Concolic Testingを用いたWebアプリケーションに対するテストデータ生成に関する研究

石川 太一郎[1,a)]　高田 眞吾[1,b)]　丹野 治門[2,c)]　生沼 守英[2,d)]

概要：テストデータ生成はテストの自動化において重要な役割を果たす．Web アプリケーションでは従来のソフトウェアとは違い, 更新の頻度が高く, さまざまなプラットフォームから実行されるため, Web アプリケーションに対するテストデータ生成は従来のソフトウェアに対するテストデータ生成以上に重要である．テストデータ生成手法の一つとして, Concolic Testing と呼ばれる手法がある．Concolic Testing では全ての実行パスを探索するが, 実行パスの数はプログラムの分岐数に対して指数関数的に増える．このため, Concolic Testing はテストデータ数や計算時間が分岐数に対して指数関数的に増えてしまうという問題を抱えている．本論文では, ページ遷移の網羅を目的とした Web アプリケーションの統合テストデータを, Web アプリケーションの設計モデルから生成する手法を提案する．提案機構は, シミュレーション用のソースコードを CATG に対して入力し, Concolic Testing によって実際の Web アプリケーションに対するテストデータを生成する．また, CATG により Concolic Testing を用いてテストデータを生成する際に画面遷移の網羅度に寄与しないテストデータを二つの枝刈り手法 (使用済み遷移に対する枝刈り, 重複パスに対する枝刈り) を用いて削減する．これらの枝刈り手法はシミュレーション用のソースコードに含む形で生成し, CATG 上でテストデータを生成する際に枝刈りを行う．評価を行った結果, 枝刈りによってページ遷移の網羅度を保ちつつテストデータ数を減らすことが出来, 最大で 0.4%に減らしている事を確認した．

キーワード：Concolic Testing, テストデータ生成, 統合テスト

1. 序論

近年のインターネットインフラの発達, クライアント側のブラウザの処理能力の向上からスタンドアロンで提供されていたアプリケーションが Web アプリケーションとして提供されるようになった．

しかし, Web アプリケーションに対するテストデータ生成は従来のソフトウェアに対するテストデータ生成より研究されていない．この理由として, Web アプリケーションのアーキテクチャではテストデータ生成を自動化しづらい事, 開発のイテレーションが短く更新頻度が高い事の二つの理由が挙げられる．

1 慶應義塾大学
Keio University, Yokohama, Kanagawa 223–8522, Japan
2 NTT ソフトウェアイノベーションセンタ
NTT Software Innovation Center, Minato–ku, Tokyo 108–8019, Japan
a) taichi1992@a2.keio.jp
b) michigan@doi.ics.keio.ac.jp
c) tanno.haruto@lab.ntt.co.jp
d) oinuma.m@lab.ntt.co.jp

まず, 自動化しづらい理由は, Web アプリケーションはサーバとクライアントで別のプログラムが協調して動作するアーキテクチャであるためである．このため, Web アプリケーションに対するテストではサーバとクライアント両方の動作をテストする必要がある．このサーバとクライアントの両方を統合的にテストするには, サーバとクライアントの動作を共に考慮する必要があり, 自動的に行うには複数のプログラムに対応しなくてはならない．

次に, 更新頻度が高い理由は, Web アプリケーションが従来のソフトウェアとは違うライフサイクルであるためである．Web アプリケーションの使い方は, 従来のソフトウェアとは違い, その場のニーズに合わせ, 必要な時に必要な分だけ使われる．このニーズに応えるため, 開発のスピードは可能な限り早めなくてはならず, 従来のソフトウェアテスト以上にテストを行なう事が出来ない．また, テストはソフトウェア開発の最終工程であるため開発の遅れの影響を受けやすく，ソフトウェアに対して十分なテストを行えないままで稼動することもある．このため, 普通のソフ

© 2014 Information Processing Society of Japan

トウェアよりも更新頻度が激しい Web アプリケーションにおいて, 更新毎にテストを十分に行う事は非常に厳しく, テストを効率化する事は従来のスタンドアロンで提供されているソフトウェア以上に重要である.

