
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

Derivation of Test Case Constraints
for Program Assertion using ESC/Java2

小林 和貴* 宮本 敬三[†] 岡野 浩三[‡] 楠本 真二[§]

Summary. Dynamic generation of assertions is important for software maintainance. Dynamic generation methods usually execute the target program and obtain its traces, which are used for inference of assertions. Therefore, the quality of test cases used in obtaining the traces is important. We have proposed a method to generate such test cases suitable for generating assertions. This paper provides an improved version of ours, which uses ESC/Java2 to construct relationship among variables and types. The improvement enlarges the applicable classes. Experiments also show good results.

1 はじめに

Design by Contract [1]に基づくアサーション記述(以下, 表明という)は, ソースコードの仕様理解の補助やプログラム検証に役立つ. 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表明生成は困難になる. 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一つである表明の静的生成手法は, プログラムを静的に解析することで表明を生成する. ツールが公開されていた Houdini [2] は静的解析器 ESC/Java2 [3] を繰り返し適用することにより, 表明を生成する. この方法では, 表明生成にかかる時間が長いことが課題である. 一方表明の動的生成手法は, 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表明の生成が可能である.

表明の動的生成ツールの一つに Daikon [4, 5] がある. Java 言語で記述された対象プログラムに対し, テストケースを与え実行しメソッドの返り値を監視することで, 表明を生成する. 同様のツールとして DySy [6] 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で, 表明を生成する. これらのツールはいずれもテストケースを必要とし, 動的に表明を生成する.

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 [5]. この問題に対し, インバリアントカバレッジ [7] 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のある命令文の組み合わせをどれだけ実行するかという割合を計算することで求まる. 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表明を生成できる. 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の提案を行ってきた [8–10].

本研究では, 既存手法 [8–10] 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 [11] 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

*Kazuki Kobayashi, 大阪大学大学院情報科学研究科

[†]Keizo Miyamoto, 大阪大学大学院情報科学研究科 (在学時に従事) 現在日本ユニシス (株)

[‡]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

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記号実行を行うことは実装上困難であり、適用可能なクラスが限定されていた。提案手法では、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。

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、有用なテストケース制約が導出できた。

以降、2 章にて背景事項について説明し、3 章で提案手法を述べる。4 章にて評価と考察を行い最後に 5 章でまとめる。

2 準備

2.1 表明

ソースコード中に表明と呼ばれる記述を行うことにより、Designed by Contract における契約を記述できる。この表明により、ESC/Java2 [3] などを用いた静的解析によりプログラムの妥当性を検証でき、開発者の意図しない不具合の混入を防ぐことができる。

2.2 表明の自動生成手法

近年のソースコードサイズの増加に伴い、手動による表明生成は困難になりつつある。そこで、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、静的手法と動的手法の 2 種類がある [12]。

静的手法 [2, 3] 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、精度の高い表明の自動生成 [2] や自動検査 [3] が可能である。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。

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。

テストケースの品質が低い場合、生成される表明の精度が低下する問題が指摘されている [5] が、一般的に比較的少ない時間とメモリで表明の生成が可能である。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 [4] がある。このツールを用いることで、手作業で表明を記述するより表明生成に必要な時間的、人的コストを軽減できる。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。

2.3 テストケース依存問題

動的生成手法により表明を自動生成する際、生成される表明の精度は入力と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 [5] と呼ぶ。動的生成手法で用いるテストケース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が、この引数の条件が十分でない場合、得られる実行データが少なくなる。このとき、限定的あるいは誤った表明が生成されてしまう場合がある。

2.4 インバリエントカバレッジ

インバリエントカバレッジ [7] 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アサーションを生成するために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 [7] では、このような実行パスをプログラム上の特定の箇所であるプログラムポイント (本論文ではメソッドの出口とする) 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。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。直前の d が次の DUP の u である。ここで、 d, u はプログラム記述中の位置を表す。プログラムポイント S にて参照可能な変数 v の DUC とは、この DUP の (有限) 系列である。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。ただし、系列の最後において d, u が同一の位置のとき、この DUP の繰り返しを許す。

2.5 プログラム依存グラフ

プログラム依存グラフ [13] とは、プログラム中の命令をノード、命令間の依存関係を有向辺で表したグラフである。主な依存関係に制御依存とデータ依存がある。提案手法では、対象メソッドの DUC を算出し、ESC/Java2 に与える JML 仕様を挿入するために使用する。

