特別研究報告

題目

ESC/Java2 の反例出力を利用した Java メソッド実行パス取得ツールの試作および評価

指導教員
楠本 真二 教授

報告者
小林　和貴

平成 22 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 21 年度 特別研究報告

ESC/Java2 の反例出力を利用した Java メソッド実行パス取得ツールの試作および評価

小林　和貴

内容梗概

オブジェクト指向ソフトウェアの設計における，Design by Contract(DbC) はプログラムにおける仕様をソースコード中に記述することにより，ソフトウェア開発における誤りの箇所を明確にする手法である．

プログラムのソースコード中に事前条件・事後条件・不変条件などをアサーションという形で記述し，静的解析ツール ESC/Java2 を用いて検証することができる．これによりプログラムの保守性や信頼性を向上することができる．しかしながら，アサーションをすべて人手で記述するにはコストが掛かるという問題点がある．これを解決するために，テストケースを用いることによりアサーションの自動生成をするツール Daikon があるが，生成されるアサーションの品質はテストケースの品質に依存するという問題点がある．また，あらかじめ用意されたルール以外のアサーションを自動生成することは困難である．一方，すでに作成したソースコードに対しアサーションを人手で記述する場合，必要な仕様記述が漏れてしまう事が考えられ，プログラムに対する静的解析が不十分となる可能性がある．

そこで，本研究では，アサーション記述を支援するため，メソッドの実行パスの情報を提示する手法の提案を基にツールの実装を行った．本ツールは，対象プログラムのメソッドに対して，実行可能性のあるパスをプログラム依存グラフ (PDG) を利用して導出したのち，ESC/Java2 の反例出力機能を利用し，アサーション記述や静的解析に必要な，メソッドの実引数に関する条件式や内部変数に関する条件式を提示し，テストケースの雛形を提示する．

ツールの出力の妥当性を検証するため，いくつかの Java ソースコードに対してツールを適用した結果，有用なテストケース制約式を得られることを確認した．さらに，ツールの出力を用いて Daikon のテストケースを作成し，アサーションを自動生成した結果，いくつかの正しいアサーションを導出することができた．

主な用語

Java, 事前条件, 事後条件,JML,Esc/Java2,Daikon,PDG

目 次

1 まえがき **1**

2 研究背景 **2**

2.1 Design by Contract 2
2.2 アサーション 2
2.3 プログラム依存グラフ 3
2.4 ESC/Java2 4
2.5 Daikon 4
2.6 Inveriant Coverage 5

3 研究グループで提案された手法 **7**

3.1 概要 7
3.2 実行パスの検出 8
3.3 ESC/Java2 反例出力の解析 10

4 評価 **16**

4.1 実験環境 16
4.2 本ツールの制限 17
4.3 評価対象 17
4.4 評価実験の手順 17
4.5 実行時間の評価 18
4.6 テストケース制約出力の評価 19
4.7 Daikon によるアサーション生成結果の評価 20

5 あとがき **23**

謝辞 **24**

1 まえがき

オブジェクト指向ソフトウェアの設計における，Design by Contract(DbC)[1] はプログラムにおける仕様をソースコード中に記述することにより，ソフトウェア開発における誤りの箇所を明確にする手法である．Java 言語においては，プログラムのソースコード中に事前条件・事後条件・不変条件などをアサーションという形で記述し，静的解析ツール ESC/Java2[2] を用いて検証することができる．これによりプログラムの保守性や信頼性を向上することができる．

しかし，プログラムのソースコード量の増加にともない既存のソースコードに対して人手によってアサーションを記述するのは困難になるという問題点がある．これを解決するために，テストケースを用いることによりアサーションの自動生成をするツール Daikon [3, 4, 5, 6, 7, 8] があるが，生成されるアサーションの品質はテストケースの品質に依存するテストケース依存問題がある [8]．よって，アサーションの動的生成においては，対象となるプログラムに対して，十分なテストケースを用意する必要がある．

そこで，著者の所属する研究グループでは，アサーション動的生成を目的としたテストケース制約導出についての研究を行っている．本研究においては，研究グループにおいて提案されている，アサーション動的生成を目的としたテストケース制約導出に関する提案 [9] を基に，提案の一部である Java メソッドの実行パス取得部の実装および関連する実験を行った．本研究での実装部分は，対象プログラムのメソッドに対して，実行可能性のあるパスをプログラム依存グラフ (PDG)[10] を利用して導出したのち，ESC/Java2 の反例出力機能を利用するための JML [11] によるアサーションコードを挿入する部分である．これにより，アサーション記述や静的解析に必要な，メソッドの実引数に関する条件式や内部変数に関する条件式を ESC/Java2 の反例出力として取得できる．

また，研究グループにおいて提案されている手法を実装したツール全体での出力の妥当性を検証するため，Java プログラムのソースコードに対してツールを適用する実験を行った結果，有用なテストケース作成に必要な条件を導出できた．さらに，いくつかの Java プログラムのソースコードに対してはテストケースを自動生成でき，Daikon を用いてアサーションを生成した結果，いくつかの正しいアサーションを導出することができた．

以後，2 章では研究背景として，本研究に関連する概念，ツールについて説明を行う．続いて 3 章で実装手法に関して説明を行う．最後に 4 章で実装したツールを適用した結果および Daikon によって自動生成されたアサーションについて評価を行う．

2 研究背景

本章では，本研究に関連する概念，ツールについての説明を行う．

2.1 Design by Contract

Design by Contract(DbC)[1] とは，オブジェクト指向ソフトウェア設計に関する概念の 1 つであり，クラスとそのクラスを利用する側との間で仕様の取り決めを契約とみなし，ソフトウェアの品質，信頼性，再利用性を向上させることを目的とするものである．契約とは，クラスの呼び出し側がそのクラスを利用する時点においてある条件 (事前条件) を満たすことを保証すれば，呼び出されたクラスはある条件 (事後条件) を満たすことを保証することである．この契約により，事前条件を満たさないクラスの利用はクラスの呼び出し側の不具合であり，事後条件を満たさないクラスの動作はクラス側の不具合であると言える．これによって，ソフトウェアにおける不具合箇所の特定を容易にすることができると考えられる．

2.2 アサーション

Java プログラムにおいては，ソースコード中にアサーションと呼ばれる記述を行うことにより，DbC における契約を表明することができる．この表明により，ESC/Java2 [2] などの静的解析器を用いた静的解析において，プログラムの妥当性を検証することができ，開発者の意図しない不具合の混入を防ぐことができる．

主なアサーションとして，メソッドの入口で成立するメソッドの事前条件やメソッドの出口で成立するメソッドの事後条件，オブジェクトの生存中に成立するクラスの不変条件などがある．また，DbC に基づく仕様記述において，事前条件・事後条件はソースコードそのものの情報を端的に表しており，これをソースコードに付加することでソースコードの仕様理解の補助を行うことも可能であると考えられる．

プログラムのアサーションは，Java Modeling Language(JML) [11] や J2SE 1.4 から導入された Java の assert 文 [12] を用いて記述する．以下，図 1 に JML によるアサーションの記述例を示す．図 1 の例はクラス JML_Example のメソッド sum に対して JML によるアサーションを記述した例である．“requires” が事前条件，“ensures” が事後条件，“maintaining” がループインバリアントを表している．また，“\result” はメソッドの返り値を表している．

