Vol.2020-CLE-30 No.9
2020/3/8

# シンボリック実行によるテストケース生成を行う空欄補充式Java学習システムの開発

三輪 航也[1,a)]　北村 泰彦[2,b)]

**概要**：プログラミング教育において，実際にプログラミングを行うことは重要であり，それを補助する目的でプログラミング学習システムについての研究がなされている．中でも初学者向けに空欄補充式プログラミング学習システムが開発されている．これらのシステムでは正誤判定を行う際，プログラムが期待される実行結果を出力するかを確認するためにテストケースを用いることがある．しかし手作業でテストケースを用意する場合，完全な判定を行うには不十分であったり，また教授者の負担となることがある．そこで，入力をシンボル値という具体値を伴わない値として扱うシンボリック実行によるテストケースの自動生成を行うことが考えられる．本研究では，シンボリック実行によるテストケース生成を行う空欄補充式 Java 学習システムを開発した．しかし，シンボリック実行の問題点として，ループ文を含む場合に大量のテストケースが生成されてしまうことがある．そこで，空欄箇所のみをシンボリック実行対象とすることで，テストケース生成数を削減する手法を提案した．

## 1. はじめに

近年，情報化社会が加速するにつれて，IT 人材がさらに不足することが予想されている [5]．これにより，プログラミング教育の重要性が増大すると考えられる．それに関連して，プログラミング学習システムに関する研究が多数なされており [1], [4], [7]，中でもプログラミング初学者向けに，プログラムの一部を空欄に置き換え，学習者が空欄に当てはまる解答を入力する空欄補充式プログラミング学習システムに関する研究が行われている．西川は Java プログラミング初学者向けの「穴埋め式 JAVA 学習システム」を開発した [6]．本システムでは学習者の解答を埋め込んだプログラムの出力と，模範解答を埋め込んだプログラムの出力を比較することで正誤判定を行う．このような実行出力を比較する正誤判定において，テストケースを用いることが考えられる．テストケースとは，プログラムが期待される実行結果を出力するか否かを確認するために用いられる入出力のペアを指す．テストケースを用いる利点として，入力に任意の値を設定することができる点がある．しかし正誤判定時に用いるテストケースが不十分な場合，適切な正誤判定を行うことができない場合がある．加えて，完全なテストケースを手作業で用意するのは手間がかかり，教授者に大きな負担がかかってしまう．そこで，本研究ではシンボリック実行 [2] によって，プログラムに与えるテストケースを自動生成する．しかし，シンボリック実行を行った際に，対象プログラムによっては大量のテストケースが生成されてしまう場合がある．

本研究では，シンボリック実行によるテストケース生成を行う空欄補充式 Java 学習システムを開発する．また，空欄補充問題の特徴を利用してテストケース生成数を削減する手法を提案する．

## 2. プログラミング学習システムとシンボリック実行

プログラミング教育において，実際にプログラミングを行うことは重要であり，それを補助する目的でプログラミング学習システムについての研究がなされている．これらのシステムは記述式と空欄補充式の 2 種類に分類することができる．

記述式プログラミング学習システムは，学習者が出題された要件を満たすプログラム全体を記述する解答方式を採るプログラミング学習システムであり，意味解析によって論理エラーなどに対する助言を行う INTELLITUTOR[4] や，プログラム入力補助機能やステップ実行機能などを備えた PEN[7] などが開発されている．しかし，プログラミング初学者にとってはプログラム記述量が多くなり，高難度となってしまうことが考えられる．

1 関西学院大学大学院理工学研究科情報科学専攻
2 関西学院大学理工学部情報科学科
a) dbi93290@kwansei.ac.jp
b) ykitamura@kwansei.ac.jp

ⓒ 2020 Information Processing Society of Japan

1から15までの数字において3の倍数の時に「Fizz」，5の倍数の時に「Buzz」，3と5の両方で割り切れるときに「FizzBuzz」と出力するプログラムを完成させなさい．

```
public class FizzBuzz{
  public static void main( String[] args ) {

    for(int i=1; i<=15; i++){
      System.out.print(i);
      if(        ){
        System.out.print("        ");
      }
      …
```

