Vol.2014-SE-185 No.27
2014/7/11

動的記号実行によるメソッドの複雑度を考慮したテストケース自動生成

高松 宏樹[†1,a)]　佐藤 晴彦[†1,b)]　小山 聡[†1,c)]　栗原 正仁[†1,d)]

概要：オブジェクト指向プログラミングにおけるソフトウェアテストでは，テストの入力データだけではなく，前処理としてのメソッド列が必要となる．前処理では，引数として要求されるオブジェクトのインスタンスの生成や状態の変更などを行う．従って，テストケース自動生成技術においては，テスト対象の状態を変化させるメソッド列を生成することは重要である．このようなテストケース自動生成ツールの一つに Seeker がある．しかしながら，Seeker はコードをカバーするために一つの変数の値を変更する場合には適切なメソッド列を生成可能だが，複数の変数の値を変更する必要がある場合には対応していない．そこで，本研究では，コードをカバーするために複数の変数の値を変更しなければならない場合にも，適切なメソッド列を生成できるよう，これを拡張した．提案手法では，条件式に関係するすべての変数を検出し，それらの値を変更するメソッド列に優先度を付けることで，テスト生成時の組み合わせ爆発を抑えることを試みた．また，複数のオープンソースプロジェクトに Seeker と提案手法を適用した比較実験を行うと共に，テスト対象メソッドの複雑度に応じてテスト生成にかける時間を調整することにより得られる結果の変化を調査した．

1. はじめに

近年，ソフトウェア開発プロジェクトは，大規模・複雑化の傾向にあり，それに伴い，ソフトウェアの品質を保つために行うテストに掛かる工数も増加しており，現代のソフトウェア開発プロジェクトに掛かる全体の工数に対してテスト工数が占める割合は，新規開発では約 3 割，改良開発では約 4 割と言われている [18]．一方で，ビジネスの目紛しい変化に対応するため，ソフトウェア開発プロジェクトの短納期・低予算化が求められている．

このような状況の中で，テスト工程の効率化のため，テストケース自動生成技術の研究が盛んに行われている．中でも，オブジェクト指向プログラムにおけるテストでは，カプセル化されたインスタンスが持つ状態を，適切なメソッド呼び出しを行うことによって，テストを行うにあたり必要な状態へと変化させる必要がある．そのため，従来のテストデータを生成することを目的としたテストケース自動生成技術では不十分であり，様々な手法が提案されている [13], [16]．そのようなオブジェクト指向プログラム向けのテストケース自動生成ツールの一つに Seeker[11] がある．Seeker はインスタンスの状態を変化させるためのメソッド呼び出しを含むテストケースの自動生成を行うが，ある条件分岐を通る際に複数の変数の値を変更する必要がある場合には適切なテストケースを生成することができない．本研究では，Seeker を拡張し，複数の変数に対する値の変更を考慮したテストケース自動生成手法を提案する．この際，テストケースとして出力するメソッド呼び出しの組み合わせ数が膨大になり，現実的な実行時間で出力を得ることが出来ないことが考えられるため，メソッド毎に評価値を与え，探索空間を削減することを試みた．また，テスト対象メソッドの cyclomatic 複雑度に基づき，テストケースの生成に費やす時間の上限を動的に変更することによる生成することができるテストケースと実行時間の変化を調査した．

本論文の構成は以下の通りである．2 節では関連研究について説明し，3 節では既存手法の問題点と，その解決案としての提案手法について述べ，4 節では既存手法と提案手法をそれぞれ適用して行った実験について説明・実験結果の考察を行い，5 節でまとめを述べる．

[†1] 現在，北海道大学 大学院情報科学研究科
Presently with Hokkaido University
[a)] takamatsu@complex.ist.hokudai.ac.jp
[b)] haru@complex.ist.hokudai.ac.jp
[c)] oyama@ist.hokudai.ac.jp
[d)] kurihara@ist.hokudai.ac.jp

