



Title	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
Author(s)	宮本, 敬三; 堀, 直哉; 岡野, 浩三 他
Citation	コンピュータ ソフトウェア. 2011, 28(4), p. 306-317
Version Type	VoR
URL	https://hdl.handle.net/11094/51343
rights	ここに掲載した著作物の利用に関する注意 本著作物の著作権は日本ソフトウェア科学会に帰属します. 本著作物は著作権者である日本ソフトウェア科学会の許可のもとに掲載するものです. ご利用に当たっては「著作権法」に従うことをお願いいたします.
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

<https://ir.library.osaka-u.ac.jp/>

The University of Osaka

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

宮本 敬三 堀 直哉 岡野 浩三 楠本 真二

プログラム中に表明を記述することで、プログラムの保守性、信頼性を高めることができる。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の 1 つに動的生成手法がある。動的生成手法は比較的少ないコストで表明を生成できる。しかし表明の妥当性がテストケースに依存するという問題がある。そこで著者らは、表明の妥当性向上のためにインバリアントカバレッジとモデル検査の技術を用いたテストケース自動生成手法を提案している。本論文では、提案手法および評価実験について報告する。提案手法を実装したツールは対象 Java プログラムに対しテストケース依存問題を改善したテストケースを自動生成する。評価実験の結果、生成されたテストケースによる表明の妥当性向上を確認した。

Program assertion increases correctness, usability, and reusability of the software. Recently dynamic generation of such assertions attracts attention. The dynamic approach consumes less time and memory. However, the quality of generated assertions depends on the quality of the test-cases. The authors have proposed a method to generate test-cases for the dynamic generation of assertions, which uses notion of invariant coverage and model checking techniques. The paper reports experimental results of the quality of generated assertions based on our technique. The results show improvement of the quality of the assertions.

1 はじめに

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(Design by Contract [9])に基づいてプログラムの表明を記述することで、ソースコードの可読性の向上やプログラム検証が行える。しかし、近年のソースコードサイズの増加にともない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、大別して静的手法と動的手法の 2 種類がある [11]。静的手法 [5][16] はソースコードの状態を表すモデルを生成し、実行しうる全ての状態を求めるため、妥当性の高い表明の自動生成 [5][16] や自動検査が可能である。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り、その代表的ツールに Daikon [4] がある。しかし、表明の動的生成手法の問題点として、生成表明の妥当性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 [10]。

我々の研究グループは、表明の動的生成手法に着目し、そのテストケース依存問題を改善することを目指し、インバリアントカバレッジ [6]、モデル検査器 Java PathFinder [17]、記号実行 [7] の技術を活かしテストケースを生成する手法を提案してきてい

Evaluation Experiments on Automated Generation of Test Suites for Improvement of Assertions Generated by Daikon.

Keizo Miyamoto, 大阪大学, Osaka University. 現在, 日本ユニシス (株), Nihon Unisys.

Kozo Okano and Shinji Kusumoto, 大阪大学大学院 情報科学研究科, Graduate School of Information Science and Technology, Osaka University.

Naoya Hori, 大阪大学, Osaka University. 現在, (株) 野村総合研究所, Nomura Research Institute.

コンピュータソフトウェア, Vol.28, No.4 (2011), pp.306–317. [研究論文] 2010 年 4 月 9 日受付.

る [18]. インバリエントカバレッジ [6] 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カバレッジである. 文献 [6] で提案されているインバリエントカバレッジに基づくテストケースを自動生成する手法というのが、我々が提案した手法である [18].

本論文では文献 [18] で提案した手法を実装したツールを作成し、評価実験を行った結果を報告する.

本手法の特色として次が挙げられる. インバリエントカバレッジを用いたツールを実際に作成したこと, パスの実行可能性判定のためにモデル検査器 Java PathFinder を用いたこと, 実行条件を求めるために記号実行等の技術を活用したこと, 等である.

評価実験では例題としてサイクロマティック複雑度の異なる複数の Java プログラムを用い, プログラムの複雑さと生成表明の妥当性の関係性を評価した. また, ステートメントカバレッジ, ブランチカバレッジ, マルチコンディションカバレッジに基づくテストケースによる生成表明との妥当性を比較した.

その結果, 本手法により生成されたテストケースを用いた場合に表明の妥当性が向上したことを確認した. また, 対象プログラムのサイクロマティック複雑度が高いほど本手法による表明の妥当性向上が大きかった.

以降, 2 章では本研究に関連するツールや概念について述べ, 3 章で提案手法について述べ, 4 章で例題への適用実験について述べ, 5 章で考察を行い, 6 章でまとめる.

2 準備

この章では準備として諸概念について簡単に述べる.

2.1 Daikon

Daikon [4] は対象プログラムをテストケースを用いて実行し, 実行中の変数値を観測する. そして, 観測結果と Daikon が持つ表明パターンとを照合して各メソッドの入口と出口 (以降, プログラムポイント) における表明を自動生成する.

2.1.1 表明の生成方法

本節では Daikon で用いられている表明の生成方法について簡単に述べる. Daikon は以下の手順で表明

を生成する [12].

