平成２８年度卒業論文

マルチスレッドプログラミングにおける
デバッグ研究のための
バグ入りプログラムの作成

情報・通信工学科 コンピュータサイエンスコース

１３１１１７１ 藤本 明優

指導教員　寺田 実　准教授

提出日 ２０１７年 １月３１日

概要

目的

マルチスレッドプログラミング特有のバグである並行性バグは再現性の問題によりデバッグが困難である. その解決のために様々なデバッグ手法が提案されてきた. こうしたデバッグツールの開発におけるテストやデバッグ手法を学ぶためのサンプルとしてマルチスレッドのバグ入りプログラムを作成することが本研究の目的である.

方法

手作業によるバグ入りプログラムの作成とバグ入りプログラムの自動生成の二つを試みる.

結論

作成したバグ入りプログラムはデバッグ手法およびツールの評価に役立てることができた.

目次

第 1 章　序論

1.1　背景

マルチスレッドプログラミングを行う場合, シングルスレッドプログラミングでは発生しなかった特有のバグが発生する. これを並行性バグという. 並行性バグは複数のスレッドにより並行処理を行う場合に, スレッド間の相互作用やメモリモデルの仕様によって発生するバグである.

ここで, メモリモデルとはキャッシュやレジスタなど実際に扱われるメモリを抽象化したモデルである.

並行性バグには再現性の問題がある. 並行性バグにおける再現性とはプログラムを実行した際のバグの顕在化する確率の高さである.

並行性バグはスケジューリングに由来し, 同じ動作をさせてもスケジューリングが異なる場合必ず発生するとは限らない. 特に再現性の低いバグは発見されないままリリースされることが多く, 発見できてもその再現性の低さのためにシングルスレッドプログラムのバグに比べ発見と原因追及が困難である. そのため今日まで様々なデバッグ手法が提案されてきた [9][6].

1.2　目的

本研究の目的はデバッグツールの開発におけるテストやデバッグ手法を学ぶためのサンプルとしてマルチスレッドのバグ入りプログラムを作成することである.

またバグ入りプログラムを自動生成することでサンプルを増やし, デバッグ手法やデバッグツールのベンチマークテストに利用できないか検討する.

1.3　本論文の構成

論文の構成を簡単に説明する．本章では， 序論として研究の背景， 目的について述べた．第 2 章では，マルチスレッドのバグとデバッグ手法について述べる．第 3 章では，関連研究について述べる．第 4 章では，作成したバグ入りプログラムについて述べる．第 5 章では，作成したバグ入りプログラムおよびその自動生成の評価について述べる．第 6 章では，まとめと今後の課題について述べる．

第 2 章　マルチスレッドのバグとデバッグ手法

2.1　バグの種類

マルチスレッドには以下の種類のバグが存在する.

- メモリの可視性由来のバグ
- デッドロック
- 競合状態
- データ競合
- 飢餓状態
- ライブロック

以下, 各バグとそのバグによって引き起こされるエラーについて説明する (参考 [5]).

2.1.1　メモリの可視性

メモリの可視性由来のバグはメモリモデルに由来するバグである.

例えば Java メモリモデル (JMM) ではマルチスレッドプログラミングの場合, マルチコアで動作させると変数の同期化されていない更新が行われてもその CPU のローカルメモリの値のみが更新され, 別の CPU のローカルメモリの値は更新されないなどの仕様がある [5]. これは高速化のための仕様である.

メモリモデルの仕様によってはロックで区切られていない区間をシングルスレッドにとって問題ないように処理の順番を変えることが許されるのでプログラマの想定と異なる動作をしてしまうバグやマルチコアで動作させた場合に異なる CPU にスレッドが割り当てられ, ロックで同期していない場合に値がスレッドごとに異なってしまうバグなどが発生する (リスト 2.1). そのため環境によって発生するか否かが変化する.

```
1  // Memory mは1000フレームの停止の後aを更新するが同期をしていないため
2  // 環境によってmainメソッドを動かすスレッドを持つCPUのローカルメモリの値が更新されない
3  // その結果, 永遠に更新前の値, つまり0が読み込まれる場合がある.
4  // このプログラムそのバグが発生すると
5  // aの値である1が表示された後もmainメソッドのwhile文が無限ループしプログラムが終了しない
6
7  // これを避けるためには同じキーに対するsynchronizedブロックで
8  // M1とM2をロックするなどで同期をとるか, 変数aにvolatile修飾子をつける
9  public class Memory extends Thread {
10     static int a = 0;
11
12     public void run()
```

```
13      {
14          try {
15              sleep(1000);
16          } catch (InterruptedException e) {
17              e.printStackTrace();
18          }
19          a = 1;
20          System.out.println(a);
21      }
22
23      public static void main(String[] args) throws InterruptedException
24      {
25          Memory m = new Memory();
26          m.start();
27
28          while(Memory.a == 0) {}
29      }
30  }
```

リスト 2.1: メモリモデルに由来するバグの例 (Java)

2.1.2　デッドロック

スレッド X がロック A を取得している間にロック B を取得し, スレッド Y がロック B を取得している間にロック A を取得しようとするとき, スレッド X がロック A を, スレッド Y がロック B を同時に取得した場合に互い違いのロックがお互いの動作を永遠に停止させてしまうバグをデッドロックと呼ぶ (リスト 2.2). デッドロックは発生するとフリーズなどの危険なエラーを引き起こす.

```
1   // 別のスレッドがそれぞれメソッド M1 と M2 を同時に呼ぶとデッドロックとなる
2   public void M1() {
3       synchronized(A) {
4           synchronized(B) {
5               A.method1();
6           }
7       }
8   }
9
10  public void M2() {
11      synchronized(B) {
12          synchronized(A) {
13              B.method2();
14          }
15      }
16  }
```

リスト 2.2: デッドロックを起こすプログラムの例の断片 (Java)

2.1.3　競合状態

特定の処理が同時に起こるなど複数のスレッドが限られたタイミングで実行した場合に正しい計算結果が得られないエラーを引き起こすバグを競合状態と呼ぶ (リスト 2.3). a++ などは一見一つの動作のようだが, a の値を読み込み, 1 を足して a に書き込むという三つの動作から成り立っている. このような操作はリード・モディファイ・ライト操作と呼ばれる.

リスト 2.3 のようなコードでメソッド M が二つのスレッドに呼ばれた場合, 最終的な値は最初の値に 2 増えた値が期待される. しかし実際には a++ がリード・モディファイ・ライト操作であるためあるスレッドが値 a を読み込んだ後 1 を足した値を返す前にもう片方のスレッドが a を読み込んでしまう可能性がある. するとどちらのスレッドも最初の値を読み込み, 1 を足して a に書き込むため最終的な値が最初の値に 1 を足した値になってしまう.

```
1 // 別のスレッドが同時にメソッドMを呼ぶとa++が競合状態となる
2 volatile int a;
3
4 public void M() {
5     a++;
6 }
```

リスト 2.3: 競合状態を起こすプログラムの例の断片 (Java)

2.1.4　データ競合

スケジューリングによって異なるスレッド間の処理の順序がプログラマの想定と異なる場合があり得るとき, その順序によってエラーが発生するバグをデータ競合と呼ぶ (リスト 2.4).

```
1  // それぞれ別のスレッドがM1とM2を呼び, その順番が制限されていない場合
2  // スケジューリングによってM1とM2の呼ばれる順番が変わり,計算結果が変わってしまう
3  public void M1() {
4      synchronized(A) {
5          a*=2;
6      }
7  }
8
9  public void M2() {
10     synchronized(A) {
11         a++;
12     }
13 }
```

リスト 2.4: データ競合を起こすプログラムの例の断片 (Java)

2.1.5　飢餓状態

スレッドが必要な資源にアクセスできないままでいる状態を飢餓状態と呼ぶ. 例としてあるスレッドがロックを保持したまま無限ループなどを起こしてしまい, 他のスレッドがそのロックの解法を待つ場合などがある.

2.1.6　ライブロック

スレッドが動作しながらも処理が進められなくなることをライブロックと呼ぶ.

2.1.7　本研究で焦点を当てるバグ

デッドロックは再現できれば IDE やツールによってデッドロックを引き起こしているスレッドやそのロックの箇所の検出が可能であり, その場合原因の特定は難しくない.

競合状態およびデータ競合ではバグの原因を特定する際には原因となるコードを特定するために経過を print 文などを使って追っていく必要がある. よってデッドロックに比べ原因の特定が困難である. また, print 文を用いた場合はプローブ効果に留意する必要がある. プローブ効果とは print 文の追加やデバッガーのプログラムへの干渉などにより元のプログラムになかった新しいバグが発生してしまうことである. プローブ効果を回避するトレーサーの研究もされている [8].

特に後述する関連研究 [2] よりデータ競合は並行性バグの中でも大きい割合を占める. またライブロックや飢餓状態は比較的少ない.

したがって本研究では並行性バグの中でも特にデータ競合や競合状態に焦点を当てる.

2.2　デバッグ手法

手作業で並行性バグのデバッグを行うのは困難である.

デッドロックのデバッグを行うのであれば, Java の場合 Java Debugger(jdb) や IDE の提供するデバッガなどを用いる方法がある. 手作業でこれらのようなツールや機能を用いずにデバッグする場合, 各スレッドから print 文などでスレッドの状態遷移を調べ, デッドロックした箇所を探す必要がある. しかし, こうした場合にはどのロックがデッドロックを引き起こしているかわからないため, 広範囲にわたって調べなくてはならない.

よりデバッグの容易な IDE によるデバッグの例として Eclipse でデッドロックをデバッグをする場合を示す. まず問題のプログラムをデバッグビルドで実行し, デッドロックの症状が出るまで繰り返す. デッドロックした場合, Eclipse の機能を使ってプログラムを一時停止するとデッドロックしている箇所が示されるのでその情報を基に原因の箇所を調べてプログラムを書き直せばよい. 図 2.1 において赤字で示されているスレッドがデッドロックを引き起こしているメソッドである. どの行で停止しているか, どのスレッドとデッドロックを引き起こしているかが表示される. 図 2.1 では Thread-0 が id=20 のオブジェクトを所有して id=19 のオブジェクトの解放を待ち, Thread-1 が id=19 のオブジェクトを所有して id=20 のオブジェクトの解放を待っているためデッドロックが発生していること, 両スレッドがどちらもソースコードの 27 行目でデッドロックを起こしていることがわかる.