既存研究では, Web アプリケーションに対するテストデータ生成に対してクローラを用いたモデル検査ツール [8] や Concolic Testing を用いた単体テスト生成ツール [1] [10], Concolic Testing を用いたシナリオテスト生成手法 [9] が提案されている. しかし，クローラでは Web アプリケーションの状態数が多い場合, 状態爆発を起こしてしまい, Concolic Testing を用いた既存研究は単体テストの粒度でしかテストを行う事ができないという問題があった. また, シナリオテストでは Web アプリケーションに対する網羅的なテストを行えないという問題があった.

そこで, 本研究では Web アプリケーションに対してのページ遷移の網羅度の向上を目的とした統合テストデータ生成に着目した. 本研究では設計モデルを用いて Web アプリケーションの動作をシミュレーションするソースコードを生成し, Concolic Testing ツールを用いてテストデータを生成する.

本論文の貢献は, 次の二つである.

(1) ページ網羅度に注目した, Concolic Testing による Web アプリケーションに対する統合テストデータ生成手法の提案
(2) Concolic Testing 時に発生するパス爆発に対する枝刈り手法の提案

以降, 2 章では Web アプリケーションに対するテストデータ生成手法に対する既存研究とそれらに対する問題を述べる. 3 章では, 従来のテスト手法と Concolic Testing に対する説明を行い, 4 章で提案手法を紹介する. 5 章で評価について述べ, 最後に 6 章で本論文の結論を述べる.

2. 関連研究：Concolic Testing

この章では, Concolic Testing の前提となるテストデータ生成手法と, Concolic Testing について述べ, Web アプリケーションに対して Concolic Testing を適用した既存研究も述べる.

Concolic Testing の前提となるテストデータ生成手法は具体値を用いた手法と, 記号的実行を用いた手法に分けられる. この三つの手法について解説を行う.

2.1 具体値を用いたテストデータ生成手法

具体値を用いたテストデータ生成手法として，Random Testing [2] が挙げられる. この手法は実際のプログラムの入力値に対して，ランダムな値を割り当ててテストデータとする手法である. 入力値がランダムであるため，どのような入力に対しても一様にテスト出来るという特徴を持つ.

この手法の利点は，テストデータ生成に対する準備が少なくて済むという点である. Random Testing を適用する場合, プログラムの入力部に対して乱数値を入れるように変更するだけでテストデータを生成できる. このような利点があるため, 負荷テストを兼ねて Random Testing を行う場合もある.

実際にこのテスト手法を用いる場合, テスト回数の上限をを指定してテストデータ生成を行う. この理由は, Random Testing の場合, 網羅率を目標とするとテスト完了までの時間を予測出来ず，指標として扱いづらいためである.

2.2 記号的実行を用いたテストデータ生成手法

記号的実行ではプログラムを実際に実行するのではなく，プログラムを制御フローグラフに変換し，この上でのパスに対する入力値の制約を見つける.

図 1, 図 2 で記号的実行について説明する. まず, 図 1 で示したプログラムを，図 2 で示した制御フローグラフに変換し，制御フローグラフ上のパスを求める問題に置き換える. この制御フローグラフのパスを実際にたどり, パスに関わる制約を解くことで入力値の条件を求める. この条件から実際に値を生成する際には，制約ソルバを用いて値を生成する. 制約ソルバとは，変数の定義と制約を与えると制約を満たせるかどうか判定し，満たせる場合は入力値の例を出力するツールである. 記号的実行で用いる場合は, プログラムの変数の定義，代入が制約ソルバで扱う変数になり，分岐文の条件と結果が制約ソルバに渡す制約になる. 実際には, SMT ソルバと呼ばれる制約ソルバの一種を用いてテストデータを生成する. この SMT ソルバの例として，CVC3[5] や Yices[11] があげられる.

```
bool test_me(int x){
  if(x == 0xcafebabe){
    return false;
  }else{
    return true;
  }
}
```