1. 予め用意された表明テンプレートが全て成立すると仮定する.
2. 対象プログラムをテストケースを用いて実行させた際に得られた実行中の対象プログラムの変数値を使用し, 各表明が成立するか検証する. 成立しない表明があれば破棄する.
3. 対象プログラムの全ての変数値に対して成立した表明を出力する.

2.1.2 Daikon が生成する表明の精度

本節では Daikon が生成する表明の精度に影響を与える要因について述べる. 動的生成手法では, 生成表明の妥当性が入力するテストケースの品質に依存する. これをテストケース依存問題 [10] と呼ぶ. テストケースが実行する対象メソッドの実行パスに偏りがある場合, 生成表明の妥当性が低下する.

あるプログラムに対して Daikon が生成する表明の精度に影響を与えるのは以下の 2 点である.

1. Daikon に用意されている表明テンプレートの質
2. テストケースが実行する対象プログラム中の実行パスの偏り

Daikon は 2.1.1 節で述べたアルゴリズムにより表明を生成しているため, Daikon が生成できる表明の種類は予め用意されている表明テンプレートに依存する.

また, テストケースが実行する対象プログラム中の実行パスが不十分だと, 2.1.1 節で述べたアルゴリズムの手順 2 における表明のフィルタリングが十分に行われず生成表明中に誤った表明が多くなってしまう. 図 1, 2 にテストケースによる生成される表明集合の違いを示す.

図 1 に示すように必要な表明をすべて Daikon が生成できるわけではない. また実際に生成しうる表明の量は入力のテストケースの質によってさらに小さくなる. 本研究は Daikon のツール改善ではなく, 文献 [6] のアイディアに基づき入力のテストケースの質の向上によって有用な表明導出を目指す (図 2).

本手法では, このアイディアに基づき具体的なテストケースの生成まで自動で行っている. 文献 [6] では後述のインバリエントカバレッジの定義は与えている

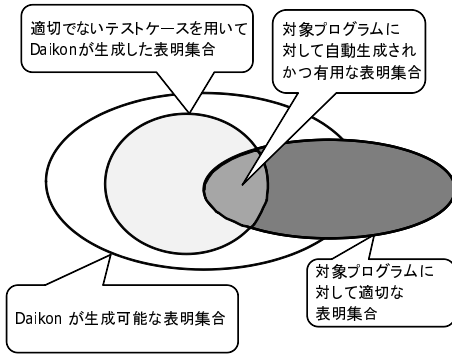


図 1 テストケースの質が悪い場合

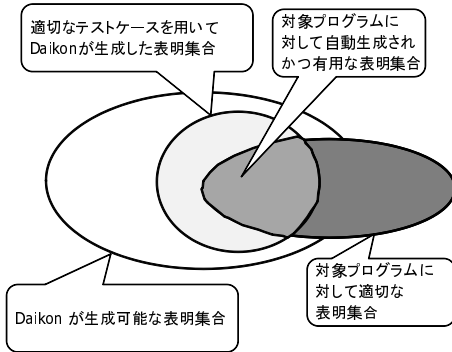


図 2 テストケースの質が良い場合

が、具体的なテストケースの導出方法については触れられていない。本手法のテストケースは文献[6]のカバレッジを厳密には満たすものではないが、実行時間で自動化する範囲で近似を行っている。

2.1.3 インバARIANTカバレッジ

本節ではインバARIANTカバレッジ[6]について簡単に述べる。インバARIANTカバレッジ (Invariant Coverage) [6] 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するカバレッジである。インバARIANT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。インバARIANTカバレッジ [6] 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 [6] では、必要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 (以下 DUC) を含む実行パ

スに近似している。

Definition-clear path の定義は一般に次のよう解釈できる。

定義 1 Definition-clear path は変数 v を左変数で代入する文から変数 v を右値で使う文へのパスである。ここでパスの要素は文であり、各文は実行順に並んでいる。変数 v への代入は最初の文を除いて存在しない。

文献 [6] では definition-clear path は未定義であるが、これを用いて以下のように DUC を定義している。

定義 2 definition-use pair は 3 項組 $v(dst, ust)$ で与えられる。ここで v は変数であり、変数 v の定義位置 dst の文から使用位置 ust の文への definition-clear path が存在する。

プログラム P のプログラムポイント S における変数 v の単純 definition-use chain を以下のように定める。

定義 3 単純 definition-use chain (DUC): プログラム P のプログラムポイント S における変数 v の単純 definition-use chain は definition-use pair の連鎖である。ただし、同一の definition-use pair が 2 回以上現れることは連鎖の最後をのぞいて、ない。

定義 4 インバARIANTカバレッジスーツ: プログラム P のプログラムポイント S におけるインバARIANTを表しているであろう式 inv に対するインバARIANTカバレッジスーツ $T(inv, S, P)$ とは、式 inv 中の全ての変数に関する最長の単純 definition-use chain の全ての組み合わせを通るパス集合を実行する入力の集合である。

なお、最長の意味はこれ以上単純 definition-use chain を追加できないパスとして定義されている。

最長の単純 definition-use chain を考えることで Daikon のアサーション生成の精度に影響を与えるパスを求めることを意図している。

- 「最長性」は冗長な部分パスの排除
- 「単純性」は繰り返しの排除

をそれぞれ意図する。

2.1.4 関連研究

Daikon 関連あるいは静的手法による表明導出、さらにはこれらを組合わせた表明導出に関する研究に

は以下のものがある。

ツールが公開されていた Houdini [5] は表明を対象プログラムに対して付加し、静的解析ツール ESC/Java2 [3] を適用して、対象プログラムに適合しない表明を除去することを繰り返し、対象プログラムに適合する表明を生成する。この方法では、動的生成に比べ表明生成にかかる時間が長いことが課題である。

