

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 ～想定外のイベントを自動テストする～

下村 翔
デンソー

syo_shimomura@denso.co.jp

古畑 慶次
デンソー技研センター

keiji_kobata@denso.co.jp

松尾谷 徹
デバッグ工学研究所

matsuodani@debugeng.com

要旨

画面遷移を含む GUI アプリケーションプログラムのテストを組合せ論理で取り扱うことは、原理的に難しく、順序論理と呼ばれる状態とイベントの時系列を考慮する必要がある。ブラックボックス的な順序論理の組合せ数は膨大になるため、状態やイベントを厳密にモデル化する必要があるが、現場では実践できていない。

本稿では、実際のプログラムから状態とイベントを探索し、モデル化を行う。テストは、得られたモデルから時系列の各ノードに対し、イベントの集合、即ちイベントハンドラが実在するイベントを網羅的にテストする。これにより、ある状態において想定外のイベントに対するテストを自動化できる。さらにテストの効率化のために、SMT ソルバの Z3 を用いて不要なテストケースを削減する。

提案手法を過去に開発したアプリケーションプログラムに適用し、その有効性を確認した。提案手法によりこれまで発見できなかった想定外のイベントによる不具合が明らかになった。

1 はじめに

近年、ソフトウェアは大規模化・複雑化が進んでいる。その中で、品質を確保するためのテスト工程は開発全体の約 4 割程度を占めるという調査結果がある [1][2]。我々は、GUI (Graphical User Interface) を持つアプリケーションのテストに取り組んでいる。GUI アプリケーションのテストは、そのテスト操作に画面上のマウスの位置やタッチパネルの入力などが存在し、一般的なテストと

比べ手間のかかるテストである。

非効率な人で操作に対しては、テスト実行の自動化やテストケースの自動組合せ生成など、様々な自動化手法が提案されている。一方で GUI アプリケーションでは、テスト入力とテスト出力はユーザ操作から生ずるイベントと画面などオブジェクトの状態に左右されるため、組合せによるテストケース生成には限界がある。

本論文の目的は、大量の状態を持ち、かつ、多くのイベント処理を持つプログラムに対して、状態遷移を網羅できるテストケースを自動生成することである。一般的には、状態遷移を正しく表現したモデルを定義し、そのモデルの挙動からテストケースを作成する形式手法が知られている。しかしモデルの作成には専門知識が必要なほか、モデルの作成には追加の工数が必要であるため、現実の開発に適用するのは容易でない。そこで本稿では、仕様ベースのモデル化ではなく、実装ベースのモデル化に基づいたテスト手法を提案する。実装ベースのモデル化とは、実コードに対してマイニング技術を使って、イベント処理や状態を探し出し、その挙動を確認するために必要なテストケースを探索する。提案手法は、実装から状態やイベント処理の抽出処理と、抽出したイベント処理の妥当性を確認する処理で構成する。

本論文の構成は次の通りである。2 章ではテスト対象とする GUI アプリケーションの性質について説明する。3 章は関連研究について述べる。4 章では提案手法のアルゴリズムについて述べ、5 章では提案手法の実装について述べる。6 章で提案手法を実際のアプリケーションに対して適用して手法の有効性を示し、7 章でまとめと今後の課題について述べる。

2 GUI アプリケーションの特徴とテストの課題

GUI アプリケーションは同時に受け付けられる操作が多いため、その操作系列の組み合わせの数は膨大になる。このため手動で GUI アプリケーションのテストケースを作成、実行することは現実的ではない。

ユーザの入力操作をランダムに自動生成したり、あるいは部分的に作成したユーザの入力操作を組み合わせる手法を使うと、テストケース生成を自動化できる。しかし同様に組み合わせの数が膨大になるため、現実的な時間内にテスト実行が終了しない。このため実用上は、自動生成された膨大な数のテストケースに対し、重要なものに絞り込んでテストを実行する必要がある。

実行するテストケースを絞り込む手段として、形式手法に適用できるアプリケーションの動作モデルを用意する方法があるが、現実のアプリケーションにはそのような厳密なモデルは存在しないことが多く、今後作られることも期待できない。

開発者に余計な工数をかけずにテストケースを絞り込む手段がなければ、GUI アプリケーションのテストを効果的に運用していくことは難しい。

3 関連研究

3.1 Capture/Replay

現在の GUI アプリケーションのテスト自動化は Capture/Replay 型のものが主流であり、多くのツールが存在する。Capture/Replay 型のテスト自動化では、まず GUI アプリケーションを実際に操作し、操作内容と期待値を記録してテストケースを作成する (Capture)。その後、記録した操作を再生することでテストを自動実行する (Replay)。しかし Capture/Replay 型のテスト手法は、ユーザ操作の内容やテスト結果の確認方法が GUI のレイアウトに強く依存してしまうことが多く、GUI の変更でテストケースが影響を受けやすいという問題がある。