この例のメソッド sum は，引数として与えられた int 型の配列変数の総和を求めその値を返す．そのための事前条件として，与えられる配列変数が null でないこと，配列変数の長さが 0 より大きいことが必要である．また事後条件として，このメソッドの返り値が与えられた配列変数の要素の総和になるという条件が必要である．そして，ループインバリアントとして，ローカル変数 s は配列変数の要素の部分和になるという条件が成立する．

```
public class JML_Example {
    //@ requires a != null;
    //@ requires a.length >0;
    //@ ensures \result == (\sum int j; 0 <= j && j < a.length; a[j]);
    public int sum(int[] a) {
        int s = 0;
        //@ maintaining s == (\sum int j; j <= 0 && j < i; a[j]);
        for (int i = 0; i < a.length; i++) {
            s = s + a[i]; 10
        }                                                              10
        return s;
    }
}
```

図 1: JML によるアサーションの記述例

```
public class PDG_Example {
    int example(int x, int y) {
        int ret = 0;
        if (x > 0 && y > 0) {
            ret = x + y;
        }
        return ret;
    }
}
```

図 2: メソッド example

2.3 プログラム依存グラフ

プログラム依存グラフ (PDG)[10] とは，プログラム中の命令をノード，ノード間の依存関係を有向辺で表したグラフである．主な依存関係には制御依存とデータ依存がある．制御依存辺は次に実行しうるノード間に構築され，データ依存辺はデータ依存があるノード間に構築される．図 3 に図 2 に示したメソッドに対する PDG を示す．

本研究においては，ESC/Java2 による反例出力を取得するため，メソッドの実行パスのうち事後条件に影響を与えうるパスへの分岐となるノードに，3.2 章で説明するフラグを JML アサーションとして挿入し，出口ノード直前で反例を出力させる．このとき，メソッドの実行パスの分岐点を探索するために PDG を利用している．

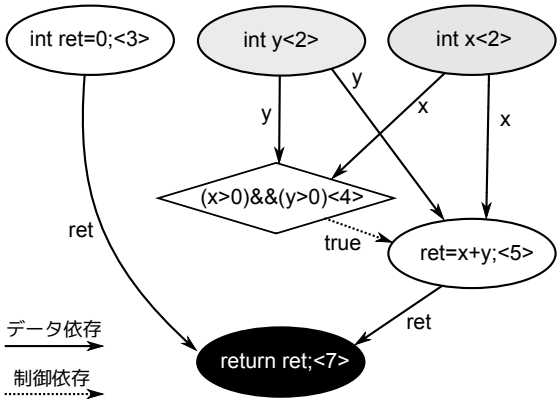


図 3: メソッド example の PDG

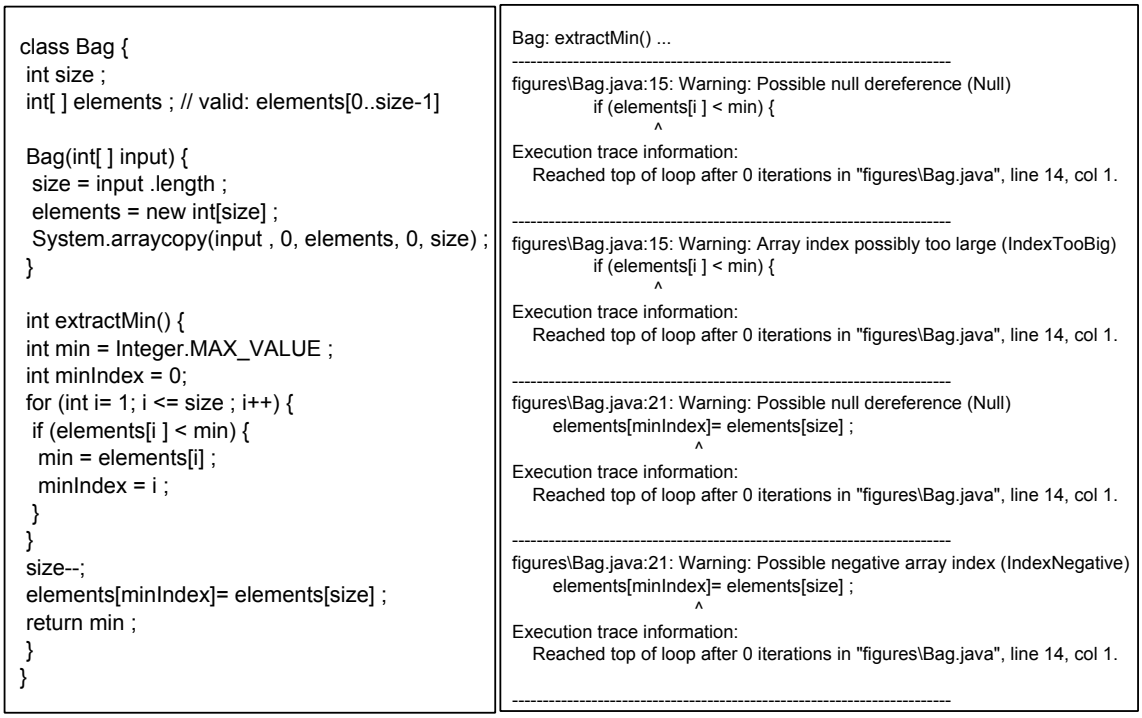
2.4 ESC/Java2

ESC/Java2 は，Java プログラムに対する静的検証器である．Java プログラムを入力とし，JML などで記述されたアサーションと Java プログラムの整合性を検証し結果を出力する．ESC/Java2 は対象プログラムを述語論理に変換し，その充足不能性を判定することによって検証を行う．判定エンジンとして Simplify [13] を使用している．

図 4(b) は ESC/Java2 により図 4(a) に対して静的検査を実行した場合の出力結果である．この例では 3 種類の警告が出力されている．1 つ目は Bag.java の 15 行目と 21 行目における配列変数 elements への参照が Null 参照となる可能性があるという警告，2 つ目は Bag.jav の 15 行目における変数 i の値が境界違反となる可能性があるという警告，最後は Bag.java の 21 行目における変数 size の値が負になる可能性があるという警告である．またオプションで指定することで，警告が出力される際に成立する変数の値などを反例として出力することも可能である．

2.5 Daikon

Daikon [3, 4, 5, 6, 7, 8] はアサーションの動的生成を行うツールである．Daikon は入力としてソースコードとテストケースを与えると，実行時にメソッドの入口および出口にて参照可能な変数の値を観測し，その実行から得られた変数の値と Daikon が持つアサーションパターン [14] とを照合して各プログラムポイントにおけるアサーションを自動生成し，出力する．(図 5) また，このとき Daikon 独自のアサーション推測アルゴリズムを適用することで再現率，適合率が高いアサーションの生成を可能にしている．このツールを用いることにより，手作業でアサーションを記述するよりもアサーション生成にかかる時間的・人的コス



```
class Bag {
 int size ;
 int[ ] elements ; // valid: elements[0..size-1]

 Bag(int[ ] input) {
  size = input .length ;
  elements = new int[size] ;
  System.arraycopy(input , 0, elements, 0, size) ;
 }

 int extractMin() {
 int min = Integer.MAX_VALUE ;
 int minIndex = 0;
 for (int i= 1; i <= size ; i++) {
  if (elements[i ] < min) {
   min = elements[i] ;
   minIndex = i ;
  }
 }
 size--;
 elements[minIndex]= elements[size] ;
 return min ;
 }
}
```

