Vol.2014-SE-185 No.23
2014/7/11

# 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

北野 翔一郎[1,a)]　片山 徹郎[1,b)]

**概要**：マルチスレッドのプログラムはその実行の非決定性のために，バグを発見した時の状況を再現することが難しい．そのため，バグを発見した時のプログラムの挙動を，正確に把握するための情報を得ることが難しく，バグの原因の特定が困難になってしまう．そこで本論文では，ペトリネットを用いて Java 言語で書かれたマルチスレッドのプログラムの動作の可視化を行い，マルチスレッドのプログラムに再現性を持たせることでバグの原因の発見を支援する手法を提案する．従来のペトリネットではマルチスレッドの複雑な挙動を表現するには不十分である．そのため，本論文ではペトリネットの拡張も行う．提案手法を実現したツールを試作し，提案手法とツールの有効性を検証した．検証実験を行った結果は，本ツールを使用すると，本ツールを使用しない場合の約 33.6%の時間で，バグの原因を特定できた．このことから，本提案手法およびそれを実現したツールは，マルチスレッドプログラムのデバッグ作業効率を向上させることに有効であることがわかる．

**キーワード**：マルチスレッドプログラム，再現性，ペトリネット，デバッグ

# Prototype of a debugging support tool using reproduction of the execution by extended Petri-net for Java multi-thread programs

Shoichiro Kitano[1,a)]　Tetsuro Katayama[1,b)]

***Abstract:*** In multi-threaded programs, it is difficult to reproduce the situation when existing bugs are discovered because execution of the multi-threaded programs is usually non-deterministic. Therefore, it is difficult to obtain the information for understanding the behavior of the program when bugs are discovered. And, to identify the cause of bags becomes difficult. This paper proposes a supporting method for debugging to reproduce Java multi-threaded programs by visualizing the behavior of the programs with Petri-net. Conventional Petri-net cannot enough express the complicated behavior of the multi-threaded programs. Therefore, we extend Petri-net. We have confirmed the effectiveness of our method by implementing a prototype of a debugging supporting tool based on our method. In experiment for confirmation, to use our tool could identify the cause of the bug in about 33.6% of the time it takes without our tool. This result shows that our method and tool can improve efficiency in debugging for the multi-threaded programs.

***Keywords:*** multi-threaded program, reproducibility, Petri-net, debugging

## 1. 初めに

近年，マルチコアの CPU が多く普及している．そのような資源を活かすために，マルチスレッドのプログラムに注目が集まっている．しかし，マルチスレッドのプログラムは，シングルスレッドのプログラムよりもバグを作りこみやすく，修正も困難である [1]．そのため，そのようなバグは確実に早期に発見する必要がある．バグの早期発見を行うためには，徹底的に単体テストを行う必要がある．しかし，単体テストをマルチスレッドのプログラムに対して

[1] 宮崎大学
University of Miyazaki
[a)] kitano@earth.cs.miyazaki-u.ac.jp
[b)] kat@cs.miyazaki-u.ac.jp

行った場合，処理するタスクの規模が小さいため，多くの場合たった 1 つインターリービングのみを実行するだけで終わってしまう．

単体テストでマルチスレッドのプログラムをテストする方法の 1 つとして，ソースコード内の任意の箇所に，スレッドを一時的に停止させる処理を挿入し，各スレッドが処理を行うタイミングをずらして実行する方法がある [2]．この方法を用いれば，単体テストの規模でも複数のインターリービングでプログラムをテストできるため，ある程度はバグの存在を発見しやすくすることができる．しかし，バグの存在が確認できたとしても，マルチスレッドのプログラムはその実行の非決定性のために，バグを発見した時と同じ状況を再現することが難しい．プログラムの挙動を，print 文などを使って把握するデバッグ手法 [3] を使ったとしても，バグの原因を特定するための情報を得ることが困難である．また，上記のようなソースコードに変更を加える作業は，非常に手間のかかる作業である．バグの原因が確認できるような箇所に，任意の処理を挿入できなかった場合，再び変更を行う時間と，バグが顕在化するまでにかかる時間によって，バグの原因発見は遅れることになる．加えて，正しく動作していた部分を壊してしまう可能性もあり，非常にリスクの高い行為だと言える．

そこで本研究では，Java 言語で記述したマルチスレッドのプログラムのデバッグ作業効率の向上を目的とし，ペトリネットを用いたマルチスレッドのプログラムに再現性を持たせるデバッグ支援手法を提案する．具体的には，各スレッドが行った処理など，プログラムの実行時に得られる情報をデータ化する．そのデータとペトリネットでモデル化したプログラムの実行経路を用いて，プログラムの実行時の挙動をシミュレーションする．これによって，マルチスレッドのプログラムに再現性を持たせる．さらに，本研究で提案するデバッグ支援手法を実現するツールの試作を行う．