競合状態やデータ競合によって値が不正になる場合, 上記の方法ではデバッグできない. ツールを用いない場合, 不正な値を assertion を用いて検出する, print 文を用いて途中経過を確認するなどで原因箇所を特定する必要がある. Java では assert 文により assertion を実装できる. assert 文で条件を指定し, その条件に満たない場合 assert 文で指定された例外がスローされる.

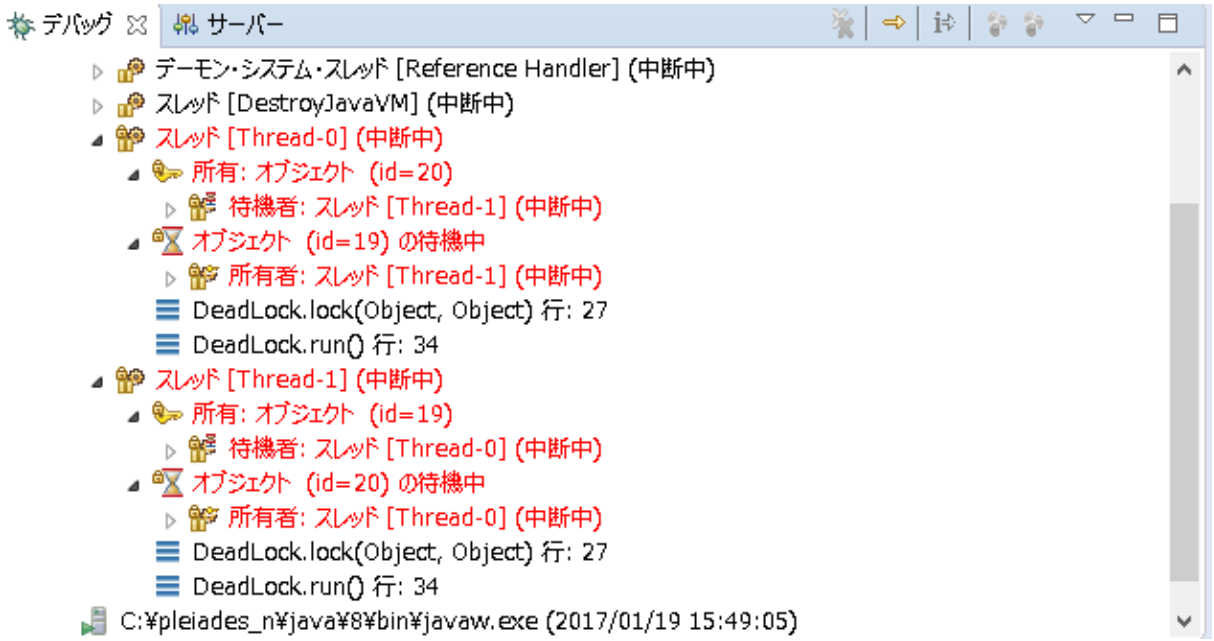

図 2.1: Eclipse でデッドロックをデバッグする例

他にもデバッグを支援する方法が研究されてきた. デバッグ支援のツールとしてはペトリネット (図 2.3) を拡張したものを Java のマルチスレッドプログラムの実行時の情報を基に作成し, そのペトリネットを基にそのスケジューリングを何度も再現し, さらにプログラムの動作を可視化することでデバッグを支援するツール [9] が例に挙げられる. ペトリネットとは離散分散システムを表現するためのモデルである. 図 2.3 は図 2.2 のプログラムをペトリネットで表したものである. 並行性バグを検出するツールとしては各オブジェクトにアクセス対象のメモリオブジェクト, アクセスするスレッドの ID, アクセス時にそのスレッドが保持するロック集合, アクセスが READ か WRITE か, アクセスするファイル名と行番号を合わせてアクセスイベントと定義して保持し, それを用いてデータ競合を検出するツール, およびそれを C プログラムのために拡張したツールなどが考案されている [6].

```
class  MyThread extends Thread{
   public void run(){
        statementR1;
        synchronized(lock){
             statementR2;
             lock.wait();
             statementR3; }
        statementR4; } }
void method(){
   statementM1;
   new MyThread().start();
   statementM2;
   while(ex){ statementM3; }
   statementM4; }
```

図 2.2: ペトリネットの元になったプログラム

(参考文献 [9] より引用)

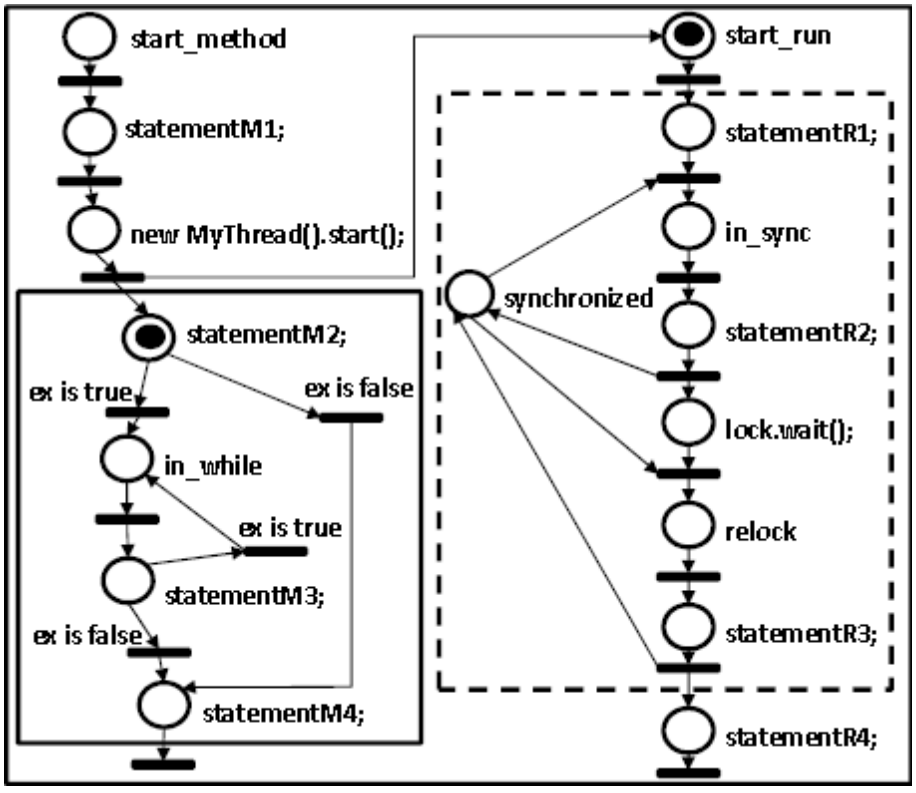

図 2.3: ペトリネットの例

(参考文献 [9] より引用)

2.2.1　モデル検査

確実でないバグを発見するためにモデル検査をデバッグに用いる方法がある [7]. ここで, モデル検査とは以下のようなものである.

> モデル検査とは，対象システムを有限状態遷移系にモデル化し，そのモデルが時相論理式で表現した性質を満たすか否かをモデルの取り得る全状態空間の探索により網羅的に検査する手法である (モデル検査のデバッグへの適用 [7] より引用)

また, 同論文では以下のように述べられている.

> デバッグでは，作業者が原因を予測した上で，その仮説に基づいて，ソースコードを関係箇所のみを精査したり，特定の処理に着目したテストデータを入力して処理を実行している．(中略) しかし，発生頻度の低い不具合を見出すには，何回も繰り返して多くの組み合わせをチェックする必要があり，その労力が大きくなる．さらに，予測した原因が正しくなかった場合には，新たな原因予測を行い，その都度，ソースコードを見直したり，テストデータを作成して処理実行を繰り返すため，作業の効率化は難しい.
>
> 一方，モデル検査は，明示的な状態遷移モデルを作成する必要があり，そのための作業時間を必要とするが，モデルに発生する全ての組み合わせ条件をモデル検査器が自動検査するため，必要な要素をモデル化できれば見逃しがなく確実にチェックできる．さらに，1 つのモデルに対して複数の検査項目を自由に設定して検査できるため，原因究明が難しい不具合で検査項目が多くなるほど効率が良くなる．論理演算により自動検査を行うので，チェックしている過程の全てを直接的に確認することはできないが，検査項目を満足しない場合には反例が出力されるため，これを読み取ることで，不具合の発生原因を究明できる．これらのことから，モデル化が容易であればモデル検査をデバッグに使用するができ，デバッグ作業の効率化に有効であると考えられる．(モデル検査のデバッグへの適用 [7] より引用)

以上より, モデル検査によるデバッグは再現性の低いバグの発見と原因追及において有効であり, その点で並行性バグのデバッグとして有効であるといえる.

しかし全ての状態を検査するためプログラムが大きくなると状態爆発が発生してしまうことも同論文に示されている. 特にマルチスレッドプログラミングにおいては複数のスレッドが動作しているため, スレッドの数および各スレッドにあり得る状態数の乗算した分だけ全体の状態数が発生してしまう.

2.2.2　Java Path Finder

手作業でデバッグを行う場合, 並行性バグではバグが検出されるまでテストを繰り返す必要がある.

しかし Java Path Finder(JPF) を用いるとマルチスレッドのプログラムにおいて全てのスケジューリングを検査できる. JPF は Java プログラムのバイトコードを基にモデル検査 [7] を行うオープンソースソフトウェアである. NASA, エイムズ研究センター, RIACS によってサポートされている.

JPF によってマルチスレッドのプログラムについて網羅的に全てのスケジューリングで実行できる．またデッドロックを検出し, その箇所を示す機能がある.