2.6 ESC/Java2

ESC/Java2 [3] は、Java プログラムに対する静的検証器である。Java プログラムを入力とし、JML で記述された表明と Java プログラムの整合性を検証し結果を出力する。ESC/Java2 は対象プログラムを述語論理に変換し、その充足不能性を判定することにより検証を行う。

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 [8–10] 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。

1. 対象ソースコードから DUC 生成用プログラム依存グラフ (DUC 生成用 PDG) を生成し、対象メソッドの出口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 を用いて (手順 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 [8–10] を参照されたい。

2.7.2 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、記号実行の実装の問題がある。テストケース制約の算出に用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記号実行を行うことを考えた場合、それぞれのクラスのメソッドに対して記号実行のルールを追加して記述する必要がある。このため、一般的に記号実行を適用することは困難であると

ということが挙げられる．

3 提案手法

既存手法の問題点である実行時間と記号実行について，本研究では静的解析器 ESC/Java2 の反例出力機能を用いて解決する手法を提案する．ESC/Java2 は，JML 付きのソースコードを入力として与えると，内部の定理証明器により与えられた性質について，プログラムが満たすかどうかを判定し，満たさない場合は反例として実行系列に関する情報を出力する．このとき，『ある実行パスを通らない』という性質を与えると，反例である『ある実行パスを通る』時の実行系列に関する情報を取得できる．この情報からテストケース制約を求めることができる．

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 1 に示す．図中にチェックマークで示した処理または外部ツールは，既存手法と比較して変更した箇所である．あわせて，提案手法の入力と出力を以下のとおり示す．

- 入力：アサーション生成対象メソッドを含む Java1.4 ソースコード
- 出力：テストケース制約

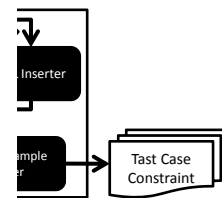


図 1 本ツールによる処理の概要

提案手法では，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 [8–10] での課題であった適用可能なクラスの拡張を行う．従来手法では Java PathFinder [11] と記号実行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 [14]．これが適用可能なクラスを小さくする原因であった．

提案手法ではこの問題を改善するために ESC/Java2 を用いる．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 内部の定理証明器 Simplify 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通り．

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 表明をもとの 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．
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．

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 型の場合のみ 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．その他の仮引数を持つメソッドの場合は，本ツールが同時に出力するテンプレートをを用い手動で作成する必要がある．

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 を生成する．本ツールでは対象ソースコード全体ではなく，対象メソッドに限定して 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．従って，提案手法はメソッド内の依存関係のみを得ることでメソッドに対する表明を生成する．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 [15] のみを対象とする．なお，本研究ではこの PDG 生成部の実装には MASU [16] を用いた．

3.3 ESC/Java2 用いたテストケース制約の導出

ESC/Java2 を用いて DUC を通るパスの実行条件を導出する方法について述べる．

3.3.1 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 [17] による表明を元の Java ソースコードに挿入する．このとき，JML の `ghost` 文，`set` 文，`assert` 文を用いて対象パスが実行されないことを意味する JML 仕様を図 2 のように挿入し，ESC/Java2 に反例を出力させる．

```
outside of if-block*/
r inside of if-block*/
```

図 2 反例取得のための JML 仕様の挿入

3.3.2 反例の解析

ESC/Java2 が出力する反例に含まれる情報には，変数の型や値および変数間の関係や，オブジェクトのメモリ上への配置さらにデッドロックの可能性がある．提案手法では，これらの反例のうち，変数の型，値に関するものや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．その他情報は不要な情報とし，提案手法では扱わない．

テストケース生成に必要な情報を抽出するためには，対象となる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 を用いて作成した．ESC/Java2 の開発者に問い合わせたところ，反例の式の厳密な文法定義は存在しないとの回答を得た．そのため，提案手法では著者らのグループにおいて必要な式の文法を独自に定義した．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おらず，必要に応じて拡張する必要がある．

3.3.3 テストケース制約導出

提案手法では，テストケース生成に必要な情報を人間が理解しやすいように出力するため，ESC/Java2 の反例を構文解析し以下の 4 種類に分類する．

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：“`==`”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などに置換する．例として，反例中に「`cand : 5.18 != @true`」という表現があるとする．ここから予約語を置換すること「`x > 0 && y > 0 != true`」という式が得られる．このように PDG の情報を活用することでより可読性の高い情報を提供できる．