今回試作したツールは，Java 言語で記述したマルチスレッドのプログラムのソースコードを入力とし，そのプログラムを複数のインターリービングで自動的にテストを行う．その時の実行経路をデータ化し，そのデータと入力したソースコードから自動生成したペトリネットを用いることによって，プログラムの実行を再現できる．

一方で，従来のペトリネットでは，マルチスレッドのプログラムの複雑な挙動を表現するには不十分である．そこで，本研究ではペトリネットの拡張を行い，本ツールが生成するペトリネットとして用いる．

本論文の構成について説明する．2 章では，提案手法について説明する．3 章では，今回試作したツールの使用方法とその動作を述べる．4 章では，本ツールついての考察を述べる．

## 2. 提案手法について

### 2.1 ペトリネットを用いたデバッグ支援手法

本研究で提案するデバッグ支援手法について述べる．本手法はマルチスレッドのプログラムをテストした時の挙動を，ペトリネットでシミュレーションする．これにより，バグが顕在化した時の状況を何度でも再現できるため，バグの原因発見の効率を向上できる．

以下に，その流れを示す．

(1) テスト実行時に得られる情報のデータ化
バグが顕在化した時の状況を再現するために，実行した処理，スレッドが処理を行ったタイミング，処理を行ったスレッドの ID，生成したスレッドの ID, ロックに使用したインスタンスの ID から成るデータを生成する．

(2) テスト対象の Java プログラムのソースコードを基にペトリネットを生成
プログラムの実行のシミュレーションを行うためのペトリネットを生成する．ソースコードからペトリネットへの変換方法については，次節で説明する．

(3) テスト時のプログラムの挙動を再現
上記の 2 つの過程で生成したデータとペトリネットを組み合わせることによって，マルチスレッドのプログラムの実行を再現する．

### 2.2 Java プログラムからペトリネットへの変換規則

本研究では，Java プログラムのソースコードをペトリネットに変換する．表 1 に，その変換規則を示す．図 1 に，ソースコードの例を示し，図 2 に，図 1 のソースコードから変換したペトリネットを示す．この節では，Java プログラムからペトリネットへの変換規則について，図 1 のソースコードを使って説明する．紙面の都合上，while 文と synchronized ブロックの変換についてのみ説明する．

#### 2.2.1 while 文の変換

while 文の変換について説明する．図 2 の実線の枠で囲った部分に，変換後の while 文を示す．

‘‘while 文の前の文のトランジション’’ の変換規則に従い，statementM2 の true トランジションと false トランジションを作成する．statementM2 のプレースと結合している ‘‘ex is true’’ と ‘‘ex is false’’ がそれぞれに該当する．‘‘ex is true’’ の発火はループの実行を，‘‘ex is false’’ の発火はループを実行しなかったことを，それぞれ表現している．while 文内の処理は ‘‘while 文内の処理の開始’’ の変換規則に従い，in_while プレースとトランジションを作成する．‘‘文’’ と ‘‘while 文の最後の文のトランジション’’ の変換規則に従い，statementM3 のプレースと，true トランジションと false トランジションを

表 1　ソースコードからペトリネットへの変換規則
**Table 1**　A rule for conversion from source code to Petri-net.