(a) input Java source

```
Bag: extractMin() ...
----------------------------------------------------------------------
figures\Bag.java:15: Warning: Possible null dereference (Null)
            if (elements[i ] < min) {
                ^
Execution trace information:
   Reached top of loop after 0 iterations in "figures\Bag.java", line 14, col 1.

----------------------------------------------------------------------
figures\Bag.java:15: Warning: Array index possibly too large (IndexTooBig)
            if (elements[i ] < min) {
                ^
Execution trace information:
   Reached top of loop after 0 iterations in "figures\Bag.java", line 14, col 1.

----------------------------------------------------------------------
figures\Bag.java:21: Warning: Possible null dereference (Null)
      elements[minIndex]= elements[size] ;
                          ^
Execution trace information:
   Reached top of loop after 0 iterations in "figures\Bag.java", line 14, col 1.

----------------------------------------------------------------------
figures\Bag.java:21: Warning: Possible negative array index (IndexNegative)
      elements[minIndex]= elements[size] ;
                          ^
Execution trace information:
   Reached top of loop after 0 iterations in "figures\Bag.java", line 14, col 1.

----------------------------------------------------------------------
```

(b) output warning

図 4: ESC/Java2 による静的解析

トを軽減することができる．また，実際にプログラムを実行した結果を用いることでプログラマがソースコード記述時には気づかなかったアサーションを生成することもできる．このことはプログラムの保守，デバッグに有効である．また，Daikon は生成したアサーションを JML 形式など様々な形式で出力でき，さらにコメントとして元の Java プログラムに挿入できるなど機能面においても充実している．

2.6 Inveriant Coverage

Daikon など，テストケースを用いたアサーションの動的生成を行う場合，生成されるアサーションの精度が与えるテストケースに依存してしまうというテストケース依存問題がある．

インバリアントカバレッジ (Invariant Coverage)[15, 16] は，アサーションの動的生成ツールに用いるテストケースの品質測定を目的として提案されたカバレッジであり，インバリアントカバレッジの値が高いテストケースを用いることで，アサーションの動的生成の問題であるテストケース依存問題を改善できる．

まず，テストケース依存問題を改善するためには，アサーションの動的生成に用いるテストケースは，対象ソースコードの全ての必要な実行パスを実行する必要がある．

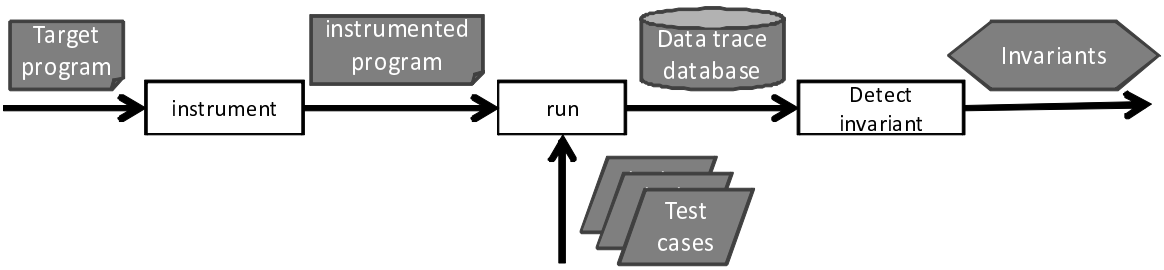


図 5: Daikon の処理

インバリアントカバレッジ [16] は一般的なアサーションの生成に必要な実行パスをテストケースが実際に実行する割合で定義する．文献 [16] では，このような実行パスをプログラムポイントにて参照可能な変数の定義–使用連鎖を含む実行パスに近似している．

[**定義 2.1**] 定義–使用連鎖 (Definition-Use Chain，以降 DUC)

プログラムポイント S にて参照可能な変数 v の DUC は，次のように定義できる．

Definiton-Use Pair(以降，DUP とする) を変数 v，定義部 d，使用部 u の 3 項組で定義し，$v(d,u)$ で表記する．ここで，d，u はプログラム記述中の位置を表す．DUC は DUP の (有限) 系列として定義でき，以下のように表記する．

$$v(x_1,x_0) \Leftarrow v(x_2,x_1) \ldots \Leftarrow v(x_n,x_{n-1})\ (n \geq 1)$$

直前の d が次の DUP の u である．

この系列において，位置 d，u の複数回の出現は許すが，同一 DUP の複数回の出現は許さない．系列の最後において d，u が同一の位置のとき，この DUP の繰り返しを許し，その場合，以下のように表記する．なお,「||」は直前の DUP の繰り返しを意味する．

$$v(x_1,x_0) \Leftarrow v(x_2,x_1) \ldots \Leftarrow v(x_n,x_{n-1}) \Leftarrow v(x_n,x_n) \mid\mid (n \geq 1)$$

[**定義 2.2**] インバリアントカバレッジ

全プログラムポイントにて参照可能な全変数の DUC の総数を DUC_{all}，テストケースによって実行された DUC の数を $DUC_{executed}$ とする．このとき，あるテストケースにおけるインバリアントカバレッジ C_{Inv} の値は式 (1) で定義される．

