記号的実行による 統合テスト用テストデータ自動生成

吉原　慧†　　高田　眞吾†
†慶應義塾大学大学院理工学研究科

1　はじめに

ソフトウェアの品質保証に欠かせないテストは非常にコストがかかる．そのため，コストを削減するためのテストの自動化手法が数多く研究されている．しかしその多くは単体テストを対象としており，統合テストを対象としたものは少ない．

単体テストとは，プログラムの基本構成単位ごとに行われるテストである．これに対して統合テストとは，プログラムの単体同士が正しく連携して動作するかを確認するテストである．メソッド間のインタフェースなど，単体テストでは発見できないエラーを発見するために行われる．

統合テストの自動化に関する既存研究としては，UML や OCL などの仕様に基づいてテストデータを自動的に生成する手法 [1] や，2 つの単体のコントロールフローグラフを結合し，統合テストが網羅すべきパスを自動的に生成する手法 [2] などが提案されている．しかし，これらの手法には，いくつかの問題点がある．例えば，テストに用いるデータが全く生成出来なかったり，詳細な仕様が必要だったりする点などである．

本研究では，統合テストを対象にしたテストデータ自動生成手法を提案する．

2　記号的実行によるテストデータ自動生成

単体テストのテストデータ自動生成手法については多くの研究が行われている．その手法の 1 つに記号的実行によるテストデータの自動生成がある．

記号的実行とは，ある変数の値を記号値のままプログラムを実行する方法である．この手法は，数値計算の領域を中心に，昔から様々な研究が行われている [3][4].

2.1　既存手法: 手順

記号的実行によるテストデータ生成手順を例を用いて述べる．図 1(a) のコントロールフローグラフで表されたメソッド foo を，変数_in について記号的に実行し，テストデータを生成する例を考える．メソッド foo の記号的実行の様子を以下に示す．

1. ステートメント a=0 を実行し，変数 a に 0 という値を代入する．
2. 条件分岐文_in>0 を評価する．
   - ここで変数_in には記号的な値が代入されており，値が一意に定まらない．そのため実行パスを 2 つに増やし，条件が真・偽の両方の場合を実行する．
3. 条件が真の場合の実行パスを考える．
   (a) 実行パスの分岐条件に_in > 0 を加える．
   (b) メソッド bar(_in) を記号的に実行し，変数 a に戻り値を代入する．
4. ステップ 3 と同様に，条件が偽の場合の実行パスを考える．

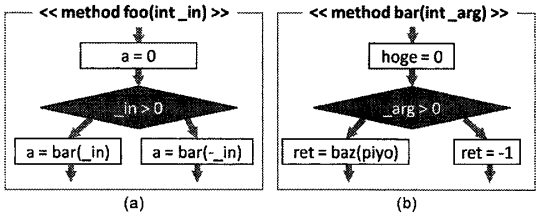

図 1: コントロールフローグラフとメソッド呼び出し

以上の手順から複数のパスと各パスを通るための分岐条件を得ることが出来る．この分岐条件を解くことによって foo のテストデータを得ることが出来る．ここで条件を解くとは，全ての条件を満たすような変数の値を少なくとも 1 つ導くことである．

2.2　問題点

ステップ 3b ではメソッド bar が記号的に実行される．このメソッド bar が図 1(b) のように分岐を含むと，実行パスがさらに増加する．このようなメソッド呼び出しが多段になると，実行パス，ならびにテストケース数が爆発的に増加してしまう．

3　提案: 統合テスト用テストデータ自動生成

本研究では，記号的実行を用いた統合テスト用テストデータ自動生成手法を提案する．

3.1　提案概要

提案手法では下記の方法を用いることでテストケース数の爆発的な増加を防ぐ．

- テスト対象は 2 メソッド間のインタフェースとする．
- Callee(呼び出される側のメソッド) の単体テストの結果を統合テストに利用する．

なお，以下では呼び出す側のメソッドを Caller，呼び出される側のメソッドを Callee と呼ぶ．

従来の手法では，多段に渡るメソッド呼び出しを全て記号的に実行することで，テスト対象パスが爆発的に増加していた．提案手法では，上記の方法を用いて記号的実行のパスを制限することで，爆発的な増加を防ぐ．

提案手法では Caller を記号的に実行する．但しこの時 Callee メソッドの呼び出しの部分は実行せず，代わりに Callee の単体テストの結果を利用する．

Automatic Test Data Generation for Integration Tests using Symbolic Execution
†Akira YOSHIHARA　†Shingo TAKADA
†Graduate School of Science and Technology, Keio University

3.2 提案手法: 手順

まず，ユーザがテスト対象となるメソッド (Caller) を指定する．提案ツールは Caller から直接呼ばれるすべての Callee の単体テスト結果を利用する．

