修士論文

ランダムテストとシンボリック実行の組み合わせによる網羅率の向上

平成 27 年 2 月 5 日　提出

指導教員

佐藤 周行　准教授

横山 直人
東京大学大学院 工学系研究科 電気系工学専攻 融合情報学コース
学生証番号：37-136500

要旨

ソフトウェアは，現代社会のあらゆるシステムにとって欠かせないものである．そのため，ソフトウェアが正しく動作する事を何らかの手法で検証する事が求められる．ソフトウェアの正しさを検証する手法として，演繹的手法，モデル検査などの厳密な形式的手法があるが，現在ソフトウェア業界で最も広く使われているのは，非形式的手法であるソフトウェア・テストである．テストケースは主に人の手によって作成されるが，コストが莫大であるため，テストケースを自動的に生成する手法が盛んに研究されている．ランダムテストはその中でも，適用の容易さ，スケーラビリティに優れているが，依然としてプログラムの挙動の網羅率が低いという問題がある．そこで，本研究では，特に文字列型の値を含む分岐条件に着目して，ランダムテストと文字列をサポートしたシンボリック実行を組み合わせたテストケース自動生成手法 Randsym を提案した．Randsym を，文字列処理を含む分岐条件を用意した小さいベンチマークで評価した結果，既存手法のフィードバック駆動型ランダムテストでは網羅できない分岐条件が網羅できる事が分かった．今後の課題として，Randsym の安定化・高速化，そしてより規模の大きい，実際に使われているソフトウェアでの評価がある．

謝辞

本研究を進めるにあたり，ご指導頂いた指導教員である佐藤周行准教授に深く感謝致します．先生には，ミーティング等を通して大変貴重なご指摘を何度も頂きました．また，研究室のご先輩であった馬雷さん，Mario さんに深く感謝致します．特に，馬雷さんには，研究の方向性だけでなくより具体的な数々のご助言を頂きました．誠に有難うございました．

目 次

第1章　序論

1.1　ソフトウェア検証の重要性

ソフトウェアは，現代社会のあらゆるシステムにとって欠かせないものである．そのため，ソフトウェアが正しく動作する事を何らかの手法で検証する事が求められる．特に，安全性が厳しく要求されるシステムにおいては，ソフトウェアの欠陥が大規模な経済的損失や人命の損失といった致命的な被害をもたらす事がある．従って，ソフトウェア検証（software verification）を行う必要がある．

ソフトウェアの欠陥がそのような致命的な被害をもたらす有名な例として，Ariane 5 ロケットの打ち上げ失敗事故 [1, 2] が挙げられる．1996 年 6 月 4 日，欧州宇宙機関（European Space Agency, ESA）が打ち上げた Ariane 5 (Flight 501) は，打ち上げの約 40 秒後，高度約 3.7km において，飛行経路から大きく外れ，飛行中断システムによって自壊した．事故の直接の原因は，慣性基準装置内部のソフトウェアにおいて，64 bit 浮動小数点数から 16 bit 符号付き整数への変換によって生じたオペランドエラーが捕捉されなかった事である．これは，慣性基準装置のソフトウェアに対して，十分な動作検証が行われていなかった事に因る．この事故によって，3.7 億ドル以上の損失が起きた．

また，人命が失われてしまった有名な例として，放射線医療機器 Therac-25 の誤作動による大量被曝事故 [3] が挙げられる．この機器の誤作動により，1985 年 6 月から 1987 年 1 月にかけて 6 件の大量被曝事故が起き，数名が死亡あるいは重傷を負っている．Therac-25 は，加速された電子や X 線光子を照射する事によって，周りの組織を傷つけずに腫瘍を取り除く放射線医療機器である．Therac-25 のオペレーティング・システムは，一人のプログラマによって，PDP 11 アセンブリ言語で書かれており，機器の状態の監視，ユーザからの入力による治療の設定，ビーム照射のオン・オフといった機能を持っていた．開発元の資料に拠ると，システムの同期は共有変数のみによって行われ，その共有変数には，並列アクセスが許されていた．このような設計によって引き起こされる競合状態が，事故の主要な原因となった．

以上の例は，ソフトウェアの欠陥が致命的な被害をもたらす可能性がある事を示すと同時に，安全性が厳しく求められる分野においても，20 年以上前からソフ

トウェアが重要な役割を担っている事を示している．また，安全性の要求が先の例程高くないシステムにおいても，ソフトウェアの検証は重要である．

例えば，2014 年 4 月に明らかになった，OpenSSL 暗号ライブラリ [4] の Heartbleed バグ（CVE-2014-0160）[5] が挙げられる．OpenSSL は TLS [6] を実装したオープンソースライブラリであり，ユーザが個人情報を入力する必要のある多くの Web サービスにおいて，通信の安全性を高めるために利用されている．このバグは，OpenSSL が heartbeat コマンドにおいて入力の境界チェックを怠っていた事に因る．攻撃者は，それを利用してサーバのメモリをオーバーリードする事により，サーバの秘密鍵を入手する事ができる．従って，暗号を破り，本来アクセスされてはいけないユーザ情報を入手する事が可能になり，システムの安全性が大きく損なわれる．OpenSSL は大規模なサイトにおいても広く利用されているため，その影響は大きい．

以上の例で確認したように，ソフトウェアが社会的基盤の一つとなっている現在，ソフトウェア検証の重要性は高い．

1.2 ソフトウェア検証の手法

ソフトウェア検証の手法は，形式的手法（formal method）と非形式的手法（informal method）に大別できる．何れの手法も，これらの手法間のスペクトル上に位置付けられる．

形式的手法は，プログラムが与えられた仕様を満たす事（あるいは満たさない事）を証明するための手法である．通常，仕様は論理式として記述される．形式的手法の特徴は，入力空間上の全ての入力に対して，プログラムの振る舞いが仕様に従うかどうかを証明できる事である．一般に，形式的手法は非形式的手法より厳密な検証が可能だが，仕様は人間の手で書く必要があり，かつ証明の一部を人間が行わなければならない場合があるため，適用が難しい．また，より多くの計算資源を必要とする事が多い「重い」手法でもある．形式的手法には，演繹的手法（deductive method）とモデル検査（model checking）がある．

形式的手法には理論的限界が存在する．例えば，よく知られているように，プログラムの停止性を判定する事は一般的には不可能である [7]．この停止性問題の決定不可能性定理を一般化したライスの定理 [8] に拠ると，プログラムがある自明でない性質を持つかどうかは一般に決定不能である．つまり，プログラムが与えられた仕様を満たすかどうかを証明する健全かつ完全なアルゴリズムは存在しない．従って，形式的手法では，人の手を借りて対話的に証明を行ったり，健全性のみを保ち完全性を捨てる抽象化を行ったり，有限のシステムのみを抽出して考える事でこの限界を回避している．

一方，非形式的手法は，入力空間上の全ての入力ではなく，一部の入力を選んで，それに対して期待する出力が得られるかどうかを確認する手法である．これは，プログラムの振る舞いを部分的に確認するだけであるため，形式的手法と比べて厳密性は低い．しかし，プログラムの仕様を形式的に記述したり，証明を行う必要がないため，適用は容易である．また，一般に，要求される計算資源は形式的手法のそれよりも少ない．そのため，非形式的手法がソフトウェア業界で現在最も広く使われている．ソフトウェア・テスト（software testing）が非形式的手法に属す.

1.2.1 演繹的手法

演繹的手法は，プログラム C の仕様として事前条件（precondition）P，事後条件（postcondition）Q が与えられた時，P を満たす全ての状態において C を実行した後，必ず Q を満たす状態にある事を証明（あるいは反証）する．C の停止性を問わない場合を部分正当性（partial correctness），C の停止性を問う場合を完全正当性（total correctness）といい，それぞれ Hoare トリプル $\{P\}C\{Q\}$，$[P]C[Q]$ で表す．多くの場合，部分正当性の証明と停止性の証明は分けて行われ，それらを証明する事で完全正当性を証明する．P や Q は一階述語論理や高階述語論理で記述され，証明は Floyd-Hoare 論理 [9, 10] や述語変換意味論 [11] 用いて行なう．多くの場合，$\{P\}C\{Q\}$ の証明は，検証条件（verification condition）を生成し，その妥当性を自動定理証明器や対話的定理証明器を使って証明する事で行われる．一般に，仕様に用いる論理の表現力と検証プロセスの自動化の度合いはトレードオフの関係にある.

検証条件

検証条件は，その妥当性の証明がプログラム C の部分正当性 $\{P\}C\{Q\}$ の証明と等価であるような論理式であり，Floyd-Hoare 論理などの健全な演繹体系を用いて，プログラムから導出される．一般に，ループ文における繰り返し回数を静的に求める事はできないため，ループ文を含むプログラムから検証条件を導出するには，各ループの不変条件（loop invariant）をプログラマが与える必要がある．生成された検証条件は，定理証明器に渡され，その妥当性が証明（あるいは反証）される.

定理証明器

定理証明器は，与えられた検証条件の否定をとり，それが充足可能であるかを求める．それが充足可能である場合，元の検証条件は妥当ではない．逆に，それが充足不能である場合，元の検証条件は妥当である．

近年，自動定理証明器である SMT ソルバ（Satisfiability Modulo Theories solver）[12, 13, 14] の登場によって，証明できる論理式の幅が広がった．SMT ソルバは，ビットベクタ，配列，より複雑な再帰的データ構造などの理論を必要とする，量化子を含まない一階述語論理式の充足可能性を，命題論理における充足可能性問題に帰着させ，SAT ソルバ（Satisfiability solver）を用いて解く．SMT ソルバとして有名なものに，CVC3 [15] や Z3 [16] がある．

一方，一階述語論理式や高階述語論理式の妥当性は一般に決定不能であるから，人の手を借りる対話的定理証明器も重要である．対話的定理証明器は，表現力の高い高階述語論理を扱う事ができる．例えば，Isabelle [17] や Coq [18] が有名である．

1.2.2 モデル検査

モデル検査 [19, 20, 21] は，並列プログラムの検証のために考え出された手法である．従来，並列プログラムの検証は，Floyd-Hoare 論理を用いて手動で証明する事によって行われていたが，モデル検査では，状態探索手法と並列プログラムの性質を記述するのに有効な時相論理を組み合わせて，検証を自動的に行う事ができる [22].