$$C_{Inv} = DUC_{executed}/DUC_{all} \tag{1}$$

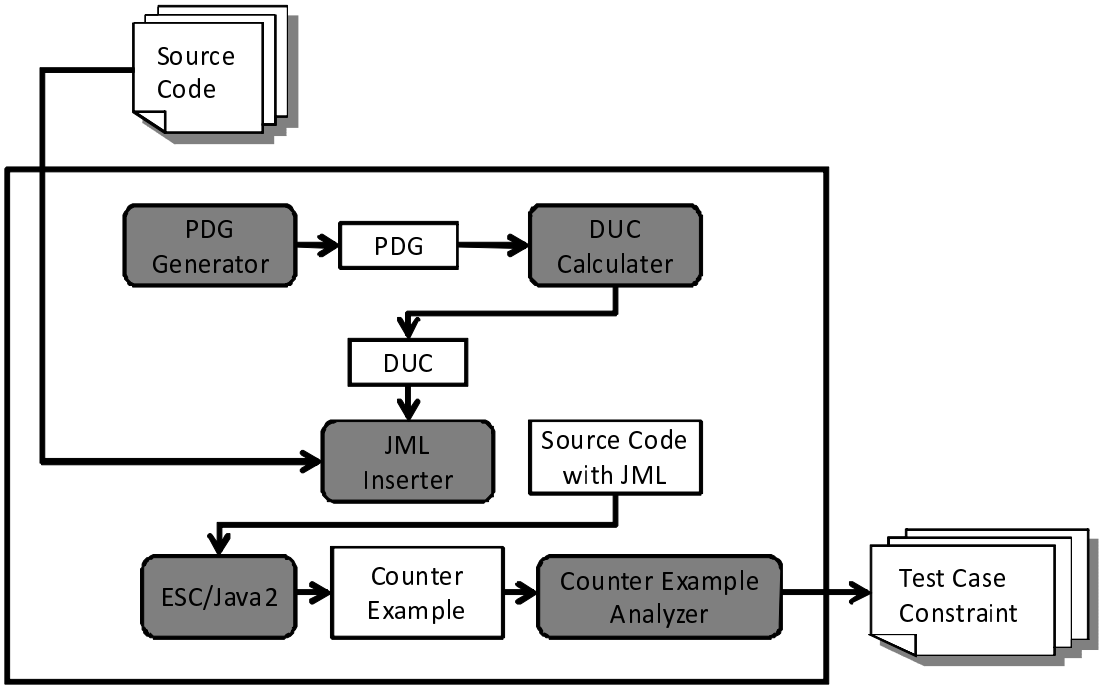


図 6: 研究グループの提案手法

3 研究グループで提案された手法

本章では，本研究において実装した手法およびツールについて説明を行う．

3.1 概要

本研究で実装した手法は，著者の所属する研究グループにおいて提案されている手法 [9] で，Daikon などを用いて動的にアサーションを生成する際に必要となるテストケース制約を導出する手法である．この手法は，次の流れで実行する．(図 6)

1. 対象メソッドの PDG を生成する
2. PDG より，DUC を求め反例出力用の JML の挿入位置を決定し挿入する
3. ESC/Java2 を実行し反例を取得する
4. 得られた反例出力を解析し，テストケース制約導出に必要な情報を抽出する
5. 抽出した情報からテストケース制約および Daikon 用のテストケースの雛形を出力する

本研究ではこのうち手順 2 と手順 4 の一部を実装した．なお，本ツールにおいて，対象メソッドの PDG 作成部は，MASU [17] を利用している．

3.2 実行パスの検出

実行パスの検出は，対象メソッドの PDG を利用している．PDG は制御依存辺とデータ依存辺によってノード間の依存関係を表している．ここで，対象メソッドの事後条件となる出口ノードの値は，入口ノードからの制御依存辺とデータ依存辺の連鎖によって決定される．今回は出口ノードの値に影響のあるノードを実行するパスが必要であるため，以下に示すアルゴリズムによって反例出力用の JML を挿入するパスを決定している．

1. 対象メソッドの PDG を取得する
2. メソッドの各出口ノードに対して，次の処理を行う
   (a) フラグを設定すべき箇所をフラグ挿入位置決定アルゴリズムにより決定し，挿入する
   (b) 出口ノード直前に各フラグ箇所を通過有無の場合の数に応じた，反例出力用の JML アサート文を挿入する．
   (c) フラグに利用したアサーションの変数宣言をメソッド先頭に挿入する

フラグ挿入位置決定アルゴリズムは以下の通り．

1. 注目ノードがデータ依存する定義ノードがあれば 1 つ取得し次の動作を行う
   (a) 取得した定義ノードが処理済みリストになければ追加し次の動作を行う
      i. 定義ノードが制御依存する制御依存ノードを取得
      ii. 制御依存ノードが処理済みリストになければ追加しフラグを挿入
      iii. さらに制御依存しているノードが取得できなくなるまで制御依存ノードを取得しフラグを挿入
   (b) この定義ノードに対してこのアルゴリズムを再帰的に適用
2. すべての定義ノードを処理したら終了

例として Java ソースコード Path_Example.java(図 8) の PDG(図 7) を挙げる．このアルゴリズムは，事後条件となる値が変化しないノード “System.out.println(”input x is 0.”);< 5 >” を通るパスへはデータ依存や制御依存するノードが存在しないため処理対象としない．これは，ノード “System.out.println(”input x is 0.”);< 5 >” は事後条件に影響のあるノードでないため，出口ノードより探索する DUC 上にノードが存在しないからである．これにより，対象となる出口ノードの値に影響しない実行パスを検出することなくテストケース制約に必要なパスのみを算出でき，不要な ESC/Java2 の反例出力を出力することを抑制してい

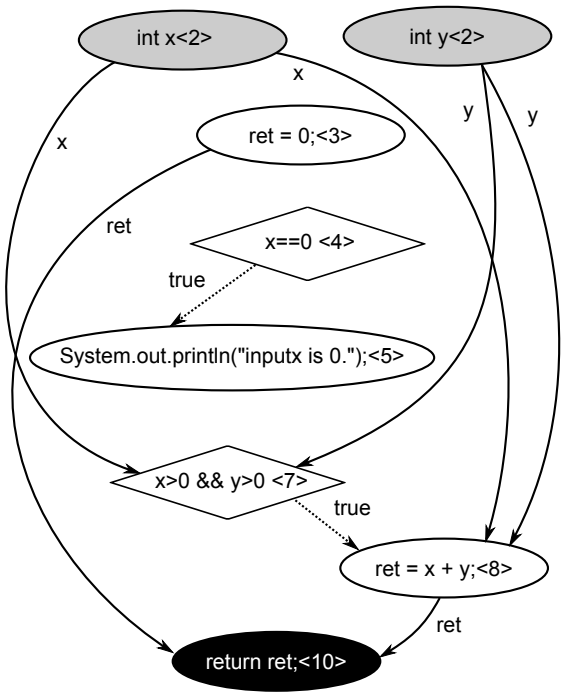


図 7: 図 8 に示した Path_Example.java の PDG

る．また，すべての DUC の系列上のノードに対して制御依存するノードが存在するか調べ，アサーション検出用 JML フラグを挿入しているため，データ依存により出口ノードの返り値が変化する命令を実行する条件となる制御依存ノードへアサーション検出用 JML フラグを挿入できる．これにより出口ノードでの値の変化する可能性のあるパスを全て検出することが可能である．

元の Java ソースコードに挿入する反例検出用 JML アサーションは以下に示す 3 種類である．

1. ghost 文：“ghost boolean $$f1 = false;” のように，フラグ用ブーリアン変数を宣言する．フラグ用変数は条件分岐 1 つに対して 1 つ用意する．ghost ステートメントの挿入位置はメソッド宣言の開始直後である．

2. set 文：“set $$f1 = true;” のようにフラグ用変数に値を代入する．set ステートメントの挿入位置は条件分岐のブロックの入口直後である．一つの条件分岐に対して，一つのフラグ用変数の値を変化させる．

3. assert 文：“assert !$$f1;” のように ESC/Java2 に警告，反例を出力させるアサーションを記述する．アサーションは各フラグ用変数の真偽の組合せを網羅するため，デー

```
public class Path_Example {
    int example99(int x, int y) {
        int ret = 0;
        if (x == 0) {
            System.out.println("input x is 0.");
        }
        if (x > 0 && y > 0) {
            ret = x + y;
        }
        return ret;
    }
}
```

図 8: Path_Example.java

タ依存の連鎖に含まれるノードが制御依存しているノードが n 個とすると，各条件分岐を通る組合せを考え assert 文の数は 2^n 個になる．挿入位置は対象メソッドの出口ノードの直前である．