4 評価

本章では評価実験の結果および考察について述べる．本研究では評価実験として，手法を実装したツールを本章にて示す Java プログラムに対して適用した．本ツールを実行時間および出力結果の妥当性の点で評価を行う．実験環境には，Intel Xeon W3520 2.67GHz × 2, Memory: 6.00GB を使用し，Windows Vista Business, JDK1.4.2_19(ESC/Java2 実行用), JDK1.6_17(本ツール, Daikon 実行用), ESC/Java2: version2.0.5, および Daikon: version4.6.3 とした．

4.1 評価実験の手順

以下の手順で評価を行った．

1. 対象ソースコードに本ツールを適用しテストケースに必要な情報及びテストケースを取得する．同時にテストケースを得るまでの実行時間を計測した．
2. テストケース制約を有用な制約，冗長な制約，誤った制約の3つに分類し要素数を計測した．
3. いくつかの対象に対しテストケースを Daikon により実行し，表明を出力させる．
4. 出力された表明をソースコードのアルゴリズムと比較し，ソースコードの表明として妥当かどうか複数人で評価し，それぞれ要素数を計測した．

4.2 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．提案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．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．指定された角度のサインを返す `static double sin(double)` や，指定された double 値の自然対数値を返す `static double log(double)` などであった．

4.3 テストケース制約の有用性

本節では，提案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提案手法を適用し，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提案手法を適用し得られたテストケース制約を調べた．その結果を，表1に示す．なお，表中の有用な制約とは，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制約のことをいい，不要な制約とは，誤った制約またはテストケー

作成時に必要でない制約のことをいう。

表 1 の通り，各対象に対して有用なテストケース制約が生成できた．テストケース制約の数は，compareHase, equals, BMmatch, regionMatches において多く生成されている．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く，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．

4.4 Daikon による生成された表明の評価

Java プログラムに対して，従来手法・提案手法をそれぞれ適用し生成したテストケースを用いて Daikon による表明の生成を行い，生成された表明の比較を行う．

対象 Java プログラムとして Boyer-Moore 文字列探索アルゴリズムを実装したメソッド BMmatch(String, String) を選んだ．このアルゴリズムは 2 つの文字列に対し，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 を返すアルゴリズムである．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存在しアルゴリズムが難解であるため，表明を生成することが難しいと考えられる．Daikon が生成した表明の評価基準として適合率，再現率，F 値を用いた．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 BMmatch に対して Daikon で表明の生成を行った．提案手法の場合は事前条件が 4 個，事後条件が 3 個生成された．従来手法の場合は事前条件，事後条件ともに 4 個生成された．これより適合率，再現率，F 値を求め，比較した結果を表 2 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，再現率が同じであった．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，再現率ともに高かった．

4.5 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．

4.5.1 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について評価する．ESC/Java2 の実行時間は例えば isCredit の実行時間が約 39 秒と長く，他のプログラムに対しては 10 秒以内に完了した．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る．

4.5.2 Daikon の実行時間

BMmatch に対して，従来手法・提案手法それぞれで生成したテストケースを利用し Daikon の実行時間を比較・考察する．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は 3 秒，従来手法によるテストケースを用いた場合は 630 秒となり，提案手法のほうが短いことが分かった．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．従来手法では対象メソッド内のループ部を 2 回まで展開するが，提案手法のツールの実装では 1 回しか展開しない．よって，従来手

表 1 生成されたテストケース制約の評価結果

プログラム名	N	U	T	プログラム名	N	U	T
BMmatch	19	14	33	getDate	4	2	6
calKai	14	5	19	getProperty	8	3	11
compareHash	21	4	25	isAlphabet	15	5	20
equals	45	11	56	isNullOrEmpty	7	1	8
formatDate	2	1	3	isCredit	7	1	8
formatDateWithTime	2	1	3	regionMatches	25	17	42

N:有用なテストケース制約，U:不要なテストケース制約，T:テストケース制約の総数

表 2 生成された表明の適合率・再現率・F 値の比較

テストケースの種類	表明の種類	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

法で生成されたテストケースはループ部を 2 回まで実行し、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とき、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り、Daikon が表明生成時の実行時情報を解析し表明を生成する時間が短縮されたと思われる。

4.6 考察

4.6.1 提案手法が適用可能なクラス

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。また、従来手法で対応していたプリミティブ型のデータを含むクラスに対しても、従来手法と同じく適用が可能である。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、従来手法と比較してより実用的なプログラムに対して使用できる。ただし、4.2 節で述べたように、適用できないクラスは存在する。

4.6.2 テストケース制約の有用性

表 1 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。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。

4.6.3 生成された表明

事前条件 生成された表明の内容を調べたところ、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表明は同じであった。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)」は生成されなかった。これは Daikon では引数の型に関する表明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)」が含まれているため、Daikon が生成した表明と合わせることで、より良い表明を作成することが可能であると思われる。

