成蹊大学理工学研究報告
J. Fac. Sci.Tech., Seikei Univ.
Vol.51 No.1 (2014)pp.49-56

画面遷移を伴うシナリオ分岐型アプリケーションのための実行時テスト手法の提案

山本　拓哉[*1]，甲斐　宗徳[*2]

Proposal of Runtime Test Approach for Scenario Branch Application with Screen Transitions

Takuya YAMAMOTO[*1], Munenori KAI[*2]

ABSTRACT：In development of software which become larger-scale and more complicated every year, the importance of software test grows more and more. However, some domains where it is difficult to apply any software test exist. One of such domains is a GUI test. In order to conduct a GUI test, the method of "preparation of the complicated test code in con-sideration of various states", "the software test in manual operation", and "automating using expensive software" is taken. Since those methods are very high-cost, the method of describing a software test easily is desired. However, it is generally difficult to test to all the software containing GUI at the runtime. Then, in this paper, we propose runtime test approach for scenario branch applications with screen transitions like ATM screen or software installers. Proposed method can conduct software test for such specific applications easily and at low cost. It is also shown that most performance degradation by applying the proposed method to actual applications cannot be found.

Keywords：automated software test, unit test, user interface, screen transition

(Received March 31, 2014)

1．はじめに

1．1　研究背景

年々大規模化および複雑化しているソフトウェア開発において，ソフトウェアテストの重要性がますます高まっている。ソフトウェアテストは開発者側の視点で見ると，不具合の修正やリファクタリングによるエンバグやデグレードからの不安の低減，ソフトウェアの品質面から見るとテストが仕様を満たすように記述されていればその品質は満たすことが出来るというメリットが存在する。しかしソフトウェアテストはソフトウェア開発においてすべてのソースコードに対して行うことが出来るわけではない。アルゴリズムのように入力と出力を数値で定義しやすいものに対しては非常にテストを行いやすいが[1]，GUI の様な入力が画面操作，出力が画面の状態であるような，数値として定義が難しいものはソフトウェアテストを定義することが非常に難しい。このためGUIのテストは操作を定義するための複雑なコードを記述して自動化するか，人力によるソフトウェアテストを行わなくてはいけない。それらの方法は非常にコストが高く，容易にソフトウェアテストを記述する方法が望まれている。しかしながらすべてのGUI を含むソフトウェアに対応させることは現状では難しいため，本研究では決まった操作と画面を多く持つ画面遷移を伴うシナリオ分岐型アプリケーションにおいて容易にソフトウェアテストが出来るような手法を提案する。画面遷移を伴うシナリオ分岐型アプリケーションはGUI テストの困難さと同じ問題を抱えているが，同じような操作を繰り返すという性質を持っている。「似たような操作を繰り返す」ということは自動化が行いやすいということであるが，この繰り返しを如何に抽象的に様々な画面遷移を伴うシナリオ

*1：理工学研究科理工学専攻博士前期学生
*2：理工学研究科理工学専攻教授（kai@st.seikei.ac.jp）

分岐型アプリケーションに適用するかが問題となる。この抽象化の問題を解決することによって自動化を行うことが出来るため，テストコード化を行うことが出来，コストの低減が見込まれる。

1．2 目的

本研究では「動作しているアプリケーションにおける遷移の自動化」を目的としている。具体的にはシナリオ分岐型アプリケーションに共通する必要最低限の項目をモデル化し，各アプリケーションはそのモデルを派生させて適した実装を行う。そしてテスト実行環境がそのモデルを元に実際にアプリケーションを起動させ，自動的に入力を行い，想定している結果が得られているか検査を行えるようにすることである。自動化が出来ることによってヒューマンエラーを減らすことが出来ることや，正確なコントロールにより何度でも同じ状況が再現可能という利点が挙げられる。また実際にアプリケーションを起動させてテストを行うことにより実際に使用しているような環境でテストをすることが出来，アプリケーションが起動しないと読み込まれないようなライブラリ(GUI ライブラリなど)による不具合の発見や再現なども可能となる。

更にモデルは特定のシナリオ分岐型アプリケーションに特化した限定的なモデルではなく，様々なシナリオ分岐型アプリケーションに適用できるようになっている。共通のモデルを使用出来る利点は，あるアプリケーションのモデルの調査に使用したアプリケーションやアルゴリズムをシームレスに他のアプリケーションに適用できることである。

