ソフトウェアテストを評価するための バグ入りアプリケーションの自動生成

安積 直道　丹野 治門　岩崎 英哉

アプリケーションにバグが含まれているかを確認するには，テストスイートという入力を与え，期待した動作を行うかを確認するソフトウェアテストを行う必要がある．テストスイートのバグ検出能力の評価手法として，アプリケーションに意図的にバグを埋め込みテストスイートを入力として実行し，どれほどのバグを検出できるかを測定するミューテーション解析がある．これを実用的に行うためには，適切なアプリケーションの選定や適切なバグの含まれるベンチマークの作成など様々な課題がある．本論文では, あらかじめ用意されたアプリケーションの構成要素を組み合わせて，バグ入りアプリケーションを自動的に生成するツールを設計し実装した．さらに，現実的に存在しえないような不自然な処理を行うアプリケーションの生成を防ぐため，メソッドの引数と戻り値の型の性質による制約と，処理順に関する制約を指定可能にした．その結果，不自然な処理を行うアプリケーションの生成を防ぐことができた．

Recently there are various applications everywhere. In order to check bugs in developed applications, we need to give test inputs called test suites to the applications. There is a method called mutation analysis to measure a bug detection capability of test suites. It measures how many bugs can be detected by intentionally embedding bugs in the application and executing them using test suites. It has some problems such as selecting an appropriate application and creating a benchmark that contains appropriate bugs. In this paper, we present a tool that automatically generates buggy applications in Java. In the tool, we can specify two kinds of constraints to avoid unnatural applications. The first is a constraint about properties of method arguments and return values in the application. The second is a constraint that specifies how to connect methods. By specifying suitable constraints, the tool can generate various buggy applications of high quality.

1　はじめに

近年, コンピュータやスマートフォンなどの電子機器から，鉄道システムや市役所及び図書館システムなどの公共の機関まで，至るところで様々なアプリケーションが利用されている．また, 世の中をより便利にするために，現在も多くの企業によって多様なアプリケーションが開発されている．しかし, 実際に信頼性の高いアプリケーションを開発することは容易ではない．その理由のひとつは，多くの場合アプリケーションにはバグが混入してしまい，それらを発見して修正する作業には多大な労力が必要であるためである．

一般的に, アプリケーションのバグを検出するには，そのアプリケーションにテストとなる入力を与え，開発者が期待した動作を行うかどうかを確認する必要がある．これをソフトウェアテスト [8] と言う．また，このテスト入力の集合をテストスイートと呼ぶ．ソフトウェアテストにおいて，テストスイートがバグを検出する十分な能力を持つかを評価する方法として，ミューテーション解析 [4] [9] という手法が広く知られている．

ミューテーション解析では，開発されたアプリケーションに意図的にバグを埋め込み，テストスイートを

Automatic generation of buggy applications for evaluating software tests.

Naomichi Asaka, 電気通信大学大学院情報理工学研究科, Dept. of Information and Computer Science, The University of Electro-Communications.

Haruto Tanno, 日本電信電話株式会社　ソフトウェアイノベーションセンタ, Dept. Software Innovation Center, Nippon Telegraph and Telephone Corporation.

Hideya Iwasali, 電気通信大学大学院情報理工学研究科, Dept. of Information and Computer Science, The University of Electro-Communications.

入力として与えてそのアプリケーションを実行した際に，どれほどのバグを検出できるかを測定することで，テストスイートのバグ検出能力を評価する．しかし，実用的なミューテーション解析を行うためには，適切なアプリケーションを選定し，そのアプリケーションに対してミューテーション解析ツールを適用して，適切なバグが含まれるベンチマークを作成する必要がある．これらは，多大な労力を必要とする．

本論文では，あらかじめ用意されたアプリケーションの構成要素を組み合わせて，ソフトウェアテストを評価するためのバグ入りアプリケーションを自動的に生成するツールを設計し実装した．そして，このツールを用いて，テストスイートのバグ検出能力測定用の，様々なバリエーションのバグ入りアプリケーションを自動生成した．また，その際に現実的に存在しえないような不自然なアプリケーションの生成を防ぐため，型の性質に関する制約と操作順に関する制約を適用可能にした．