この点を改善するために、テストケースを作成・管理するための IDE を用意する手法 [3] や、操作対象を実行時に画像認識で探すことで小変更に対応する手法 [4] や、キーワード駆動型テストと呼ばれる手法などがある [5][6]。キーワード駆動型テストでは、テスト実行に必要な情報を外部ファイルに記述する。これらの情報は保守

がしやすいよう抽象的に記述され、テスト実行時に自動テストツールが具体的な情報に置き換える。

しかしながら、これらの手法にはテストケース内での条件分岐処理の記述が難しいという問題があり、同じような構造を持ったテストケースが多く生成される傾向がある。特に開発フェーズの初期では GUI が頻繁に変化するため、テストケースの保守にコストがかかるという問題がある。

3.2 モデルベーステスト

Capture/Replay 型のテスト手法の問題を解決するためのアプローチとして、モデルベースのテストケース自動生成手法がある [7]。モデルベースのアプローチでは、開発者がアプリケーションの動作を表すモデルを記述すると、モデルからテストコードを自動生成することができる。

モデルで表現された動作に対して網羅的にテストケースを生成することができるが、モデルの作成には専門知識が必要なほか、モデルの作成には追加で工数がかかる。また、モデルで表現されない動作に対してはテストケースが生成されないため、設計時に想定していなかった入力を受け取ったときに、意図しない振る舞いをしてしまうといった不具合を見つけることはできない。

3.3 ランダムテスト

モデルを使わずにテストを自動化するアプローチとして、ランダムテストやファジングと呼ばれる手法があり [8]、これを GUI アプリケーションに適用する手法も提案されている [9][10]。これらの手法では、アプリケーションへ入力するデータや操作をランダムに生成してテストを自動実行し、異常終了などの想定外の動作が起こらないかどうかを確認する。

テスト入力をランダムに生成することで、テスト実行のための専門知識が必要なくなる他、担当者の先入観にとらわれることなくテストを行うことができるという利点がある。しかし一般に GUI アプリケーションは可能な操作のパターンが多いため、これらの手法を GUI アプリケーションに適用する場合、実行するべきテストケースの数が膨大になり、現実的な時間内にテストが終了しないという問題がある。このため、実用上は生成するテスト入力を適切に絞り込む必要がある。

網羅的にテストケースを生成したあとテストケースを絞り込むのではなく、少数のテストケースから初めて徐々に網羅率を高めていくアプローチとして、遺伝的アルゴリズムを用いたテストケースの生成手法が提案されている。[11]. GUI アプリケーションに入力するイベント列を個体とし、よりテストの網羅率を高める個体を選択していくことで、よりよいテストケースを探すことができる。しかし遺伝的アルゴリズムは問題に適用する一般的な方法がなく、かつ設定すべきパラメータが多いため、テスト対象のアプリケーションごとに最適なパラメータを探さなければならないという課題がある。

4 提案手法

本論文では、設計時に考慮から漏れた異常系の扱いに基づく不具合や、実装時に埋め込まれた不具合を見つけることを目的に、GUI アプリケーションを対象とした自動テスト手法を提案する。提案手法では、GUI アプリケーションの実装から状態遷移モデルを自動生成しつつ、テスト入力となるイベント列を自動生成して自動実行する。到達可能な全ての状態を網羅するまでテストの実行を繰り返し、意図しない不具合動作を発見する。

経験上、状態遷移モデルを手動で作成することはコストが高い。また、アプリケーションの仕様書や設計書は状態遷移モデルを自動で導出できるような形式では書かれていないことが多い。そこで提案手法では、テスト対象のアプリケーションのソースコードのみを入力とする。このアプローチには、テスト実行のために追加に必要な作業が少ない他、新規に作成しなければならないドキュメントがないため、既存の開発プロセスに容易に組み込むことができるという利点がある。

正常系の検査は既に開発プロセスに組み込まれていることが多いため、提案手法は異常系の不具合の検出に重点を置いている。提案手法を既存の開発プロセスに組み込むことで、効率的にアプリケーションの品質を高めることができる。

4.1 提案手法のアルゴリズム

提案手法のアルゴリズムを図1に示す。提案手法の基本的なアイデアは、一般に非常に数が増えるテスト対象のアプリケーションの実装上の状態を適切な基準で同値分割し、得られた同値クラスを状態とする状態遷移モ