2．画面遷移を伴うシナリオ分岐型アプリケーション

画面遷移を伴うアプリケーションとは，ユーザが何かしらの行動を起こすことによって画面の状態が変わっていくアプリケーションのことである。画面の状態の変化とは表示される内容の変化であり，テンプレートを用いたような同じ構成で一部の中身が差し替わるものから，ページの構成が全部変わるものまで様々である。基本的には「画面を確認-> 選択や入力-> 次の画面へ移動する行動」という操作を一画面の中で行い，この動作を最後の画面が出てくるまで何度も繰り返す。また操作が可逆（前の操作の取り消しまたは修正が行える）であれば前の画面に戻ることも可能になっていることもある。シナリオ分岐型アプリケーションとは，先ほどの行動の「選択や入力」の結果に応じて「次の画面へ移動する行動」を行った場合に次に出てくる画面が変わるアプリケーションのことである。アプリケーションの種類としては，当研究室で研究を行っているWeb学習システム[2]のX-Webやアドベンチャーゲーム,インストーラ,ATMなどが挙げられる。次にアドベンチャーゲーム,インストーラで以下の項目がどのようになっているか説明を行う。

3．GUI を含んだテスト手法の適用と問題点

現在単体テストは幅広いプロジェクトにおいて採用されているが，その適用範囲はほとんどがアルゴリズム，ロジックに限定されており，GUI を伴うテストには適用されない場合が多い。これは 1.1 節でも述べたとおりGUI の状態の定義や操作が非常に複雑だからである。しかしいくつかの方法を用いてGUI を含んだテストを行うことは可能である。もしそのような方法を用いて画面遷移を伴うシナリオ分岐型アプリケーションのテストを行った場合どのように行うかと，どのような問題点が発生するかについて説明を行う。

● マクロ

マクロはマウスやキーボードの動作を記録し再生する手法である。ソフトウェア自体がマクロをサポートすればテストの自動化を行うことが出来るが，画面遷移を伴うシナリオ分岐型アプリケーションにおいてユーザに自動化を提供するメリットはあるとは言えないため，デバッグ用の為だけにマクロをサポートするのはコストが非常にかかる。また動作する環境は様々であり正確なタイミングで操作を行うことが難しい。

● UI Automation

UI Automation とはUI の自動操作機構であり.Net Framework やiPhone SDK のように公式でサポートされているものから，MarathonやSWTBot の様にサードパーティーのライブラリなど様々なものが存在する。UI Automation を利用するとマクロに比べて非常にGUIの操作は行いやすいが，API は非常に抽象化されており実際に使用するためにはキャストやリフレクションを多く行わなければいけない。このため画面構成変更によるテストコードの変更に非常に弱いという弱点がある。またゲームの様にUI を自分で描写する場合などは，その機能を実装した場合を除いてUI Automation を利用することが出来ない。

● モデル検査

モデル検査は本研究とは目的が異なっているが，誤解されやすい部分があるため補足を行う。モデル検査とは，モデルの取り得るすべての状態においてある条件を満たすかどうかを検査する方法である。単体テストはモデルの性質の一部を調べるには非常に有効であるが，モデル全体の性質を書きたい場合は単体テストが不向きである。モデル検査はこのすべての組み合わせの中で条件を満たすかどうかを調べることに非常に有効かつ強力な手法であるが，高度な知識と膨大な計算資源が必要となる。

4．提案するテスト手法

本研究で提案するテスト手法は安価な方法でテストを行えることを目的としているため，特別な外部ツールや手動で行っていたテストという方法を用いず，既存のユニットテストで簡単に行えるようにする至ってシンプルなものである。具体的には

1. 様々な画面遷移を伴うシナリオ分岐型アプリケーションで使用可能な画面遷移モデルの作成
2. 画面遷移モデルに基づいて遷移を行う遷移マシンの作成
3. 遷移マシンを自動操作する機能の作成
4. 自動操作機能をユニットテストフレームワークに組み込みやすくするための機能の作成

を行うことによって実現する。これにより「GUI を含んだテスト手法の適用と問題点」において挙げた

1. 入力デバイスからの正確な入力方法
2. 特定フレームワークへの依存
3. コードのメンテナンス性や型安全性
4. 高度な専門知識や高価なツール

という問題点を解決する。

4．1　従来のモデルの問題点

現在利用されている画面遷移を伴うシナリオ分岐型アプリケーションは，それぞれ独自のモデルを採用している。独自モデルは利用されるアプリケーションの記述のしやすさにおいて非常に優れているが，外部ツールなどを使用して処理を行いたい場合に非常に扱いが難しい。