結果として，多くのバグ入りアプリケーションを生成でき，かつ現実的に存在しえないような不自然な処理を行うアプリケーションの生成を抑制できた．

2 バグ入りアプリケーション

2.1 バグ入りアプリケーションの必要性

バグ入りアプリケーションは，主に以下の 3 つのケースで必要となる．

ケース 1 自動生成したテストスイートのバグ検出能力の測定

ケース 2 手動テスト支援の手法の評価

ケース 3 複数のデバッグ手法の比較評価

2.1.1 ケース 1

テストスイートの自動生成については多くの研究が行われている [2]．自動生成したテストスイートのバグ検出能力の評価を行いたい場合，あるバグ入りアプリケーションにそのテストスイートを与えて実行し，どれほどのバグを検出できるかを測定する必要がある．

2.1.2 ケース 2

手動テストの例として，記述式テストと探索的テストという 2 つのテスト手法がある．記述式テストとは事前にテストケースを用意しておき，それに基づいてテスト行う手法である．それに対して，探索的テストとは事前にテストケースを用意せず，それまでのテスト実行の結果などをもとに動的にテストケースを作成する手法である．これらの手法を比較実験 [1] する場合，各テストのバグ検出能力は重要な指標の 1 つであり，これを計測するためにはバグ入りアプリケーションが必要となる．

2.1.3 ケース 3

デバッグ手法 [10] を比較する場合には、各手法においてどれほどのバグを特定できるかを測定するためにバグ入りアプリケーションが必要となる．

2.2 既存のバグ入りアプリケーション生成方法

バグ入りアプリケーションを生成する既存の方法には，主に以下の 2 つがある．

1 つ目は，現実的なバグが含まれるアプリケーションを生成する方法である．この方法に基づいてバグ入りアプリケーションを生成する研究として，Just らによる研究 [5] や Gyimesi らによる研究 [3] がある．これらの研究では，実際のアプリケーション開発プロジェクトのバグ報告データに基づいてバグを混入させる．そのため，開発者が実際に起こしそうな現実的なバグを考えることができる．しかし，これらはあらかじめ用意されているベンチマークの分しかアプリケーションが存在しないため，生成できるアプリケーションの数が限られている．

2 つ目は，ミューテーション解析によって人為的にバグ入りアプリケーションを生成する方法である．ミューテーション解析では，テスト対象のソフトウェアを意図的に書き換えることにより，ミュータントと呼ばれる人為的なバグを含むソフトウェアを生成する．ミューテーション解析を用いる方法では，多くのバグ入りアプリケーションを自動生成できるが，等価ミュータント [6] を手動で取り除く作業が必要になりコストがかかるという問題点がある．

