Vol.2013-SE-182 No.6
2013/10/24

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

丹野 治門[1,a)]　星野 隆[1]　Koushik Sen[2]　高橋 健司[3]

概要：本研究は，関係データベース（DB）を用いる業務システムの結合テストを対象として，DB アクセスを伴う各テストケースに対し，適切なテストデータ（画面入力値及び DB 初期状態）を自動生成する問題を扱う．既存技術では，複数回の参照系アクセス，更新系アクセスを扱っていないため，テストデータ生成の適用範囲が狭く，設計モデルとして単一画面遷移しか扱っていないため，複数画面遷移を伴うテストを扱うことができないといった問題があった．本研究では複数回の参照系アクセス，更新系アクセスを扱えて，かつ複数画面遷移も記述することが可能である設計モデルを規定し，その設計モデルからシミュレーション用ソースコードを生成し，生成したソースコードに対して Concolic 実行を行うことで，テストデータの生成を行う手法を提案する．本論文では，簡単な例題を用いたケーススタディにより，提案手法の実現性についても考察した．

キーワード：ソフトウェアテスト, テストデータ生成，モデルベーステスト，Concolic Testing

Test Data Generation for Integration Testing by Using Concolic Testing

Abstract: This research focus on how to automatically generate suitable test data composed of input value and the initial state of relational database for each test case of the integration testing of enterprise systems which use database. The existing methods cannot handle test data generation for systems that have complicated logic such as reading or updating database more than once. In addition, the existing methods use the design model which can express only a screen transition between 2 screens and cannot express screen transitions among more than 3 screens. To solve the problems, we proposed a design model which can express a business logic where it reads or updates database more than once and the screen transitions among more than 3 screens, and we also propose test data generation method where it converts the design model to source codes and applies concolic testing to the source codes. In this paper, we also show the proof of the concept with a simple case study.

Keywords: Software Testing, Test Data Generation, Model Based Testing, Concolic Testing

1. はじめに

関係 DB を用いた業務システムの結合テストを行う際には，確認したいソフトウェアの特定のシナリオを実行するために，テストケースごとに適切な**テストパス**と**テストデータ**を用意する必要がある．テストパスとは，ソフトウェア設計書の画面遷移図上の特定の画面遷移列と定義する．図 1 では画面が 6 つある画面遷移図であり，実線で表された矢印は，正常フロー，点線で表された遷移は，エラー処理のための遷移である準正常フローを表す．テストの際には，正常フローを遷移していく画面遷移列（例えば画面 1,3,4）などと，何らかの意味のあるテストパスをテスト対象として抽出してテストを行う．テストデータとは特定のテストパスに従って画面を遷移させていくための，各画面の入力値（以降，**画面入力値**と呼ぶ）と，あらかじめ DB へ投入しておく DB の初期状態（以降，*DB* **初期状**

[1] NTT ソフトウェアイノベーションセンタ，東京都港区港南 2-13-34 NSS-□ビル 6F
[2] University of California, Berkeley, California 94720, USA
[3] NTT Innovation Institute Inc., San Mateo, California 94401, USA
[a)] tanno.haruto@lab.ntt.co.jp

ⓒ 2013 Information Processing Society of Japan

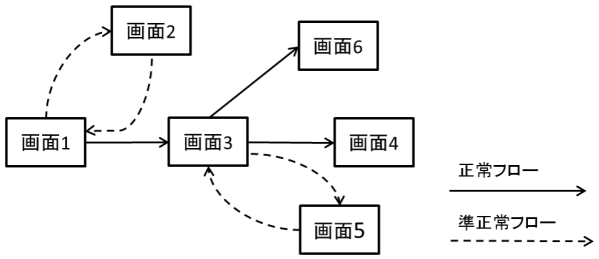

図 1 画面遷移図の例

態と呼ぶ）と定義する．画面遷移図からテストすべき意味あるテストパスを自動抽出する手法については過去に有用な研究がなされているが [11]，テストデータ作成に関しては，テスト担当者の負担を減らすため，自動化が望まれているが，未だ自動化が不十分な領域である．本研究では，テストすべき特定のテストパスに対して適切なテストデータ (DB 初期状態と各画面ごとの画面入力値) を生成する問題を扱う．

結合テスト向けのテストデータを，自動で生成する手法としては，ソフトウェア設計情報などから DB 初期状態と画面入力値の両方を自動生成する手法 [13,15–18] などが存在するが，これらの既存手法では，本研究がスコープとする「業務システムの結合テスト」に適用するにあたり，以下のような問題点が存在する．