| 変換する要素 | 変換結果 |
|---|---|
| 文 | プレースとトランジション |
| メソッドの開始 | プレースとトランジション |
| synchronized ブロックの開始 | プレース（in_sync プレース）とトランジション |
| while 文内の処理の開始 | プレース（in_while プレース）とトランジション |
| while 文の前の文のトランジション | in_while プレースに遷移するためのトランジション（true トランジション）と while 文の次の文に遷移するためのトランジション（false トランジション） |
| while 文の最後の文のトランジション | true トランジションと false トランジション |
| wait() メソッドの呼び出し | 自身を表すプレースとトランジションおよびロックの再取得を表すプレース（relock プレース）とそれに対応したトランジション |
| synchronized 予約語 | プレース（synchronized プレース） |
| 文から文への遷移 | プレースと対応するトランジションに向かうアークおよびトランジションから次のステートメントのプレースへ向かうアーク |
| ロックの取得 | synchronized プレースから in_sync プレースへのトランジションに向かうアーク |
| ロックの解放 | synchronized ブロックの最後のステートメントに対応するトランジションから synchronized プレースに向かうアーク |
| wait() メソッドによるロックの解放 | wait() メソッドを表すプレースへのトランジションから synchronized プレースに向かうアーク |
| ロックの再取得 | synchronized プレースから wait() メソッドのトランジションに向かうアーク |
| スレッド | トークン（スレッドトークン） |
| ロックに使用するインスタンス | トークン（ロックトークン） |
| java.lang.Thread クラスを継承したクラスのコンストラクタ呼び出し | コンストラクタ呼び出しのトランジションからそのクラスの run() メソッドの開始を表すプレースに向かうアーク |
| スレッドの生成 | java.lang.Thread クラスを継承したクラスのコンストラクタ呼び出しのトランジションの発火 |

```
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 1　ソースコードの例
**Fig. 1**　An example of source code.

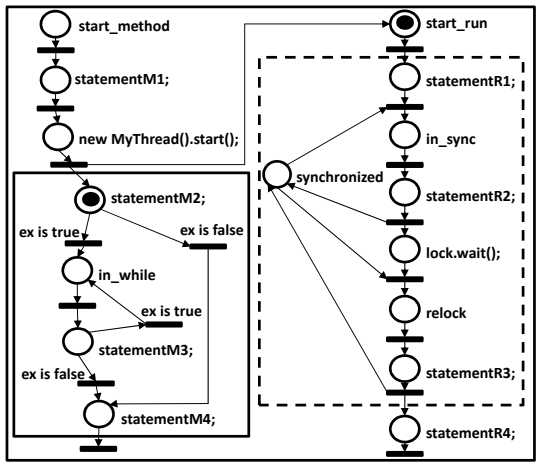


図 2　ペトリネットに変換した図 1 のソースコード
**Fig. 2**　Petri-net converted from source code of Fig.1.

作成する．statementM3 のプレースと結合している‘‘ex is true’’と‘‘ex is false’’がそれぞれに該当する．‘‘ex is true’’の発火はループの継続を，‘‘ex is false’’の発火はループの終了を，それぞれ表現している．‘‘文から文への遷移’’の変換規則に従い，全てのプレースとトランジションをアークで結合する．

以上で，while 文のペトリネットによるモデルが完成する．

### 2.2.2　synchronized ブロックの変換

synchronized ブロックの変換について説明する．図 2 の破線の枠で囲った部分に，変換後の synchronized ブロックを示す．

‘‘synchronized 予約語’’の変換規則に従い，synchronized プレースを作成する．ロックしていないロックトークンは，全てこのプレース上に設置する．‘‘synchronized ブロックの開始’’の変換規則に従い，in_sync プレースとトランジションを作成する．ブロック内の処理については，‘‘文’’と‘‘文から文への遷移’’の変換規則に従い，全てのプレースとトランジションを作成し，アークで結合する．‘‘ロックの取得’’と‘‘ロックの解放’’の変換規則に従い，statementR1 と statementR3 のトランジションをそれぞれ synchronized プレースと結合する．この変換に

より，synchronized ブロック内の処理に対する排他制御を表現する．

以上で，synchronized ブロックのペトリネットによるモデルが完成する．

### 2.3 ペトリネットの拡張

従来のペトリネットでは記述できる情報が少ないため，マルチスレッドのプログラムの挙動を正確にモデル化すると複雑なモデルになってしまう．この問題を解決するために，我々はペトリネットの拡張を行い，マルチスレッドのプログラムの挙動を理解しやすいものにする．この節では，具体的な問題点とその解決のための拡張方法について説明する．

#### 2.3.1 同一のパス上のトークンの識別

従来のペトリネットでは，同様の実行経路を通るトークンが複数存在するとき，それらを識別することが困難になってしまう．

そこで各トークンにユニークな ID を割り振り，その情報をラベルとしてトークンに付加する．この拡張によりトークンの識別が容易になる．

#### 2.3.2 トークンの役割の識別

従来のペトリネットでは，スレッドトークンと，ロックトークンを外見から識別することは困難である．

そこで図 3 に示すように，スレッドトークンを内側を塗りつぶしていない円，ロックトークンを内側を塗りつぶした円でそれぞれ表現することで，トークンの役割を外見から識別できるようにする．この拡張によって，各トークンの役割が直感的に理解できるため，識別が容易になる．