**図 1**　空欄補充問題の例

```
public static void method1(int x){ //xは入力された西暦
        {
                System.out.println(x+"年はうるう年です．");
        }       {
                System.out.println(x+"年は平年です．");
        }       {
                System.out.println(x+"年はうるう年です．");
        }else{
                System.out.println(x+"年は平年です．");
        }
}
```

**図 2**　うるう年判定メソッド

空欄補充式プログラミング学習システムは，図 1 のように問題文と空欄が設定された問題プログラムが学習者に与えられ，学習者はその空欄に解答を入力する解答方式を採るプログラミング学習システムである．記述式と比較すると学習者が解答するプログラムが少なく済み，また教授者が出題したい学習内容に焦点を絞ることが可能になることで，よりプログラミング初学者向けとなっている．

### 2.1　Java プログラミング学習支援システム JPLAS

船曳らは空欄補充式 Java プログラミング学習システムである JPLAS を開発した [1]．JPLAS では，教授者の負担軽減のため自動採点を行う．また JPLAS では，予約語といった Java の文法を学ぶ初学者を対象にした空欄補充問題を出題する．学習者の解答が正しいかどうかを自動で採点する際には，空欄に入力されている文字列が正答と一致しているか否かといった文字列マッチングによって正誤判定を行う．また塔 [10] は JPLAS 向けに，制約グラフに基づく空欄エレメント選択アルゴリズムを開発した．このアルゴリズムを用いることによって，空欄に設定できるのは予約語や一度プログラム内で現れた変数名など解が一意に定まる箇所のみに限定することが可能となった．しかし，解の一意性を満たさない箇所は空欄に設定することができないため，空欄設定の自由度は低い．

### 2.2　穴埋め式 JAVA 学習システム

西川は Java プログラミング初学者向けに穴埋め式 JAVA 学習システムを開発した [6]．穴埋め式 JAVA 学習システムは Web アプリケーションであり，学習者の解答をプログラム内の空欄に埋め込み，サーバ側でコンパイル・実行を行うという特徴を持つ．また JPLAS の文字列マッチングによる正誤判定とは異なり，学習者の解答を埋め込んだプログラムの出力と，模範解答を埋め込んだプログラムの出力を比較することで正誤判定を行う．そのため，空欄設定の自由度は制限されない．しかし，問題プログラムの入力はあらかじめに記述しておく必要があり，それ以外の入力に対する正誤判定は行うことができない．

### 2.3　テストケースを用いた正誤判定の問題点

上記の関連研究から，空欄補充式学習システムにおいて正誤判定を行う手段として，文字列マッチングによる正誤判定と実行出力を比較する正誤判定が存在する．実行出力を比較する正誤判定を行う場合，解が一意に定まる箇所のみを空欄として設定する必要がある文字列マッチングによる正誤判定と比べて，空欄設定の自由度が制限されないことから出題の幅を広げることができる．実行出力の比較による正誤判定では，テストケースを用いることが考えられる．ここで述べるテストケースとは，プログラムが期待される実行結果を出力するか否かを確認するために用いられる，入出力のペアからなるものを指す．西川の穴埋め式 JAVA 学習システムは，問題プログラムの入力をあらかじめ記述するため，正誤判定で入力値が固定されてしまうが，テストケースを用いた正誤判定では入力値として任意の値を扱うことができる．その一方でテストケースが不十分だった場合，学習者の解答が正しくなくても正解と判定されてしまうことがある．図 2 を例に説明する．このメソッドは入力 $x$ がうるう年であるか否かを判定するものである．模範解答は最上部の空欄からそれぞれ，`if(x%400==0)`，`if(x%100==0)`，`if(x%4==0)` である．ここで，表 1 のテストケースがあらかじめ用意されているとする．このとき，学習者が最上部の空欄に `if(x%10==0)` と誤った解答を入力したとしても，表 1 のようにテストケースの出力と学習者の解答を埋め込んだプログラムの実際の出力が一致してしまうため，正解と扱われてしまう．このように手作業でテストケースを用意すると不十分な場合がある．竹内らの研究 [8] では空欄補充問題のテストケース生成に，Haskell で記述されたプログラムを自動でテストするツールである QuickCheck を用いることでテストケースの自動生成を行っているが，QuickCheck 用のプログラムを手作業で記述する必要があり，完全な自動生成とは言えない．そこで本研究では，シンボリック実行を用いることでテストケースの自動生成を行う．

