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

Improving Daikon's output with Automated Generation of Test Suites based on Invariant Coverage and Model Checking Techniques

堀　直哉* 岡野　浩三† 楠本　真二‡

Summary. Daikon dynamically detects program invariants. Considering its effectiveness and outputs' qualities, Daikon is a useful tool. It, however, has a problem that its outputs depend on the test suites used for the dynamic analysis. This paper proposes an automated generation method of the test suites based on invariant coverage and model checking techniques. Applying the method to an example shows that the proposed method improves quality of the test suites as well as Daikon's generating invariants.

1　まえがき

Design by Contract(以降，DbC とする) [1] は，仕様を決定する段階でメソッドやクラスの制約を決めておくことで，プログラムの信頼性や再利用性を向上させる．また，プログラムの表明 (assertion) 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 に基づいてプログラムの表明を記述することで，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．近年のソースコードサイズの増加にともない，既存コードに対して手動による表明生成は困難になってきている．そのため，表明の自動生成手法や自動検査手法が注目されている．

表明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 種類のアプローチがある [2]．静的手法 [3–9] 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を求めるため，精度の高い表明の自動生成 [3–6] や自動検査 [7–9] 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．一方，動的手法は比較的少ない時間，メモリで表明の生成が可能であるが，得られる実行データが少ない場合，自動検査には不向きである．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，その代表的ツールとして Daikon [10, 11] がある．しかし，表明の動的生成手法の問題点として，生成される表明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 [12]．

本研究では，表明を動的に自動生成する手法に着目し，そのテストケース依存問題を改善するため，インバリアント被覆 [13]，モデル検査器 Java PathFinder [8, 9]，記号実行 [14, 15] の技術を活かした新手法を提案する．

インバリアント被覆 [13] とは，動的生成ツールに用いるテストケースの有用性を測定するために提案されたカバレッジである．この値が高いとき，動的生成ツールは信頼性の高い表明を生成できる．文献 [13] ではインバリアント被覆のために定義–使用連鎖を提案している．この連鎖を通る実行パスはテストケース生成に有用であるが，そのようなテストケース生成法については言及していない．

Java PathFinder(以降，JPF とする) [8, 9] は，Java のソースコードが指定した条件を満たすかどうかを判定するモデル検査器である．JPF は指定した条件を満たさない場合にその反例となる実行パスを求めることができる．記号実行 [14, 15] は実

*Naoya HORI, 大阪大学大学院情報科学研究科
†Kozo OKANO, 大阪大学大学院情報科学研究科
‡Shinji KUSUMOTO, 大阪大学大学院情報科学研究科

行パスからその実行パスを通るために必要な入力の条件を算出する．よって，本提案手法はJPFで取得した実行パスに対して記号実行を行い，その結果生成された入力の条件を用いることで，テストケースの自動生成を可能としている．

本稿は文献 [13] で提案されているインバリアント被覆に基づくテストケースの定義をもとに，さらに記号実行などの既存の概念やJPFなどのツールを用いてテストケースの生成を自動化することでDaikonの出力を改善する手法を新たに提案する．文献 [6] は本論文と同様に記号実行を用いて表明生成を行っているが，[6] は表明を静的に生成しているのに対して，本手法は表明の生成に動的生成を用いている．

本手法で生成されたテストケースと表明を，既存のカバレッジに基づいたテストケースと表明とをそれぞれインバリアント被覆，信頼性という観点で評価した．その結果，テストケース，表明ともに本手法で改善されたことがわかった．

以降，2章で本研究が対象とする問題について述べ，3章で本研究に関連するツールや概念について述べ，4章で提案手法と例題への適用結果を示し，5章で考察を行い，6章でまとめる．

2　テストケース依存問題とその改善方法

本章では，本研究が対象とする問題とその改善方法の概略について述べる．

2.1　テストケース依存問題

表明の動的生成手法における課題としてテストケース依存問題がある [12]．動的生成手法は対象のソースコードをテストケースを用いて実行し，そこで得たデータを解析することで表明を生成する．この実行を行う際に用いるテストケースが生成される表明に大きな影響を与えてしまうのがテストケース依存問題である．

2.1.1　動的生成に適したテストケース

動的生成に適したテストケースを対象のソースコードが実行し得るパスを全て実行するテストケースとする．これは，特定のパスを実行しないと，テストケース依存問題が発生する可能性があるためである．

このような考えで提案されたカバレッジにインバリアント被覆 (Invariant Coverage) [13] がある．インバリアント被覆は対象のソースコードが実行し得るパスを求め，その必要なパスをテストケースによってどの程度実行されるかでテストケースの有用性を測定するもので，このカバレッジに基づいて生成されたテストケースは有用なものであることが示されている [13].