- 単一画面遷移のみを伴うテスケースしか扱っていないため，図 1 で示したような複数回の画面遷移を伴うテストケース向けのテストデータ生成を行うことができない．
- 現実的な業務システムでは，参照系アクセス，更新系アクセスを伴うテストケースが多く存在するが，既存手法では DB への参照系アクセスを伴うテストケースのみしか扱うことができず，テストパスの途中に更新アクセスを伴う，テストケースに対しては，適切なテストデータを生成することができず，テストデータ生成の適用範囲が狭い．

既存手法で，テストデータが生成できないテストケースの例を，図 2 に示す．このテストケースは書籍を購入するサイトのシステムにおいて，ログインをしてから，書籍を購入するというシナリオが正しく動作するか確認するためのテストケースである．このテストケースでは，図 2「(1) ログイン画面」から，図 2「(5) 購入完了画面」までの 5 つの画面の遷移で構成されており，例えば，画面 (1),(2) では顧客や書籍の DB テーブルへの参照アクセス，画面 (4) では購入履歴に関する DB テーブルへの更新アクセスを伴う．そのため，既存手法では，このようなテストケースを扱うことができず，適切なテストデータを自動で生成することができない．

本研究では，このような既存手法の問題点を解決し，図

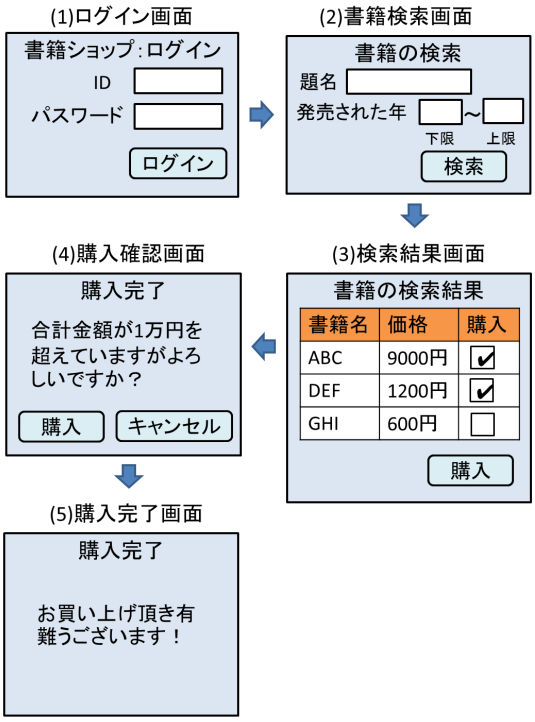

図 2 既存手法でテストデータを生成できないテストパスの例

2 で示したような，より現実的なテストケースに対して，適切なテストデータを現実的な時間内で自動生成することを目的とする．

本論文の貢献は以下の 2 点である．

(1) 参照系アクセス，更新系アクセスを伴い，複数の画面間を遷移するテストケースに対しテストデータを自動で生成する手法を提案した．本手法の特徴は，ソフトウェアの設計モデルから，エラー処理，通信処理等を含まず，処理ロジックの本質的な流れのみから成るコードを，シミュレーション用のソースコードを生成し，生成したソースコードに対して Concolic Testing [14] を実行することでテストデータを生成している点である．提案手法では，余計なテストパスへの探索を打ち切ることにより，効率良く必要なテストパスに対してテストデータを生成できるような工夫も行っている．

(2) ケーススタディとして，Online Book Store という簡単な例題を題材に，提案手法の実現性に関する考察を行った．また，上記で述べた，必要なテストパス以外への探索を打ち切る工夫も，テストデータを効率良く生成するために有効であるとわかった．

以降，2 章では，テストデータ生成に関する既存技術とその問題について述べる．3 章で，本研究が提案する手法について紹介し，4 章でケーススタディについて述べる．そして，最後に 5 章で本論文の結論を述べる．

ⓒ 2013 Information Processing Society of Japan

2. 関連研究

テストデータの自動生成に関する研究 [9] は多く行われている．本章では，本研究がスコープとする業務システムの結合テスト向けのテストデータ生成の既存手法と，本研究の提案手法で利用する Concolic Testing を用いた単体テスト向けのテストデータ生成技術について述べる．

2.1 業務システム結合テスト向けテストデータ生成の既存手法

業務システムの結合テスト向けのテストデータ生成の既存手法は大きく分けて 2 つ存在する．

第一の方法はテストデータのうち，単純に DB 初期状態のみを DB スキーマから自動で生成する方法である．この方法は，乱数を用いて DB レコードの自動生成を行うものであり，既存ツールとしては DBMonster [1]，疑似個人情報ジェネレータ [4]，Visual Studio Database Edition [2] などがある．これらのツールを用いることで，DB のテーブル間の参照関係や主キーなどを満たした DB 初期状態を用意することが可能であるが，テストケースごとにレコードの追加や削除など，DB 初期状態の微調整を行ったり，画面入力値については各テストケースごとに手動で用意したりといったことが必要になる．