静的手法は多く提案されている [16][1][15]。例えば、文献 [15] はプログラムインバリエントをプログラム上の推論規則で求める手法を提案している。

Daikon と同様に動的な表明の生成ツールに DySy [2] がある。C# 言語の対象プログラムを実行し、内部変数の更新を監視することによって、表明を出力する。

2.2 Java PathFinder

Java PathFinder(以降、JPF) [17] は Java を対象としたモデル検査器である。JPF は Bytecode に変換された対象ソースコードと検証条件を入力とし、対象ソースコードが条件を満たすかを判定し、満たさない場合に反例を出力する。JPF のコアは JVM の 1 つと見なすことができる。実行のバックトラックやスレッドの制御を行うことにより、モデル検査に必要な状態探索を実現している。当初 NASA Ames Research Center で開発され、現在はオープンソースとして開発が進められている。検証条件としてはデッドロックの可能性やキャッチされない例外等の他にユーザが指定する様式も与えることができる。また、拡張機能を容易に実装できる枠組みも提供されつつある。例えば文献 [13] では、後述の記号実行機能を JPF に組み込んだ事例が紹介されている。本研究ではパスを検証条件として与え、反例出力として得られるステートメントの系列を利用する。

2.3 記号実行

記号実行 [7] は、古典的には、ソースコード中のある実行パスを実行するためのプログラム変数の条件(以降、テストケース制約)を記号操作的に論理式として求める手法である。プログラム解析やソフトウェ

アモデル検査などにおいて用いられている。近年では種々の論理問題決定手続きの性能向上により、再び注目を集めており、通常の変数だけでなく、データ構造などの抽象化などとともに用いられることもある。例えば文献 [14] では JPF などのモデル検査器で用いるためのデータ構造抽象化の技法を与えている。本研究ではパスの実行条件を求めるために使用する。

3 提案手法

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。

提案手法では以下の手順でインバリエントカバレッジに基づいたテストケースを自動生成する。

Step 1 対象ソースコードから PDG 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 (PDG Generator)。

Step 2 PDG から対象メソッド内の全 DUC 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。

3.1.1 実行パスの検出

ここでは Step3 についてやや詳しく述べる。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。この方法はパスの実行の検出を複数のブール変数の組の値の組み合わせで行う単純な方法(トラップ変数の監視)である。

1. 対象のソースコード内の各定義部 a に対して `$$flag a` という実行パス検出用のブーリアン変数を用意し、定義部があるメソッドの第 1 行に `“boolean $$flag a = 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 `“$$flag a = false;”` となる文を挿入する。

3. Step 1 で取得した変数から 1 つを選ぶ。選択した変数について DUC を 1 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で表明例外が発生し、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 return 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。

3.1.2 記号実行

プログラム中 (今回においてはメソッド中) のある部分パスを実行するためにプログラムの開始時 (メソッドの入り口) で各変数 (メソッドの引数) の値、および変数間で成立すべき関係式を、記号実行を用いて算出する。この結果を Step 5 以降で用いる。

記号実行では、JPF で生成された実行パスとソースコード解析情報から、入力の実行パスを実行するために必要なメソッドの引数およびフィールド変数の条件を求めている。ここでは、現状態を変数コンテキスト、制約式、プログラムの 3 項組で表し、文の種類に応じた推論規則 (10 種類の文に対する 22 個の推論規則) を適用させることで命令文後の状態を求める。変数コンテキストは、現状態の変数についての情報を保持し、変数と対応する式の対応表、同値クラスの 2 項組である。これらのプログラムは自作した。記号実行の際、代入式を逐う必要があるが、この処理についてはプリミティブ型の演算のみ対応しており、参照変数については対応していない。

3.2 本手法におけるインバリアントカバレッジ

本節では本手法におけるインバリアントカバレッジの定義の詳細について述べる。

我々の実装したツールでは以下のように文献 [6] を基に幾つかの定義を追加変更している。

定義 5 インバリアントカバレッジパス: メソッド m のインバリアントカバレッジパス $T(m)$ とは、以下の条件を満たす変数 v , vf に関する最長の単純 *definition-use chain* パスの集合である。メソッド m の各 *return* 文で参照される変数 v , およびメソッドの引数変数 vm , メソッド m が属するクラスのフィールド変数 vf 。

定義 6 インバリアントカバレッジ: メソッド m のインバリアントカバレッジは、メソッド m のインバリアントカバレッジパス $T(m)$ 中の *DUC* の数を D_{all} , $T(m)$ 中の *DUC* のうちメソッド m へのテスト入力で実行された *DUC* の数を $D_{executed}$ とするとき、 $D_{executed}/D_{all}$ で表される。

なお、我々の実装では、*definition-use pair* を直後で定める PDG を用いて以下のように定義する。

定義 7 *definition-use pair*: 本論文の *definition-use pair* は 3 項組 $v(dst, ust)$ で与えられる。ここで v は変数で PDG において dst は変数 v の定義が行われるノード、 ust はその変数 v の右値が直接使われるノードであり、両者は PDG においてデータ依存辺または、制御依存辺で直接結ばれている。