図 2 のメソッドに対する単体テストの例を表 1 に示す．単体テスト結果は具体的な値でも記号的実行の結果でも構わない．また単体テスト結果には，無効な値が入力された場合を含むことが望ましい．

```
public class MyClassB{
  public int methodB(int capital, int year, double rate) {
    int ret = ...
    ...
```

図 2: Callee メソッドの例

表 1: 図 2 のメソッドに対する単体テストの例

	capital	year	rate	ret
1	10,000	1	0.5	10,050
2	10,000	5	0.5	10,253
3	10,000	< 0	0.5	-1

続いて，Caller の記号的実行を行う．既存の記号的実行手法では，Caller において Callee を呼び出すステートメントが実行されると，Callee の記号的実行を行い，その戻り値を用いて Caller の記号的実行が続けられる．提案手法では，Callee の記号的実行を行うことなく Caller の記号的実行を続ける．そのためには，Callee メソッドの呼び出しを，単体テストの結果を用いた条件分岐で置き換える．図 3 は，図 2 のメソッド呼び出しを表 1 の単体テスト結果で置き換えた様子を示す．この例ではメソッド呼び出しを 3 分岐で置き換える．分岐の数は単体テスト結果のレコード数と一致する．

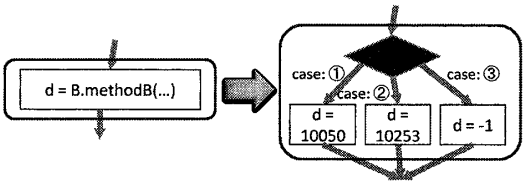

図 3: Callee メソッド呼び出しの置き換えの模式図

一番左のケース 1 を通る実行パスを考える．この実行パスには，単体テストの結果より，分岐条件 (capital=10,000 AND year=1 AND rate=0.5) が追加される．そして Callee の変数 d には，単体テストの結果より，戻り値 10,050 が代入される．

以降の Caller の記号的実行は従来手法と同様に行われる．

得られたパスの条件を解決することで，テストデータを得る．ユーザは，得られたテストデータを用いて，統合テストを実際に実行する．その際に想定と異なる出力が得られた場合，それは Callee の挙動がユーザの想定と異なっていたためである．このようにして，提案ツールで得られたテストデータを用いることで，統合エラーを発見することが出来る．

4 実装

実装は Java PathFinder[3][5] を拡張することで行った．

Java PathFinder はテスト対象プログラムのバイトコードを読み込み命令を解釈する，Java で実装された Java 仮想マシンのようなソフトウェアである．[3] で実装された記号的実行機能を拡張することで提案手法を実現した．全体像を図 4 に示す．

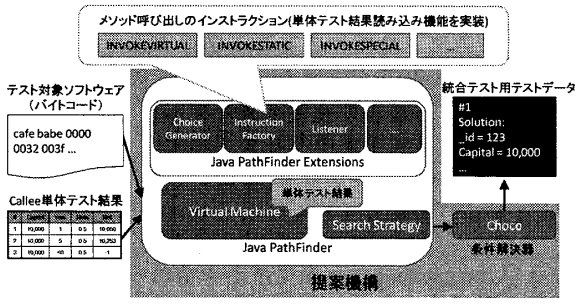

図 4: 提案機構の全体像

5 適用の限界

提案手法には，実装に依存するいくつかの限界がある．

- 記号的に実行するメソッドの引数はプリミティブ型か文字列のみ対応
- 一部の演算子が未対応
- native 修飾子をもつメソッドに対応していない

しかしながら，本実装のベースとなった Java PathFinder は，NASA の有人飛行シャトルのコンポーネントのテストデータを生成した実績 [3] もあるなど，数値計算の領域においては有用であると考えられる．

6 まとめ

記号的実行を用いて統合テスト用のテストデータを自動的に生成する手法を提案した．提案手法では，テスト対象を 2 メソッド間のインタフェースとし，Callee の単体テスト結果を用いることで，生成されるテストデータ数の削減を図った．

参考文献

[1] A. Bandyopadhyay and S. Ghosh. Test Input Generation using UML Sequence and State Machines Models. In *Proc. of ICST 2009*, pp. 121–130, 2009.

[2] O. Lemos, I. Franchin, and P. Masiero. Integration testing of Object-Oriented and Aspect-Oriented programs: A structural pairwise approach for Java. *Sci. of Comp. Program.*, Vol. 74, No. 10, pp. 861–878, 2009.

[3] C. Păsăreanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software. In *Proc. of ISSTA 2008*, pp. 15–26, 2008.

[4] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution. In *Proc. of TACAS 2005*, pp. 365–381, 2005.

[5] Java PathFinder, 2009. http://babelfish.arc.nasa.gov/trac/jpf/.