図 1 プログラム例

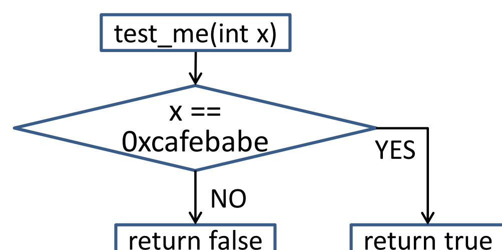

図 2 制御フローグラフの例

2.3 Concolic Testing

Concolic Testing は上記の二つの手法 (具体値を用いたテストデータ生成手法, 記号的実行を用いたテストデータ生成手法) を組み合わせた中間的なテストデータ生成手法である [7]. DART(Directed Automated Random Testing)[6], DSE(Dynamic Symbolic Execution) と呼ばれることもある．この手法では具体値で実行する段階と，制約を解く段階の二段階の処理を繰り返してパス網羅度の高いテストデータを生成する．

2.3.1 Concolic Testing の実行例

```
int main(int x, int y){
  int z = y * 2;

  if(x != z){
    return  1;
  }else{
    return -1;
  }
}
```

図 3 Concolic Testing の対象関数

最初は関数に引数に対してランダムにテストデータを生成してプログラムを実行し, 実行されたパスを取得する．今回は $x = 1, y = 1$ で実行され, 1 が返される．そして，実行されたパスに対して記号的実行を行い，通ったパスの分岐文から実行パスを通るための制約を取得する．この記号的実行で, 今回通ったパスの制約 $x! = y * 2$ が得られる．

次は, 得られたパス制約 $x! = y * 2$ を満たさないように入力値を決めて実行する．ここでは, テストデータとして $x = 2, y = 1$ が得られたとすると, 実行した結果 -1 が返される．この通ったパスの制約を記号的実行を用いて求める．ここで実行パスの制約として $x == y * 2$ が得られる．

最後に, 既に得られたパス制約 $x! = y * 2$ と $x == y * 2$ を満たさないテストデータを生成するが, この条件を満たさないテストデータを生成することは出来ない．このため, Concolic Testing はこの時点で終了する．

この結果, プログラムの実行パスを網羅する 2 つのテストデータとして，$x = 1, y = 1$ と $x = 2, y = 1$ が得られる．

2.4 Concolic Testing を用いた単体テスト

Web アプリケーションに対して，Concolic Testing[7] によってテストデータを生成する手法を Artzi ら [1] や Wassermann ら [10] が提案している．

Artzi らは PHP に対する Concolic Testing を用いた単体テストデータ生成手法を提案している [1]. Well-formed ではない HTML，PHP の実行時エラーをテストオラクルとして Web アプリケーションに対するテストデータを生成する．また，テスト対象の PHP のソースコードが生成する HTML に対して，リンクとして PHP ファイルが指定されている場合その PHP ファイルを読み込む事で部分的に統合テストを行える．しかし，Artiz らの手法では同じファイルを何度も読まないようにしているため，何回も同じページを通るパスに対してテストを行えない．

Wassermann らは PHP の SQL 文に対する脆弱性を Concolic Testing を用いて検出する手法を提案している [10]. Concolic Testing を用いる事で部分部分を解析し，SQL Injection が可能であるパスを検出する．この手法は PHP の SQL 文のみを対象としている. しかし, Wassermann らの手法は SQL 文のみを対象としているため, Web アプリケーション全体をテストする事が出来ない． Artzi らや Wassermann らのツールは，ソースコードレベルから Web アプリケーションのテストデータ生成を行える．しかし，これらのツールは単体テスト用，SQL インジェクション対策というように，Web アプリケーション全体をテストするために作られているわけではない.