2.1.2　改善方法の概略

本提案手法では，モデル検査技術を用いて対象のコードが実行し得るパスとそのパスを通る入力の条件を求め，その条件からテストケースを自動で生成することで，テストケース依存問題を改善する．表明の動的生成ツールとしてDaikon, モデル検査器としてJava PathFinder [8,9] を用いた．また，パスから入力の条件を求める方法として記号実行 [14, 15]，入力条件の簡略化を行うツールとしてSimplify [19] を用いている．

3　準備

本章では，本研究に用いる，あるいは，対象にするツールの概要やインバリアント被覆などの有用な概念について簡単に述べる．

3.1　Daikon

Daikon [10, 11] 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．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．

3.2　インバリアント被覆

テストケース依存問題を改善するためには，表明の動的生成に用いるテストケースは全ての実行パスを通るものである必要がある．インバリアント被覆 [13] は一般的な表明の生成に必要な実行パスに対するテストケースが通る実行パスの割合を示す．文献 [13] 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 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 ペア *(*以降，DU ペアとする*)* 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 DU ペアの *(*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 DU ペアの 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 DU ペア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 DU ペアの繰り返しを許し，その場合，以下のように表記する．なお,「||」は直前の DU ペアの繰り返しを意味する．

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

定義 2　インバリアント被覆

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被覆 C_{Inv} の値は式 *(1)* で定義される．

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

以降，インバリアント被覆に基づいて作成したテストケースをインバリアント被覆テストケースと呼ぶ.

3.3　Java PathFinder

JPF [8,9] は Java のソースコードを対象としたモデル検査器である．JPF はソースコードとそのソースコードが満たすべき条件を記述したプロパティファイルを入力とし，ソースコードがプロパティファイルに記述された条件を満たすかどうかを判定し，満たさない場合に反例としてその実行パスを出力する．ここでメソッドの実行パスとは，メソッドの引数の取得から条件に違反するまでに実行するソースコード中の命令文の系列である．

3.4　記号実行とその出力の簡略化

記号実行 [14,15] は，ソースコード中の各実行パスにおける実行可能な入力条件を記号操作的に論理式として求める手法である．記号実行では実行パスの入力条件を求める際に，メソッドの引数として X や Y などといった記号を代入することで，各実行パスを通る一般的な条件を求めることができる．

また，算出された条件式が冗長になることがある．そのため，定理証明器 Simplify [19] を用いて条件式の簡略化を行う．Simplify は入力された式が恒真か否かを判定する定理証明器である．また，Simplify は入力式が恒真でないと判定したとき，その式が充足可能な場合はその反例を出力する．この反例を用いることで，式の簡略化を行えることがある．

図 1 は簡単な記号実行の例である．この例では，変数 x，y に対してそれぞれ記号値 X と Y を代入し，各実行パスを通るための条件 (図中の PC) を求めている．例えば，ソースコードの 2 行目を実行する条件が $(X > Y)$，4 行目を実行する条件が偽 $(X > Y \& X > X + 1)$ であることがわかる．

表 1　提案手法の対象とする問題の入力と出力

入力	入力: 対象 Java ソースコード
出力	出力 **1**: 対象 Java ソースコードの表明 出力 **2**: インバリアント被覆テストケース

また，3 行目の else を満たす条件 $(X > Y \& X <= X + 1)$ に対して Simplify を適用すると，与条件は恒真ではないと判定され，その反例として $(X <= Y)$ が出力され，この否定をとることで，与条件の簡略形 $(X > Y)$ が得られる．