2. 関連研究

2.1 動的記号実行

2.1.1 概要

単体テストのテストケース自動生成手法の一つに動的記号実行 (Dynamic Symbolic Execution，以下 DSE)[12] がある．DSE は，具体的な値を用いたプログラムの実行と，記号実行 [5] を組み合わせた手法であり，ある関数に対して，テストデータとなる引数の値の組を出力する．DSE では，ある関数について，プログラム中の条件式を一階述語論理式に変換し，それを SMT ソルバ (Satisfiability Modulo Theories Solver，背景理論付き SAT ソルバ)[2], [17] を用いて解くことにより，論理式を満たすような各変数への値の割り当てを決定する．具体的な値 (Concrete Value) を用いた実行と記号実行 (Symbolic Execution) を組み合わせることから，Concolic Testing[4], [9], [10] とも呼ばれる．

2.1.2 アルゴリズム

動的記号実行のアルゴリズムは，以下のようになっている．

(1) テスト対象プログラムに実行トレースを取得するための処理を挿入する (計測化).
(2) テスト対象となる関数の引数の値をランダムに割り当てる．
(3) 割り当てられた引数の値を用いてテスト対象の関数を実行する．
(4) 記号実行により，その実行トレースにおける条件式を含む論理式を生成する．
(5) 4. で生成した論理式のうち，反転されていない式のうち最後のものを反転する．反転されていない式がない場合は，処理を終了する．
(6) 一部を反転した論理式を SMT ソルバで解き，論理式が真となるような変数への値の割り当てを得られた場合は 3. に戻り再度関数を実行する．得られなかった場合は，5. に戻る．

図 1 について，DSE を適用した際の動作は以下のようになる．始めに，実行トレースを取得する処理をテスト対象プログラムに挿入する．次に，関数 `foo` の引数 `bar`，`baz` の値をランダムに決定する．例として，$\texttt{bar} = 1$，$\texttt{baz} = 2$ とし，この値を用いてテスト対象関数を実行する．この実行では，2 行目，3 行目の条件分岐を真として通過するため，4 行目を実行して終了する．実行が一度終了すると，記号実行により直前の実行パスにおいて通過した条件分岐の各条件式を論理積で結合した論理式を生成する．この場合は，$\texttt{bar} > 0 \wedge \texttt{baz} > 0$ が得られる．この式を実行順序が後のものから順に，DSE の試行においてまだ反転させていないものを反転させることで，直前の実行とは異なる実行ブロックを表す論理式 $\texttt{bar} > 0 \wedge \texttt{baz} \leq 0$ が得られる．

```
1  int foo(int bar, int baz) {
2      if (bar > 0) {
3          if (baz > 0) {
4              return bar * baz;
5          } else {
6              return baz;
7          }
8      } else {
9          return bar;
10     }
11 }
```

図 1 DSE のサンプル

得られた式を SMT ソルバにより解き，解として $\texttt{bar} = 1$，$\texttt{baz} = -1$ が得られたとする．得られた解を引数の値として，テスト対象関数を再度実行すると，3 行目の条件式は偽となるため，先ほどとは異なる実行ブロックを通り，6 行目を実行してプログラムが終了する．DSE ではこの操作を繰り返すことで網羅的なテストデータの生成を試みる．

2.2 事前メソッド列

従来のテストデータを生成する手法は，オブジェクト指向言語におけるテストケース自動生成に用いるには不十分である．オブジェクト指向プログラミングにおけるテストでは，初期状態から特定の状態に変化させたインスタンスを引数として要求することやテスト対象メソッドが属するクラスの状態を変化させる必要があることなど，テスト対象メソッドを呼び出す以外の処理を必要とする場合が多い．本研究では，このようにテストに必要な前処理を行うメソッド呼び出し系列を事前メソッド列と呼ぶこととする．ここで，テストケースの自動生成を行うにあたって事前メソッド列に起因する問題を示す．例として，図 2 の `AddEdge` メソッドをテスト対象とした場合を考える．辺の追加の動作をテストするには，図 3 のように，予めグラフに点を追加しておく必要がある．このように，メソッドのテストケースを設計する際には，その引数の値だけでなく，事前メソッド列を考慮しなければ網羅的なテストケース群を作成することが不可能な場合がある．