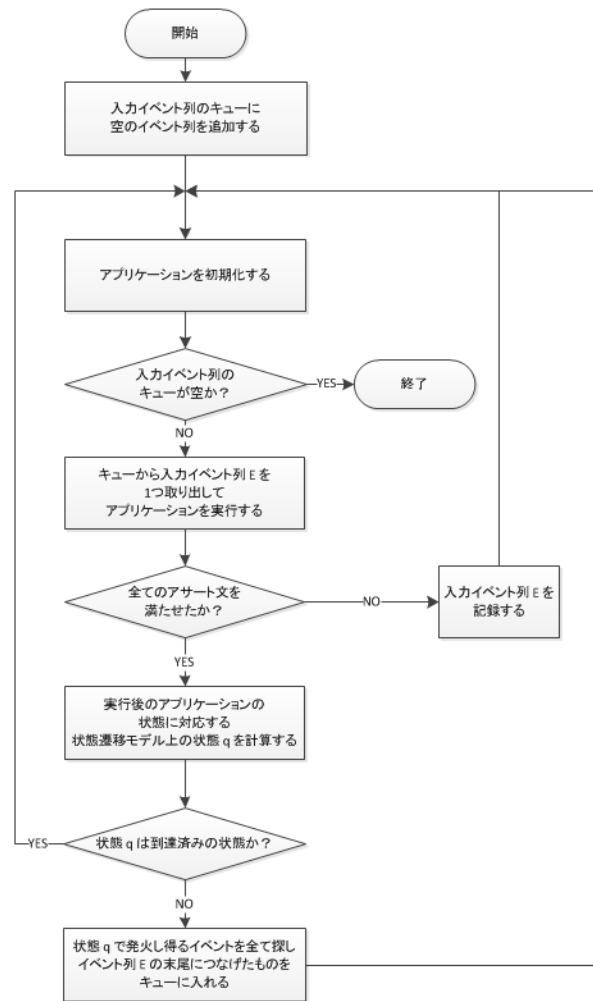


図 1. 状態遷移モデルの自動生成とテストの自動実行のアルゴリズム

デルを構築することである。アルゴリズムの出力は、アサート文を満たせなかった、すなわち異常動作を引き起こした入力イベント列の集合である。

キューから取り出したイベント列をテスト入力とし、テスト対象のアプリケーションに対して発火することでテストを実行する。テストの実行中にアプリケーションに埋め込まれたアサート文を満たせなかった場合は、使った入力イベント列を記録する。またアサート文を満たせなかった場合、アプリケーションは異常状態に陥っていると考えられるため、それ以降の探索は行わずに次のテストケースの実行に進む。

全てのアサート文を満たした場合にはアプリケーション

ンは正常に動作していると判断し、今のアプリケーションの実装上の状態がどの同値クラスに含まれるか、すなわち状態遷移モデル上のどの状態に対応するか計算する。得られた状態遷移モデル上の状態が既に到達したことのある状態であった場合は、それ以降の探索を行わず次のテストケースの実行に進む。新しい状態であった場合は、その状態で発火し得るイベントを全て探索し、それぞれを今の入力イベント列の末尾に追加した上でキューに加える。この処理をキューが空になるまで繰り返すことで、状態遷移モデル上で到達可能な全ての状態に到達できる。

テスト対象のアプリケーションの状態を同値分割する基準は、テスト対象のアプリケーションに合わせて開発者が定義する。例としてウェブブラウザを考えると、例えばアドレスバーに入力された URL の内容で同値分割を行うことが考えられる。この場合、同じ URL のウェブページを開いていれば、例えばページの読み込み状況やスクロール位置などは考慮されず、同一の同値クラスに丸められる。ここで同値分割の基準に URL とページの読み込み状況の組を使うように変更すると、ページの読み込み状況に応じて正しく UI が更新されるか、などのテストも行うことができるようになる。

状態遷移モデルの粒度は同値分割の粒度に依存し、状態遷移モデルを詳細にすると、得られるテストカバレッジは高くなるが、テスト実行時間は長くなる。同値分割の基準を調整しながら上記アルゴリズムの実行を繰り返すことで、テスト時間と得られるテストカバレッジの最適点を探すことができる。この方法には、状態遷移モデルの粒度を変更するためにテスト対象のアプリケーションの実装を変更する必要があるという利点がある。

同値分割の基準が決まれば、状態遷移モデルの構築やテスト実行は全て自動化することができる。実装のポイントは5章で述べる。

5 実装

GUI フレームワークである Qt [12] で書かれたアプリケーションを対象とするライブラリとして、提案手法の実装を行った。ただし、提案手法は Qt の機能には依存しておらず、一般の GUI フレームワークに対して実装可能である。