#### 2.3.3 ロックする対象の識別

本研究では，スレッドのロック取得を表現するために，スレッドトークンとロックトークンを同時に消費して発火するトランジションが存在する．このトランジションが発火するために必要なトークンの組み合わせは，シミュレーション時にすでに決定している．しかし，ロックトークンが複数存在する場合，トランジションが発火するために必要なトークンの組み合わせは，シミュレーションを進めていく過程でしか把握できない．この状況では，あるスレッドはどのスレッドが原因でロックの取得待ち状態に陥っているのか，といったようなデバッグにおいて有益な情報を得ることが困難になってしまう．

そこで図 4 に示すように，スレッドを表すトークンに，取得するロックの ID をラベルとして記述する拡張を行う．この拡張により，常にロックの対象を確認することができ，デバッグの効率が向上する．

#### 2.3.4 ロック状態の可視化

上述した現時点での我々のペトリネットでは，スレッドトークンがロックトークンを取得した場合，ロックを解放するまで，そのロックトークンはペトリネット上から完全に消えてしまう．複数のスレッドの状態を見比べながら，どのスレッドがどのロックを取得しているのかを常に意識しておくことは，非常に困難な作業である．そのため，デバッグにおいて有益な情報を見逃してしまう可能性がある．

そこで図 5 に示すように，スレッドを表すトークンに取得したロックを記述する拡張を行う．この拡張により，ロックを表すトークンの状況を，常に視覚的に確認できるようになるため，デバッグの効率が向上する．

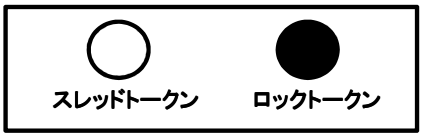


図 3 トークンの役割の表現
**Fig. 3** The expression of roles of tokens.

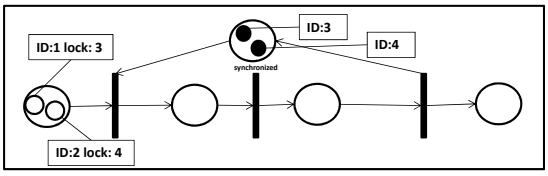


図 4 ロック対象の ID 情報を付加したスレッドトークン
**Fig. 4** Thread tokens with ID of locked tokens.

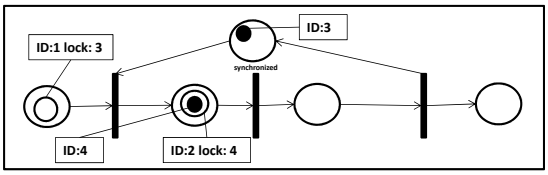


図 5 ロック状態の可視化
**Fig. 5** Visualization of locked tokens.

## 3. 支援ツールの試作

本研究では，提案手法を実現した支援ツールを試作した．この章では，本ツールの機能と本ツールにおけるペトリネットの表現方法について説明する．適用例を使った詳細なツールの使用方法を示すとともに，本ツールが正しく動作することを示す．

### 3.1 マルチスレッドのプログラムの自動テスト機能

本ツールのテスト機能は，Java 言語で記述したマルチスレッドのプログラムのソースコードを入力とし，そのプログラムを複数のインターリービングでテストする．この機能は一度実行すると連続で複数回テストを行う．現在何回目のテストを行っているのかをユーザに知らせるために，各テストの開始と終了を出力する．複数回のテストを全て完了すると，テスト機能を終了する．

1 回のテストを実行中に 2 秒経過した場合は，プログラムが異常な状態になったと判断し，テスト機能を終了する．これは，デッドロックなどの並列処理特有のエラーによっ

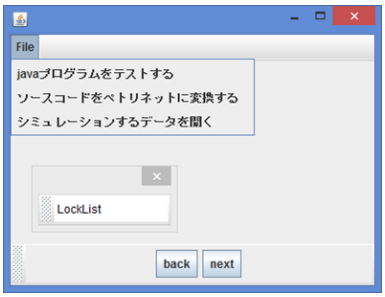


図 6　ツールの外観
**Fig. 6**　An overview of our tool.

て，テストが終了できない状態を防いでいる．ここで，2 秒という時間は，単体テストの規模でプログラムを実行した場合にかかる処理時間としては非常に長い．我々はこの時間を過ぎても終了しないテストは，プログラムが何らかの理由で処理が進まなくなっているものと判断し，タイムアウトする時間として 2 秒を設定した．テストが異常な状態で終了した場合は，終了時に動作していたスレッドの状態と生成したスレッドの数を出力し，各スレッドが行った処理など，テスト実行時に得られる情報をデータとしてファイルに保存する．

### 3.2　Java プログラムのソースコードからペトリネットへの自動変換機能

本ツールは，Java プログラムのソースコードを入力とし，2.2 節で示した変換規則に従って，ソースコードをペトリネットに自動変換する．変換してできたペトリネットはツールの画面上に表示する．

### 3.3　ペトリネットを用いたシミュレーション機能

自動テスト機能で得られたファイルと，表示したペトリネットを使い，テスト時の状況をシミュレーションする．

### 3.4　ツール上のペトリネットの表現方法

本ツールでは，ペトリネット上の情報を見やすくするために，一部の要素に対して表現方法の変更を行った．具体的には，各トークンに割り振った ID はトークンの色で表現した．各スレッドがロックするインスタンスの情報はビューに表示する．このビューは，シミュレーション開始時に全てのスレッドのロック対象を表示する．

通常のトランジションとアークを黒で表示するのに対して，ロックの取得と解放を表現するトランジションとアークは，ロックの取得の場合は緑，ロックの解放の場合は赤でそれぞれ表示する．また，通常のプレースを黒で表示するのに対して，synchronized のプレースは赤で表示する．

### 3.5　適用例を用いたツールの説明

適用例を使って，本ツールの具体的な使用方法と，ツールが正しく動作することを検証する．図 6 に，本ツールの外観を示す．図の左上の File の項目から，本ツールに実装した機能を呼び出すことができる．各機能がどの項目に対応しているのかについては，ツールの使用方法を説明する過程で示す．図の左下の ‘‘LockList’’ と表示しているビューは，スレッドトークンが取得するロックトークンの ID を表示する．図の一番下にあるツールバー上の 2 つのボタンによって，シミュレーション時にペトリネットの状態を操作できる．