JPF を key1 と key2 の文字列を表示し, デッドロックを起こすプログラム (リスト 2.5) にかけた時の出力結

果はリスト 2.6 のようになる. 13 行目以下の error1 からどのようなエラーまたはバグが発生したか, 22 行目以下の snapshot #1 にはそのエラーが起きた時のスタックトレースが, 40 行目以下の results にはスローされた例外などが表示される.

```
1  public void lock(Object k1, Object k2)
2  {
3      synchronized(k1)
4      {
5          synchronized(k2)
6          {
7              System.out.println(k1);
8          }
9      }
10 }
11
12 public void run()
13 {
14     lock(key1, key2);
15 }
16
17 public static void main(String[] args)
18 {
19     String key1, key2;
20     key1 = new String("key1");
21     key2 = new String("key2");
22     DeadLock d1, d2;
23     d1 = new DeadLock(key1, key2);
24     d2 = new DeadLock(key2, key1);
25
26     d1.start();
27     d2.start();
28 }
```

リスト 2.5: デッドロックを起こすプログラムの例の断片 (Java)

```
1  ====================================================== system under test
2  test.DeadLock.main()
3
4  ====================================================== search started: 17/01/19
5  16:59
6  key1
7  key2
8  key1
9  key2
10 key2
11 key2
12
13 ====================================================== error 1
14 gov.nasa.jpf.vm.NotDeadlockedProperty
15 deadlock encountered:
16   thread test.DeadLock:{id:1,name:Thread-1,status:BLOCKED,priority:5,isDaemon:fa
17 lse,lockCount:0,suspendCount:0}
18   thread test.DeadLock:{id:2,name:Thread-2,status:BLOCKED,priority:5,isDaemon:fa
19 lse,lockCount:0,suspendCount:0}
20
21
22 ====================================================== snapshot #1
23 thread test.DeadLock:{id:1,name:Thread-1,status:BLOCKED,priority:5,isDaemon:fals
24 e,lockCount:0,suspendCount:0}
25   owned locks:java.lang.String@15b
```

```
26    blocked on: java.lang.String@15e
27    call stack:
28          at test.DeadLock.lock(DeadLock.java:19)
29          at test.DeadLock.run(DeadLock.java:28)
30
31  thread test.DeadLock:{id:2,name:Thread-2,status:BLOCKED, priority:5,isDaemon:fals
32  e,lockCount:0,suspendCount:0}
33    owned locks:java.lang.String@15e
34    blocked on: java.lang.String@15b
35    call stack:
36          at test.DeadLock.lock(DeadLock.java:19)
37          at test.DeadLock.run(DeadLock.java:28)
38
39
40  ====================================================== results
41  error #1: gov.nasa.jpf.vm.NotDeadlockedProperty "deadlock encountered:    thread
42   test.DeadLock:{id:..."
43
44  ====================================================== statistics
45  elapsed time:       00:00:01
46  states:             new=15,visited=3,backtracked=11,end=5
47  search:             maxDepth=10,constraints=0
48  choice generators:  thread=14 (signal=0,lock=7,sharedRef=0,threadApi=2,reschedul
49  e=5), data=0
50  heap:               new=374,released=69,maxLive=366,gcCycles=18
51  instructions:       3746
52  max memory:         59MB
53  loaded code:        classes=62,methods=1475
54
55  ====================================================== search finished: 17/01/19
56   16:59
```

リスト 2.6: デッドロックを起こすプログラムの実行例の断片 (Java)

全てのスケジューリングを試すため, プログラムが大きくなるほど実行性能が著しく悪化する. また仕様上サーバ/クライアント通信など通信を行うプログラムは検査にかけることができない.

2.3　バグ入りプログラムの利用と収集

バグ入りプログラムはデバッグ手法の学習に利用することができる. バグ入りプログラムに対してデバッグを行うことでデバッグ手法を実践的に学ぶことができる.

また, デバッグ手法やデバッグツールの評価に利用することができる. 実際に Toward a Benchmark for Multi-Threaded Testing Tools[4] ではバグ入りプログラムをマルチスレッドのテストツールのベンチマークに利用している.

バグ入りプログラムを収集する方法として既存のオープンソースソフトウェアのバグレポートから収集する方法がある. 例えばオープンソースのフレームワークである Apache Hadoop[1] ではバグレポートが公開されている. 関連研究 [2] では, 実際に Apache Hadoop のバグレポートから並行性バグに関するものを抜き出し, 研究に利用している.

こうした既存のバグの情報を基にバグを引き起こす傾向をまとめたものをバグパターンと呼ぶ.

また, 既存のバグを収集する方法以外には新たにバグを作る方法が考えられる. Toward a Benchmark for Multi-Threaded Testing Tools[4] ではマルチスレッドのテストツールのベンチマークのために学生に並行性バグのバグ入りプログラムを作成させている.

第 3 章　関連研究

3.1　A Study of Concurrency Bugs in an Open Source Software[2]

この研究は並行性バグと非並行性バグを比較, 検証するものである. オープンソースのフレームワークである Apache Hadoop[1] のバグレポートから並行性バグに関するものとそれ以外に分けて抽出し, 集計している.

並行性のバグはそうでないバグに比べて少ないこと, 修正にかかる時間の差はあるが大きくないこと, 非並行性のバグに比べわずかに並行性バグのほうが深刻であることを示している.

3.1.1　本研究との関連

並行性バグの各種類についてその深刻さと割合が示されている (図 3.1).

図 3.1 よりデータ競合が並行性バグ全体の内 40% を占めており, また深刻さが高くなるほどデッドロックの

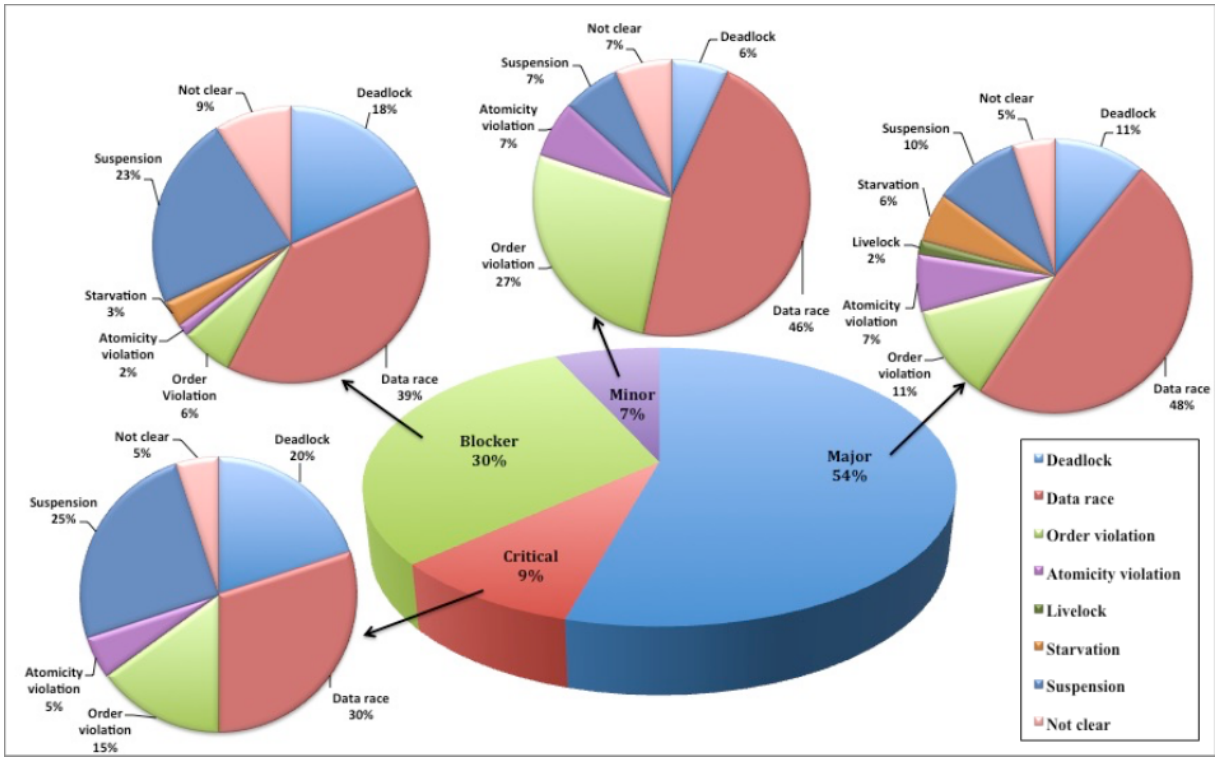

図 3.1: 並行性バグの深刻さと割合 ([2] より引用)

割合が増えていることや競合状態が一定量含まれていること, ライブロックや飢餓状態が占める割合が非常に少ないことがわかる.

このことから並行性バグの中でデータ競合や競合状態がライブロックや飢餓状態に比べ大きい割合を占めていることがわかるので, データ競合や競合状態のデバッグが重要であることがわかる.

デッドロックも深刻で大きい割合を示しているが, 第 2 章マルチスレッドのバグとデバッグ手法で前述したように再現後のデバッグが IDE などを用いることで可能である.

したがって本研究ではデータ競合や競合状態を中心にバグ入りプログラムを作成する.

3.2　モデル検査のデバッグへの適用 [7]

この研究はモデル検査をデバッグに利用できないか検討する研究である.

この研究では C 言語で書かれ, 2 つのプロセスを同期させずに使用する 1730 行のプログラムをモデル検査を用いてデバッグし, モデル化にかかった時間, モデル検査にかかった時間などをまとめている. その結果従来よりも早くデバッグが可能であると示している.

3.2.1　本研究との関連

同じくモデル検査の仕組みを用いる JPF によってデバッグを行う. また, この研究における実験ではモデルの準備や作成, 改造にかかった時間を除いた 1 回のモデル検査にかかった時間は最大で 6 時間となっている.

第 4 章　作成したバグ入りプログラム

第 2.3 節よりバグ入りプログラムはデバッグ手法の学習およびデバッグ手法やデバッグツールの評価に用いることができるので, それらを促進するためにバグ入りプログラムを作成する.

本研究では既存のバグの収集ではなく, 新たにバグ入りプログラムを作成する. バグ入りプログラムの作成は手動による作成と自動生成プログラムによる作成を試みた.

メモリの可視性は環境によって変化するため本研究ではメモリの可視性に由来するバグを可能な限り避けることとする.

原因の特定が困難で並行性バグの中でも比較的大きい割合を占める競合状態とデータ競合を中心にバグの作成を試みた.

4.1　実行環境

使用言語は全て Java である. Java ではマルチスレッドプログラミングがサポートされている. Java SE は Java 8 を使用し, IDE は Eclipse Neon.1 Release (4.6.1) を使用した.

4.2　手動で作成したバグ入りプログラム

並行動作を行う二つのサンプルケースについて, 正しいプログラムを基にその一部を手作業で変更することで複数のバグ入りプログラムを作成した.

4.2.1　BuggyBank

正しいプログラムを用意し, そのプログラムを基に三つのバグ入りプログラムを作成した.

