特別研究報告

題目

Javaの実行パスからパス実行可能性判定式への変換とSMTソルバによる求解

指導教員

楠本 真二 教授

報告者

佐々木　幸広

平成 24 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 23 年度 特別研究報告

Java の実行パスからパス実行可能性判定式への変換と
SMT ソルバによる求解

佐々木　幸広

内容梗概

Design by Contract に基づく表明の記述はソースコードの仕様理解の補助やプログラムの検証に役立つ．しかし，ソースコードのサイズの増加に伴い，表明の記述量も増大する．そのため，表明の自動生成手法が注目されている．表明の自動生成手法には静的生成手法と動的生成手法が存在する．表明の静的生成手法では，プログラムを静的に解析することで，表明を導出する．しかし，表明生成にかかる時間が長い点が問題となる．表明の動的生成手法では，テストケースを用いて引数やフィールド変数の値を変えながら対象メソッドを解析し，データを取得する．それにより，比較的少ないコストで表明を生成できる．代表的な表明の動的生成ツールに Daikon などがある．一方で，表明の動的導出手法はテストケースにの質が生成表明に影響を与えるという問題がある．ここでテストケースとの質とは，テストケースがどれだけの実行範囲をカバーしているかを表す．この問題に対し，インバリアントカバレッジという指標が存在する．インバリアントカバレッジとはテストケースがどれだけプログラムの返り値にデータ依存関係を持つ命令文を実行するかを表す指標である．著者が所属する研究グループではインバリアントカバレッジの値が高いテストケースを静的解析器 ESC/Java2 の反例出力を用いて導出する手法を提案してきた．しかし，既存研究では導出したテストケース制約が本来求めるべき制約より狭いという課題点があった．これは既存研究では反例情報に基づいたテストケース制約の導出を行っているため起こる問題である．提案手法では SMT ソルバを用いてこの課題を解決する．まず，対象メソッドの PDG からインバリアントカバレッジを満たす実行パスを列挙する．その実行パスを SMT ソルバの文法に変換し，求解する．その解を用いて，そのパスの実行可能性と具体的な引数値を導出する．

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，提案手法が有用であることを確認した．

主な用語

表明，インバリアントカバレッジ，Daikon，SMT ソルバ，PDG

目 次

1 はじめに **1**

2 研究背景 **3**

2.1 Design by Contract 3
2.2 表明 3
2.3 表明の自動生成手法 4
2.3.1 静的手法 4
2.3.2 動的手法 4
2.4 ESC/Java2 5
2.5 インバリアントカバレッジ 6
2.6 PDG 8
2.7 既存手法 8
2.7.1 概要 8
2.7.2 問題点 9
2.8 SMT ソルバ 10
2.9 Z3 12

3 提案手法 **13**

3.1 提案手法概要 13
3.2 PDG から 1 実行パスの抽出 13
3.3 1 実行パスから SMT 言語への変換 14
3.3.1 変換補助関数 14
3.3.2 変数の変換 15
3.3.3 プリミティブ型の対応 15
3.3.4 式の変換 15
3.3.5 プリミティブ型のキャスト変換 18
3.3.6 配列の変換 18
3.3.7 文字の変換 18
3.3.8 クラスのフィールドの変換 19
3.3.9 メソッド起動式の変換 19
3.3.10 条件式の変換 20
3.3.11 変数宣言の変換 21
3.3.12 代入文の変換 21

3.3.13 if，while，for 文の変換 21
3.3.14 複文の変換 22
3.3.15 変換例 22
3.4 SMT ソルバを用いたテストケースの導出 22

4 評価実験 **24**
4.1 評価実験の目的 24
4.2 実行環境 24
4.3 適用対象 24
4.4 評価実験の手順 24
4.5 評価実験の結果 24
4.6 考察 26
4.6.1 今後の課題 26

5 あとがき **28**

謝辞 **29**

参考文献 **30**

1 はじめに

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．これにより，仕様理解の補助やプログラムの妥当性を確認できる．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・不変条件などを表明という形で記述し，静的解析ツール ESC/Java2 [2, 3] などを用いてプログラムの検証ができる．これによりプログラムの保守性や信頼性が向上する．しかし，プログラムのソースコード量の増大により，人手で表明をソースコードに記述するのは困難となる．そこで，表明の自動生成手法が注目されている．表明の自動生成手法には，静的生成手法と動的生成手法がある．表明の静的生成手法では，プログラムを静的に解析することで，表明を導出する．しかし，表明生成にかかる時間が長い点が問題となる．表明の動的生成手法では，テストケースを用いて引数やフィールド変数の値を変えながら対象メソッドを解析し，データを取得する．それにより，比較的少ないコストで表明を生成できる．これを解決するために，表明を自動で導出するツールとして Daikon [4, 5] や DySy[6] などがある．本研究では Daikon を用いている．動的生成手法では，テストケースがどれだけの実行範囲をカバーしているかというテストケースの質が生成される表明の質に影響を与える [7]．この問題に対し，インバリアントカバレッジ [8, 9] という指標が存在する．インバリアントカバレッジとはテストケースがどれだけプログラムの返り値にデータ依存関係を持つ命令文を実行するかを表す指標である．著者らが所属する研究グループでは，表明導出に利用するために有効なテストケースを自動で生成するために，インバリアントカバレッジを満たすテストケースの自動生成手法を研究してきた [10, 11, 12, 13, 14]．既存研究 [10, 11, 12, 13, 14] では，テストケース制約を静的解析器 ESC/Java2 の反例出力から取得しているため，本来求めたいテストケース制約よりも厳しい条件しか取得できないという問題がある．この問題点の解決のために提案手法では，SMT ソルバの Z3 を用いる．Z3 でプログラムの実行パスを実行するための条件を解くことにより，テストケースを得る．ここでパスの実行可能性も得ることができる．手順として，まず対象メソッドの PDG からインバリアントカバレッジを満たす実行パスを取得する．次にインバリアントカバレッジを満たすパスを SMT ソルバ [15, 16] の文法 SMT-lib2.0 Standard [17] 文法に準拠した制約式に変換する．そして SMT ソルバを用いてその式の解を導出し，その解を用いてテストケースを生成できる．また，本稿において提案した手法を実装した．そしてそのツール全体での出力の妥当性を検証するため，複数の 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．評価実験においては提案手法で求めたテストケースの有用性についての評価を行った．その結果，いくつかのメソッドに関して既存研究で生成されない有用なテストケースを導出することができた．