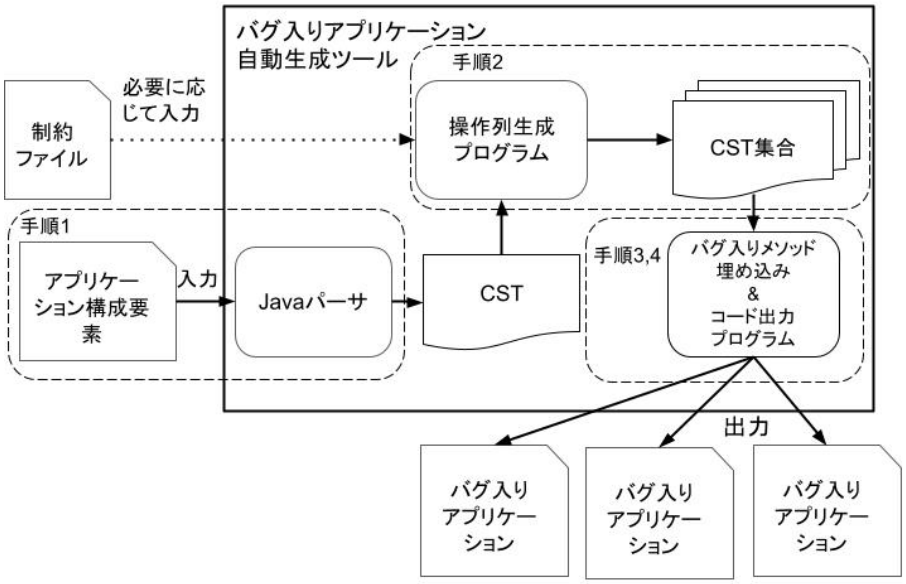

図 1: ツールの全体像

3 提案手法

3.1 スコープと要件

3.1.1 スコープ

既存手法の問題点を解決するため，本論文では事前に作成して用意された Java アプリケーションに対する構成要素をもとに，アプリケーションの様々なバリエーションを自動生成する．ここで，構成要素とは，対象アプリケーションを構成するバグのないメソッド定義を指す．また，各構成要素のメソッドに対して，変異を混入させたバグ入りメソッドもあらかじめ用意しておく．これらのバグ入りメソッドは，等価ミュータントを発生させないことがあらかじめ保証されているものとする．このようなバグ入りメソッドを用意する方法としては，手動による方法，既存のミューテーション解析技術 [7] を活用する方法などが考えられる．

ユーザが指定した入出力の型をもとに，構成要素のメソッドを直線的に接続したメソッド列を多数生成する．個々のメソッド列を以下，操作列と呼ぶ．操作列を構成するメソッドの一部をあらかじめ用意しておいたバグ入りメソッドに置換し，バグを含む操作列とする．最後にこれを Java プログラムとして出力することにより，バグ入りアプリケーションを生成する．

3.1.2 要件

提案するツールは，以下の 3 つの要件を満たすように設計する．

要件 1 事前準備のコストをなるべく小さくしたうえで，生成されうる多くのバリエーションのアプリケーションを生成する．

要件 2 生成されたアプリケーションは構文誤りや型の誤りがなく，コンパイルが正常終了する．

要件 3 現実的に存在しえない不自然なアプリケーションの生成を防ぐ．

3.2 特徴

前節の要件を満たすため，本ツールは以下の特徴を持つものとする．

- 事前に人為的に用意したアプリケーションの構成要素を組み合わせ，様々なバグ入りアプリケーションの自動生成を可能とする．(要件 1 に対応)

- 構成要素を組み合わせる際，型の整合性を保証することで，正しくコンパイルが可能なプログラムを生成する．(要件 2 に対応)
- 型の性質と操作順に関する制約を人が与えて適用することを可能にし，その制約を満たすように構成要素を組み合わせることで，不自然なアプリケーションの生成を防ぐ．(要件 3 に対応)

3.3 全体像

本ツールの全体像を図 1 に示す．本ツールにおいては，以下の手順でバグ入りアプリケーションを生成する．

手順 1 アプリケーション構成要素のメソッドの構文解析

手順 2 操作列の生成

手順 3 操作列を構成するメソッドの一部をバグ入りメソッドに置換

手順 4 バグ入りアプリケーションの出力

3.4 入力

本ツールには，アプリケーションの構成要素を Java プログラムとして入力する．さらに，操作順に関する制約を適用する場合にはその制約について記した制約ファイルも入力する必要がある．

構成要素のメソッド定義は，前提として以下のものを考える．

- 引数は 1 つ又は 2 つ
- 第 1 引数は次節で説明するアプリケーション生成時に用いる．
- 第 2 引数がある場合は，フォームやプロンプトからの実行時のユーザ入力が与えられる．

以下，図書館の書籍管理に関するアプリケーションを例として提案手法を説明する．このアプリケーションの構成要素は `Library` クラス上に用意されており，このクラス内にはデータモデルと操作が定義されている．データモデルとは主にフィールドのことを指し，操作とはメソッドのことを指す．

