修士学位論文

題目

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

指導教員
楠本 真二 教授

報告者
宮本 敬三

平成 22 年 2 月 8 日

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

平成 21 年度 修士学位論文

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

宮本 敬三

内容梗概

Design by Contract に基づいてプログラムのアサーションを記述することで，ソースコードの仕様理解の補助やプログラム検証が行える．しかしソースコードサイズの増加にともない，既存のソースコードに対する手動によるアサーションの作成は困難になる．そのため，アサーションの自動生成手法が注目されている．アサーションの自動生成手法の一つであるアサーションの動的生成手法は，テストケースを用いて引数・フィールド変数の値を変化させながら対象メソッドを実行し，そこで得た情報を解析するため，比較的少ないコストでアサーションの生成が可能である．しかし，生成アサーションの精度がテストケースに依存してしまう問題がある．

この問題を改善するため，インバリアントカバレッジが提案されている．インバリアントカバレッジの値は，テストケースが対象メソッドの return 文とデータ依存関係にある命令文の組み合わせをどれだけ実行するかで求められる．著者が所属する研究グループでは，モデル検査技術を利用しインバリアントカバレッジの値が高いテストケースを自動生成する手法を提案してきた．

本研究では，既存手法において問題となっていた適用可能なクラスの制限を改善する手法を提案する．既存手法では対象ソースコードに対して記号実行を行うことでテストケース制約を導出していた．しかし，プリミティブでないデータ型に対して記号実行を行うことは難しいため，適用可能なクラスが狭くなっていた．本手法では静的解析器 ESC/Java2 が対象プログラムに対して出力する反例を解析することで，テストケース制約を導出する．ESC/Java2 が出力する反例中から変数間に成立する関係式，変数の型情報の抽出，整理を行う．これらは ESC/Java2 内の定理証明器により簡単化されているため記号実行と同様の結果を得ることができる．提案手法を複数の Java プログラムに対して適用し，評価実験を行った．その結果，適用可能なクラスが拡張されたこと，有用なテストケース制約が導出できたことを確認した．

主な用語

Design by Contract, アサーション, コードカバレッジ, ESC/Java2

目 次

図 目 次

表 目 次

1 はじめに

Design by Contract[1] に基づいてプログラムのアサーションを記述することで，ソースコードの仕様理解の補助やプログラム検証が行える．しかし，ソースコードサイズの増加にともない既存のソースコードに対して手動によるアサーション生成は困難になる．そのため，アサーションの自動生成手法が注目されている．アサーションの自動生成手法の一つであるアサーションの動的生成手法は，テストケースを用いて引数・フィールド変数の値を変化させながら対象メソッドを実行し，そこで得た情報を解析するため，比較的少ないコストでアサーションの生成が可能である．アサーションの動的生成ツールの一つに Daikon[2, 3, 4, 5, 6, 7] がある．しかし，アサーションの動的生成手法の問題点として，生成されるアサーションが実行データを取得する際に用いるテストケースに依存するテストケース依存問題がある [7].

この問題を改善するため，インバリアントカバレッジ [8, 9] が提案されている．このカバレッジの値が高いとき，動的生成ツールは信頼性の高いアサーションを生成できる．インバリアントカバレッジの値は，テストケースが対象メソッドの return 文とデータ依存関係のある命令文の組み合わせをどれだけ実行するかで求められる．著者が所属する研究グループでは，Daikon が生成するアサーションの評価 [10] とモデル検査技術を利用しインバリアントカバレッジの値が高いテストケースを自動生成する手法の提案を行ってきた [11, 12, 13].

本研究では，既存手法 [11, 12, 13] において問題となっていた適用可能なクラスの制限を改善する手法を提案する．既存手法では対象ソースコードに対してモデル検査器 Java PathFinder[14, 15] を用いて実行パスを取得し，取得した各パスに対して記号実行 [16, 17] を行うことでテストケース制約を導出していた．しかし，プリミティブでないデータ型に対して記号実行を行うことは難しいため，適用可能なクラスが狭くなっていた．本手法では静的解析器 ESC/Java2[18] が対象プログラムに対して出力する反例を解析し，実行パスの取得とテストケース制約の導出を行うことで，既存手法における問題点の改善を図る．ESC/Java2 が出力する反例中からインバリアントカバレッジに必要なパスを実行する際に成立する変数間の関係式，変数の型情報の抽出し整理する．これらは ESC/Java2 内の定理証明器 Simplify[19] により簡単化されているため記号実行と同様の結果を得ることができる．提案手法を複数の Java プログラムに対して適用し，評価実験を行った．評価実験では，適用可能なクラス，テストケース制約の有用性の評価を行った．その結果，適用可能なクラスが拡張されたこと，有用なテストケース制約が導出できたことを確認した．

以降，2 章で研究背景として，本研究に関連する概念，ツール，既存手法 [11, 12, 13] について説明を行う，続いて 3 章で提案手法について述べ，4 章で評価実験および考察について述べる．最後に 5 章でまとめる．

2　研究背景

本章では，本研究で用いる概念，ツールについて簡単に述べる．

2.1　Design by Contract

Design by Contract[1](以降，DbC とする) は，オブジェクト指向のソフトウェア設計に関する概念の１つで，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すことにより，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指している．契約は，クラスの利用側がそのクラスを利用する際にある条件 (事前条件) を保証すれば，そのクラスはある性質 (事後条件) を満たすことを保証するというものである．事前条件が満たせない場合はクラスを利用する側，事後条件が満たせない場合はクラス側の責任となる．このような責任の分離は開発者ごとの作業の分担を明確にし，ソフトウェアの欠陥の原因を切り分けるのに役立つ．

2.2　アサーション

プログラムのアサーションは，プログラムがソースコード中のある特定の場所で満たすべき条件を表す．DbC に基づいてアサーションを記述し，ソースコード内に直接アサーションを式の形で挿入することで，プログラムの実行時に契約が満たされるかどうか検証するができる．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，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のがある．また，DbC に基づいて記述された事前条件・事後条件は，ソースコードそのものの情報を端的に表わしているものであり，これら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．

プログラムのアサーションは，Java Modeling Language[20] や J2SE 1.4 から導入された Java の assert 文を用いて記述する．以下，図 1 に JML によるアサーションの記述例を示す．

図 1 の例ではクラス JML_Example のメソッド sum に対して JML によりアサーションを記述している．“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 型の配列変数の要素の総和を求め，その値を返す．そのため事前条件として，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
                }
                return s;
        }
}
```

図 1: JML によるアサーションの記述例

2.3 アサーションの自動生成手法

近年のソースコードサイズの増加に伴い手動によるアサーション生成は困難になってきているため，アサーションの自動生成手法や自動検査手法が注目されている．アサーションの生成と検査の自動化手法には，静的手法と動的手法の 2 種類がある [6]．次小節ではそれぞれについて詳細に述べる．

2.3.1 静的手法

静的手法 [14, 15, 21, 22, 23, 24, 25, 26, 27] はソースコードの状態を表すモデルを生成し，実行し得る全ての状態とそのときの実行条件を求めることで，アサーションを生成する．そのため，精度の高いアサーションの自動生成 [21, 22, 23, 24, 25, 26] や自動検査 [14, 15, 27] が可能である．しかし，一般的にはモデルの状態数に対するスケーラビリティが課題である．

2.3.2 動的手法

動的手法は対象ソースコードとテストケースを入力とし，テストケースを用いて対象ソースコードを実行し，得られたデータからアサーションを生成する．ここで，テストケースとは，対象メソッドの引数生成部とその引数が満たすべき条件，対象メソッド呼び出しの 3 つからなる手続きを表す．

そのため，テストケースの品質が低い場合，生成されるアサーションの精度が低下するという問題 (以降，テストケース依存問題とする)[7] が指摘されているが，テストケースの

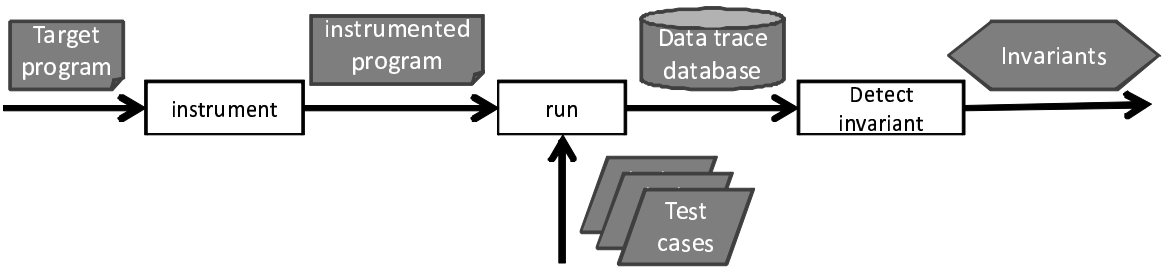

図 2: Daikon の処理

品質が高い場合は比較的少ない時間，メモリでアサーションの生成が可能である．そのため，動的手法はアサーションの自動生成に用いられることが多く，その代表的ツールとして Daikon[2, 3, 4, 5, 6, 7] がある．

Daikon は入力であるソースコードとテストケースから，実行時にメソッドの入口，出口 (以降，プログラムポイントとする) にて参照可能な変数の値を観測する準備を行い，テストケースを用いてソースコードを実行する (図 2)．そして，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．また，このとき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 [28].

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，デバッグにも有効である．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．

2.4 ESC/Java2

ESC/Java2[29] は Java プログラムに対する静的検証器である．ESC/Java2 は Java プログラムを入力とし，NullPointerException などの例外が発生する可能性がある箇所に対して警告を出力する．また，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の検証も行うことができる．ESC/Java2 は対象プログラムの検証を，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行う．判定エンジンとして Simplify[19] を用いている．

ESC/Java2 の簡単な動作例として，図 3 に示すソースコードのメソッド $int\ extractMin()$ を ESC/Java2 により検証した場合の出力結果を図 4 に示す．この例では，3 種類の警告が

