ペトリネットを用いたJavaマルチスレッドプログラムの実行を再現することによるデバッグ支援手法の提案

北野　翔一郎[a)]・片山　徹郎[b)]

Proposal of a Supporting Method for Debugging to Reproduce the Execution of Java Multi-threaded Programs with Petri-Net

Shoichiro KITANO, Tetsuro KATAYAMA

Abstract

It is difficult to implement multi-threaded programs. One of the reason is that the behavior of each thread is non-deterministic. Also it is difficult to reproduce the situation in which an incident occurs. This paper proposes a supporting method for debugging to reproduce Java multi-threaded programs by visualizing the behavior of the programs with Petri-net. Here, it is difficult to express the behavior of multi-threaded programs by ordinal Petri-net. Therefore, We extend Petri-net to reproduce the execution of multi-thread programs to realize our proposal method. Moreover, we have confirmed the effectiveness of our method by implementing a tool which generates a Petri-net model from a multi-threaded program.

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

1.　はじめに

近年、マルチコアのCPUを採用するコンピュータが普及している。その資源を有効に利用するために、マルチスレッドのプログラムの需要が高まってきている。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と比べて、バグを作りこみやすい[1)]。また、そのバグの発見も困難であるため、多くのバグがソフトウェア開発プロセスの後半や、出荷後にユーザが使用した時などに顕在化し、修正することが困難になってしまう。

バグの発見、および、その修正を困難にする原因の一つとして、単体テストで十分なテストが行えないことがあげられる。単体テストは並列タスクの規模も小さく、多くの場合たった1つのインターリービングのみを実行するために、バグを発見することができない。

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1つの方法として、各スレッドの実行のタイミングをずらし、複数のインターリービングでプログラムを実行する方法がある[2)]。この方法を用いることで、マルチスレッドのプログラムの潜在的なバグを発見できる。しかし、このテスト方法でバグの存在が明らかになったとしても、マルチスレッドのプログラムは、バグの原因を発見することが難しい。この原因として、マルチスレッドのプログラムには再現性が保証されていないことがあげられる。複数のスレッドが別々に処理を行うマルチスレッドのプログラムは、処理の時間やタイミングが非決定的であり、以前と全く同じ状態を再現することが難しい。さらに、プログラムの動作は目に見えないため、どのスレッドがどのタイミングで動作して問題を引き起こしたかを把握することは困難であり、再現性のなさがこの作業をより困難なものにしている。

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の効率向上を目的とした、ペトリネットによる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手法の提案を行う。具体的には、プログラムの実行経路をデータ化し、そのデータをもとにペトリネットでプログラムの振る舞いをシミュレーションする。そして、マルチスレッドのプログラムに再現性を持たせることによって、バグの原因の発見の支援を行う。

ここで、従来のペトリネットでは記述できる情報が少ないため、並列に動作するスレッドの挙動を表現すると、複雑なモデルとなり、動作の理解が困難になってしまう。そこで我々は、Java言語で記述したマルチスレッドのプログラムの理解を促進させるために、ペトリネットの拡張も行う。

さらに、マルチスレッドのプログラムを複数のインターリービングで自動テストし、その実行経路をペトリネットでシミュレーションする支援ツールを試作する。このツールを用いて、実際にマルチスレッドのプログラムに提案手法を適用し、提案手法が正しいことを検証する。

a)情報システム工学専攻大学院生
b)情報システム工学科准教授

```
public class Example {
  static class ThreadExample extends Thread {
    public void run() {
      Object lock = new Object();
      synchronized (lock) {
        System.out.println("in synch");
      }
      System.out.println("lock release");
    }
  }

  public static void main(String[] args) {
    System.out.println("Hello");
    new ThreadExample().start();
    System.out.println("World");
  }
}
```

図 1. ソースコードの例 1.