第二の方法は，よりテストケースごとに合った DB 初期状態を作成する方法で，初期の研究としては，DB 初期状態が満たすべき事前条件をユーザが記述し，テストを実施する前に使用する DB が事前条件を満たしているかを確認する手法 [6] が提案されている．最近では，テストケースごとに適切な画面入力値と DB 初期状態の両方を同時に自動生成する手法も提案されている．筆者らの手法 [15–17] や，藤原らの手法 [13,18] では，ソフトウェアの設計情報である DB 定義，処理フロー，画面入力値定義などの情報や，テストケースの事前条件に基づき，テストデータが満たすべき条件を制約として全て抽出し，これらの制約を SMT ソルバ [7] や制約プログラミング [12] を用いて解くことでテストケースごとに適切なテストデータの具体値を生成している．

第二の方法は第一の方法に比べると，画面入力値と DB 初期状態の両方を同時に生成することができるため，より有望な手法ではある．しかしながら，これらの手法では，テストパスが単一画面遷移のみ，かつ参照系アクセスのみを伴うテストケースを扱っており，本研究がスコープとするような，テストパスが複数画面遷移で，かつ複数回の参照系アクセス，更新系アクセスを伴うテストケースについては考慮されていないため，テストパス中で，DB の状態がテスト中に何度も更新されるテストケースに対しては，適切なテストデータを生成することができない．更新系アクセスを伴う場合には，DB の状態が刻々と変わるため，DB 初期状態の生成にあたって，DB のライフサイクルを考慮する必要があり，単純に DB 初期状態が満たすべき制約を集めて一度に解く方法は使えず，参照系アクセスのみを扱う場合と違った工夫が必要となる．

2.2 Concolic Testing を用いた単体テスト向けのテストデータ生成技術

Concolic Testing [14] とは，主に単体テストを対象とし，効率よくパスカバレッジを向上することを目指したテストデータの自動生成技術である．Concolic Testing はソースコード上で，未到達のパスを探索しながら，繰り返しプログラムの実行を行い，具体値によるプログラムの実行と同時に記号実行を行うことを特徴とする．

Concolic Testing を適用したときに，テストケース，テストデータ生成がどのように行われるのかを，以下のソースコードを具体例として簡単に説明する．この例では，パスカバレッジを 100% にするためには，それぞれ `Goal A`，`Goal B` に到達するための 2 つのテストケースが必要となる．

```
void testme(int x, int y){
    int z = y * 2;
    if (x != z){
        print("reach goal A"); //Goal A
    }
    else{
        print("reach goal B"); //Goal B
    }
}
```

このソースコードに対して Concolic Testing の実行を行うと，以下のように 2 回のプログラム実行が行われる．

(1) 1 回目の実行では，最初に入力値 `x,y` に対して任意の初期値 (ここでは，`x=1,y=1` とする) を生成して実行を行い，このとき分岐条件「`x != z`」が真となるため，`Goal A` へ到達する．これにより，テストデータ (入力値) が `x=1,y=1` であり `Goal A` に到達するテストケースが得られる．プログラム実行の際には，具体的な実行と同時に入力値を `x=x0,y=y0` とおき記号実行も行っており，1 回目の実行では分岐条件「`x0 != y0*2`」が真となっていたことが記録される．

(2) 2 回目の実行では未到達のパスを探索するため，前回記録した分岐条件「`x0 != y0*2` 」が偽となるよう，制約式 「 `!(x0 != y0*2)`」を解き，`x0=2,y0=1` の解を得て，これらを 2 回目に実行の入力値として用いる．2 回目の実行では，`Goal B` へ到達し， テストデータ (入力値) が `x=2,y=1` であり `Goal B` へ到達するテストケースが得られる．

このように，Concolic Testing を用いると，ソースコード

上の未到達パスへ到達するようなテストケース (テストパス) とテストデータを自動で次々に得ることが可能である．Concolic Testing を用いると，プログラムの実行に従って変数の状態が更新されていくような場合でも問題なく扱うことができる．