```
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 ;
        }
}
```

図 3: ESC/Java2 への入力例

ESC/Java2 により出力されている．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.java 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 Inveriant Coverage

インバリアントカバレッジ (Invariant Coverage)[8, 9] 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る．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

```
Bag: extractMin() ...
--------------------------------------------------------
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.

--------------------------------------------------------
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.

--------------------------------------------------------
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.

--------------------------------------------------------
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.

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

図 4: ESC/Java2 の出力例

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．

インバリアントカバレッジ [9] 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 [9] 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．

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

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．

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

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．

$$C_{Inv} = DUC_{executed}/DUC_{all} \qquad (1)$$

2.6 既存手法

本節では著者が所属する研究グループが提案しているテストケース生成手法 [11, 12, 13] の概要と問題点について述べる．

2.6.1 概要

既存手法では以下の手順でテストケースを生成する．また，既存手法を実装してツールの概要図を図 5 に示す.

1. 対象ソースコードから DUC[9] 生成用プログラム依存グラフ [12] を生成し，プログラムポイントにて参照可能な変数を取得する．

2. DUC 生成用 PDG から対象メソッド内の全 DUC（定義部，使用部の位置はソースコード中の命令文の位置）を算出する．

3. Java PathFinder[14, 15] を用いて (Step 2) で取得した全 DUC の実行パスを取得する．

4. 取得した全実行パスを記号実行 [16] し，テストケース制約を算出する．

5. テストケース制約を満たすテストケースを生成する．

手順 3 の Java PathFinder を用いた実行パスの取得，手順 4 の記号実行によるテストケース制約の算出がこの手法で中心である．詳細は文献 [12] を参照されたい．

2.6.2 問題点

既存手法が適用可能なクラスには以下の制限がある．

1. 対象メソッドの仮引数の型がプリミティブ型もしくは String 型である．

2. 対象メソッド中の条件分岐に定数が含まれていない．

3. 対象メソッド中から呼び出されるメソッドに native メソッドが含まれていない．

4. 対象メソッド中にフィールド変数が含まれていない．

これらの制限がある理由として，テストケース制約の算出に用いている記号実行を対象プログラムのソースコードに対して行っているため，ライブラリクラスなどを含む一般的なプログラムに対して適用することが困難であるということが挙げられる．

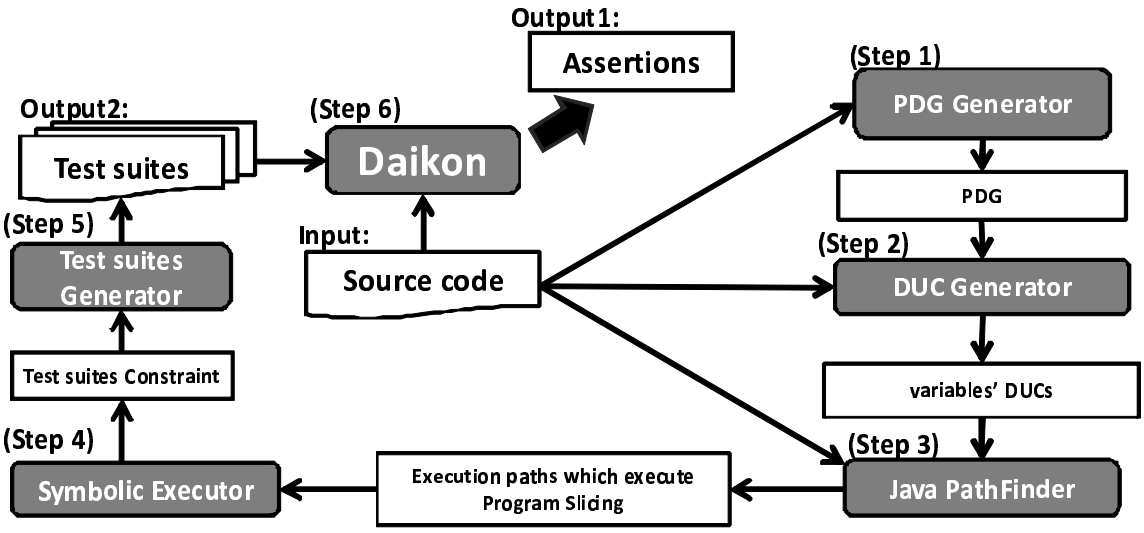

図 5: 既存手法の概要

3 提案手法

本章では，提案手法と簡単な例題への適用例について述べる．

3.1 提案手法概要

本節では提案手法の概要について述べる．提案手法を実装したツールの概要図を図 6 に示す．また，本手法の入力と出力は以下のとおりである．

- 入力：対象メソッドの Java ソースコード
- 出力：インバリアントカバレッジに基づくテストケース生成に必要なテストケース制約

提案手法では，静的解析器 ESC/Java2 の反例を解析しテストケース制約を求めることで，従来手法 [11, 12, 13] での課題であった適用可能なクラスの拡張を行っている．従来手法では Javapath Finder[14, 15] と記号実行 [16] を用いてテストケース生成に必要な情報を算出していた．しかし，参照型の変数を含む式に対する記号実行を実装することは困難である [30, 31]. これが適用可能なクラスを小さくする原因となっていた．

本手法ではこの問題を改善するために静的解析器 ESC/Java2 を用いる．また，ESC/Java2 は一般的な Java プログラムを解析するためのモジュールを持っているため，Java ライブラリクラスを利用したプログラムの解析が可能である．また，ESC/Java2 が出力する反例にはプログラム中の変数変数間に成立する条件や型情報が含まれている．これらの条件は ESC/Java2