以降，2 章で研究の背景について述べ，3 章で提案手法について述べ，4 章で評価について述べ，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　表明

プログラムの表明は，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．Java プログラムにおいては，ソースコード中に表明の記述を行うことにより，DbC における契約を記述できる．これにより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．主な表明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．静的解析器を用いてプログラムの妥当性を検証することで，開発者の意図しない不具合の混入の防止もできる．

Java プログラムにおいて表明は，Java Modeling Language[18] 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 による表明の記述例を示す．

図 1 の例ではクラス JML_Example のメソッド sum に対して JML により表明を記述している．“requires” が事前条件，“ensures” が事後条件，“maintaining” がループ不変条件を表している．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，ループ不変条件として，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 による表明の記述例

2.3 表明の自動生成手法

ソースコードサイズの増加に伴い，人手による表明記述は困難になる．そこで，表明の自動生成手法が注目されている．表明生成の自動化手法に，静的手法と動的手法の 2 種類がある [19]．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

表明の静的自動導出手法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，表明を生成する．そのため，精度の高い表明を自動生成できる．Houdini[20] は静的解析器 ESC/Java2[2] を繰り返し適用することにより，表明を生成する．しかし，一般的に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

表明の動的自動導出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表明を生成する．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 つからなる手続きである．テストケースが特定の実行パスを通らないなど品質が低い場合，生成される表明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．一方でテストケースの品質が高い場合は比較的少ない時間，メモリで表明を生成できる．そのため，動的手法は表明の自動生成に用いられることが多く，その代表的

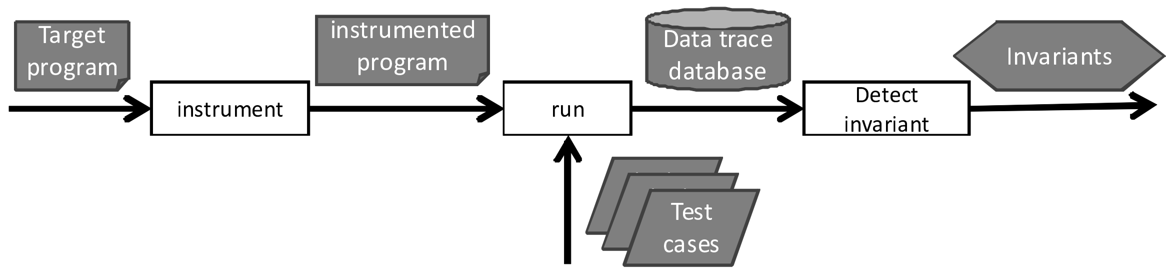

図 2: Daikon の処理

ツールとして Daikon[4, 5] 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と出口 (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 が持つ表明パターンとを照合して各プログラムポイントにおける表明を自動生成し，出力する．また，このとき独自の表明推測アルゴリズムを適用することで再現率，適合率が高い表明の生成を可能にしている [4]．

Daikon を用いることにより，手作業で表明を記述するよりも表明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ため，プログラマがソースコード記述時には気づかなかった表明を生成することもできる．これはプログラムの保守，デバッグにも有効である．さらに，Daikon は生成した表明を JML 形式など様々な形式で出力できるため，JML をコメントとして元の Java プログラムに挿入することにも使え，機能面においても充実している．同様のツールとして DySy [6] などがある．

2.4 ESC/Java2

ESC/Java2[2] は Java プログラムに対する静的検証器である．ESC/Java2 は Java ソースコードを入力とし，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 などで記述された表明と Java ソースコード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．

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 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

```
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) ;
    }
                                                              10
    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−−;                                                   20
    elements[minIndex]= elements[size] ;
    return min ;
    }
}
```

図 3: ESC/Java2 への入力例

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　インバリアントカバレッジ

インバリアントカバレッジ (Invariant Coverage)[8, 9] は，表明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．このカバレッジに基づいてテストケースを生成することで，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，表明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となる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は一般的な表明の生成に必要な全実行パスに対して，テストケースが実際に実行する実行パスの割合を示す．文献 [8] 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．

```
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] ;
                                    ^                                           20
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.
                                                                                30
------------------------------------------------------------------------
```

図 4: ESC/Java2 の出力例

[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．

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

2.6 PDG

プログラム依存グラフ (PDG) 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 [24]．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．また，PDG に含まれるデータ依存辺をたどることにより DUC を算出することができる．提案手法では PDG に含まれるデータ依存辺から DUC を算出し，DUC に実行時の制約を付加した制約付き DUC を生成する．これにより提案手法ではプログラムの実行パスを実行する条件を導出する．

2.7 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 [10, 11, 12, 13] の概要と問題点について述べる．

2.7.1 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. 対象メソッドの関数内プログラム依存グラフ（以降，PDG という）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 で記述した表明を元の Java ソースコードに挿入する．