```
public class NumberPrinter {
  private Object lock = new Object();
  private boolean isNotPrintedOne = true, isNotPrintedTwo = true;
  private class OnePrinter extends Thread {
    public void run() {
      synchronized (lock) {
      System.out.println(1);
      lock.notifyAll(); }
      isNotPrintedOne = false; } }
  private class TwoPrinter extends Thread {
    public void run() {
      try { synchronized (lock) {
            while (isNotPrintedOne) lock.wait();
            System.out.println(2);
            lock.notifyAll(); }
      isNotPrintedTwo = false;
      } catch (InterruptedException e) { }}}
  private class ThreePrinter extends Thread {
    public void run() {
      try { synchronized (lock) {
            while (isNotPrintedOne || isNotPrintedTwo) lock.wait();
            System.out.println(3); }
      } catch (InterruptedException e) { } } }
  public NumberPrinter() {
    new OnePrinter().start();
    new TwoPrinter().start();
    new ThreePrinter().start(); }
  public static void main(String[] args) {
    new NumberPrinter(); } }
```

図 7　適用するバグを含んだソースコードの例
**Fig. 7**　An example source code including a bug to apply to our tool.

図 7 に，対象のプログラムのソースコードを示す．このプログラムは，java.lang.Thread クラスを継承した 3 つのクラス，OnePrinter，TwoPrinter，ThreePrinter がそれぞれ別のスレッドとして，整数 1，2，3 を出力するプログラムである．期待する動作は，常に 1，2，3 の順番に 3 つの整数を表示することである．プログラムは我々が意図的にバグを埋め込んだため，必ずしも期待する動作にはならない．埋め込んだバグによって，このプログラムは，1 あるいは 1 と 2 しか出力しない場合がある．このバグの原因は，2 つのフラグ変数，isNotPrintedOne と isNotPrintedTwo に false を代入する処理を行う場所が不適切なことである．本来これらの代入処理は synchronized ブロック内にあるべきである．しかし，図 7 のソースコードでは，ブロックを抜けた後に代入処理を行っており，プログラムの動作によっては，TwoPrinter と ThreePrinter が生成するスレッドは，wait() メソッドによる停止状態から永久に復帰できなくなる．

#### 3.5.1　テストの実行

まず，対象のプログラムに対して本ツールを使ったテストを行う．テストを実行するには，File の項目の ‘‘Java

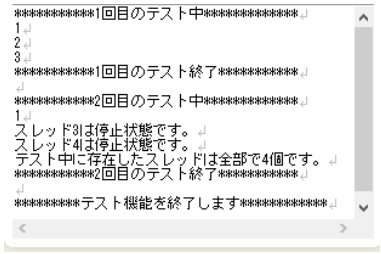


図 8　テストの出力結果
**Fig. 8**　An output of testing.

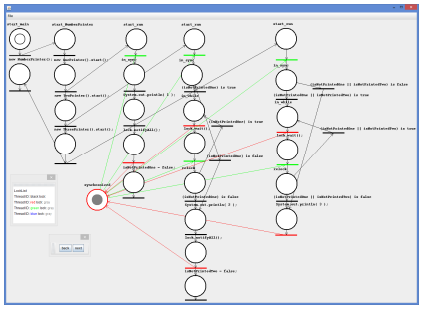

図 9　ツールが生成したペトリネット
**Fig. 9**　A Petri-net generated by our tool.


図 10　ロックするインスタンスを表示したビュー
**Fig. 10**　A view which has displayed locked instances.

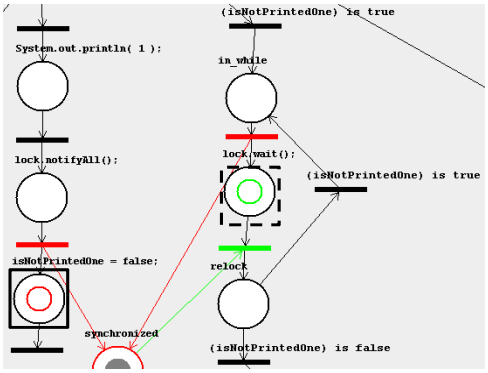


図 11　バグの原因
**Fig. 11**　A cause of a bug.

プログラムをテストする’’を選ぶ．ファイル選択画面が現れるので，そこからプログラムのソースコードを選択するとテストを実行する．