2.3 Seeker

2.3.1 概要

事前メソッド列の考慮したテストケース自動生成を行うため，Microsoft Research で開発されたツールの一つに Seeker がある．Seeker は DSE と実行トレースに基づき，事前メソッド列を含むテストケースの自動生成を行う C#言語向けのツールである．

2.3.2 アルゴリズム

始めに，Seeker の事前メソッド列生成アルゴリズムにつ

```
1  class Graph {
2    private ArrayList<Edge> edges;
3    private ArrayList<Node> nodes;
4
5    public void AddNode(Node n) {
6        if (n == null) throw new Exception();
7        nodes.Add(n);
8    }
9    public void AddEdge(Node src, Node dest) {
10       if (nodes.Contains(src) &&
11           nodes.Contains(dest)) {
12           edges.Add(src, dest);
13       } else {
14           throw new Exception();
15       }
16   }
17   ...
18 }
```

図 **2** Graph クラス

```
1 Graph graph = new Graph();
2 Node s1 = new Node();
3 Node s2 = new Node();
4 graph.AddNode(s1);
5 graph.AddNode(s2);
6 graph.AddEdge(s1, s2);
```

図 **3** AddEdge メソッドのテストケース例

いて全体的な処理の概要を述べた後，アルゴリズムの構成要素となる主要な処理について説明する．疑似コードを Algorithm1 と Algorithm2 に示し，その概略を以下に示す．

(1) テスト対象メソッドを呼び出すのみのテストケースを生成する．
(2) DSE を用いて，引数の値を変更してプログラムを繰り返し実行する．これまでの実行で到達できなかった実行ブロックに到達した場合は，その事前メソッド列と引数の組をテストケースとする．
(3) DSE で実行することができなった実行ブロックを通るための条件分岐について，条件を満たすために値を変更する必要がある変数 (目標フィールド) を特定する．
(4) 目標フィールドが属するクラスの継承・包含関係を解析する．
(5) フィールド階層・メソッド呼び出しグラフを構築し，目標フィールドの値を変更し得るメソッド群を抽出する．
(6) 抽出したメソッド群をそれぞれ既存の事前メソッド列に追加することで，新たなテストケースの候補とし，2. に戻る．

始めに，DSE を適用すること得られるテストケースの生成と，通過することができなかった条件分岐の条件式につ

Algorithm 1 DynamicAnalysis

Require: tb of TargetBranch (TB)
Require: $inputSeq$ of MethodSequence (MSC)
Ensure: $targetSeq$ of MethodSequence (MSC) or null

```
1: Method m = GetMethod(tb)
2: MSC tmpSeq = AppendMethod(inputSeq, m)
3: DSE(tmpSeq, tb, out tSeq, out covBranch, out uncovBranch)
4:
5: if tb ∈ covBranch then
6:    return targetSeq
7: end if
8:
9: if tb ∈ uncovBranch then
10:   return StaticAnalysis(tb, inputSeq)
11: end if
12:
13: if tb ∉ uncovBranch then
14:   List<TB>tbList = ComputeDominants(tb)
15:   for all TB dominantBranch ∈ tbList do
16:      inputSeq = DynamicAnalysis(dominantBranch, inputSeq)
17:      if inputSeq == null then
18:         break
19:      end if
20:   end for
21:   if inputSeq ≠ null then
22:      return DynamicAnalysis(tb, inputSeq)
23:   end if
24: end if
25: return null
```

