ペトリネットを用いたソースコードの規模の正規化に関する研究

情報システム専攻　中村 翔吾　　　指導教員：辻　孝吉

1　研究背景・目的

現在、ソフトウェア開発では見積りに人月という単位が使われているが、人月による見積りは様々な問題点を抱えている。このことから、人月を用いた見積りに代わる新しい見積り方法が必要とされている[1]。まず、文献[2]では「ソフトウェアエンジニア評価方式の見積り法」を提案した。この見積もり方式を用いて評価を行うためにはエンジニアを正確に、短い期間で評価しなければならない。そのための評価値の一つとしてソースコードの行数に着目した。しかし、単なる行数で評価することは空行の挿入などの理由から誤差が生じやすいので危険である。そのため、正規化されたソースコードの規模を求めることが必要である。正規化されたソースコードの規模を求める方法を示し、その後、求めたソースコードの正規化された規模を用いてエンジニアを実際に評価する手法を示す。

2　ソースコードの縮約

プログラミングの目標の一つとして、できるだけ短くソースコードを書くことが挙げられる。同じ処理をするプログラムでも、ソースコードを短く書くことが出来ればより短い時間でプログラムを書けるようになり、修正や拡張も容易になる。

プログラムを短く書くための基本的な方法が 2 つある。1 つは while 文などの制御構造をうまく使うこと、もう 1 つはメソッドを利用することである。同じ処理の繰り返しを発見し、これら 2 つの方法を適切な箇所で利用することがソースコードを短く書くために必要である。

3　ソースコードのペトリネットによるモデル化

今回は Java のソースコードをモデル化する。ソースコードをペトリネットでモデル化するにあたって、まず各々の概念を対応付けする必要がある。ソースコードの概念をカラーペトリネットの概念に対応付けた場合、以下の表 1 のようになる。

表 1　ソースコード上の概念とカラーペトリネット上の概念の対応付け

ソースコード上の概念	カラーペトリネットの概念
クラス	CPN(Color Petri Net)の集合
メソッド	CPN
コンストラクタ	CPN
インスタンス	クラス名と識別子のカラートークン
変数	プレース
変数の値	カラートークン
命令	トランジション
プログラムの進路	アーク

ソースコード縮約のために特に重要なメソッド呼び出しのペトリネットによるモデル化アルゴリズムを示す。

《メソッド呼び出しの変換アルゴリズム》

(Step1) トランジション ti にラベル付けされた命令がメソッド呼び出しを表すものだった場合、トランジション ti から呼び出すメソッドの起動プレース pi にアークを繋ぐ。

(Step2) 制御プレース pj を置き、トランジション ti から制御プレース pj にもアークを繋ぐ。

(Step3) メソッド呼び出し終了を表す[end entry]とラベル付けされたトランジション tj を置き pj から tj へアークを繋ぐ。

(Step4) 呼び出したメソッドの動作終了を表す最後の出力プレース pA から、トランジション tj へアークを繋ぐ。

4　正規化されたソースコードの規模

エンジニアを評価する時の指標として、ソースコードの行数を利用したいが、単に長いソースコードを書いたエンジニアを優れていると評価することは正確な評価と言えない。これまで示した制御構造やメソッドを上手く扱い、短くまとまったソースコードを作る方が優れている。しかし、短くても、何をやっているのか分かりにくいソースコード(for 文の入れ子構造等)も保守の観点から良くない。そこで、それらを加味した「正規化された規模」でエンジニアを評価する。正規化された規模を求める手順を以下に示す。

《正規化されたソースコードの規模》
同じ動作をする全てのソースコードをペトリネットでモデル化した時、トランジションの合計数で最小のもの(メソッド呼び出しの終了を表す[end entry]を除く)

同じ動作をするプログラムの場合、よりトランジションの数が少ないほうを優れているものとして扱う。トランジションの数が同じ場合、よりソースコードが視認性に優れていたり、わかりやすかったりする方を優れているものとする。

5　エンジニア評価のシミュレーション

実際に 2 つのソースコードを用いてエンジニアの評価を行う。

表 2:冗長なソースコード

```
public static void main(String args[]){

   int eigo = 78;
   int suugaku = 90;
   int kokugo = 68;

   System.out.print("英語の試験結果は");
   if (eigo > 80){
     System.out.println("合格です");
   }else{
     System.out.println("不合格です");
   }
```

```
    System.out.print("数学の試験結果は");
    if (suugaku > 80){
      System.out.println("合格です");
    }else{
      System.out.println("不合格です");
    }
  }

    System.out.print("国語の試験結果は");
    if (eigo > 80){
      System.out.println("合格です");
    }else{
      System.out.println("不合格です");
    }
```