これらのパス検出用の JML コードを挿入した例を図 9 に示す．図 9(a) が元のソースコード，図 9(b) がアサーション挿入後のソースコードである．

3.3 ESC/Java2 反例出力の解析

ESC/Java2 が出力する反例に含まれる情報には以下のようなものがある．

1. 変数の型
2. 変数の値
3. 変数間の関係
4. オブジェクトのメモリ上への配置
5. デッドロックの可能性

著者の所属する研究グループにおいて提案されている手法では，これらの，1. 変数の型に関するもの，2. 変数の値に関するもの，3. 変数間の関係に関するもの，をテストケース生成に必要な情報とし，解析の対象とする．4，5 の情報は不要な情報とし，本手法では扱わない．ESC/Java2 は上記の情報をまとめて出力するため，各情報に分類する必要がある．4，5 の情報に関する反例の式には特定の予約語が含まれているため，文字列マッチングを行う

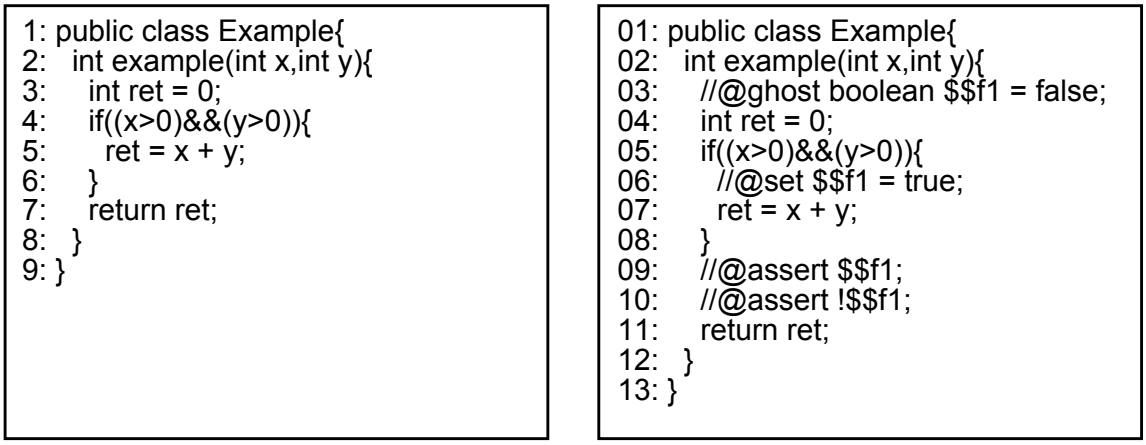


図 9: パス検出用 JML コードの挿入例

ことで 4，5 の式を取り除くことができる．1，2，3 の情報に関しては著者の所属する研究グループにおいて，ESC/Java2 が出力する反例の構文解析器を JavaCC [18, 19, 20] を用いて作成した．ESC/Java2 の作者によると ESC/Java2 が出力する反例の厳密な文法定義が存在しないため，研究グループで独自に文法定義を行った．今回対象とする範囲の文法定義を表 1,2,3,4 に示す．

解析された反例出力は，テストケースの自動生成や人手によるテストケース作成時に利用しやすいよう，それぞれメソッドの各 return 文ごとに分類しファイルに出力する．(図 10)

例として，図 10 に示したテストケース制約において@15 から始まる 8 行は元のソースコードの 15 行目に存在する return 文に到達するためのテストケース制約を示している．このうち変数の関係については，text.length() < pattern.length() のように，引数 text の長さが引数 pattern の長さより短いという条件があることがわかる．また，変数の型についての情報は “\typeof(text)==\type(java.lang.String)” や “\typeof(pattern.charAt(j)) <: \type(char)” のように出力され，前者の場合，引数 text の型は String 型であり，後者は引数 pattern の j 番目の文字を返す pattern.charAt(j) は char 型のサブタイプの値を返すことがわかる．

研究グループで提案されている手法においては，現在非プリミティブ型や String 型以外の参照型など，乱数を用いてテストケースを作成することが難しい型の引数を持つメソッドに対しては，自動的にテストケースを生成することができない．このような場合は，出力されたテストケース制約をもとに同時に出力されたテストケースの雛形に対し，テストケース制約を人手によって記述することによりテストケースを作成する．

```
**********************************
@15
text.length() < pattern.length()
\typeof(text.length()) <: \type(int)
\typeof(text) == \type(java.lang.String)
\typeof(pattern.length()) <: \type(int)
\typeof(pattern) == \type(java.lang.String)
(BM.buildLastFunction(pattern) != null)
(text != null)
(pattern != null)
**********************************
**********************************
@43
text.charAt(i) == pattern.charAt(j)
i == 0
pattern.length() == 1
0 < text.length()
\typeof(text.length()) <: \type(int)
\typeof(text) == \type(java.lang.String)
\typeof(pattern) == \type(java.lang.String)
\typeof(text.charAt(i)) <: \type(char)
(BM.buildLastFunction(pattern) != null)
(text != null)
(pattern != null)
**********************************
**********************************
@74
pattern.length() <= BM.buildLastFunction(pattern)[text.charAt(i)] + 1
text.charAt(i) < BM.buildLastFunction(pattern).length
0 < pattern.length()
\typeof(text) == \type(java.lang.String)
\typeof(pattern.length()) <: \type(int)
\typeof(pattern) == \type(java.lang.String)
\typeof(BM.buildLastFunction(pattern)[text.charAt(i)]) <: \type(int)
\typeof(pattern.charAt(j)) <: \type(char)
\typeof(text.charAt(i)) <: \type(char)
(BM.buildLastFunction(pattern) != null)
(text.charAt(i) != pattern.charAt(j))
(text != null)
(pattern != null)
**********************************
```

図 10: BM.java のメソッド BMmatch に対するテストケース制約

表 1: 文法定義 1

counterexample	::=	expression_list
expression_list	::=	{ expression }
expression	::=	logical_and_expression { "||" logical_and_expression }
logical_and_expression	::=	inclusive_or_expression { "&&" inclusive_or_expression }
inclusive_or_expression	::=	exclusive_or_expression { "|" exclusive_or_expression }
exclusive_or_expression	::=	and_expression { "^" and_expression }
and_expression	::=	equality_expression { "&" equality_expression }
equality_expression	::=	relational_expression { "==" relational_expression }
relational_expression	::=	shift_expression { relational_op shift_expression }
shift_expression	::=	addtive_expression
addtive_expression	::=	mult_expression { additive_op mult_expression }
mult_expression	::=	unary_expression { mult_op unary_expression }
unary_expression	::=	sign unary_expression
		| unary_expression_not_plus_minus
unary_expression_not_plus_minus	::=	("!" | "~") unary_expression
	::=	| postfix_expression
postfix_expression	::=	primary_expression { primary_suffix }
primary_suffix	::=	"(" expression_list ")"
	::=	| "[" expression "]"
primary_expression	::=	built_in_type()
		| ref_name
		| "null"
		| "this"
		| constant
		| "(" expression ")"
		| esc_primary_expressrion
built_in_type	::=	"T_boolean" | "T_byte" | "T_char" | "T_float" | "T_int"
		| "T_long" | "T_short" | "T_void"
		| "T_boolean[]" | "T_byte[]" | "T_char[]" | "T_float[]"
		| | "T_int[]" | "T_long[]" | "T_short[]"