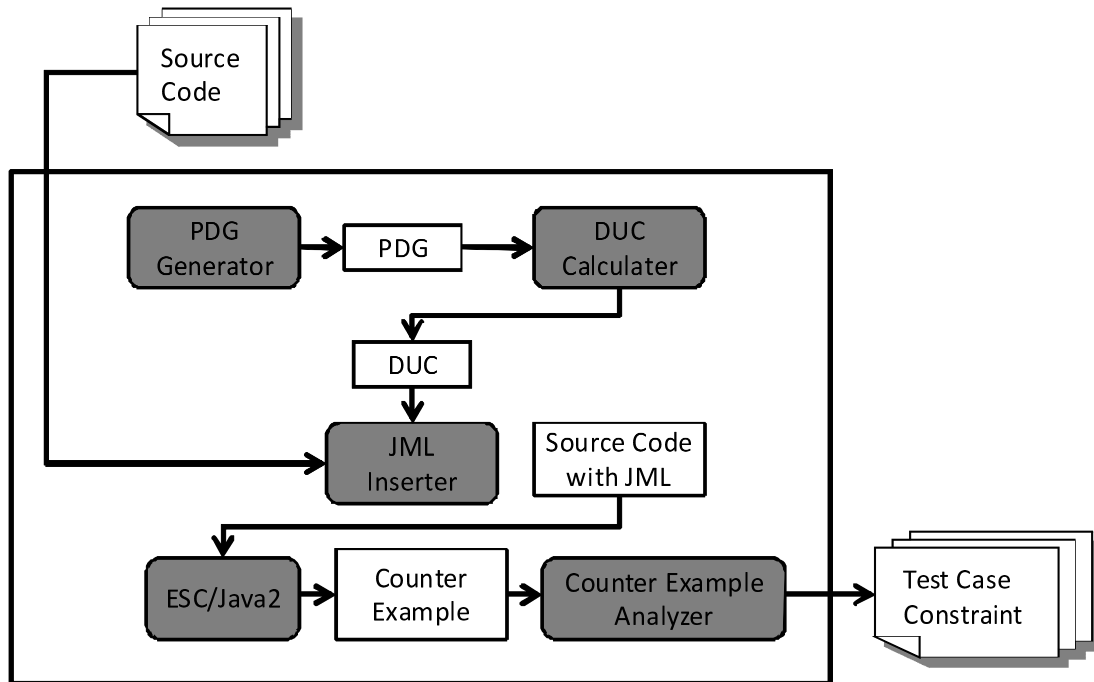

図 5: 既存手法による処理の概要

4. ESC/Java2 を実行し，反例を取得する．ESC/Java2 は手順 3 で挿入した表明以外に対する警告や反例も出力するが，既存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．

手順 4 の ESC/Java2 による反例の取得と手順 5 の反例の解析がこの手法で中心である．詳細は文献 [11] を参照されたい．

2.7.2 問題点

既存手法には以下の問題点がある．

1. 反例情報から取得できるテストケース制約は本来のテストケースより厳しい制約である．

2. Java1.5 以降の文法に対応できない．

3. ESC/Java2 の反例文法が明確に定義されていない．

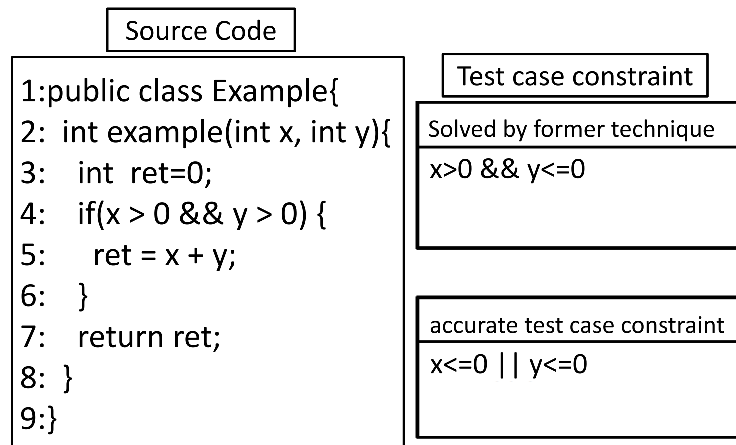

図 6: 4 行目の条件式が成り立たない実行パスのテストケース制約

本研究では問題 1 について解決する手法を提案する．この問題はテストケース制約を反例より求めているため生じる問題である．ESC/Java2 で実行パスを実行した時の制約を出力させる場合，反例を用いることで実現が可能である．しかし，一般的に検査における反例は検査に不合格となる条件の一部分を表すことで十分である．したがって，“指定した実行パスを通ることはない” という制約を与えた場合，得られる制約は，“指定した実行パスを通る制約の一部” となる（図 6）．

得られるテストケース制約が本来求めたい制約より狭くなることによって，インバリアントカバレッジを満たすテストデータを得ることが難しくなる．既存手法では，テストデータはランダムに生成し，それをテストケース制約によって絞り込む操作を行なっている．これによりインバリアントカバレッジを満たすテストケースを得ている．しかし，テストケース制約が本来求めたいテストケース制約より狭くなることで，テストデータの網羅性が低くなるおそれがある．これはインバリアントカバレッジを満たすテストケースを生成するとき，問題である．

2.8 SMT ソルバ

既存手法の問題点を解消するため，提案手法ではSMT ソルバを利用する．SMT(Satisfiabilitu Modulo Theories)[16] は特定の分野の数学的な体系を表す理論である．例えば，算術の理論，集合の理論などがある．SMT では述語論理の充足可能性判定を行う．SMT ソルバはSMT の制約を満たす解を求める静的解析器である．SMT ソルバに問題とその制約を変数や関数