図 8 は，実際にテストを実行した時の出力結果である．図からわかるように，1 回目のテストでは，プログラムは期待する動作を行っている．2 回目のテストでは，我々の埋め込んだバグによって，プログラムは整数 1 のみを出力している．そして，4 つのスレッドのうち，2 つのスレッドが停止していることを出力している．このことから，2 回目のテストを実施中に 2 秒経過したため，テスト機能が終了したことがわかる．

### 3.5.2　ペトリネットの表示とシミュレーションの開始

前述したバグの原因を把握するために，対象のソースコードをペトリネットに変換する．File の項目の ‘‘ソースコードをペトリネットに変換する’’ を選択すると現れるファイル選択画面からソースコードを選択すると，ソースコードをペトリネットに変換する．次に，File の項目の ‘‘シミュレーションするデータを選ぶ’’ を選択すると現れるファイル選択画面から，テスト機能が生成したファイルを選択する．main メソッドの開始を表すプレース上と，synchronized プレース上に初期トークンが現れる．図 9 に，変換したペトリネットを表示した本ツールの画面を示す．このときに，ビューの更新も行う．図 10 に，更新後のビューを示す．

### 3.5.3　シミュレーション

初期トークンを設置後，ツールバーにある 2 つのボタンから，ペトリネットの状態を操作できる．next ボタンを押すと，ペトリネットは次の状態へ進む．back ボタンを押すと，ペトリネットは一つ前の状態に戻る．back ボタンを使うことで，一からシミュレーションをやり直すことなく，部分的にプログラムの挙動を再現できる．これらのボタンを使ってシミュレーションを繰り返しながら，バグの原因を特定する．

図 11 に，バグの原因を特定できるような状態にあるペトリネットの一部を示す．図 11 の実線の枠で囲った部分からは，OnePrinter が生成したスレッドが，フラグ変数 isNotPrintedOne に false を代入する処理を行っている状態であることがわかる．そして，図 11 の破線の枠で囲った部分から，TwoPrinter の生成したスレッドは，isNotPrintedOne に格納している値が true の時に実行する while ブロック内の wait() メソッドによる停止状態であることがわかる．しかし，停止中のスレッドが復帰するための処理は，OnePrinter が生成したスレッドが現在行っている処理の一つ前の処理のみである．このことから，フラグ変数への代入処理を行うべき箇所が不適切であるために，プログラムは整数 1 のみを出力しただけで停止したことがわかる．このように本ツールを使用することでバグの原因を特定できる．

また，ツールから特定できるバグの原因と，我々の埋め込んだバグの原因が一致していることから，バグが顕在化した時の動作を本ツールが正しく再現していることがわかる．

## 4.　考察

本研究では，被験者を用いた検証実験と，関連研究との比較を行い，本提案手法および試作したツールの有用性を

表 2　ツールを使わなかった場合の実験結果
**Table 2**　A result of experiment without our tool.

| | 原因特定にかかった時間 |
|---|---|
| 被験者 A | 15 分 56 秒 |
| 被験者 B | 14 分 15 秒 |
| 被験者 C | 26 分 37 秒 |
| 平均 | 18 分 56 秒 |

表 3　ツールを使った場合の実験結果
**Table 3**　A result of experiment using our tool.

| | 原因特定にかかった時間 |
|---|---|
| 被験者 D | 6 分 56 秒 |
| 被験者 E | 7 分 14 秒 |
| 被験者 F | 4 分 07 秒 |
| 平均 | 6 分 22 秒 |

示す．また，現在のツールの課題について考察する

### 4.1　検証実験

実験方法は，前章と同様のプログラムを対象として，プログラム内に存在するバグの原因の特定を，ツールを使う場合と使わない場合で，それぞれ別の被験者を用いて行った．実験の際，バグの原因特定までにかかった時間の計測を行った．被験者については Java 言語によるプログラミングの経験と，実践的なマルチスレッドプログラミングの経験のある者を対象に行った．被験者の人数は 6 人であり，ツールを使うグループと使わないグループを，それぞれ 3 人ずつに分けて実験を行った．

実験結果として，表 2 に，ツールを使用していない被験者，表 3 に，ツールを使用した被験者がそれぞれバグの原因の特定に要した時間を示す．それぞれの表からわかるように，本ツールを使用した場合，ツールを使用しない場合の約 33.6%の時間で，原因の特定が可能となる．

このような結果となった理由について述べる．ツールを使用していないグループの被験者は，実験を行っている間，何度もソースコードの書き換えと実行を繰り返しながら原因の特定を行っていた．以下に，この原因を示す．

- バグの内容を正しく把握できない
- プログラムに再現性が無い
- 単純にプログラムを実行しただけでは，各スレッドの実行順序を把握できない