Concolic Testing を利用し，単体テストにおける入力値や，プログラムがアクセスする DB の初期状態を自動生成する試みとしては，未到達のパスへ到達するよう入力値と DB 初期状態の両方を調整しながらプログラムを繰り返し実行する Emmi らの手法 [10] や，既に存在する DB 初期状態を考慮しプログラムの入力値のみを次々に生成していく Pen らの手法 [5] が存在する．これらの手法は単体テスト向けではあるが，本研究の提案手法で利用している技術に近いものである．しかしながら，これらの手法では，あくまでパスカバレッジの向上を目的としているため，基本的には無作為に選んだ未到達のパスを通るようなテストデータを次々に生成を行う．そのため，特定のパスに対して効率よくテストデータの生成を行うことができない．本研究では，図 1 における「正常フローを遷移していく画面遷移列」のような，テストとして意味のあるテストパスを通るテストデータの生成を効率よく行うための工夫（3.2 節で詳しく述べる）を行っている．

3. 提案するテストデータ生成手法

前節で述べたような既存技術の問題点を解決し，業務システムの結合テストにおいて，より多くのテストケースに対してテストデータを自動生成できるようにするため，本研究では，モデルベーステスト [8] に基づき，評価対象システムの設計情報をモデル化した設計モデルからテストケースと，そのテストケースの適切なテストデータ (DB 初期状態と各画面の入力値) を自動生成する手法を提案する．手法の全体像を図 3 に示す．

提案する手法の特徴を以下に示す．

- 提案手法が入力とする設計モデルは，筆者らが過去に提案した設計モデル [17] を拡張しており，**処理フロー**，**入力定義**，*DB* **スキーマ**に加えて，**画面遷移図**，**大域内部変数**を記述することができる．処理フローでは，DB 参照，DB 更新を複数回行う複雑なビジネスロジックの振る舞いを記述できるようにしたため，様々な業務システムの振る舞いを記述することが可能である．画面遷移図は，業務システムの画面とその遷移 (正常フロー，準正常フロー) の情報である．大域内部変数とは，大域的に各画面遷移の処理において参照される変数であり，例えばログインしているユーザの ID に関する情報などである．設計モデル中に処理や変数の宣言をソースコードの断片で記述できるようにし，複雑な型や，処理ロジックなども記述可能にした．
- DB 初期状態がテストパス中で更新される場合でも問題なくテストデータを生成できるようにするため，本手法では，設計モデルをシミュレーション用ソースコード（エラー処理，通信処理等を含まず，処理ロジックの本質的な流れのみから成る）へ変換し，このソースコードに対して，Concolic Testing を実行することでテストケースとテストデータの自動生成を行う．本手法では，画面入力値，DB 初期状態は，全て変数として扱い，処理フローのロジックについては，DB への参照や更新などの操作も含め，全て手続き的なコードへと変換する．そして，このコードに対して Concolic Testing の実行を行い，各変数の値の具体値，すなわちテストデータの具体値を得る．
- シミュレーション用のソースコードに対して，Concolic Testing を単純に適用すると，画面遷移図上の様々なテストパスを探索し，大量のテストケースが生成されてしまう．本研究では，テストすべき特定のテストパスのみに対して，テストデータを得たいので，それが効率よく行えるよう，指定した特定のテストパス以外のパス探索を打ち切る「探索打ち切り機構」を導入した．

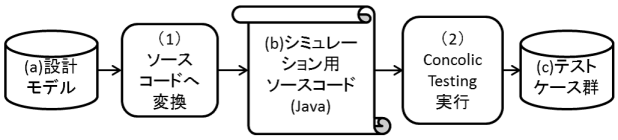

図 3 提案手法の概要

これらの特徴により，業務システムの結合テストにおいて，参照系アクセス，更新系アクセスを伴い，複数の画面間を遷移するテストケースとそのテストデータを自動で効率よく生成することが可能である．

以降，本章の以降の節では，本手法で扱う設計モデルからどのようにソースコードへ変換するかの要点を述べた後に，特定のテストパスに対して効率よくテストデータを生成するための探索打ち切り機構について述べる．

3.1 設計モデルからソースコードへの変換

本説では，設計モデルにおける各設計情報がどのようにソースコードへ変換されるかを掻い摘んで述べる．

画面遷移図は switch 文を用いた状態機械のソースコードへと変換される．例えば，図 1 のような画面遷移図から生成されるソースコードのイメージを以下に示す．

```
state = SCREEN_01;
switch (state) {
    case SCREEN_01:
        state = Screen01_Flow();
        break;
    case SCREEN_02:
        state = Screen02_Flow();
        break;
```

```
    ... 中略...
    case SCREEN_06:
        state = Screen06_Flow();
        break;
}
```

SCREEN_01 などは画面名の列挙値である．Screen01_Flow() メソッドはその画面において何らかのアクションが行われた時の処理ロジックが記述されている処理フローに相当する．メソッドの戻り値には，次に遷移する画面名を設定する．