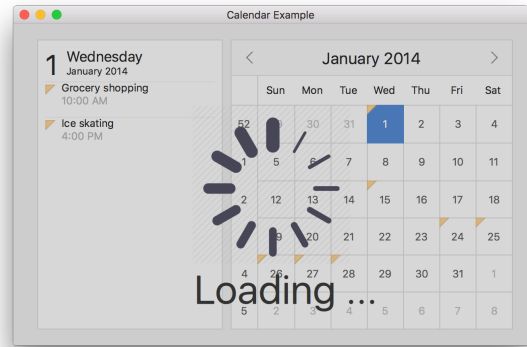


図 2. 上位のコンポーネントが下位のコンポーネントを隠す例

5.1 アルゴリズムの実装

提案手法を実開発に適用するためには、手法の導入・運用コストを低くすることと、テストの実行時間を抑えることが重要である。そのためには、開発者が実装しなくてはならない同値分割処理の実装コストを下げることで、生成する入力イベント列の数を必要最小限に絞り込むことが有効である。

同値分割の基準は、テストの実行時間やテストしたい内容などを考慮して何度も更新されるため、何度も実装を修正する必要がある。この修正コストを下げるために、自動生成された状態遷移モデルを開発者に見やすい形で提示し、開発者がそれを参考にしながら基準を更新できるようにする。

また、発生し得る全てのイベントを取得する処理の実装を工夫することで、状態遷移モデルを網羅できる入力イベント列を生成しつつも、重複していたり実際には起こりえない組み合わせの入力イベント列を取り除く。特に、実際には起こりえない入力イベント列を多く生成してしまうと、テスト実行時間が長くなるだけでなくテスト結果の偽陽性率が高くなってしまい、結果として開発者がテスト結果を無視するようになってしまうため、これを防ぐことは実用上非常に重要である。

以下では、これらの課題を解決するために行った実装上の工夫について述べる。

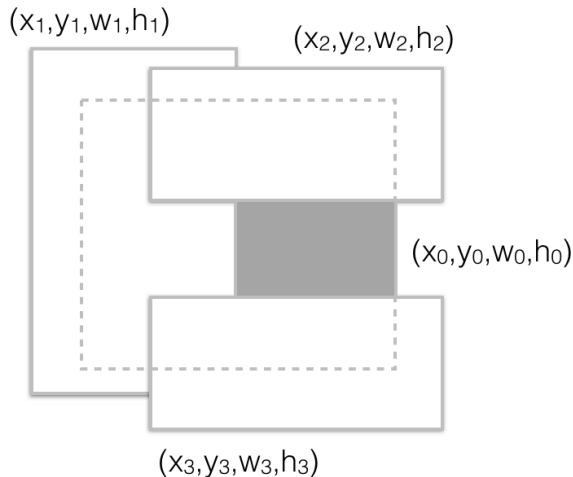


図 3. GUI コンポーネント同士の重なりと制約式

イベントハンドラを持つコンポーネントの座標が (x_0, y_0, h_0, w_0) , 重なった3つのコンポーネントの座標が $(x_i, y_i, h_i, w_i)_{[i=1,2,3]}$ で表されるとき, タッチ可能な座標 (x, y) は以下を満たす.

$$\begin{aligned} & (x_0 < x < x_0 + w_0) \quad \& \quad (y_0 < y < y_0 + h_0) \\ & \& \quad !((x_1 < x < x_1 + w_1) \quad \& \quad (y_1 < y < y_1 + h_1)) \\ & \& \quad !((x_2 < x < x_2 + w_2) \quad \& \quad (y_2 < y < y_2 + h_2)) \\ & \& \quad !((x_3 < x < x_3 + w_3) \quad \& \quad (y_3 < y < y_3 + h_3)) \end{aligned}$$

全イベントの取得処理 テスト対象のアプリケーションがある状態で発火し得る全てのイベントを取得する処理は, その状態で有効な全てのイベントハンドラを取得する処理と同等である. よって, アプリケーション内の全ての GUI コンポーネントを探索して, 登録された有効なイベントハンドラを抽出すれば, 発火し得る全てのイベントを得ることができる.

しかしながら, GUI コンポーネントを探索して得られた全てのイベントハンドラをそのまま利用すると, 得られるイベントの数が非常に多くなってしまう. テストの実行時間を抑えるために, テストに使用するイベント列の数を減らすことが実用上は重要である.

特定のイベントで任意の状態に到達する確率が実数で表されるとき, 任意の精度で確率を離散値に丸めることによって, SMT ソルバで効率的に解ける問題に変換する手法が知られている [13]. そこで, この手法を GUI アプリケーションに応用し, 探索対象にするイベント数を削減する. 具体的には, 一般に最も頻繁に生成されるイベントのひとつであるタッチイベントに注目し, GUI コンポーネント同士の重なりを考慮して探索対象にするイベントハンドラを絞り込むよう実装上の工夫を行った.

GUI アプリケーションではユーザの操作を意図的に制限するため, 画面の上位に GUI コンポーネントを重ねて下位のコンポーネントを操作させないようにすることがよくある. 例えば図 2 のアプリケーションでは, データ読込中はアプリケーション全体に半透明のエフェクトがかかり, アプリケーションを操作できなくなっている.