4．2　画面遷移のモデル化

画面遷移を伴うシナリオ分岐型アプリケーションで利用されるモデルは，スクリプトもしくはXML などのデータ構造によって構成されている。画面遷移を伴うシナリオ分岐型アプリケーションにおいてスクリプトが利用されることが多い。これはDSL を利用してアプリケーションの内部を効率的に制御することに重点が置かれていることと，スクリプトエンジンさえあればコンパイラなどの開発環境が不要という点においてメリットがあるからである。しかしDSL であるために不得意もしくは出来ないという可能性が出てくる。このため提案するモデルではスクリプトを利用せずXML などのデータ構造でモデルの定義を行う。次にモデルで使用する要素の説明を行う。

4．2．1　ViewPoint

ViewPoint はテキストや画像など選択（ユーザに入力を受け付ける場所が無い）を伴わない画面で構成されているものを表現するためのモデルである。ViewPoint は PointId とNextPointId の二つのプロパティを定義している。PointId はその画面を表すユニークなId である。PointId は遷移を行う際に遷移場所指し示すNextPointId の指定先として使われる。遷移についての情報はPointId とNextPointId の関係でしか持たない。画像やテキストなどの情報はそのアプリケーションによって異なるため，それぞれのアプリケーションがViewPoint を継承してプロパティを付け加える。

4．2．2　SelectionPoint

SelectionPoint はテキストや画像の他にもユーザの入力を受け付けるUI を持っている画面で構成されているものを表現するためのモデルである。SelectionPoint は ViewPoint を継承しているためViewPoint が持っているプロパティの他にもSelections を定義している。SelectionsはISelection を複数保持している。ISelection は選択肢を表すものでSelectionId というプロパティを持っている。SelectionId はSelection をインスタンス化したときに選択肢を分別するためのId として利用される。

4．2．3　BranchPoint

BranchPoint は画面遷移を行う際に分岐が必要なときに利用するモデルである。ViewPoint とSelectionPoint との最大の違いは画面表示に関する情報を一切持たず遷移先を複数持っているということである。ViewPointとSelectionPoint は遷移先を一つしか持たないため複数の

遷移候補がある場合に分岐処理を行うことが出来ない。そこで遷移先の分岐を行いたい場合は，ViewPoint とSelectionPoint のNextPointId をBranch のPointId に設定する。BranchPoint はPointId とNextPointIds の二つのプロパティを定義している。PointId はViewPoint やSelectionPoint と同じ役目を持っている。NextPointIds は複数の遷移先，つまり複数のNextPointId を持っている。Branch はViewPoint やBranchPoint と同じように，遷移条件に関する情報（BranchPoint の場合は遷移候補のどれに遷移を行えば良いか）を持っていない。これはアプリケーションによって遷移候補先に使用される判断の情報は様々だからである。BranchPoint が遷移条件とどのように遷移先を決めるのかについては 4.6.1 節で説明を行う。

4．3　遷移表現

画面遷移を伴うシナリオ分岐型アプリケーションで使用される遷移構造は様々なものが考えられる。遷移構造はどんなに複雑でもいくつかの基本的な遷移構造を使用して作られている。ここでは遷移構造の基本となる三つのパーツである「逐次」「分岐」「ループ」を先ほど説明した遷移モデルを使用してどのように構成するかについて説明を行う。

4．3．1　逐次

逐次は一直線であり制御を含まない遷移構造である。画面を表すView-Point は図 1 の様に自分自身を表すPointId と，遷移先を指定するNextPointId を持っている。それらを「画面P1 はP2 遷移，画面P2 は画面P3 へ遷移，画面P3 は遷移しない」のように現在の画面が次へ遷移したい画面につなぐことによって逐次構造を表現する。画面の終端はNextPointIdに該当するものが無いため空白またはNull にして終端を表す。

図 1　逐次モデル

4．3．2　分岐

分岐は条件によって複数有る画面遷移先候補のどれかに遷移行う為の遷移構造である。分岐を表すBranch はViewPoint などと違い，図2 のBranch の様にNextPointIdsに遷移先を複数持つことが出来る。それを別々の画面につなぐことによって分岐を表現する。