### 2.4　シンボリック実行

シンボリック実行とは，入力をシンボル値という具体値

| テストケース | | 実際の出力 |
|---|---|---|
| 入力 $x$ | 出力 | |
| 2000 | 2000 年はうるう年です. | 2000 年はうるう年です. |
| 2019 | 2019 年は平年です. | 2019 年は平年です. |
| 2020 | 2020 年はうるう年です. | 2020 年はうるう年です. |

**表 1** テストケースが不十分な例

| テストケース | | 実際の出力 |
|---|---|---|
| 入力 $x$ | 出力 | |
| 400 | 400 年はうるう年です. | 400 年はうるう年です. |
| 100 | 100 年は平年です. | 100 年はうるう年です. |
| 4 | 4 年はうるう年です. | 4 年はうるう年です. |
| 1 | 1 年は平年です. | 1 年は平年です. |

**表 2** テストケースが十分な例

を伴わない値として扱う実行方式である [2]．代入や演算などに含まれているシンボル値に対する条件式を解くことで条件を満たす入力値を特定することができる．Java 向けシンボリック実行ツールとしては Symbolic Pathfinder[3] があり，メソッドごとにシンボリック実行を行うことが可能である．図 3 はうるう年判定メソッドに対してシンボリック実行を行った例である．まず入力 $x$ にシンボル値 $Symx$ を代入する．そして，$Symx$ に対する条件分岐において，真になる場合と偽になる場合それぞれの条件式をパス条件として加えていく．これをすべての条件分岐に対して行い，最終的に赤枠内のような各条件分岐を通るための条件式が求まる．これらの条件式が充足可能な場合，そのシンボル値に対する具体値を出力する．この具体値がテストケースの入力となる．表 2 は実際にシンボリック実行によってうるう年判定メソッドのテストケースを求めた例である．学習者が先述の誤った解答を行った場合でも，入力 $x$ が 100 の場合でテストケース出力と学習者の解答を埋め込んだプログラムの出力が一致しないため，正誤判定を正しく行うことができる．

シンボリック実行を用いる際の問題点が 2 点存在する．1 つ目は参照型変数や再帰を含むプログラムではシンボリック実行が不可能な点である．これはシンボリック実行が抱える問題であるため，本研究では扱わない．2 つ目はシンボリック実行対象プログラムがループ文を含む場合に大量のテストケースが生成されてしまうことがある点である．この問題に対しては，設定された空欄箇所のみに対してシンボリック実行を行うことで，生成されるテストケース量を削減することができると考えられる．本研究では，シンボリック実行を用いてテストケースを自動生成可能な空欄補充式 Java 学習システムを開発し，テストケース生成数を削減可能なプログラムについて考察する．