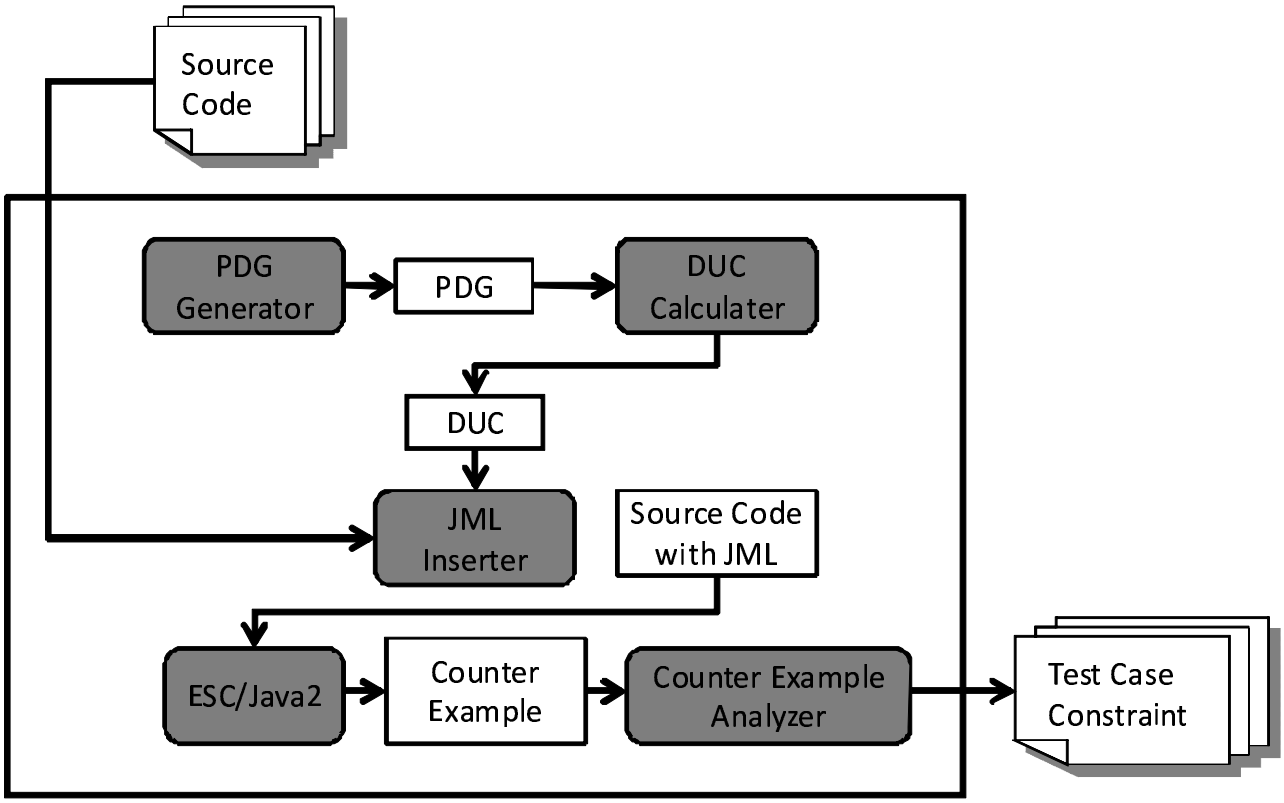

図 6: 本ツールによる処理の概要

内部の定理証明器 Simplify[19] により簡単化されているため，記号実行と同様の結果を得ることができる．

提案手法の処理の手順は以下のとおりである．

1. 対象メソッドの関数内プログラム依存グラフ (Intraprocedural Program Dependence Graph, 以降 PDG)[32] を生成する．
2. 手順 1 で生成した PDG より対象メソッドの DUC を取得する．
3. ESC/Java2 を用いて DUC を含むパスを実行するための条件を反例として取得するために，JML で記述したアサーションをもとの Java ソースコードに挿入する．
4. ESC/Java2 を実行し，反例を取得する．ESC/Java2 は手順 3 で挿入したアサーション以外に対する警告，反例も出力するが，本手法においてこれらへの処理は行わない．
5. 手順 4 で取得した反例を解析し，Daikon を用いる際に使用するテストケース生成に必要な情報を抽出する．

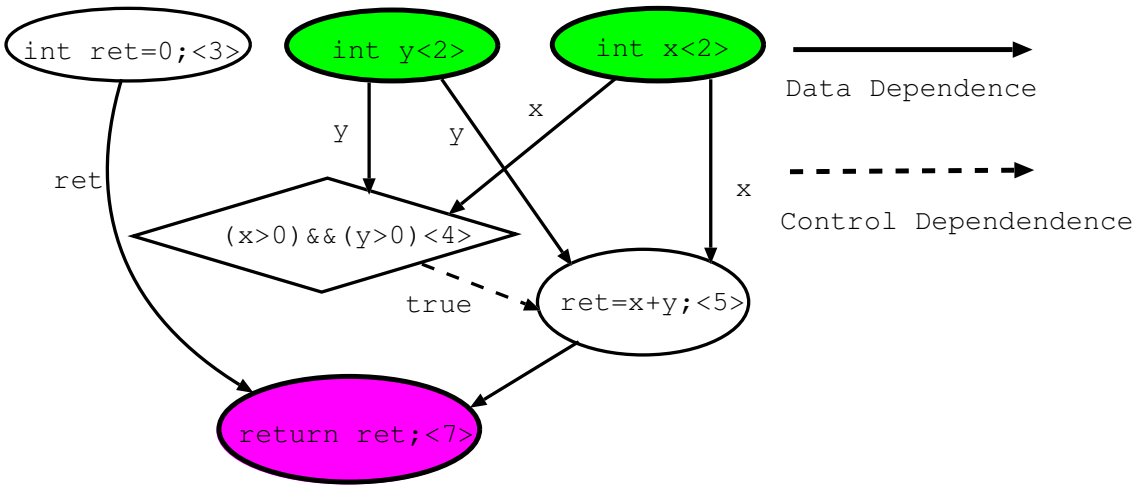

図 7: PDG の例

現在テストケースを自動生成する部分は一部未実装であるため，対象メソッドの仮引数の型が，プリミティブ型，String 型の場合 Daikon に入力するテストケースを自動生成することが可能である．その他の場合は，Daikon に入力するテストケースは手動で生成する必要がある．

3.2 PDG の生成

入力として与えられたソースコードから PDG[32] を生成する．PDG は命令文に対応するノードと，制御依存辺，データ依存辺からなるグラフである．また，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．

ここで，本ツールでは対象ソースコード全体に対して PDG を生成するのではなく，一部のみに対して PDG を生成している．これは，ソースコード全てを対象として PDG を生成するとクラスの多態性や動的束縛を考慮する必要があり，非常に時間・メモリなどのコストがかかるためである．また，データ依存関係はループ繰越依存関係は扱わずループ独立依存関係 [33] のみを対象としている．図 7 に図 8(a) に示したメソッド example(int, int) のソースコードから生成した PDG を示す．本研究ではこの PDG 生成部の実装には Masu[34]，Scorpio[35] を用いた．

3.3 DUC の取得

DUC[9] は，PDG の出口ノード（return 文などのメソッド終了時に実行される可能性のあるノード）からデータ依存辺を辿ることで得られる．また，DUC を通るパスを実行するた

めの条件を求めるために，PDG の制御依存辺の探索を行う．詳細な手順は以下のとおりである．

1. PDG の出口ノードを取得する．
2. 各出口ノードのデータ依存関係のあるノードを探索.
3. データ依存関係のあるノードとさらにデータ依存関係のあるノードを探索.
4. 3. で得られる各ノードと制御依存関係のあるノードを探索.
5. 3.4. をデータ依存関係，制御依存関係のあるノードがなくなるまで繰り返す.

3.4 ESC/Java2 用いたテストケース制約の導出

ESC/Java2 を用いて DUC を通るパスを実行するための条件を導出する方法について述べる．

3.4.1 反例の取得

DUC を含むパスを実行した際に ESC/Java2 が警告を出力するように JML[20] によるアサーションを元の Java ソースコードに挿入する．本手法において使用する JML ステートメントは以下の 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，DUC 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．

図 8 に上記の手順によるアサーションの挿入例を示す．図 8(a) が元のソースコード，(b) がアサーション挿入後のソースコードである．また，図 7 に図 8(a) に示したソースコードから生成した PDG を示す．この例では return 文中で使用される変数 ret の DUC を取得するた

めのアサーションを挿入している．メソッド emaple の PDG を生成し出口ノードを取得すると，7 行目の return 文がこのメソッドの出口ノードであることが分かる．そして，出口ノードからデータ依存辺を探索すると 7 行目から 5 行目，5 行目から 2 行目というデータ依存と 7 行目から 3 行目というデータ依存があることが分かる．このことから，変数 ret に関する具体的な DUC は以下の 3 つであるとわかる．

1. ret(5, 7) ← x(5, 2)
2. ret(5, 7) ← y(5, 2)
3. ret(7, 3)

また，DUC に含まれるノードの制御依存辺を探索すると 5 行目が 4 行目に制御依存していることが分かる．

図 8 のソースコード (a) には if 文が 4 行目に 1 つあるのでフラグ用変数$$1 を初期値を false として ghost 文を用いて宣言する．そして，if 文の条件を満たす場合には set 文を用いてフラグ用変数に true を代入する．最後に assert 文を用いてアサーションを記述する．ESC/Java2 は図 8(b) の 9 行目の assert 文に対して if 文の条件を満たさない場合，10 行目に assert 文に対して if 文の条件を満たす場合の反例を出力する．図 9 に図 8(b) に対して ESC/Java2 が出力した反例を示す．(a) が 9 行目の assert 文に対する反例，(b) が 10 行目の assert 文に対する反例である．