2. ペトリネットを用いたデバッグ支援

2.1 デバッグ支援の手法

この節では、ペトリネットを用いたデバッグ支援の手法について述べる。具体的には、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、バグの原因発見の効率を向上する方法を提案する。

以下に、その手順を記述する。

1. テスト時の実行経路のデータの生成
   バグが顕在化した時の状況を再現するために、プログラムのテスト時の実行経路をデータ化する。このデータは、処理を実行したスレッドの ID、実行した処理、処理を行ったタイミング、生成されたスレッドの ID から成る。
2. テストする Java プログラムからペトリネットによるモデルを生成
   Java プログラムをもとにペトリネットによるモデルを生成する。具体的なモデル生成の方法は、次の 2.2 節で説明する。
3. 上記 2 つを使い、テスト時のプログラムの動作を再現する

2.2 Java プログラムのペトリネットによるモデル

本研究では以下に示す規則で、Java プログラムをペトリネットでモデル化する。図 1 にソースコードの例を示し、図 2 に、図 1 のソースコードをペトリネットでモデリングした例を示す。

- 1 つのステートメントは、1 つのプレースと 1 つのトランジションに変換
- synchronized 予約語は、1 つのプレースに変換
- メソッドの開始は、1 つのプレースと 1 つのトランジションに変換
- synchronized 予約語によるロック取得の処理は、1 つのプレースとトランジションに変換

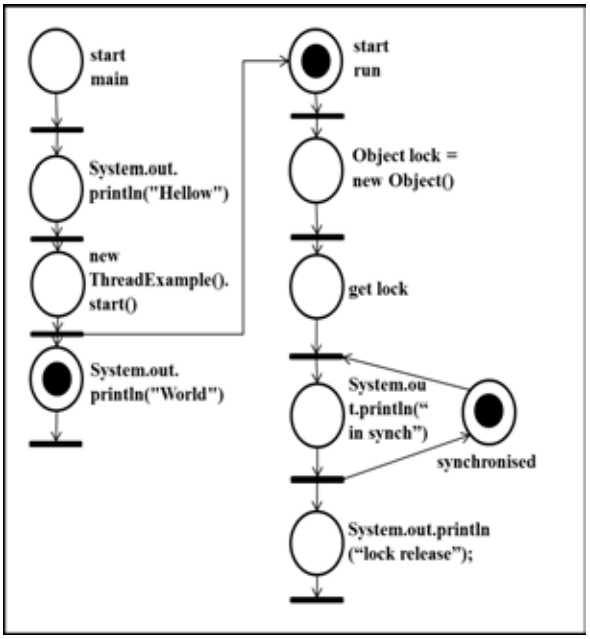

図 2. モデルの例.

- ステートメントのプレースは、次のステートメントへのトランジションに向かうアークで接続
- トランジションは、次のステートメントのプレースに向かうアークで接続
- java.lang.Thread クラスを継承したクラスのコンストラクタを呼び出しているステートメントのトランジションは、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で接続
- synchronized 予約語は、プレースをロック取得のトランジションへ向かうアークで接続
- synchronized ブロック内の最後のステートメントのトランジションは、synchronized 予約語のプレースに向かうアークで接続
- 1 つのスレッドは、1 つのトークンに変換
- 1 つのロックされるインスタンスは、1 つのトークンに変換

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、ペトリネットによるモデルを、マルチスレッドのプログラムの挙動が理解しやすいものにした。以下では、具体的な問題点とその解決方法としての拡張について説明する。

3.1 トークンに ID を付加

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。

図3.　色によるトークンの役割.