定義 8 PDG: 本論文の PDG は DAG 構造をもつグラフ $G(V, E)$ で表現される。ここで $E = ED \cup EC$ (但し、 $ED \cap EC = \emptyset$) である。与えられたプログラム P のメソッド m に対し、 V の要素は m 中の各文である。 ED の要素はデータ依存辺であり、 EC の要素は制御依存辺である。

一般に、PDG におけるデータ依存辺、制御依存辺の定義は、定まったものがないが本論文では以下のように定める。

まず、変数集合 VS の要素を次のように定める。

1. メソッド m の引数変数
 2. メソッド m のローカル変数
 3. メソッド m の属するクラスのフィールド変数
- VS 中のある変数 a と、 V 中の制御文を除く各ノード s に対し、もし、変数 a の値が文 s で変更され、かつ、 s 以降に実行される直近の文であって変数 a の値を参照するノード t があれば、ノード s, t 間にデータ依存辺が存在する。

VS 中のある変数 a と、 V 中の各制御文を表すノー

<pre> 01: public class Example { 02: public int example(int x, int y) { 03: int ret = 0; 04: if (x > 0 && y > 0) { 05: ret = x + y; 06: } 07: return ret; 08: } 09: } </pre>	<pre> 01: public class Example { 02: public int example(int x, int y) { 03: int ret = 0; 04: if (x > 5) { 05: System.out.println("x is larger than"); 06: } 07: if (x > 0 && y > 0) { 08: ret = x + y; 09: } 10: return ret; 11: } 12: } </pre>
(a) source code 1	(b) source code 2

図 3 対象ソースコードの例

ド s に対し、もし、 s において a の変数が参照され s 以降に実行されうる直近の文に相当するノード t に対してノード t が変数 a の値を参照するのであれば、ノード s, t 間に制御依存辺が存在する。

なお、我々の実装したツールでは、メソッドの引数変数 vm 、メソッド m に属するクラスのフィールド変数 vf の最後の参照文に相当するノードから return 文へのデータ依存辺を仮想的に設けている。

一般に return 文で指定された変数以外にも当該メソッドにより変更されうる変数が存在するため、それらを検出できるよう仮想的に辺を追加する。なお、この処置により、依存関係が多めに設定される可能性が生じるが、最終的な目的には大きく影響しないと考ええる。このことは一般に、DUC の増加につながり、算出時間の幾分の増加に影響するが、一方で、表明導出のための精度向上には悪い方向には作用しないと考えられるからである。また、参照変数を介した変数依存には統一的な方法で対応できないので、部分的に変数依存を解析できるように次の 2 点の変更を行った。

1. オブジェクトの操作は全てそのオブジェクトの定義・参照とする。
2. 配列の各要素も VS の要素と見なす。

図 3(a), (b) に対象ソースコード、図 4, 5 に図 3(a), (b) に示したソースコードに対して生成した PDG を示す。これらを用いて必要なパスを具体的に説明する。まず図 3(a) のソースコードを用いて説明する。このメソッド `example` 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

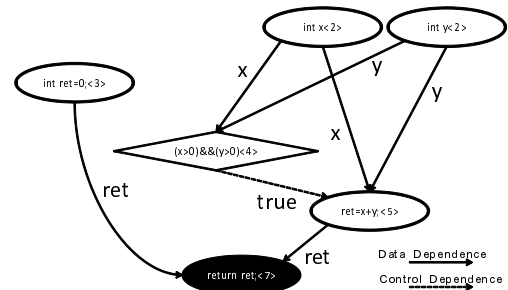


図 4 図 3(a) の PDG

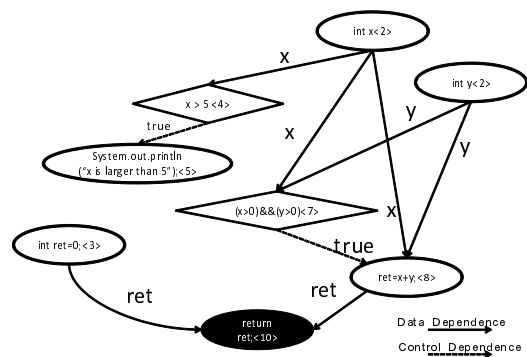


図 5 図 3(b) の PDG

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(2, 5)`
2. `ret(5, 7) ← y(2, 5)`
3. `ret(3, 7)`

また、DUC に含まれるノードの制御依存辺を探索すると 4 行目が 3 行目に制御依存していることが分かる。

次に図 3(b) のソースコードを用いて説明する。先ほどと同様に出口ノードを取得すると、10 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、このノードからデータ依存辺を探索すると 10 行目から 8 行目、8 行目から 2 行目というデータ依存と 10 行目から 3 行目というデータ依存があることがわかる。また、図 3(b) の 5 行目の命令文に対応するノードには PDG の出口ノードからデータ依存辺をたどることでは到達することができない。このため、5 行目の命令文は表明に影響を与えない命令文であることが分かる。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(8, 10) ← x(2, 8)`
2. `ret(8, 10) ← y(2, 8)`
3. `ret(3, 10)`