3.4.2 反例の解析

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

本手法では，これらの反例のうち，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

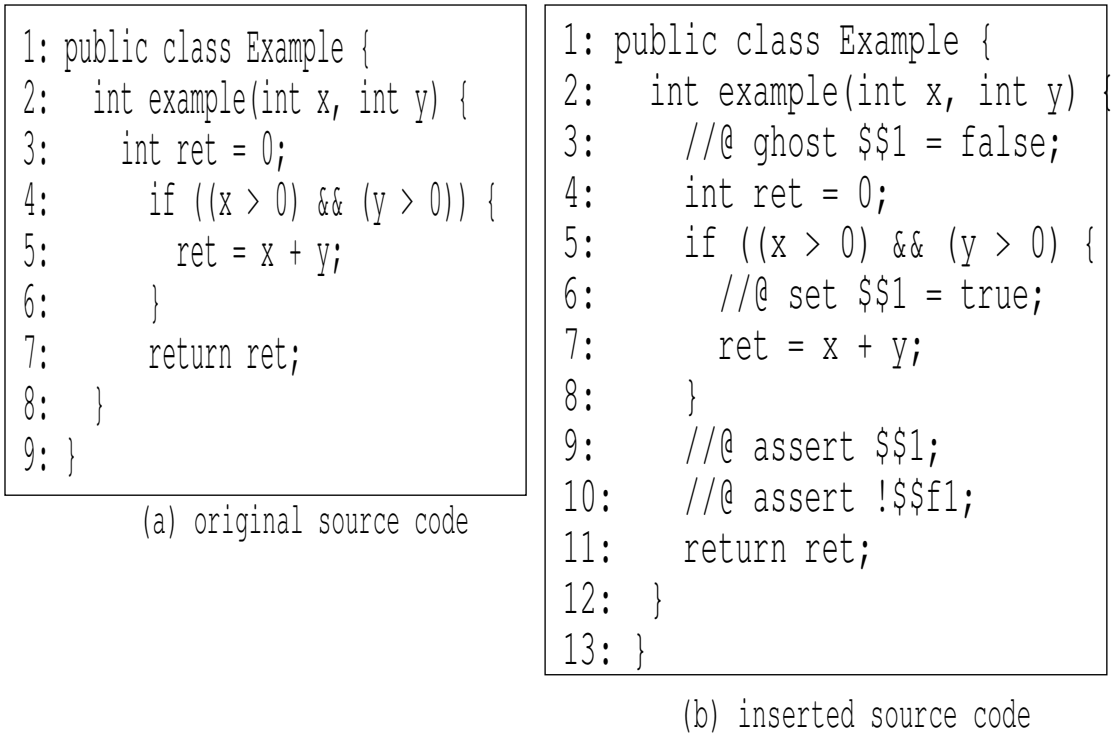

図 8: 条件取得用アサーションの挿入例

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ことで 4，5 の式を取り除くことができる．

テストケース生成に必要な情報を抽出するためには，1，2，3 の式を詳細に解析する必要がある．そのために本研究では ESC/Java2 が出力する反例の構文解析器を JavaCC[36, 37, 38] を用いて作成した．構文解析器を作成する上で ESC/Java2 が出力する反例の式の文法定義が必要となるが，ESC/Java2 の開発者によると反例の式の厳密な文法定義は存在しない．そのため，本手法では必要な式の文法を独自に定義している．しかし，ここで定義している文法は ESC/Java2 が出力する反例の式すべて網羅してはいないので，必要に応じて適宜拡張する必要がある．表 1, 2, 3, 4 に本研究で解析対象とした ESC/Java2 が出力する反例の式の文法定義を示す．

3.4.3 テストケース制約導出

本節では 3.4.1 節で述べた ESC/Java2 が出力する反例の処理方法について述べる．提案手法ではテストケース生成に必要な情報を分かりやすく出力するために，ESC/Java2 の反例を構文解析し以下の 4 種類に分類する．

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”

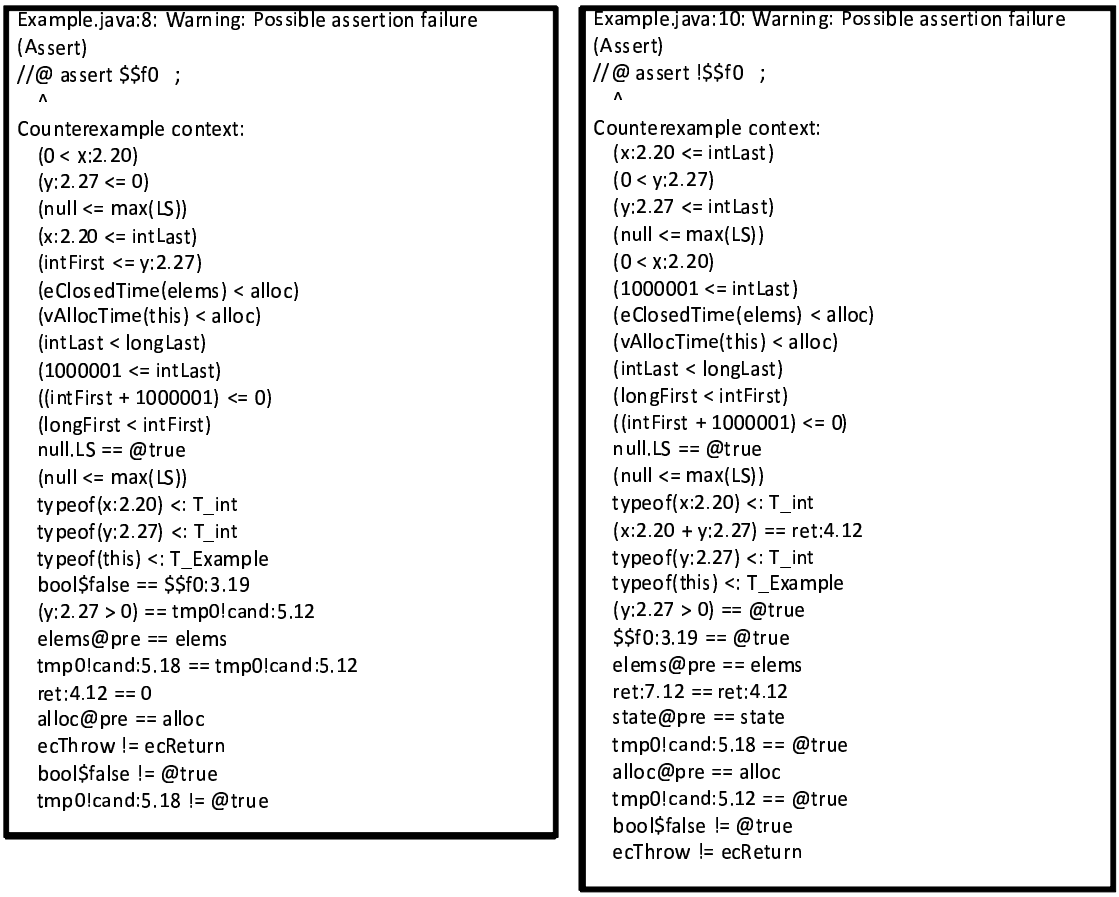

図 9: ESC/Java2 が出力した反例

1. 変数の型に関する式：type_expression，elemtype_expression，サブタイプ演算子”<:”を含む式.
2. 等式：”==”で結ばれた式で，変数の型に関する式でない式.
3. 関係式：”>”, ”<”, ”<=”, ”>=”で結ばれた式.
4. not equal 式：記号”!=”で結ばれた式.