表 2: 文法定義 2

esc_primary_expression	::=	typeof_expression | isEmpt_expression | arrayLength_expression
		| length_expression
		| size_expression | matches_expression | max_expression
		| | min_expression
		| isNewArray_expression | elemtype_expression | equals_expression
		| cast_expression | charAt_expression | entrySet_expression
		| HashCode_expression | hasMapObject_expression | mapsObject
		| hasMap_expression | intern_expression
		| interned_expression | isDigi_expressiont | listGet_expression
		| get_expression | getProperty_expression | getTime_expression
		| keySet_expression
		| contains_expression | containsAll_expression | containsKey_expression
		| containsValue_expression | containsObject_expression
		| indexOf_expression | lastIndexOf_expression | subList_expression
		| toArray_expression
		| toUpperCase_expression | values_expression
typeof_expression	::=	"\typeof" "(" expression ")"
isEmpty_expression	::=	"\isEmpty" "(" expression ")"
arrayLength_expression	::=	"\arrayLength" "(" expression ")"
size_expression	::=	"\size" "(" expression ")"
matches_expression	::=	"\matches" "(" expression expression ")"
max_expression	::=	"\max "(" expression expression ")"
min_expression	::=	"\min "(" expression expression ")"
isNewArray_expression	::=	"\isNewArray" "(" expression ")"
elemtype_expression	::=	"\elemtype" "(" expression ")"
equals_expression	::=	"\equals" "(" expression expression ")"
cast_expression	::=	"\cast" "(" expession expression ")"
charAt_expression	::=	"\charAt" "(" state expession expression ")"
entrySet_expression	::=	"\entrySet" "(" expression ")"
hasMap_expression	::=	"\hasMap" "(" expession expression ")"
HashCode_expression	::=	"\HashCode" "(" expession ")"

表 3: 文法定義 3

hasMapObject_expression	::-	”\hasMapObject” ”(” expression expression ”)”
intern_expression	::=	”\intern” ”(” expession expression ”)”
interned	::=	”\interned” ”(” expression ”)”
isDigit_expression	::=	”\isDigit” ”(” expression ”)”
listGet_expression	::=	”\listGet” ”(” expession expression ”)”
get_expression	::=	”\get” ”(” expession expression ”)”
getProerty_expression	::=	”\getProperty” ”(” expession expression ”)”
keySet_expression	::=	”\keySet_expression” ”(” expression ”)”
contains_expression	::=	”\contains” ”(” [state] expession expression ”)”
containsAll_expression	::=	”\containsAll” ”(” expession expression ”)”
containsKey_expression	::=	”\containsKey” ”(” expession expression ”)”
containsValue_expression	::=	”\containsValue” ”(” expession expression ”)”
containsObject	::=	”\containsObject” ”(” expession expression ”)”
indexOf_expression	::=	”\indexOf” ”(” expession expression ”)”
subList_expression	::=	”\subList” ”(” expession expression ”)”
toArray_expression	::=	”\toArray” ”(” expession expression ”)”
toUpperCase_expression	::=	”\toUpperCase” ”(” expession expression ”)”
values_expression	::=	”\value_expression” ”(” expession expression ”)”
ref_name	::=	this ”.” ”(” ident location ”)” | regexAt
		| ident { ”[” [expression] ”]” } location | ”RES” [location]
location	::=	LOC_LITERAL
		| RES_LOC_LITERAL LOC_LITERAL
		| LOC_WITH_LOOP_LITERAL INT_LITERAL LOC_LITERAL
ident	::=	IDENTIFER { ”.” IDENTIFIER } [LOOPOLD]
constant	::=	number { number } | ” {(”_” | ”$” | alphabet | number)} ”
		| ”false” |”true”

表 4: 文法定義 4

relational_op	::=	”<” | ”>” | ”<=” | ”>=” | ”<:”
additive_op	::=	”+” | ”-”
mult_op	::=	”*” | ”/” | ”%”
sign	::=	”+” | ”-”
LOC_LITERAL	::=	”:” number { number } ”.” number { number }
RES_LOC_LITERAL	::=	”-” number { number } ”.” number { number }
LOC_WITH_LOOP_LITERAL	::=	”-” number { number } ”.” number { number } ”#”
IDNTIFER	::=	(”_” | ”$” \| alphabet) { (”_” \| ”$” | alphabet | number) }
number	::=	”0” | ”1” | ... ”9”
alphabet	::=	”a” | ”b” | ... ”z” | ”A” | ”B” | ... | ”Z”

4 評価

本章では評価実験の結果および考察について述べる．本研究では評価実験として，手法を実装したツールを 4.3 節に示す Java プログラムに対して適用した．本研究は，研究グループにおいて提案されている手法を実装したツールの一部であるため，本論文での評価実験は本ツール全体に対する評価実験として行う．本ツールを実行時間および出力結果の妥当性の点で評価を行う．

本研究においてテストケース制約が以下の 2 つの条件を満たす場合，出力が妥当であると定義する．

1. 本ツールが出力したテストケース制約が対象ソースコードに対して矛盾が無い
2. 本ツールの出力したテストケース制約がテストケース作成時に必要な条件である

また，Daikon が生成したアサーションが妥当であるとは人手による確認を行った際に対象ソースコードに対して矛盾が無いことと定義する．

4.1 実験環境

本ツールを以下の実験環境にて実行し，実行時間を測定，出力結果の分析を行った．

- 計算機:HPZ400
- OS:Windows Vista Business

- CPU:Intel Xeon W3520 2.67GHz × 2
- Memory:6.00GB
- Java:JDK1.4.2_19(ESC/Java2 実行用)，JDK1.6_17(本ツール，Daikon 実行用)
- ESC/Java2:version2.0.5
- Daikon:4.6.3

4.2 本ツールの制限

本ツールの適用可能なメソッドは，ESC/Java2 が Java1.4 までしか対応していないので，対象ソースコードは Java1.4 である必要がある．また，実装の都合上メソッド内での分岐は以下の要素に限定しており，他の分岐条件については同様の対応している構文に書き換える必要がある．

1. if - then - else 構文
2. for ループ構文
3. while ループ構文

4.3 評価対象

実験対象は，本研究グループで以前提案されていた手法 [21, 22] および文献 [23] において使用された，以下の 4 つのテストデータである．

1. BM.java:Boyer-Mooer 法による文字列マッチングを行う
2. calc.java:二次方程式の解を求める
3. MethodHash.java:2 つの byte 型配列に対しハッシュ値の比較を行う
4. TestArrays.java:2 つの int 型配列の比較を行う

4.4 評価実験の手順

以下の手順で評価を行った．

1. 対象ソースコードに本ツールを適用しテストケースに必要な情報及びテストケースを取得する．同時にテストケースを得るまでの実行時間を計測した．

2. テストケース制約を有用なもの，冗長なもの，誤っているもの，の 3 つに分類し要素数を計測した．

3. テストケースを自動生成できた場合，テストケースを Daikon により実行し，出力されたアサーションをソースコードのアルゴリズムと比較し，ソースコードのアサーションとして妥当なものと妥当でないものに分類し，要素数を計測した．