各処理フローに記述された DB への参照，更新操作を含む処理ロジックも全て Concolic Testing が実行可能な手続的なソースコードへと変換する．処理フローは SQL 文が付与されたフローチャート形式で記述される．例えば，図 2 の (2) 書籍検索画面の処理フローから生成されるソースコードのイメージを以下に示す．

```
//(2) 書籍検索画面における処理ロジック
int S02_BookSerachFlow(){
    //入力定義における各画面入力値の定義において，
    //input.YearMin の範囲が 1900 以上，2030 以下
    if(!(input.YearMin >= 1900))return INVALID;
    if(!(input.YearMin <= 2030))return INVALID;
    …中略…

    //SELECT * FROM Books
    //  WHERE input.Title == Books.Title
    //        AND input.YearMin <= Books.Year
    //        AND input.YearMax >= Books.Year
    resultSet = {};
    for(int i=0;i<Books.recordCount();i++){
        if(input.Title == Books[i].Title &&
          input.YearMin <= Books[i].Year &&
          input.YearMax >= Books[i].Year){
            resultSet.add(Books[i]);
        }
    }
    if(resultSet.recordCount() > 0){
        //(3) 検索結果画面へ遷移する
        return S03_BOOK_SEARCH_RESULT;
    }
    else{
        //(6) 検索結果 0 件ヒットの画面へ遷移する
        return S06_BOOK_SEARCH_NO_RESULT;
    }
}
```

input は画面入力値を表す変数であり，Books は DB テーブルを表す構造体で，この構造体はメンバとして DB レコードの変数を持ち，各 DB レコードは各フィールドの変数を保持している．参照系アクセスの操作である SELECT 文は，上述したようなソースコードへと展開され，特に WHERE 句に相当する部分は，Books の各レコードのフィールドに相当する変数に対して条件を満たしているかどうかを確認する条件分岐へと展開される．上述した参照操作からのソースコード生成と同様な考え方で，更新系アクセス (UPDATE 文，INSER 文，DELETE 文)，例えば，図 2 の (4) 購入確認画面における購入履歴に関する DB テーブルへの更新も扱うことが可能である．

このように，各画面の画面入力値や，DB 初期状態は全て変数として扱い，DB アクセスの操作も全てソースコード上のロジックで表すことにより，設計モデルをシミュレーションでき，これに Concolic Testing を適用することで，テストデータを生成することが可能となる．

3.2 探索打ち切り機構

前節で述べたようにして生成した Java ソースコードに対して，Concolic Testing を適用すると，画面遷移図上の様々なテストパスに基づき大量のテストケースが生成されるが，実際のテストでは，テストすべきテストパスについてのみ効率よくテストデータを得られると良い．

例えば，図 1 で，テスト対象システムの正常動作を確認するために，画面 1,3,4 というテストパスについてのテストケースのみが必要な場合を考える．図 1 の設計モデルから生成したソースコードに対して Concolic Testing を適用すると，画面 1,2,1,2…，画面 1,3,5,3,5…といったような不必要なテストパスに基づくテストケースが多く生成され，必要なテストパスに基づいたテストケースをなかなか得ることができない．

テストすべきテストパスとそのテストデータを効率よく得ることができるようにするため，ユーザが得たいテストパスの画面遷移列を指定し，その情報を Concolic Testing の実行時に用い，指定したテストパスから外れた画面へと探索が行われそうになった場合はプログラムの実行を打ち切り，次の実行へ移る機構を考案した．図 1 の画面遷移図から生成したソースコードに対してこの探索打ち切り機構を埋め込むと以下のようになる．★印の箇所が探索打ち切り機構のコードである．

```
★ targetTathPathStates=指定したテストパスの画面遷移列;
★ count = 0;
state = SCREEN_01;
switch (state) {
    ★ if(targetTathPathStates[count]!=state){
    ★     return;//ここで探索を打ち切り，次の実行へ
    ★}
    case SCREEN_01:
        state = Screen01_Flow();
        break;
    case SCREEN_02:
```

```
        state = Screen02_Flow();
        break;
    ... 中略...
    case SCREEN_06:
        state = Screen06_Flow();
        break;
    ★ count++;
}
```

これにより，必要なテストパスについてのみ効率よくテストデータを得ることが可能となる．

上述したコードは，1 つの設計モデルから，1 つのテストパスに対するテストデータを生成しているが，実際のテストでは，1 つの設計モデルから複数のテストパスに対するテストデータを生成する場合が多い．例えば，図 1 で正常系のテストをするためであれば，「画面 1,3,4」，「画面 1,3,6」の 2 本に対してテストデータを生成したい．本研究では，複数のテストパスに対してテストデータを生成するため，「(a)1 本ずつ実行する方法」と「(b) 同時に実行する方法」という 2 つの方法を提案した．