また，JML 挿入後のソースコードに対して再度 PDG を生成する．この PDG の情報をもとに ESC/Java2 の反例に含まれている予約語を対応する元のソースコード上での変数名等に置換する．例えば，図 10 に示した出力例に「cand：5.18」というものがある．これは元のソースコード上の 5 行 18 桁目に存在する条件式に対応している．これを置換することで

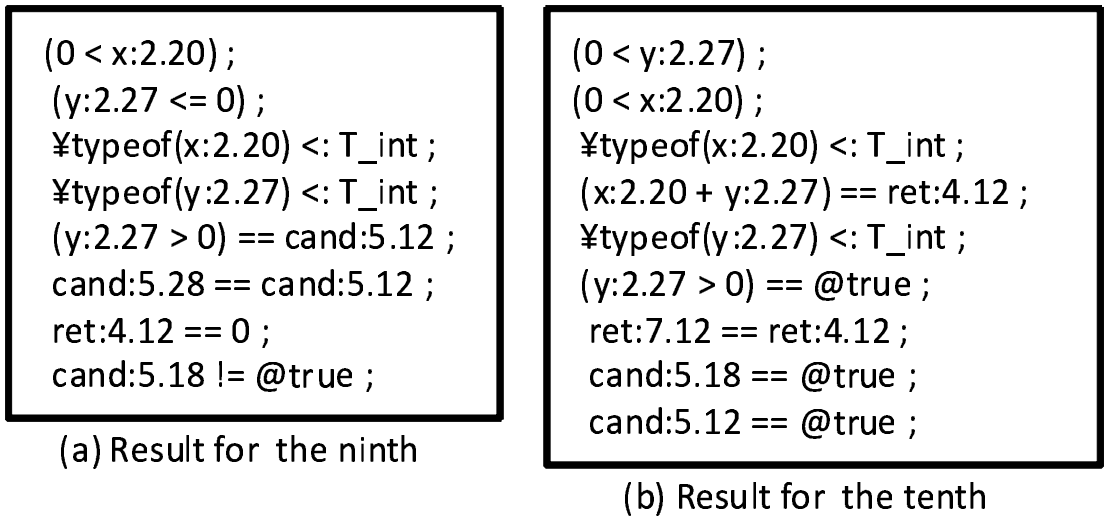

図 10: テストケース生成に必要な情報の抽出結果

「cand：5.18 != @true」から「x > 0 && y > 0 != @true」という式が得られる．このように PDG の情報を利用することでより可読性の高い情報を提供することができる．

3.5 テストケース

本節では 3.4.3 節で述べた手法により得られた結果から最終的なテストケースを生成する方法について述べる．

3.5.1 テストケースの構成

特定のパスを実行するようなテストケースを生成するためには，メソッドの引数やクラスのフィールド変数が特定の条件 (以降，テストケース制約) を満たす必要がある．あるメソッド m のテストケースは以下の 3 つの部分から構成される．

1. m の引数，m が属するクラスのフィールド変数生成部

   メソッド m へ入力する引数，m が属するクラスのフィールド変数の値を生成する．値の生成には基本的に乱数を用いる．このとき，効率よく値を生成するために乱数生成ライブラリ [39] を利用する．

2. テストケース制約判定部

   1 で生成された引数，フィールド変数の値がテストケース制約を満たしているかを判定する．テストケース制約を満たさない場合はその値は 3 のメソッド m の実行には用

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”

表 5: 分類結果 1

式の種類	式
型に関する式	\typeof(x) <: \type(int)
	\typeof(y) <: \type(int)
等式	y > 0 == (x > 0 && y > 0)
	(cand) == (x > 0 && y > 0)
	ret == 0
関係式	0 < x
	y <= 0
not eaual 式	(cand) != true

表 6: 分類結果 2

式の種類	式
型に関する式	\typeof(x) <: \type(int)
	\typeof(y) <: \type(int)
等式	x + y == ret
	y > 0 == true
関係式	0 < y
	0 < x
not equal 式	(cand) != true

いない.

3. m の実行部

   テストケース制約を満たす値を用いてメソッド m を実行する.

4. 1 から 3 の処理を一定回数繰り返す．Daikon で精度のよいアサーションを生成するためには数十回程の繰返しが必要である.

3.5.2 テストケースの作成

本節では 3.4.3 節で述べた方法に従って分類したからテストケースを作成する方法について述べる．現在，プリミティブ型，String 型の仮引数を持つメソッドに対するテストケースは自動生成することが可能である．図 12 に図 8 の (a) に対して自動生成された Daikon 用のテストケースを示す．テストケースは以下の手順により生成する.

1. 必要なパッケージのインポート，クラス宣言部などを生成する.

2. 対象メソッドの実引数生成部分を生成する．プリミティブ型のデータ生成には jav.util.Random クラス [40]，String 型のデータ生成には Apache Commons プロジェクトの commons lang ライブラリ [39] を利用している.

3. 3.4.3 節で述べた方法により得られたテストケース制約の論理積をとったものを if 文の条件とし手順 2 で生成された値が Invariant カバレッジに必要なパスを実行するために必要な値かを判定する．この条件を満たす入力データが Invariant Coverage に必要なパスを実行するために必要なデータである.

```
while(true){
    [1] Parameters  and Fields Generation
    target method's paramters = …
    target class's fields = …
    [2] Condition check
    if ( testcase constraints  are satisfied){
        [3] Method Call
        method call with fields and parameters
        break;
    }
}
```

図 11: テストケースの処理

4. 手順 3 で得られた値を入力とし対象メソッドを実行する部分を生成する.

5. 手順 2 から 4 をメソッドの return 文の数だけ行う.

なお，プリミティブ型，String 型以外の実引数は単純に乱数を用いて生成することが難しいため，人手により作成する必要がある.

```
import org.apache.commons.lang.RandomStringUtils;
import java.util.Random;
public class TestCase_example{

public static void main(String[] args) {
        PDG_Test18 $$test = new PDG_Test18();
        Random random = new Random();
        int $$i = 0;
        while (true) {
                int y = random.nextInt(10);                                    10
                int x = random.nextInt(10);
                int ret = random.nextInt(10);
                try{
                        if ( ret == 0 && 0 < x && y <= 0) {
                                $$test.example(x, y );
                                $$i++;
                        }
                } catch (Exception e) {
                }
        if ($$i > 20) {                                                        20
                break;
        }
        }

        $$i = 0;

        while (true) {
                int y = random.nextInt(10);
                int x = random.nextInt(10);
                int ret = random.nextInt(10);                                  30
                try{
                        if ( y > 0 == true && 0 < y && 0 < x) {
                                $$test.example(x, y );
                                $$i++;
                        }
                } catch (Exception e) {
                }
                if ($$i > 20) {
                        break;
                }                                                              40
        }
        $$i = 0;
        }
}
```

図 12: 図 8 の (a) に対して自動生成されたテストケース

4 評価と考察

本手法の有用性評価のために行った実験について述べる．なお，実験環境は以下の通りである．

- 計算機：HP xw8600
- OS：Windows Vista Business
- CPU：Intel Xeon E5405 2.00GHz × 2
- Memory：8.00GB
- Java：JDK 1.4.2_16(ESC/Java2 実行用)，JDK 1.6_17(本ツール，Daikon 実行用)
- ESC/Java2：version 2.0.5
- Daikon：version 4.5.3

実験対象は，あるショッピングサイトの Web アプリケーション，オープンソースのメソッド，文献 [11, 12, 13] の実験で用いられたプログラムを使用した．

4.1 評価方法概要

4.2 評価基準

以下の項目について評価を行った．

1. 本ツールが適用可能なクラス

   既存手法 [11, 12, 13] に比べて適用可能な範囲が広がっているか評価する．

2. Daikon が生成したアサーションの妥当性

   本手法により得られたテストケース制約を用いて作成したテストケースを用いて Daikon を実行し，生成されたアサーションの妥当性を評価する．

3. 本ツールの実行時間

   本手法を実装したツールの実行時間を評価する．現在，本ツールでは一部の対象のみテストケース自動生成が行えるため，対象 Java プログラム入力からテストケース制約導出部までを，本ツールの実行時間とする．

```
public class BM {

    public static int BMmatch(final java.lang.String text, final java.lang.String pattern) {
    int[] last = BM.buildLastFunction(pattern);
    int n = text.length();
    int m = pattern.length();
    int i = m - 1;
    if (i > n - 1) {
        return -1;
    }
    int j = m - 1;
    do {
        if (pattern.charAt(j) == text.charAt(i)) {
            if (j == 0) {
                return i;
            }
            else {
                i--;
                j--;
            }
        }
        else {
            i = i + m - Math.min(j,1 + last[text.charAt(i)]);
            j = m - 1;
        }
    }while (i <= n - 1);
    return -1;
  }

  public static int[] buildLastFunction(String pattern) {
    int[] last = new int[128]; // assume ASCII character set
    for (int i = 0; i < 128; i++) {
        last[i] = -1; // initialize array
    }
    for (int i = 0; i < pattern.length(); i++) {
        last[pattern.charAt(i)] = i; // implicit cast to integer ASCII
    }
    return last;
  }
}
```

