立つ．Assert.2 は Assert.1 で仮定した公平性を仮定しなくても成り立つ．また，これらのアサーションを用いた検査は同時にできないため，アサーション毎にテストコードを新たに生成し，テストを実施することで各アサーションによる検査を行う．

混入する不具合

実験 1 では，Suspend_Ready 状態から Suspend 状態に遷移する際に必要な CompleteStop を応答し忘れる不具合を混入した．実験 1 の状態遷移モデルにおいて，Suspend_Ready 状態に遷移すれば必ずこの不具合が顕在化する．不具合を回避するパスも存在しないため，実験 1 では，Suspend 状態に到達することはできない．

テストケースの生成方法

実験 1 では，状態遷移のきっかけとなる 4 つの関数を 5 回呼び出すすべてのテストケース(4^5=1024 件)を生成した．また，RQ.2-2 を検証するため，Start から開始して Terminate で終了するテストケース（以下，抽出ケース)の結果を抽出し，全体の結果と比較する．

5.3.　実験 2

状態遷移モデル

実験 2 では，アプリフレームワークの状態遷移にアプリの状態遷移を追加したものを使用する．今回は録音アプリを仮定して，簡単な状態遷移モデルを図 5 の通りに定義する．アプリの状態遷移はアプリフレームワークの状態における内部状態として考えることができる．

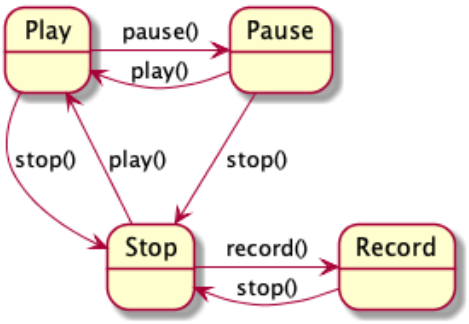

図 5　アプリの状態遷移図

前提条件

実験 2 ではアプリの状態遷移モデルを定義したが，アプリの状態遷移モデルは明示されておらず，開発者以外が状態遷移モデルを知るには実装を紐解く必要があると仮定する．そのため，アプリの状態遷移の情報を用いずにテストケースを生成することで，アプリの状態遷移モデルをブラックボックスとみなすことにする．また，アプリフレームワークの状態遷移モデルは実験 1 と同じ前提条件とする．

使用するアサーション

アプリフレームワークの状態遷移モデルは変わらないため，実験 1 と同じアサーションを用いる．これにより，アプリの状態遷移が明示されていない場合でも，アプリフレームワークの状態遷移モデルのみを用いてアサーションを作成できることを確認する．

混入する不具合

実験 2 では，実験 1 と同じく Suspend_Ready 状態においてアプリが応答を返さない不具合を混入する．ただし，今回はアプリの状態遷移も考慮する．Suspend から Suspend_Ready に遷移する際のアプリの状態に関する条件として，Stop 状態か Pause 状態になっていることと定義する．この条件を満たすためには，Suspend_Ready 状態になったら，Stop 状態にすることを意図して stop を呼べばよいが，誤って pause を呼ぶ実装にしてしまったものとして不具合を混入させる．Record 状態で Suspend_Ready 状態に遷移した場合はアプリの状態に関する条件を満たさないため，応答を返さないことにより状態遷移不具合になる．

テストケースの生成方法

実験 2 では，アプリの関数も含めると状態遷移のきっかけとなる関数は全部で 8 個ある．

これらの関数を 5 回呼び出すテストケースを網羅する．また，これらのテストケースの前後に start，suspend，terminate，を呼ぶことによりテストの結果の変化を確認する．生成するテストを下記に記載する．それぞれテスト 1～テスト 5 と呼ぶこととする．
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 の各テストケース＋最後に Terminate
テスト 3: テスト 1 の各テストケース＋最後に Suspend
テスト 4: 最初に Start+テスト 1 の各テストケース＋最後に Terminate
テスト 5: 最初に Start+テスト 1 の各テストケース＋最後に Suspend