(a)1 本ずつ実行する方法とは，各テストパスについて，単純に上述のコードで示した探索打ち切り機構の入ったコードで Concolic Testing を実行し，テストデータを生成する方法である．この方法だと，各テストパスで共通して遷移する画面があった場合に，テストパスごとに探索が重複してしまう部分が出て，効率が悪くなる可能性がある．例えば，図 4 で示すように，「画面 1,2」，「画面 1,3,4」，「画面 1,3,5」という 3 つのテストパスに対してテストデータを生成したい場合を考えると，1 本ずつ実行する方法 (図 4(a)) では，画面 1 や画面 3 の部分のパス探索に関して重複が発生する．

そこで，このような重複を解消し，より効率よく各テストパスに対してテストデータを生成する方法を提案する．(b) 同時に実行する方法では，各テストパスの情報を一度図 4(b) のような木構造にまとめ，Concolic Testing 実行時にこの情報を参照し，「画面 1,2」のテストパスについてテストデータ生成が完了したら，次の実行では，画面 1 から画面 2 への遷移を行おうとした場合は，探索を打ち切るようする，というふうに，既に抽出したテストパス，もしくは抽出する必要のないテストパスへの探索を打ち切るようにし，各テストパスについて必要なテストデータを効率よく生成できるようにした．

4. ケーススタディ

4.1 Research Question

本研究では，図 2 で示したような，複数回の参照系アクセス，更新系アクセスを伴うテストケースに対して，適切なテストデータを現実的な時間内で自動生成することを目的とする．そのため，以下の 2 点を今回のケーススタディ

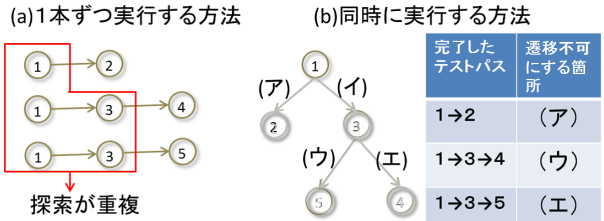

図 4 複数テストパスに対する 2 種類の探索打ち切り機構の例

の Research Question とした．

- RQ1：一定の時間内で，対象としたテストケースのうち，どれだけに対して適切なテストデータを生成できるか？
- RQ2：3.2 節で述べた，探索打ち切り機構は有効であるか？

4.2 手順

前節で述べた Research Question を確認するための実験は以下の様な手順で行う．

最初に，テスト対象アプリケーションの図 3(a) の設計モデルを作成する．そして次に，設計モデルから，シミュレーション用のソースコードを生成する (図 3(1))．将来的には，この部分は自動化する予定であるが，今回は手作業でソースコードの作成を行うものとした．そして，最後にソースコードに対して Concolic Testing の実行を行い，テストケースとテストデータを得る．テストの対象とするテストケースは，あらかじめ何らかのテスト基準に従って，設計モデルの画面遷移図から抽出しておくものとする．これらのテストケースに対してどれだけのテストデータを生成できたかを調べることで，RQ1 が確認できる．また，探索打ち切り機構を組み込まなかった場合と，組み込んだ場合のそれぞれについて Concolic Testing を行い，RQ2 の確認も行う．

4.3 Concolic Testing 実行エンジン

Concolic Testing 実行エンジンとしては，オープンソース・ソフトウェアとして BSD ライセンスで公開されている CATG を用いた．また，CATG は Java で実装された Concolic Testing 実行エンジンであり，Java ソースコードに対して，Concolic Testing を実行することができる．CATG は整数型，文字列型，日時型，DB テーブル型など，業務システム向けのテストデータ生成を行うために必要な型と，その型に対する操作を扱うことが可能である．例えば，DB テーブル型に対しては，参照系アクセス (SELECT)，更新系アクセス (UPDATE,INSERT,DELETE) などの操作がメソッドとして定義されている．Concolic 実行を行った環境は CPU が Intel Core i7 3.0GHz，メモリは 6GB，OS は Windows Vista Ultimate SP2 である．CATG では

ⓒ 2013 Information Processing Society of Japan

複数の制約ソルバを切り替えて使用することが可能であるが，今回は Yices [3] を使用した．

4.4 テスト対象のアプリケーション

提案手法の実現性を確認するための用途に，設計モデルとして Online Book Store という簡単なアプリケーションを想定し，そのアプリケーションの設計モデルを筆者ら自身で作成した．Online Book Store は以下の様な特徴を持っている．