本手法は対象メソッド中の全パスのうち、必要なパスを網羅するテストケースを生成する。定義 4 で定めるインバリアントカバレッジスーツと定義 5 で定めるインバリアントカバレッジパスは同一ではない。これらの関係を以下で説明する。

インバリアントカバレッジパスは以下の意味でインバリアントカバレッジスーツの近似となっている。

1. 「式 *inv* 中の全ての変数」の代わりに「メソッド *m* の各 `return` 文で参照される変数 *v*, およびメソッドの引数変数 *vm*, メソッド *m* が属するクラスのフィールド変数 *vf*」を用いている。
2. 「最長の単純 definition-use chain の全ての組み合わせを通るパス集合」の代わりに「最長の definition-use chain パスの集合」を用いている。

1. の変更の意図はテストケースを作成する際に制御可能な変数に限定することにある。一方、2. の意図は自動化可能ということと、すべてのコンビネーショ

ンを生成することのコスト増大を避け、実行時間内で有効なパスを得ることにある。

なお、定義 4 で可能なインバリアントカバレッジスーツであって定義 5 のインバリアントカバレッジスーツに含まれないパスとして複数のループを持つ DUC の結合が考えられる。これは最終的に対象にならないが、この結合のうち途中のループを除いた部分は DUC の最長性により対象となる。

3.3 実行回数

本節では対象メソッドの実行回数と Daikon が生成する表明の妥当性について述べる。ループ部に対してはループ展開を行う。ループの展開回数は一般的に決定することはできない。多くの静的手法が採用しているループの `unwrap` における回数を参考にして、本手法において展開回数は 2 回とする。なお、予備実験としてループ展開を変更して実験を行ったところ、1 回よりは複数回したほうが生成される表明の質がおおむね向上する反面、実行時間が長くなり、効果が少なくなると判断した。4 章の表 5 に予備実験の結果を示す。

テストケースを実行するにあたって、制約を満たす値を具体的に定める必要がある。簡単には制約を満たすランダムな値をとる戦略が考えられ、自動実行の際には望ましいと思われる。この際、ある 1 つのテスト制約を満たす実行の実行回数が極端に少ない場合、生成される表明がテストケース制約下での振る舞いを表すものにならない可能性がある。例えば $x + y$ の値が単純に返されるメソッドにおいて x に関する実行条件が真であり x の値として 0 のみが与えられた実行系列が実行されると、事後条件として $\backslash \text{result} = y$; という誤った条件が生成される (正しくは $\backslash \text{result} = x + y$)。したがって、値が変化するという前提で実行回数もテストの実行の際は重要になるが、一般にこの回数をどれほどにすべきかは、対象メソッドやテストケース制約式・表明の種類に依存するため、一意に定まらない。本ツールでは表明の精度と実行時間の兼ね合いから経験的に 20 前後の値を採用している。

4 実験

3章で述べた提案手法を実装したツールを試作し、評価実験を行った。本実験では提案手法により生成されたテストケースと既存のカバレッジに基づいて生成されたテストケースを用いて Daikon による表明の生成を行う。本実験の目的は、各テストケースから生成された表明の妥当性の比較である。また、プログラムの複雑度と生成表明の妥当性の関連を調べる。プログラムの複雑度にはサイクロマティック複雑度 (以降, CC)[8]を用いた。

なお、今回試作したツール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. 対象メソッド中で呼び出されるメソッドに native メソッドが含まれない。
3. 対象メソッドの条件式にフィールド変数が含まれない。
4. 対象メソッド内に switch 文が含まれない。

クラス限定の制限理由は主に自作の記号実行の推論式の制限による。

また、実験環境は Windows Vista (64bit), Xeon E5405 2.00GHz × 2, 8.00GB RAM である。

4.1 評価の基準

評価方法として、以下の理由により Daikon との比較を生成される表明の質という観点から行う。

1. 比較的名かつか公開されている Houdini が現在の環境では動作しない。
2. Daikon 自体は現在でもさまざまな関連研究の成果をいれて常に改良され続けられている。
3. 静的アプローチをとっている論文もあるが多くの手法の公表でありツール実装まではされていない。

表明の質の観点は次の2つである。生成表明が論理的に正しいか、生成表明が対象メソッドの振る舞いを正しく表しているか。

そこで、比較的小さなプログラムに対して、まず人手で表明を記述し、ついで同一プログラムについて

Daikon 単体での出力結果、本手法による出力結果を得て、生成表明の妥当性、生成表明の妥当性とプログラムの複雑度の関連性、本ツールの効率性の3項目について評価を行った。

適合率は生成表明の質を示す。また、再現率は予め用意した有用な表明のうち、実際に Daikon が生成した有用な表明の割合を示す。

外的脅威を防ぐために、表明記述においては複数人でプログラムを読み、動作を理解したうえでメソッドの振る舞い (入出力の関係, 制約) をメソッドの事前条件, 事後条件として、協議しながら表明記述を行った。比較的小さく、目的のはっきりしているメソッドを選んでいるので、表明の取捨選択は容易であった。

2つの byte 型配列変数のハッシュ値を比較する CompareHashs のメソッドに対して与えた表明の例を以下に示す。

事前条件

2つの引数 a, b の型に関する条件「`\typeof(a) == \type(byte[])`」, 「`\typeof(b) == \type(byte[])`」。