6.　実験結果

6.1.　実験 1 の結果

実験 1 の結果を下記に示す．まず，各メトリクス項目について説明する．不具合顕在化数は不具合が顕在化したテストケースの数である．今回混入させた不具合は 1 件であるから，顕在化した不具合の原因は全て同一である．また，今回の実験では，不具合が顕在化するテストケースは予め検証してある．Assert.検出数は Assert によりテスト実行結果が FAILED になった件数であり，不具合の誤検出（偽陽性）も含んでいる．不具合密度は不具合が顕在化したテストケースがテストケース全体に占める割合である．Assert.精度は顕在化した不具合を正しく検出した割合である．今回の実験では顕在化した不具合を見逃したケースはなかったので，Assert．検出数に対する不具合顕在化数の割合になる．

表 1　実験 1 の結果

メトリクス	全体	抽出ケース	抽出ケース以外
テストケース数	1024	64	960
不具合顕在化数	299	29	270
Assert.1 検出数	781	64	717
Assert.2 検出数	574	29	545
不具合密度	0.292	0.453	0.281
Assert.1 精度	0.383	0.453	0.377
Assert.2 精度	0.522	1	0.495

表 1 から分かることについて述べる．まず，不具合密度に着目すると，抽出ケースは全体と比べて高くなっているため，不具合が顕在化しやすいテストケースを抽出できていることがわかる．また，Assert．精度についても，抽出ケースは全体と比べて精度が向上しており，誤検出が少ないことがわかる．更に，Assert.2 精度の方が高いことから，公平性を仮定せずに成り立つアサーションを用いた方が良いことが示唆される．

6.2.　実験 2 の結果

続いて，実験 2 の結果を示す．各テストの内容は 5.3.に記載の通りである．

各テストの不具合密度は実験 1 と比較して低いものの，不具合が顕在化していることがわかる．不具合密度に着目すると，テスト 1 とテスト 2 は等しいが，テスト 3 は高くなっている．このことから，コマンドを追加して状態遷移する可能性が高くなったとしても，必ずしも不具合が顕在化しやすくなるわけではないことがわかる．顕在化しやすくなるのは，不具合の発生条件に依存すると考えられる．次に，テスト 4 はテスト 2 と比較すると，また，テスト 5 はテスト 3 と比較すると不具合密度が高い．これらの比較により，不具合が顕在化しやすくなったことがわかる．これは，初めに Start を呼ぶことで，アプリが状態遷移しやすくなったためと考えられる．

続いて，Assert．精度に着目すると，テスト 1 と比較してそれ以外のテストでは精度が向上している．これは，コマンド追加により Execute 状態で終了するテストケースがなく

なったためと考えられる．また，公平性の観点では，実験 1 と同じ傾向が見られる．なお，Assert.2 精度が 1 になっているのは，コマンド追加による精度向上に加えて，混入不具合が 1 件で，同件不具合の顕在化により過不足なく検出できたためと考えられる．

表 2　実験 2 の結果

メトリクス	テスト 1	テスト 2	テスト 3	テスト 4	テスト 5
テストケース数	32768	32768	32768	32768	32768
不具合顕在化数	684	684	3582	2133	5851
Assert.1 検出数	13547	13547	6620	24677	15702
Assert.2 検出数	10509	684	3582	2133	5851
不具合密度	0.021	0.021	0.109	0.065	0.179
Assert.1 精度	0.050	0.050	0.541	0.086	0.373
Assert.2 精度	0.065	1	1	1	1

6.3　実験結果のまとめ

これらの実験の結果から，下記の通り RQ.についてそれぞれ確認することができた．

RQ.1　いずれの Assert.精度も 0 ではないことから，今回提案したアサーションにより不具合を検出できることが確認できた．

RQ. 2-1 いずれの不具合顕在化数も 0 でないので，不具合が顕在化することを確認できた．

RQ. 2-2 不具合密度をテスト間で比較した結果，状態遷移モデルを活用したテストケース絞り込みにより，不具合が顕在化しやすくなることを確認できた．また，テスト間での Assert.精度を比較することで，アサーションの不具合検出精度向上も確認できた．

7.　考察

7.1. 実験結果を受けて

今回の実験は，アプリフレームワークの状態遷移モデルのみ明示されているという仮定の下で実施した．また，混入した不具合はアプリフレームワークの状態遷移モデルにおける不具合として検出可能であった．不具合を検出できた要因はこの 2 点にあると考えられる．更に，今回の実験では，アプリの状態遷移を意識しなかったことから，アプリフレームワークにおける検査項目を検証してアサーションを作成しておくことで，別のアプリにおいても同じアサーションを流用してテストを実施することができると考えられる．