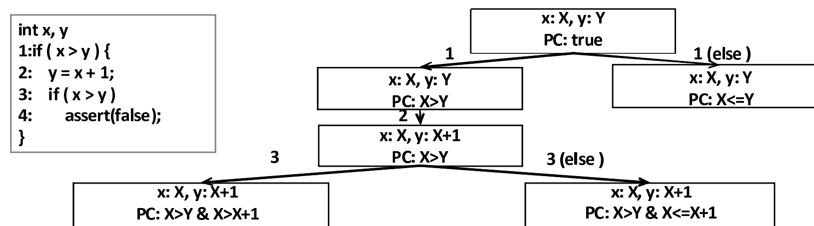

図 1　記号実行の例

記号実行はその特性からテストケース生成やデバッグツールなどのソースコードの静的解析手法の一部として用いられている [17, 18].

4　提案手法と例題への適用

本章では，提案手法について述べ，例題への適用について述べる．

4.1　提案手法の対象とする問題と手法の基本アイディア

この節では，提案手法の入力や出力，また出力の評価基準について述べ，その後，提案手法の基本アイディアを述べる．

本手法の対象とする問題の入力，出力を表 1 にまとめる．

また，出力の評価基準は以下のとおりである．

1. 出力 **2** が既存のカバレッジに基づいたテストケースよりインバリアント被覆の観点から改善されていること．
2. 出力 **1** が既存のカバレッジに基づいたテストケースを用いた Daikon の実行で生成された表明に比べ，適合率，再現率の観点から改善されていること．

この節では，提案手法の中心概念となるテストケースの質の改善，インバリアント被覆テストケースの自動生成の 2 つを先に簡単に述べる．

4.1.1　テストケースの質の改善

ソースコードを静的に解析し，DUC を求める．これにより，プログラムポイントにて参照可能な変数の値が変わる実行パスがわかるため，それに基づいて生成したテストケースのインバリアント被覆が高い値になり，その結果として，より信頼性の高い表明が生成される [13].

4.1.2　インバリアント被覆テストケースの自動生成

特定の実行を行うようなテストケースを生成するためには，その特定の実行を行う実行パスを通るためにメソッドの引数やローカル変数が満たすべき条件が必要である (以降，テストケース制約とする)．そのため，全プログラムポイントと分岐点にて参照可能な全変数の全 DUC を通るためのテストケース制約が必要である．本手法では JPF を用いて DUC を通る実行パスを取得し，その実行パスを記号実行することでテストケース制約を取得し，そのテストケース制約を簡略化したうえで,

その制約を満たすテストケースを生成する．

4.2 提案手法

この節では，提案手法の具体的な手順について述べる．提案手法では図 2 に示す手順でインバリアント被覆テストケースを自動生成する．

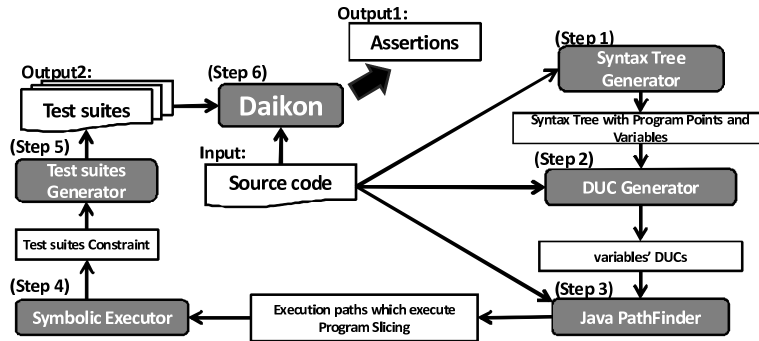

図 **2** 提案手法の概要

(Step 1) ソースコードから構文木を生成し，プログラムポイント，プログラムポイントにて参照可能な変数を取得する．(Syntax Tree Generator)；
(Step 2) 取得した全変数について DUC（定義部，使用部の位置は構文木のノード）を生成する．(DUC Generator)；
(Step 3) JPF を用いて (Step 2) で取得した全 DUC の実行パスを取得する．(Java PathFinder)；
(Step 4) 取得した全実行パスを記号実行し，テストケース制約を算出する．(Symbolic Executor)；
(Step 5) テストケース制約を満たすテストケースを生成する．(Test suites Generator)；
(Step 6) 生成したテストケースを用いて Daikon を実行する．