2つの引数が null でないことを表す「`a!= null`」, 「`b!= null`」。

事後条件

返値の範囲に関する条件

「 $(\exists(x); a[x] \neq b[x]) \Rightarrow (-255 \leq \text{result} \ \&\& \ \text{result} \leq 255)$ 」。

返値に関する条件

「 $((a.\text{length} \geq b.\text{length}) \Rightarrow (\forall \text{int } y; 0 \leq y \ \&\& \ y < b.\text{length}; a[y] = b[y])) \Rightarrow (\text{result} == a.\text{length} - b.\text{length})$ 」など。

出力された表明の質を重視したい場合は適合率の大きさが重要であり、一方、量を重視したい場合は再現率が重視される。このように、適合率と再現率はトレードオフの関係にあるため、生成表明の論理的な正しさと有用性を総合的に評価するために F 値も用いる。

妥当性：入力された対象プログラムとテストケースより Daikon が生成した表明の適合率と再現率, F 値を求め, F 値を生成表明の妥当性とする。

Daikon が生成した表明の集合を A, 対象のソースコードに対して人手で記述した表明の集合を

表 1 対象プログラムに対する生成表明の比較

		calc			compareHashes			BMmatch			equals		
M	表明	P	R	F	P	R	F	P	R	F	P	R	F
M0	pre	0.50	1.0	0.67	1.0	1.0	1.0	1.0	0.30	0.50	1.0	1.0	1.0
	post	0.14	0.17	0.18	0.23	0.60	0.33	0.75	0.75	0.75	0.80	0.57	0.67
M1	pre	0.50	1.0	0.67	1.0	1.0	1.0	1.0	0.30	0.50	1.0	1.0	1.0
	post	0.14	0.17	0.18	0.20	0.40	0.27	0.67	0.50	0.57	0.60	0.43	0.50
M2	pre	0.50	1.0	0.67	1.0	1.0	1.0	1.0	0.30	0.50	1.0	1.0	1.0
	post	0.14	0.17	0.18	0.20	0.40	0.27	0.67	0.50	0.57	0.60	0.43	0.50
M3	pre	0.50	1.0	0.67	1.0	1.0	1.0	1.0	0.30	0.50	1.0	1.0	1.0
	post	0.14	0.17	0.18	0.22	0.60	0.32	0.67	0.50	0.57	0.75	0.43	0.55

表 2 サイクロマティック複雑度と生成表明の F 値の関連性

対象メソッド	CC	事前条件の F 値					事後条件の F 値			
		M0	M1	M2	M3		M0	M1	M2	M3
ResultQE#calc	2	0.67	0.67	0.67	0.67		0.18	0.18	0.18	0.18
Hash#compareHashes	3	0.33	0.27	0.27	0.32		0.33	0.27	0.27	0.32
BoyerMoore#BMmatch	5	0.75	0.57	0.57	0.57		0.75	0.57	0.57	0.57
Arrays#equals	7	0.67	0.50	0.50	0.55		0.67	0.50	0.50	0.55

表 3 Daikon の実行時間の比較

対象メソッド	M	実行時間	対象メソッド	M	実行時間
ResultQE#calc	M0	4.11 秒	BoyerMoore#BMmatch	M0	16.39 秒
	M1	4.50 秒		M1	3.84 秒
	M2	4.50 秒		M2	3.84 秒
	M3	4.52 秒		M3	4.10 秒
Hash#compareHashes	M0	4.50 秒	Arrays#equals	M0	5.42 秒
	M1	5.04 秒		M1	6.06 秒
	M2	5.01 秒		M2	6.17 秒
	M3	5.04 秒		M3	6.43 秒

B として、適合率 ($P = (|A \cap B|)/|A|$), 再現率 ($R = (|A \cap B|)/|B|$), F 値 ($F = 2 \times P \times R / (P + R)$) を定義する。

効率性: 本ツールと Daikon の実行時間を評価する。

4.2 対象プログラム

本実験では以下の Java プログラムを用いた。

二次方程式の解を計算する ResultQE#calc (CC:2), 2 つの byte 型の配列変数に対してハッシュ値の比較を行う Hash#compareHashes (CC:3), Boyer-Moore 文字列検索アルゴリズムを実装した Boyer-Moore#BMmatch (CC:5), 2 つの int 型の配列変数の比較を行う Arrays#equals (CC:7) である。

4.3 結果

4.2 節の各対象プログラムに対し提案手法を用いて生成された表明と、他のカバレッジを用いて生成された表明の適合率 (P), 再現率 (R), F 値 (F) の結果を示す。対象プログラムに対して生成された事前条件と事後条件における各値の比較結果を表 1 に示す。なお、表 1, 2, 3 では、テストケース生成手法を M とし、M0, M1, M2, M3 はそれぞれ提案手法、ステートメントカバレッジ、ブランチカバレッジ、マルチコンディションカバレッジに基づくテストケース生成手法を表す。また、pre, post はそれぞれ事前条件、事後条件を表す。次に、対象プログラムの CC と、各テストケースから生成された事前条件、事後条件から求めた F 値の比較結果を表 2 に示す。そして、各テス