いての情報を収集する．DSE 終了後，未実行ブロックを実行するために満たさなければならない条件について，変更しなければならない変数 (これを目標フィールド (target field) とする) を特定し，目標フィールドが属するクラスの継承・包含関係を解析する．解析したクラスの依存関係から，目標フィールドの値を変更することができるメソッド，またそのメソッド呼び出しに必要なインスタンスの生成を行う事前メソッド列を生成する．新たに生成した事前メソッド列を含むテストケースを用いて，再度 DSE を適用し，テストケースの生成を行う．この操作を繰り返し行うことにより，インスタンスを特定の状態に変化させる必要がある実行ブロックを通るようなテストケースの生成を試みる．

2.3.2.1 目標フィールドの特定

前述したように，未実行ブロックを実行するために満たさなければならない条件を真とするためにその値を変更しなければならない変数を目標フィールドとし，適切な事前メソッド列を生成するには，この目標フィールドを特定す

Algorithm 2 StaticAnalysis

Require: tb of TargetBranch (TB)
Require: $inputSeq$ of MethodSequence (MSC)
Ensure: $targetSeq$

```
1: Field targetField = DetectField(tb)
2: List<TB>tbList = SuggestTargets(targetField)
3: for all TB prevTb ∈ tbList do
4:     MSC targetSeq = DynamicAnalysis(prevTb, inputSeq)
5:     if targetSeq ≠ null then
6:         targetSeq = DynamicAnalysis(tb, targetSeq)
7:         if targetSeq ≠ null then
8:             return targetSeq
9:         end if
10:    end if
11: end for
```

る必要がある．例として，条件式が `list.size > 0` のように，public なメンバ変数やメソッドの引数といった，外部から直接値を変更することができる変数で構成されている場合はそのものが目標フィールドである．しかし，先の例の public なメンバ変数 `size` が private であり，`size` メソッドの返り値となっている場合，先の条件式は `list.size() > 0` となる．この場合はメソッド呼び出しで最終的に返される変数を辿り，特定する必要がある．

2.3.2.2 フィールド階層・メソッド呼び出しグラフの構築

特定した目標フィールドについて，フィールド階層を解析する必要がある．フィールド階層は，目標フィールドを含むクラスの継承・包含関係を表し，目標フィールドの値を変更するためにどのクラスのどのメソッドを経由する必要があるかを特定する際に用いる．

フィールド階層を用いてソースコードを解析することにより，目標フィールドの値を変更し得るメソッド群を抽出し，抽出したメソッド群とそのメソッドが属するクラスの関係を表すメソッド呼び出しグラフを構築する．このグラフの終端ノードが事前メソッド列に追加される候補となるメソッドを表す．

2.4 Cyclomatic 複雑度

cyclomatic 複雑度 (循環的複雑度)[6] は，ソフトウェアメトリクスの一つであり，ソースコードの複雑さを定量的に示したものである．cyclomatic 複雑度は，ある関数についてその制御フローをグラフとして表したとき，E をグラフのエッジの数，N をグラフのノードの数，P をコンポーネント数として cyclomatic 複雑度 M は，

$$M = E - N + 2P \tag{1}$$

として計算する．McCabe は，cyclomatic 複雑度が 10 以下であれば良い構造であるとしている．cyclomatic 複雑度は，関数内の分岐の数から算出されるため，cyclomatic 複雑度 ≃ 全ての分岐を通るために必要なテストケースの数であると言える．

3. 提案手法

3.1 概要

本章では，本研究で提案する事前メソッド列を含むテストケース自動生成手法について説明する．本研究では，従来の手法では生成することができない場合に対応する，より網羅的なテストケース生成のための事前メソッド列生成手法について提案する．

既存手法である Seeker では，未実行ブロックを実行するために満たさなければならない条件式を真にするためにその値を変更する必要がある変数である目標フィールドが一つの条件式中に複数存在する場合，目標フィールドを特定するステップにおいて一番最初に見つけたもののみを目標フィールドとして認識する．そのため，複数ある目標フィールドのうち，2 つ以上の変数の値を変更しなければ条件式を真にできない場合に適切な事前メソッド列を生成することができない．