- 書籍を購入する機能と，書籍の購入を取り消す機能という 2 機能から成る．
- 16 画面で構成され，各機能は複数の画面を遷移していくことで実現される．またエラー処理に相当する準正常フローや，エラー処理のための画面も含まれている．
- 参照系アクセスである SELECT を 4 つ，更新系アクセスである INSERT を 1 つ，UPDATE を 2 つ含む．

4.5 対象とするテストパス

今回は，開発現場でも使用されている以下のテスト観点に基づきテストパスを抽出することとした．

- 観点 (1)：正常フローを順次実行するテストパス．例えば，図 1 の画面 1,3,4.
- 観点 (2)：それぞれの準正常フローを 1 回実行するテストパス．例えば，図 1 の画面 1,2,1,3,4.
- 観点 (3)：すべての準正常フローを実行するパス．例えば，図 1 の画面 1,2,1,3,5,3,4.
- 観点 (4)：それぞれ準正常フローを指定された回数実行するテストパス (今回は回数を 2 とした)．例えば，図 1 の画面 1,2,1,2,1,3,4.

この観点に従いテストパスを抽出すると，Online Book Store では合計 27 本のテストパス (及びテストケース) が存在する．

4.6 タイムアウト

最初に，探索打ち切り機構有り (1 本ずつ実行する方法) で，指定された各テストパスにつき実行上限回数を 1000 回として，Concolic Testing を適用し，このときに各テストパスのテストデータ生成にかかった時間の合計値をまず記録し，次に，探索打ち切り機構なし，探索打ち切り機構有り (同時に実行する方法) については，この合計時間をそれぞれタイムアウトの時間として用いた．

4.7 結果

実験の結果を表 1 に示す．合計値を見ると，探索打ち切り機構がなしの場合では，全体の 4% のテストケースにしかテストデータが生成できなかったのに対し，探索打ち切り機構がある場合では最大 78% のテストケースについてテストデータの生成を行うことが出来た．

表 1 実験結果

探索打ち切り機構	実行回数	テストデータ生成率
観点 (1)（実行時間： 2 分 30 秒）		
無し	128	1/2(50%)
(a)1 本ずつ	116	2/2(100%)
(b) 同時	101	2/2(100%)
観点 (2)（実行時間：68 分 00 秒）		
無し	2276	0/12(0%)
(a)1 本ずつ	3175	9/12(75%)
(b) 同時	2538	8/12(67%)
観点 (3)（実行時間：27 分 00 秒）		
無し	1079	0/3(0%)
(a)1 本ずつ	1045	2/3(67%)
(b) 同時	1000	2/3(67%)
観点 (4)（実行時間：45 分 00 秒）		
無し	1628	0/10(0%)
(a)1 本ずつ	2106	8/10(80%)
(b) 同時	1543	5/10(100%)
観点 (1)〜(4) の合計		
無し	-	1/27(4%)
(a)1 本ずつ	-	21/27(78%)
(b) 同時	-	17/27(63%)

4.8 考察

4.8.1 RQ1

探索打ち切り機構がある場合でもテストデータを生成することができなかったテストケースには「(i) 指定した時間内に生成できなかったテストケース」，「(ii) 絶対にテストデータを生成することができないテストパスをもつテストケース」の 2 種類が存在した．

(i) に関しては，今回用いた Online Book Store のような規模の極めて小さなアプリケーションでさえも，ある程度存在するため，アプリケーションの規模が大きくなり，結果として設計モデルから生成される Java ソースコード内の分岐条件が増えると，このようなテストケースは更に増えてしまうと考えられるため，更なる性能改善の工夫が必要である．

(ii) に関しては観点 (2) において 1 件だけ存在した．これは，テストパス抽出において，アプリケーションのセマンティクスを全く考慮せずにテストパスを抽出したためであり，「自分の購入履歴が存在しないことを確認した後に，自分の購入履歴から 1 件以上の購入書籍を選んで購入のオーダをキャンセルする」という矛盾したテストパスを抽出していたためである．

4.8.2 RQ2

(1)〜(4) 全てのテスト観点において，探索打ち切り機構は，有効であった．観点 (2),(3),(4) のような複雑なテストパスにおいては，探索打ち切り機構が存在しない場合，全くテストデータが生成できていなかった．これは複雑であるほど，探索打ち切り機構が存在しない場合には，取得したいテストパスとは別の余計な未到達パスの探索に時間がかかってしまうためである．そのため，このような場合に