4.2.1.1　正しいプログラム

正しいプログラムとして, デッドロックの例としてよく用いられる架空の銀行口座のプログラムを作成した. 元となるプログラムはサーバ/クライアント通信でクライアント側から命令を送り, サーバ側がその命令に応じて口座の作成や口座に対する入金, 引き出し, 送金の操作を行うプログラムである. 各操作は口座の持ち主である一つのクライアント以外は送金先として以外は使用しないこととした. スレッドは各クライアントに対して一つ作成される. この内, バグ入りプログラムはサーバ側のみ作成した.

正しいプログラムの口座の作成のメソッドを例に挙げる (リスト 4.1). このメソッドでは口座の ID を取得し, 口座を作成する動作がロックされている. 変数 curNum は次に読み込まれる時は一つ大きくなっているため同じ ID を持つ口座は生成されない.

```
1  public long makeAccount(long pass, long money)
2  {
3      synchronized(accounts)
4      {
5          long id;
6          id = curNum++;
7
8          accounts.put(id, new Account(id, pass, money));
9          return id;
10     }
11 }
```

リスト 4.1: 正しい口座の作成

4.2.1.2　作成したバグ入りプログラム: BuggyBank

まず初めにロックを全く使わずスレッド間での同期を取らないようにした. 具体的には口座の作成ならリスト 4.1 の 3, 4, 10 行目, 入金ならリスト 4.2 の 3, 4, 13 行目, 引き出しならリスト 4.3 の 3, 4, 18 行目を削除した. 口座の作成に対し図 4.1 のように異なる口座には異なる ID が与えられることが期待されるが, このプログラムでは口座の作成が同時に行われた場合に作成された口座の数 curNum に対し競合状態が発生し, ID の重複および欠番が発生するバグがある (図 4.2).

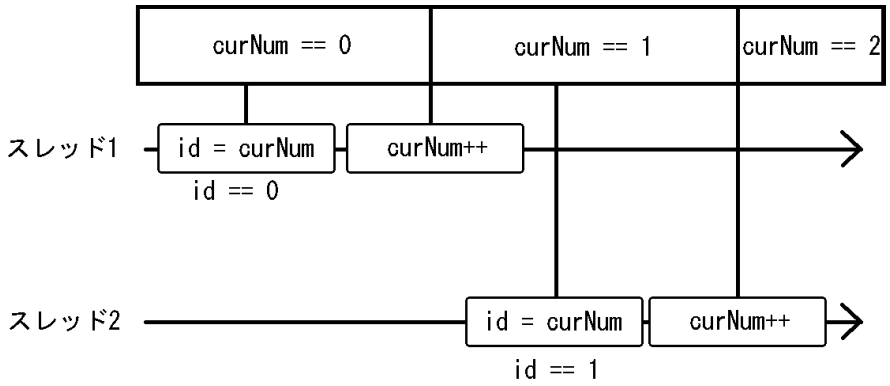

図 4.1: 口座の作成において期待される動作

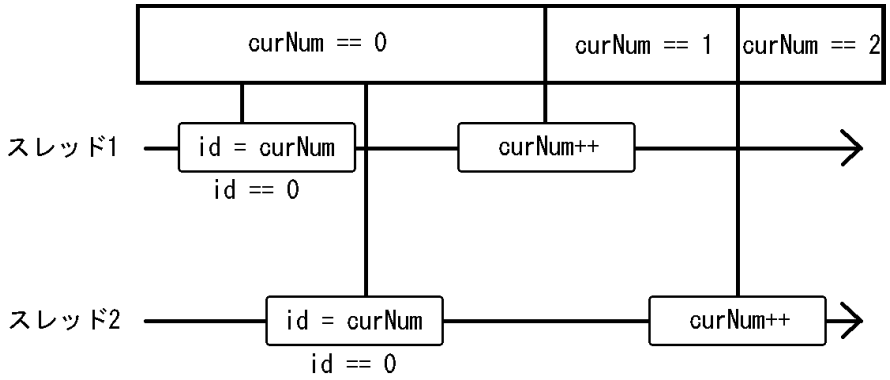

図 4.2: 口座の作成における競合状態

また, 入金および引き出しはリード・モディファイ・ライト操作である. したがって正しく動作する場合 (リスト 4.2) は図 4.3 のように動作する. ここでこのバグ入りプログラムでは入金操作 (リスト 4.2) のロックおよび引き出し操作 (リスト 4.3) のロックが存在しないため, 送金操作は送金元からの引き出しと送金先への入金の二つから成り立つため, 送金と送金先の入金, または引き出しが同時に起こった場合, 口座の残高に対して競合状態が発生し, 残高が不正な値になるバグがある (図 4.4).

```
1  public long inputMoney(long value) throws IllegalArgumentException
2  {
3      synchronized(this)
4      {
5          if(money + value > MONEYMAX)
6          {
7              throw new IllegalArgumentException(getErSt() + "money + value MONEYMAX");
8          }
9
10         money += value;
11
12         return value;
13     }
14 }
```

リスト 4.2: 正しい入金

```
1  public long outputMoney(long inpass, long value) throws IllegalArgumentException
2  {
3      synchronized(this)
4      {
5          if(!checkPass(inpass))
6          {
7              return -1;
8          }
9
10         if(money < value)
11         {
12             throw new IllegalArgumentException(getErSt() + "money < value");
13         }
14
15         money -= value;
16
17         return value;
18     }
19 }
```

リスト 4.3: 正しい引き出し

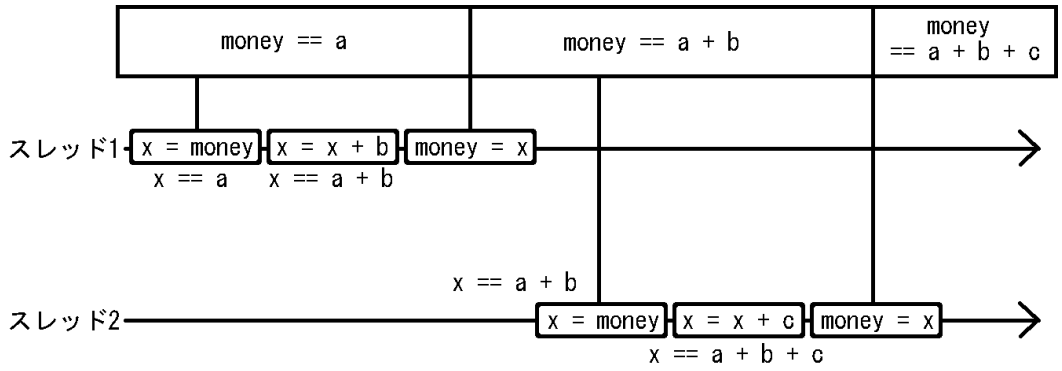

図 4.3: 入金, および引き出しに期待される動作

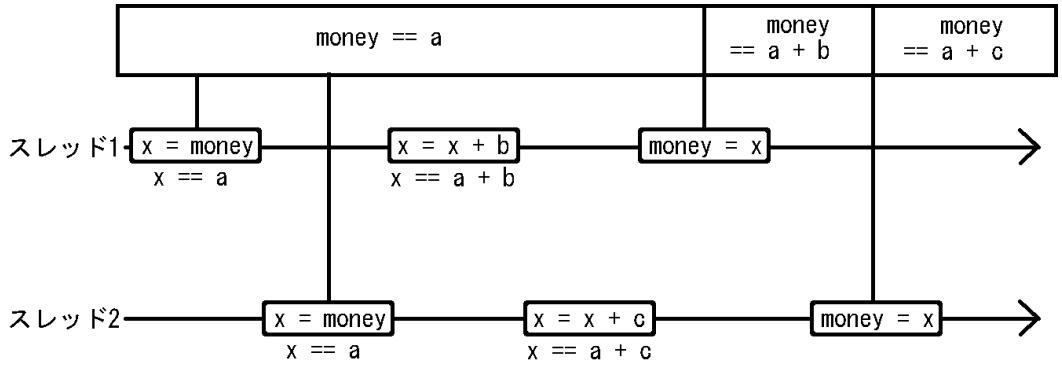

図 4.4: 入金および引き出しにおける競合状態

次に口座の作成のみロックした. このプログラムでは残高に対する競合状態が解決していない.

さらに入金, 引き出し, 送金をロックしたが, 送金について送金元と送金先をそのままの順序でロックするようにした (リスト 4.5) このとき互い違いに送金を同時に行うときお互いに送金元, 送金先の順でロックを取得するため, ロックの取得順序は逆になる. したがってこのバグ入りプログラムはロック順デッドロックを引き起こす. 正しいプログラムでは口座の ID を使ってロック順序が一意に定まるようにしている (リスト 4.4).

```
1  private String sendMoneysub(long id, long pass, long value, long toid, Account ac1, Account
     ac2)
2  {
3      long v = ac1.outputMoney(pass, value);
4
5      if(v < 0)
6      {
7          return new String("error: ID: " + id + ": " + "pass is not correct");
8      }
9
10     v = ac2.inputMoney(v);
11
```

```
12      return new String("send: " + id + " to " + toid + ": " + v);
13  }
14
15  public String sendMoney(long id, long pass, long value, long toid)
16  {
17      Account ac1 = getAccount(id),
18              ac2 = getAccount(toid);
19
20      String s;
21      if(id < toid)
22      {
23          synchronized(ac1)
24          {
25              synchronized(ac2)
26              {
27                  s = sendMoneysub(id, pass, value, toid, ac1, ac2);
28              }
29          }
30      }
31      else
32      {
33          synchronized(ac2)
34          {
35              synchronized(ac1)
36              {
37                  s = sendMoneysub(id, pass, value, toid, ac1, ac2);
38              }
39          }
40      }
41
42      return s;
43  }
```

リスト 4.4: 正しい送金

```
1   public String sendMoney(long id, long pass, long value, long toid)
2   {
3       Account ac1 = getAccount(id),
4               ac2 = getAccount(toid);
5
6       long v;
7       synchronized(ac1)
8       {
9           synchronized(ac2)
10          {
11              v = ac1.outputMoney(pass, value);
12
13              if(v < 0)
14              {
15                  return new String("error: ID: " + id + ": " + "pass is not correct");
16              }
17
18              v = ac2.inputMoney(v);
19          }
20      }
21
22      return new String("send: " + id + " to " + toid + ": " + v);
23  }
```

リスト 4.5: デッドロックを起こす送金

4.2.2　MapReduce[3] モデルに基づくプログラム

4.2.2.1　MapReduce モデル

MapReduce とは分散プログラミングモデルの一つである, 入力を分割, クラスター上で並列に処理してキーと値の組を出力とする Map 処理と Map 処理の結果をキー毎にまとめたものを入力としてクラスター上で並列に処理し, 出力を作る Reduce 処理の二段階に分けられている.