表 4 本ツールの実行時間

対象メソッド	モジュール名				全実行時間
	DUC	JPF	記号実行	TC	
ResultQE#calc	6 秒	5 秒	1 秒	1 秒	47 秒
Hash#compareHashes	5 秒	29 秒	3 秒	1 秒	74 秒
BoyerMoore#BMmatch	83 秒	969 秒	197 秒	16 秒	1558 秒
Arrays#equals	4 秒	43 秒	2 秒	1 秒	76 秒

表 5 Boyer-Moore に対するループ展開回数変更の結果

展開回数	事前条件適合率	事前条件再現率	事後条件適合率	事後条件再現率
4	1	0.5	0.75	0.75
2	1	0.5	0.75	0.75
1	1	0.5	0.67	0.5

トケースごとの Daikon の実行時間を表 3 に示す。

最後に、各対象プログラムにおける本ツールの実行時間を表 4 に示す。モジュール名の DUC, JPF, 記号実行, TC は、それぞれ DUC 生成部, JPF 実行部, 記号実行部, テストケース生成部を表す。

表 5 に予備実験として Boyer-Moore のアルゴリズムに対してループ回数の変更を行った予備実験のデータを与える。このデータでは 2 回展開と 4 回展開では生成表明に差がないことが読み取れる。

5 考察

4.3 節に示した評価実験の結果について考察を行う。

5.1 生成表明の妥当性

本節では生成表明の妥当性の評価、考察を行う。

表 1 を見ると、事前条件は各テストケースで生成表明の適合率、再現率が同じであるが、事後条件は提案手法を用いた場合に適合率、再現率が向上していることがわかる。また、既存のカバレッジに基づくテストケースの中では、マルチコンディションカバレッジに基づくテストケースの適合率、再現率、F 値が他の 2 つのカバレッジに比べてよいことも分かる。

動的生成により妥当性の高い事後条件を生成するにはメソッド内部の実行パスを十分な回数実行する必要がある。ステートメントカバレッジは全ステートメント中一度でも実行されたステートメントの割合によりカバレッジの値が決まる。このため、カバレッジの値が高くても必要な実行パスが十分に実行され

ない場合がある。このときには生成表明の妥当性は低くなる。ブランチカバレッジは全条件分岐中一度でも実行された分岐の割合によりカバレッジが決まる。よって、ステートメントカバレッジと同様の理由により生成表明の妥当性が低くなると考えられる。マルチコンディションカバレッジは各条件分岐の組み合わせによりカバレッジの値が決まるために、このカバレッジの値が高い場合は実行されるパスの割合はステートメントカバレッジ、ブランチカバレッジに比べて高くなる。このため、生成表明の妥当性はステートメントカバレッジ、ブランチカバレッジに比べて高くなると考えられる。

事前条件の適合率、再現率、F 値は各テストケースによる違いが見られない。これは、Daikon は実引数の値の観測結果から事前条件を生成するので、メソッド内部の実行パスの実行状況が適合率などに影響を与えないためと考えられる。

5.2 妥当性とプログラムの複雑度の関連性

生成表明の妥当性と対象プログラムの複雑度の関連性について考察を行う。

表 2 を見ると、事前条件の妥当性とプログラムの複雑度には関連性がないと言える。これは、Daikon が事前条件を生成する際、対象プログラムの内部構造は考慮しないためと考えられる。

次に事後条件の結果について考察を行う。CC が 2 である ResultQE#calc では、各テストケースにより生成される表明の妥当性を表す F 値に変化が見られ

ない。次に、CC が 3 である Hash#compareHashes は、提案手法を用いた場合の方が妥当性は上がっているがその F 値の差は小さく、また全体的に F 値が低い。F 値の差が小さい理由としては、プログラムが単純なために各テストケースによって実行されるパスに差が出にくいと考えられる。また、CC が低いプログラムの場合、テストケースによるプログラム実行回数が減少する。その結果、適合率、再現率が低下し、全体的に F 値が低くなると考えられる。CC が高い BoyerMoore#BMmatch(CC:5) や Arrays#equals(CC:7) では、提案手法を用いた場合の F 値が他のテストケースに比べて F 値の向上が確認でき、また全体的に F 値が高い。提案手法の F 値が高くなるのは、他のテストケースでは実行されない DUC のパスを提案手法のテストケースでは実行するためと考えられる。全体的に F 値が高い理由は、プログラムが複雑になると実行されうるパス数が増加しテストケースによるプログラム実行回数が増加するためである。これにより多くの実行結果から表明が生成され妥当性が高くなると思われる。

以上より、対象プログラムの CC が高いほど本手法による生成表明の妥当性が上がり、提案手法の有効性が向上すると言える。

5.3 効率性

Daikon, 本ツールの実行時間について考察を行う。

まず、Daikon の実行時間について述べる。表 3 の BoyerMoore#BMmatch に対する結果を見ると、提案手法を用いた場合に Daikon の実行時間が長くなっている。Daikon の実行時間は対象プログラムの実行回数に比例する。この対象プログラムはテストケースが実行すべき実行パスが多いために、Daikon の実行時間が長くなったと考えられる。また、テストケース制約の簡略化を行っていないため、テストケース制約が複雑になり、条件を満たす引数生成に時間がかかったというのも理由として考えられる。表 3 のその他のプログラムに対する結果を見てみると、総じて提案手法を用いた場合の時間が短い。この理由として、対象プログラム全体のパスから、必要なインバリアントカバレッジを満たすパスを見つけて、他のカバレッ