そこで各トークンにIDを割り振り、その情報をラベルとしてトークンに付加した。この拡張によりトークンの識別が容易になる。

3.2 色によるトークンの役割の識別

従来のペトリネットは、スレッドを表すトークンと、ロックされるインスタンスを表すトークンを識別することが困難である。

そこで図3に示すようにスレッドを表すを白い円、ロックされるインスタンスを表すトークンを黒い円で表現することで、それぞれのトークンの役割を明確にした。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。

3.3 ロックする対象の記述

現段階におけるペトリネットは複数のロックされるインスタンスが存在する場合や、1つのインスタンスのロック権の取得を複数のスレッドで行う場合に、非常に理解が困難になる。また、どのスレッドがどのインスタンスのロックを行うのかがわからないという問題点もある。

例として、図4はID1,3,4のスレッドはID10のインスタンス、ID2はID20のインスタンスのロック権を取得する場合を表している。しかし、このような情報は従来のペトリネット上に記述できない。さらに図5のように、ID1とID2のスレッドが遷移した場合に、ID3,4のスレッドは、どちらのスレッドによって待ち状態に陥っているのかがわからない。

そこで、図6に示すように、スレッドを表すトークンにロック対象のインスタンスのIDを「lock ロック対象のインスタンスのID」で表記したラベルを付加する拡張を行う。さらに、インスタンスをロックしている状態であることを表現するために、スレッドを表すトークン内にロックされたトークンを記述する拡張を行う。これらの拡張により、スレッドの待ち状態の原因などを直感的に理解出来るようになる。

4.　支援ツールの試作

本研究では、提案した手法を支援するツールの試作を行った。ここでは実装したツールについて説明する。

4.1 ツールの処理の流れ

以下に、本ツールの処理の流れを示す。

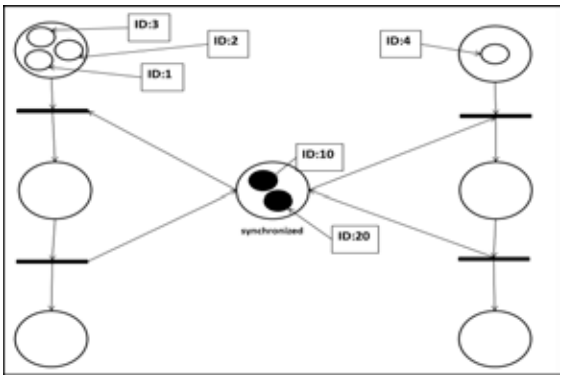

図 4.　ロックする対象の記述がないペトリネット.

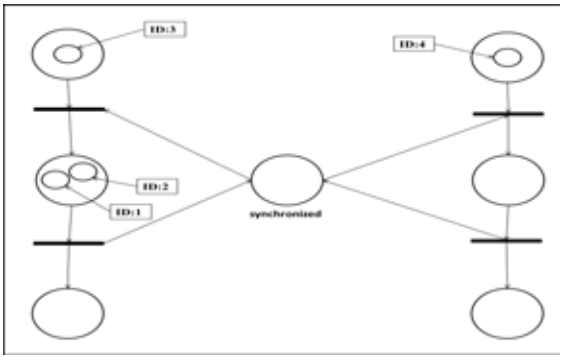

図 5.　待ち状態の原因が把握できない状態.

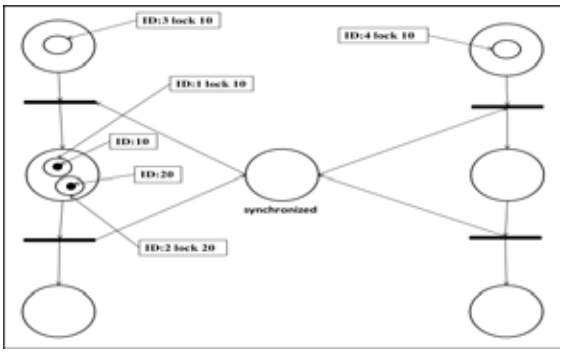

図 6.　ロックする対象を記述できるペトリネット.

1. テストするコードにプローブを挿入したテスト用コードを実行
2. 実行時に抽出した実行データをファイルに保存
3. テストするコードからペトリネットによるモデルを生成
4. 実行データのファイルとペトリネットを使って実行時のプログラムの挙動を再現

4.2 生成するペトリネット

本ツールが生成するペトリネットは、本論文で提案した拡張の一部のみしか適応していない。具体的には、ペトリネット上に存在する、スレッドを表すトークンの識別のためのIDのみ適応している。このIDはトークンの中心に表示している。

```
public class Example {

   private class ThreadExample extends Thread{

      public void run(){
         System.out.println("statement 1");
         System.out.println("statement 2");
      }

   }

   public Example(){
      new ThreadExample().start();
   }

   public static void main(String[] args){
      new Example();
   }
}
```

図 7. ソースコードの例 2.