4.2.2.2　正しいプログラムの概要

この MapReduce モデルに基づいたプログラムを作り, そのサーバ側のプログラムを基に六つのバグ入りプログラムを作成した.

正しいプログラムは数字列のファイルから 0 から 9 までの各数字がそれぞれいくつあるのかを数えるプログラムである.

各クライアントがそれぞれ一つずつファイルを読み込みファイルの一行をサーバに送る. サーバはその行をクラスタに見立てた複数の計算用クライアントに送り, 計算用クライアントが数字を数える. これが Map 処理であり, 各行が分割された Map 処理の入力となっている. 計算用クライアントから結果を得たサーバが数字毎に結果をまとめて計算用クライアントに送り, 計算用クライアントが合算した結果をサーバに返すようになっている.

サーバはクライアントから入力の行を受け取る, あるいは Map 処理の結果をクライアントから受け取るとそれをタスクとしてキューに入れるようになっている. サーバはキューに入っているタスクを順に計算用クライアントへ送るようになっている (図 4.5).

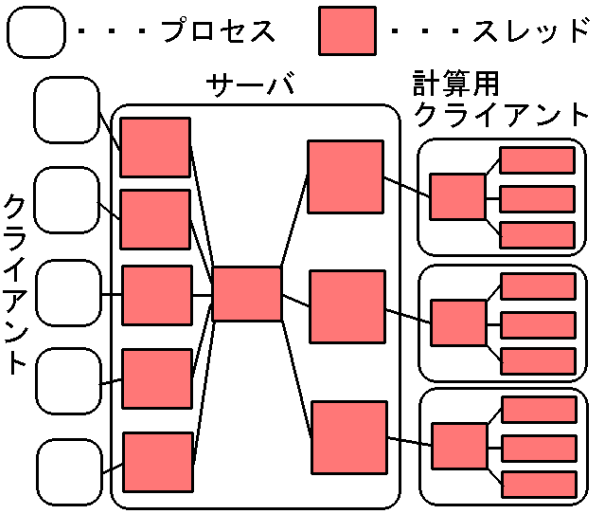

図 4.5: MapReduce モデルのプログラムの概略図

4.2.2.3　作成したバグ入りプログラム: MapReduce モデル

バグ入りプログラムはデバッグ手法の学習およびデバッグ手法やデバッグツールの評価に用いるのでバグパターンを大きく無視したものはバグ入りプログラムとして有用ではないと考えられる.

作ったバグ入りプログラムの内三つはどんなエラーに対しても必ず処理をやり直すようにする, 意味のないロックによりデッドロックを作るなどしたため評価から除外した.

正しいプログラムでは計算用クライアントの SocketException が発生した際にその計算用クライアントに任せていたタスクをキューに戻す処理があるが, 一つ目のバグはその処理でロックが不十分であり, SocketException によりタスクをキューに戻す処理中にタスクをキューに加えようとするとキューの範囲外に対し処理を行ってしまう, タスクが行方不明になり無限ループが発生するなどのバグが生じる競合状態である.

二つ目のバグはリカバリできないエラーに対して計算用クライアントを終了させる際, ロックの不足が原因で計算用クライアントの停止前にその計算用クライアントへタスクを送ってしまい見失うデータ競合である.

三つめはサーバが計算用クライアントを取得する前にタスクが送られた場合初期化されていないフィールドが参照されてしまうデータ競合である.

4.3　バグ入りプログラムの自動生成

完全に何もない状態からバグ入りプログラムの自動生成を試みるプログラムと正しいプログラムを基にバグ入りプログラムを生成する二つの方法を試みた.

4.3.1　完全ランダムな自動生成

一か所だけ並行性バグを含むようランダムにロックを作り, 間を無意味なプログラムで埋めるバグ入りプログラムの自動生成プログラムを作った. リスト 4.6 はこの方法により自動生成されたプログラムである. このプログラムでは Test1 クラスのオブジェクト Test1_o と Test2 クラスのオブジェクト Test2_o が別々のスレッドで動いている. Test1_o の Method5 の実行と Test2_o の Method11 の実行が同時に行われた時, リスト 4.6 の 9 行目と 12 行目, 23 行目と 25 行目が互い違いのロックになっているためデッドロックを引き起こす.

```
1  // class Test1
2  public int Method5() {
3      int result = 0;
4
5      int r23[] = new int[2];
6
7      r23[0] = 963; r23[1] = 908;
8      if(r23[0] > r23[1]) { int temp = r23[0]; r23[0] = r23[1]; r23[1] = temp; }
9      synchronized(this) {
10         int r24[] = new int[2];
11         // (中略)
12         synchronized(Test1_o) {
13             // (中略)
14         }
15         result += r24[0] + r24[1];
16     }
17     return result;
18 }
19
20 // class Test2
```

```
21 public int Method11() {
22     // (中略)
23     synchronized(this) {
24         // (中略)
25         synchronized(Test2_o) {
26             // (中略)
27         }
28         // (中略)
29     }
30     return result;
31 }
```

リスト 4.6: 完全ランダムな自動生成により作成されたプログラムの断片

しかしソースコードに意味がないためデバッグ手法を学ぶサンプルとしては不適切である. またある競合状態を作るコードはそのロックの場所が変わりソースコードが変化していても本質的には同じバグであり, 自動生成する意義が見いだせなかった.

4.3.2　正しいプログラムからの生成

正しいプログラムを基にバグ入りプログラムを作成するプログラムとその手法および失敗したが試みた方法, 自動生成によって作成したかったバグ入りプログラムについて説明する.

4.3.2.1　実際に用いた方法: ロックの消去

正しいプログラムに存在する各 synchronized ブロックについてそれぞれをそのまま残したもの, 削除したものを網羅的に組み合わせた. 例えばリスト 4.7 において 4, 5, 16 行目を消去したものとそうでない物などである. また, プログラムのファイルから各バグ入りファイルの作成を行うステップとそれらを組み合わせるステップに分け, 一部手作業で作成したバグ入りファイルを混ぜた. 例えばリスト 4.8 のようにロック順を制御しているロックをリスト 4.9 のようにロックを消去するなどである.

```
1  // 正しいプログラム
2  public int inputMoney(int pass, int money) throws IOException
3  {
4      synchronized(this)
5      {
6          // 入力確認
7          if(money < 0)
8          {
9              throw new IOException("input < 0");
10         }
11
12         // (中略)
13
14         this.money += money;
15         return money;
16     }
17 }
```

リスト 4.7: ロックの消去例

```
1  // 送金
2  private int sendMoney(Account acc1, Account acc2) throws IOException
3  {
```

```
4       int pass2 = acc2.getPass();
5       int v = 0;
6
7       // (中略)
8
9       v = acc2.inputMoney(pass2, v);
10      acc2.RLockMinus();
11
12      return v;
13  }
14
15  @Override
16  public void run()
17  {
18      try
19      {
20          Account acc1 = bank.getAccount(account1),
21          acc2 = bank.getAccount(account2);
22
23          int v;
24          if(account1 < account2)
25          {
26              synchronized(acc1) {
27                  synchronized(acc2) {
28                      v = sendMoney(acc1, acc2);
29                  }
30              }
31          }
32          else
33          {
34              synchronized(acc2) {
35                  synchronized(acc1) {
36                      v = sendMoney(acc1, acc2);
37                  }
38              }
39          }
40
41          // (中略)
42      }
43      catch (IOException e)
44      {
45          customer.sendMessage("error: can't send: " + e.getMessage());
46          e.printStackTrace();
47      }
48  }
```

リスト 4.8: 手作業で混ぜたファイルの例: 正しいファイル

```
1   @Override
2   public void run()
3   {
4       try
5       {
6           Account acc1 = bank.getAccount(account1),
7           acc2 = bank.getAccount(account2);
8
9           int v = 0;
10          int pass2 = acc2.getPass();
11
12          // (中略)
13
14          if(v >= 0)
```

```
15          {
16              v = acc1.outputMoney(pass, value);
17              acc1.RLockMinus();
18
19              if(v >= 0)
20              {
21                  v = acc2.inputMoney(pass2, v);
22                  acc2.RLockMinus();
23              }
24          }
25
26          // (中略)
27      }
28      catch (IOException e)
29      {
30          customer.sendMessage("error: can't send: " + e.getMessage());
31          e.printStackTrace();
32      }
33  }
```

リスト 4.9: 手作業で混ぜたファイルの例: 誤ったファイル

また各バグ入りプログラムをコンパイルすることでコンパイルエラーを起こすプログラムが生成したバグ入りプログラム群に混入するのを防げるが, 実行性能の関係上困難であるため断念した. ただし今回作成したバグ入りプログラムにコンパイルエラーを起こすものがないことは確認済みである.

ただしロックの数と手作業で作成したバグ入りファイルの数から, 組み合わせの数が 9000 万通り以上であり, 全ての組み合わせを作ることが実行性能の関係上困難であり, また本質的に同じバグ入りプログラムが多数生成されてしまっている. そこである synchronized ブロック X が存在しない時別の synchronized ブロック Y だけが存在するのは明らかに不自然であるというような場合にそれらのロックを一つの組として扱い, 一つが存在する場合は他も全て存在し, そうでない場合は他も全て削除するようにした. その結果 196607 のバグ入りプログラムができた. このロックの対応付けは手作業で行い, ソースコードに直接注釈として書き込んだ. この注釈は自動生成時には生成されたバグ入りプログラムのソースコードから削除される.

またバグ入りプログラムの自動生成時にある synchronized ブロックが削除された場合どのようなバグが起きるかおよびそのバグは何が原因かを注釈として正しいソースコードに書き込み, それをドキュメントに書き出すようにした. 複数個所の synchronized ブロックが削除されたことによりバグが発生する場合はそれを明記してその内一つの synchronized ブロックの手前に書き込んだ.

またこの手法では競合状態のバグが発生することが期待できるが, 同時にメモリの可視性に由来するバグが発生してしまう. 本研究ではメモリの可視性に由来するバグの評価を除外したいので評価の際にはメモリの可視性に由来するバグの原因になる変数に volatile 修飾子をつけて揮発性変数にする. 揮発性変数はロックされていない場合でもローカルメモリからではなく更新された値を参照するようになる変数のことである.