## 3. シンボリック実行によるテストケース生成を行う空欄補充式 Java 学習システムの開発

本研究では，シンボリック実行を用いてテストケース

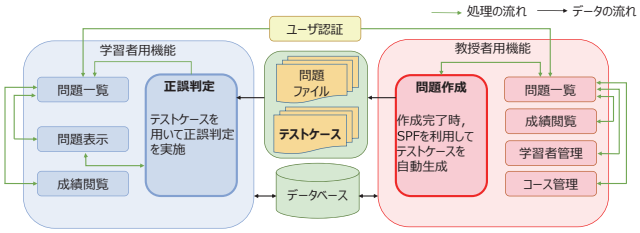


**図 4** システム構成図

を自動生成し，それを用いた正誤判定を行う空欄補充式 Java 学習システムを開発した．本システムでは，教授者によって問題が作成された時に，問題プログラムに対してシンボリック実行を行い，正誤判定を行うのに必要なテストケースを自動生成する．このとき，Java のメソッドごとにシンボリック実行をすることができる Symbolic Pathfinder(SPF)[3] を用いる．生成されたテストケースはサーバに保存される．学習者の解答時，問題プログラムとそれに対応するテストケースを用いて正誤判定を行う．

図 4 にあるように本システムは，ユーザ認証機能，教授者用機能，学習者用機能，出題する問題やテストケースなどのファイル，データベースから構成される．各機能やデータベースは西川 [6] と長野 [9] が開発したものをもとに実装した．

### 3.1 ユーザ認証機能

ユーザの認証を行う機能である．ユーザは認証画面において，ID とパスワードを入力してログインする．この際，データベースの教授者情報の中から，入力された ID と一致する ID を検索する．一致する ID が存在する場合，入力されたパスワードの照合を行う．一致した場合，入力した ID の教授者として承認する．ID が教授者情報の中で一致しない場合，学習者情報の中から検索する．学習者情報内で一致する ID が存在する場合，パスワードの照合を行い，一致すると入力した ID の学習者として承認する．

### 3.2 教授者用機能

教授者用機能では，教授者が学習者に出題する問題の作成,，コースの管理，学習者の管理，学習者の成績の閲覧を行うことができる．ユーザ認証成功後，教授者用の問題一覧画面を表示する．そして，教授者は問題作成画面に移行し空欄補充問題の作成を行う．作成が完了すると，自動でテストケースの生成を行う．また，学習者にどの問題を出題するかを設定するため，コース管理画面へと移行する．また，新たに学習者を新規登録する場合は，学習者登録画面において登録を行う．学習者の成績は成績閲覧画面で確