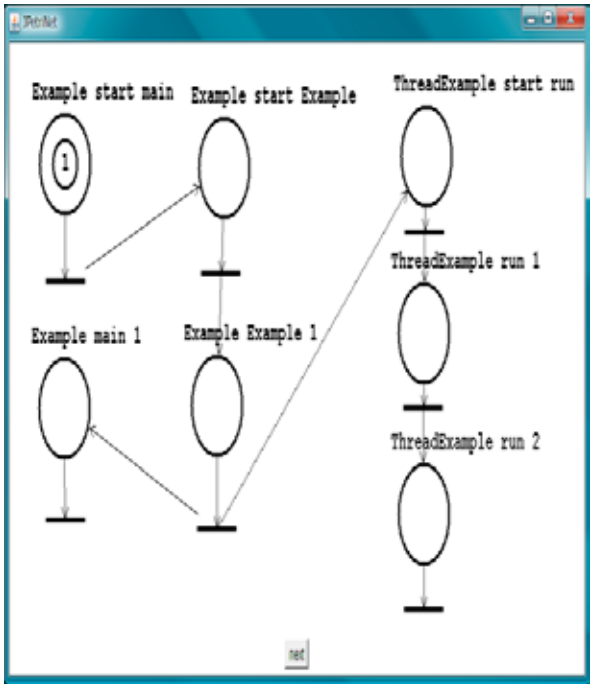

図 8. ツールが生成したペトリネット.

また、生成したプレースには、識別のためにラベルを付加している。このラベルは、そのプレースがステートメントを表している場合は「クラス名 メソッド名 ステートメントの番号」から成る文字列を表示しており、メソッドの開始を表している場合は「クラス名 start メソッド名」から成る文字列を表示している。

図 7 にツールに適用するソースコードの例を示し、図 8 に、試作したツールによって図 7 のソースコードから生成したモデルを示す。図 8 より、このツールが 2.2 節で述べた規則に従った通りのモデルを生成できていることがわかる。

```
public class NakedNamePrinter {

   private final String firstName;
   private final String surName;

   public NakedNamePrinter(String firstName, String surName) {
      this.firstName = firstName;
      this.surName = surName;
      new FirstNamePrinter().start();
      new SpacePrinter().start();
      new SurnamePrinter().start();
   }

   private class FirstNamePrinter extends Thread {
      public void run(){
         System.out.print(firstName);
      }
   }

   private class SpacePrinter extends Thread {
      public void run() {
         System.out.print(" ");
      }
   }

   private class SurnamePrinter extends Thread {
      public void run() {
         System.out.println(surName);
      }
   }

   public static void main(String[] args) {
      new NakedNamePrinter("Shoichiro", "Kitano");
   }
}
```

図 9. テスト対象のコード.

5. 検証

この章では、前述したツールを使い、提案したデバック支援方法の有用性を検証する。また、関連研究との比較を行う。

5.1 検証実験

適応例として、Java 言語で書かれた、同期処理を使用しないマルチスレッドのプログラムを用いる。図 9 に、このプログラムを示す。

このプログラムは java.lang. Thread を継承した 3 つのサブクラス FirstNamePrinter, SpacePrinter, SurnamePrinter がそれぞれ、文字列「”Shoichiro”」「” ”」「"Kitano”」を表示するプログラムである。期待される出力は「”Shoichiro Kitano”」である。しかし、前述したとおりこのプログラムには同期処理を実装していないため、期待する出力にならない場合がある。

上記プログラムをテストしたときの実行データを基に、期待出力と異なる出力を行ったときの状況を本ツールを使って再現し、提案手法がバグの原因の発見を支援できることを検証する。