このような場合, 下位の GUI コンポーネントに対してタッチイベントが発生することはないため, それらを探索対象から取り除くことができる.

実際にタッチイベントが発生し得る GUI コンポーネントのみを選択するには, その GUI コンポーネント内に他のコンポーネントと重ならない領域が存在するかどうか, という充足可能性問題を解く必要がある. これは例えば, 図 3 中の色付きの領域を求めることに相当する. 本実装では SMT ソルバの Z3[14] を用いてこの問題を解き, 上記問題を満たせない GUI コンポーネントを取り除いた.

タッチイベントのイベントハンドラを探索する処理はライブラリ関数として提供され, 開発者が実装する必要はない. また, テスト対象のアプリケーションに固有のイベントを探索対象に加えられるよう, 開発者が実装したイベント探索関数を簡単に並用できるように実装を行った.

同値分割の基準 同値分割の基準は開発者が定義, 実装する必要がある. 基準は通常のプログラムコードとして記述する. 具体的には, 今のアプリケーションの状態を受け取り, 所属する同値クラスを返す関数である. この処理はテスト対象のアプリケーションとリンクし, テスト実行時に使用される. 基準の作成には, アプリケーション内の任意の情報を使うことができる.

テストの自動実行 入力イベント列に含まれるイベントを順番に発火させることでテスト対象のアプリケーション

表 1. 提案手法を組み込んだ開発プロセス

-
- 1). アプリケーションの設計
 - 2). アプリケーションのコーディング, アサート文の挿入
 - 3). アプリケーションの実装上の状態を同値分割する基準を定義
 - 4). 状態遷移モデルの自動生成とテストの自動実行
 - 5). 不具合が発見された場合, アプリケーションを修正
 - 6). 得られた状態遷移モデルを元に, 必要であれば手順 2 から繰り返し
 - 7). アプリケーション開発の進捗に合わせて手順 2 もしくは手順 3 から繰り返し
 - 8). 機能検査を実施
-

ンをテスト実行し, アプリケーションの最終状態を得る. この際, イベントハンドラの絞り込みを行った際に得られた実際にタッチ可能な座標の具体的な値をテスト実行時に利用することで, ダミーの座標を使ってタッチイベントを生成する場合に比べてユーザ操作の再現度を高めた. この処理はライブラリ関数として提供されるため開発者が実装する必要はない.

状態遷移モデルの出力 テスト終了後に同値分割の基準を修正するコストを下げるため, アプリケーションの状態とイベントをそれぞれノードとエッジとした状態遷移図として, 構築した状態遷移モデルを出力するようにした. 出力形式は HTML で, 開発者はブラウザ内でノードを自由に動かして内容を確認することができる. ノードにはアプリケーションのスクリーンショットを使い, 生成された状態遷移モデルを開発者が理解しやすくした.

5.2 開発プロセスの実装

提案手法を追加した開発プロセスの例を表 1 に示す. 提案手法はコーディングと同時に実行する. 提案手法を適用するために追加で必要なプロセスは手順 3 から手順 7 で, 追加で作成が必要なドキュメントはない.

手順 4 では, 手順 3 で定義される状態遷移モデル上の状態のうち, 初期状態から到達可能な全ての状態を探索し, 入力イベントをラベルとした状態遷移モデルとして出力する. 探索中にアサート文を満たせない入力イベントの列が見つかった場合には, その入力イベントの列も合わせて出力する.

手順 3 から手順 5 まで実行したあとは, 必要に応じて状態遷移モデル上の状態の割当てを変更して手順 3 から再実行する. 状態遷移モデル上の状態を詳細に分割すれば, より詳細にテストを行うことができるが, それに応じてテストの実行時間は増加する. 詳しく確認したい部分に関する状態だけを詳細に分割することで, 段階的に状態遷移モデルの粒度を変更することができる. その際, 開発者は手順 4 で得られた状態遷移モデルを参照することで, 状態分割を効率的に行うことができる.

6 評価

USB メモリ内の写真の一覧表示と拡大表示, スライドショーが行えるイメージビューアアプリケーションに対して提案手法を適用し, 評価を行った. このアプリケーションは Qt/QML で実装されており, 437 行の C++ コードと 363 行の QML コードからなる. アプリケーションのスクリーンショットを図 4 に示す.

このアプリケーションは, USB メモリが挿入されると USB メモリ内の写真を探索して一覧表示する. 一覧表示された写真をタッチするとタッチした写真が拡大表示され, その後任意の場所をタッチすると一覧表示画面に戻る. 右下のスライドショー開始ボタンをタッチするとスライドショーが始まり, その後任意の場所をタッチすると一覧表示画面に戻る.