図 13: 対象ソースコード

4. Daikon の実行時間

   Daikon による対象プログラムの実行時情報取得にかかる時間，アサーション生成にかかる時間を評価する．

4.3 適用可能なクラス

本節では適用可能なクラスの評価について述べる．本手法が適用可能な範囲は ESC/Java2 が適用可能なクラスに依存するところが大きい．ESC/Java2 は Java1.4 を対象としているため，Java1.5 以降に導入された文法，クラスは扱うことはできない．また，ESC/Java2 が解析できないクラスは明確に公表されていないため，Java1.4 中のクラスであっても適用可能かを確認する必要がある．

本研究では Java ライブラリクラス [40] のうち使用頻度が高いと考えられる，List, Map, Set インターフェース，Math クラスに対して解析可能か確認を行った．確認方法として，各ライブラリクラスのメソッドを使用した小規模なプログラムを作成し本ツールの入力とし，出力結果にテストケース生成に必要な情報が含まれているかを確認した．確認の結果，Math クラスには ESC/Java2 により解析できないメソッドがあった．それらのメソッドを表 7 に示す．

4.4 テストケース制約の有用性

本節では，本手法により導出されたテストケース制約の有用性について評価する．従来手法では適用できなかった対象プログラムに対して本手法を適用し，出力された全テストケース制約の内，実際にテストケースを作成する上で必要なものがどの程度の割合かを調べる．各対象プログラムに対して本手法を適用し得られたテストケース制約を調べた．その結果，アサーションの動的生成用のテストケース作成に有用なもの，不要なものと判断されたものの数をまとめたものを表 8 に示す．なお，“有用なもの”，“不要なもの”は以下のように定義する．

- 有用なもの：インバリアントカバレッジに必要なパスを実行するために対象メソッドの実引数が満たすべき条件
- 不要なもの：誤った条件，またはテストケース作成時に必要でない条件

表 8 を見ると各対象に対して有用なテストケース制約が生成できていることが確認できる．生成されたテストケース制約の数を見ると，compareHase, equals, BMmatch, regionMatches において多く生成されていることが分かる．これらのメソッドは複数の return 文を持ち，また仮引数の数も他の対象よりも多ため，考慮すべきテストケース制約が多くなったと考えられる．

表 7: 適用不可能なメソッド

メソッド名	機能
static double acos(double)	指定された値のアークコサインを返す.
static double asin(double)	指定された値のアークサインを返す.
static double cbrt(double)	指定された値の立方根を返す.
static double cos(double)	指定された値のコサインを返す.
static double expm1(double)	$e^x - 1$ を返す.
static double floor(double)	引数の値以下で、計算上の整数と等しい、 最大の (正の無限大にもっとも近い) double 値を返す.
static double hypot(double, double)	中間のオーバーフローやアンダーフローなしに $sqrt(x^2 + y^2)$ を返す.
static double IEEEremainder(double, double)	IEEE 754 標準に従って，2 個の引数の剰余を計算する.
static double log(double)	指定された double 値の自然対数値 (底は e) を返す.
static double log10(double)	double 値の 10 を底とする対数を返す.
static double log1p(double)	引数と 1 の合計の自然対数を返す.
static long round(double)	引数にもっとも近い long を返す.
static int round(float)	引数にもっとも近い int を返す.
static duble signum(double)	引数の符号要素を返す.
static float signum(float)	引数の符号要素を返す.
static double sin(double)	指定された角度のサインを返す.
static double sinh(double)	指定された値の双曲線正弦を返す.
static double tan(double)	指定された角度のタンジェントを返す.
static double tanh(double)	指定された double 値の双曲線正接を返す.
static double ulp(double)	指定された値の ulp のサイズを返す.
static double ulp(float)	指定された値の ulp のサイズを返す.

表 8: 生成されたテストケース制約の評価結果

対象プログラム名	有用なもの (個)	不要なもの (個)	総数 (個)
formatDate	2	1	3
formatDateWithTime	2	1	3
getDate	4	2	6
getProperty	8	3	11
isAlphabet	15	5	20
isNullOrEmpty	7	1	8
calKai	14	5	19
compareHash	21	4	25
equals	45	11	56
BMmatch	19	14	36
isCredit	7	1	8
regionMatches	25	17	42

表 9: 本ツールの実行時間

対象プログラム名	DUC 取得部 (秒)	ESC/Java2(秒)	反例解析部 (秒)	全実行時間 (秒)
formatDate	4.4	7.1	10.4	22.1
formatDateWithTime	4.1	6.7	7.9	18.9
getDate	4.3	6.4	8.3	20.7
getProperty	4.5	9.4	8.5	22.9
isAlphabet	4.2	9.4	8.4	22.5
isNullOrEmpty	4.5	11.8	8.5	25.4
calKai	2.2	8.1	10.1	20.1
compareHash	2.8	5.8	11.1	20.0
equals	2.0	5.6	11.7	19.5
BMmatch	3.4	10.3	12.2	26.2
isCredit	4.6	38.5	8.7	54.2
regionMatches	7.4	6.1	10.2	24.0

4.5 Daikon による生成アサーションの評価

Java プログラムに対して，提案手法を用いて生成したテストケースと従来手法を用いて生成したテストケースを使用し Daikon によるアサーションの生成を行い，生成されたアサーションの比較を行う．

対象 Java プログラムとして図 13 に示した Boyer-Moore 文字列探索アルゴリズムを実装したメソッド BMmatche(String, String) を選んだ．Boyer-Moore 文字列検索アルゴリズムは 2 つの文字列に対して適用するアルゴリズムで，片方の文字列がもう片方の文字列を含む場合は先頭文字列のインデックスを，含まない場合は-1 を返す．複数の while ループ，複数の return があることから多くの実行パスがあるためアルゴリズムが難解であり，アサーションを生成することが難しいと考えたため，このソースコードを例題として選んだ．

Daikon が生成したアサーションのの評価基準として適合率，再現率，F 値を用いた．各値の定義は以下のとおりである．

定義 4.1. 適合率，再現率，*F* 値

生成されたアサーションの集合を A_g，対象のソースコードから人が必要だと判断したアサーションの集合を A_n とする．このとき，適合率 P，再現率 R は以下のように定義される．

$$P =\mid A_g \cap A_n \mid / \mid A_g \mid, R =\mid A_g \cap A_n \mid / \mid A_n \mid$$

F 値 F は適合率 P と再現率 R の調和平均であるので，F は以下のように定義される．

$$F = 2 \times P \times R/(P + R)$$

提案手法を用いて生成したテストケース，従来手法を用いて生成したテストケースを使用し図 13 に示した Java プログラムに対して Daikon でアサーションの生成を行った．提案手法を用いて生成したテストケースの場合は事前条件が 4 個，事後条件が 3 個生成された．従来手法を用いて生成したテストケースの場合は事前条件，事後条件ともに 4 個生成された．これらの結果より適合率，再現率，F 値を求め，比較した結果を表 10 に示す．この結果をみると，事前条件においては提案手法を用いた場合，従来手法を用いた場合の適合率，再現率が同じであることが確認できる．そして，事後条件においては提案手法を用いた場合より従来手法を用いた場合のほうが適合率，再現率ともに高いことが確認できる．

最初に事前条件の結果について考察を述べる．生成されたアサーションの内容を調べると提案手法を用いた場合と従来手法を用いた場合に生成されたアサーションは同じであった．正しい事前条件として対象メソッドの 2 つの引数が null でないことを表す「text != null」，

「pattern != null」は生成できていたが，引数の型に関する条件「\typeof(text) == \type(String)」，「\typeof(text) == \type(String)」は生成されなかった．これは Daikon では引数の型に関するアサーションは生成することができないためである．提案手法が出力するテストケース生成に必要な情報には「\typeof(text) == \type(String)」,「\typeof(text) == \type(String)」が含まれているため，Daikon が生成したアサーションと合わせることで，より良いアサーションを作成することが可能であると思われる．

次に事後条件の結果について考察を述べる．2 つの引数がメソッドの実行の前後で変化しないことを表す「text.toString().equals(\old(text))」,「pattern.toString().equals(\old(pattern))」というアサーションは提案手法を用いた場合，従来手法を用いた場合ともに生成された．従来手法を用いた場合にのみ生成された正しいアサーションに「\result >= -1」がある．これは，対象メソッドの返り値の下限の条件として正しいものであり，このメソッドの事後条件として重要なものであると言える．対象メソッドの返り値の上限の条件は提案手法を用いた場合，従来手法を用いた場合ともに正しく生成されなかった．