データモデルは `Library` クラス内の以下の 3 つのクラスで定義されている．

- `LibData` クラス: 書籍に関するデータを保持する．
- `History` クラス: 書籍の貸出履歴に関するデータを保持する．
- `Member` クラス: 図書館の会員に関するデータを保持する．

操作は 20 個のメソッドで定義されている．それらの操作の仕様を表 1 に示す．

3.5 操作列の生成

3.5.1 生成方法

図 1 の手順 2 では，まず構成要素内の操作の中で入力 (第 1 引数) と出力 (戻り値) の型が同じ操作をまとめ，操作をグループ化する．ユーザは生成したいバグ入りアプリケーションの入力の型と出力の型を指定する．その指定に対して，本ツールはまずユーザ指定の入力の型と操作の引数の型が一致するグループからある操作を一つ選択する．その後，選択した操作の戻り値の型を引数の型に持つグループから同様に操作を選択する．この処理をユーザ指定の出力の型と戻り値の型が一致するまで繰り返す．このようにしてできる操作の列を全て生成する．

例として，操作が図 2 のようにグループ化されているとする．これに対して，生成されるアプリケーションの入力と出力の型をそれぞれ型 A，型 D と指定した場合に生成されうる操作列は全部で 27 個である．図 3 にはその中から，

操作列 1: 操作 a1 → 操作 b2 → 操作 c3
操作列 2: 操作 a2 → 操作 b1 → 操作 c1
操作列 3: 操作 a3 → 操作 b1 → 操作 c2

の 3 つの操作列が示されている．

また，生成される操作列のバリエーションを増やすため，条件分岐 (`if`) を含む操作列も生成可能とする．条件分岐でも型の整合性を失わないようにするため，条件分岐の直前の操作の戻り値は，条件式として用いる操作の引数と一致するようにする (戻り値は `boolean`)．さらに，前の操作の戻り値からユーザ指定の出力までの全ての操作列から 2 つ選び，`then` 部分と `else` 部分に割り当てる．例えば，表 1 の図書館の書籍管理アプリケーションの構成要素での操作列を生成する場合において，次の操作の引数の型

表 1: 図書館の書籍管理アプリケーションにおける構成要素メソッドの仕様

引数	戻り値	メソッド名
String	List<LibData>	search_book search_members_books
String	LibData	search_by_bookname
String	void	register_book register_member
List<LibData>, String	LibData	narrowdown_book_author narrowdown_book_genre
List<LibData>	List<LibData>	sort_by_borrowers
List<LibData>	void	borrow_books display_bookList
LibData	boolean	is_borrowed
LibData	void	borrow_book dispose_book
LibData	List<LibData>	sameAuthors_book
LibData	String	get_book_author
LibData	List<Member>	get_borrowers
List<Member>	List<Member>	sort_member_Jalphaorder
List<Member>	Member	get_top_member
List<Member>	void	display_memberList
Member	String	get_members_name

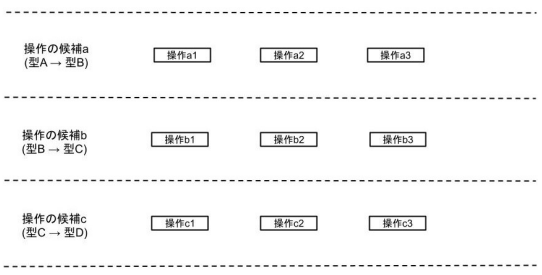

図 2: グループの例

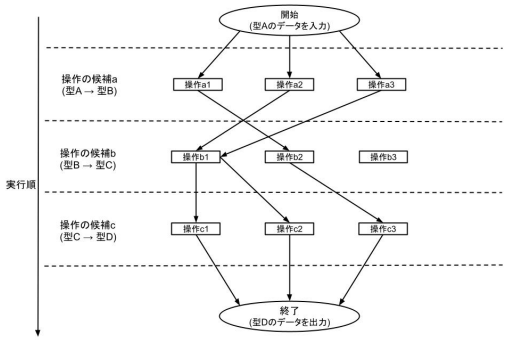

図 3: 3 つの操作列の例

が LibData の場合，条件式として is_borrowed が選択され，then 部分に borrow_book，else 部分に dispose_book が選択される新たな操作列が生成される．生成される操作列に条件分岐を含めるかどうかは，実行時にオプションとして指定できる．