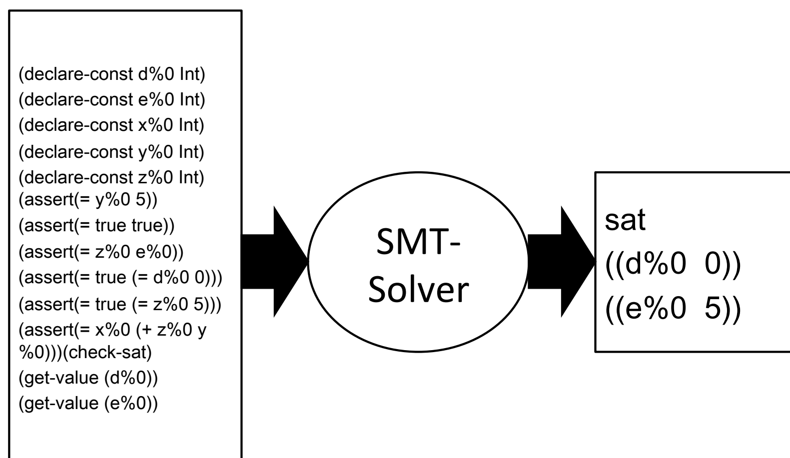

図 7: SMT ソルバによる求解

に関する制約式として入力する．そして，SMT ソルバは充足可能性を判定しその解と充足する場合の変数への値の割り当てを出力する（図 7）.

SMT ソルバには様々な実装があり [16]，それぞれに扱える論理が異なる．幾つかの SMT ソルバでは一階述語論理の充足可能性判定問題を解くことができる．本研究では Java プログラムの実行パスを実行する条件を求めるために SMT ソルバを利用する．SMT ソルバに Java プログラムの実行パス上の DUC と，その DUC を実行するための条件を変数間の関係式や制約条件として入力する．SMT ソルバは DUC による変数間の関係式や制約条件を満たす変数の割り当てがあるかどうか検査する．満たす変数割り当てがあれば解とその変数割り当てを出力する．変数割り当ては Java プログラムの実行パスを実行するときの入力値であるテストケースとすることができる．

Java プログラムの実行パスは PDG より制約付き DUC として導出し，SMT ソルバの入力である SMT-LIB へ変換する．SMT-LIB は SMT ソルバの競技会の標準の入力形式であり，この入力形式に対応することが事実上の標準となっている．なお，通常充足可能性判定問題の解といった場合は，充足の可否だけでなく，充足する場合の変数への割り当て方の情報も含む.

2.9 Z3

Z3[21] は，Microsoft 社が開発した SMT ソルバ [16] である．自社で開発するプログラム解析，テスト，検証など様々なツール内で Z3 を利用している [11]．Z3 は本研究で用いる SMT ソルバであり，線形の実数や整数の算術や配列，レコードの表現など多様な機能を持ち，また解の導出速度が速く正確である．文法は SMT-lib2.0standard [17] 言語 (以下，SMT 言語という) に従う．

3 提案手法

2.7.2 小節であげた課題を解決するため，テストケース生成手法に SMT ソルバ Z3 を用いる．SMT ソルバでは反例として制約を導出せず，パスの分岐条件そのものを用いてテストケースを求めるため，得られた制約が狭くなることはない．SMT ソルバを用いたテストケースの導出のため，入力として SMT 言語で書かれた，制約を満たすための式を与える必要がある．そこで，Java の実行パス候補から SMT 言語で書かれたパス実行条件の式への変換を実装した．これにより，既存研究のテストケース生成手法を改善できる．なお本提案手法では，既存手法と異なりテストケース制約ではなくテストケースを直接求めることができる．

3.1 提案手法概要

入力:Java の 1 メソッド
出力:テストケース

1. 対象メソッドのプログラム依存グラフ (PDG) を生成する．
2. 手順 1 で生成した PDG よりインバリアントカバレッジに影響を及ぼす 1 実行パスを取得する．
3. 手順 2 で得られたパス集合を実行するための条件式を SMT 言語の文法に変換する．
4. SMT ソルバに入力し，パス実行条件を満たす変数割り当てを求める．
5. (2)-(4) をインバリアントカバレッジに影響を及ぼす全ての実行パス候補分繰り返す.
6. 変数割り当てからテストケースを求める．

3.2 PDG から 1 実行パスの抽出

まず対象メソッドに対し，MASU[22] を用いて PDG を生成する．PDG を利用し，インバリアントカバレッジに影響を及ぼすすべての実行パスを取得する．実行パスの取得法は次の通りである．まず PDG から，返り値にデータ依存をするノードを取得する．次に変数定義が制御依存する式を取得する．制御依存する式中で使用されている変数に関しても，データ依存・制御依存を取得する．取得した変数定義を組み合わせて選択することで実行パス候補を導出する．ただし，これらの実行パスは PDG から抽出したものであるため，実際のそのパスの実行可能性は，パスを抽出した時点では不明である．

3.3　1 実行パスから SMT 言語への変換

入力を Java の 1 実行パスとし，その実行パスを通るための変数割当を求める SMT 言語の式とする．本節では，変換規則の一部について説明を行っていく．変換は抽出された実行パスの系列に対し行う．以下, 変換規則 μ を次のように与える $\mu : EP \rightarrow ES$

ここで EP，ES はそれぞれ，PDG の 1 ノードに相当する Java の 文，SMT 言語への部分式である．

3.3.1　変換補助関数

変換に当たり，以下の補助関数を用いる．

[定義 3.1] String name(String variable)
補助関数 $name$ とは，変数名 $varibale$ を引数にとり，変数名にインクリメントされた ID をつけたものを返す．なお，ID は変数が更新された回数をあらわし，各変数ごとに ID を持つものとする．ID の初期値は 0 である．すなわち，i 回値が更新されている変数が，$name$ 関数により呼び出されたときは，$variable\%i + 1$ を返す．