4.6 実行時間

本節では本ツールの実行時間，Daikon の実行時間を評価する．

4.6.1 本ツールの実行時間

各実験対象に本ツールを適用した際の実行時間の結果を表 9 に示す．ESC/Java2 の実行時間を見ると isCredit の実行時間が約 39 秒と長いが，他のプログラムに対しては 10 秒以内に実行が終了している．isCredit に対する実行時間が長い原因は，isCredit が属しているクラスには isCredit 以外にも複数のメソッドが存在しているためであると考えられる．ESC/Java2 はクラス単位で入力しクラス内の全てのメソッドを解析対象とするため対象クラス内メソッドが多いと実行時間が長くなってしまう．実行時間全体を見ると全ての対象に対して 1 分以内で実行できており，比較的実行時間が短いと言える．

表 10: 生成アサーションの適合率・再現率・F 値の比較

テストケースの種類	アサーションの種類	適合率	再現率	F 値
提案手法	事前条件	1.00	0.50	0.67
	事後条件	0.67	0.50	0.57
従来手法	事前条件	1.00	0.50	0.67
	事後条件	0.75	0.75	0.75

表 11: BMmatch に対する Daikon の実行時間

手法	対象プログラムの実行時間	アサーション生成にかかった時間
提案手法	3 秒	7 秒
従来手法	630 秒	6 秒

4.6.2 Daikon の実行時間

図 13 に示したソースコードに対して提案手法により生成したテストケースと従来手法により生成したテストケースを用いた場合の Daikon の実行時間を表 11 に示す．この結果をみると Daikon の実行時間は提案手法によるテストケースを用いた場合のほうが短いことが分かる．この理由はループ部の展開回数によるものであると考えられる．従来手法では Java プログラム中のループ部を 2 回まで展開するが，提案手法の現在の実装では 1 回しか展開しない．よって，従来手法で生成されたテストケースはループ部を 2 回まで実行する．一方，提案手法で生成されたテストケースはループを 1 回しか実行しない．このため Daikon が実行時情報取得のために対象プログラムを実行した場合に提案手法の方が実行時間が短くなる．これに伴い，Daikon が取得する実行時情報も提案手法のほうが少なくなる．その結果 Daikon がアサーション生成時の実行時情報解析時間が短くなるので，アサーション生成にかかる時間が短縮されたと思われる．

4.7 考察

本節では，4.3，4.4，4.5，4.6 節に示した評価実験の結果に対する考察を述べる．

4.7.1 本手法が適用可能なクラス

本節では，4.3 節に示した結果に対する考察を述べる．適用実験の結果従来手法を適用することができなかった対象に対してテストケース制約を導出することができた．プリミティブでない参照型のデータを含むプログラムに適用できるようになったことで，従来手法に比べより実用的なプログラムに対して使用できると言える．本手法の適用可能なクラスは基本的に ESC/Java2 に依存する．表 7 に示したメソッドなど，ESC/Java2 が解析不可能なプログラムには本手法は適用できない．また，ESC/Java2 は Java1.4 までしか対応していないため，Java1.5 以降導入された機能を含むプログラムに対しては適用不可能である．

4.7.2 テストケース制約の有用性

本節では，4.4 節に示した結果に対する考察を述べる．表 8 を見ると各対象プログラムのテストケースを作成する上で必要なテストケース制約が生成されていることが分かる．生成されたテストケース制約と対象プログラムのソースコードを調査し，生成されたテストケース制約を用いて実際にテストケースを作成することが可能であることを確認した．しかし，不要な条件も生成されている．不要と判断した条件を調べたところ，論理的に誤りのある条件は存在しなかった．不要な条件の内容はローカル変数しか含まない条件，冗長な条件の 2 種類であった．

テストケース作成時には対象メソッドへ入力する実引数，対象メソッドが属するクラスのフィールド変数に関する条件が必要となる．このため，メソッド内のローカル変数のみに関する条件が得られてもテストケース作成には役立たない．このような条件が生成された理由は，本ツール内の ESC/Java2 の反例解析部におけるフィルタリング処理が不完全であるためである．具体的には，現在の実装では反例中の変数を処理する際に，その変数がローカル変数であるかの判定を行っていない．このため，最終的に出力されるテストケース制約にローカル変数のみに関する条件が含まれてしまっている．

冗長な条件とは他の条件に包含される条件のことである．例えば regionMatches に対して生成されたテストケース制約中に変数 match の型が char 型の配列であることを示す「\typeof(match) == \type(char[])」と変数 match が char 型の配列のサブタイプであることを示す「\typeof(match) <: \type(char[])」が含まれている．Java において char 型はプリミティブ型であるので,「変数 match が char 型の配列である」という情報があれば,「char 型の配列のサブタイプである」という情報は不要である．このように冗長な条件が含まれていると可動性が下がってしまう．このような冗長な条件が生成される理由は，現在の反例解析部において条件式間の論理関係を解析できていないためである．

4.7.3 実行時間

本節では，本手法の実行時間について考察を述べる．従来手法を実装したツールの実行時間と本手法を実装したツールの実行時間との比較を行うために，表 12 に従来手法を実装したツールの実行時間を，表 13 に本ツールを実装したツールの実行時間を示す．テストケース制約の導出は，従来手法では “JPF” と “記号実行” で行れており，本手法では “ESC/Java2”, “反例解析部” で行われている．

従来手法では全実行時間のうち Java PathFinder の実行時間が占める割合が大きいことが分かる．Java PathFinder は対象プログラムのバイトコードを独自の JVM 上で実行し解析を行う．このため DUC を含む実行パス取得には対象プログラムが入力値として取りうる値を

表 12: 従来手法の実行時間

	モジュール				
対象メソッド	DUC 生成部	JPF	記号実行	テストケース生成部	全体実行時間
calc	6 秒	5 秒	1 秒	1 秒	47 秒
compareHashes	5 秒	29 秒	3 秒	1 秒	1 分 14 秒
BMmatch	1 分 23 秒	16 分 9 秒	3 分 17 秒	16 秒	25 分 58 秒
equals	4 秒	43 秒	2 秒	1 秒	1 分 16 秒

表 13: 本手法の実行時間

	モジュール				
対象メソッド	DUC 生成部	ESC/Java2	反例解析部	テストケース生成部	全体実行時間
calc	2 秒	8 秒	10 秒	0.8 秒	22 秒
compareHashes	3 秒	6 秒	11 秒	1 秒	20 秒
BMmatch	4 秒	10 秒	12 秒	2 秒	28 秒
equals	4 秒	5 秒	12 秒	1 秒	20 秒

網羅的に入力し，繰り返し実行を行う必要がある．このことが本手法において JPF 実行時間が長くなる原因となっている．一方，ESC/Java2 は対象プログラムのソースコードより論理式を生成し静的検証を行う．このため，対象プログラムの繰り返し実行が不要になり Java PathFinder に比べ実行時間の短縮が可能になっている．

本ツールでは ESC/Java2 の反例解析部の実行時間が全体にしめる割合が大きく，また，従来手法ではテストケース制約導出に 6 秒程度で処理が終わった対象に対しても本手法では 20 秒程度かかっている．これは，ESC/Java2 が規模の小さいプログラムに対しても多くの反例を出力するために反例解析部の実行時間が長くなったためである．

4.7.4 既存手法との比較

本節では，従来手法と本手法を比較について述べる．従来手法と本手法の利点，欠点を表 14 に示す．本手法の最大の利点は従来手法では適用することができなかった参照型のデータを含む対象に対して適用可能な点である．しかし，欠点として対象プログラムによっては Java PathFinder と記号実行を用いたテストケース導出よりも本手法のテストケース導出部の

表 14: 既存手法との比較

	既存手法	提案手法
適用可能なクラス (プリミティブ型)	○	○
適用可能なクラス (非プリミティブ型)	×	○ (Java1.4 に限る)
テストケース制約導出部の実行時間	△ (対象による差が大きい)	○
テストケースの自動生成 (プリミティブ型)	○	○
テストケースの自動生成 (非プリミティブ型)	×	×

実行時間が長くなっていることがあげられる．この理由としては，ESC/Java2 が出力する反例は対象プログラムの規模が小さい場合でも非常に多いため，反例解析部の実行時間が長くなってしまうからである．