このアプリケーションは, 通常のタッチイベントに加え, USB メモリの抜き差しという固有のイベントを持つ. そのため, USB メモリの抜き差しを表すイベントを探索する関数を実装し, 探索対象に加えた. この探索関数は 100 行程度で実装することができた. USB メモリの抜き差しを表すイベントが発生した場合の動作は, 結合テスト用に実装されていたスタブ実装をそのまま流用した.

また, アプリケーションが満たすべき性質として, 以下のようなアサート文を複数埋め込んだ. 埋め込むアサート文の量に制限はなく, 多ければ多いほど不具合を見つけやすくなる.

- USB メモリ内に画像が 1 枚もないときに, スライドショーボタンが無効になっている
- 拡大表示中もしくはスライドショー表示中に画像が 0 枚になることがない
- 拡大表示中に直接スライドショー表示に遷移しない

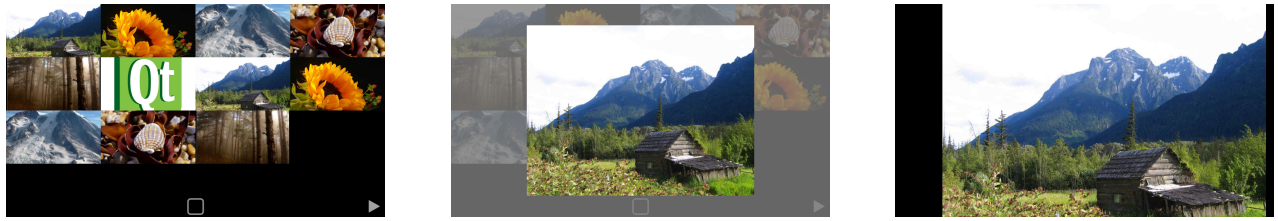


図 4. イメージビューアのスクリーンショット (左から、一覧表示、拡大表示、スライドショー)

評価は 2.4 GHz Intel Core i7 CPU と 16 GB のメモリを搭載したマシンを用いて行った。

試行 1 回目 Qt/QML には、内部状態の変化に応じて GUI を変化させるという動作を実装しやすくする機能がある。この機能を使うと、開発者は内部状態を表す変数の値を変更するだけで、GUI コンポーネントの表示・非表示を切り替えたり、スクリプトやアニメーションが駆動させたりすることができる。テスト対象のアプリケーションもこの機能を使って実装されていた。

そこで、この機能のために実装されていた内部状態を表す変数の値を、同値分割の基準として利用することにした。このアプリケーションは、サムネイル表示中か拡大表示中かを表すブール型の変数と、スライドショー表示中かを表すブール型の変数の 2 つの状態変数を持っていたため、アプリケーションの状態は 4 つに同値分割された。

この同値分割の基準でテストを実行したが、不具合を見つけることはできなかった。得られた状態遷移モデルを図 5 に示す。

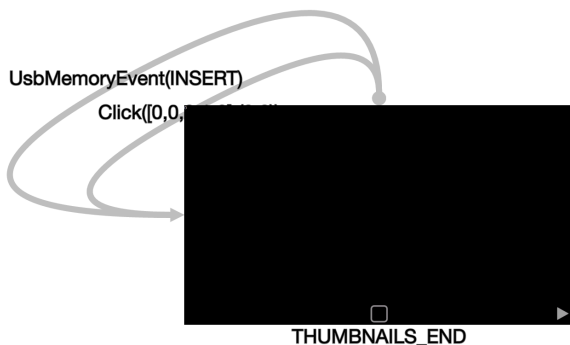


図 5. 1 回目の試行で得られた状態遷移モデル

図 5 において、ノードは各状態におけるアプリケー

ションのスクリーンショットで、エッジはその状態遷移を起こすイベントの文字列表現で表される。エッジの名称は開発者が必要に応じて自由にカスタマイズできる。また状態遷移モデルはブラウザで表示され、ノードを自由に動かして内容を確認することができる。

図 5 を見ると、スライドショーボタンをクリックしても、USB メモリを挿入しても、同一の初期状態に遷移してしまっており、全てのテストを実行しても初期状態にしか到達できていないことがわかる。これは状態の同値分割の基準が荒すぎたことが原因であり、これでは十分にテストを実行できているとは言えない。

試行 2 回目 1 回目の試行の結果を考慮し、同値分割の基準をより詳細にすることにした。アプリケーションの動作仕様を考慮すると、USB メモリが挿入されているかどうかで GUI が大きく変化するにも関わらず、図 5 では、USB メモリの有無に関係なく同一の状態になってしまっている。

そこで、USB メモリが挿入されているかどうかを同値分割の基準に加えることにした。具体的には、USB メモリが挿入されているかどうかを表すブール型の変数を基準に加え、アプリケーションを 8 個の状態に同値分割した。この同値分割を行う関数は 50 行程度で実装することができた。