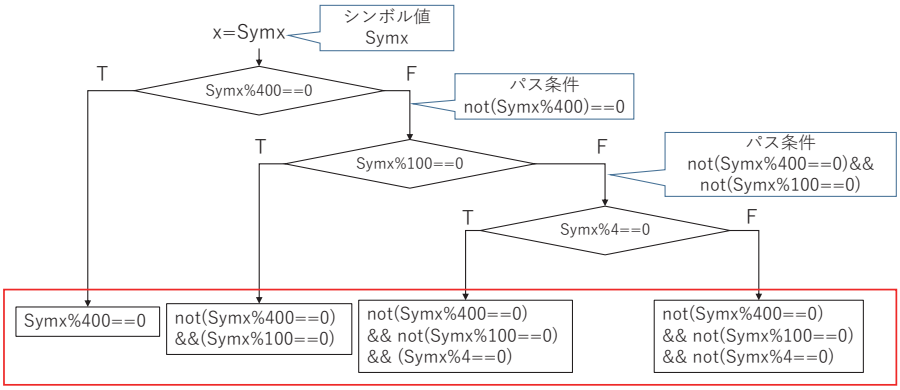


**図 3**　うるう年判定メソッドにシンボリック実行を行った例

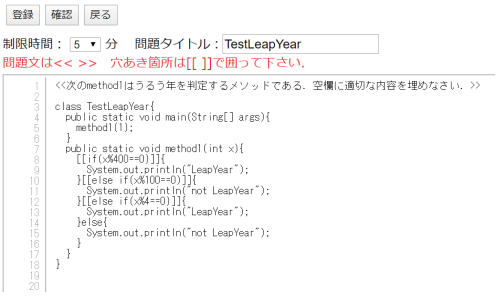


**図 5**　問題作成画面

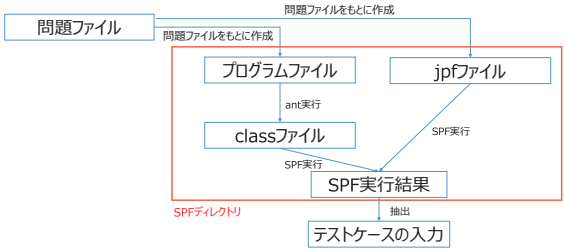


**図 6**　SPF によるテストケース生成

認できる．

### 3.2.1　問題作成機能

空欄補充問題を作成する機能である．教授者は図 5 のような問題作成画面において，制限時間の設定，問題タイトル，問題プログラムおよび”[[”,”]]”,”<<”,”>>”を入力する．”[[”と”]]”は，空欄に設定する箇所であり，その間に含まれている部分は，その空欄箇所の模範解答となる．また，”<<”と”>>”は問題文として表示する部分となる．教授者は，それらの記号を問題プログラム中に記述することで，それぞれ空欄箇所と問題文として設定することができる．

問題作成完了時，Symbolic Pathfinder を用いてテストケースの自動生成を行う．テストケース生成の流れは図 6 のようになっている．まず，空欄箇所情報などが含まれている問題ファイルからシンボリック実行可能な形となるようにプログラムファイルを生成し，ant によるビルドを行うことで class ファイルを生成する．また，問題ファイルをもとに jpf ファイルを作成する．jpf ファイルとは，Symbolic Pathfinder 実行時に必要な設定を保存しておくファイルである．class ファイルと jpf ファイルを生成した後 Symbolic Pathfinder によるシンボリック実行を行い，その実行結果からテストケースの入力となる具体値を取り出し，保存する．

### 3.2.2　コース管理機能

教授者が担当するコースを管理するための機能である．本システムにおけるコースとは 1 人の教授者と，その教授者が担当する学習者，出題する問題から構成される．図 7 のようなコース管理画面にて，学習者と出題する問題の選択，制限時間オン・オフの設定が可能である．図 7 の左表は，システムで作成された問題一覧となっており，問題 ID，問題タイトル，問題プログラムのクラス名を表示している．右表は，システムに登録されている学習者一覧となっており，未登録学習者はどのコースにも属していない学習者を，登録済学習者はこのコースに属している学習者を表している．これらはすべてチェックボックスのオン・オフによって選択し，完了ボタンを押すことで設定を反映させる．

### 3.2.3　学習者管理機能

システムを利用する新規学習者の追加を行うための機能である．教授者は図 8 のような学習者管理画面にて，登録

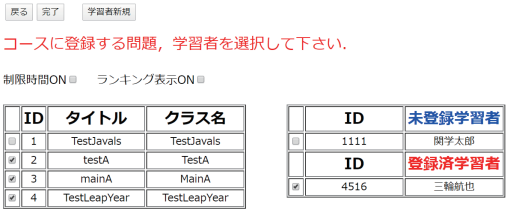


図 7　コース管理画面

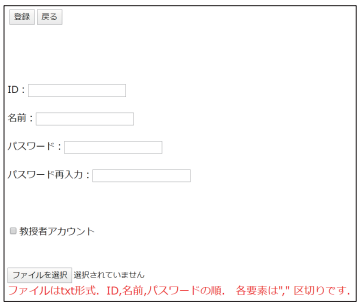


図 8　学習者管理画面

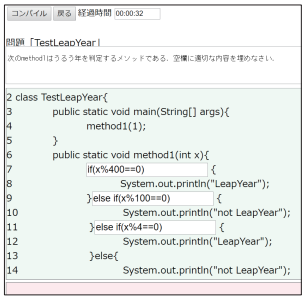


図 9　問題表示画面

する学習者の ID，名前，パスワードを入力する．また，登録する学習者の数が多い場合，学習者情報をあらかじめ記述したテキストファイルをアップロードすることで一括登録が可能である．これらの学習者情報はデータベースに登録される．