[定義 3.2] String use(String variable)
補助関数 use は，変数名 $variable$ を引数にとり，変数名に ID をつけたものを返す．ID は変数 $variable$ が更新された回数を表す．i 回値が更新されている変数が，use 関数により呼び出されたときは，$variable\%i$ を返す．

[定義 3.3] String type(String Java_type)
補助関数 $type$ は，Java での変数の型 $Java_type$ を引数にとり，それに対応する SMT 言語での型を返すものとする．

[定義 3.4] int class(String className)
補助関数 $class$ は，クラス名を表す変数 $className$ を引数にとり，クラスごとに定義した固有の整数値を返す．

[定義 3.5] int point(String Instance)
補助関数 $point$ は，インスタンスを表す変数 $Instance$ が何番目にインスタンス化されたかを返す関数である．
インスタンスが参照しているメモリを区別するために用いる．

3.3.2 変数の変換

SMT 言語では代入という概念が存在せず，式は等式か不等式でしか表すことができない．よって SMT 言語では同一の変数名であっても，変数の後に連番をつけることで区別をする．そこで前述の補助関数 $name$, use を用いる．

3.3.3 プリミティブ型の対応

Java プログラムにおいて，定義される変数は SMT-LIB では表 1 のように表す．SMT で定義されている基本型は 3 種類のみである．

3.3.4 式の変換

本節では代入以外の式の変換方法について説明する．Java 演算子の優先順位に従って，前置記法で表現する．例えば $1 + 2 * x$ のような変換は演算子の優先順位にしたがい,
$(+\ 1\ (*\ 2\ use(x)))$
のように変換される．
演算子の変換を示す．ただし，本稿ではビット演算に対しては適用対象外とした．これは，SMT ソルバ Z3 において整数型からビット型への変換が存在しなかったからである．以下，表の左側を変換規則 μ の引数，表の右側を変換規則 μ の返り値とし，式中で Java の演算子の優先順位に従って再帰的に変換規則を適応していくものとする．すなわち，優先順位の高い演算子から変換を行い，次の優先順位の演算子に対し変換を行っていく．この操作を全ての演算子が変換されるまで行う．2 項演算子の変換を表 2 に示す．なお 2 項演算子の左側の部分式を L，右側の部分式を R とする．単項演算子の変換を表 3 に示す．なお前置演算子の被演算子を R とし，後置演算子の被演算子を L とする．また，単項の変換を表 4 に示す．この時 16 進数や 8 進数の項も全て 10 進数の整数に変換する．

ただし，前置演算子 ++ に関しては,

```
(declare-const name(R) type(R))
(assert (= (+ use(R) 1))
```

前置演算子 −− に関しては,

表 1: Java 基本型と SMT 基本型の対応

Java の基本型	byte	short	int	char	long	float	double	boolean
SMT 基本型	Int	Int	Int	Int	Int	Real	Real	Bool

```
(declare-const name(R) type(R))
(assert (= (- use(R) 1)
```

を出力した後に変換を行うものとする．
後置演算子 ++ に関しては，変換後に

```
(declare-const name(L) type(L))
(assert (= (+ use(L) 1)
```

を出力する．
後置演算子 −− に関しては，変換後に

```
(declare-const name(L) type(L))
(assert (= (- use(L) 1)
```

表 2: 2 項演算子の変換

Java	SMT
$L + R$	$(+\ L\ R)$
$L - R$	$(-\ L\ R)$
$L * R$	$(*\ L\ R)$
L/R (整数型の場合)	$(div\ L\ R)$
L/R (実数型の場合)	$(/\ L\ R)$
$L\%R$	$(mod\ L\ R)$
$L\&R$	$(and\ L\ R)$
$L\&\&R$	$(and\ L\ R)$
$L\vert R$	$(or\ L\ R)$
$L\vert\vert R$	$(or\ L\ R)$
$L\hat{}R$	$(xor\ L\ R)$
$L == R$	$(=\ L\ R)$
$L! = R$	$(not\ (=\ L\ R))$
$L > R$	$(>\ L\ R)$
$L < R$	$(<\ L\ R)$
$L >= R$	$(>=\ L\ R)$
$L <= R$	$(<=\ L\ R)$
$L\ instanceof\ R$	$(=\ (\%class\ use(L))\ class(R))$

を出力する．

なお，3 項演算子 $L?C : R$ の変換は，

```
(declare-const name(sankou) type(C))
(assert (xor (and L (= C use(sankou))) (and (not L) (= R use(sankou)))))
```

という文を出力した後，$use(sankou)$ と変換する．ここで $sankou$ とは三項演算子の返り値

表 3: 単項演算子の変換

Java	SMT
$+R$	R
$-R$	$(-\ R)$
$++R$	$use(R)$
$--R$	$use(R)$
$!R$	$(not\ R)$
R	$(-\ (-\ R)\ 1)$
$L++$	$use(L)$
$L--$	$use(L)$

表 4: 単項の変換

Java	SMT
$integer$	$integer$
$realNumber$	$realNumber$
$character$	$CodeNumber$
$variable$	$use(variable)$
$String$	$CodeNumberArray$
$true$	$true$
$false$	$false$

$integer$:整数
$realNumber$:実数
$character$:文字
$variable$:変数
$String$:文字列
$CodeNumber$:文字コードを表す整数
$CodeNumberArray$:文字コードを表す整数の配列

Java: integerExpression; SMT: (to_real integerExpression)
Java: realExpression; SMT: (to_int realExpression)

図 8: プリミティブ型のキャストの表現

を表す変数である．

3.3.5 プリミティブ型のキャスト変換

Java 言語では整数と実数の演算が可能であるが，SMT 言語では整数と実数の演算は不可能である．例えば Java では 1+1.0 などの計算は自動でキャストが行われるが，SMT 言語では 1+1.0 のように型の違う演算を行うとエラーを出力する．そこで，整数-実数間のキャストを行う SMT の関数 to_real，to_int を用いる．整数の部分式 $integerExpression$, 実数の部分式 $realExpression$ はそれぞれ図 3 のように変換を行う．図 8-9,11-13 では上段を変換規則 μ の引数とし，下段を変換規則 μ の返り値とする．