そこで，本提案手法では満たすべき条件式に複数の変数が関係している場合にも適切な事前メソッド列を生成するため，各目標フィールドについてその値を変更し得るメソッド群を検出し，それらを組み合わせて条件式を真にする事前メソッド列の生成を行うことを試みる．候補となるメソッド群を組み合わせて事前メソッド列を生成する際，組み合わせ数が膨大になることが予測されるため，ある事前メソッド列を持つテストケースを実行した際の条件式と目標フィールドにフィットネス関数を適用し求めた値をその事前メソッド列の評価値として，これを基に組み合わせ数を削減することとした．

3.2 アルゴリズム

本提案手法は，事前メソッド列を含むテストケース自動生成手法である Seeker を基に拡張を施した．疑似コードを Algorithm3 に示し，以下で，それぞれの処理において，既存手法とは異なる点を示す．

3.2.1 目標フィールドの特定

目標フィールドを特定するステップでは，既存手法のアルゴリズムを基本とするが，対象となる条件式に複数の変数が関係していた場合には，既存手法とは異なり，それら全てを目標フィールドとして扱う．その後，目標フィールドそれぞれに対し，フィールド階層とメソッド呼び出しグラフの構築を行う．

これを基に事前メソッド列の生成を行うが，その際に組み合わせの数が膨大になることが予想されるため，以降で説明する方法により，候補の削減を行う．

3.2.2 候補メソッドの削減

本提案手法では，複数の目標フィールドについて，それ

Algorithm 3 StaticAnalysisForMultiTargetFields

Require: tb of TargetBranch (TB)
Require: $inputSeq$ of MethodSequence (MSC)
Ensure: $targetSeq$

```
 1: List<Field>targetFields = DetectAllFields(tb)
 2: List<TB>tbList = new List<TB>
 3: for all Field targetField ∈ targetFields do
 4:     tbList.Append(SuggestTargets(targetField))
 5: end for
 6: for all TB prevTb ∈ tbList do
 7:     MSC targetSeq = DynamicAnalysis(prevTb, inputSeq)
 8:     if targetSeq ≠ null then
 9:         calculatePriority(targetseq)
10:         if isCandidate(targetseq) then
11:             targetSeq = DynamicAnalysis(tb, targetSeq)
12:             if targetSeq ≠ null then
13:                 return targetSeq
14:             end if
15:         end if
16:     end if
17: end for
```

ぞれの値を変更し得るメソッド群を取得する．取得したメソッド群を組み合わせることで，複数の変数を値をそれぞれ変更するような事前メソッド列の生成を行うが，この際，その組み合わせ数が膨大になることが考えられる．そのため，提案手法では，フィットネス関数を用いて事前メソッド列に評価値を付与し，生成した事前メソッド列の数が上限数を超えた場合は，評価値の低い事前メソッド列を削除する．これにより，より高いカバレッジを達成するテストケースのための事前メソッド列を現実的な実行時間内に膨大な候補の中から選び出すことを期待する．

3.2.2.1 フィットネス関数

ある条件式に含まれる変数の値が，条件式が真となる値にどの程度近いかをフィットネス関数を適用することで評価値として表す．本研究では，フィットネス関数は 32bit 整数型の変数についてのみ定義することとする．条件式の左辺を a，右辺を b としたとき，フィットネス関数が返す値は表 1 のようになる．これは，Xie らによるフィットネス関数の定義 [14] と同様である．

4. 実験

4.1 概要

本研究では，以下の 2 つの実験を行う．

実験 1 既存手法と提案手法を複数のオープンソースプロジェクトに適用した比較実験

実験 2 提案手法に Cyclomatic 複雑度を用いたタイムアウト時間の調整を適用した実験

本実験の実行環境は，Windows 7 SP1，CPU は Intel Core2Duo 3.00GHz，RAM4GB である．また，実験を行うにあたり，テストケース生成手法の各設定値は以下のように設定した．