事後条件 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表明「text.toString().equals(\old(text))」、「pattern.toString().equals(\old(pattern))」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表明に「\result >= -1」がある。これは、対象メソッドの戻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。提案手法によってこの条件が出力されなかった理由としては、本ツールの実装の制限でループの展開回数を 1 回までとしたため、対象プログラムの戻り値の値域が狭くなり、表明を生成できなかったことが考えられる。なお、対象メソッドの戻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。

4.6.4 実行時間

従来手法を実装したツールの実行時間と提案手法を実装したツールの実行時間との比較を行うため、それぞれの実行時間を表 3 に示す．テストケース制約の導出は、従来手法では Java PathFinder(JPF) と記号実行で行われており、提案手法では ESC/Java2(E/J2) と反例解析で行われている．DUC 生成やテストケース生成 (TC) および全体の実行時間は表 3 左列が従来手法、右列が提案手法である．

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メソッド C および D で特に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．このことが従来手法において JPF 実行時間が長くなる原因となっている．メソッド C および D では、メソッド内でループにより繰り返し処理を行う箇所が存在する．このため、特に従来手法のような実際に対象プログラムを動作させる手法では実行時間が長くなる．

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 に比べメソッド A を除く対象プログラムで実行時間が短縮された．

また、提案手法では ESC/Java2 の反例解析部の実行時間が全体に占める割合が大きい．これは、ESC/Java2 は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、反例解析部の実行時間が相対的に長くなったためである．

4.6.5 既存手法との比較

本節では、従来手法と提案手法を比較について述べる．従来手法と提案手法の利点、欠点を表 4 に示す．提案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．また、ループ展開回数が従来手法では 2 回、提案手法では 1 回となっている．これは、提案手法の現在の実装では反例解析部が 1 回展開にのみ対応しているためである．

現在の反例解析部の文法は 3.3.2 節で述べたように著者らの研究グループで独自に定義する必要があった．そのため、ループ部を複数回展開する際に ESC/Java2 が出力する反例に対応できない部分がある．DUC 生成部および ESC/Java2 など他の部分については、2 回以上のループの展開に対応できることを確認した．

表 3 本ツールの実行時間 (秒)

Method	DUC 生成		JPF	E/J2	記号実行	反例解析	TC		全体	
A	6	2	5	8	1	10	1	0.8	47	22
B	5	3	29	6	3	11	1	1	74	20
C	83	4	969	10	197	12	16	2	1558	28
D	4	4	43	5	2	12	1	1	76	20

A:calc, B:compareHash, C:BMmatch, D>equals

表 4 既存手法との比較

	適用可能クラス		TC 制約導出	テストケース自動生成	
	P	U		P	U
既存手法	○	×	1	○	×
提案手法	○	○ ²	○	○	×

P:プリミティブ型, U:非プリミティブ型, 1:対象差が大きい, 2:Java1.4 のみ

5 あとがき

本研究では、既存手法のクラス制限を静的解析器 ESC/Java2 を用いて改善する手法を提案し、表明動的生成に有用なテストケース制約情報を出力できることを実験により確認した。また、提案手法を複数の Java プログラムに対して適用し、有用なテストケース制約を導出できることを確認した。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検討が、今後の課題である。

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。

参考文献

- [1] B. Meyer. Applying Design by Contract. in *Computer(IEEE)*, Vol. 25, No. 10, pp. 40–51, 1987.
- [2] 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.
- [3] D. L. Detlefs, K. Rustan M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. *SRC Research Report 159, Compaq SRC*, 1998.
- [4] 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.
- [5] 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.
- [6] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. *Proc. 30th ACM/IEEE Int. Conf. on Software Engineering (ICSE)*, pp. 281–290, 2008.
- [7] 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.
- [8] 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エントを含む 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.
- [9] 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エ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 XV 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.
- [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ソフトウェア工学の基礎ワークショップ FOSE2009, pp. 183–190, 2009.
- [11] 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.
- [12] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: An empirical evaluation. in *Proc. of SIGSOFT Symp. on Foundations of Software Engineering 2002, FSE 2002*, pp. 11–20, 2002.
- [13] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, pp. 319–349, 1983.
- [14] 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.
- [15] 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. 日本ソフトウェア科学会大会講演論文集, 2006.
- [16] 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.
- [17] G. T. Leavens, Albert L. Baker, and C. Ruby. JML:A Notion for Detailed Design. in *Behavioral Specifications of Businesses and Systems*, pp. 175–188, 1999.