この同値分割の基準でテストを実行したところ、拡大表示中またはスライドショー表示中に USB メモリを抜くと、画像が 0 枚になったにも関わらず拡大表示またはスライドショー表示画面にとどまってしまう、画面表示が乱れてしまうという不具合を見つけることができた。得られた状態遷移モデルを図 6 に示す。

このアプリケーションを搭載した環境では、USB メモリが取り外されるとアプリケーション自体が非表示とされていたため、この不具合は顕在化していなかった。提案手法により潜在的な不具合を発見することができた。提案手法は有効であると言える。

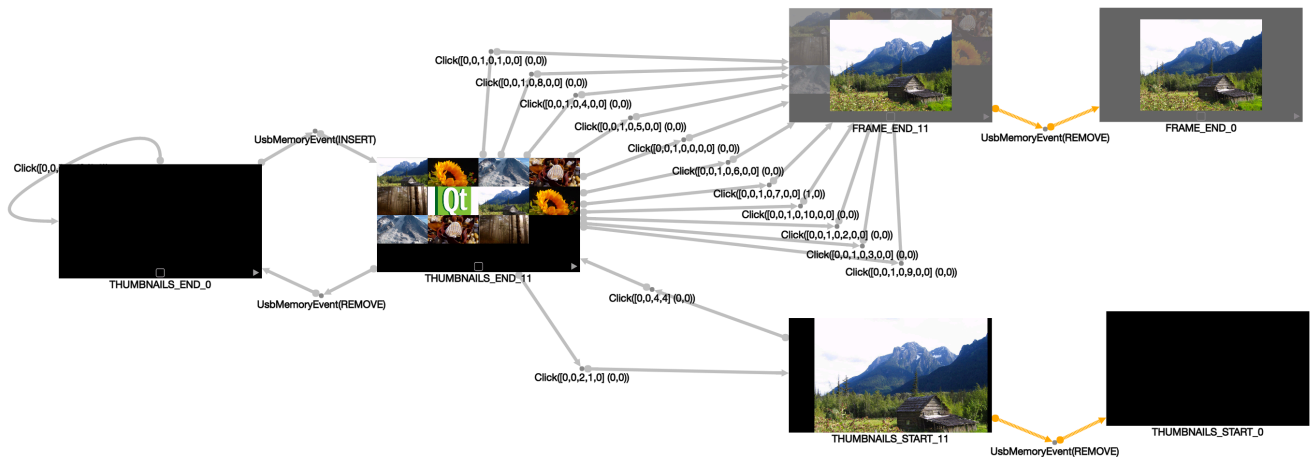


図 6. 2 回目の試行で得られた状態遷移モデル

試行 3 回目 より詳細にテストを実行するため、拡大表示されている写真の名称を同値分割の基準に加えることにした。その結果、不具合に至る操作系列を 23 個見つけることができたが、これらは全て試行 2 回目で見つめられた不具合と意味的に同一のものであった。

試行 2 回目の結果と比較して不具合に至る操作系列の数が増えている原因は、たとえば「1 枚目の写真を拡大表示中に USB メモリを抜く」という操作と「2 枚目の写真を拡大表示中に USB メモリを抜く」という操作が区別されているためである。しかしながら、これらは同じ不具合を複数の操作系列で再現しているに過ぎず、冗長な出力であり、開発者が結果確認にかかる工数を増加させてしまう。またテスト自体の実行時間も大幅に増加した。

この結果、どの写真が拡大表示されているかという状態はこのアプリケーションの動作においてはあまり重要ではなく、これを同値分割の基準に加えることは不適切であることがわかった。

性能評価 3 回の試行でのテスト実行結果と、比較のために Z3 によるテストケースの削減を行わなかった場合の試行 2 と試行 3 のテスト実行結果を表 2 に示す。

表 2 において、同一の不具合を起こす操作系列が複数得られた場合は、重複として数えている。また別の GUI コンポーネントの下に隠れた GUI コンポーネントを操作するなど、実際には起こりえない操作を実行した結果、不具合状態に至ったものは偽陽性として数えている。Z3

を使うことで、偽陽性率を低く抑えられることが確認できた。

重複や偽陽性が増加すると開発者が結果確認にかかる工数が増加するため、これらを低く抑えつつ、不具合数を増やせる同値分割の基準をうまく探す必要がある。今回の 3 回の試行の中では 2 回目の基準が最適であったが、よりよい基準が存在する可能性がある。

7 まとめと今後の課題

これまで自動化が難しかった GUI アプリケーションのテストに対し、導入コストを抑えて既存の開発プロセスに容易に組み込めるテスト自動化の手法を提案した。経験上、アプリケーションの動作モデルを開発者が手動で作成することはコストが高く、また既存のアプリケーションの仕様はモデルを導出できるほど形式的に書かれることはない。そこで、テスト対象のアプリケーションのソースコードからモデルとして状態遷移モデルを自動で構築し、それを元に網羅的にテストを実行するアプローチを取った。