さらに wait メソッドはその wait メソッドを呼ぶオブジェクトに対する synchronized ブロックの中から呼び出さなければエラーとなる仕様であるため, その synchronized ブロックを取り除くとエラーになってしまう. これは並行性バグではない.

4.3.2.2　正しいプログラム

正しいプログラムは既存のプログラムを利用することも考えられるが, 今回は自作したプログラムを使用した. 正しいプログラムは BuggyBank と同じ架空の銀行口座プログラムだが, 操作に口座の削除とパスワードの

変更を加えクライアントからの入力をタスクとして一つの ExecutorService で処理する.

ExecutorService は Java の提供するクラスの一つである. 複数のスレッドを保持し, 与えられたタスクをそのスレッドに与えて並行に処理する仕組みをスレッドプールと呼ぶが, ExecutorService はそのスレッドプールを容易に利用できるようにしたクラスである. スレッドプールの全てのスレッドがタスクを実行している場合に ExecutorService にタスクが送信された場合は実行中のタスクが停止するまで待つなどの機能があり, これを利用することでスレッドプールの仕組みを実装する労力を省くことができる. 本研究では ExecutorService の内スレッドプールの持つスレッド数が固定される FixedThreadPool を用いて, スレッドの数は二十個とした.

また, Thread クラスの wait メソッドを用いると前述した非並行性バグが発生してしまう. これを避けるため wait メソッドの代わりに yield メソッドで代用している.

ソースコードの行数は 1188 行である.

第 4.3.2.1 の方法により正しいプログラムから作成したバグは

- 口座の作成における競合状態
- 入金および引き出し操作の競合状態
- ある口座の他の操作が終わるまでその口座の削除やパスワードの変更を停止する処理の競合状態による無限ループあるいは停止の失敗

などである.

4.3.2.3　ロックする区間の変更

次に synchronized ブロックの区間を変更する方法を試みた. リスト 4.10 はロックする区間の変更により正しいプログラムから作成され得る競合状態のバグ入りプログラムの断片である.

ただしロック周辺の行数から計算すると, この方法では一つのファイルで多い場合は 40 億通り以上の組み合わせが発生するため実行性能の関係上断念した.

またこの方法ではバグではなく単に冗長なロックが発生することがあり, 作成された全てがバグを含むとは限らない.

```
1  synchronized(bank.getAccLockKey())
2  {
3      if(bank.NoAction(acc2))
4      {
5          resbo = true;
6          break;
7      }
8      acc = bank.getAccount(acc2);
9      acc.RLockPlus();
10 } // 間違った位置
11 acc = bank.getAccount(acc1);
12 acc.RLockPlus();
13 bank.newTask(new Send(acc1, acc2, Integer.parseInt(inputs[2]),
14     Integer.parseInt(inputs[3]), bank, this));
15 // 正しい } の位置
16 break;
```

リスト 4.10: synchronized ブロックをずらした場合のプログラムの断片

リスト 4.10 は送金タスクを ExecutorService に加える処理の一部である. bank.getAccLockKey メソッドの

返り値に対するロックはその間口座の削除をタスクに加える処理を一時停止させる. ロックの中では送金先の口座 (acc2) が口座の削除タスクを ExecutorService に送っていないことを確認し, 送金が終わるまで口座の削除タスクを処理しないようにしている. ロック終了の間違った位置から正しい位置までの間では送金元の口座 (acc1) に対する処理と ExecutorService に送金タスクを加える処理が行われている.

正しい位置でロックが終了していれば ExecutorService に送金タスクを加えるまで口座の削除タスクは ExecutorService に送られない. したがって図 4.6 のように必ず送金タスクより後に口座の削除タスクが加わる. つまり口座の削除タスクの処理が開始されているならば送金タスクも必ず処理が開始されているか既に処理が終了している.

ここでロックの終了する位置が間違っている場合, 送金先の口座について口座の削除タスクがないことを確認してから送金タスクを ExecutorService に加えるまでの間に口座の削除タスクが ExecutorService に加えられる可能性がある. その場合口座の削除タスクが送金タスクより前に ExecutorService に加えられる (図 4.7).

このプログラムではスレッドプールの持つスレッド数を二十に制限しているため, 同様の競合が同時に二十起きた場合, スレッドプールの全てのスレッドが口座の削除タスクを行おうとして送金タスクの終了を待ち, 送金タスクはスレッドプールの全てのスレッドが実行中であるため実行中のタスク, つまり口座の削除タスクを待つ. これによってスレッドプールの全てのスレッドが処理を停止してしまう (図 4.8).

このバグは競合状態が同時に二十発生した場合にのみ顕在化するバグであるため発生確率は極めて低い.

またこの synchronized ブロックがない場合でも同様のバグを含んでいるが別の発生確率の高いバグも誘発される.

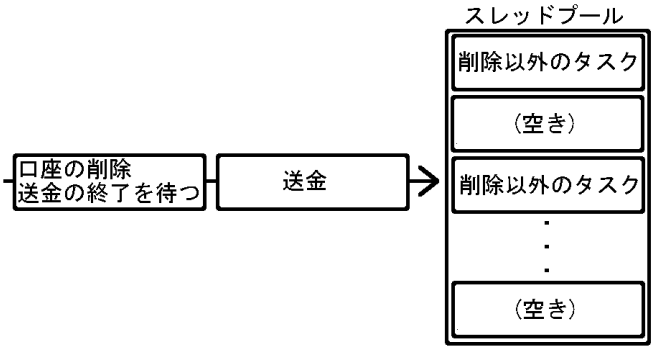

図 4.6: 正しい送金と口座の削除の順序

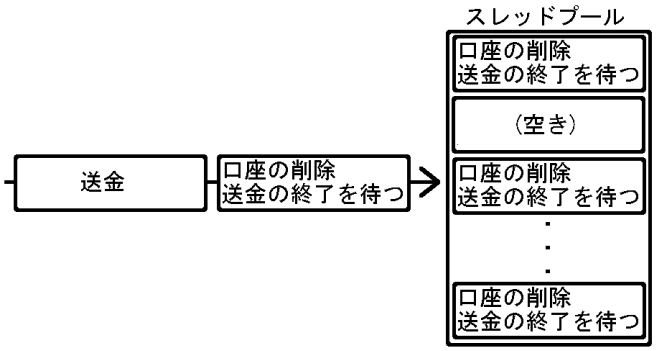

図 4.7: 誤った送金と口座の削除の順序

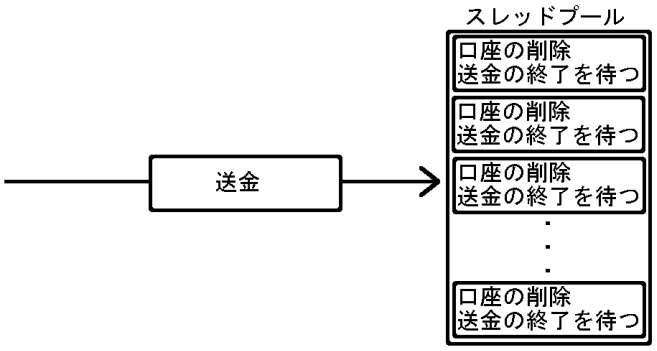

図 4.8: スレッドプール全てが停止する場合

4.3.2.4　データ競合

データ競合について, まず正しいプログラムでは口座の削除があった後に ExecutorService にその口座への送金処理を行わせないこと, その口座への送金処理が終わらない内に口座の削除をしないことという二つの順序の制限を行っている. この制限を取り払った場合送金先の口座の消失データ競合のバグが発生するが, そのためにメソッドを作り処理しているためこれらを自動的に認識して取り払うことが難しい.

データ競合のバグを正しいプログラムを基に手作業で作成した. 上記の順序制限二つを片方ずつ取り払ったものと両方取り払ったものである.

前者の制限を取り払ったものは口座の削除が決定した後に送金処理を受け付けてしまうため, 存在しない口座への送金が試みられてエラーが発生する.

後者の制限を取り払ったプログラムでは送金タスクが終了しない内にその送金先の口座の削除タスクが ExecutorService に与えられ処理が完了した場合, 存在しない口座への送金が発生する.

第 5 章　作成したバグ入りプログラムの評価

5.1　概要

実行環境は前章と同様である.

作成したバグ入りプログラムについて手作業によるデバッグと JPF によるデバッグを行い, それによってわかることをまとめた.

5.2　バグ入りプログラムのデバッグとその評価

手作業によるデバッグは第 2.2 節で示した方法を用いる.

手作業でデバッグする場合は途中経過の確認などで柔軟にデバッグが可能である.

JPF を用いる方法ではプログラムがサーバ/クライアント通信であるため, プログラムを書き換えて通信の代わりに特定の値を入出力するプログラムに書き換える. また競合状態を検出するために assertion や throw 文を用いてエラーを検出できるようにする.

書き換えたプログラムを JPF にかけてエラーのスタックトレースからデッドロックや競合状態を確認しソースコードを調べる.

JPF を用いる場合は全てのスケジューリングを検査するため, バグが顕在化するに確率に関わらず確実にバグを発見することが可能である. ただし第 2.2 章で前述したようにプログラムが大きい場合は実行効率が著しく悪化する.

以下 BuggyBank の口座作成のみをロックした例に手作業によるデバッグと JPF によるデバッグの例を示し, 各バグ入りプログラムについて手作業によるデバッグと JPF を用いたデバッグの違いを示す.

5.2.1　手作業によるデバッグの例

残高が不正な値になることから口座に対する各操作が原因だと考えられる. また, クライアント側のプログラムではリスト 5.1 のように命令を送りその結果を受けて記録をするのみであり, 計算結果に齟齬が起きていないことがわかる.

```
1  // 入金の命令を送るコード
2  public void input(Random r) throws IOException
3  {
4      int num = r.nextInt(idsnum);
5      long inputm = ((r.nextLong() % 1000) * 10);
6      if(inputm < 0)
```

```
7       {
8           inputm *= -1;
9       }
10
11      // 入金命令の送信
12      os.println("input " + ids[num] + " " + inputm);
13      String s = is.readLine();
14      if(s.contains("error"))
15      {
16          errornum++;
17          System.out.println(s);
18      }
19      else
20      {
21          inputm = Long.parseLong(s);
22          inputmoney += inputm;
23          System.out.println("ID: " + ids[num] + ": input: " + inputm);
24      }
25  }
```

リスト 5.1: クライアントのプログラムの断片

前提としてメモリの可視性に由来するバグを避けるため, メモリの可視性に由来するバグを防ぐ揮発性変数を用いることとする. サーバ側の入金操作, 出力操作を assert 文で確認すると引き出し操作が競合していることがわかる (図 5.1).