3.3.6 配列の変換

図 9 に 1 次元配列の宣言と使用, 代入についての変換を示す．ただし，配列の要素数についてはプログラム実行時に動的に決定されるため，ここでは変換を定義しないものとする．また，SMT 言語では図 9 の 6 行目の $Array$ の後に続く Int という記述の数により配列の次元数を変えることができる．$Array$ の後に Int を i 個記述すれば i 次元配列を表現可能である．

3.3.7 文字の変換

Java 言語では文字は文字コードを用いて整数として表すことができる．よって文字を表す char 型に関しては整数型 Int を用いて表す．String 型も同様に整数型の配列として表現する．

```
Java:
array_Type array_Name;
a[i];
a[i]=expression;
```

```
SMT:
(declareconst name(array_Name) Array Int type(array_Type))
(select use(a) i)
(assert (= (store use(a) i expression) name(a)))
```

図 9: 配列の変換例

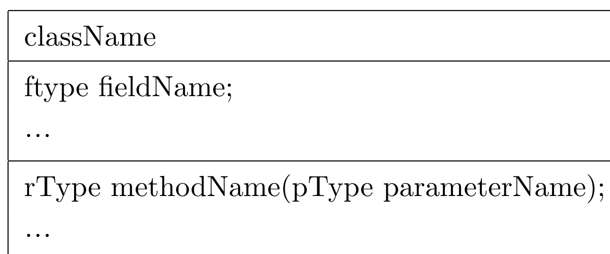

図 10: 定義対象クラスの UML 表現

3.3.8 クラスのフィールドの変換

まずクラスのフィールドを表すために，そのフィールドを持つレコードを定義しておく．図 10 のクラスがあるとき，インスタンス化とフィールドの参照の変換方法は図 11 のようになる．ここで，*%pointer* はインスタンスが参照している場所，*%class* はクラスの種類を表すものとする．

本変換により，既存の研究では表現できなかったクラスに対してもテストケースが導出できる．

3.3.9 メソッド起動式の変換

式中でメソッドを呼出したときの変換方法について述べる．メソッドの返り値に関しては，厳密な値を求めようとすると，状態爆発を起こす場合がある．そのためメソッドの返り値は自由な値をとるものとする．void 型のメソッドに関しては，変換しない．図 10 のクラスのメソッド *methodName* の変換は，図 12 のようになる．

この変数 *name*(*methodName*) を式中に使うことで，メソッド起動式の返り値を表現する．既存研究ではうまく定めることができなかったメソッド起動式の返り値を本手法では求

Java:
className oName = new className();
oName.ftype;
SMT:
(declare-datatypes () ((mk-obj (field (fieldName type(ftype))
(%pointer Int) (%class Int)))))
(declare-const name(oName) type(className))
(assert (= (%pointer use(a)) point()))
(assert (= (%class use(a)) class(className)))
(fieldName use(oName))

図 11: インスタンス化の変換

Java:
method_Name(parameter_Name)
SMT:
(declare-const name(methodName) type(method_type))
(declare-const name(instance) type(className))

図 12: メソッド起動式の変換

解できる．またこのメソッドを呼出したインスタンス *instance* はメソッドによりフィールド値が変更される場合があるため，インスタンスそのものも更新する．

3.3.10 条件式の変換

条件文を *condition* としたときに，PDG 上では条件文のみが表示される．その条件が真となるパスを通るとき，

```
(assert  condition)
```

条件が偽となるパスを通るときは，

```
(assert (not condition))
```

と変換をおこなう．

Java:
variable_type variable;
variable = Expression;
SMT:
(declare-const name(variable) type(variable_type))
(declare-const name(variable) type(variable_type))
(assert (= use(variable) Expression))

図 13: 代入文，宣言文の変換

複合演算子
$variable\ ope =\ Expression$;
複合演算子のノーマライゼーション後:
$variable\ =\ variable\ ope\ Expression$;

図 14: 複合演算子のノーマライゼーション

3.3.11 変数宣言の変換

変数名を $variable$，変数の型を $variable_type$ とすると，変数宣言は図 13 のように変換を行う．

3.3.12 代入文の変換

変数名を $variable$，変数の型を $variable_type$ とし，代入する式を $Expression$ とすると，代入文は図 13 のように変換する．ただし，複合演算子 $ope =$ については，図 14 のように変換前にノーマライゼーションを行う．ope は演算子を表す．

3.3.13 if，while，for 文の変換

if 文は PDG 上では分岐条件のみ与えられ，その分岐条件が真となるパス，偽となるパスをそれぞれ取得することで対応する．while 文と for 文はユーザが指定した任意の整数回だけループするようにループを展開し，そのパスそれぞれに対して変換を行うことで対応する．

