CB-003

Java バイトコード逆実行シミュレータ
Reverse execution simulator of Java bytecode

門馬 悠太† 平中 幸雄† 武田 利浩†
Yuta Monma Yukio Hiranaka Toshihiro Taketa

1. 背景

プログラム開発ではバグは頻繁に発生する。したがって、プログラムの品質を保証するためにはテストが必要である。プログラムのテスト手法として一般的なのは、テスト入力に対応する出力が、意図した通りになるか確かめる順方向のテストである。順方向テストは、判明している入力条件に対する出力値が正しいか検査するのに適している。しかしプログラムの設計や構築でミスが生じ、テストしていない入力条件での意図しない出力が発生する場合がある。順方向テストでこういったバグを検出するのは難しい。

そこで、本研究では逆方向テストに着目する。逆方向テストは、プログラムの出力値に対応する入力条件を特定するテスト手法である[1,2]。逆方向テストで出力値を検査することで、出力に対する入力条件が設計通りであるか検査することが可能である。

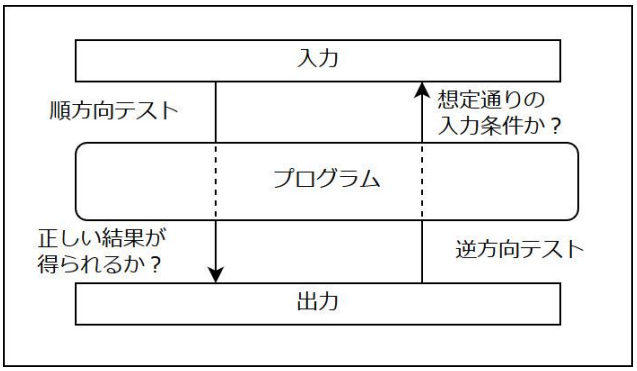

図 1 順方向テスト・逆方向テスト

2. 目的

順方向テストで発見することが難しいプログラムのバグなどを、逆方向テストを用いて発見することができるシミュレータを開発することが本研究の目的である。

3. 順方向テスト・逆方向テスト

順方向テストでは、入力パターンの数が多いほど必要なテスト数が増加してしまう傾向があるため、入力数の数が多いとテストの負担が増加することになる。

例えば、入力値の範囲が 0~99 までの 100 通りの整数値に、出力値が 0,1 の 2 通りに設定されているプログラムを解析する場合、入力値のテストケースは 100 通りである。一方で出力値をテストする場合、2 通りである。したがって出力値から逆算する、すなわち逆方向でテストしたほうが効率よく解析を行うことができる。

また、設計上の入力条件は 0~99 までであるが、ミスによりこれ以外の入力パターンが作成されてしまう可能性がある。こういった入力パターンは開発者にとって想定外の入力条件であり、対策が行われない場合が多い。

こういった場合も、逆方向テストが有効である。出力値 0、および 1 に対する入力条件を特定し、結果が設定した入力条件と一致すれば、プログラムは正しく動作する。一方、設計上の入力条件を満たしていない、あるいは意図していない入力条件が存在するという結果が得られれば、プログラムにバグが存在する可能性を示している。逆方向テストではバグの原因となる入力条件を特定できるため、バグ修正の効率向上が期待される。

4. Java バイトコード

本研究では Java のバイトコードを対象とし、バイトコードの逆実行をシミュレートするプログラムを開発する。バイトコードは Java 仮想マシン上で実行する命令である。バイトコードは比較的単純な命令で構成されているためシミュレータ開発に好条件である[3]。

Java プログラムは Java 仮想マシンと呼ばれるスタック型の仮想マシン上で動作する。Java 仮想マシンはプログラム実行時、メモリ上に様々なデータ領域を定義している。本研究では、バイトコード実行時に必要な値を取得したり、実行結果を戻すためのオペランド・スタックと呼ばれる LIFO スタックと、ローカル変数と呼ばれるプリミティブ型、参照型のいずれかの値を保持する配列をシミュレータ上で再現することで、バイトコード実行を実現している。

5. バイトコードの順実行動作

Java バイトコードは Java コードをコンパイルして得られる Java 仮想マシンの命令セットである。バイトコード命令には、さまざまな操作が定義されている[4]。表 1 はその整数処理用の命令の一例である。