- タイムアウト: 500 秒 (デフォルト: 120 秒)
- SMT ソルバのタイムアウト: 10 秒 (デフォルト: 2 秒)
- 新規テストが生成されない場合の連続試行回数: 214748367 (デフォルト: 100)
- 最大試行回数: 214748367 (デフォルト: 100)

提案手法は既存手法を拡張する形で C#言語で実装した．

表 1 フィットネス関数の定義

式	True	False
$a == b$	0	$\|a-b\|$
$a > b$	0	$(b-a)+1$
$a \geq b$	0	$(b-a)$
$a < b$	0	$(a-b)+1$
$a \leq b$	0	$(a-b)$

4.2 実験 1

実験 1 では 4 つのオープンソースプロジェクト [3], [7], [8], [15] に既存手法と提案手法をそれぞれ適用した結果を比較し，提案手法の有用性を考察することを目的とする．実験結果の評価を行うにあたり，より良いテストケースが生成されていることを検証する方法として，テストの質を表す代表的な指標の一つである分岐網羅率と，テストケース生成の実行に費やした時間を比較した．

4.2.1 結果

実験 1 の結果を表 2 に示す．すべてのプロジェクトについて，提案手法が既存手法よりも高い分岐網羅率を達成していることが分かる．しかしながら，実行時間については，提案手法が既存手法の約 1.5 倍から 2 倍になっている．また，QuickGraph のように実行時間の伸び率に対して，分岐網羅率の向上幅が小さいプロジェクトがあり，実行時間の伸び率と分岐網羅率の間に一定の関係があるようには見られない．

4.2.2 考察

実験 1 の結果から考察を行う．すべてのプロジェクトについて，提案手法を適用することにより，分岐網羅率の向上が認められたが，その向上幅は小さく，最大で 5%程度である．この理由として，以下の原因が考えられる．第一に，提案手法で着目した条件式に複数の目標フィールドが関係しているケースが少ないということが考えられる．第二に，事前メソッド列の候補の削減を行う際に，事前メソッド列の評価値の計算方法が適切ではないために適切な事前メソッド列を棄却してしまっていることが考えられる．本提案手法の事前メソッド列の評価値は，条件式が整数型比較の場合のみ計算しており，それ以外の場合は，32 ビット整数型の最大値 (214748367) としている．このため，評価値の計算方法を再検討する必要があると考えられる．

表 2　各プロジェクトの実験結果

プロジェクト名	Seeker			提案手法		
	テスト数	分岐網羅率 (%)	実行時間	テスト数	分岐網羅率 (%)	実行時間
DSA	961	87.2	5 時間	1387	88.2	8.5 時間
QuickGraph	1923	68.2	8 時間	2694	69.5	17.3 時間
xUnit	1360	41.1	6.3 時間	2391	46.8	8.5 時間
NUnit (Util 名前空間)	1804	44.3	12.8 時間	5125	45.5	23.3 時間

提案手法を適用することによる分岐網羅率の向上幅は小さいものの，提案手法と同等の時間をかけて既存手法を適用した場合には同様の効果は得られないため，提案手法には有用性があると考えられる．実世界での応用を考えると，実行時間と分岐網羅率のトレードオフとなるため，テスティング戦略に応じた使い分けが必要となると考えられる．

また，QuickGraph のように，実行時間の伸びに対して，分岐網羅率の向上幅が小さい場合がある．この点について考察する．QuickGraph は，グラフのデータ構造とアルゴリズムに関するライブラリであり，あるグラフのメソッドを取り上げると，エッジやノードなど 1 つのメソッドについても様々なオブジェクトが関係しており，依存関係が複雑になっているものが多い．このような場合は，ある条件式を満たすために必要となる操作が多く，長い事前メソッド列を必要とするため，実行時間が長くなったと考えられる．