モデル検査において，通常，システムは有限状態の Kripke 構造（状態遷移システム）M として表され，仕様は時相論理式 f として与えられる．このとき，$M \models f$，即ち，M の全ての状態 s について f が真であるかを決定する事がモデル検査の問題である．

通常，モデル検査では部分的な仕様のみを検証する．それらは安全性についての性質（safety properties）と生存性についての性質（liveness properties）の二種類に大別できる．直観的には，安全性についての性質は，バッファオーバーフローや API の呼び出し規則違反等のような「悪い事」が起こらないというものであり，生存性についての性質は，プログラムが停止する，リクエストがいつかは受理されるといった「良い事」が最終的には起こるというものである．

モデル検査の基本的なアルゴリズムでは，M において到達可能な全ての状態について f が真であるかチェックする．従って，f がある状態において偽であった場合，その状態に遷移するまでの実行トレースである反例（counterexample）が得られ，問題を修正するために役立てる事ができる．一方，状態の数は一般にシステ

ムの規模に対して指数関数的に増加するため，状態爆発が起きる．状態爆発を如何に回避するかという事が，モデル検査における本質的な問題である．そのため，状態を記号で表したり，抽象化を行う事で状態数を減らす工夫がなされている．

状態の表現方法

モデル検査におけるシステムの状態の表現方法は，明示的状態（explicit state）と記号的状態（symbolic state）の二種類に大別できる．

明示的状態は，システムの状態を，スレッドスタックやヒープなどの具体的な状態で表す．明示的状態を用いるモデル検査では，各状態はそのままインデックスされる．

一方，記号的状態は，状態をそのまま扱うのではなく，状態の集まりを一つの記号で表す．これにより，状態爆発に対処している．記号表現には，二分決定図（Binary Decision Diagram, BDD）[23] がよく使われる．二分決定図は，任意のブール関数を一意に，かつ最小のノード数で表す事ができる．

状態の抽象化

最も広く使われている状態の抽象化手法として，述語抽象（predicate abstraction）がある．これは，状態を論理述語の真偽で分割し，分割されたそれぞれの状態の集合を抽象状態としてまとめて扱う手法である．その述語を決定する反復的手法として，CEGAR（Counterexample-Guided Abstraction Refinement）[24] がある．

CEGAR は，抽象解釈（abstract interpretation）[25, 26] を用いて状態の抽象化を行う．抽象解釈は，健全（だが不完全）な抽象化を行うための理論である．CEGAR は精度の低い抽象化からモデル検査を始め，反例を見つけた場合，それが真の反例か，あるいは低精度の抽象化による誤りかを確認し，後者の場合はその反例の情報を用いて抽象化の精度を反復的に高めていく．

状態遷移の抽象化

モデル検査は，元々並列プログラムを検証するために考え出された手法であるが，並列プログラムにおいては各命令がインターリーブする可能性があるため，扱う必要のある状態がさらに増える事になる．それをできるだけ軽減するための手法として，部分的命令低減（partial order reduction）[27] がある．部分的命令低減は，結果的に同じ状態に遷移する命令列の実行をまとめて扱う事で，状態空間のサイズを小さくする．

モデル検査を行うツール

BLAST [28] は，C プログラムのためのモデル検査ツールであり，CEGAR や述語抽象等を実装している．BLAST は，Linux Driver Verification プロジェクトにも使われている．

SPIN [29] は，分散型システムのためのモデル検査ツールであり，部分的命令低減等を実装している．システムの仕様の記述には，PROMELA（PROcess MEta LAnguage）という言語を用いる．また，モデルの記述に C 言語のコードを埋め込む事もできる．

Java PathFinder（JPF） [30, 31] は，Java プログラムのためのモデル検査ツールであり，部分的命令低減等を実装している．JPF は，明示的状態（スレッドスタック，ヒープ等）を用いてモデル検査を行う．

1.2.3 ソフトウェア・テスト

ソフトウェア・テスト [32]（以下，単にテストと書く）は，テスト対象のプログラムに入力を与えて実際に実行し，その入力に対して期待する出力が得られるかどうかを確認する手法である．通常，テストを構成する入力及び出力は，テスト対象のソフトウェアが仕様を満たしているかどうかを確認するために，開発者によって選択される．各々の入力・出力のペアはテストケース（test case）と呼ばれる．テストケースの自動生成については，1.3 で扱う．

通常，ソフトウェア開発において，テストは単体レベル（unit level），統合レベル（integration level），システムレベル（system level）の三つの異なる構成単位について適用される．

単体テスト

単体テスト（unit testing）は，システムの独立な構成要素を対象に，他の構成要素と切り離して行われるテストであり，システムの各々の構成要素が期待通りに機能するかを確認したり，バグを発見・修正してコードの品質を向上させるために行われている [33]．オブジェクト指向プログラミング言語においては，通常，単体テストの対象は各々のクラスやそれに属すメソッドとなる．現在，ほとんどのプログラミング言語について，その言語のための単体テストフレームワークが存在し，単体テストはコーディングと並んで行われる共通のプラクティスとなっている．

```
1  class Point {
2    double x, y;
3
4    public Point(double x, double y) {
5      this.x = x;
6      this.y = y;
7    }
8
9    public double distance(Point other) {
10     double dx = this.x − other.x;
11     double dy = this.y − other.y;
12     return Math.sqrt(dx ∗ dx + dy ∗ dy);
13   }
14
15   @Override
16   public boolean equals(Object otherObj) {
17     if (otherObj == null)
18       return false;
19     if (otherObj == this)
20       return true;
21     if (!(otherObj instanceof Point))
22       return false;
23     Point other = (Point) otherObj;
24     return (this.x == other.x) && (this.y == other.y);
25   }
26 }
```

Fig. 1.1: An example of Java class under test. The class represents a point in 2-dimensional space.

例えば，Java の単体テストフレームワークである JUnit[34] が挙げられる．ここで，単体テストの対象となる Java クラスの例を Fig.1.1 に，そのクラスに対する JUnit テストの例を Fig.1.2 に示す．

Fig.1.1では，二次元平面上の点を表す Point クラスを定義している．クラス Point

```
1  import static org.junit.Assert.assertEquals;
2
3  import org.junit.Test;
4
5  public class PointTest {
6    @Test
7    public void distance() {
8      Point p = new Point(0.0, 0.0);
9      Point other = new Point(1.0, 1.0);
10     double d = p.distance(other);
11     assertEquals(Math.sqrt(2), d, 0.0);
12   }
13
14   @Test
15   public void isEqual() {
16     Point p = new Point(0.0, 0.0);
17     Point other = new Point(0.0, 0.0);
18     assertEquals(true, p.equals(other));
19   }
20
21   @Test
22   public void isNotEqual() {
23     Point p = new Point(0.0, 0.0);
24     Point other = new Point(1.0, 1.0);
25     assertEquals(false, p.equals(other));
26   }
27 }
```

Fig. 1.2: JUnit test case examples for Point class. Each method composes a test case that checks the behavior of a method.

は，二次元平面上の x 座標，y 座標に対応する倍精度浮動小数点数型（double 型）の変数 x，y をメンバー変数として持つ（2 行目）．コンストラクタは，単に x，y を引数としてとり，それを対応するメンバー変数に保存する（4-7 行目）．Point は二

つのメンバーメソッドを定義している．一つ目は distance メソッドであり，Point 型の引数を一つとり，その Point インスタンスと自身との間の距離を計算して返す（9-13 行目）．二つ目は equals メソッドであり，これは全てのクラスのスーパークラスである Object クラスのメソッドをオーバーライドしたものである（15-25 行目）．equals メソッドは，与えられたオブジェクトが，自身と二次元平面上で同じ点を表しているかどうかを真偽値として返す．

Fig.1.2 では，Point クラスの各メソッドに対する JUnit テストケースを定義している．それぞれのテストケースは，PointTest クラス内の各メソッド毎に一つ定義されている．PointTest クラスの distance メソッドは，それぞれ点 $(0,0)$，$(1,1)$ を表す Point インスタンスを用意して，その二点間の距離を Point クラスの distance メソッドによって計算し，それが $\sqrt{2}$ に等しいかどうかを確認している（6-12 行目）．isEqual メソッドは，点 $(0,0)$ を表す Point インスタンスを二つ用意して，Point クラスでオーバーライドした equals メソッドを呼び出し，true を返すかどうかを確認している（14-19 行目）．isNotEqual メソッドは，それぞれ点 $(0,0)$，$(1,1)$ を表す Point インスタンスを用意して，equals メソッドを呼び出し，false を返すかどうかを確認している（21-26 行目）．

Fig.1.2 に示されるように，オブジェクト指向言語の単体テストは通常，以下のような手順で行われる．

1. テスト対象のメソッドが属すクラスのインスタンスを生成する．
2. テスト対象のメソッドの引数オブジェクトを生成する．
3. テスト対象のメソッドを，生成したオブジェクトを渡して呼び出す．
4. 得られた結果が正しいかどうかを確認する．

統合テスト

統合テスト（integration testing）は，システムの各構成要素を統合したものをテストする．通常，全ての構成要素を一度に全て統合するのではなく，各構成要素をインクリメンタルに統合し，テストする．それにより，問題の箇所を特定し易くなる．

システムテスト

システムテスト（system testing）は，システム全体の振る舞いをテストする．システムテストは通常，システムの安全性，効率性，正確性，信頼性といった，非機能的要件（non-functional requirement）をテストするために行われる．

単体テストの妥当性の評価

単体テストの妥当性を評価する客観的な指標として，以下のものがよく使われる [35].