以上より，これらの手法では統合テスト用のテストデータを生成できないという問題がある．

2.4.1 Concolic Testing を用いたシナリオテスト

丹野らは, Concolic Testing を用いてシナリオテスト用テストデータ生成手法を提案している [9]. この手法は, Web アプリケーションに対して, 設計時のモデルの情報を用いてシミュレーション用のソースコードを生成することにより, あらかじめ指定したテストパスを通るテストデータ生成を行う．

しかし, 丹野らの手法はシナリオテストを対象としているため, あらかじめ指定したテスト対象のページ遷移を通るテストデータしか出力できない．

このため, ページ遷移の網羅度の向上に注目したテストデータを出力できないという問題がある．

2.4.2 パス爆発問題への対処

記号的実行を用いたテストデータ生成や, Concolic Testing によるテストデータ生成では, パス爆発という現象が問題となる．

パス爆発とは, プログラムの分岐の数に対して指数関数的に実行パスが増える現象の事を指す．Concolic Testing ではテストデータ生成を入力値が生成できなくなるまで行い, パスの網羅を目的としているため, テストデータ生成にかかる時間も指数関数的に増えてしまう．

また, 統合テストの場合, 単体テストに比べてプログラムの分岐数は格段に多くなってしまいプログラムのパス全てをテストする事が現実的な時間で出来なくなってしまう．

このパス爆発という問題によって, 特に統合テストにおいて Concolic Testing を適用することは困難になってしまう．

既存の Concolic Testing に対するパス爆発を回避するための手法を Burnim ら [3] は提案している．この手法は, プ

© 2014 Information Processing Society of Japan

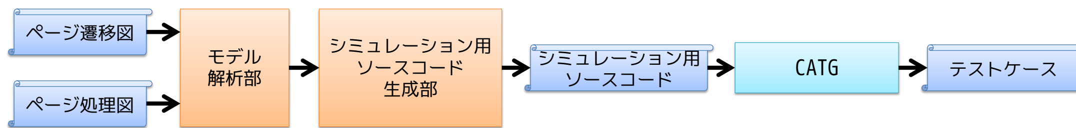

図 4　提案概要

ログラムに対するパス数ではなく，プログラムの各分岐をどれだけ少ないテストケースで網羅できるかを指標として Concolic Testing を行う．

3.　提案

提案手法は設計モデルを用いた Web アプリケーションのページ遷移の網羅度の向上に注目した統合テスト用テストデータ生成手法である．

提案手法に用いる提案機構の概要を図 4 に示す．提案機構では，設計モデルとしてページ遷移図とページ処理図を与えると，テストデータ生成用のシミュレーション用ソースコードを出力する．シミュレーション用ソースコードを CATG[4] で実行することでテストデータを生成する．なお，CATG は Java を対象としたオープンソースの Concolic Testing ツールである．

次に，設計モデル，シミュレーション用ソースコード生成，枝刈り機構について述べる．

3.1　設計モデル

提案機構ではページ遷移図，ページ処理図の二種類の設計モデルを用いた．

3.1.1　ページ遷移図

ページ処理図には状態遷移図に基づいている．ページ遷移図は各ページから各ページへの遷移関係を有向グラフで示したものである．

図 5 でページ遷移図の例を示す．この例では，Web アプリケーションに対して，ページ 1 →ページ 2，ページ 1 →ページ 3，ページ 2 →ページ 3 の三つのページ遷移が存在する事を示している．

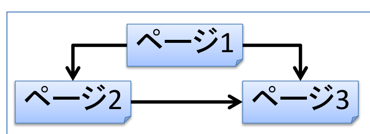

図 5　ページ遷移図の例

3.1.2　ページ処理図

ページ処理図は，各ページで実行される処理のフローチャートとして表す．フローチャートを用いた理由は，テスタやプログラマになじみやすい形で各ページの処理を記述することが出来るためである．

図 6 でページ処理図の例を示す．この場合，ページ 1 ではある条件を満たすとページ 2 へ遷移し，満たしていない場合はページ 3 へ遷移する事を示している．図を重ねている理由は，ページの数だけページ処理図があるためである．