この例では引き出し操作が競合していることがわかるが, 引き出し操作だけロックしても競合は改善されない. ここでこの競合状態は二つのスレッドが同時に money フィールドを操作していることが原因であるとわかるので引き出し操作と競合している別の操作も assert 文により競合状態が検出されるはずである

よって引き出し操作の assert 文を一旦停止してから実行する. これにより入金操作が競合していることがわかる. 入金操作をロックするとバグが発生しなくなる. したがって入金と引き出しの両方の操作をロックすることでバグが修正される.

```
62
63          long m = money;
64          money -= value;
65          assert money == m - value: "output cause race condition";
66
67          return value;
Exception in thread "Thread-1" java.lang.AssertionError: output cause race condition
    at test.BuggyBank$Account.outputMoney(BuggyBank.java:65)
    at test.BuggyBank.sendMoney(BuggyBank.java:182)
    at test.BuggyBank$Window.run(BuggyBank.java:277)
```

図 5.1: assertion による競合状態の確認

5.2.2　JPF を用いたデバッグの例

バグ入りプログラムを JPF にかけた結果はリスト 5.2 の通りとなる. したがってリスト 5.2 の 23, 24 行目より結果の値が不正であるため, 競合状態あるいはデータ競合により不正な計算結果が行われていることがわかる.

```
1  (中略)
2  server: 1900
3  criant: 2000
4
5  ====================================================== error 1
6  gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
7  java.io.IOException: server: 1900 != criant: 2000
8          at test.BuggyBank.fin(test/BuggyBank.java:529)
9          at test.BuggyBank.run(test/BuggyBank.java:470)
10         at test.BuggyBank.main(test/BuggyBank.java:537)
11
12
13 ====================================================== snapshot #1
14 thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,isDaemon:false
15 ,lockCount:0,suspendCount:0}
16   call stack:
17         at test.BuggyBank.fin(BuggyBank.java:519)
18         at test.BuggyBank.run(BuggyBank.java:470)
19         at test.BuggyBank.main(BuggyBank.java:537)
20
21
22 ====================================================== results
23 error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.io.IOException: ser
24 ver: 1900 != criant: 2000 ..."
25
26 ====================================================== statistics
27 elapsed time:       00:00:01
28 states:             new=213,visited=71,backtracked=190,end=2
29 search:             maxDepth=136,constraints=0
30 choice generators:  thread=212 (signal=1,lock=14,sharedRef=126,threadApi=4,resch
31 edule=67), data=0
32 heap:               new=2455,released=1657,maxLive=988,gcCycles=273
33 instructions:       49569
34 max memory:         75MB
35 loaded code:        classes=82,methods=1643
36
37 ====================================================== search finished: 17/01/20
38  23:58
```

リスト 5.2: バグ入りプログラムを JPF にかけた結果

よってバグ入りプログラムの金額の計算に assert 文を加えて JPF にかけるとリスト 5.3 の通りとなる. したがってリスト 5.3 の 36, 37 行目より assert 文による警告がなされているため, 入金操作が競合していることがわかる.

```
1  (中略)
2  ====================================================== error 1
3  gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
4  java.lang.AssertionError: input cause race condition
5          at test.BuggyBank$Account.inputMoney(test/BuggyBank.java:165)
6          at test.BuggyBank.sendMoney(test/BuggyBank.java:311)
7          at test.BuggyBank$Window.run(test/BuggyBank.java:401)
```

```
8
9
10 ============================================================ snapshot #1
11 thread java.lang.Thread:{id:0,name:main,status:WAITING,priority:5,isDaemon:false
12 ,lockCount:0,suspendCount:0}
13   waiting on: test.BuggyBank$Window@298
14   call stack:
15         at java.lang.Thread.join(Thread.java)
16         at test.BuggyBank.fin(BuggyBank.java:505)
17         at test.BuggyBank.run(BuggyBank.java:470)
18         at test.BuggyBank.main(BuggyBank.java:537)
19
20 thread test.BuggyBank$Window:{id:1,name:Thread-1,status:RUNNING,priority:5,isDae
21 mon:false,lockCount:0,suspendCount:0}
22   call stack:
23         at test.BuggyBank$Account.inputMoney(BuggyBank.java:165)
24         at test.BuggyBank.sendMoney(BuggyBank.java:311)
25         at test.BuggyBank$Window.run(BuggyBank.java:401)
26
27 thread test.BuggyBank$Window:{id:2,name:Thread-2,status:RUNNING,priority:5,isDae
28 mon:false,lockCount:0,suspendCount:0}
29   call stack:
30         at test.BuggyBank$Account.inputMoney(BuggyBank.java:165)
31         at test.BuggyBank.inputMoney(BuggyBank.java:294)
32         at test.BuggyBank$Window.run(BuggyBank.java:375)
33
34
35 ============================================================ results
36 error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.AssertionError
37 : input cause race conditi..."
38
39 ============================================================ statistics
40 elapsed time:       00:00:01
41 states:             new=231,visited=81,backtracked=216,end=2
42 search:             maxDepth=147,constraints=0
43 choice generators:  thread=230 (signal=1,lock=15,sharedRef=137,threadApi=4,resch
44 edule=73), data=0
45 heap:               new=2529,released=1779,maxLive=988,gcCycles=301
46 instructions:       54105
47 max memory:         75MB
48 loaded code:        classes=87,methods=1669
49
50 ============================================================ search finished: 17/01/21
51  0:02
```

リスト 5.3: assert 文を加えて JPF にかけた結果

ここで入金操作をロックしても入金操作が競合状態を起こすので, 手作業によるデバッグの場合と同様に入金操作の assert 文を一旦停止させて JPF にかける. すると引き出し操作が競合していることがわかるため引き出し操作をロックする.

入金と引き出しの両方をロックして JPF にかけるとリスト 5.4 の通りとなり, 出力結果であるリスト 5.4 の 2 行目から 8 行目と 11 行目のメッセージより並行性バグが取り除かれたことが確認できる.

```
1 0:03
2 server: 2000
3 criant: 2000
4 server: 2000
5 (中略)
6 criant: 2000
7 server: 2000
```

```
criant: 2000

====================================================== results
no errors detected

====================================================== statistics
elapsed time:       00:00:21
states:             new=33106,visited=52743,backtracked=85849,end=8
search:             maxDepth=185,constraints=0
choice generators:  thread=33106 (signal=129,lock=5268,sharedRef=24453,threadApi
=278,reschedule=2978), data=0
heap:               new=250543,released=63671,maxLive=988,gcCycles=82163
instructions:       5898105
max memory:         296MB
loaded code:        classes=78,methods=1609

====================================================== search finished: 17/01/21
 0:04
```

リスト 5.4: 入金および引き出し操作をロックして JPF にかけた結果

手作業によるデバッグとの相違点は実行に JPF を用いるか否かだが, 手作業の場合と異なり再現性に関わらず確実に並行性バグとその原因を発見することが確認できた. また並行性バグが修正されたことも JPF によって確認することができた.

5.3　手動で生成したプログラムのデバッグからわかること

5.3.1　BuggyBank

表 5.1: デバッグ方法ごとの結果: BuggyBank

バグ入りプログラムの種類	手作業によるデバッグ	JPF を用いたデバッグ
同期を全くとらない	assert 文を用いてデバッグが可能	assert 文を用いてデバッグが可能
口座の作成のみロックする	assert 文を用いてデバッグが可能	assert 文を用いてデバッグが可能
デッドロックを含む	IDE のデバッグ機能を用いてデバッグが可能	JPF の機能のみでデバッグ可能

手作業でも JPF でもデバッグが可能だが, JPF を用いることで再現性によらない確実なデバッグが可能だった (表 5.1). また JPF を用いることで並行性バグが確実に修正されているかどうかを確認することができた.

JPF がこのサイズのプログラムにおいて実用可能な速度で動作することが確認できた.

ただし BuggyBank のような銀行口座モデルのプログラムはマルチスレッドの例としてよく用いられるためあえて新たに作るバグ入りプログラムとしては不適切である可能性がある.

5.3.2　MapReduce モデルに基づくプログラム

JPF でのデバッグは四つの数字を数えるプログラムでも動作が停止せず, またその間バグを発見することもできなかった. . よって実行性能の問題から JPF を用いてデバッグすることが困難であることがわかった. このことにより JPF の実行性能上の問題がこのバグ入りプログラムで証明できることがわかった.

また SocketException 時の競合状態, および計算用クライアント終了時のデータ競合によるタスクの喪失は

表 5.2: デバッグ方法ごとの結果: MapReduce モデル

バグ入りプログラムの種類	手作業によるデバッグ
SocketException の 発生時に起こる競合状態	SocketException を何度も起こす テストにより発見, デバッグ可能
計算用クライアント終了時の データ競合によるタスクの喪失	異常な入力を行うテストの 繰り返しにより発見, デバッグ可能
計算用クライアントの取得と タスクの送信の間のデータ競合	データを送るクライアントと計算用クライアントの 起動する順序により発覚, デバッグ可能

正しいプログラムにおける通信に当たる部分やクライアントからの入力などが正常に動作している間は並行性バグの可能性を含まないため JPF のように網羅的に検査する場合, SocketException や異常な入力を行うテストなどを行う必要がある.

手作業によるデバッグの結果を表 5.2 にまとめた.

SocketException 発生時の競合状態は通信の切断など例外が発生した場合にのみ発生するため BuggyBank のバグに比べて発見が難しい. またデバッグのために SocketException を故意に発生させてテストする必要がある.

計算用クライアント終了時のデータ競合によるタスクの喪失は正常に計算用クライアントが動作している間は発見できないので故意に異常な入力を行うなどのテストが必要となる.

計算用クライアントの取得とタスクの送信の間のデータ競合では起動する順序により発覚するため再現性が高く, あえて JPF を用いずとも再現できる.

5.4　自動生成したバグ入りプログラムのデバッグからわかること

5.4.1　正しいプログラムからの生成

表 5.3: デバッグ方法ごとの結果: 自動生成したバグ入りプログラム

バグ入りプログラムの種類	手作業によるデバッグ
口座作成の競合状態	assert 文を用いてデバッグが可能
入金, 引き出しの競合状態	assert 文を用いてデバッグが可能
口座の操作中の口座の削除やパスワード変更禁止処理の競合状態	assert 文を用いてデバッグが可能

JPF でのデバッグはプログラムを口座の作成と送金のみに絞って 6 時間以上稼働しても停止せず, またその間バグを発見することもできなかった. よって MapReduce モデルに基づくバグ入りプログラムの場合と同様に実行性能の問題から JPF を用いてデバッグすることが困難であることがわかった. このことにより JPF の実行性能上の問題がこのバグ入りプログラムで証明できることがわかった.