- **文網羅率（statement coverage）**
  テストによって，プログラムの全体の文に対して，どれ程の割合の文が実行されたかを表す．この割合を用いて，テストの妥当性を評価する．

- **分岐網羅率（branch coverage）**
  テストによって，プログラムの全ての分岐に対して，どれ程の割合の分岐が網羅されたかを表す．この割合を用いて，テストの妥当性を評価する．

- **経路網羅率（path coverage）**
  テストによって，プログラムの実行開始から終了までの全てのパスが，どれ程の割合で網羅されたかを表す．この割合を用いて，テストの妥当性を評価する．

- **変異妥当性（mutation adequacy）**
  ソフトウェアに意図的に欠陥箇所を作り，それがテストによって検出されたかを指標とする．意図的に作った全ての欠陥に対して，どれ程の割合の欠陥が検出されたかによって，テストの妥当性を評価する．

1.2.4 ソフトウェア検証手法の比較

ソフトウェア検証の手法として，演繹的検証，モデル検査，ソフトウェア・テストを述べた．以下に，それぞれの特徴を整理する．

- **演繹的検証**
  三つの手法の中で最も厳密だが，最も適用が難しく，最も自動化の度合いが低い．開発者はシステムの仕様を形式化する必要があり，必要に応じてループ不変条件を与えたり，対話的照明器を駆使して証明を行う必要がある．

- **モデル検査**
  モデル検査では通常，安全性・生存性についての性質についてのみ証明を行うため，システム特有の性質を示したい場合は形式化する必要があるが，自動化の度合いは高い．しかし，規模の大きいシステムに適用するには，状態爆発を回避するため，厳密性を犠牲にする事が行われる．

- **ソフトウェア・テスト**
  三つの手法の中で最も非形式的であり，厳密性はテストの設計に大きく依存する．しかし，規模の大きいシステムにも適用が容易で，学習コストも低いため，現在ソフトウェア業界で最も広く使われている手法である．1.3 で述べるように，自動的にテストケースを生成する手法も存在する．

以上より，最も適用コストの低い現実的な手法は，ソフトウェア・テストである．それにも関わらず，現在，テストはほとんどが手動で書かれているため，ソフトウェア開発に占めるコストが大きいという問題がある．その問題を解決するために，テストケースを自動的に生成する手法が盛んに研究されている．従って，本研究でも，テストケースの自動生成を扱う．

1.3 テストケースの自動生成

単体テストにおいて，通常テストケースは人の手によって書かれるが，対象のプログラムの様々な挙動を網羅するテストケースを作成するのは一般に難しく，コストが高い [36]．その問題を解決するために，テスト対象のプログラムの挙動をできるだけ網羅するテストケースを自動的に生成する手法が現在盛んに研究されている [37, 38, 39, 40, 41, 42, 43]．

自動的にテストケースを生成する手法は，シンボリック実行などのより形式的なもの [44, 45]，非形式的なランダムテスト [37, 38]，進化的アルゴリズムを用いたもの [43] に大別できる．多くの手法は，形式的なものと非形式的なものとの間のスペクトル上に位置付けられる．

1.3.1 ランダムテスト

ランダムテストは，テスト対象の関数（あるいはメソッド）について，その引数をランダムに生成し，それを使ってテスト対象の関数・メソッドを呼び出すテストケースを作る手法である．テスト対象の関数・メソッドや，プリミティブ型（整数，文字など）の引数の値は一般にランダムに選択される．多くの場合，テストが通ったかどうかの判断は,「アサーション例外を投げない」といった，あらかじめ決められた基準を元に行われる．ユーザがその基準を指定できるフレームワークも存在する．

ランダムテストの主な課題は，テスト対象のメソッドの様々な挙動を網羅する引数をどのように生成するか，という事である．オブジェクト指向言語の場合，テスト対象のメソッドが属すクラスのオブジェクトも生成する必要がある．ランダ

ムテストにおいて，テスト対象のメソッドを呼び出すために必要なオブジェクトを生成する手法は，オブジェクト操作法（object manipulation）とメソッド列生成法（method sequence generation）に大別できる．オブジェクト操作法は，必要な型のオブジェクトのコンストラクタを呼び出して新しいオブジェクトを作り，そのオブジェクトのメンバ変数の値を直接設定する事によって，プログラムの挙動を網羅するために必要な状態のオブジェクトを生成する手法である．一方，メソッド列生成法は，コンストラクタを含む適当なパブリックメソッドをいくつか呼び出す事によって，所望の状態のオブジェクトを生成する手法である．

どちらの手法においても，実用的な規模のソフトウェアに適用する場合，問題空間が非常に大きいため，有効な状態のオブジェクトを効率的に生成する工夫が必要となる．しかし，オブジェクト操作法では，オブジェクトのメンバー変数の値を直接変更するため，実際には有り得ない状態のオブジェクトを生成してしまう可能性が高い一方で，メソッド列生成法では，オブジェクトのメソッドを実際に呼び出して状態を変更するため，無効な状態は生成されにくい．そのため，ランダムテストでは，メソッド列生成法が広く使われている．

JCrasher [37] は，メソッド列生成法を用いた初期のランダムテスト手法である．JCrasher はパラメータグラフを構築し，それを用いて所望の引数を逐次生成する．JCrasher は，後述するフィードバック駆動型ランダムテストと比べて，冗長なテストケースや，意味のないテストケースを生成し易いという問題がある．

フィードバック駆動型ランダムテスト（Feedback-directed Random Testing，F-RT）は，既に生成されたメソッド列を再利用する手法である．フィードバック駆動型ランダムテストを初めて実装したのは Eclat [46] である．Eclat は与えられた既存のテストケースからプログラムのモデルを抽出し，それを元にメソッド列を normal，fault-revealing，illegal に分類する．Eclat は生成したテストケースの分類に焦点を当てている．

フィードバック駆動型ランダムテストを用いたランダムテストフレームワークとして，Randoop [38, 47] が最も広く使われている．Randoop は，既に生成されたメソッド列を再利用する事で，テスト対象のメソッドに必要なオブジェクトを生成する新たなメソッド列を生成する．新たに生成されたメソッド列がテストケースとなる．生成されたメソッド列は直ちに実行され，不正な例外やエラーが発生しない場合は再利用可能なメソッド列のプールに加えられる．これにより，無駄なテストケースを生成する可能性が低くなり，より効率の良いテストケースの生成が可能となる．フィードバック駆動型ランダムテストは，現在ランダムテストにおいて最高水準の手法である．実用的な規模のプログラムにも直ちに適用できるが，依然として網羅率が低いという問題がある．

ARTOO [39] は，適応的ランダムテスト [48] をオブジェクト指向プログラムに

拡張したものである．適応的ランダムテストは，入力ドメイン内にできるだけ均等に分布する入力を生成する手法である．ARTOO は，オブジェクト間の距離を導入する事でこれを実現している．これにより，テスト対象のプログラムの挙動がより広く網羅される事を期待している．しかし，他のランダムテストと同様に，網羅率が低いという問題がある．

Randoop を改良したものとして，Jaygarl et al.[49] の研究もある．これは，ARTOO のオブジェクト距離をより簡略化して高速化したものや，配列を生成する際に，その要素の型の値を生成するメソッド列を再利用する手法等を組み合わせたものである．

1.3.2 より形式的な手法

ランダムテストより形式的な手法においては，テスト対象のプログラムを静的・動的解析し，テストケースの網羅率を向上させる工夫がなされている．しかし，そのために人の手によって書かれたテストケースを必要としたり，実用的な規模のプログラムに適用するのが難しくなるという問題が存在する．

シンボリック実行（symbolic execution）[50] は，プログラムの通常の実行（concrete execution）と異なり，変数の値を具体的な値として持つのではなく，シンボル（記号）として持ち，分岐が行われる度に，そこに至るために必要な条件をシンボルの式として表すことで，後で制約ソルバ（constraint solver）によってその分岐を網羅するための具体的な入力の値を得る事ができる手法である．この手法の起源は古いが，近年，計算機の性能の向上と，制約ソルバの発達によって，テストケースの自動生成への応用が注目されている．

Symbolic PathFinder（SPF）[51] は，モデル検査ツールである Java PathFinder を拡張した，Java プログラムを対象とするシンボリック実行ツールである．SPF は，文字列が関わる条件にも対応している [52].

コンコリックテスト（concolic testing）[53, 54] は，外部のライブラリの結果に依存したプログラムや，制約ソルバでは解けない非線形な制約条件に対処したコンコリック実行（concolic execution）を用いた手法である．これは，ランダム値を用いた通常の実行とシンボリック実行を同時に行い，制約ソルバで解けない条件は，通常の実行で得られた値をそのまま条件に用いる手法である．

MSeqGen [40] は，テスト対象のプログラムを用いているコードベースからメソッド列を抽出し，シンボリック実行を適用して所望の分岐条件を満たすメソッド列を生成する．

OCAT [41] は，既存のシステムテストや対象となるプログラムを実際に用いるプログラムからオブジェクトの状態をキャプチャし，それを利用して Randoop [47]

の網羅率を改善している．さらに，静的解析の結果を元にオブジェクトの状態を変更し，まだ網羅されていない分岐をできるだけ網羅する．

Palus [42] は，サンプルプログラムを動的解析し，オブジェクトの状態とメソッド呼び出しの関係を表す呼び出し列グラフ（call sequence graph）を生成する．さらに，静的解析によりメソッドとフィールド間の書き込み・読み出し関係を抽出する．これらを利用して，メソッド列を構築する事でテストケースを生成する．

1.3.3 進化的アルゴリズムを用いた手法

進化的アルゴリズムを用いた手法として，EvoSuite [43] が代表的である．EvoSuite は，メソッドの呼び出し列からなるテストケースの集合の網羅率が最大になるように，変異や交差などを行いつつ，適合度による選別を繰り返す手法である．適合度はテストスート全体の網羅率から計算される．また，シンボリック実行や局所探索なども，あらかじめ決められた世代数ごとに行われる．オープンソースプロジェクトのホスティングサイトである SourceForge からランダムに選ばれた 100 個のオープンソース Java プロジェクトからなる SF100 コーパス [55] で初めて実験された手法であり，全体で平均 48% の分岐網羅率を記録している．

1.3.4 自動テストケース生成手法の比較

ランダムテスト，より形式的な手法，及び進化的アルゴリズムを用いた手法を比較すると，実用的な規模のプログラムへの適用の容易さという面では，ランダムテストが最も優れている．より形式的な手法では，シンボリック実行などを用いる事によって網羅できる分岐の種類は増えるものの，実用的な規模のプログラムに適用する場合に，実行時間が長すぎるという欠点がある．また，進化的アルゴリズムを用いた手法でも，大規模な計算資源を必要とする．一方で，ランダムテストには，網羅率が低いという欠点がある．