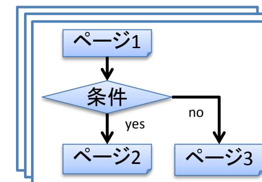

図 6　ページ処理図の例

3.2　シミュレーション用ソースコードコード生成

シミュレーション用ソースコード生成部では，ページ遷移図とページ処理図からシミュレーションを行うためのソースコードを生成する．以下で生成する二種類の関数について述べる．

3.2.1　網羅度算出関数

この関数は Concolic Testing 実行中のページ網羅度を算出する．ページ遷移の網羅度の計算方法は，今まで実行したテストデータがページ遷移図に書かれた遷移をどれくらい通過しているかである．

このページ遷移の網羅度の算出方法は，ページ遷移図から各ページからページへの遷移をシミュレーション用ソースコードに保持しておき，テストデータを生成するたびに通過した遷移にマークをつける事で，使った遷移かどうかを区別しページ網羅度を算出する．この網羅度は，テストデータ生成毎に更新される．

例を図 7 に示す．図 7 の点線の矢印が未マークであるページ遷移であり，実線の矢印がマーク済みのページ遷移である．初期状態では遷移を使用していないので 0%であるが，実行を進める毎に使用した遷移の数は増える．この例では，二回目で 8 本のページ遷移中 6 本をマークしているため，ページ遷移の網羅度は 75%になる．

3.2.2　ページ遷移関数

ページ処理図の情報を用いてソースコードを生成する．ページ処理図では，各ページの遷移がフローチャートで記述されているためこのフローチャートを機械的にソースコードへと変換する．

3.3　枝刈り機構

統合テストでは，全ての組み合わせを評価しようとする

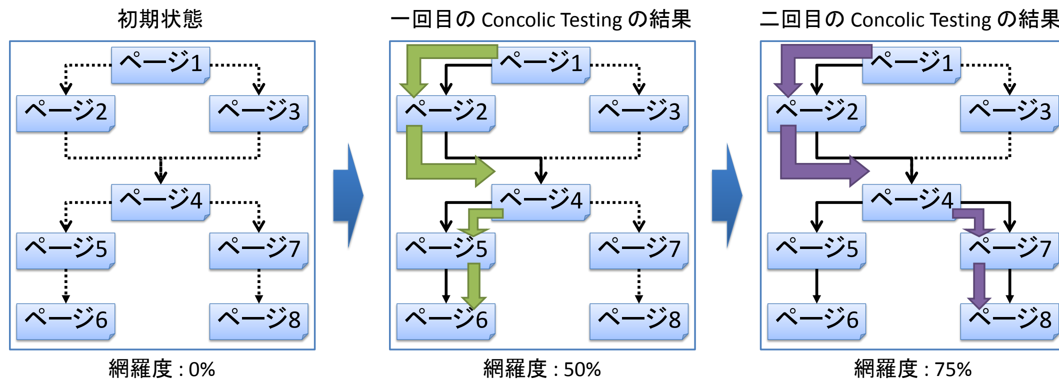

図 7　ページ網羅度算出の例

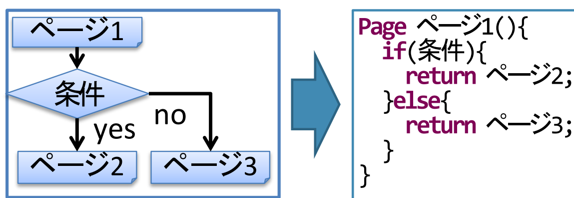

図 8　ページ遷移関数の生成

と組み合わせの数が指数爆発してしまう．このため，今回は複数の枝刈りを組み合わせることで，ページ遷移の網羅度を考慮しつつテストケースの数を削減する．

本手法では，以下の二種類の枝刈りを用いた．

3.3.1　使用済み遷移の枝刈り

使用済み遷移の枝刈りは，深さ優先のパス探索を前提とした枝刈り手法である．

深さ優先の探索であれば，ある遷移の使用を一度やめた場合，その遷移を使用した実行パスによって得られるページ遷移は全て網羅していると考えられる．このため，ある遷移の使用を一度やめた場合，二度と使えないようにすることでページ網羅度から見て無駄なテストデータ生成を防ぐことが出来る．