次節から各ステップの詳細について述べる．但し，(Step 6) はほぼ自明なため略する．

4.2.1 (Step 1) 構文木の生成

入力として与えられたソースコードから構文木を生成する．また，構文木生成の過程でプログラムポイント，各プログラムポイントにて参照可能な変数を取得する．

4.2.2 (Step 2)DUC の生成

(Step 1) で取得した全ての変数について DUC の定義部，使用部を次のように定める．使用部は出口側のプログラムポイント，定義部は対象変数への代入文の位置である．構文木からこの定義と DUC の定義に従い複数の DUC を得る．

4.2.3 (Step 3) 実行パスの検出

JPF が判定する条件として表明例外を設定することで，DUC を通る実行パスを求める．手順は次のとおりである．

1. 対象のソースコード内の各定義部 a に対して a_def という実行パス検出用のブーリアン変数を用意し，定義部があるメソッドの第 1 行に "boolean $a_def =$ true;" となる文を挿入する．この処理を各定義部に対して行う．
2. 各定義部の直後に "$a_def =$ false;" となる文を挿入する．
3. (Step 1) で取得した取得した変数から一つを選ぶ．選択した変数について DUC を一つ選ぶ．この DUC の最後の使用部にアサート文を挿入する．アサート文の

条件式は DUC が通る定義部のブーリアン変数の否定と通らない定義部のブーリアン変数の論理和である．このようにすることで必要な定義部，使用部を通るパスが実行された際にその実行パスを表明例外が発生し，JPF で実行パスを得ることができる．
4. JPF を実行し，表明例外が発生した実行パスを得る．
5. 3.4. を DUC の数だけ繰り返す．また，ソースコード内に複数の変数を組み合わせたリターン文がある場合，その変数の DUC の組み合わせについても同様の処理を行う．

JPF の実行結果は Java の命令文，対応するバイトコードの命令文の系列が得られる．

4.2.4 (Step 4) テストケース制約の算出

テストケース制約は，実行パスに対して記号実行することで算出できる．

記号実行をする際には Java の命令文を用いるが，本手法では条件分岐などの解析を行うためバイトコードの命令文も用いる．これは JPF の出力結果中の Java の命令文には，通った条件分岐に関する明示的な記述がないため，バイトコードを解析することで分岐情報を取得する必要があるためである．

4.2.5 (Step 5) テストケースの生成

テストケースは乱数を用いて，その中で制約を満たすものを選ぶことで実現する．

その手順は以下のとおりである．

1. テストケース制約を Java に変換する．
2. そのコードを用いてテストケースを生成するコードを作成する．このコードでは Java の random 関数を用いて乱数を生成し，生成された乱数のうち全ての制約を満たすものを入力として，対象のプログラムを実行する．

4.3 提案手法の例題への適用

この節では，提案手法を具体的な例題に適用する．なお，提案手法は未実装であるため，ここで述べている結果は一部手動によるものである．既存のカバレッジとしては命令網羅率，それに基づくテストケースとして命令網羅率が 100% であるテストケースを用いた．命令網羅率とは，ソースコード中の全命令文のうちテストケースが実行する命令文の割合である．

Boyer-Moore 文字列検索アルゴリズムを例題として用いる (図 3，以降 BMmatch.java とする)．

```
public static int BMmatch(String text, String pattern) {
    int[] last = buildLastFunction(pattern);
    int n = text.length();
    int m = pattern.length();
    int i = m - 1;
    if (i > n - 1)
      return -1; //【Exit 1】
    int j = m - 1;
    do {
      if (pattern.charAt(j) == text.charAt(i))
        if (j == 0)
          return i; //【Exit 2】
        else {
          // looking-glass heuristic: proceed right-to-left
          i--;
          j--;
        }
      else { // character jump heuristic
        i = i + m - Math.min(j, 1 + last[text.charAt(i)]);
        j = m - 1;
      }
    } while (i <= n - 1);
    return -1; //【Exit 3】
}
```