このように，型の整合性だけから操作を接続すると，現実的には不自然な処理を行う操作列も生成されてしまう．それを回避するために，以下に述べるような操作列に対する 2 種類の制約を指定可能とした．

3.5.2 型の性質による制約

図 1 の手順 2 において，単純な型の一致のみで操作列を生成すると，意味のない操作列が作成される可能性がある．例えば，書籍管理アプリケーションの search_book メソッドの引数はある単語を示す String 型だが，get_book_author メソッドの戻り値は本の著者を示す String 型である．この時，操作中の

get_book_author メソッドの次の操作に search_book メソッドを選択すると，search_book メソッドの引数として本の著者の String 型が与えられることになり，この操作列は意味のないアプリケーションを生成する操作列となってしまう．

このような問題を解決するために，メソッドの引数や戻り値の型に性質を付与し，操作を接続する際に，型だけでなくその性質の一致も条件に加えた．

3.5.3 操作順に関する制約

型の性質による制約以外に，操作順に関する制約も指定可能とする．この制約は, 構成要素の各操作に対して，型とは別の性質を自由に付与することができ，次の操作で接続されないという制約と，それ以降の操作で接続されないという二つの制約を指定できる．

この制約は，型の性質による制約によって防ぐことができない，現実的に存在しえないような不自然な処理を行うアプリケーションの生成を，独自に防止するために必要となる．例えば，書籍管理アプリケーションの search_book メソッドや search_by_bookname メソッドなどの本の検索を行う操作を図書館の利用者側が行う操作，本の削除を行う dispose_book メソッドを図書館の司書側が行う操作とみなしてこれらの操作を分離したい場合，型の性質による制約ではこれらの操作が同時に現れる操作列の生成を防止できない．このような場合に操作順に関する制約が必要となる．

3.6 バグ入りメソッドへの置換

図 1 の手順 3 において，生成された各操作列に対して，操作列中に含まれる構成要素の一部をバグ入りメソッドに置換する．

3.7 出力

バグ入りメソッドを含む操作列全てに対して，Java プログラムとしてファイルに出力し，バグ入りアプリケーションとする．

4 実装

4.1 Java パーサの生成

本論文では，パーサジェネレータとして，ANTLR (ANother Tool for Language Recognition) †1 を用いた．また，ANTLR でパーサを生成する為の Java の文法ファイルとして，GitHub 上で公開されている Java.g4 †2 を用いた．

4.2 構成要素の構文解析

図 1 の手順 1 では，生成されたパーサを用いて構文解析を行い具象構文木 (CST) を生成する．生成された CST の一部を図 4 に示す．これは，プログラム中のフィールドへの値の代入”author = a;” という部分の CST である．

4.3 操作の選択

図 1 の手順 2 において，ユーザが指定したアプリケーションの仕様 (入力と出力の型) に合致する全ての操作列を生成するため，頂点を型，有向辺をメソッドとするような有向グラフをまず構築する．ここで，各辺の名前はメソッド名，始点は引数の型，終点は戻り値の型とする．

例えば，search_book メソッドでは，名前が search_book，始点が String，終点が List<LibData> の有向グラフとなる．

図書館の書籍管理に関する構成要素をもとに生成された有向グラフを図 5 に示す．この有向グラフにおいて，ユーザが生成したいバグ入りアプリケーションの入力の型を始点，出力の型を終点として深さ優先探索を行って，辺の重複がない全ての経路 (操作列) を探索する．各操作列はメソッド呼び出しのつながりという形で，main メソッドから呼ばれる新しいメソッド invoke_method として実装する．