テストケースの自動生成は従来手法と同様に対象メソッドの仮引数がプリミティブ型，String 型に限り可能である．この理由は，テストケース内における対象メソッドへの入力データは乱数を用いて生成しているからである．非プリミティブ型のデータは単純に乱数などでの生成が難しいため，現時点では実装できていない．このため，非プリミティブ型の仮引数を持つメソッドに対するテストケース作成には人手による作業が必要となる．著者が所属する研究グループの学部 4 年生に手動でテストケースを作成してもらったところ，一つのテストケース作成には 30 分程度かかった．本人に作業後に話を伺ったところ，テストケース作成に慣れていないために，どのような手順で作業を進めていけばよいのか分からなかった，という意見が得られた．このため，テストケース作成は慣れないうちは時間がかかるが，ある程度経験を積むことで作成にかかる時間を短くすることができると考えられる．また，テストケースのテンプレート，入力データ生成用のユーティリティメソッドを用意することでテストケース作成者の負担を軽減することができると考えている．

現在，一般的なプログラムのテスト時に用いる入力データを乱数を用いずに自動生成する手法に関する研究は様々行われている [41, 42, 43, 44, 45, 46]．従来研究，本研究のテストケース自動生成部を改良していくためには，これらの研究を調査する必要があると考えている．

5　あとがき

本研究では，著者が所属する研究グループで提案されてきたインバリアントカバレッジの値が高いテストケースを生成する手法の問題点である適用可能なクラスの制限を，静的解析器 ESC/Java2 を用いることで改善する手法の提案を行った．本手法では ESC/Java2 が持つ Java プログラムに対する解析能力，定理証明器による式の簡単化を利用することで，ライブラリクラスを含むプログラムなど，従来手法に比べより広いクラスに対しテストケース制約を導出可能である．また，本手法を複数の Java プログラムに対して適用し，有用性の評価を行った．その結果，有用なテストケース制約を導出できたことを確認した．

今後の課題としては，現在実現できていない非プリミティブ型のデータを含むプログラムに対するテストケースの自動生成手法の実現があげられる．また，さらなる評価実験を行い手法の有用性をより詳細に評価したい．

謝辞

本研究を行うにあたり，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じ，的確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.

本研究において，常に適切なご助言およびご指導を頂きました肥後芳樹助教に深く感謝致します.

本研究に多大なるご助言およびご指導を頂きました柿元健特任助教に深く感謝致します.

本研究について多くの協力，助言頂いた大阪大学基礎工学部情報科学科 4 年 小林 和貴氏に深く感謝致します.

本研究で使用した評価実験のデータを提供いただいた日本ユニシス・ラーニング株式会社 HRD 価値提供事業部　毛利幸雄様，三井利江子様に深く感謝致します.

その他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，本学基礎工学部所属時より現在に至るまで，講義，演習，実験等を通じてお世話頂きました諸先生方にお礼申し上げます.

参考文献

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

[2] 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.

[3] M. D. Ernst: “Dynamically Discovering Likely Program Invariants,” *PhD thesis,University of Washington Department of Computer Science and Engineering,Seatle Washington*, 2000.

[4] 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, 2001.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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, ASE 2003*, pp. 49–58, 2003.

[10] 宮本敬三, 岡野浩三, 楠本真二: “アサーション動的生成のためのテストケース自動生成手法の生成アサーションの妥当性評価”, ソフトウェア工学の基礎 XVI 日本ソフトウェア科学会 FOSE2009, pp. 183–190, 2009.

[11] 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.

[12] 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.

[13] 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.

[14] W. Visser, K. Havelund, k. G. Brat, S. Park, and F. Lerda: “Model Checking Programs,” *Automated Software Engineering Journal 2003*, Vol. 10, No. 2, pp. 202–232, 2003.

[15] F. Lerda and W. Visser: “Addressing Dynamic Issues of Program Model Checking,” *in Proc. of Int. Workshop on SPIN Model Checking 2001*, pp. 88–102, 2001.

[16] J. C. King: “Symbolic Execution and Program Testing,” *Communications of the ACM 1976*, Vol. 19, No. 7, pp. 385–394, 1976.

[17] C. S. P. S. Khurshid and W. Visser: “Generalized Symbolic Execution for Model Checking and Testing,” *in Proc. of Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2003*, pp. 553–568, 2003.

[18] 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.

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

[20] G. T. Leavens, A. L. Baker, and C. Ruby: “JML:A Notion for Detailed Design,” *in Behavioral Specifications of Businesses and Systrems*, pp. 175–188, 1999.

[21] P. Cousot and R. Cousot: “Modular Static Program Analysis,” *in Proc. of Int. Conf. on Compiler Construction, LNCS*, Vol. 2304, pp. 159–178, 2002.

[22] P. M. Cousot and R. Cousot: “Automatic Synthesis of Optimal Invariant Assertions: Mathematical Foundations,” *in Proc. of the ACM Symp. on Artificial Intelligence and Programming Language*, pp. 1–12, 1997.

[23] C. C. Gannod and B. H. C. Cheng: “Strongest Postcondition Semantics as the Formal Basis for Reverse Engineering,” *in Proc of Second Working Conf. on Software Engineering, WCRE 1995*, pp. 188–197, 1995.

[24] F. C. N. Tillmann and W. Schulte: “Discovering likely method specifications,” *Int. Conf. on Formal Engineering Methods, ICFEM 2006, LNCS*, Vol. 4260, pp. 717–736, 2006.

[25] C. Flanagan and K. R. Leino: “Houdini, an Annotation Assistant for ESC/Java,” *in Proc. of Int. Symp. of Formal Methods Europe on Formal Methods for Increasing Software Productivity, FME 2001*, pp. 500–5178, 2001.

[26] P. H. Schmitt and B. Weiss: “Invariants by Symbolic Execution,” *in Proc. of the 4th International Verification Workshop, VERIFY 2007*, pp. 195–210, 2007.

[27] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe: “Extended static Checking,” *SRC Research Report 159, Compaq SRC*, 1998.

[28] J. W. Nimmer and M. D. Ernst: “Automatic Generation of Program Specification,” *in Proc. of SIGSOFT Int. Symp. on Software Testing and Analysis 2002*, pp. 232–242, 2002.

[29] L. Burdy, Y. Cheon, D. R. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll: “An overview of JML tools and applications,” *In T. Arts and W. Fokkink, editors, Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS03), Electronic Notes in Theoretical Computer Science*, Vol. 80, pp. 73–89, 2003.

[30] Coen-Porisini, Alberto, Denaro, Giovanni, Ghezzi, Carlo, and M. Pezzé: “Using symbolic execution for verifying safety-critical systems,” *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 142–151, 2001.

[31] K. Sarfraz and S. Y. Lai: “Generalizing symbolic execution to library classes,” *PASTE ’05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 103–110, 2005.

[32] 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.

[33] 伊藤宗平, 萩原茂樹, 米崎直樹: “中間コードを表すプログラム依存グラフの操作的意味”, 日本ソフトウエア科学会大会講演論文集, 2006.

[34] 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.

[35] “Scopio,” *http://www-sdl.ist.osaka-u.ac.jp/ higo/cgi-bin/moin.cgi/scorpio*.

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

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

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

[39] “Apache Commons,” *http://commons.apache.org/*.

[40] “JavaTM Platform, Standard Edition 6,” *http://java.sun.com/javase/ja/6/docs/ja/api/overview-summary.html*.

[41] C. Boyapati, R. Boyapati, S. Khurshid, and D. Marinov: “Korat: automated testing based on Java predicates,” *ISSTA ’02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 123–133, 2002.

[42] N. Tillmann and W. Schulte: “Parameterized unit tests,” *SIGSOFT Softw. Eng. Notes*, Vol. 30, No. 5, pp. 253–262, 2005.

[43] S. Kevin, Y. Jinlin, C. David, K. Sarfraz, and J. Daniel: “Software assurance by bounded exhaustive testing,” *SIGSOFT Softw. Eng. Notes*, Vol. 29, No. 4, pp. 133–142, 2004.

[44] R. P. Pargas, M. J. Harrold, and R. R. Peck: “Test-Data Generation Using Genetic Algorithms,” *Software Testing, Verification And Reliability*, Vol. 9, pp. 263–282, 1999.

[45] D. Marinov and S. Khurshid: “TestEra: A Novel Framework for Automated Testing of Java Programs,” *In Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, Vol. 9, pp. 263–282, 1999.

[46] J. Miller, M. Reformat, and H. Zhang.: “Automatic test data generation using genetic algorithm and program dependence graphs,” *Information and Software Technology*, Vol. 48, pp. 568–605, 2006.