図 3　**BMmatch.java**

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，インバリアント被覆に基づいたテストケースの効果が

より顕著であると考え，このソースコードを例題として選んだ．

4.3.1 (Step 1) 構文木の生成

図 4 は BMmatch.java から生成した構文木であり，表 2 はプログラムポイントにて参照可能な変数である．

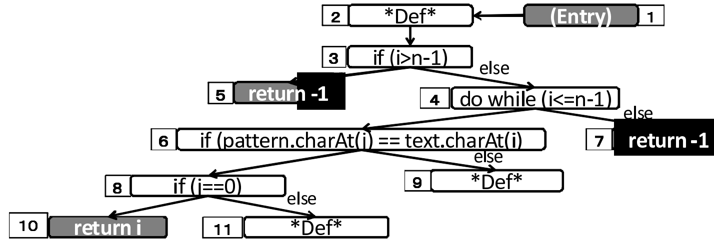

図 4 (Step 1) で得られた BMmatch.java の構文木

表 2 (Step 1) で得られた BMmatch.java の各プログラムポイントにて参照可能な変数

プログラムポイント	変数
Entry	text, pattern
Exit1	last, n, m, i, text, pattern, return
Exit2	last, n, m, i, j, text, pattern, return
Exit3	last, n, m, i, j, text, pattern, return

4.3.2 (Step 2)DUC の生成

表 2 のプログラムポイントにて参照可能な変数と図 4 にある構文木から，図 5 に示したような DUC を生成することができた．なお，ここでは DUC における位置は構文木のノード番号で表わしている．

i(2, 10)	i(11, 10) <= i(11, 11) <= i(9, 11) <= i(9, 9) <= i(2, 9)
i(9, 10) <= i(2, 9)	i(11, 10) <= i(11, 11) | |
i(9, 10) <= i(9, 9) <= i(2, 9)	text(1, 10)
i(9, 10) <= i(9, 9) | |	pattern(1, 10)
i(11, 10) <= i(2, 11)	text(1, 5)
i(11, 10) <= i(11, 11) <= i(2, 11)	pattern(1, 5)
i(11, 10) <= i(9, 11) <= i(2, 9)	text(1, 7)
i(11, 10) <= i(9, 11) <= i(9, 9) | |	pattern(1, 7)
i(11, 10) <= i(11, 11) <= i(9, 11) <= i(2, 9)	

図 5 (Step 2) で得られた変数 i, text, pattern の全 DUC

4.3.3 (Step 3) 実行パスの検出

図 5 で示した DUC を通る実行パスを JPF を用いて得た．

まず，BMmatch.java 内の定義部は "2:*Def*","9:*Def*","11:*Def*" の 3 つである．

定義部に対応したブーリアン定数を以下のように作成する．

- boolean a_def = true; // 2:*Def*に対応
- boolean b_def = true; // 9:*Def*に対応
- boolean c_def = true; // 11:*Def*に対応

それぞれの DUC に対応したアサート文を DUC の使用部に挿入する．その例としては以下のようなものがある．

- assert a_def || !b_def || !c_def; // DUC i(11,10)⇐i(9,11)⇐i(2,9) に対応
- assert a_def || !b_def || c_def ; // DUC i(9,10)⇐i(2,9) に対応

- assert a_def || b_def || c_def; // DUC i(2,10) に対応

このようなアサート文を挿入した対象ソースコードに対して JPF を実行した．図 6 は，その実行結果はの一部とである．

```
java.lang.AssertionError:
at BM.BMmatch(BM.java:35)
at test03.main(test03.java:34)
------------------------ path to error (length: 8)

BM.java:6                  boolean flag1 = true;
          BM.BMmatch(Ljava/lang/String;Ljava/lang/String;).0:iconst_1
          BM.BMmatch(Ljava/lang/String;Ljava/lang/String;).1:istore_2
BM.java:7                  boolean flag2 = true;
          BM.BMmatch(Ljava/lang/String;Ljava/lang/String;).2:iconst_1
          BM.BMmatch(Ljava/lang/String;Ljava/lang/String;).3:istore_3
BM.java:8                  boolean flag3 = true;
          BM.BMmatch(Ljava/lang/String;Ljava/lang/String;).4:iconst_1
          BM.BMmatch(Ljava/lang/String;Ljava/lang/String;).5:istore 4
BM.java:10                 int[] last = buildLastFunction(pattern);
          BM.BMmatch(Ljava/lang/String;Ljava/lang/String;).7:aload_1
          BM.BMmatch(Ljava/lang/String;Ljava/lang/String;).8:invokestatic BM/buildLastFunction(Ljava/lang/String;)[I
```