従って，ランダムテストとより形式的な手法とを組み合わせ，両者の欠点を補う手法が有望である．具体的には，ランダムテストの中で最高水準の手法であるフィードバック駆動型ランダムテストと，シンボリック実行を組み合わせた手法が考えられる．本研究では，ランダムテストとシンボリック実行を組み合わせた手法を提案する．

1.4 問題設定

ランダムテストの主な問題点は，網羅率が低い事である．つまり，できるだけ多くの分岐を網羅するようなテストケースを生成する必要がある．それを達成するため，ランダムテストとシンボリック実行を組み合わせた手法が考えられるが，単純なシンボリック実行では扱える条件も限られているため，jpf-symbc のような文字列をサポートした手法を利用する事が望ましい．文字列処理は広く使われており，かつ，従来のランダムテスト手法では，文字列が関わる条件を持つ分岐を網羅する事は難しい．

文字列処理を行う Java プログラムの例を Fig.1.3 に示す．Fig.1.3 のプログラムは，コマンドライン引数のロングオプションを解析する単純な Java クラスである．コンストラクタは，文字列の配列（args）を渡し，それをクラスのメンバ変数として保持する（5-7 行目）．メソッド longOptionExists は，文字列を引数（option）にとり，そのロングオプション（「–option-name」という形式で指定される）が指定されているかどうかをブール値で返す（9-21 行目）．

Fig.1.3 の 16 行目の分岐の真側，即ち args[i].equals(searchFor) という条件式が真となるようなテストケースを生成するには，先頭の 2 文字がハイフンとなるような 4 文字以上の文字列を args の要素として与え，その文字列のハイフン以降の部分文字列に等しい文字列を longOptionExists メソッドに与える必要がある．このような単純な例でも，フィードバック駆動型ランダムテストで網羅するテストケースを生成する事は難しい．当然，文字列に関するさらに複雑なメソッド等を含む場合も同様である．

本研究では，このように文字列が関わる条件を持つ分岐も網羅できるようにランダムテストを改善する事で，網羅率の向上を目指す．

1.5 概要

ソフトウェア検証は，ソフトウェアの欠陥を検出・修正するために欠かせない手法である．特にソフトウェア・テストは，現在ソフトウェア業界において広く開発サイクルに組み込まれているプラクティスである．その中でも，単体テストは，早期に欠陥を検出・修正するために有効な手法として受け入れられているが，テストケースは人の手によって構築される事が多く，莫大なコストがかかっている．

その問題を解決するため，テストケースの自動生成手法が盛んに研究されている．その中で，スケーラビリティ，適用の容易性で最も優れている手法はランダムテストである．フィードバック駆動型ランダムテストは，現在最高水準のランダムテスト手法であるが，網羅率が低いという問題がある．実際，Fig.1.3 で示さ

```
1  public class ArgsParser {
2    public static final String LONG_OPTION_INDICATOR = "--";
3    private String[] args;
4
5    public ArgsParser(String[] args) {
6      this.args = args;
7    }
8
9    public boolean longOptionExists(String option) {
10     boolean isLongOption = (option.length() > 1);
11
12     if (isLongOption) {
13       String searchFor = LONG_OPTION_INDICATOR + option;
14       for (int i = 0; i < args.length; i++) {
15         if (args[i] != null) {
16           if (args[i].equals(searchFor)) {
17             return true;
18           }
19         }
20       }
21     }
22
23     return false;
24   }
25 }
```

Fig. 1.3: An example of Java class using String methods in branch condition

れるような，文字列を含む単純な分岐条件を網羅する事が難しい．そこで，文字列をサポートしたシンボリック実行手法をランダムテストと組み合わせる手法が考えられる．

本研究では，フィードバック駆動型ランダムテストと，文字列をサポートしたシンボリック実行を組み合わせる事により，生成されるテストケースの網羅率の向上を目指す．

本論文の以降の構成は以下の通りである．まず，第 2 章で，提案手法が用いるフィードバック駆動型ランダムテスト，文字列をサポートしたシンボリック実行，及びそれらを組み合わせた提案手法を詳しく述べる．その後，第 3 章で，提案手法を評価する．最後に，第 4 章で，結論及び今後の課題を述べる．

第2章　提案手法

フィードバック駆動型ランダムテストは，生成したメソッド列を成長させていく手法であるため，徐々にメソッド列の実行時間が支配的になり，テストケースの生成速度が落ち，それによって新たに網羅される分岐は少なくなる．つまり，時間と共に生成されるテストケース全体の網羅率は時間的に飽和する．本研究の提案手法の基本的な発想は，その時間をシンボリック実行に費やす事で有効活用するというものである．言い換えれば，簡単に網羅できる分岐はランダムテストにより網羅するテストケースを素早く生成し，網羅するのが難しい分岐に対しては，ランダムテストが有効に機能しない時間を利用してシンボリック実行に任せるという事である．また，フィードバック駆動型ランダムテストによって生成されるメソッド列を利用すれば，シンボリック実行において問題となる複雑なオブジェクトの生成の問題をある程度回避できる．

本章の構成は以下の通りである．まず，2.1 と 2.2 で，提案手法で用いるフィードバック駆動型ランダムテストと文字列に対応したシンボリック実行についてそれぞれ詳しく述べる．そして，2.3 で，それらを組み合わせた提案手法 Randsym とその実装について述べる，

2.1　フィードバック駆動型ランダムテスト

フィードバック駆動型ランダムテスト [38] は，フィードバック駆動型メソッド列生成法を用いたオブジェクト指向プログラムのランダムテスト手法である．フィードバック駆動型メソッド列生成法 [38] は，生成されたメソッド列を再利用する事で，インクリメンタルにメソッド列を生成する手法である．

2.1.1　メソッド列

メソッド列とは，メソッド呼び出しの列である．メソッド呼び出しは，呼び出すメソッド名とそのメソッドに与える引数からなる．引数は，プリミティブな値か，前に呼び出されたメソッドが返したオブジェクトリファレンスである．但し，

メソッドのレシーバは第一引数として表す．また，メソッド列中の各メソッドの返り値は，一意な変数に格納されるとする．メソッド列 s について，$s.i$ は s の i 番目のメソッド呼び出しの返り値を表す．

メソッド列生成法では，テスト対象のメソッドの引数を，(コンストラクタを含む）メソッドの呼び出しによって生成し，テスト対象のメソッドを呼び出す．テスト対象のメソッドの呼び出しが末尾の要素であるメソッド列は，テストケースと見なせる．

メソッド $m(T_1, \cdots, T_k)$ について，m の各引数の型 T_i（$i = 1, \cdots, k$）に互換な値を生成する（複数の）メソッド列を繋ぎ，最後に m の呼び出しを加えて新たなメソッド列を生成するオペレータを $extend(m, seqs, vals)$ とする．ここで，m は最後に呼び出すメソッド，$seqs$ は m の各引数を生成するメソッド列のリスト，$vals$ は，m に渡す引数の値のリストである．

2.1.2 フィードバック駆動型メソッド列生成法

フィードバック駆動型メソッド列生成法では，新たなメソッド列が生成されると，そのメソッド列を直ちに実行する．メソッド列は，その実行結果があらかじめ決められた条件に違反するかしないかで分類される．違反しないものは後で別のメソッドの入力を生成するために，つまり別のメソッド列の一部として再利用される．一方で，違反するものはそれ自体が失敗するテストケースであり，再利用されない．このようなフィードバックによって，テストケースとして無意味なメソッド列が生成される可能性が低くなる．

フィードバック駆動型ランダムテストは，網羅率・エラー発見において，現在ランダムテストの中で最高水準の手法である．本研究の提案手法は，このフィードバック駆動型メソッド列生成法を元にしている．

フィードバック駆動型メソッド列生成アルゴリズムを Fig.2.1 に示す．アルゴリズムの引数は，テスト対象のクラスのリスト $classes$，実行結果を分類するあらかじめ決められた条件 $contracts$，生成されたメソッド列をフィルタリングする $filters$，制限時間 $timeLimit$ である．$errorSeqs$ と $nonErrorSeqs$ は，それぞれ実行結果が条件 $contracts$ に違反するメソッド列，違反しないメソッド列を蓄える集合であり，いずれも空に初期化される．アルゴリズムの戻り値は，$nonErrorSeqs$ と $errorSeqs$ のペアである．$errorSeqs$ に属すメソッド列は，失敗するテストケースとして出力される．$nonErrorSeqs$ に属すメソッド列は，回帰テストのためのテストケースとして出力される．このアルゴリズムは，与えられた制限時間内で，メソッド列の生成を繰り返す（3-26 行目）．

始めに，与えられたクラスリスト $classes$ のクラスからパブリックメソッド $m(T_1, \cdots, T_k)$

```
GenerateMethodSequences(classes, contracts, filters, timeLimit)
1   nonErrorSeqs ← φ // Their execution violates no contract.
2   errorSeqs ← φ // Their execution violates a contract.
3   while timeLimit not reached do
4     // Create a new sequence.
5     m(T_1, ··· , T_k) ← randomPublicMethod(classes)
6     ⟨seqs, vals, failed⟩
7         ← randomSeqsAndVals(nonErrorSeqs, T_1, ··· , T_k)
8     // Discard if cannot construct inputs.
9     if failed = true then
10      continue
11    end if
12    newSeq ← extend(m, seqs, vals)
13    // Discard duplicates.
14    if newSeq ∈ nonErrorSeqs ∪ errorSeqs then
15      continue
16    end if
17    // Execute the new sequence and check contracts.
18    ⟨o⃗, violated⟩ ← execute(newSeq, contracts)
19    // Classify the new sequence and outputs.
20    if violated = true then
21      errorSeqs ← errorSeqs ∪ {newSeq}
22    else
23      nonErrorSeqs ← nonErrorSeqs ∪ {newSeq}
24      setExtensibleFlags(newSeq, filters, o⃗)
25    end if
26  end while
27  return ⟨nonErrorSeqs, errorSeqs⟩
```

Fig. 2.1: Feedback-directed generation algorithm for method sequences. This figure is basically the same as Figure 3 of Pacheco et al. [38] but emphasizes that it skips a method if inputs cannot be constructed (line 9-11).