表 1 オペコードの名称と動作の一例

コード	動作
iconst	int 定数をプッシュする
iload	ローカル変数から int をロードする
istore	ローカル変数に int をストアする
iadd	int の加算を行う

図 2 は演算“2 + 3”を実行するプログラムのバイトコードである。左から先頭命令からのオフセット、バイトコー

† 山形大学, Yamagata University

ド、オペランド・スタック、ローカル変数の順に並んでおり、オペランド・スタックとローカル変数の内容はバイトコード命令を実行した結果を表示している。

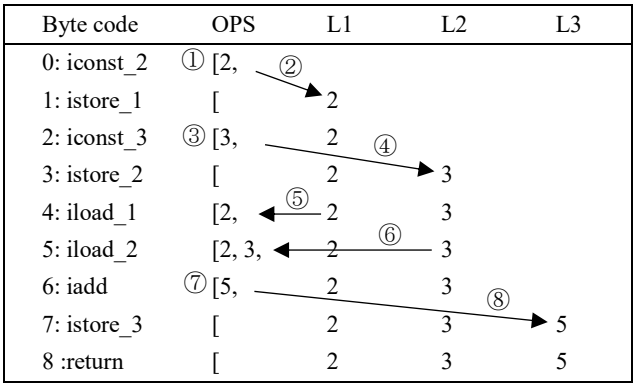

図 2　バイトコードの順方向実行

（OPS ： オペランド・スタック，Ln ： ローカル変数 n)

まず、iconst 命令で int 値 2 がオペランド・スタックにプッシュされる（①)。次に istore 命令でスタックの先頭の値をローカル変数 1 にストアする（②)。同様に、int 値 3 をプッシュし（③)、その値をローカル変数 2 にストアする（④)。次に、それぞれのローカル変数から値を順番にオペランド・スタックにロードする。iadd 命令で、オペランド・スタックの先頭にある int 型 2 値(=2,3)をポップし(⑤⑥)、加算処理を行い、結果(=5)をスタックにプッシュする (⑦)。実行結果をローカル変数 3 にストアし（⑧)、最後に return 命令でオペランド・スタックにあるすべて値を破棄して処理を終了する。これがバイトコードの通常の動作である。

6. バイトコードの逆実行動作

本研究では出力から逆演算するためバイトコードの逆実行が必要になるが、その実現にシンボリック実行という手法を用いる。シンボリックな手法で逆実行を行う場合、命令の実行前後を区別して表現する必要がある。

命令逆実行前の値は未確定な状態で始めるしかない。命令を逆実行すると、未確定にした値またはその条件が決まってくる。たとえば、未確定な状態を V1, V2 といった仮の変数で定義し、逆実行を進めていく上で仮の変数の値が判明した場合、仮の変数をその値に置き換える。こういった処理を繰り返すことでバイトコードの逆実行を実現している。

図 3 はバイトコードの逆実行動作例である。逆実行では return 命令の逆実行後のオペランド･スタックの状態が不明であるため、スタックの先頭に着目して逆実行を行う。逆実行開始時、ローカル変数 1 の値が不明である為、仮の変数として L1(0)と置く。括弧内の数字はローカル変数の値を区別するための番号であり、命令によってローカル変数の値が書き換えられる毎に更新する。次に istore_1 の処理について、istore_1 の逆実行後スタックにはローカル変数 1 にストアする値が置かれていなければならない。したがって逆実行前のローカル変数 1 に格納されている L1(0)を、逆実行後状態のスタックの先頭に置く（①)。また、istore_1 の逆実行後にローカル変数 1 にほかの値が格納されていた可能性がある。したがって、逆実行前状態のローカル変数にある L1(0)の括弧内の値を 1 増加し、L1(1)と表す（②)。

そして iconst_2 の処理について、この命令は int 値の 2 をプッシュする命令なので、逆実行前のスタックに 2 があるはずである。これにより L1(0) = 2 であることが判明する（③)。また、逆実行後スタックではその値はなくなっている。以上で一連の逆実行は終了する。