図 **6**　**(Step 3)** で得られた **JPF** の出力結果 (一部)

4.3.4　(Step 4) テストケース制約の算出

図 7 は，DUC「$i(11,10) \Leftarrow i(11,11) \Leftarrow i(9,11) \Leftarrow i(2,9)$」を実行するためのテストケース制約である．

```
pattern.length()-1<=text.length()-1
pattern.charAt(pattern.length()-1) == text.charAr(pattern.length()-1)
pattern.length() -1 != 0
pattern.length() -2 <= text.length()-1
pattern.charAt(pattern.length()-2) == text.charAr(pattern.length()-2)
pattern.length()-2 == 0
```

図 **7**　**(Step 4)** で得られた記号実行の結果

4.3.5　(Step 6)Daikon の実行

(Step 5) で手動により生成したテストケースの Daikon に対する結果を図 8 の右に示す.

5　考察

本章では，4.3 で示した例題への適用結果から，提案手法の有用性について考察する．比較を行う際の判定基準は次のとおりである．

- (Step 5) で生成されたテストケースが既存のカバレッジに基づいたテストケースよりインバリアント被覆の観点から優れているか？
- 既存のカバレッジに基づいたテストケースを用いて生成した表明より (Step 6) で生成した表明が適合率，再現率の値の観点から優れているか？

なお，適合率は生成された表明中の有用な表明の割合を示す値で，再現率は有用である表明中の生成された有用な表明の割合を示し，それらの値が高いとき信頼性の高い表明を生成できるツールだと考えられる．

ここで，有用な表明とは，以下の 2 つの条件のどちらかを満たすものである．

- 特定のパスに依存しない．
- 特定のパスに依存する条件 a はそのパスを通る条件 b によって含意 $(b \to a)$ されている．

また，適合率，再現率の定義は次のとおりである．

定義 3 適合率と再現率

生成された表明の集合を A_g，対象のソースコードから人が必要だと判断した表明の集合を 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．

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

5.1 テストケースの比較

既存のカバレッジに基づいたテストケースのインバリアント被覆の値は 3/17 であり，(Step 5) で生成されたテストケースのインバリアント被覆の値は 17/17 である．よって，提案手法を用いて生成されたテストケースはインバリアント被覆の観点から優れている．

5.2 生成された表明の比較

```
===============================================
BM.BM():::EXIT
===============================================
BM.BMmatch(java.lang.String, java.lang.String):::ENTER
text.toString one of { "abcdefg", "abdced", "c" }
pattern.toString one of { "aaa", "abc", "ef" }
===============================================
BM.BMmatch(java.lang.String, java.lang.String):::EXIT13
text.toString == "abcdefg"
pattern.toString == "ef"
return == 4
orig(text) has only one value
orig(pattern) has only one value
===============================================
BM.BMmatch(java.lang.String, java.lang.String):::EXIT
text.toString one of { "abcdefg", "abdced", "c" }
pattern.toString one of { "aaa", "abc", "ef" }
return one of { -1, 4 }
text.toString == orig(text.toString)
pattern.toString == orig(pattern.toString)
===============================================
```

```
===============================================
BM.BM():::EXIT
===============================================
BM.BMmatch(java.lang.String, java.lang.String):::ENTER
text.toString >= pattern.toString
===============================================
BM.BMmatch(java.lang.String, java.lang.String):::EXIT13
===============================================
BM.BMmatch(java.lang.String, java.lang.String):::EXIT
return == 0
text.toString >= pattern.toString
text.toString == orig(text.toString)
pattern.toString == orig(pattern.toString)
===============================================
```