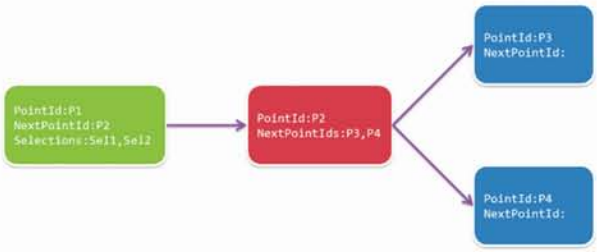

図 2　分岐モデル

4．3．3　ループ

ループはある条件を満たすまで繰り返し同じ画面を表示したいときに使用する遷移構造である。ループは分岐と同じくBranch を使用して表現を行うが，自分がたどってきたものを指すのでは無く図 3 の様に今までたどってきたものを指すことによって表現する。

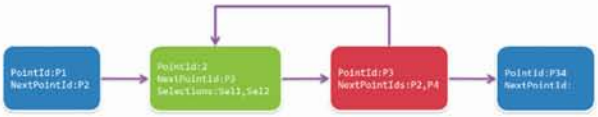

図 3　ループモデル

4．4　提案する遷移モデルの利点

提案する遷移モデルの利点として，モデルに透過的な操作が行えるということが挙げられる。従来はそれぞれが独自のモデルを採用していたため，モデルに対して操作を行うためには独自モデルに対応したプログラムを書く必要があった。しかし今回提案したモデルを使用したアプリケーションを作成した場合，一度遷移モデル用のソフトウェアを作成したらこの遷移モデルを使用するソフトウェアすべてに使用可能となる。

4．5　遷移条件を持たない理由

Scene/SelectionScene は，ユーザがUI などを入力デバイスなどで操作して「遷移」を行うことを明示する場合に移動が行われる（インストールの進捗状況を表示する画面など自動で遷移する場合もある）。この遷移条件はアプリケーションによって異なっており，ほとんどの場合はマウス，キーボード，タッチパネルのいずれかからの入力と考えられるが，場合によっては特殊なデバイスからの入力ということもあり得る。もしこの様な遷移条件をモデルに組み込んだ場合，モデルは入力デバイスに依存することになってしまう。モデルはどの入力デバイスで遷移を行おうとも，モデルは遷移の関係だけを決定するべきである。このためScene/SelectionScene は遷移条件を持たないようにしている。ではどこに遷移条件を記述するかというと，UI からの入力を受け取りモデルを操

作する部分に記述を行う。これはMVC パターンのController 部分に該当する。こうすることによりモデルとUI は完全に切り離されることになり，それぞれの再利用性が高まる。

Branch はユーザに表示する情報を持っていないためScene/SelectionScene の場合とは別の理由が存在する。Branch はその場所へ遷移すると，どの遷移先に行くかを決定する。このどこへ行くかという遷移条件は，モデル内の情報で完結するものからPC 内情報やネットワークを使用したものまで様々なものが考えることが出来る。モデル内の情報で簡潔するならば個数や論理演算などのモデルを定義することによって遷移条件を持たせることで対応出来るが問題はモデル外の情報を使う場合である。本システムは巨大なライブラリを使用してモデルの開発を行いやすくすることは目的としておらず，テストの自動化（自動遷移）を行いやすいシステムを開発することを目的としている。モデル外の情報を使用するならば，遷移条件の判断はモデル外のロジックに移譲すべきである。このためBranch は遷移条件を持たない。

4．6　遷移の管理

遷移モデルの説明を行ったが，モデルだけでは遷移を行うことが出来ない。遷移を行うためには遷移モデルから適切な情報を取得し遷移に関する情報を管理する必要がある。ここでは先ほどの遷移モデルを利用して，実際に遷移を管理するためのクラスの説明を行う。遷移を行うクラスはScenarioReader,RouteSelector の二つによって構成されている。

4．6．1　ScenarioReader/RouteSelector

ScenarioReader は遷移動作に関する処理を管理しているクラスである。ScenarioReader は遷移を行うために，Scenario,IRouteSelector,StartPointId の三つが必要となる。Scenario は遷移モデルで説明を行ったとおり，遷移に関する情報を持っているモデルである。RouteSelector はBranch において遷移先を決定するものであり，現在地(Branch) を渡されると現在の状況を判断して行き先を決定するものである。またRouteSelector はアプリケーション固有の処理であるため，アプリケーション側が実装する必要がある。StartPointId はScenario の中でどの位置から遷移をスタートするかを指定するものである。ScenarioReader はこの三つの情報を用いて，次の画面へ遷移させる関数であるNext 関数が呼ばれる毎に遷移を行っていく。