ジのテストケースに比べて効率のいいテストケースを生成できたためと考えられる。

次に、本ツールの実行時間について述べる。表 4 を見ると、BoyerMoore#BMmatch に対する実行時間が他に比べて長いことがわかる。これは、プログラム中のループ内部の構造が複雑なため必要な DUC の数が多くなった、プログラムの引数が文字列であるため取りうる組合せが多い、テストケース制約が複雑ということが考えられる。また、全体的に JPF の実行時間が大部分であることが確認できる。

6 おわりに

本研究では、我々の研究グループによって提案された表明動的生成のためのテストケース自動生成ツールの実装とその評価実験を行った。評価実験を行った結果、ツールが生成したテストケースから得られた表明は、既存のカバレッジに基づくテストケースから得られた表明よりも妥当性が高いことを確認した。また対象プログラムの複雑度が高いほど、妥当性の向上が大きいことを確認した。

謝辞 本研究の一部は科学研究費補助金基盤 C(21500036) と文部科学省「次世代 IT 基盤構築のための研究開発」(研究開発領域名: ソフトウェア構築状況の可視化技術の普及) の助成による。また、本研究での実験を手伝っていただいた大阪大学情報科学研究科西本哲氏、および同、小林和貴氏に感謝する。

参考文献

- [1] Cousot, P. M. and Cousot, R.: Automatic Synthesis of Optimal Invariant Assertions: Mathematical Foundations, in *Proc. of the ACM Symp. on Artificial Intelligence and Programming Language*, 1997, pp. 1–12.
- [2] Csallner, C., Tillmann, N. and Smaragdakis, Y.: DySy: Dynamic symbolic execution for invariant inference, in *Proc. 30th ACM/IEEE Int. Conf. on Software Engineering (ICSE 2008)*, 2008, pp. 281–290.
- [3] Detlefs, D. L., Leino, K. R. M., Nelson, G. and Saxe, J. B.: Extended static Checking, *SRC Research Report 159*, Compaq SRC, 1998.
- [4] Ernst, M. D., Perkins, J. H., Guo, P. J., Mccamant, S., Pacheco, C., Tschantz, M. S. and Xiao,

- C.: The Daikon system for dynamic detection of likely invariants, *Science of Computer Programming*, Vol. 69, No. 1-3(2007), pp. 35–45.
- [5] Flanagan, C. and Leino, K. R.: 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*, 2001, pp. 500–517.
 - [6] Gupta, N. and Heidepriem, Z. V.: A New Structural Coverage Criterion for Dynamic Detection of Program Invariants, in *Proc. of Int. Conf. on Automated Software Engineering, ASE 2003*, 2003, pp. 49–58.
 - [7] King, J. C.: Symbolic Execution and Program Testing, *Communications of the ACM*, Vol. 19, No. 7(1976), pp. 385–394.
 - [8] McCabe, T. J.: A Complexity Measure, *IEEE Trans. on Software Engineering*, Vol. 2, No. 4(1976), pp. 308–320.
 - [9] Meyer, B.: Applying Design by Contract, *Computer(IEEE)*, Vol. 25, No. 10(1987), pp. 40–51.
 - [10] Nimmer, J. W. and Ernst, M. D.: Static Verification of Dynamically Detected Program Invariants: Integrating Daikon and ESC/Java, in *Proc. of First Workshop on Runtime Verification, RV 2001*, 2001, pp. 152–171.
 - [11] Nimmer, J. W. and Ernst, M. D.: Invariant Inference for Static Checking: An Empirical Evaluation, in *Proc. of SIGSOFT Symp. on Foundations of Software Engineering 2002, FSE 2002*, 2002, pp. 11–20.
 - [12] Perkins, J. H. and Ernst, M. D.: Efficient incremental algorithms for dynamic detection of likely invariants, *SIGSOFT Softw. Eng. Notes*, Vol. 29, No. 6(2004), pp. 23–32.
 - [13] Păsăreanu, C. S. and Rungta, N.: Symbolic PathFinder: symbolic execution of Java bytecode, in *Proceedings of the IEEE/ACM international conference on Automated software engineering, ASE '10*, ACM, 2010, pp. 179–180.
 - [14] Khurshid, S., Păsăreanu, C. S. and Visser, W.: 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*, 2003, pp. 553–568.
 - [15] Schmitt, P. H. and Weiss, B.: Invariants by Symbolic Execution, in *Proc. of the 4th International Verification Workshop, VERIFY 2007*, 2007, pp. 195–210.
 - [16] Tillmann, N., Chen, F. and Schulte, W.: Discovering likely method specifications, in *Int. Conf. on Formal Engineering Methods, ICFEM 2006*, LNCS, Vol. 4260, Springer-Verlag, 2006, pp. 717–736.
 - [17] Visser, W., Havelund, K., Brat, G., Park, S. and Lerda, F.: Model Checking Programs, *Automated Software Engineering Journal*, Vol. 10, No. 2(2003), pp. 202–232.
 - [18] 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, 2008, pp. 41–50.