### 3.3　学習者用機能

学習者用機能では，出題された問題の解答と学習者自身の成績を閲覧することができる．ユーザ認証成功後，学習者用の問題一覧画面を表示する．学習者は，解答する問題を選択することで問題表示画面へ移行し，その問題に対しての解答を行う．その際，システムはあらかじめ生成したテストケースを用いて正誤判定を行う．解答終了後，学習者は成績閲覧画面にて自身の成績を確認することができる．

#### 3.3.1　問題表示機能

問題作成機能で教授者が作成した空欄補充問題を，学習者に出題する機能である．図 9 のように，学習者は問題表示画面上部の問題文を読み，テキストボックスに解答を入力する．

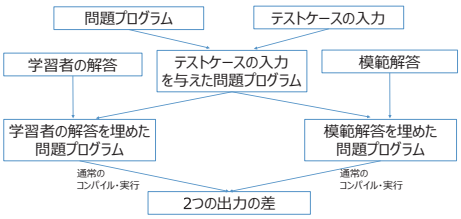


図 10　テストケースを用いた正誤判定

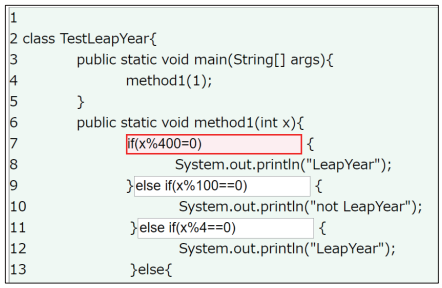


図 11　コンパイルエラーが発生した場合の例

図 12　不正解画面例

#### 3.3.2　正誤判定機能

正誤判定には自動的に生成されたテストケースを用いる．正誤判定の流れは図 10 のようになっている．はじめに問題プログラムに対してテストケースの入力を与えるようにプログラムを書き換える．このとき，正誤判定を行うメソッドに対して入力を与え，そのメソッドを main メソッドから呼び出す形にプログラムを書き換える．次に，書き換えたプログラムに対して，学習者の解答と模範解答を埋めたプログラムをそれぞれ生成する．最後に，それら 2 つをコンパイル・実行し，出力を比較することで正誤判定を行う．コンパイルエラーが発生した場合，図 11 のように原因となる空欄を赤くハイライトすることで修正を促す．コンパイルが成功し，通常実行した際の 2 つの出力が一致した場合には正解と判定し，正解した旨を通知する正解画面を表示する．出力が一致しない場合には不正解と判定し，図 12 のように，入力値，学習者の出力結果と期待される出力結果の両方を表示する不正解画面を表示し，解きなおしするよう指示する．

図 13 は，図 9 の学習者の解答を埋めた問題プログラム

```
1  ↓
2  class TestLeapYear[↓
3    public static void main(String[] args)[↓
4      method1(400);↓
5      method1(100);↓
6      method1(4);↓
7      method1(1);↓
8  ↓
9    ]↓
10   public static void method1(int x)[↓
11     if(x%400==0)[↓
12       System.out.println("LeapYear");↓
13     ]else if(x%100==0)[↓
14       System.out.println("not LeapYear");↓
15     ]else if(x%4==0)[↓
16       System.out.println("LeapYear");↓
17     ]else[↓
18       System.out.println("not LeapYear");↓
19     ]↓
20   ]↓
21 ]↓
22 [EOF]
```

**図 13**　学習者の解答を埋めた問題プログラム生成例

```
1 public void method1(int x) {
2   int i=0;
3   while(i<x){
4     System.out.println(i+"回ループ");
5     i++;
6   }
7 }
```

空欄箇所

**図 14**　テストケースが大量に生成されるメソッド例

の生成例である．あらかじめ生成させたテストケースの入力である，400，100，4，1 を用いて正誤判定を行いたいメソッドである method1 を main メソッドから呼び出す形にプログラムを書き換える．同様の処理を模範解答を埋めた問題プログラムに対しても行い，それぞれの出力を比較することで正誤判定を行う．