しかし，スレッドの実行のタイミングをずらしても，バグが顕在化する保証はない．変更したプログラムから得られる情報が，原因の特定に十分なものでない場合は，再びプログラムに変更を加え，バグが顕在化するまで実行を繰り返す必要がある．これに対して，本ツールを使用したグループは，プログラムの実行は最初の 1 回のみであった．実行後はソースコードを一切変更することなく，ペトリネットによるシミュレーション機能のみで原因を特定できた．さらに，本ツールは，テストがタイムアウトした際に，その時の各スレッドの状態を出力する．この出力結果を基に，ある程度までならバグの原因を推測できる．加えて，シミュレーション機能は，プログラムの挙動の一部を繰り返し再現できるため，バグの原因が存在していそうな箇所を重点的に調査できる．以上のことから，本ツールを使用した被験者達は，本ツールを使用していない被験者達よりも速い時間でバグの原因が特定できた．

また，実験終了後にツールを使用した被験者達に，バグの原因の特定を行っている際に，有効に働いた本ツールの機能について回答してもらった．以下は，その一部である．

- 常にロックトークンが確認できること
- 各スレッドのロックする対象が表示してあったこと

これらの回答から，本研究で行ったペトリネットの拡張はバグの原因の特定に有効に機能したことがわかる．

以上のことから，本研究で提案したデバッグ支援手法およびそれを実現したツールは，デバッグの効率の向上に有効であることがわかる．

### 4.2　関連研究との比較

関連研究として，ペトリネットを用いたマルチスレッドのモデルの自動生成を行うことで，潜在的なバグを発見する研究がある [4], [5], [6]．この研究は静的解析によって得られたペトリネットモデルを解析して，バグの存在を発見する．しかし，そのバグの原因については発見できない．

また，複数のインターリービングでプログラムをテストすることで，潜在的なバグを発見できるツールに ConTest[7] がある．ConTest はマルチスレッドのプログラムに対して，徹底的な単体テストを実施できる．しかし，毎回ランダムにコンテキストスイッチを発生させてテストを行うため，バグが顕在化した時の状態の再現は困難である．そのため，バグの原因の発見はユーザの能力に依存する．

これに対して，本ツールは，プログラムを実際に実行した時の挙動をデータ化し，何度でも同じ状況を再現できる．そのため，バグの原因の特定が容易である．よって本ツールは，マルチスレッドのプログラムのバグを取り除くことに有効だと言える．

また，Erlang で記述したマルチプロセスのプログラムの単体テストツールに Concuerror がある [8]．ここで，Erlang においてのプロセスとは，スレッドと同義である．Concuerror は ConTest や本ツールと同様に，複数のインターリービングでテストを行う．Concuerror はソースコードを解析し，プログラムが動作しうるあらゆるインターリービングを，1 回のテスト機能の実行でテストする．バグが顕在化した場合には，詳細な実行時の情報を表示することができるため，デバッグの支援も行うことができる．

これに対して，本ツールのテスト機能は，場当たり的に複数のインターリービングでテストを実施している．その

ため，バグを発見できる確率は Concuerror に劣る．しかし，Concuerror が出力する実行時の情報は，パネル上にプロセスの挙動を一行ずつ順番に文字列で表示したものである．本ツールはペトリネットで実行時の情報を可視化している．そのため，プログラムの挙動を視覚的にとらえることができるため，Concuerror の表示する実行時の情報よりも理解が容易であると考えられる．よって，ペトリネットを用いてプログラムの挙動を表示する本ツールは，デバッグ作業の効率向上において有効だと言える．

### 4.3 現在のツールの課題

今回試作したツールの課題について考察する．

本ツールは Java 言語の構文の一部にしか対応していない．実際の開発において必要になる構文やライブラリ [9], [10] に対応させて，本ツールの実用性を向上させる必要がある．

テスト機能については，場当たり的に複数のインターリービングでテストを実施する．このテスト方法は，潜在するバグを確実に発見するのに十分なものではない．そのため，バグ発見の確率向上のための工夫が必要である．また，テストをタイムアウトするまでの時間として，1 回のテスト毎に 2 秒という時間を設定している．この時間は，経験に基づき，単体モジュールのテストにかかる時間として十分に長い，と判断して設定した時間であるため，適切なものであるとは言えない．適切な時間を設定するためには，さらなる調査が必要である．

本ツールが生成するペトリネットが巨大になる場合，ペトリネットから情報を得ることが困難になる．現在ペトリネットを見やすくする機能は，要素の移動と，描画範囲の切り替えのみである．文字情報の見せ方を工夫することや，表示しているペトリネットの要素の大きさの切り替え，任意のパスを折り畳む機能の実装を行うなど，ペトリネットの肥大化に対応する必要がある．

## 5. おわりに

本研究では，ペトリネットを用いたマルチスレッドのプログラムに再現性を持たせるデバッグ支援手法を提案した．これを実現するために，Java 言語で記述したソースコードからペトリネットへの変換規則を提案した．また，ソースコードから変換したペトリネットの複雑化を，ペトリネットを拡張することで回避し，プログラムの挙動を理解しやすいものにした．