をランダムに一つ選ぶ（5 行目）．$randomSeqsAndVals$ は，m の引数の型 T_i（$i = 1, \cdots, k$）に合う値を次のように生成する（6-7 行目）．

- T_i がプリミティブ型の場合，あらかじめ決められた値（例えば，0，1.0，'a'）の中からランダムに選ぶ．
- T_i がオブジェクトリファレンス型の場合，T_i と等しい（あるいは互換な）型の値を生成するメソッド列を $nonErrorSeqs$ からランダムに一つ選ぶ．

$randomSeqsAndVals$ の戻り値は，m の引数を生成するメソッド列のリスト $seqs$，m の引数として使われる変数あるいはプリミティブ値のリスト $vals$，及び，引数の生成に失敗したかどうかを表す真偽値 $failed$ である．$failed$ は，T_i の値が生成できない場合，即ち，オブジェクトレファレンス型で，かつその型と互換な値を生成するメソッド列が $nonErrorSeqs$ に存在しないとき $true$，そうでないとき $false$ となる．また，2.1.3 で述べるように，$randomSeqsAndVals$ にはパラメータがいくつかある．

引数の生成に失敗した場合は，ループの先頭に戻る（9-11 行目）．引数が生成できた場合，$seqs$ のメソッド列と m の呼び出しを $extend$ によって繋げて新しいメソッド列 $newSeq$ を生成する（12 行目）．$newSeq$ が生成済みのメソッド列である場合，ループの先頭に戻る（14-16 行目）．

生成されたメソッド列 $newSeq$ は，直ちに実行される（18 行目）．$execute$ は，与えられたメソッド列を実行する．メソッド列中の各々のメソッドは，その実行結果が $contracts$ の各条件に違反しないか確認される．$execute$ は，実行結果の値の列 $\vec{\mathbf{o}}$ と，$contracts$ に違反したかを表す真偽値 $violated$ を返す．$violated$ は，メソッド呼び出しの実行結果が $contracts$ のどれか一つでも違反した場合は $true$，そうでない場合は $false$ となる．

生成されたメソッド列は，$violated$ の値に従って，$contracts$ に違反する場合は $errorSeqs$ に，違反しない場合は $nonErrorSeqs$ に加えられる（20-25 行目）．$setExtensibleFlags$ は，メソッド列 s，$filters$，s の実行結果列 $\vec{\mathbf{o}}$ を引数に取り，s の各メソッド呼び出しの実行結果列 $\vec{\mathbf{o}}$ に $filters$ を適用し，各 $s.i$ が新しいメソッド呼び出しの引数として使えるかどうかを示すフラグ $s.i.extensible$ を設定する．2.1.3 において，実際に使われる $contracts$ や $filter$ を述べる．

2.1.3 Randoop

Randoop [47] は，Java あるいは C# で書かれたソフトウェアを対象に，フィードバック駆動型ランダムテストを実行するツールである．Randoop は，テスト対象のクラスリストを入力とし，フィードバック駆動型メソッド列生成アルゴリズ

ムによって生成したメソッド列を，JUnit[34] 互換のテストケースとして出力する．ユーザは，$contracts$，$filters$，$timeLimit$ 等をオプションとして設定できる．

Randoop では，$randomSeqsAndVals$ のパラメータとして，以下のものがある．

- **forbid-null**
  Boolean 値（$true$ あるいは $false$）を値に取る．デフォルト値は $true$ である．forbid-null が $false$ の場合，$randomSeqsAndVals$ は，要求されている型を結果に持つメソッド列が見つからない時，代わりに $null$ を用いる．$true$ の場合，$randomSeqsAndVals$ は $failed$ を返す．

- **null-ratio**
  0 から 1 までの浮動小数点数を値に取る．デフォルト値は 0 である．$randomSeqsAndVals$ が要求されている型の値を生成できる場合，null-ratio で指定された確率に従って，代わりに $null$ を用いる．例えば，null-ratio が 0 の場合は引数として $null$ は使わず，0.5 の場合は 50%の確率で $null$ でない値の代わりに $null$ を用いる．

- **alias-ratio**
  0 から 1 までの浮動小数点数を値に取る．デフォルト値は 0 である．alias-ratio は，メソッド列中で生成された値を引数としてできるだけ再利用する確率である．例えば，alias-ratio が 0 の場合，メソッド列中で生成された値は最大でも 1 回しか使われない．1 の場合，メソッド列中で生成された値を最大限再利用してメソッド列を生成する．

- **small-tests**
  Boolean 値（$true$ あるいは $false$）を値に取る．デフォルト値は $false$ である．small-tests が $true$ の場合，引数を生成するメソッド列は短いものがより大きな確率で選ばれる．つまり，全体的にテストケースのサイズが小さくなる効果がある．具体的には，メソッド列のサイズを l としたとき，各メソッド列は $\frac{1}{l}$ の確率で選ばれる．

Randoop は，デフォルトの $contracts$ として，以下を用いている．

- 引数に null が一つもないとき，NullPointerException が発生しない．
- AssertionError が発生しない．
- $o.equals(o)$ が true で，例外が発生しない．
- $o.hashCode()$，$o.toString()$ で例外が発生しない．

また，デフォルトの $filters$ は，以下のような場合に限り $s.i.extensible$ を false に設定する．

- $s.i$ の値 **o** が，それまでに生成された全てのオブジェクト **allobjs** について，$\exists \mathbf{o}' \in \mathbf{allobjs} : \mathbf{o}.equals(\mathbf{o}')$ が成立するとき
- **o** が null のとき
- 例外が発生したとき

2.2 シンボリック実行

シンボリック実行は，変数の値を具体的な値でなくシンボルを用いて表し，プログラム中の各点に到達する条件である経路条件式（Path Condition，PC）を集める．経路条件式はシンボルを含む式であり，プログラム中で分岐が起きる度に更新される．経路条件式の更新時に，制約ソルバを用いて経路条件式が充足可能か（satisfiable）確認され，充足不能であればその経路の実行は中止してバックトラックし，充足可能であれば，経路条件式を充足する具体的な値が得られる．この得られた値を使って，その分岐を網羅するテストケースを構成する事ができる．

2.2.1 シンボリック実行の基本的なアルゴリズム

シンボリック実行の基本的なアルゴリズムを Fig.2.2 に示す．アルゴリズムの引数は，現在の経路条件式 pc，プログラム（命令列）$program$，及び変数の値をシンボルで表したシンボリック状態 $symbolicState$ である．はじめ，pc として真（true），$program$ として対象のプログラムの全ての命令列を与える．このアルゴリズムは自身を再帰的に呼び出し，pc，$program$，及び $symbolicState$ を更新していく．終了条件は，それ以上呼び出す命令が無い場合，即ち対象のプログラムが終了した場合である（1-4 行目）．

まず，対象のプログラムから次の命令 $inst$ を取り出す（5 行目）．$inst$ が分岐命令であるかそうでないかによって，処理が以下のように分かれる（6-22 行目）．

- **$inst$ が分岐命令である場合**
  シンボルで表される分岐条件式 $cond$ を用いて新しい分岐条件式 $pc \wedge cond$ 及び $pc \wedge \neg cond$ を構成し，制約ソルバでそれぞれの充足可能性を求める．充足可能である分岐先の命令を同様に実行していく（9-18 行目）．制約ソルバによって，それぞれの分岐に到達するための具体的なシンボリック変数の値を得る事ができる．

```
SymbolicExecution(pc, program, symbolicState)
1   if program is empty then
2     // Reached the end of program.
3     return
4   end if
5   inst ← the next instruction of program
6   if inst is a branching instruction then
7     cond ← the symbolic condition of inst
8     // Use constraint solver to solve satisfiability.
9     if pc ∧ cond is satisfiable then
10      // We can obtain the concrete values satisfying cond.
11      thenPart ← then part instructions of inst
12      SymbolicExecution(pc ∧ cond, thenPart, symbolicState)
13    end if
14    if pc ∧ ¬cond is satisfiable then
15      // We can obtain the concrete values satisfying ¬cond.
16      elsePart ← else part instructions of inst
17      SymbolicExecution(pc ∧ ¬cond, elsePart, symbolicState)
18    end if
19  else
20    execute inst symbolically in symbolicState
21    SymbolicExecution(pc, rest of program, symbolicState)
22  end if
```

Fig. 2.2: An algorithm of basic symbolic execution

- ***inst* が分岐命令でない場合**
  その命令をシンボリックに実行して，必要に応じて *symbolicState* を更新する．その後，以降の命令を同様に実行していく（20-21 行目）．

2.2.2 Symbolic PathFinder (SPF)

Symbolic PathFinder（SPF）[51] は，Java プログラムを対象としたモデル検査ツールである Java PathFinder（JPF）[30, 31] を拡張した，Java プログラムのシ

ンボリック実行ツールである．SPF は，Java 仮想マシン（Java Virtual Machine, JVM）上で動作する独自の JVM を実装している JPF のフレームワークを利用して，シンボリック実行を実現している．

SPF は，Gideon et al. [52] の手法を用いて文字列を含む条件式をサポートしている．従って，本研究の提案手法では，シンボリック実行部分に SPF を（一部改変して）用いている．シンボリック実行の基本的なアルゴリズムは，2.2.1 に示した通りである．文字列をサポートする上で異なる部分は，充足可能性を確認する制約ソルバだけである（Fig.2.2 では，制約ソルバは 9 行目と 14 行目で使われる）．従って，本節では，文字列を含む条件式をサポートする制約ソルバ [52] について述べる．

SPF は，経路条件式が文字列型のシンボリック変数を含む場合，以下のように充足可能性（及びそれを充足する具体的な文字列）を求める．ここでは，オートマトンを用いた手法を述べる（ビットベクタを用いる手法も存在する）．

1. 経路条件式を文字列グラフ（string graph）に変換する．文字列グラフは，頂点が文字列あるいは整数型のシンボリック変数（あるいは定数）を表しており，ラベルが一つの String の操作を表している有向ハイパーグラフ（hypergraph）である．つまり，ある辺 e にはある String の操作が対応しており，e に接続されている各頂点は，その操作の引数あるいは戻り値に対応する．

2. 文字列に対応していない従来の制約ソルバを用いて，整数型，実数型，ブーリアン型のシンボリック変数について解く．整数型のシンボリック変数の解を，文字列グラフに反映させる．それによって，各文字列の長さを固定できる．