図 9 を用いて説明する．前回のパスは (ページ 4 →ページ 5 →ページ 6) の遷移を使用しているが，今回のパスは (ページ 4 →ページ 5 →ページ 6) ではなく，(ページ 4 →ページ 7 →ページ 8) を通っている．このため，ページ遷移の枝刈りでは，(ページ 4 →ページ 5 →ページ 6) を使用不可能にする．

3.3.2　重複したパスに対する枝刈り

重複したパスに対する枝刈りは，カバレッジを犠牲にしてもテストデータ数を減らす事を目的とした，アグレッシブな枝刈り手法である．

この枝刈りを行う前提として，Concolic Testing では分岐を全て考慮してしまうため複数の条件で同じページにいく場合，それらを別個の遷移として扱う．このため，ページ遷移図では一つの遷移として書いてある部分が，実際には多重な遷移である事がある．

図 10 を用いて説明する．前回のパスと今回のパスでは，(ページ 1 →ページ 2 →ページ 4 →ページ 5 →ページ 6) と，完全に重複したパスが生成されている．このため，一番最後の遷移である，(ページ 5 →ページ 6) を使用不可能にすることで (ページ 5 →ページ 6) への遷移が多く，何度も使ってしまう場合にテストデータ数を減らすことが出来る．

4.　評価

ケーススタディの評価は以下の 2 点に関して行った．

(1) 得られたテストデータの数
(2) 得られたページ遷移網羅度

枝刈りを用いずに得られたテストデータに対して，枝刈りを用いて得られたテストデータを比較することで提案機構の評価を行った．

4.1　評価環境

評価環境は次の通りである．

- オペレーティングシステム
  – Windows 7 64bit
- CPU
  – Intel Core i7-4820K 3.70 GHz
- メモリ
  – 32.0 GB

4.2　評価方法

枝刈りの有効性を評価するため，枝刈りを行わない手法との比較を行った．実験手順を以下に示す．

(1) 枝刈りを行わない，ナイーブな手法でテストデータを生成する．
(2) 二種類の枝刈りを両方とも適用してテストデータを生成する．
(3) 以下の項目について，枝刈り無しの場合と比較を行う．
  - 生成されたテストデータの数
  - 生成されたテストデータのページ遷移網羅度
  - テストデータ生成にかかった時間