本研究で提案したデバッグ支援手法を実現したツールを試作した．試作したツールは，Java 言語で記述したマルチスレッドのプログラムのソースコードを入力とし，自動テスト機能とペトリネットへの自動変換機能，シミュレーション機能を提供する．自動テスト機能は，入力したプログラムに対して複数のインターリービングでテストを実施する．テストがデッドロックなどの並列処理特有のエラーによってタイムアウトした場合は，テスト機能を終了し，実行した処理など，実行時に得られる情報をファイルに保存する．ペトリネットへの自動変換機能は，入力したソースコードをペトリネットに自動変換し表示する．シミュレーション機能は，テスト機能で得たファイルと表示したペトリネットを用いて，バグが顕在化した時の状況を再現することでデバッグ支援を行う．このツールを用いた実験の結果は，ツールを使用すると，ツールを使用しない場合の約 33.6%の時間で，バグの原因を特定できた．このことから，本提案手法およびそれを実現したツールは，デバッグ作業効率の向上において有効であることを示した．

以下に，今後の課題を示す．

- ペトリネットおよびツールの適応範囲の向上
- テスト機能の有用性の向上
- 巨大なペトリネットへの対応

### 参考文献

[1] Ousterhout, J.K.: Why Threads Are A Bad Idea, Usenix Annual Technical Conference. available from 〈http://www.softpanorama.org/People/Ousterhout/Threads/〉(1996).
[2] ConTest を使用したマルチスレッド・ユニットのテスト ― IBM，developerWorks，入手先 〈www.ibm.com/developerworks/jp/java/library/j-contest/〉(参照 2006-04-05)．
[3] Zeller, A.: Why Programs Fail 2nd Edition A Guide to Systematic Debugging, Elsevier / Morgan Kaufmann(2009). 中田秀基（監訳），今田昌宏，大岩尚宏，竹田香苗，宮原久美子，宗形紗織（訳）：デバッグの理論と実践―なぜプログラムはうまく動かないのか，O'Reilly Japan（2012）．
[4] Kavi, M.K., Moshtaghi, K. and Chen, D.: Modeling Multithreaded Applications Using Petri Nets. International Journal of Parallel Programming, Vol.30, No. 5, pp.353-371(2002).
[5] Govindarajan, R., Suciu, F. and Zuberek, W.M.: Timed Petrinet Models of Multithreaded Multiprocessor Architectures, IEEE Published in Petri Nets and Performance Models, pp.153-162(1997).
[6] Liao, H., Wang, Y., Cho, H., Stanley, J., Kelly, T., Lafortune S., Mahlke S., Reveliotis S.: Concurrency bugs in multithreaded software: modeling and analysis using Petri nets, Discrete Event Dynamic Systems Vol.23, Issue 2, pp.157-195(2013).
[7] ConTest - A Tool for Testing Multi-threaded Java Applications, ConTest - IBM Research, available from 〈https://www.research.ibm.com/haifa/projects/verification/contest/〉.
[8] Gotovos. A, Christakis, M. and Sagonas, K.: Test-Driven Development of Concurrent Programs using Concuerror, Erlang'11, pp.51-61 (2011).
[9] 結城浩：Java 言語で学ぶデザインパターン入門　マルチスレッド編，ソフトバンク クリエイティブ（2006）．
[10] Goetz, B., Peierls, T., Bowbeer, J., Holmes, D., Lea, D.: Java Concurrency in Practice, Addison-Wesley Professional(2006). 岩谷宏（訳）：Java 並行処理プログラミング ―その「基盤」と「最新 API」を究める―，ソフトバンク クリエイティブ（2006）．