3. 文字列グラフの各頂点にオートマトンを割り当てる．そして，文字列グラフで与えられた制約を満たすようにオートマトンを更新していく．

4. 各オートマトンが受理する文字列が見つからない場合，現在の整数解を否定する制約を経路条件式に加える事で，別の整数解を試みる．

5. 2 から 4 を，制限時間を超えるか，あるいは充足可能性が決定するまで繰り返す．

Fig.2.3 に文字列制約ソルバのアルゴリズムを示す．SolveStringConstraints の引数は，文字列を含む経路条件 pc であり，pc が充足可能である場合は $true$ を，そうでない場合は $false$ を返す．

まず，経路条件式 pc を，文字列グラフ sg に変換する（1 行目）．BuildStringGraph は，この変換を行った後，ヒューリスティクスを用いて，文字列グラフを単純化したり，グラフの形から直ちに充足不可能であるものを検出する事で，冗長な処

```
SolveStringConstraints(pc)
1   sg ← BuildStringGraph(pc)
2   while time limit not reached do
3     if ¬SolveOtherContraints(pc, sg) then
4       break
5     else
6       for each string vertex s_i ∈ sg do
7         // Attach a fixed length automaton to each string vertex.
8         M_i ← [l_i]
9       end for
10      while M_1, ··· , M_n not converged do
11        sat ← HandlePositiveEdges(sg)
12        if ¬sat then
13          pc ← interchanged constraints
14          break
15        end if
16        sat ← HandleNegativeEges(sg)
17        if ¬sat then
18          pc ← interchanged constraints
19          break
20        end if
21      end while
22      if pc not interchanged ∧ M_1, ··· , M_n converged then
23        return true
24      end if
25    end if
26  end while
27  return false
```

Fig. 2.3: The automaton-based algorithm for solving string constraints

理を省く．例えば，明らかに，辺 equals をグラフから削除する事ができる．また，文字列グラフの文字列を表す各頂点には，長さ（整数）を表す頂点を対応させる．

そして，従来の制約ソルバを用いて，整数や実数，ブーリアン値についての制

約を解く（3 行目）．得られた整数解によって，各文字列の長さを固定する．各々の文字列頂点に，対応する長さのオートマトンを割り当てる（6-9 行目）．

その後，各オートマトンが収束するまで，文字列グラフの各辺の種類に基づいて，各文字列のオートマトンを更新していく（10-26 行目）．HandlePositiveEdges は，文字列グラフの，否定を含まない操作の各辺について処理を行い，オートマトンを更新していく（11 行目）．否定を含まない操作の辺を処理する場合，その辺に接続された頂点のオートマトン M は，$M' = M \cap X$ というように，共通部分を取る操作に還元される．しかし，否定を含む操作の辺を処理する場合，3 つ以上の頂点が関係したり，別の否定を含む制約が存在するとき，同様に処理する事はできない．そこで，HandleNegativeEdges は，否定辺に接続された頂点のオートマトンが受理する文字列を列挙し，全ての組み合わせを試して充足するものがないか調べる（16 行目）．

HandlePositiveEdges や HandleNegativeEdges において，制約を満たすオートマトンが存在しない場合，経路条件式 pc に現在採用している整数解の否定を加える事で，別の整数解（長さ）でリスタートする（13 行目と 18 行目）．

制約を満たすオートマトン群が見つかった場合，$true$ を返す（23 行目）．

2.3 Randsym: 提案手法

本研究の提案手法 Randsym は，フィードバック駆動型ランダムテスト [38] と，文字列をサポートしたシンボリック実行 [52] を組み合わせたものである．基本的に，以下のようにテストケースを生成する．

1. フィードバック駆動ランダムテストによって，ベースとなるメソッド列（テストケース）を生成する．
2. 1 で得られたテストケース群によって網羅できなかった分岐の中から，シンボリック実行を行う分岐を選択する．このとき，その分岐を含むメソッド，そしてそのメソッドを対象とするメソッド列を得る．
3. 2 で得られたメソッド列中の全てのプリミティブ型の変数をシンボリック変数としてシンボリック実行を行い，得られた値を使って対象の分岐を網羅する新しいテストケースを生成する．

2.3.1 Randsym のアルゴリズム

Fig.2.4 に，Randsym のアルゴリズムを示す．Randsym は，テスト対象のクラスリスト $classes$ を引数に取り，テストケースを返す．

```
Randsym(classes)
1   // Execute feedback directed random testing.
2   seqs ← FeedbackDirectedRT(classes)
3   // Get uncovered branches and corresponding sequences.
4   ⟨uBranches, uSeqs⟩ ← getUncoveredBranchesAndSeqs(classes, seqs)
5   // Select branches and corresponding sequences to be targeted by symbolic execution.
6   ⟨uBranches, uSeqs⟩ ← selectBranchesAndSeqs(uBranches, uSeqs)
7   for each seq ∈ uSeqs do
8     symVars ← φ
9     for each var ∈ seq do
10      if var is primitive then
11        symVars ← symVars ∪ {var}
12      end if
13    end for
14    ⟨values, success⟩ ← do symbolic execution on symVars
15    if success then
16      seqs ← seqs ∪ {replaceVars(seq, values)}
17    end if
18  end for
19  return seq
```

Fig. 2.4: Algorithm of the proposed method Randsym

まず，与えられたクラス群に対して，フィードバック駆動型ランダムテストを行い，テストケースでもあるメソッド列 *seqs* を得る（2 行目）．次に，*seqs* によって網羅できなかった分岐及び，それらの分岐が属すメソッドを対象とするメソッド列群を計算する（4 行目）．さらに，それらの分岐の中から，シンボリック実行の対象とするものを選びだす（6 行目）．ここでは,「メソッド列によって分岐の片側はカバーされていたもの」を選ぶ．そうする事によって，シンボリック実行によって残りの側もカバーされる可能性を高める事ができる．

そうして選んだ各メソッド列 *uSeqs* について，その中に現れるプリミティブな変数を全てシンボリック変数とし，その変数上でシンボリック実行を行う（7-18 行目）．ここで，プリミティブな値とは，整数型，実数型，ブーリアン型，文字型に加え，当然も文字列型も含む．プリミティブな値のみをシンボリック変数とす

る事で，シンボリック実行によって解が得られる可能性を高める事ができる．また，メソッド列は引数オブジェクトの生成過程を表しているから，その途中のプリミティブな値をシンボリック変数とする事で，複雑なオブジェクトが必要である場合でも，シンボリック実行が有効に働くと考えられる．

シンボリック実行によって解が見つかった場合，*values* としてその解が，*success* として *true* が得られる（14 行目）．その場合，得られた値 *values* を使って対象の分岐を網羅するテストケースを作成する事ができる（16 行目）．

2.3.2 Randsym の実装

Randsym は，フィードバック駆動型ランダムテスト，シンボリック実行，そしてそれらを統合するプログラムによって構成されている．

- **フィードバック駆動型ランダムテスト**
  この部分では，Randoop [47] を用いている．また，網羅されていない分岐を調べるために，Randoop 内部に実装されているクラスを利用している．その実装では，テスト対象のソースコードに分岐をカウントするコードを直接埋め込む必要があるため，テスト対象のソースコードが必要となる．

- **シンボリック実行**
  この部分では，Symbolic PathFinder （SPF）[51, 52] の一部を改変したものを用いている．文字列グラフにおいて，文字列の長さを l と置いたとき，$l > 0$ が全ての文字列頂点に制約条件として付与されるという問題を修正した．$l \geq 0$ が正しい制約条件である．また，SPF には制約ソルバによって得られた解を取り出すインターフェースが備わっていなかったため，独自のリスナを実装して対処した．

- **統合部分**
  統合部分は，複数の Java クラスからなる．それぞれのクラスは，Apache Ant [56] によって統合され，自動化されている．Randoop とは，通常のテキストファイルを介してデータをやり取りしている．SPF を実行するために，各メソッド列毎にドライバファイルを作り，一つ一つコンパイルしている．SPF からの出力は，SPF 側に実装した独自のリスナが XStream [57] を用いてオブジェクトを XML ファイルにシリアライズしたものを読み込んでいる．

第3章　評価

本章では，以下の研究課題に基づいて提案手法を評価する．

- **研究課題**：提案手法 Randsym は，Fig.1.3 に示したような文字列操作を分岐条件として含むプログラムについて，既存手法であるフィードバック駆動型ランダムテストと比べて網羅率が高いテストケースを生成できるか．

以降では，この研究課題の達成度を確認するための実験を述べる．3.1.1 で実験の設定を述べ，3.1.2，3.1.3 においてそれぞれ実験の結果及び考察を述べる．

3.1　実験

3.1.1　設定

ベンチマーク

先に述べた研究課題の達成度を評価するために，文字列処理が分岐条件として含まれるメソッドを 3 つ持つクラスを用意した．

Fig.3.1 は，これら 3 つのメソッドに共通のクラス部分である．ショートオプションとロングオプションの接頭辞を値として持つ二つの定数が定義されている（4-5 行目）．また，このクラスのオブジェクトの状態として，文字列の配列 args がある（6 行目）．これは，引数を格納する配列であり，コンストラクタで与えられたものが設定される（8-10 行目）．Fig.3.2，Fig.3.3，Fig.3.4 は，12 行目以降に書かれている．

Fig.3.2 の longOptionExists は，Fig.1.3 のそれと同じメソッドである．引数として指定されたロングオプション option を持つ引数が存在するかどうかを調べる．longOptionExists において網羅する事が難しい分岐は，8 行目のものである．その分岐の真側を通過するには，ロングオプションの接頭辞であるハイフン二つから始まり，かつ 4 文字以上の文字列が args に存在する必要がある．

Fig.3.3 の shortOptionExists は，引数として指定されたショートオプション option （1 文字）を持つ引数が存在するかどうかを調べる．shortOptionExists にお

```
1  package simpleprog;
2
3  public class ArgsParser {
4    public static final String LONG_OPTION_INDICATOR = "--";
5    public static final String SHORT_OPTION_INDICATOR = "-";
6    private String[] args;
7
8    public ArgsParser(String[] args) {
9      this.args = args;
10   }
11
12   // Target method definitions here...
13 }
```