4．7　自動遷移

本システムの代表的な機能となる自動遷移はアプリケーション内のScenarioReader を操作することにより実現する。自動遷移に必要な「停止条件」「進行条件」「遷移操作」はTransitionAutomator がすべて請け負う。また直接アプリケーション内部のScenarioReader を操作してしまってはアプリケーションの実装を無視して遷移することになってしまう。そこでSystemConnector というTransitionAutomator とアプリケーションを仲介する役目を持ったクラスを経由することで，アプリケーションの実装を気にすること無く遷移を行う。次にSystemConnector とTransitionAutomator の詳細について説明する。

4．7．1　SystemConnector

SystemConnector はアプリケーションとTransitionAutomator をつなぐクラスである。このクラスが必要となる理由は，TransitionAutomatorがSystemConnector 経由しないと実現できないことや不都合なことがあるからである。

一つ目は選択肢に関する取り扱いである。ScenarioReader は選択肢を受け取る機能は無いため，選択肢はアプリケーションのどこかのメソッドが受け取って処理を行う必要がある。そこでTransitionAutomatorはアプリケーションに何かしらの方法を用いて選択肢を渡す必要がある。このため選択肢をTransitionAutomator から受け取り，アプリケーションの適切なメソッドに処理を行わせるために必要となる。

二つ目はScenarioReader を操作する手段である。通常ScenarioReader は外部に公開する必要性が無いため操作する手段が無い。そこで何かしらの手段を用いてScenarioReader を操作する必要がある。またScenarioReader を直接操作して良いかどうかの問題も存在する。たとえば遷移する前に特定の処理（たとえばロギングなど）を行わなければいけない場合は，直接ScenarioReaderを操作することは出来ない。このため適切な手順を用いて遷移を行うためにSystemConnector が必要となる。このクラスはアプリケーションの内部実装に依存する処理のため，アプリケーション側が実装する必要がある。

4．7．2　TransitionAutomator

TransitionAutomator は 4.7.1 節で説明するSystemConnector を経由してアプリケーションの内部状態を取得操作することによって自動遷移を行うクラスである。TransitionAutomator は選択肢情報と停止情報の二つの情

報を元に遷移を操作していく。遷移方法はPull とPush の二種類の方法があり，Pull はアプリケーションの更新処理に組み込んで更新毎に遷移，選択肢の決定，停止の判断を行っていく。Push はアプリケーションとは別スレッドで動き，自分自身で定期的に更新を行いながらアプリケーションに遷移，選択肢の決定，停止の情報を送っていく方法である。

5．評価

5．1　評価方法

今回作成したシステムの評価として，アドベンチャーゲームを作成し自動遷移が行えるか確認を行った。ゲームのシナリオモデルは図 4 の様な構造になっている。S1, S2, S3 以外の四角形は画面のみで構成されたChapter を表しており，S1, S2, S3 の四角形はそれら以外の四角形の構造に選択肢画面を一つ含んだChapter を表している。シナリオモデルの大きさはSmall,Middle,Largeの三種類を用意し，それぞれのChapter が 20,500,1000 個の画面を保持している。このモデルを使用するゲームに対してテストを実行して自動遷移が行えるか確認を行った。このテストはS1 で「Selection1-1」を選択しS2 で「Selection2-0」を選択した場合にE2-5 の最後に到達出来るかどうかを行うテストである。使用したテストコードは図 5 である。

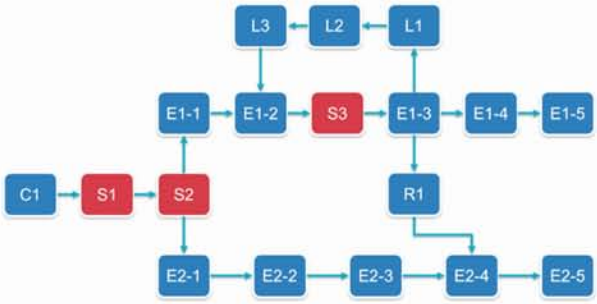

図 4　評価のシナリオモデルの概要

```
var connector = new GameConnector();
{
    using (var automator = new PullTransitionAutomator (
          connector,new List<Condition>
              {
                  new Condition("Selection1", new Selection () {
                  SelectionId = "Selection1-1" }) ,
                  new Condition ("Selection2", new Selection(){
                  SelectionId = "Selection2-0" })
              },new Scene() { PointId = "Ending 0-4-199" }))
    {
        var save = new SaveData();
        var game = new Game(
             automator ,
             (manager , spriteBatch) =>
             {
                 var scenarioReader =
                        ScenarioHelper.GetScenarioReader(save ,
                        "Scenario_Middle");
                 connector.Connect(scenarioReader , _ =>
                 save.SelectedIds.Add(_.First().SelectionId));
                 manager. ChangeView(new ScenarioView(scenarioReader ,
                 save , spriteBatch ));
    });
    connector.Game = game;
    game.Run();
    Assert.Equal("Ending 0-4-199", automator.CurrentPoint.PointId);
    }
}
```