ⓒ 2013 Information Processing Society of Japan

は特に探索打ち切り機構が有効であるとわかった．

また，今回の評価では 1 本ずつ実行する方法の方が，同時に実行する方法よりもテストデータ生成率が高かった．これは，1 本ずつ実行する方法では個別のテストパスごとにタイムアウトを設けているのに対し，同時に実行する方法では，全体に対して 1 つのタイムアウトを設けていることに起因していた．1 本ずつ実行する方法では，Concolic Testing の実行回数の上限を 1000 回と決めて打ち切ったたことに対し，同時に実行する方法では，テストデータを生成すべきテストパスのどこか一箇所にでも，分岐条件などが多くテストデータの生成に時間がかかるところがあると，そこに時間がかかってしまい，多くのテストパスに対してテストデータ生成が行えなくなってしまうためである．

5. 結論

本研究では，参照系アクセス，更新系アクセスを伴い，複数の画面間を遷移する結合テストのテストケースに対し，テストデータである DB 初期状態と各画面の画面入力値を自動で生成する手法を提案した．

本手法の特徴は，ソフトウェアの設計モデルからシミュレーション用ソースコードを生成し，生成したソースコードに対して Concolic Testing を実行することでテストデータを生成している点である．提案手法では，余計なテストすべきテストパスへの探索を打ち切ることにより，効率良く必要なテストケースのテストデータを生成できるような工夫も行っている．本論文では Online Book Store という簡単な例題を用いて，提案手法の実現性についての考察を行った．

今後は，性能的な問題の改善に取り組むとともに，実開発案件への適用実験を行い，提案手法の有効性を確認していく予定である．

参考文献

[1] Dbmonster. http://dbmonster.kernelpanic.pl/.
[2] Visual studio database edition. http://www.microsoft.com/japan/msdn/vstudio/.
[3] The yices smt solver. http://yices.csl.sri.com/.
[4] 擬似個人情報ジェネレータ．http://www.start-ppd.jp/.
[5] Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors. *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*. IEEE, 2011.
[6] David Chays, Saikat Dan, Phyllis G. Frankl, Filippos I. Vokolos, and Elaine J. Weber. A framework for testing database applications. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '00, pp. 147–157, New York, NY, USA, 2000. ACM.
[7] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, Vol. 4590 of *Lecture Notes in Computer Science*, pp. 20–36. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-73368-3_5.
[8] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, WEASELTech '07, pp. 31–36, New York, NY, USA, 2007. ACM.
[9] Jon Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linköping*, pp. 21–28. ECSEL, October 1999.
[10] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pp. 151–162, New York, NY, USA, 2007. ACM.
[11] Andre Takeshi Endo and Adenilso Simao. Model-based testing of service-oriented applications via state models. In *Proceedings of the 2011 IEEE International Conference on Services Computing*, SCC '11, pp. 432–439, Washington, DC, USA, 2011. IEEE Computer Society.
[12] Barry O'Sullivan Frederic Benhamou, Narendra Jussien. *Trends in Constraint Programming*. ISTE, 2010.
[13] Shoichiro Fujiwara, Kazuki Munakata, Yoshiharu Maeda, Asako Katayama, and Tadahiro Uehara. Test data generation for web application using a uml class diagram with ocl constraints. *Innovations in Systems and Software Engineering*, Vol. 7, pp. 275–282, 2011. 10.1007/s11334-011-0162-3.
[14] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pp. 263–272, New York, NY, USA, 2005. ACM.
[15] Haruto Tanno and Takashi Hoshino. Reducing the number of initial database states for integration testing. In *Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*, COMPSACW '13, Kyoto, Japan, 2013. IEEE Computer Society.
[16] Haruto Tanno, Xiaojing Zhang, and Takashi Hoshino. Design-model-based test data generation for database applications. In *Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*, ISSREW '12, pp. 201–206, Washington, DC, USA, 2012. IEEE Computer Society.
[17] 丹野治門, 張暁晶, 星野隆. 設計モデルを利用したテスト用データベース自動生成手法. 情報処理学会論文誌, Vol. 53, No. 2, pp. 566–577, feb 2012.
[18] 藤原翔一朗, 宗像一樹, 片山朝子, 前田芳晴, 大木憲二, 上原忠弘, 山本里枝子. 1b-3 smt solver を利用した web アプリケーション用テストデータの生成 (テスト・検証, 一般セッション, ソフトウェア科学・工学, 情報処理学会創立 50 周年記念). 全国大会講演論文集, Vol. 72, No. 1, pp. 1–281–282, mar 2010.

ⓒ 2013 Information Processing Society of Japan