## 4. テストケース生成数削減可能なプログラムについての考察

シンボリック実行によってテストケースを自動生成した場合に，シンボリック実行対象メソッドの引数がループの条件式に関係していると，大量のテストケースが生成されてしまう．図 14 の場合，このメソッドにシンボリック実行を行うと引数 $x$ がシンボル値 $symx$ をとる．この場合，while 文の条件式にシンボル値 $symx$ が含まれている．while 文を実行する際，ループが起こるたびにカウンタ変数 $i$ は 1 加えられることになり，各条件分岐の条件式は $0<symx$，$1<symx$，$2<symx$，…，といったように生成される．本システムで用いている Symbolic Pathfinder ではシンボル値の範囲をあらかじめ設定しておくので，範囲を 1 から $n$ としたとき条件式は計 $n$ 通りとなる．その後，各条件式の生成された全ての充足可能な条件式に対する具体値を求めると，結果として生成されるテストケース数は $n$ 通りとなる．

しかし，空欄補充問題において，正誤判定は空欄が設定されている箇所のみ正しい処理が行われていることを確認することができればよく，それ以外の箇所の処理を確認する必要はない．図 14 であれば，4 行目の println 文が正しく記述されているかを確認できればよく，それ以外の処理に対して正誤判定を行う必要はない．そのため，シンボリック実行を行う範囲も空欄の存在する箇所に限定することができる．このように，正誤判定を行う必要のある限定範囲を選択し，新たなシンボリック実行対象メソッドとして独立させる実行対象の限定手法を提案する．実行対象を限定することによって条件式内に存在するシンボル値を取る変数，すなわち対象メソッドの引数の個数を減らす，または任意の値を取ることができるようにすることでテストケース数を削減する．

```
1 public void method1(int x) {
2   int i=0;
3   while(i<x){
4     System.out.println(i+"回ループ");
5     i++;
6   }
7 }
```

空欄箇所　限定範囲

**図 15**　条件分岐が含まれなくなることで削減可能なプログラム例

```
1 public void method1(int x) {
2   int i=0;
3   while(i<x){
4     method2(i);
5     i++;
6   }
7 }
```

```
8   public void method2(int i) {
9     System.out.println(i+"回ループ");
10  }
```

空欄箇所

**図 16**　図 15 に対する実行範囲の限定例

テストケース数を削減できるプログラム構造は次の 2 通りに分類することができる．1 つ目は，引数が関係する条件式が含まれなくなった場合である．例えば図 15 のような問題プログラムにおいて，シンボル値の範囲を 1 から $n$ と設定していた場合，生成されるテストケースは $n$ 通りとなる．ここで，実行範囲の限定を行うことを考える．正誤判定を行う箇所を別のメソッドとして独立させたものが図 16 の method2 となる．このとき，シンボル値を取る引数 $i$ は任意の値を取ることができるため，必要なテストケースは 1 通りとなる．このように，シンボル値に対する条件式の数を減少させることで，テストケース生成数を削減することができる．

2 つ目は，シンボリック実行対象メソッド内に複数の条件分岐が存在し，最も外側を除く分岐制御部に空欄が存在する場合である．ここでの分岐制御部とは，if，switch 文や for，while 文の処理部以外の部分を指す．例えば，図 17 のような問題プログラムにおいて，シンボル値の範囲を 1

```
1 public void method1(int x, int y) {
2   for(int i=0;i<x;i++){
3     for(int j=0;j<y;j++){
4       System.out.println(i+","+j);
5     }
6   }
7 }
```

空欄箇所　限定範囲

**図 17**　条件分岐を減らすことで削減可能なプログラム例

```
1   public void method1(int x, int y) {
2     for(int i=0;i<x;i++){
3       method2(y,i);
4     }
5   }
6   public void method2(int y,int i) {
7     for(int j=0;j<y;j++){
8       System.out.println(i+","+j);
9     }
10  }
```