© 2014 Information Processing Society of Japan

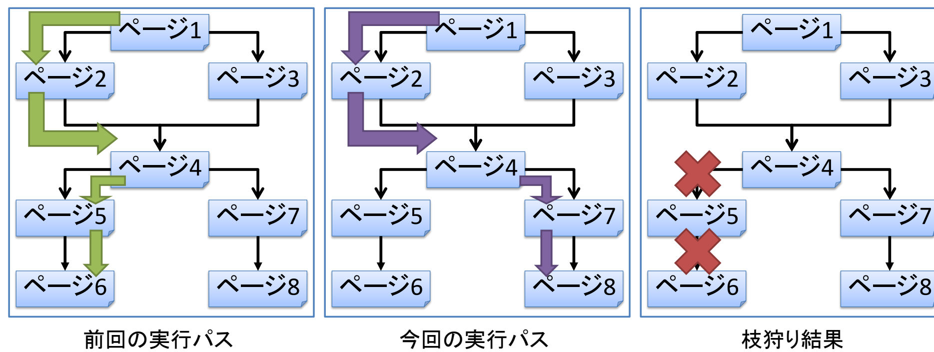

図 9　使用済み遷移の枝刈り

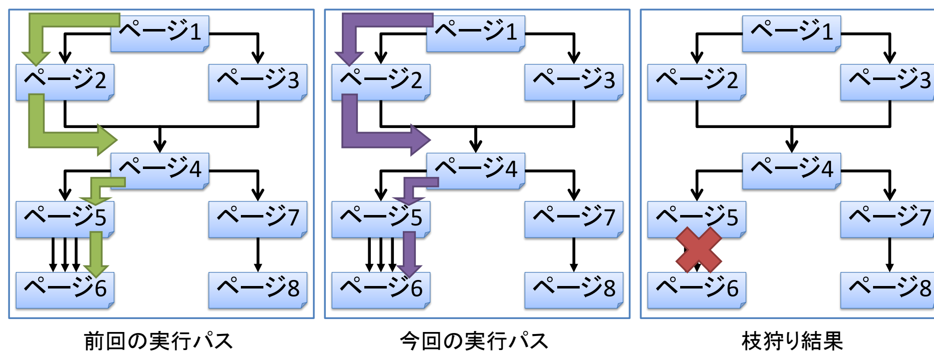

図 10　重複遷移の枝刈り

ナイーブな手法に対しては，テストデータ生成にかかる時間が予測しづらいため生成テストデータ数の上限を100,000 に設定してテストデータ生成を行った．

テスト対象アプリケーションとして次の 3 つのアプリケーションに対して評価を行った．

- ログイン/新規登録
- 書籍購入機構
- ゲーム

それぞれの特徴を表 1 で示す．

表 1　評価対象アプリケーション

対象アプリケーション	ページ遷移の数	セッション変数	DB の行数
ログイン/書籍購入	16	1	5
書籍購入システム	11	1	20
ゲーム	10	2	2

4.3　評価結果

表 2 で生成テストデータ数を，表 3 でページ遷移の網羅度を，表 4 でテストデータ生成にかかった時間を示した．表 2 と表 3 の括弧で囲んである数値はそれぞれの枝刈り手法で枝刈りなしの場合に比べた際の変化量を表している．

まず，表 2 で示した生成テストデータ数からは，各種枝刈りによってテストデータ数は大分削減出来ているという事が見て取れる．特に重複パスに対する枝刈りは全てのアプリケーションに対してテストデータを削減している．また，ゲームに対する枝刈り無しのテストデータ数が 100,000 になっているのは生成テストデータ数の上限に達してしまったためであり，本来はもっと多いと予想される．

次に，表 3 で示したページ遷移の網羅度からは，使用済み遷移に対する枝刈りは網羅度のロスせず枝刈りを行えていることが見て取れる．これとは対照的に，重複パスに対する枝刈りでは網羅度をロスしてしまっていることが見て取れる．

最後に，表 4 からは，100,000 個のテストデータを生成するためには 84 時間かかることが示されており，全てのテストデータを生成する事が非現実的であり枝刈りを行う正当性を裏付ける証拠の一つである．

表 2　評価用アプリケーションに対する生成テストデータ数

アプリケーション	枝刈り無し	両方の枝刈り
ログイン/書籍購入	85	18(22%)
書籍購入システム	14	13(93%)
ゲーム	100000	43(0.4%)

表 3　評価用アプリケーションに対する遷移網羅度

アプリケーション	枝刈り無し	両方の枝刈り
ログイン/書籍購入	100%	100%(±0%)
書籍購入システム	100%	73%(-27%)
ゲーム	80%	50%(-30%)

© 2014 Information Processing Society of Japan

表 4 評価アプリケーションに対してテストデータ生成にかかった時間

アプリケーション	枝刈り無し	両方の枝刈り
ログイン/書籍購入	3 分 6 秒	37 秒
書籍購入システム	1 分 25 秒	1 分 19 秒
ゲーム	84 時間 38 分 52 秒	1 分 41 秒

4.4 考察

提案機構で用いた枝刈りの有用性について, 本節で考察を行う.

4.4.1 二種類の枝刈りを両方とも適用した場合の考察

二種類の枝刈りを適用した場合に関しては, 重複パスの枝刈りが支配的であった. これは使用済み遷移の枝刈りが遷移を使用してから枝刈りするため, 重複パスの枝刈りが行える場合は必ず重複パスの枝刈りが先に適用されるためである.