Fig. 3.1: The structure of the benchmark class for evaluation

いて網羅する事が難しい分岐は，7 行目の真側の分岐と，9 行目の真側の分岐である．まず，7 行目の真側の分岐をカバーするには，ハイフンともう 1 文字からなる文字列が args に存在する必要がある．また，9 行目の真側の分岐をカバーするには，それに加えて，ハイフンの後の文字が，このメソッドの引数として与えられた文字に等しくなければならない．

Fig.3.4 の countNormalArgs は，args の中に存在する非オプションの引数を数える．countNormalArgs において網羅する事が難しい分岐は，6-7 行目の偽側の分岐である．それをカバーするには，ハイフンで始まる文字列が args に存在する必要がある．

方法

Randoop と Randsym を使って，前述したベンチマークのテストケースを生成し，フィードバック駆動型ランダムテストと提案手法の網羅率を比較する．網羅率は，生成された JUnit テストケースに対して，JaCoCo [58] を用いて測る．実験は，Mac OS X 10.9.5，8 GB RAM，Intel Core i7 1.7GHz のマシン上で行う．パラメータの設定は以下の通りである．

```
1   public boolean longOptionExists(String option) {
2     boolean isLongOption = (option.length() > 1);
3
4     if (isLongOption) {
5       String searchFor = LONG_OPTION_INDICATOR + option;
6       for (int i = 0; i < args.length; i++) {
7         if (args[i] != null) {
8           if (args[i].equals(searchFor)) {
9             return true;
10          }
11        }
12      }
13    }
14
15    return false;
16  }
```

Fig. 3.2: A benchmark method for evaluation

- **Randoop**
  パラメータは，時間制限（timelimit）を除いてデフォルトとする．時間制限は600秒（10分）とする．これは，対象のベンチマークの規模のプログラムに対して，網羅率が飽和するのに十分な時間である．

- **Randsym**
  Randoop 部分のパラメータは，時間制限（timelimit），入力制限（inputlimit）を除いてデフォルトとする．入力制限は，生成されるメソッド列（テストケース）の数を制限するパラメータである．時間制限はなし，入力制限は100個とする．勿論これは，10分間で生成されるテストケースの数に比べて遥かに少ない．SPF 部分のパラメータは，ソルバとして choco（symbolic.dp=choco），文字列ソルバとしてオートマトン（symbolic.string_dp=automata）を用いる．

```
1   public boolean shortOptionExists(String option) {
2     boolean isShortOption = (option.length() == 1);
3
4     if (isShortOption) {
5       for (int i = 0; i < args.length; i++) {
6         if (args[i] != null) {
7           if (args[i].length() > 1 && args[i].startsWith(SHORT_OPTION_INDICATOR)) {
8             int index = args[i].indexOf(option);
9             if (index > 0) {
10              return true;
11            }
12          }
13        }
14      }
15    }
16
17    return false;
18  }
```

Fig. 3.3: A benchmark method for evaluation

3.1.2 結果

実験結果を Table.3.1，Table.3.2 に示す．Randoop の合計実行時間は 10 分 18 秒，Randsym の合計実行時間は 5 分 4 秒であった．

Table.3.1，Table.3.2 から，Randsym の実行時間は Randoop のそれの半分であるが，ベンチマークの全てのメソッドにおいて，網羅率（命令網羅率と分岐網羅率の両方）が向上している事が分かる．Randsym によって新たに網羅された分岐を詳しく述べる．

shortOptionExists（Fig.3.3）においては，Randoop と Randsym の間で最も顕著な差が出た．Randoop では，そもそも 1 行目の isShortOption を true にするような引数 option を生成できなかった．従って，Randoop は，5-14 行目の命令を全く網羅できなかった．一方，Randsym は，7 行目の分岐と 9 行目の分岐の真側を網羅しており，文字列を含む難しい分岐を網羅できている．それらを網羅す

```
1   public int countNormalArgs() {
2     int count = 0;
3
4     for (int i = 0; i < args.length; i++) {
5       if (args[i] != null) {
6         if (!args[i].startsWith(SHORT_OPTION_INDICATOR) &&
7               !args[i].startsWith(LONG_OPTION_INDICATOR)) {
8           count++;
9         }
10      }
11    }
12
13    return count;
14  }
```

Fig. 3.4: A benchmark method for evaluation

Table. 3.1: Coverage results of Randoop

Target method	**Instruction Cov.** [%]	**Branch Cov.** [%]
shortOptionExists(String)	20	14
longOptionExists(String)	95	80
countNormalArgs()	100	62
ArgsParser(String[])	100	n/a
Total	68 (91 of 134)	47 (15 of 32)

Randoop's timelimit is 600 seconds. The total run time was 618 seconds.

る Randsym が生成したテストケースの例を Fig.3.5 に示す．しかし，それらの分岐の偽側は網羅できなかった．その理由は，Randsym 内部のバグに因ると考えられるが，現在調査中である．

longOptionExists（Fig.3.2）においては，Randoop は 8 行目の分岐の真側を網羅できなかったが，Randsymb は網羅できた．それを網羅する Randsym が生成したテストケースの例を Fig.3.6 に示す．

countNormalArguments（Fig.3.4）においては，Randoop は 6-7 行目の偽側の分

Table. 3.2: Coverage results of Randsym

Target method	**Instruction Cov.** [%]	**Branch Cov.** [%]
shortOptionExists(String)	96	64
longOptionExists(String)	100	90
countNormalArgs()	100	75
ArgsParser(String[])	100	n/a
Total	99 (132 of 134)	75 (24 of 32)

Randsym's inputlimit is 100. The total run time was 304 seconds.

```
1  public void test1() throws Throwable {
2    java.lang.String var0 = "- ";
3    java.lang.String[] var1 = new java.lang.String[] { var0 };
4    simpleprog.ArgsParser var2 = new simpleprog.ArgsParser(var1);
5    java.lang.String var3 = " ";
6    boolean var4 = var2.shortOptionExists(var3);
7  }
```

Fig. 3.5: A test case generated by Randsym that covers branches in line 7 and 9 in Fig.3.3

```
1  public void test116() throws Throwable {
2    java.lang.String var0 = "-- ";
3    java.lang.String[] var1 = new java.lang.String[] { var0 };
4    simpleprog.ArgsParser var2 = new simpleprog.ArgsParser(var1);
5    simpleprog.ArgsParser var3 = new simpleprog.ArgsParser(var1);
6    java.lang.String var4 = " ";
7    boolean var5 = var3.longOptionExists(var4);
8  }
```

Fig. 3.6: A test case generated by Randsym that covers branch in line 8 in Fig.3.2

```
1   public void test49() throws Throwable {
2     java.lang.String var0 = "−";
3     java.lang.String[] var1 = new java.lang.String[] { var0 };
4     simpleprog.ArgsParser var2 = new simpleprog.ArgsParser(var1);
5     int var3 = var2.countNormalArgs();
6   }
```

Fig. 3.7: A test case generated by Randsym that covers branch in line 6-7 in Fig.3.4

岐は網羅できなかったが，Randsym は網羅できた．それを網羅する Randsymb が生成したテストケースの例を Fig.3.6 に示す．

3.1.3 考察

実験結果から，既存のランダムテストでは網羅できない分岐が提案手法によって網羅できるようになったため，少なくとも本実験の単純なベンチマーク上では，研究課題が達成されたと言える．しかし，実際に使われているソフトウェアでは，文字列処理を含むより複雑な分岐が有り得る．例えば，文字列処理の否定を含む分岐条件で，かつそれに関わる文字列が全てシンボリック変数である場合，SPF の限界から，その分岐を網羅する入力を生成する事は難しい．

また，現在の Randsym の実装には性能に影響を及ぼすバグが含まれている可能性が高い．SPF を実行する際に，メソッド列毎にドライバファイルを作ってコンパイルしているため，大規模なソフトウェアに適用する場合，そのオーバーヘッドが無視できない．さらに，SPF を実行する分岐を選ぶ際にも，Randoop によって片側が網羅されている分岐全てとしているため，余分なシンボリック実行が行われている可能性が高い．従って，研究課題をさらに追求するには，Randsym の実装を見直し，実用的な規模のベンチマークで実験する必要がある．

第4章　結論及び今後の課題

本研究では，フィードバック駆動型ランダムテストと文字列をサポートしたシンボリック実行を組み合わせたテストケース自動生成手法を提案した．文字列を含む条件を持つ分岐に着目する事でランダムテストの網羅率の向上を試みた結果，提案手法は，基本的な小さいベンチマークにおいて，ランダムテストでは網羅できない分岐をより短時間で網羅できる事が分かった．

今後の課題として，以下が挙げられる．

- Randsym の実装を見直し，性能に影響するバグを取り除く．
- 現在，Randsym はシンボリック実行を行うために，メソッド列を一つ一つ書き出してコンパイルしている．これは実用的な規模のソフトウェアに適用した場合，分岐数が多いため，そのオーバーヘッドが無視できない．従って，シンボリック実行をオンザフライで行えるように実装を見直す．
- 効率的にシンボリック実行を行えるように，分岐の選別方法を工夫する．
- 以上の Randsym の性能上の問題を解決して，より大きい実用的な規模のベンチマークで評価する．
- Randoop 以外の様々な手法と性能を比較する．
- 文字列以外の複雑な条件に対しても対処できるフレームワークを模索する．

参考文献

[1] Lions, Jacques-Louis., et al.: Ariane 5 flight 501 failure, Report by the enquiry board. `http://www.di.unito.it/~damiani/ariane5rep.html` (1996).

[2] Dowson, Mark.: The Ariane 5 software failure, *SIGSOFT Softw. Eng. Notes*, Vol. 22, No. 2, pp. 84– (1997).

[3] Leveson, N.G.. and Turner, C.S..: An investigation of the Therac-25 accidents, *Computer*, Vol. 26, No. 7, pp. 18–41 (1993).

[4] OpenSSL Project, `http://www.openssl.org/`.

[5] CVE - CVE-2014-0160, `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160`.

[6] Dierks, T.. and Rescorla, E..: The Transport Layer Security (TLS) Protocol Version 1.2, RFC 5246, RFC Editor (2008).