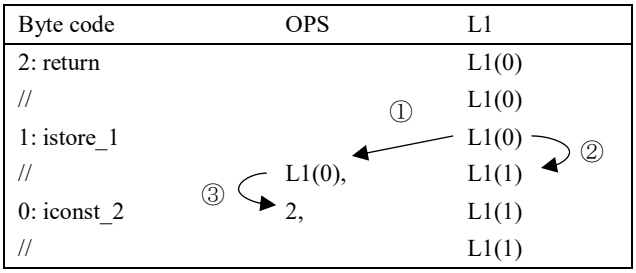

図 3　バイトコードの逆方向実行

（命令の逆実行後状態を”//”と表記する）

7. 逆実行のループ処理

逆実行でループに入ることは、順実行においてループを抜けることを意味している。すなわち、逆実行でループに入った時に得られる条件が、ループを抜ける条件となる。この条件のもとでループ内の逆実行を行い、条件を満たしたときループを抜けることができる。

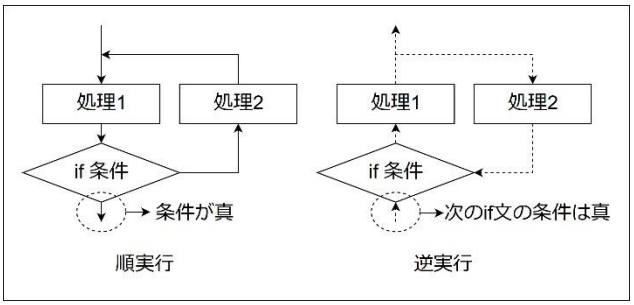

図 4　ループの判定基準

例えば、次の図 4 で示すサンプルプログラム 1 の逆実行について考える。

```
public class test {
    public static void main(String args[]) {
        int x = Integer.parseInt(args[0]);
        for (int j = 0; j < 2 j++) {
            x++;
        }
        System.out.println(x);
    }
}
```

図 5　サンプルプログラム 1

このプログラムを構成するバイトコードの逆実行の一部を図 6 に示す。ループを抜けるための分岐の条件は if_icmpge 命令から得られる。if_icmpge 命令はオペランド・

スタックが、[…, value1, value2 のとき、value1 ≧ value2 であれば指定したオフセットにジャンプする命令である。逆実行でこの命令に到達したということは、if_icmpge 命令の逆実行後スタックの先頭にある 2 値は value1 ≧ value2 であることを意味する。

Byte code	OPS	L1	L2
11: if_icmpge 23		L1(0)	L2(0)
//	V2, V3,	L1(	L2(0)
10: iconst_2	V2, 2	L1(	L2(0)
//	V2,	L1(	L2(0)
9: iload_2	L2(0),	L	L2(0)
//		L	L2(0)
8: istore_2		L1(0)	L2(0)
//	L2(0)	L1(0)	L2(1)
7: iconst_0	0	L	L2(1)

V2≧V3

V2≧2

L2(0)≧2

L2(0)= 0

図 6　分岐を抜ける条件の取得

if_icmpge23 の逆実行後スタックから、条件 V2≧V3 が得られる。次に iconst_2 の逆実行前スタックから V3=2 が得られ、条件が V2≧2 となる。また、iload_2 の逆実行前スタックから V2=L2(0)が得られ、最終的な条件は L2(0)≧2 となる。したがってこの条件を満たすときループを抜けることができる。逆実行では正しい実行パターンに到達するまで分岐を右に行くことで、ループ回数を増加させる。適合しない実行パターンであった場合、シミュレーションを停止し分岐時まで状態を復元し、もう一方の分岐を逆実行する。