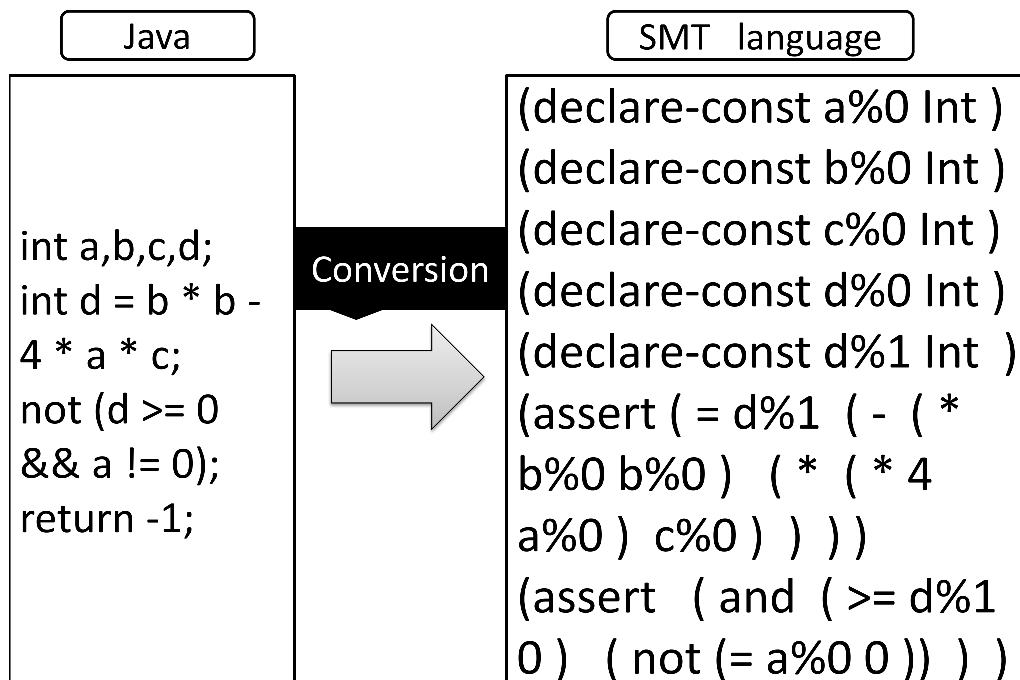

図 15: 2 次方程式の解判定の変換

3.3.14 複文の変換

PDG から得た実行パス系列に対し，本稿で示してきた変換規則 μ を各文ごとに順に適応していくことにより，複文全体を変換することが可能になる．

3.3.15 変換例

変換例を図 15 に示す．この変換は 2 次方程式 $ax^2 + bx + c$ が実数解を持たないときの変数の割り当てを求める SMT 言語の制約式へと変換を行っている．なお，return 文は変換しない．

3.4 SMT ソルバを用いたテストケースの導出

SMT 言語で表現された制約を表す式を入力として SMT ソルバに与え，満たす割り当てを求解する．この処理により，パスの実行可能性と，実行可能な場合の変数への割り当てを求めることができる．SMT ソルバの式に，引数が以前求めた値と異なるという条件を加えて繰り返し実行することにより，同じ実行パスを通る複数の異なる変数の割り当てを求める

ことができる．これは与えるテストケースの数が多いほど Daikon では質の高い表明を導出できるためである．

4 評価実験

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本章にて示す Java プログラムに対して適用した．本ツールを実行時間及び出力結果の妥当性の観点から評価を行う．

4.1 評価実験の目的

本提案手法が実プログラムにおいて適用可能な範囲や，テストケースがインバリアントカバレッジを満たすために妥当であるかを様々なプログラムを対象に適用実験を行った．

4.2 実行環境

実行環境には，`Intel(R) Core(TM)2 Duo CPU U9300 1.20GHz`
`Memory:2.00GB,Windows Vista,JDK1.6_17` および `Z3-3.2` を使用した．

4.3 適用対象

本稿における実験対象を表 5 に示す．これらのソースコードの一部は本研究グループの行った既存研究 [11] でも実験対象としている．また，既存手法では本来より狭いテストケース制約を導出する実験対象となっている．

4.4 評価実験の手順

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．以下の観点から評価する．

1. 対象メソッドに対して本ツールを適用し，所定のパス実行条件を満たす変数の割り当てを求める．その際解を得るまでの実行時間を取得する．
2. 提案手法から得られたテストケースのインバリアントカバレッジを調べる．
3. 提案手法で導出したテストケースに既存手法から得られないテストケースがあるか調査する．

なお本実験において，for 文や while 文などのループの展開回数は 2 回とした．

4.5 評価実験の結果

表 6 の結果が得られた．実行時間は PDG を生成してから，SMT ソルバが全実行パスに対して求解を終えたときまでの時間を計測している．テストケース数は本手法で導出するこ

とができたテストケースの数であり，この際同一の複数個のテストケースに関しては一つのテストケースとして数えている．また，PDG の情報から実行パス候補数や平均パス長なども求めている．実際のテストケースを用いてプログラムを動かすことにより，インバリアントカバレッジを求めた．

表 5: 実験対象となるソースコード

プログラム名	行数	実行パス候補数	平均パス長
Calc	21	2	3.5
Telephone	29	7	5.5
Flower	28	41	9.5
JanCord	26	3	13
Operator	21	6	3.7
RPS	19	5	5
Date	28	9	5.4

Calc:2 次方程式の求解
Telephone:電話番号がどの種別を表すかの判定
Flower:窓辺の花 [23] のプログラム
JanCord:8 桁のバーコードチェックディジットが正しいかを判定する
Operator:演算子を表す文字を与えたとき，対応する演算を行う
RPS:じゃんけんの判定
Date:二つの日付のうちどちらが新しいかの判定

表 6: 実験結果

プログラム名	実行時間 (秒)	インバリアントカバレッジ	テストケース数
Calc	28.6	100	40
Telephone	23.4	92	26
Flower	631.5	100	29
JanCord	161.2	100	44
Operator	25.9	100	38
RPS	26.4	100	34
Date	29.8	100	160