図 8 **Daikon** の表明出力の比較（左：既存テストケース，右：提案手法が生成したテストケース）

図 8 は，命令網羅率が 100% であるテストケースを用いた Daikon の実行結果と，(Step 6) で得た表明出力の比較を示している．この 2 つの表明出力を比較すると，一般でない表明の除去と有用な表明の追加の 2 つの差があることがわかる．

5.2.1 一般的でない表明の除去

一般的ではない表明とは，「pattern.toString one of "abcdefg", "abdced", "c" 」のようなテストケースに依存した表明を意味する．図 8 より，提案手法を適用すると一般的でない表明が除去されているので，Daikon が生成する表明の適合率を向上させたことがわかる．この理由としては，提案手法により全てのパスを網羅するテストケースを生成したために，特定のパスに依存する一般的でない表明が成り立たない実行データを得ることができたためである．

5.2.2 有用な表明の追加

有用な表明を追加することで，Daikon の生成する表明の再現率を上げることができる．図 8 より，提案手法を適用することで生成された表明の中に「text.toString >= pattern.toString」という表明が含まれていた．この表明は特定のパスに依存した表明ではないので有用な表明である．よって，より多くの有用な表明を生成することができるので，Daikon が生成する表明の再現率を向上させたことがわかる．この理由としては，既存手法では実行しなかったパスを提案手法で得られたテストケー

スで実行できたためであると考えられる．

以上より，5.1，5.2 で示したように提案手法は文献 [13] の実験結果と同様にテストケース，表明出力を改善しているので，本自動生成手法は有用であると考えている．

6 あとがき

本研究では，インバリアント被覆に基づくテストケースの自動生成手法を提案した．この手法は Daikon をベースにしながら，この欠点であるテストケース依存問題を DUC や記号実行を用いることにより改善している．BMmatch.java を例題とし，本手法と既存手法を評価した結果，テストケースと表明がともに改善されることがわかった．今後は手法の実装を行い，実用例題への適用による評価や実行効率および計算量の評価，静的生成ツールとの比較などを行いたい．

参考文献

[1] B. Meyer: “Applying Design by Contract,” Computer (IEEE), vol.25, no.10, pp.40-51, 1992.
[2] 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.
[3] P. Cousot and R. Cousot: “Modular Static Program Analysis, ” Compiler Construction, LNCS, vol.2304, pp.159-178, 2002.
[4] 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.500-517, 2001.
[5] N. Tillmann, F. Chen and W. Schulte: “Discovering likely method specifications,” Int. Conf. on Formal Engineering Methods, ICFEM 2006, LNCS 4260, 2006.
[6] P. H. Schmitt and B. Weiss: “Invariants by Symbolic Execution,” in Proc. of the 4th International Verification Workshop, VERIFY 2007, pp.195-210, 2007.
[7] D. L. Detlefs, K. Rustan M. Leino, G. Nelson and J. B. Saxe: “Extended static Checking,” SRC Research Report 159, Compaq SRC, 1998.
[8] W. Visser, K. Havelund,k G. Brat, S. Park and F. Lerda: “Model Checking Programs,” Automated Software Engineering Journal 2003, vol.10, pp.203-232, 2003.
[9] 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.
[10] 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.
[11] 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.
[12] 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.
[13] 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.
[14] S.Khurshid, C. S. Pasareanu 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.
[15] J. C. King: “Symbolic Execution and Program Testing,” Communications of the ACM 1976, vol.19, no.7, pp.385-394, 1976.
[16] 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.
[17] C. Boyapati, S. Khurshid and D. Marinov: “Korat: Automated Testing based on Java Predicates,” in Proc. of Int. Symp. on Software Testing and Analysis 2002, ISSTA 2002, pp.123-133, 2002.
[18] W. R. Bus, J. D. Pincus and D. J. Sielaff: “A Static Analyzer for Finding Dynamic Programming Errors,” Software: Practice and Experience, 30(7), pp.775-802, 2000.
[19] D. Detlefs, G. Nelson, J. B. Saxe: “Simplify: A Theorem Prover for Program Checking,” Journal of the ACM, vol.52, no.3, pp.365-473, 2005.