5.2 実験結果

図 10 に、不具合が出た時の状況をペトリネットで再現したものを示す。

本来、このペトリネットは FirstNamePrinter run 1、SpacePrinter run 1、SurnamePrinter run 1 の順にプレースをマーキングするべきである。図 10 では、青の破線で囲んでいる SpacePrinter start run 上にトークンがあることから、緑の破線で囲んでいる SpacePrinter run 1 を実行していな

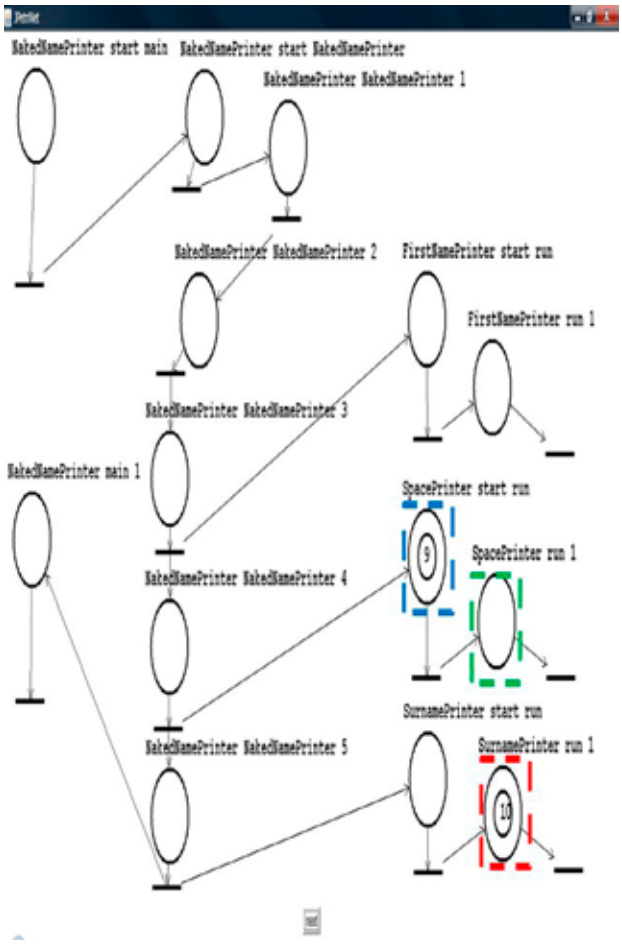

図 10. 不具合が出た状況の再現.

いことがわかる。また、赤の破線で囲んでいるSurnamePrinter run 1 上にトークンがあることから、このプレースが表わす処理を SpacePrinter run 1 より先に実行していることがわかる。このことからスレッドが期待する順序で処理を行っていないことが確認できる。これによりスレッド間の同期処理に不備が存在することに気付くことができる。

以上から、不具合が出た時の状況をペトリネットで再現したこと、またプログラムの挙動をグラフィカルに理解できることにより、バグの原因を容易に発見することができたと言える。よって、本提案手法が有効であることが確認できた。

5.3 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 [3,4,5]。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することで、バグが存在することを発見できる。しかし、そのバグの原因については、発見できない。

また、マルチスレッドのプログラムの単体テストを自動で行い、潜在的なバグを発見するツールがある [6]。このツールはランダムに複数のインターリービングでテストするので、不具合が出た時の状況を再現できない。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、本提案手法は、実際に実行された経路のデータファイルを基に、何度でも同じ状況を再現することができるため、バグの原因の発見が容易である。よって本提案手法は、マルチスレッドのプログラムの潜在的なバグを取り除くことに有効だと言える。

6. 結論

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、デバッグ支援の方法を提案した。また、マルチスレッドプログラムのペトリネットによるモデルの複雑化を、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方法についてツールを試作し、このツールを用いた実験を行い、その有用性を示した。これによって、マルチスレッドのプログラムのデバッグの効率化が期待できると考えられる。