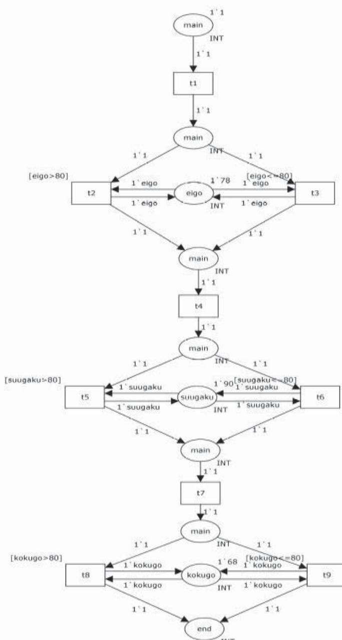
表 3：簡潔なソースコード

```
  public static void main(String args[]){
      int eigo = 78;
      int suugaku = 90;
      int kokugo = 68;

      check("英語", eigo);
      check("数学", suugaku);
      check("国語", kokugo);
    }

    private static void check(String kyoka, int seiseki){
      System.out.print(kyoka + "の試験結果は");
      if (seiseki > 80){
        System.out.println("合格です");
      }else{
        System.out.println("不合格です");
      }
    }
  }
```

これらのソースコードをペトリネットに変換すると、図 1,図 2 のようになる。



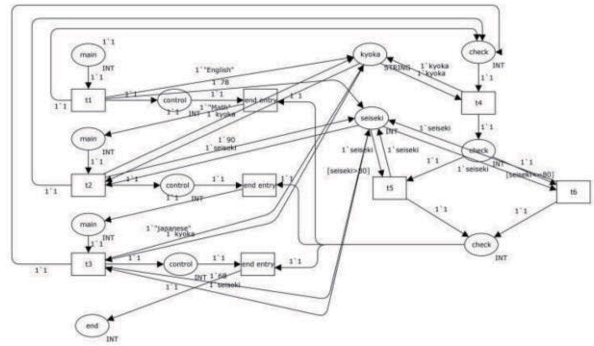


図 1：ペトリネットに変換した冗長なソースコード
図 2：ペトリネットに変換した簡潔なソースコード
表 4：ソースコードの行数とトランジション数の比較

	冗長	簡潔
行数	27	19
トランジション数	9	6(正規化された規模)

単純にソースコードの行数が長いほど優れたエンジニアと評価してしまうと、この例のような場合に冗長なソースコードの方が優れていることになってしまう。そこで同じ動作をする場合は簡潔な方を優れていると評価したい。しかし、単に行数の多い、少ないの比較をしてもソースコードの見通しの良さのための改行なども行数に含まれてしまい、誤差が生じる可能性がある。またどの部分が冗長なのか発見しにくい。

そこで、一つの評価値として、正規化された規模を用いる。今回の例では、冗長なソースコードは正規化された規模と比較すると、トランジション 3 個分(処理 3 個分)冗長な箇所が含まれていると言える。また、図 1 を見ると分かるように、一度ペトリネットに変換することで、プログラムの繰り返し箇所を発見し、冗長性の排除のための一助となる利点もある。

6 結論

ソースコードからペトリネットに変換し、正規化された規模を求めることで、エンジニアの評価値として使用できるということを示した。単なる行数の場合、見やすさのための空行の挿入など、行数の大小が評価と関係ないところで増減する可能性があり、一定の基準で評価することが難しかった。しかし、正規化された規模を用いることで、個人の癖などの影響を受けないロバスト性のある数値でエンジニアを評価することが可能になることがわかった。

また、ソースコードをペトリネットに変換したことで、ソースコードの動作をシミュレーションできるようになるので、テストの面で役に立つことも期待できる。

今後は今回の方法を実際のケースに当てはめてエンジニアを評価して、どれほど正確に評価が出来るかを確かめる必要がある。

参考文献

[1]フレデリック・P・ブルックス,Jr , "人月の神話" 丸善出版 (1975)
[2]中村翔吾『ソフトウェア開発における新しい見積り方法の提案』2016 年度愛知県立大学卒業論文 (2017)