ただし，選択された操作に第 2 引数が存在した場合，その操作の前に InputfromUser_XXXX メソッド (XXXX は型名) を挿入する．このメソッドはプロンプトから指定された型のユーザ入力を受け取ってそれを

†1 https://www.antlr.org/

†2 https://github.com/antlr/grammars-v4

図 4: 作成される CST の一部

そのまま返すメソッドである．

例えば操作列が，

1. `search_book`
2. `narrowdown_book_genre`
3. `borrow_book`

の場合，`invoke_method` を以下のように生成する．

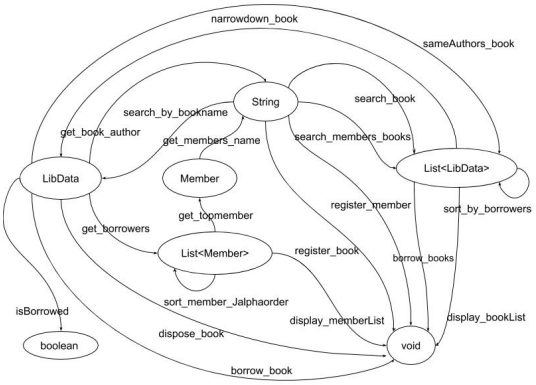

図 5: 生成される有向グラフ

```
public static void invoke_method(String arg) {
  List<Lib_Data> v1 = search_book(arg);
  String input1 = InputfromUser_String();
  Lib_Data v2 =
     narrowdown_book_genre(v1, input1);
  borrow_book(v2);
}
```

また，条件分岐を含む `invoke_method` の例を以下に示す．

```
public static void invoke_method(String arg) {
  List<Lib_Data> v1 =
     search_members_books(arg);
  String input1 = InputfromUser_String();
  Lib_Data v2 =
     narrowdown_book_genre(v1, input1);
  if(is_borrowed(v2)) {
    borrow_book(v2);
  } else {
    List<Lib_Data> v21 = sameAuthors_book(v2);
    borrow_books(v21);
  }
}
```

4.4 操作列の制約

図 1 の手順 2 で用いる型の性質による制約と操作順に関する制約は，Java のアノテーションを付与することで適用させる．

4.4.1 型の性質による制約

型の性質は，構成要素のメソッド定義に，引数の性質を表す`@Atype({`v_1`,`v_2`,`$\cdots$`})` と戻り値の性質を表す`@Rtype({`v_1`,`v_2`,`$\cdots$`})` の 2 つのアノテーションを付与する．ここで，v_1，v_2，$\cdots$ は，性質を表す文字

列とする．

例えば，register_book メソッドの引数の文字列が書名を表すという性質は，@Atype({"bookname"}) というアノテーションで与える．

4.4.2 操作列に関する制約

操作列に関する制約は，構成要素内のメソッドに @Atype，@Rtype 以外の新しいアノテーションを付与し，そのアノテーション間の接続関係を記述したテキストファイルを与えることで適用する．

例えば，書籍検索操作 (search_book，search_member_books，search_by_bookname) に@Search というアノテーションが，書籍を削除する操作 (dispose_book) に@Dispose というアノテーションが付与されているものとする．このとき，検索操作の後には削除操作を一切行わないという制約は @Search {@Dispose}と，検索操作の直後には削除操作を行わないという制約は@Search <@Dispose>と記述する．

はじめに対象のアノテーションを記述し，その直後に対象のアノテーションが付与されたメソッドが含まれる操作列との接続関係を記述する．{} 内には操作列にそれ以降接続したくないメソッドに付与されているアノテーション，<>内には次のメソッドに接続したくないメソッドに付与されているアノテーションを記述する．

4.5 バグ入りメソッドの挿入

図 1 の手順 3 において，各操作列に対して，操作列中に含まれる構成要素の一部をバグ入りメソッドに置換する．これは，操作列を表す CST を破壊的に書き換えることにより行う．