4.5 実行時間の評価

ツールの実行時間について評価を行う．本研究においては本ツールの実行時間が実用的なものかどうかを評価する．表 5 に各対象ソースコードに対する本ツールの実行時間を示す．

各対象において，全実行時間が 10 秒程度の実行時間に収まっている．実行環境による多少の変動は考慮しても十分実用的な時間で実行ができると考えられる．対象ソフトウェアの規模に関して今回は実験手法の都合により実験を行なっていないが，比較的大規模なソフトウェアに対しても実行することは可能であると考えている．

本ツールはメソッド単位で処理を行うため，メソッドの大きさが実行時間に影響を与えると考えられる．メソッドの大きさは現実的に数 100 行程度であると考えており，それを超えるような行数のメソッドは考えにくい．

本ツールの処理のうち，反例検出用 JML 挿入処理にかかる処理時間は指数的に増大せず，数 100 行程度のメソッドに対しても，実用的な時間で実行が可能であると考えている．また，ESC/Java2 の処理速度について，文献 [2] によると 500-1000 行程度の規模のメソッドに対して 5 分以内に処理が完了しており，一般的な多くのメソッドに対して ESC/Java2 は適用可能であると考えられる．反例解析やテストケース制約・テストケース出力は文字列操作が主な処理になるため，処理時間が大幅に増大する可能性は少ないと考えられる．

実際にどの程度の時間で実行が終了するかどうかについては，今後調査していく必要があると考えている．

表 5: ツールの実行時間

実験対象	実行時間 (秒)				
	条件挿入	ESC/Java2	テストケース生成	解析出力	全実行時間
BM	1.56	3.52	0.0022	0.0037	7.56
calc	0.49	2.26	0.002	0.0037	4.73
MethodHash	0.53	2.06	0.0036	0.0052	4.67
TestArrays	0.47	2.01	0.0032	0.0044	4.55

4.6 テストケース制約出力の評価

本ツールを各対象ソースコードに適用し，出力されたテストケース制約の有用性をソースコードを参照することによって調べ，それぞれ以下の基準で分類を行った．

- 有用な制約: テストケース制約として適しており，出力されたそのままの状態でテストケース作成に使用できる制約
- 冗長な制約: 他のテストケース制約に包含されていて不要なテストケース制約であるが，論理的に誤っていない制約
- 不適切な制約: 対象メソッドのテストケース制約として論理的に誤っており，テストケース制約として不適切な制約．

実験対象の各プログラムについて出力された条件を分類した結果を表 6 に示す．

表 6: 出力テストケース制約の精度

実験対象	有用な制約	冗長な制約	不適切な制約	全出力数
BM.java	25	7	0	32
calc.java	10	2	0	12
MethodHash.java	19	3	0	22
TestArray.java	34	10	0	44

出力されたテストケース制約について，有用な制約が多く出力された．出力されたテストケース制約を利用してテストケースを作成できると考えられる．冗長な制約については，特に引数の型についての制約が本実験対象から出力された冗長な制約全 22 個中 12 個確認できた．例えば，TestArray.java のメソッド equals において，変数型の制約として “\typeof(b) == \type(int[])” と “\typeof(b) <: \type(int[])” が同時に出力された．

これら変数型の情報はテストケースを自動で生成することを考えたときに，引数の型を決定する際に利用する時点でいずれか片方あれば十分であるが，型に関する情報がどのような条件の下で得られるかは現時点では判明していないため，単純に字句解析のみで消去することは難しいと考えている．

また，今回の実験対象においては不適切な制約は出力されなかったが，ESC/Java2 の反例出力において明確な文法定義がなされていないこともあり，今後より多くの対象に対して本ツールを適用し，出力テストケース制約を評価する必要があると考えている．

4.7 Daikon によるアサーション生成結果の評価

本ツールを実験対象のソースコードに適用し出力されたテストケースについて，Daikon を用いてアサーションを生成した結果について考察する．表 7，8 は，出力されたアサーションについて，元のソースコードのアルゴリズムと比較し，矛盾が無いかどうかを判定しまとめたものである．以後，出力されたアサーションが元のソースコードのアルゴリズムと比較し矛盾が無い場合，アサーションが適合するという．

事前条件　事前条件について考察する．BM.java のメソッド BMmatch に対する事前条件はすべて適合した．このとき出力された事前条件は，“arg0 != null” といった，入力となる String 型の引数が null でないという条件であった．BM.java のメソッド BMmatch では，引数が null でないことを仮定しており，今回出力されたアサーションはアルゴリズムに適合した有用なアサーションであった．

calc.java のメソッドや TestArray.java のメソッドにおいて，出力された事前条件のすべてが適合しなかった．このとき出力された事前条件は “$arg0 >= 0$” や “$arg0[]elements >= 0$” のように，与える実引数や実引数配列の要素が非負整数の範囲であるというアサーションであった．これは，自動生成するテストケースプログラムが発生している int 型の乱数の範囲を非負整数に限定しているため発生すると考えられる．

このようにテストケースとして与える乱数の範囲は出力されるアサーションに大きく影響を与えると考えられるため，今後の課題として，テストケースプログラムの与える乱数の範囲を選択する方法について考察する必要がある．

MethodHash のメソッドに対して出力された事前条件は 3 個中 2 個が適合した．適合した事前条件は BM.java と同様 “arg0 != null” といった，入力となる byte 型の配列が null でないという条件であった．MethodHash.java のメソッド compareHashes も，引数が null でないことを仮定しており，今回出力されたアサーションはアルゴリズムに適合した有用なアサーションであった．

適合しなかった事前条件は，“size(arg0[]) != size(arg1[])-1” のように与える引数配列の長さの関係に関する条件式であった．そのため，対象となるパスに対してそれぞれテストデータを 1000 組入力するようテストケースプログラムを変更したところ，この条件式は出力されなかった．したがって，Daikon に対して入力するテストケース数を工夫することによって，より適切な事前条件を得られる可能性があると考えているが，今後の課題として，適切なテストケース数に関して調査を行いたいと考えている．

事後条件　事後条件について考察する．事後条件については，対象となるメソッドによって適合率が大きく異なっている．特に適合率が低かった calc.java と TestArray.java について，

それぞれテストデータを 1000 組入力するようテストケースプログラムを変更した．テスト制約及び乱数の生成方法については修正は加えていない．この時の結果を表 9 に示す．

TestArray.java についてはテストケースの数が 20 組・1000 組いずれの場合もアサーションの数・内容に変化はなかった．calc.java については出力されるアサーションの数が減少したが，適合するアサーションの数・内容に変化はなかった．

これは，Daikon の出力可能なアサーションの範囲と，対象となるメソッドに適合するアサーションの範囲との共通部分が少なく，Daikon によって適合するアサーションが出力できないためと考えられる．Daikon には二次式の変数関係式を生成するアサーションテンプレート [14] が標準では存在しないため，二次方程式の解を求めるプログラム calc.java のアサーション生成は難しいと考えられる．

表 7: Daikon の生成する事前条件の適合率

実験対象	事前条件総数 (個)	適合事前条件 (個)	適合率
BM.java	2	2	1.00
calc.java	3	0	0.00
MethodHash.java	3	2	0.67
TestArray.java	2	0	0.00