4.3 実験 2

実験 2 では，テストケース自動生成手法の実践的な利用に向けて，より短時間で効率的にテストケース生成を行うことを目指し，タイムアウト時間を変更することで，全体の実行時間を短縮することを試みる．まず，タイムアウト時間を変更した場合に分岐網羅率がどのように変化するか計測し，次に，テストケース生成対象メソッド毎にタイムアウト時間を設定した場合の分岐網羅率の変化を計測する．前者ではすべてのメソッドに対するテストケース生成のタイムアウト時間が均一であり，30，60，120，240，500 秒とした場合の分岐網羅率を測定した．後者では，cyclomatic 複雑度を基にタイムアウト時間をメソッド毎に設定する．タイムアウト時間は，cyclomatic 複雑度が 1 のメソッドについても 30 秒を確保しつつ，cyclomatic 複雑度に応じて，異なる増加傾向を持つ 2 通りの方法で計算することとした．タイムアウト時間は cyclomatic 複雑度を M として，

$$Timeout = M^2 + M \times 10 + 20$$

$$Timeout = \begin{cases} 30 & (M \leq 5) \\ M \times 100 & (M > 5) \end{cases}$$

の 2 通りの方法で計算した．cyclomatic 複雑度の測定には ccm(ver.1.1.7)[1] を用いた．本稿では，実験 1 で実験対象プロジェクトとして用いた DSA に対して実験 2 を適用することとする．

表 3　DSA におけるタイムアウト時間と分岐網羅率の変化

タイムアウト時間 (秒)	分岐網羅率 (%)	実行時間 (秒)
30	87.36	4609
60	87.50	6146
120	87.50	9313
240	87.64	15523
500	88.19	30718
$M^2 + M \times 10 + 20$	87.64	15284
$M \times 100$	87.64	13971

4.3.1 結果

実験 2 の結果を表 3 に示す．タイムアウト時間の変化に対して，分岐網羅率の変化は小さく，その幅は 1%未満に収まっている．一方で，実行時間は最大で約 5 倍となった．また，cyclomatic 複雑度に基づき計算したタイムアウト時間を用いた場合は，達成した分岐網羅率が同じ，タイムアウト時間を 240 秒に設定した場合と比べて最大で約 10%の実行時間の短縮となった．

4.3.2 考察

実験 2 の結果から考察を行う．まず，タイムアウト時間を 30 秒から 500 秒まで変化させた場合でも達成した分岐網羅率の差は 1%未満であることから，ほとんどのテストケースは短い時間で生成されることが分かる．また，タイムアウト時間を 500 秒とした場合と，cyclomatic 複雑度を用いて算出したタイムアウト時間を設定した場合に生成したテストケースを比較すると，cyclomatic 複雑度が 3 から 5 といった低い複雑度を持つメソッドについて，生成したテストケースに違いにがあり，分岐網羅率に影響していた．このことから，cyclomatic 複雑度のみではテストケース生成の困難さを推測することは適切ではないと考えられる．しかしながら，あるプロジェクトのテストケース自動生成を行う際に，極力高い分岐網羅率を達成しつつ実行時間を短く抑えることができるようなタイムアウト時間を適切に設定することは困難であり，実験結果から，cyclomatic 複雑度を用いてタイムアウト時間を算出することはこの目標を達成するための指標の一つとなり得ると考えられる．

5. おわりに

オブジェクト指向プログラミングにおけるテストでは，

ある実行ブロックを通るためにオブジェクトの状態を考慮する必要があるため，前処理として事前メソッド列を同時に生成しなければならない．そこで本研究では，ある実行ブロックを通るために複数の変数を値を変更しなければならない場合に，適切な事前メソッド列を含むテストケース自動生成を行う手法を提案した．また，提案手法を実装し，4 つのオープンソースプロジェクトを対象として，既存手法との比較実験を行い，分岐網羅率が向上することを示した．さらに，実世界での応用に向けより効率的にテストケースを生成するため，メソッドの cyclomatic 複雑度に応じてテストケース生成にかけるタイムアウト時間を変更することによる分岐網羅率の変化を調査した．