以下に今後の課題を示す。

- ツールの対応できるソースコードの範囲の向上
  本研究で試作した支援ツールは、スレッドの生成および実行のみに対応しており、排他制御や同期処理など、マルチスレッドのプログラムの実装に必要な構文には、十分に対応していない。今後は対応可能なソースコードの範囲を向上することで、本ツールの有用性と検証の範囲を向上させる必要がある。

- 提案したペトリネットのモデリングの規則の検証
  本研究ではJava言語のペトリネットによるモデルが正しいモデルであることを検証し、行っていない。今後はモデルが正しいものであるかの検証し、現状のモデルの問題点を洗い出し、より有効なモデリング方法を検討する必要がある。

- 拡張したペトリネットの有効性の検証
  本研究では、ペトリネットの拡張の提案を行ったが、その有用性については検証していない。今後は拡張内容をツールに実装し、その有用性を検証する必要がある。

- マルチスレッドのプログラムをサポートする標準ライブラリのモデル化方法の提案およびツールへの実装
  Java言語ではスレッドの操作をサポートする特殊なライブラリが多く存在する [7,8]。そのようなライ

ブラリのモデル化を行い、ツールに実装することで、よりデバッグの支援の効果が期待できると考えられる。

- ペトリネット上のラベル情報の表現方法の変更
  現段階ではツールの生成したペトリネットのラベル情報は、デバッグ支援を行うために十分なものであるとは言えない。プレースのラベルをソースコード中のステートメントに対応したものにすることや、ラベルの情報を見やすくするためのユーザインターフェースを実装するなど、表現形式を変更する必要がある。

- ツールが生成したペトリネットの初期位置の自動生成機能の向上
  本ツールが生成したペトリネットの各要素は、画面の原点座標を初期位置として生成する。そのため、ユーザがモデルを直接整形する必要があり、プログラムの挙動の再現を行うまでに時間がかかるという問題がある。今後は、ユーザがモデルを見やすい位置にする機能を実装することで、整形にかかる時間をなくし、ツールの有用性を向上する必要がある。

- 支援ツールの有効性の検証
  本研究では、手法の有効性には言及したが、試作した支援ツールの有効性の検証を行っていない。今後は、上記の課題で述べたような、ツールに実装すべき要素を実装し、ツール有用性の検証を行う必要がある。

参考文献

1)J. K. Ousterhout: “Why Threads Are A Bad Idea (for most purposes)” Presentation given at the 1996 Usenix Annual Technical Conference, January 1996 http://www.softpanorama.org/People/Ousterhout/Threads/

2)“ConTest を使用したマルチスレッド・ユニットのテスト – IBM” www.ibm.com/developerworks/jp/java/library/j-contest/

3) Krishna M. Kavi, Alireza Moshtaghi, Deng-jyi Chen(2002): “Modeling Multithreaded Applications Using Petri Nets” International Journal of Parallel Programming, Vol.30, No. 5, pp353-371 October 2002.

4)Govindarajan, R, Suciu, F., Zuberek, W.M: “Timed Petrinet Models of Multithreaded Multiprocessor Architectures” IEEE Published in Petri Nets and Performance Models, 1997 Proceedings of the Seventh International Workshop on, pp153-162, June 1997.

5) Hongwei Liao, Yin Wang, Hyoun Kyu Cho, Jason Stanley, Terence Kelly, Stéphane Lafortune, Scott Mahlke, Spyros Reveliotis: “Concurrency bugs in multithreaded software: modeling and analysis using Petri nets” Discrete Event Dynamic Systems Vol.23, Issue 2, pp157-195, June 2013.

6) “ConTest - A Tool for Testing Multi-threaded Java Applications” https://www.research.ibm.com/haifa/projects/verification/concont/

7) 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 クリエイティブ株式会社(2006)

8) Brian Goetz, Tim Peierls,Joshua Bowbeer, David Holmes, Doug Lea : “Java Concurrency in Practice” ソフトバンク クリエイティブ株式会社(2006)