提案手法の実装にあたり、開発者の負担を抑えることと、テスト結果の偽陽性を抑えることを重視した。実装にコストがかかる入力イベント列の生成部分をライブラリとして提供するほか、GUI コンポーネントの重なりを考慮して余計な入力イベント列を生成しないようにした。

提案手法を過去に開発した試作アプリケーションに対して適用した。アプリケーションは大量の状態を持ち、かつ多くのイベント処理を持つが、現実的な時間で網羅

表 2. 各試行のテスト結果

	到達状態数	実行回数	実行時間	検出された不具合の候補			
				総数	不具合数	重複	偽陽性
試行 1 回目	1	5	4.5 秒	0	0	0	0
試行 2 回目	6	44	36.5 秒	2	2	0	0
試行 2 回目 (Z3 なし)	7	90	84.6 秒	14	2	0	12
試行 3 回目	70	226	527.9 秒	23	2	21	0
試行 3 回目 (Z3 なし)	81	1011	970.2 秒	166	2	21	143

的なテストケースを自動生成することができた。またテスト実行の結果、未知の不具合を見つけることができた。提案手法は有用であると言える。

今後の課題としては、状態の同値分割の基準の決め方と、タイマーイベントやアニメーションの扱い、イベントハンドラ内での分岐の扱いが挙げられる。

状態の同値分割の基準の決め方 同値分割の基準の定義により、テストの網羅度や実行時間は大きく変化する。網羅度とテスト実行時間のトレードオフの中で最適な基準を見つけることが重要であるが、これを機械的に探索する方法は確立できておらず、開発者の経験によるところが大きい。

同値分割の基準を機械的に決める方法を確立することは今後の課題である。

タイマーイベントやアニメーションの扱い タッチイベントが発生してから時間をおいて UI から応答が発生する場合に、イベントを取りこぼす可能性がある。このような特性を持つイベントハンドラを扱う仕組みを構築することは今後の課題である。

イベントハンドラ内での条件分岐の扱い 現状の実装では、イベントハンドラ内に条件分岐がある場合、全てのパスを網羅的に実行することは保証されない。条件式そのものや、条件式中に現れる変数を同値分割の基準に加えることで、イベントハンドラ内での条件分岐も網羅できる可能性がある。

参考文献

- [1] 独立行政法人情報処理推進機構 技術本部ソフトウェア・エンジニアリング・センター監修, <第2版>ソフトウェア開発データ白書 2010-2011, 独立行政法人情報処理推進機構ソフトウェア・エンジニアリング・センター, 2011.
- [2] 日本情報システムユーザー協会, ソフトウェア開発管理基準に関する調査報告書 (ソフトウェアメトリックス調査), 日本情報システムユーザー協会, 2011.
- [3] “Selenium”. <http://www.seleniumhq.org/>.
- [4] 平井潤, 関根智, 川野晋一郎, “ソフトウェアのテスト効率と精度を向上させる gui 画面の自動操作技術,” 東芝レビュー, vol.63, no.6, pp.36–39, 2008.
- [5] “RanoRex”. <http://www.ranorex.com/>.
- [6] “TestComplete”. <https://smartbear.com/product/testcomplete/>.
- [7] E. Bringmann and A. Kramer, “Model-based testing of automotive systems,” Software Testing, Verification, and Validation, 2008 1st International Conference on IEEE, pp.485–493 2008.
- [8] K. Claessen and J. Hughes, “Quickcheck: a lightweight tool for random testing of haskell programs,” Acm sigplan notices, vol.46, no.4, pp.53–64, 2011.
- [9] E. Alégroth, “Random visual gui testing: Proof of concept,” SEKE, pp.178–183, 2013.
- [10] A. Developers, “Ui/application exerciser monkey,” 2012.
- [11] A. Rauf, A. Jaffar, and A.A. Shahid, “Fully automated gui testing and coverage analysis using genetic algorithms,” International Journal of Innovative Computing, Information and Control (IJICIC) Vol, vol.7, pp.●●–●●, 2011.
- [12] J. Blanchette and M. Summerfield, C++ GUI programming with Qt 4, Prentice Hall Professional, 2006.
- [13] M.N. Rabe, C.M. Wintersteiger, H. Kugler, B. Yordanov, and Y. Hamadi, “Symbolic approximation of the bounded reachability probability in large markov chains,” Quantitative Evaluation of Systems, pp.388–403, Springer, 2014.
- [14] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” Tools and Algorithms for the Construction and Analysis of Systems, pp.337–340, Springer, 2008.