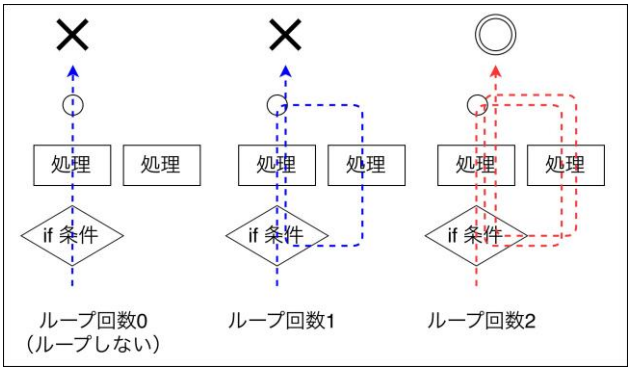

図 7　ループの繰り返し

まず、ループ回数 0 の実行パターンを考える。このパターンでは図 6 より L2(0)＝0 が得られるが、分岐を抜ける条件である L2(0)≧2 を満たしていない。そこで分岐するループを 1 回にして再度逆実行を行う。図 8 から、L2(0)＝L2(1)+1, L2(1)=0 という 2 つの条件が得られる。条件から L2(0)=1 を得るが、これも条件を満たさない。さらに、ループ回数を 2 回にして再度逆実行を行う。すると、L2(0)=L2(1)+1, L2(1)=L2(2)+1, L2(2)=0 という 3 つの条件が得られる。条件を整理すると、L2(0)=2 を得る。これは初めに設定したループを抜ける条件 L2(0)≧2 を満たす。したがって、この実行パターンが正しい実行パターンであることが判明する。このようにループの条件が一致しない場合、条件を満たすまでループ回数を増やしながら逆実行を行う。

Byte code	OPS	L1	L2
17: iinc　2,1		L1(0)	L2(0)
//		L1(0)	L2(1)
・・・			
8: istore_2		L1(1)	L2(1)
//	L2(1),		L2(2)
7: iconst_0	0,		L2(2)
//		L1(1)	L2(2)

L2(0)=L2(1)+1

L2(1) = 0

図 8　ループ回数 1 の実行パターン

（iinc x,y ：ローカル変数 x の値を y ｲﾝｸﾘﾒﾝﾄする）

しかし、この手法では正しい実行パターンを得られない場合がある。例えば、ループ回数 3 で逆実行を行うと、L2(0)=3 が得られる。これは、L2(0)≧2 を満たすため正しい実行パターンであると判断されてしまう。また、条件がユーザの入力によって決定する場合（例:value1≧(入力値)）、条件式は求まるが判定できない。またプログラムに無限ループが含まれている場合、逆実行が無限ループに陥る可能性がある。

8. 配列への対応

バイトコードには配列の生成、配列長の取得、配列へのロード・ストアといった配列を操作する命令が存在する。配列に対するロード・ストア命令はローカル変数に対するロード・ストア命令と異なり、バイトコードに操作の対象となるローカル変数番号が含まれておらず、オペランド・スタックに積まれた値によって配列要素が決定する。逆実行はシンボリック実行であるため、操作の対象が不明である。そこで逆実行では、配列のロード・ストア命令の実行時に実際にロードやストアは行わず、命令の実行に必要な値がすべて判明してからロードやストアを実行する。

Byte code	OPS	L1	L2
13: istore_2		L1(0)	L2(0)
//	L2(0)	L1(0)	L2(1)
12: iaload	L2(0),	L1(0)	L2(1)
//	R1, V1,	L1(0)	L2(1)
11: iconst_0	R1, 0,	L1(0)	L2(1)
//	R1,	L1(0)	L2(1)

図 9　iaload 命令の処理

例えば図 9 では、int 型の配列からロードする iaload 命令の逆実行後スタックが、[…, R1, V1, である場合、R1 にはストアする配列の番号、V1 には配列の中のインデックスがそれぞれプッシュされている。そして L2(0)は配列からロードした値が格納される。また iconst_0 の逆実行前から、V1=0 が判明し、ここで R1 の配列の 0 番目から値をロードすることが判明する。この時点で初めてロードを実行することができる。

また図 10 では、int 型の配列にストアする iastore 命令の逆実行後のスタックが[…,R1,V2,V3, である場合、R1 にはロードする配列への参照値、V2 には配列の番号、V3 にはストアする値がプッシュされている。iconst_5 の逆実行前のスタックから V3=5 が判明する。また、iconst_0 の逆実行前のスタックから V2=0 であることが判明する。ここで R1 の配列の 0 番目に int 値の 5 をストアすることが判明する。