図 5　ミドルサイズシナリオの評価の単体テストコード

5．2　結果

テストの実行時間は図 6 の様になった。理論値の計算式は実行時間 $= \frac{総画面数}{フレームレート}$ である。今回はフレームレートを 60fps とした。Smallの場合は誤差率が 57% と非常に高いが，画面数が少ないため遷移時間よりも初期化処理がかかったと考えられる。Middle,Large では初期化処理に比べて理論値の遷移時間が長いため誤差率が非常に低くなっている。この結果から自動遷移は成功しており，大量の遷移があるほど理論値に近づくため本システムは非常に有効であると考えられる。　今回の手法を用いるとシステムが機能の分担を上手く設計しているため，テストが通らなかった場合の原因究明方法も簡単となる。たとえば想定した停止位置と実際の停止位置が違っていた場合は，RouteSelector が悪いと考えられる。これはモデルの遷移における分岐先の決定はRouteSelector に任しているからである。またアプリケーションが途中で落ちてしまうような失敗かつロジックテストが通らなかった場合は，モデルや分岐制御に問題があるわけでは無く，それ以外の部分に問題があると考えられる。このようにシステム構成の際にテストを考慮した設計をしたため問題の切り分けをスムーズに行うことが可能となる。

5．3　まとめ

本研究では画面遷移を伴うシナリオ分岐型アプリケーションにテストの自動化を行う方法を提案した。目的としていた実際にアプリケーションを動作させてのテストを行うために，4 章で挙げた四つの問題点についてそれぞれ「入力デバイスからの正確な入力方法) System Connector を用いて内部を制御するように設計」「特定フレームワークへの依存) 特定フレームワークに依存しな

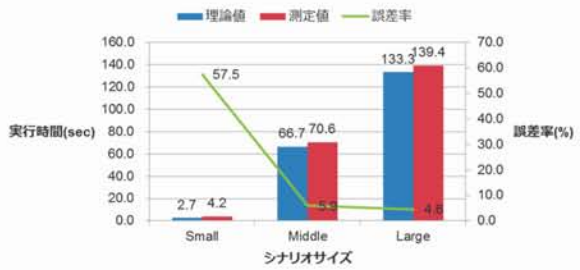

図 6　シナリオサイズ別のテスト実行時間

い完全に切り離した設計」「コードのメンテナンス性や型安全性)UI の操作ではなく内部の制御に着目したため，初期条件を与えるだけの簡単なコードでテストが記述可能となるように設計」「高度な専門知識や高価なツール)任意のユニットテストフレームワークのみで使用可能になるように設計」という方法を用いて解決を行った。実際に動くことを示すために実際にサンプルアプリケーションを作成し，本システムを利用してテストを行い実際に動作していることが確認できた。テストコードの記述面では「入力に関する条件や情報」「アプリケーションのテスト用の初期化」を行うだけで簡潔に行うことが出来た。テストの実行速度においても大きな画面遷移モデルで構成されるアプリケーションにおいて非常に効果的な結果を出している。小さな規模のアプリケーションにおいてはアプリケーションの初期化時間でテスト時間の半分ほど占めることもあるが，自動化という点において非常に有効であると考えられる。

今後の課題としてテストの有効性を認識してもらうシステムの作成が挙げられる。今回提案したシステムはそれ単体で効果を上げるようなシステムでは無く，システムの上にアプリケーションを構築することで効果を得ることが出来るシステムである。このため本システムを様々な場所で利用してもらうためには，今回の評価で使用したゲーム用のモデルのように特化したモデルを提供する必要がある。

参考文献

[1] Kent Beck: ”Test Driven Development: By Example”, Chapter 32. Mastering TDD, Addison-Wesley, pp.205, 2002

[2] 近澤 良:「シナリオ分岐型ウェブテストの設計と実装」, 成蹊大学理工学部卒業論文, 2007