7.2.　ランダム抽出によるテストケースの絞り込み

実験 2 では，大量のテストケースを自動生成してビルド，実行した．しかし，実際のテスト対象にはコマンドが多数あるため，全件テストすることは現実的ではない．そこで，実験 2 のテスト 1 から，一様乱数を用いてテストケースの無作為抽出を試みた．1000 件のテストを抽出して実行したところ，20 件程度の不具合が顕在化した．この結果は不具合密度と比較としても妥当な件数である．ただし，テストの妥当性や十分性を考えると無作為抽出では不十分である．従って，テストケース生成に対して何らかの制約条件を与えて最適化することで，効率的に自動テスト生成を実施することが可能になると考えられる．

8.　今後の展望

提案手法を実際に導入するためには，解決すべき課題が複数ある．また，考察で述べた通り，更なるテストケースの絞り込みも必要である．更に，効率化の観点では AI の利用も有効であると考えている．今後の展望として，これらに関して 4 点を述べる．

8.1.　提案手法を導入するための課題

提案手法をもとにモデルを定義して実験を行ったが，提案手法を導入するためには，よ

り具体的な条件等の検討が必要である．今回はコマンドを天下り的に与えたが，実際には実装から抽出する必要がある．また，今回は議論しなかったが，不具合検出効率と解析容易性の両立を考えたシナリオ長や一度に生成する（あるいは実行する）テスト件数についても検討が必要である．これらはテスト対象に依存すると考えられるため，実際に複数のアプリを用いて定量評価を行う必要がある．更に，実際に提案手法を導入した場合の効果を測定・検証することで，客観的に本手法の有効性について議論する必要もある．

8.2.　最適化によるテストケースの絞り込み

テストケースの絞り込みは最適化問題と捉えることができる．最適化問題とは，膨大な候補の中から，与えられた制約条件を満たし，ある基準で最も良い解を選択する問題である[7]．本手法ではコマンドに着目してテストケースを生成するので，コマンドの組み合わせを最適化すると良いと考えられる．最適化の手法として，進化計算アルゴリズム[7]や量子アニーリング[8]がある．

8.3.　状態遷移テストにおける不具合の局所化

1 件の不具合を複数回，それも大量に検出しているため，不具合解析作業も効率化する必要がある．AI を用いた不具合解析技術に不具合の局所化[9]がある．このような技術を状態遷移テストにも応用して不具合解析作業を効率的に進めることができれば，本手法を導入した場合の効果がより一層大きくなると考えられる．

8.4.　クラスタリング分析

テスト結果に対してクラスタリング分析を実施することにより，不具合を多く含んでいる可能性のある部分を推定することができると考えられる．不具合は均一的に存在するのではなく，全体の 20%の部分に 80%の不具合が存在している[10]．そのため，本手法により不具合が検出された場合は，別の不具合が存在する可能性が高いと考えられる．本手法により検出できる不具合は限られるが，不具合検出をきっかけとして，設計や実装を見直しすることにより，品質を効率的に高めることができると考える．

9.　謝辞

本研究を行うにあたり，多くの方々にお世話になりました．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．最後に，お世話になりました全ての皆様に，感謝と御礼申し上げます．

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] 下村翔,et al.「大量の状態とイベントを持つプログラムの自動解析とテスト手法の提案」, ソフトウェアシンポジウム 2016 in 米子
[4] 吉岡信和 「SPIN による設計モデル検証」 近代科学社, 2008
[5] https://matthewbdwyer.github.io/psp/
[6] Ke Mao, et al.「Sapienz: Multi-objective Automated Testing for Android Applications」ISSTA 2016
[7] 大谷紀子 「進化計算アルゴリズム入門」 オーム社, 2018
[8] T. Kadowaki, et al. Phys. Rev. E, **58**, 5355 (1998)
[9] Chris Parnin et al.「Are Automated Debugging Techniques Actually Helping Programmers? 」 ISSTA, 2011
[10] 高橋寿一 「知識ゼロから学ぶソフトウェアテスト【改訂版】」 翔泳社, 2013