5 評価実験

5.1 評価観点

型の性質による制約と操作順に関する制約によって，現実的に存在しえないような不自然な処理を行うアプリケーションの生成を防止できているかを確認する．

5.2 評価方法

図書館の書籍管理アプリケーションについて，表 1 の構成要素を用いて，条件分岐を含めない時と含める時の両者に対し，制約を適用しなかった場合，型の性質による制約を適用した場合，操作順に関する制約を適用した場合，両方の制約を適用した場合のそれぞれに対して，ツールを実行して出力されるアプリケーションの数を評価した．このとき，実行する際の設定は以下のようにした．

- 生成されるアプリケーションの入力の型と出力の型を以下の 3 つの場合に設定して実行した．
  - String → void
  - LibData → void
  - String → String
- 操作の型の性質は表 2 のように付与した．各性質は表 3 のような意味を持つ．
- 操作順に関する制約は，書籍の検索を図書館ユーザ側の操作，書籍の削除を司書側の操作とみなし，これらの操作を分離するものとした．具体的には，search_book，search_members_books，search_by_bookname を検索操作，dispose_book を削除操作とした．

5.3 結果と考察

アプリケーションの入出力の型を String → void，LibData → void，String → String とした場合，ツールを実行した際に生成されるアプリケーション数と，その時の実行時間を表 4 に示す．しかし，条件分岐を適用した場合において，制約なし及び操作順に関する制約を適用した場合は，生成されるアプリケーションが非常に多く莫大な実行時間がかかってしまい，測定を行うことができなかった．

これらの表より，入出力の型の設定によって生成されるアプリケーション数に大きく違いがあることもわかる．

また，入力の型が String，出力の型が void の際，型の性質による制約によって削除されるアプリケーションの例を以下に示す．

```
public static void invoke_method(String arg) {
```

表 2: 各操作の性質

メソッド名	引数の性質	戻り値の性質
search_book	word	unsort
search_members_books	membername	unsort
search_by_bookname	bookname	unsort
register_book	bookname	-
register_member	personname	-
narrowdown_book_author	unsort	-
narrowdown_book_genre	unsort	-
sort_by_borrowers	unsort	sort
borrow_books	unsort	-
sameAuthors_book	-	unsort
get_book_author	-	author
get_borrowers	-	unsort
sort_member_Jalphaorder	unsort	sort
get_top_member	sort	-
get_members_name	-	membername

表 3: 各性質の意味

型	性質	詳細
String	word	文字や単語
	membername	会員名
	bookname	書名
	author	著者名
	personname	人名
List<LibData>	sort	ソート有
	unsort	ソート無
List<Member>	sort	ソート有
	unsort	ソート無

```
  ⋮
  String v5 = get_members_name(v4);
  List<LibData> v6 = search_book(v5);
  ⋮
}
```

この例において，get_members_name メソッドは図書館の会員名を表す String 型の値を返すが，search_book メソッドは本に含まれるある単語を表す String 型の値を受け取るため，ここでは変数 v5 が制約違反となり，型の性質による制約によって削除された．

また，操作順に関する制約によって削除されるアプリケーションの例を以下に示す．

```
public static void invoke_method(String arg) {
  List<LibData> v1 = search_book(arg);
  LibData v2 = narrowdown_book(v1);
  dispose_book(v2);
}
```

この例において，search_book メソッドは本の検索操作に値し，dispose_book メソッドは本の削除操作に値するため，このアプリケーションは制約違反となり，操作順に関する制約によって削除された．

これらの結果から，制約を適用しない場合は多くの意味のないアプリケーションが生成されてしまうが，2 つの制約によってこれらのアプリケーションの生成を防げていることが確認できた．特に今回用いた書籍管理アプリケーションの構成要素では，Java 標準の型である String 型を引数または戻り値に持つ操作が多く存在したため，型の性質による制約の効果が大きかった．