4.6 考察

提案手法で生成したテストケースが既存手法で導出できるかについて考察を行う．例として，flower というプログラムでは既存手法では $0 \leq num \leq 3$ の否定から $num \geq 4$ というテストケース制約を得られたが，こらは $num < 0$ というテストケース制約が抜けているため，制約としては不十分であり，Daikon で正しく表明を生成できない．しかし，提案手法では，$num = -1, a[0] = true$ など既存手法では生成できないテストケースを得られた．このテストケースは表明の生成をするうえで有用である．そしてテストケースを得るときは Z3 のオプションを用いてランダムな値を求解するように設定可能なため，偏ったテストケースが生成されることは起こらない．Telephone のインバリアントカバレッジが 100%でないのは，提案手法ではメソッドの返り値を任意の値を返すと定義していたり，また配列の長さを SMT では求めることができないためである．これらは今後の課題として対応法を検討していく必要がある．実行時間に関しては現実的な実行時間でテストケースを生成することができたといえる．実行時間は PDG の生成と SMT ソルバの求解が大部分を占めている．本手法では実行時間はパス数や生成したテストケース数，平均のパス長に依存している．時間はかかるが，実行パス数の多いプログラムでも対応可能であるといえる．テストケース制約の広さの点から，提案手法がよりメソッドの状態を詳しく表現するために必要な，テストケースを生成可能であるといえる．

4.6.1 今後の課題

提案手法では以下の変換規則については未定義となっている．

1. 列挙型の変換
2. 総称型の変換
3. クラスのキャスト
4. アサーション

また，提案手法，既存手法どちらとも String 型以外の参照型のテストケースを生成することはできないため，それらの課題に対応していく必要がある．参照型のテストケースを出力するために必要なこととして，Z3 の出力結果を解析し，そのデータからテストケースを生成する必要があげられる．現在 Z3 の出力結果の解析は基本型と配列に対してしか行っていないため，参照型を表すレコード型の出力結果の解析が可能となれば，参照型のテストケースを導出できる可能性があると考えられる．また，現在メソッド起動式に関しては，そのメソッドと同じ型の任意の値を返すものとして定義している．これはメソッドを再帰的に呼び

出すなどの操作で起こる，状態爆発などを防ぐためのものであった．しかし，実際にはメソッドが本当に任意の値を返すわけではない．この問題に関しては，メソッドに記述された事前条件と事後条件を解析し，これをSMT言語に変換することで対処することが可能であると考えられる．すなわち，メソッドの事前条件を満たすような引数をとり，メソッドの事後条件を満たすような任意の値を返す値をとる数として定義すればよいと考えられる．これらの課題に対し変換を行うことができれば，参照型のテストケースを生成することが可能となるであろう．

5　あとがき

本研究では我々の研究グループが提案している，表明自動導出手法のためのテストケース生成手法の提案とその評価実験を行った．既存手法では静的解析器 ESC/Java2 を利用してテストケース制約を導出していた．しかし，反例を利用したテストケース制約の導出手法であるために求まるテストケース制約が本来求めるべき理想的なテストケース制約より狭いという課題点があった．提案手法では，SMT ソルバや，PDG を利用してこの問題を解消している．まず，対象となるメソッドから PDG を生成する．そして PDG からインバリアントカバレッジを満たす実行パス候補を抽出する．その実行パス候補を実行するための変数への割り当てを SMT ソルバを用いて導出する．この際 PDG から得られた実行パス候補を SMT ソルバの文法に変換し，それを入力として SMT ソルバに与えることで割り当てを求める．この操作を異なる複数のテストケースを導出するため，繰り返し行う．この一連の操作により課題点を解消し，本来求めるべき理想的なテストケース群を得ることができる．提案手法を複数の Java プログラムを対象に適用したところ，既存手法では導出できなかった有用なテストケースを導きだすことができたことを確認した．しかし，適用クラスの狭さが課題点である．今後は適用可能なクラスを拡張し，その評価実験を行うことで本手法の有用性を評価していきたい．

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 井垣 宏 特任准教授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 肥後 芳樹 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 小林 和貴 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] D. L. Detlefs, K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. *Compaq SRC Research Report*, 1998.

[4] M. D. Ernst, J. H. Perkins, S. McCamant, P. J. Guo, 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.

[5] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[6] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. *in Proc. of 30th ACM/IEEE Int. Conf. on Software Engineering (ICSE)*, pp. 281–290, 2008.

[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 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.

[9] 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.

[10] 小林和貴, 宮本敬三, 岡野浩三, 楠本真二. 表明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. ソフトウェア工学の基礎 XVII 日本ソフトウェア科学会ソフトウェア工学の基礎ワークショップ FOSE2010, pp. 35–44, 2010.

[11] 宮本敬三, 堀直哉, 岡野浩三, 楠本真二. Daikon 生成表明改善のためのテストケース自動生成手法とその評価実験. 日本ソフトウェア科学会誌コンピュータソフトウェア, vol.28, no.4, pp.306–317, 2011.

[12] 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 X 日本ソフトウェア科学会ソフトウェア工学の基礎ワークショップ FOSE2008, pp. 41–50, 2008.

[13] 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.

[14] 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.

[15] L. D. Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, Vol. 54, pp. 69–77, September 2011.

[16] 梅村晃広. SAT ソルバ・SMT ソルバの技術と応用. コンピュータソフトウェア, Vol. 27, No. 3, pp. 24–35, 2010.

[17] D. R. Cok. *The SMT-LIB v2 Language and Tools: A Tutorial*, *GrammaTech*, 2011.

[18] 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.

[19] 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.

[20] 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.

[21] L. D. Moura and N. Bjørner. *Z3: An Efficient SMT Solver Tools and Algorithms for the Construction and Analysis of Systems*, Vol. 4963/2008 of *Lecture Notes in Computer Science*, chapter 24, pp. 337–340. Springer Berlin, Berlin, Heidelberg, April 2008.

[22] 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.

[23] 産業技術総合研究所システム検証研究センター. 4 日で学ぶモデル検査 (初級編) (CVS 教程 (1)). エヌ・ティー・エス, 2006.

[24] 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.