今後は，提案手法の改善のため，事前メソッド列の評価値を算出するフィットネス関数の改良や，事前メソッド列に追加する候補となるメソッドのより効果的な削減方法を提案していきたい．また，実験 2 について本稿では DSA にのみ適用するに留まっているため，他のプロジェクトについても実験を行い，プロジェクトの特徴によって結果に差異が生じないかを調査していきたい．

参考文献

[1] ccm: `http://www.blunck.info/ccm.html`.
[2] De Moura, L. and Bjørner, N.: Z3: An efficient SMT solver, *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, Berlin, Heidelberg, Springer-Verlag, pp. 337–340 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1792734.1792766⟩ (2008).
[3] DSA: `http://dsa.codeplex.com/`.
[4] Godefroid, P., Klarlund, N. and Sen, K.: DART: Directed Automated Random Testing, *SIGPLAN Not.*, Vol. 40, No. 6, pp. 213–223 (online), DOI: 10.1145/1064978.1065036 (2005).
[5] King, J. C.: Symbolic execution and program testing, *Commun. ACM*, Vol. 19, No. 7, pp. 385–394 (online), DOI: 10.1145/360248.360252 (1976).
[6] McCabe, T. J.: A Complexity Measure, *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, Los Alamitos, CA, USA, IEEE Computer Society Press, pp. 407– (online), available from ⟨http://dl.acm.org/citation.cfm?id=800253.807712⟩ (1976).
[7] NUnit: `http://www.nunit.com/`.
[8] QuickGraph: `http://quickgraph.codeplex.com/`.
[9] Sen, K.: Concolic Testing, *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, New York, NY, USA, ACM, pp. 571–572 (online), DOI: 10.1145/1321631.1321746 (2007).
[10] Sen, K. and Agha, G.: CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools, *Computer Aided Verification* (Ball, T. and Jones, R., eds.), Lecture Notes in Computer Science, Vol. 4144, Springer Berlin Heidelberg, pp. 419–423 (online), DOI: 10.1007/11817963_38 (2006).
[11] Thummalapenta, S., Xie, T., Tillmann, N., de Halleux, J. and Su, Z.: Synthesizing method sequences for high-coverage testing, *SIGPLAN Not.*, Vol. 46, No. 10, pp. 189–206 (online), DOI: 10.1145/2076021.2048083 (2011).
[12] Tillmann, N. and Halleux, J.: PexWhite Box Test Generation for .NET, *Tests and Proofs* (Beckert, B. and Hhnle, R., eds.), Lecture Notes in Computer Science, Vol. 4966, Springer Berlin Heidelberg, pp. 134–153 (online), DOI: 10.1007/978-3-540-79124-9_10 (2008).
[13] Tonella, P.: Evolutionary testing of classes, *SIGSOFT Softw. Eng. Notes*, Vol. 29, No. 4, pp. 119–128 (online), DOI: 10.1145/1013886.1007528 (2004).
[14] Xie, T., Tillmann, N., de Halleux, P. and Schulte, W.: Fitness-Guided Path Exploration in Dynamic Symbolic Execution, *Proc. the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, pp. 359–368 (online), available from ⟨http://www.csc.ncsu.edu/faculty/xie/publications/dsn09-fitnex.pdf⟩ (2009).
[15] xUnit: `http://xunit.codeplex.com/`.
[16] Zhang, S., Saff, D., Bu, Y. and Ernst, M. D.: Combined Static and Dynamic Automated Test Generation, *Proceedings of the 11th International Symposium on Software Testing and Analysis (ISSTA 2011)* (2011).
[17] 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).
[18] 独立行政法人情報処理推進機構：ソフトウェア開発データ白書 2012-2013，独立行政法人 情報処理推進機構 (2012).