空欄箇所

**図 18**　図 17 に対する実行範囲の限定例

から $n$ と設定していた場合，生成されるテストケースは $n^2$ 通りとなる．ここで，実行範囲の限定を行うことを考える．空欄の存在するブロックより外側のブロックは正誤判定を行う必要がないため省略することができる．したがって正誤判定を行う範囲を別のメソッドとして独立させたものが図 18 の method2 となる．このとき，シンボル値をとる引数のうち $i$ は任意の値を取ることができるため，$i$ に関する条件式は存在しなくなることでテストケース数は $n$ 通りまで削減される．このように，実行範囲を限定することで空欄の存在するブロックより外側の条件分岐の数を減らすことが可能であり，そのため条件式に現れるシンボル値の個数を減らすことができ，テストケース数を削減することができる．

## 5. まとめ

本研究では，シンボリック実行を用いてテストケースを自動生成可能な空欄補充式 Java 学習システムを開発した．本システムによってテストケースの自動生成が可能となった．しかし，シンボリック実行によるテストケース自動生成の問題点として，ループ文を含む場合に大量のテストケースが生成されてしまうことがある．しかし空欄補充問題では空欄の存在しない箇所は正誤判定を行う必要が無い．そこで，シンボリック実行対象メソッドに対して，正誤判定を行う必要のある範囲を限定し，新たなシンボリック実行対象メソッドとして独立させる手法を提案し，テストケース生成数を削減することができるプログラム構造について考察を行った．その結果，限定範囲内にシンボリック実行対象メソッドの引数が関係する条件式が含まれなくなった場合と，限定前のシンボリック実行対象メソッド内に複数の条件分岐が入れ子構造で存在し，そのうち最も外側のブロック内に空欄が存在する場合にはテストケース数の削減が可能であることを示した．今後の課題として，実行範囲の限定手法では，正誤判定を行う必要がある限定範囲を適切に選択する必要がある．そのため，限定範囲を機械的に選択する手順を作成する必要がある．この際，数行を 1 つの塊として限定範囲に設定するのではなく，正誤判定に必要な各行を選択することで，よりテストケース数を削減できるプログラム構造を増やすことが考えられる．また，本研究では扱わなかった参照型変数や再帰を含むプログラムに対してテストケース生成をどのように行うのかという問題がある．

## 参考文献

[1] N. Funabiki, Y. Korenaga, Y. Matsushima, T. Nakanishi, and K. Watanabe. An online fill-in-the-blank problem function for learning reserved words in Java programming education. In *26th International Conference on Advanced Information Networking and Applications Workshops*, pp. 375–380. IEEE, 2012.

[2] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[3] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic Pathfinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[4] H. Ueno. Intellitutor: A knowledge based intelligent programming environment for novice programmers. In *34th IEEE Computer Society International Conference*, pp. 390–395. IEEE, 1989.

[5] 経済産業省. - IT 人材需給に関する調査 - 調査報告書. `https://www.meti.go.jp/policy/it_policy/jinzai/houkokusyo.pdf`. 2019 年 7 月 16 日アクセス.

[6] 西川知宏. 穴埋め式 JAVA 学習システムの開発. 関西学院大学理工学部情報科学科卒業論文, 2011.

[7] 西田知博, 原田章, 中村亮太, 宮本友介, 松浦敏雄他. 初学者用プログラミング学習環境 PEN の実装と評価. 情報処理学会論文誌, 48(8):2736–2747, 2007.

[8] 竹内亮太郎, 大久保弘崇, 粕谷英人, 山本晋一郎. 空欄補充問題の自動生成による Haskell プログラミング学習支援環境. 情報処理学会研究報告, 2011-SE-171(15):1–8, 2011.

[9] 長野喬祐. 教授者機能を備えた穴埋め式 Java 学習システムの開発. 関西学院大学理工学部情報科学科卒業論文, 2018.

[10] 塔娜. Java プログラミング学習支援システムにおける空欄エレメント選択アルゴリズムの提案. 岡山大学大学院自然科学研究科博士論文, 2017.