Byte code	OPS	L1	L2
9: iastore		L1(0)	L2(1)
//	R1, V2, V3,	L1(0)	L2(1)
8: iconst_5	R1, V2, 5,	L1(0)	L2(1)
//	R1, V2,	L1(0)	L2(1)
7: iconst_0	R1, 0,	L1(0)	L2(1)
//		L1(0)	L2(1)

図 10　iastore 命令の処理

9. 実行例

図 11 は 2 入力に対し、それぞれ非負の場合 1 を、負の場合 0 を出力するプログラムである。このプログラムをシミュレータで解析する。今回開発したシミュレータは出力値を与えず、全ての出力に対する入力条件を導出することも可能である。そこでコマンドライン引数を入力、println による画面への表示を出力として、全ての出力を解析する。

```
class test {
    public static void main(String[] args) {
        int a[] = new int[2];
        int x;
        for (int i = 0; i < 2; i++) {
            x = Integer.parseInt(args[i]);
            if (x > 0) {
                System.out.println(1);
            } else if (x < 0){
                System.out.println(0);
            }
        }
    }
}
```

図 11　サンプルプログラム 2

```
パターン 1
出力値　 V1 = 0
入力条件 args[0] < 0
・・・
パターン 2
出力値　 V1 = 1
入力条件 args[0] > 0
・・・
パターン 3
出力値　 V1 = 0
入力条件 args[1] < 0
・・・
パターン 4
出力値　 V1 = 1
入力条件 args[1] > 0
・・・
```

図 12　解析結果（抜粋）

実際の実行結果には、バイトコード逆実行の過程も出力されるが、今回は判明した入力条件のみ示している。

図 12 の解析結果から、例えば出力値が 1 である場合の正しい入力条件はパターン 2 とパターン 4 であり、その入力条件はそれぞれ args[0]>0 および args[1]>0 であることが分かる。さらにこの結果から、入力条件 args[0]=0 と args[1]=0 が含まれていないというバグがあることが判明した。また、今回はサンプルコードに配列やループを含めても、正しく解析を行えていることが確認できた。

10. 結論

本研究では Java のバイトコードに着目し、バイトコードの逆実行によって入力条件を導出するシミュレータを開発した。その結果、逆実行で出力から入力条件を導出することができた。また、簡単なプログラムのミスを発見することができ、プログラムの逆実行による解析はデバッグのための有効な手法であることを示した。

11. 今後の課題

今回開発したシミュレータはいくつか課題がある。現在のところ、プログラムの main メソッドの解析にしか対応していない。本シミュレータはメソッドの呼び出しに対応しておらず、invoke 命令や return 命令などが実行できない（println や parseInt は仮の処理を行って対応している）。したがって、こういったバイトコードの逆実行処理を実装するには他のメソッドの逆実行が行えるようにする必要がある。

次に、ループの問題である。ループ回数を増やしながら正しい実行パターンを検査するという手法は、正しい実行パターンが得られない場合（無限ループプログラムなど）、無限に逆実行を繰り返してしまう。また、ループや条件分岐に入力値が関係している場合も、条件が決まっていない状態となるためループを抜けることができない。こういった問題を解決するためには無限ループの判断、ループの回数制限などの対策を行う必要がある。

本研究でプログラムの逆実行による解析は有効であることが判明した。今後、逆実行に適したバグを調査することや、様々なバグを対象に検出可能であることを確認していく必要がある。

謝辞

本研究は JSPS 科研費 JP15K11989 の助成を受けたものです。

参考文献

[1] Yukio Hiranaka, Tetsuya Inafune, Shinichi Miura, Toshihiro Taketa(2017), “Backward range simulation of Java bytecodes and reduction of its processing time”, Yamagata University

[2] Tetsuya Inafune, Shinichi Miura, Toshihiro Taketa, Yukio Hiranaka, “Symbolic backward simulation of Java bytecode program”, Yamagata University

[3] Carl E. Bredlau(2008), “JVMVIEWER: an interactive bytecode interpreter for Java”, Montclair State University, Journal of Computing Sciences in Colleges, Volume 23 Issue 3, January 2008 Pages 44-49

[4] Tim Lindholm, Frank Yelin, 村上 雅章, “Java 仮想マシン仕様 第 2 版”, （株）ピアソン・エデュケーション