これより, 重複パスの枝刈りを行って問題の無いアプリケーションであった場合両方を適用したほうが効率の良いテストデータを生成できると結論付けられる. 実際に, ログイン/新規登録のアプリケーションでのテストデータ数, ページ網羅度の結果は両方の枝刈りを適用した場合の方が片方の枝刈り結果よりも良かった事から裏付けられる. また, 他のアプリケーションに関しては, 重複パスの枝刈りによって網羅度を損ねてしまったため両方の枝刈りを適用した場合でも同様に網羅度を損ねていると考えられる.

4.4.2 枝刈りの有用性に関する考察

枝刈りを行うことによって, テストケース数を減らすことに成功している. 特に, 使用済み部分の枝刈りは網羅度を維持しつつテストケースを減らせており Web アプリケーションの仕様を網羅するテストを行う際には有用だと考えられる.

重複部分に対する枝刈りは, 網羅度が悪くなってしまっている. しかし, 使用済み部分の枝刈りに対してもテストケース数を削減しており, より大規模な Web アプリケーションに対しては使用済み部分の枝刈りをもってしても枝刈り仕切れない場合が発生するため, 大規模なアプリケーションに対してこの枝刈りは必要になると考えられる.

5. 結論

Web アプリケーションに対して Concolic Testing を適用するための機構を提案し, パス爆発を抑制するための二つの枝刈り機構を適用した. また, 枝刈りに対する評価を行い, これら二種類の枝刈りに対する有効性を確認した.

今後の課題としては, 実アプリケーションを対象とした評価を行い, 枝刈り手法の有効性を確かめる事が挙げられる. 現時点では, 自作のアプリケーションのみで評価を行っているため, 実際のアプリケーションで枝刈りがどのように働くかの評価が必要であると考えられる.

また, テストデータ生成の効率化も今後の課題である. 現時点では オープンソースの Concolic Testing ツールである CATG には手を入れずシミュレーション用のソースコードだけで枝刈りを行っているためである. CATG 上で直接的に枝刈りを行うことで, テストデータ生成を高速化できると考えられる.

参考文献

[1] S. Artzi, A. Keizun, J. Julian, F. Tip, D. Dig, A. Patadkar, M. D. Ernst: “Finding bugs in dynamic web applications”, *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pp.261-272 (July 2008).

[2] B. Beizer(著): 小野間彰, 山浦恒央 (訳): “ソフトウェアテスト技法”, 日経 BP 出版センター (Feb. 1994).

[3] J. Burnim, K. Sen: “Heuristics for Scalable Dynamic Test Generation”, *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp.443-446 (Sep. 2008).

[4] “ksen007/janala2” 2013-12-15, `https://github.com/ksen007/janala2` (参照 2013-12-25).

[5] “CVC3 page” 2011-09-01, `http://www.cs.nyu.edu/acsys/cvc3/` (参照 2013-12-24).

[6] P. Godefroid, N. Klarlund, and K. Sen: “{DART}: Directed Automated Random Testing”, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, pp. 213-223 (June. 2005).

[7] K. Sen, D. Marinov, G. Agha: “Cute: A Concolic Unit Testing Engine for C”, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT international Symposium on Foundations of Software Rngineering (ESEC/FSE-13)*, pp. 263-272 (Sep. 2005).

[8] H. Tanida, M. R. Prasad, S. P. Rajan, M. Fujita: “Automated System Testing of Dynamic Web Application”, *Software and Data Technologies (Revised Selected Papers of ICSOFT 2011)*, pp.181-196 (2013).

[9] 丹野治門, 星野隆, K. Sen, 高橋健司: “Concolic Testing を用いた結合テスト向けテストデータ生成手法の提案”, 第 *182* 回ソフトウェア工学研究会 (Oct. 2012).

[10] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, Z. Su: “Dynamic Test Input Generation for Web Applications”, *Proceedings of the 2009 International symposium on Software testing and analiysis*, pp.249-260 (July 2008).

[11] “The Yices SMT Solver”, 2013-12-04, `http://yices.csl.sri.com/` (参照 2013-12-24).

© 2014 Information Processing Society of Japan