表 8: Daikon の生成する事後条件の適合率 (テストケース各出口ノードにつき 20 組入力)

実験対象	事後条件総数 (個)	適合事後条件 (個)	適合率
BM.java	3	2	0.67
calc.java	11	2	0.18
MethodHash.java	3	2	0.67
TestArray.java	11	4	0.36

表 9: Daikon の生成する事後条件の適合率 (テストケース各出口ノードにつき 1000 組入力)

実験対象	事後条件総数 (個)	適合事後条件 (個)	適合率
calc.java	9	2	0.22
TestArray.java	11	4	0.36

5　あとがき

本研究では，アサーション動的生成に必要なテストケースを作成する手法に基づき Java メソッド実行パス取得部の実装とツールの出力結果の解析を行った．対象 Java メソッドに対して PDG を用いてプログラムの実行パスを求めることにより，アサーション動的生成において有用なテストケース制約に必要な情報を ESC/Java2 に出力させることができることを実験を行い確認した．

また，出力されたテストケース制約をもとにテストケースを作成し Daikon にアサーションを動的生成させたところ，いくつか正しいアサーションを出力させることができたが，一般的に成立しないアサーションも含まれていることを確認した．

今後の課題としては，より多くの対象プログラムに本ツールを適用し，様々なクラスに対して ESC/Java2 が有用なテストケース制約に必要な情報を出力できるかどうかを確認する必要がある．さらに，メソッドに入力するテストケースの生成方法について，特に複雑なクラスのオブジェクトをテストケース入力として生成する方法について検討していく必要がある．

謝辞

本研究を行うにあたり，日頃より理解あるご指導を賜り，常に励まして頂きました 楠本 真二 教授に心から感謝申し上げます.

本研究の全過程を通し，終始熱心かつ丁寧なご指導を頂きました 岡野 浩三 准教授に深く感謝申し上げます.

本研究に関して，的確なご助言ご指導を頂きました 肥後 芳樹 助教に深く感謝申し上げます.

本研究に多大なるご助言ご指導を頂きました 柿元 健 特任助教に深く感謝申し上げます.

本研究を手伝って頂いた大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程 2 年 宮本 敬三 氏に深く感謝致します.

その他の楠本研究室の皆様のご協力に心より感謝致します.

最後に，本研究に至るまでに，講義，演習，実験等でお世話になりました情報科学科の諸先生方にこの場を借りて心から御礼申し上げます.

参考文献

[1] B. Meyer:“Applying Design by Contract,” in Computer(IEEE), Vol. 25, No. 10, pp. 40-51, 1987.

[2] C. Flabagan, K. Rustan, M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata:“Extended static checking for Java,” Proc. of the ACM SIGPLAN 2002, pp. 234-245, 2002.

[3] M. D. Ernst, J. H. Perkins, P. J. Guo, S. Mccamant, C. Pacheco, M. S. Tschantz, and C. Xiao:“The Daikon system for dynamic detection of likely invariants,” Science of Computer Programming, Vol. 69, No. 1-3, pp. 35-45, 2007.

[4] M. D. Ernst:“Dynamically Discovering Likely Program Invariants,” PhD thesis, UniversityofWashington Department of Computer Science and Engineering, SeatleWashington, August 2000.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin:“Dynamically Discovering Likely Program Invariants to Support Program Evolution,” IEEE Trans. on Software Engineering, Vol. 27, No. 2, pp. 99-123, February 2001.

[6] J. W. Nimmer and M. D. Ernst:“Automatic Generation of Program Specifications,” in Proc. of the 2002 Int. Symp. on Software Testing and analysis (ISSTA), pp. 232-242, 2002.

[7] J. W. Nimmer and M. D. Ernst:“Invariant Inference for Static Checking: An Empirical Evaulation,” in Proc. of SIGSOFT Symp. on Foundations of Software Engineering 2002, FSE 2002, pp. 11-20, 2002.

[8] J. W. Nimmer and M. D. Ernst:“Static Verification of Dynamically Detected Program Invariants:Integrating Daikon and ESC/Java,” in Proc. of First Workshop on Runtime Verification, RV 2001, pp. 152-171, 2001.

[9] 宮本敬三:“アサーション動的生成を目的としたテストケース制約の ESC/Java2 を利用した導出”, 大阪大学大学院情報科学研究科コンピュータサイエンス専攻 修士学位論文, 2010.

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren:“The program dependence graph and its use in optimization,” ACM Trasactions on Programming Languages and Systems, Vol. 9, No. 3,pp. 319-349, 1983.

[11] G. T. Leavens, A. L. Baker, and C. Ruby:“JML:A Notion for Detailed Design,” in BehavioralSpecifications of Businesses and Systrems, pp. 175-188, 1999.

[12] “Programming With Assertions”
http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html

[13] D. Detlefs:“Simplify : A Theorem Prover for Program Checking,” JACM, Vol. 52, No. 3, pp. 365-473, 2005.

[14] “The Daikon Invariant Detector User Manual,”
http://groups.csail.mit.edu/pag/daikon/download/doc/daikon.html#Invariant-list

[15] N. Gupta:“Generating Test Data for Dynamically Discovering Likely Program Invariants,” in Proc. of ICSE 2003 Workshop on Dynamic Analysis, WODA 2003, pp. 21-24, 2003.

[16] N. Gupta and Z. V. Heidepriem:“A New Structural Coverage Criterion for Dynamic Detection of Program Invariants,” in Proc. of Int. Conf. on Automated Software Engineering, ASE2003, pp. 49-58, 2003.

[17] 三宅達也, 肥後芳樹, 楠本真二, 井上克郎:“多言語対応メトリクス計測プラグイン開発基盤 MASU の開発”, 電子情報通信学会論文誌 D, Vol. J92-D, No. 9, pp. 1518-1531, 2009.

[18] “JavaCC(JavaCompilerCompiler) Java Parser Generator,”
https://javacc.dev.java.net

[19] 早乙女健治:“JavaCC コンパイラ・コンパイラ”, テクノプレス, 2003.

[20] Kodaganallur and Viswanathan:“Incorporating Language Processing into Java Applications:A JavaCC Tutorial,” IEEE Softw., Vol. 21, No. 4, pp. 70-77, 2004.

[21] 堀直哉, 岡野浩三, 楠本真二:“モデル検査技術を用いたインバリアント被覆テストケースの自動生成による Daikon 出力の改善”, ソフトウェア工学の基礎ワークショップ FOSE2008, ソフトウェア工学の基礎 XV, pp. 41-50, 2008.

[22] 堀直哉:“コードカバレッジに基づいたテストケースのモデル検査技術を用いた自動生成とそれによるアサーションの動的生成”, 大阪大学大学院情報科学研究科コンピュータサイエンス専攻修士学位論文, 2009.

[23] 宮本敬三, 堀直哉, 岡野浩三, 楠本真二, 西本哲:“Java に対するループインバリアントを含む Daikon 生成アサーションの妥当性評価”, 電子情報通信学会論文誌 D, Vol. J91-D, No. 11, pp. 2721-2723, 2008.