6 おわりに

本論文では，テストスイートのバグ検出能力の測定に用いるためのバグ入りアプリケーションを生成するツールを実装した．このツールでは, 構文解析や有向グラフの作成，引数及び戻り値の型や性質による操作の選択，操作順に関する制約の記述などを行い，様々なバリエーションを持ち，かつ現実的に自然なバグ入りアプリケーションを生成可能である．

表 4: 生成されるアプリケーション数と実行時間

(a) 入力:`String`，出力:`void`

	通常		条件分岐	
	アプリ数	実行時間 (ms)	アプリ数	実行時間 (ms)
制約なし	3,384	33,076	-	-
操作順に関する制約	2,987	31,407	-	-
型の性質による制約	116	8,496	3,366	36,506
型の性質による制約 + 操作順に関する制約	97	8,239	2,201	27,663

(b) 入力:`LibData`，出力:`void`

	通常		条件分岐	
	アプリ数	実行時間 (ms)	アプリ数	実行時間 (ms)
制約なし	2,586	15,115	-	-
操作順に関する制約	2,063	13,364	-	-
型の性質による制約	42	7,870	2,000	14,322
型の性質による制約 + 操作順に関する制約	36	7,405	1,448	11,490

(c) 入力:`String`，出力:`String`

	通常		条件分岐	
	アプリ数	実行時間 (ms)	アプリ数	実行時間 (ms)
制約なし	599	9,322	-	-
操作順に関する制約	599	9,009	-	-
型の性質による制約	30	7,261	250	8,318
型の性質による制約 + 操作順に関する制約	30	7,856	250	8,211

生成されるアプリのバリエーションをさらに増加させるため，繰り返しを含むアプリを生成できるようにすることが今後の課題である．また，生成されたアプリの類似度のような値を算出し，似ているアプリの生成を防ぐことも今後の課題である．これは，テストの際にかかるコストの削減だけでなく，ソフトウェアテストの教育の場でも，ある問題に対する類似問題を作成する際などに活用できると考えている．

参 考 文 献

[1] Afzal, W., Ghazi, A. N., Itkonen, J., Torkar, R., Andrews, A., and Bhatti, K.: An Experiment on the Effectiveness and Efficiency of Exploratory Testing, *Empirical Software Engineering*, Vol. 20, 2015, pp. 844–878.

[2] Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., and McMinn, P.: An orchestrated survey of methodologies for automated software test case generation, *Journal of Systems and Software*, Vol. 86, No. 8(2013), pp. 1978–2001.

[3] Gyimesi, P., Vancsics, B., Stocco, A., Mazinanian, D., Beszédes, Á., Ferenc, R., and Mesbah, A.: BugsJS: a Benchmark of JavaScript Bugs, *ICST* , 2019, pp. 90–101.

[4] Jia, Y. and Harman, M.: An Analysis and Survey of the Development of Mutation Testing, *TSE* , Vol. 37, No. 5(2011), pp. 649–678.

[5] Just, R., Jalali, D., and Ernst, M. D.: Defects4J: A Database of Existing Faults to Enable Controlled

Studies for Java Programs, *ISSTA* , 2014, pp. 437–440.

[6] Madeyski, L., Orzeszyna, W., Torkar, R. and Józala, M.: Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation, *TSE*, Vol. 40, No. 1(2014), pp. 23–42.

[7] Ma, Y. S., Offutt, J., and Kwon Y. R.: Mujava:A Mutation System for Java, *ICSE*, 2006, pp. 827–830.

[8] Orso, A. and Rothermel, G.: Software Testing: A Research Travelogue (2000-2014), *FOSE* , 2014, pp. 117–132.

[9] Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y. L., and Harman, M.: Mutation Testing Advances: An Analysis and Survey, *Advances in Computers* , Vol. 112, 2019, pp. 649–678.

[10] Wong, W. E., Gao, R., Li, Y., Abreu, R., and Wotawa, F.: A Survey on Software Fault Localization, *TSE* , Vol. 42, No. 8(2016), pp. 707–740.