[7] Turing, Alan M.: On computable numbers, with an application to the Entscheidungsproblem, *Proceedings of the London mathematical society*, Vol. 42, No. 2, pp. 230–265 (1936).

[8] Rice, Henry Gordon.: Classes of recursively enumerable sets and their decision problems, *Transactions of the American Mathematical Society*, Vol. 74, No. 2, pp. 358–366 (1953).

[9] Hoare, C. A. R..: An axiomatic basis for computer programming, *Commun. ACM*, Vol. 12, No. 10, pp. 576–580 (1969).

[10] Floyd, Robert W.: Assigning meanings to programs, *Mathematical aspects of computer science*, Vol. 19, No. 19-32, p. 1 (1967).

[11] Dijkstra, Edsger W..: Guarded commands, nondeterminacy and formal derivation of programs, *Commun. ACM*, Vol. 18, No. 8, pp. 453–457 (1975).

[12] Moura, Leonardo. and BjÃÿrner, Nikolaj.: Satisfiability Modulo Theories: An Appetizer, in Oliveira, MarcelVinÃŋciusMedeiros. and Woodcock, Jim. eds., *Formal Methods: Foundations and Applications*, Vol. 5902 of *Lecture Notes in Computer Science*, pp. 23–36, Springer Berlin Heidelberg (2009).

[13] Moura, de Leonardo., Dutertre, Bruno. and Shankar, Natarajan.: A tutorial on satisfiability modulo theories, in *Computer Aided Verification*, pp. 20–36Springer (2007).

[14] Shankar, Natarajan.: Automated deduction for verification, *ACM Comput. Surv.*, Vol. 41, No. 4, pp. 20:1–20:56 (2009).

[15] Barrett, Clark. and Tinelli, Cesare.: Cvc3, in *Computer Aided Verification*, pp. 298–302Springer (2007).

[16] Moura, Leonardo. and BjÃÿrner, Nikolaj.: Z3: An Efficient SMT Solver, in Ramakrishnan, C.R.. and Rehof, Jakob. eds., *Tools and Algorithms for the Construction and Analysis of Systems*, Vol. 4963 of *Lecture Notes in Computer Science*, pp. 337–340, Springer Berlin Heidelberg (2008).

[17] Paulson, Lawrence C. and Wenzel, Markus.: *Isabelle/HOL: a proof assistant for higher-order logic*, Vol. 2283, Springer (2002).

[18] Bertot, Yves. and Castéran, Pierre.: *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*, springer (2004).

[19] Clarke, EdmundM.. and Emerson, E.Allen.: Design and synthesis of synchronization skeletons using branching time temporal logic, in Kozen, Dexter. ed., *Logics of Programs*, Vol. 131 of *Lecture Notes in Computer Science*, pp. 52–71, Springer Berlin Heidelberg (1982).

[20] Queille, J.P.. and Sifakis, J..: Specification and verification of concurrent systems in CESAR, in Dezani-Ciancaglini, Mariangiola. and Montanari, Ugo. eds., *International Symposium on Programming*, Vol. 137 of *Lecture Notes in Computer Science*, pp. 337–351, Springer Berlin Heidelberg (1982).

[21] Clarke, E. M.., Emerson, E. A.. and Sistla, A. P..: Automatic verification of finite-state concurrent systems using temporal logic specifications, *ACM Trans. Program. Lang. Syst.*, Vol. 8, No. 2, pp. 244–263 (1986).

[22] Clarke, Edmund M.: The birth of model checking, in *25 Years of Model Checking*, pp. 1–26, Springer (2008).

[23] Bryant, R.E..: Graph-Based Algorithms for Boolean Function Manipulation, *Computers, IEEE Transactions on*, Vol. C-35, No. 8, pp. 677–691 (1986).

[24] Clarke, Edmund., Grumberg, Orna., Jha, Somesh., Lu, Yuan. and Veith, Helmut.: Counterexample-Guided Abstraction Refinement, in Emerson, E.Allen. and Sistla, AravindaPrasad. eds., *Computer Aided Verification*, Vol. 1855 of *Lecture Notes in Computer Science*, pp. 154–169, Springer Berlin Heidelberg (2000).

[25] Cousot, Patrick. and Cousot, Radhia.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pp. 238–252, Los Angeles, California (1977), ACM.

[26] Cousot, Patrick. and Cousot, Radhia.: Systematic design of program analysis frameworks, in *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '79, pp. 269–282, San Antonio, Texas (1979), ACM.

[27] Valmari, Antti.: A stubborn attack on state explosion, in Clarke, EdmundM.. and Kurshan, RobertP.. eds., *Computer-Aided Verification*, Vol. 531 of *Lecture Notes in Computer Science*, pp. 156–165, Springer Berlin Heidelberg (1991).

[28] Beyer, Dirk., Henzinger, ThomasA.., Jhala, Ranjit. and Majumdar, Rupak.: The software model checker Blast, *International Journal on Software Tools for Technology Transfer*, Vol. 9, No. 5-6, pp. 505–525 (2007).

[29] Holzmann, Gerard J.: The model checker SPIN, *IEEE Transactions on software engineering*, Vol. 23, No. 5, pp. 279–295 (1997).

[30] Lerda, Flavio. and Visser, Willem.: Addressing dynamic issues of program model checking, in *Model Checking Software*, pp. 80–102, Springer (2001).

[31] Visser, Willem., Havelund, Klaus., Brat, Guillaume., Park, SeungJoon. and Lerda, Flavio.: Model checking programs, *Automated Software Engineering*, Vol. 10, No. 2, pp. 203–232 (2003).

[32] Bourque, P.. and R.E. Fairley, eds..: *Guide to the Software Engineering Body of Knowledge, Version 3.0*, IEEE Computer Society (2014).

[33] Runeson, P..: A survey of unit testing practices, *Software, IEEE*, Vol. 23, No. 4, pp. 22–29 (2006).

[34] JUnit, `http://junit.org/`.

[35] Zhu, Hong., Hall, Patrick A. V.. and May, John H. R..: Software Unit Test Coverage and Adequacy, *ACM Comput. Surv.*, Vol. 29, No. 4, pp. 366–427 (1997).

[36] Tassey, Gregory.: The economic impacts of inadequate infrastructure for software testing, *National Institute of Standards and Technology, RTI Project*, Vol. 7007, No. 011 (2002).

[37] Csallner, Christoph. and Smaragdakis, Yannis.: JCrasher: an automatic robustness tester for Java, *Software: Practice and Experience*, Vol. 34, No. 11, pp. 1025–1050 (2004).

[38] Pacheco, Carlos., Lahiri, Shuvendu K., Ernst, Michael D. and Ball, Thomas.: Feedback-directed random test generation, in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pp. 75–84IEEE (2007).

[39] Ciupa, I.., Leitner, A.., Oriol, M.. and Meyer, B..: ARTOO, in *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pp. 71–80 (2008).

[40] Thummalapenta, Suresh., Xie, Tao., Tillmann, Nikolai., Halleux, de Jonathan. and Schulte, Wolfram.: MSeqGen: Object-oriented Unit-test Generation via Mining Source Code, in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pp. 193–202, Amsterdam, The Netherlands (2009), ACM.

[41] Jaygarl, Hojun., Kim, Sunghun., Xie, Tao. and Chang, Carl K..: OCAT: Object Capture-based Automated Testing, in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pp. 159–170, Trento, Italy (2010), ACM.

[42] Zhang, Sai., Saff, David., Bu, Yingyi. and Ernst, Michael D..: Combined Static and Dynamic Automated Test Generation, in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pp. 353–363, Toronto, Ontario, Canada (2011), ACM.

[43] Fraser, Gordon. and Arcuri, Andrea.: EvoSuite: Automatic Test Suite Generation for Object-oriented Software, in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pp. 416–419, Szeged, Hungary (2011), ACM.

[44] Sen, Koushik.: Concolic Testing, in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pp. 571–572, Atlanta, Georgia, USA (2007), ACM.

[45] Inkumsah, K.. and Xie, Tao.: Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution, in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pp. 297–306 (2008).

[46] Pacheco, Carlos. and Ernst, MichaelD..: Eclat: Automatic Generation and Classification of Test Inputs, in Black, AndrewP.. ed., *ECOOP 2005 - Object-Oriented Programming*, Vol. 3586 of *Lecture Notes in Computer Science*, pp. 504–527, Springer Berlin Heidelberg (2005).

[47] Pacheco, Carlos. and Ernst, Michael D.: Randoop: feedback-directed random testing for Java, in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pp. 815–816ACM (2007).

[48] Chen, T.Y.., Leung, H.. and Mak, I.K..: Adaptive Random Testing, in Maher, MichaelJ.. ed., *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, Vol. 3321 of *Lecture Notes in Computer Science*, pp. 320–329, Springer Berlin Heidelberg (2005).

[49] Jaygarl, Hojun., Chang, Carl K.. and Kim, Sunghun.: Practical Extensions of a Randomized Testing Tool, *2009 33rd Annual IEEE International Computer Software and Applications Conference*, pp. 148–153 (2009).

[50] King, James C.: Symbolic execution and program testing, *Communications of the ACM*, Vol. 19, No. 7, pp. 385–394 (1976).

[51] Pasareanu, Corina S., Mehlitz, Peter C., Bushnell, David H., Gundy-Burlet, Karen., Lowry, Michael., Person, Suzette. and Pape, Mark.: Combining unit-level symbolic execution and system-level concrete execution for testing NASA software, in *Proceedings of the 2008 international symposium on Software testing and analysis*, pp. 15–26ACM (2008).

[52] Redelinghuys, Gideon., Visser, Willem. and Geldenhuys, Jaco.: Symbolic execution of programs with strings, in *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, pp. 139–148ACM (2012).

[53] Sen, Koushik. and Agha, Gul.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools, in *Computer Aided Verification*, pp. 419–423Springer (2006).

[54] Sen, Koushik.: Concolic testing, in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 571–572ACM (2007).

[55] Fraser, G.. and Arcuri, A..: Sound empirical evidence in software testing, in *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 178–188 (2012).

[56] Apache Ant, `http://ant.apache.org/`.

[57] XStream, `http://xstream.codehaus.org/`.

[58] EclEmma - JaCoCo Java Code Coverage Library, `http://www.eclemma.org/jacoco/`.