手作業によるデバッグの結果を表 5.3 にまとめた.

ランダムに命令を送信するクライアントによるデバッグで入金, 引き出し操作の競合状態を発見するためにはスレッドを 2, 各スレッドが持つ口座を 1 にして送金と送金先の入金, 引き出しが同時に行われやすくし, 各

スレッドが 1000 万回の操作を行うようにしたテストでようやく発見できた.

手作業によるデバッグでこうした再現性の低いバグを発見するには細かに単体テストを作り根気よくデバッグを行う必要がある.

また今回は正しいプログラムから作ったバグ入りプログラムが並行性でないバグを含まないよう作ってあるが, これを他のプログラムに適用する場合ロックされていないインスタンスの wait メソッドなど非並行性バグを含む可能性や並行性バグを含まない可能性が考えられる.

この方法で作られたバグ入りプログラムはデータ競合のバグが存在しない. 関連研究 [2] よりおよそ 40% を占めるデータ競合が存在しないためデータ競合のためのデバッグツールのテストやベンチマーク, およびデータ競合のデバッグ手法を学ぶためのサンプルとしては不適切である.

5.4.2　自動生成で作成できなかったバグからわかること

表 5.4: デバッグ方法ごとの結果: データ競合および区間を変えることでできるバグ入りプログラム

バグ入りプログラムの種類	手作業によるデバッグ
口座の削除タスク受付後の送金タスクの受け付け	print 文を用いてデバッグ可能
送金タスク終了前の口座の削除タスク実行	再現性が低く困難, print 文を用いてデバッグ可能
ロックの区間を変更することによる競合状態	再現性が非常に低く困難

正しいプログラムは自動生成の場合と同じであるため JPF でのデバッグは実行性能上の問題でできなかった.

手作業によるデバッグの結果を表 5.4 にまとめた.

口座の削除タスク受付後の送金タスクの受け付けについてスレッド数を 50, 各スレッドの持つ口座の最大数を 5, 各スレッドの送る命令の回数を 1000 回にするとバグが顕在化した.

手作業でデバッグを行う場合, エラーメッセージから送金先の口座が失われていることがわかる. print 文を用いて各タスクが ExecutorService に送られた時刻を調べることでタスクの前後関係から原因が判明する. したがって print 文を用いたデバッグ手法が考えられる.

送金タスク終了前の口座の削除タスク実行について, スレッド数を 2, 各スレッドの持つ口座の最大数を 2, 各スレッドの送る命令の回数を 1000 万回にしてランダムに命令を行うテストを試行してもバグは顕在化しなかった. 同テストにおいて命令を口座の作成と送金, 口座の削除のみに制限して実行したところバグが顕在化した.

手作業でデバッグを行う場合, エラーメッセージから送金先の口座が失われていることがわかるが, その原因はすぐにはわからない. print 文を用いて各タスクが ExecutorService に送られた時刻と実行された時刻を調べるとタスクの前後関係から原因が判明する. したがって print 文を用いたデバッグ手法が考えられる.

以上よりこのバグは再現性が低く, 発見と原因の特定が自動生成したバグ入りプログラムより困難である. したがって再現性が低いデータ競合のサンプルとしてデバッグ手法やデバッグツールの評価やベンチマークとして用いることができる.

ロックの区間を変更することによるバグについて, 送金と送金先の口座の削除の競合が同時に ExecutorService に指定した数だけ実行されることによって起こるバグであるため再現性が低く, そのためデバッグが困難であり, そもそも発覚しにくい. 実際にスレッド数を 21, 各スレッドの持つ口座の最大数を 2, 各スレッドの送る命令の回数を 100 万回にしてランダムに命令を行うテストを 10 回試行してもバグは顕在化しなかった.

同テストを ExecutorService の持つスレッドの最大数を 1 に制限し, スレッド数を 2 にすることで命令の回数を 1 万回にしてもバグが顕在化し, プログラムがフリーズした. この時 Eclipse のデバッグ機能を用いると送金と送金先の口座の削除が同時に実行されていて, 口座の削除が先に実行されていることがわかる. print 文を用いて ExecutorService へ各タスクが加えられた時刻を調べるとこのバグが顕在化した時は送金タスクが口座の削除タスクの後に加えられていることがわかる. よって送金タスクを ExecutorService に送るコードと口座の削除タスクを ExecutorService に送るコードを調べると原因が判明し, 修正ができる.

以上より区間の変更によって発生するバグは再現性の低いバグおよび限られた条件で顕在化するバグのサンプルとなり, デバッグ手法やデバッグツールの評価やベンチマークとして用いることができる.

第 6 章　結論

6.1　まとめ

手作業と自動生成の二つの方法でバグ入りプログラムを作成した. 手作業では競合状態およびデータ競合のバグ入りプログラムを作成することができた. 自動生成ではロックの不足による競合状態のバグ入りプログラムを作成できたが, ロックの区間を変更することによるバグ入りプログラムの自動生成は組み合わせの多さのために, データ競合のバグ入りプログラムの自動生成は順序を制御するコードの認識の難しさのために失敗した.

また, これらによって作成したバグ入りプログラムに対し並行性バグについて手作業と JPF を用いた方法によるデバッグを行い, 手作業によるデバッグは柔軟性があるが再現性の低いバグのデバッグが困難であること, JPF によるデバッグは確実性があるが大きいプログラムのデバッグができないことが確かめられた.

バグ入りプログラムに JPF を適用することによって JPF のベンチマークのサンプルにすることができた.

したがって作成したバグ入りプログラムをその他のデバッグツールのためのサンプルやベンチマークのために使用することが考えられる.

またバグ入りプログラムをマルチスレッドプログラミングのデバッグ方法を学ぶためのサンプルとして使用することが考えられる.

6.2　今後の課題

バグ入りプログラムの自動生成について, wait メソッドに対応しているロックを消去した場合, wait メソッドの実行時に対象のロックがないことによるエラーが発生する. これは並行性バグではない. そのためロックが wait メソッドの対象になっているかどうかを判定し, 対象になっている場合はロックを取り除かないようにすることで非並行性バグを取り除く必要がある.

ロックの対象となるクラスが同じものだけを組にしてその中だけでロックの有り無しを組み合わせるようにするなどで組み合わせの数をもっと減らすべきである. 組み合わせの数が減ることでより大きいプログラムからバグ入りプログラムを作成できるようになる.

また, 組み合わせの数が減ることでロックの区間を変更する方法によるバグ入りプログラムの作成も可能になる可能性がある. 第 5.4.2 項よりロックの区間を変更することで得られるバグもサンプルとして有用なので, この方法でバグを作成できるようになることはバグの自動生成によるサンプルがより有用になることを示す.

ただし並行性バグを含まない可能性があるので正しいプログラムのロックされている区間が並行性バグを含まないための最小区間であるとして, 区間の変更を元の区間より短くすることに限るなどの工夫が必要である.

また関連研究 [2] より並行性バグのおよそ 40% はデータ競合である. したがってデータ競合のバグ入りプログラムを自動生成できるようになることが望ましい.

データ競合のバグ入りプログラムを作るためにはマルチスレッドにおける実行順序を制限しているコードを取り除く必要がある.

例えば wait メソッドおよびそれに付随する try-catch 文を取り除くなどの方法が考えられる.

謝辞

本研究は, 電気通信大学大学 情報理工学部 情報・通信工学科 コンピュータサイエンスコースの寺田研究室において, 寺田 実准教授のご指導のもと行われました.

寺田 実准教授には, 研究の方針やアイデア, 卒業論文の書き方などについて様々な助力, 御指導を頂きました.

また, 寺田研究室の仲間である修士課程 2 年の鈴木 佑樹さん, 平田 吉久さん, 本田 裕人さん, 阿部 真之さん, 修士課程 1 年の安部 文紀さん, 山本 愛美さん, 渡邊 裕貴さん, 学部 4 年の村松 啓寛さん, 肥後 亮佑さん, 岡川 翔子さん, 佐々木 透さん, 下澤 一輝さんには研究や論文についてのアドバイスを頂き, 研究室での生活など様々な面でお世話になりました.

ここに感謝の意を表します.

参考文献

[1] Apache ソフトウェア財団. “Apache Hadoop”. https://issues.apache.org/jira/browse/HADOOP.

[2] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson, and Eduard Paul Enoiu. “A Study on Concurrency Bugs in an Open Source Software”. 12st International Conference on Open Source Systems, 2016.

[3] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. Sixth Symposium on Operating System Design and Implementation, 2004.

[4] Yaniv Eytani, Klaus Havelund, Scott D. Stoller, and Shmuel Ur. “toward a benchmark for multi-threaded testing tools”. *Concurrency and Computation: Practice & Experience - Parallel and Distributed Systems: Testing and Debugging Volume 19 Issue 3*, pp. 267–279, 2005.

[5] Brian Goetz, Doug Lea, Tim Peierls, Joshua Bloch, Joseph Bowbeer, and David Holmes. “Java 並行処理プログラミング”. Softbank Creative, 2006. 岩谷 宏 訳.

[6] 荒堀喜貴, 小宮常康, 多田好克. “マルチスレッド C プログラムのための高互換かつ高効率かつ高精度な競合検出法”. 第 53 回プログラミング・シンポジウム. 情報処理学会, 2012.

[7] 篠崎孝一, 太田弘, 早水公二, 星野光勇. “モデル検査のデバッグへの適用”. ソフトウェアテストシンポジウム 2006, 2006.

[8] 大囿忠親, 新谷虎松. “マルチエージェントシステムにおける並行プロセスデバッグのためのトレーサの実現”. 電子情報通信学会論文誌 DI Vol. J83DI No. 8, pp. 873–881. 電子情報通信学会, 2000.

[9] 北野翔一郎, 片山徹郎. “Java マルチスレッドプログラム向けの拡張ペトリネットを用いた実行の再現を利用したデバッグ支援ツールの試作”. 研究報告ソフトウェア工学（SE) 2014-SE-185 23 号, pp. 1–8. 情報処理学会, 2014.