ソフトウェア工学 117－3
（1997．11．21）

カラーペトリネットによる オブジェクト指向ソフトウエアのテストと解析方法

渡辺 晴美† 徳岡 宏樹† Wu Wenxin† 佐伯 元司†
†東京工業大学 情報理工学研究科
e-mail : harumi@cs.titech.ac.jp

オブジェクト指向ソフトウエアは、従来のテスト方法では十分にテストできないという問題がある。その理由は、オブジェクト指向プログラムが、オブジェクトの生成やダイナミックバインディング、そして並行性など実行時に決定される特徴を持つためである。本稿では、カラーペトリネットを用いたオブジェクト指向ソフトウエアの新しいテストと解析の方法を提案する。カラーペトリネットは、形式記述言語の一つであるペトリネットを拡張したもので、並行性の解析能力を持つ。提案する方法では、ネットの爆発的増加を避けるために、カラーペトリネットのカラートークンにより、オブジェクトの識別を行なう。オブジェクト指向ソフトウエアをカラーペトリネットへ変換する規則を与えることで、並行性を伴ったオブジェクト指向の挙動の側面に関するテストと解析を可能にする。提案する方法は、本質的に言語と独立である。例として Javaプログラミング言語で記述したプログラムをカラーペトリネットへの変換を取り上げ、提案する方法の有効性を示す。

A Technique for Analysing and Testing Object-Oriented Software Using Coloured Petri Nets

HARUMI WATANABE HIROKI TOKUOKA
WENXIN WU MOTOSHI SAEKI
DEPT. OF COMPUTER SCIENCE,
TOKYO INSTITUTE OF TECHNOLOGY

Object-oriented software essentially has dynamic aspect such as object creation, deletion and dynamic binding during execution. This aspect makes behavior analysis and tests of object-oriented software significantly complicated. In this paper, we propose a new technique of object-oriented software analysis and testing by using Coloured Petri Net (CPN), which is an extended version of Perti Net, one of formal techniques having the ability to analyze concurrent behavior. We represent identified objects with coloured tokens of a Coloured Petri net expressing object oriented software in order to avoid the net-explosion problem. We provide translation rules of object-oriented software into a coloured petri net and a technique to analyze and test effectively the net to investigate several behavioral properties. Our translation technique is essentially independent of which language we describe software. As an example, to show the effectiveness of our technique, we pick up Java programing Language and translate a specification written in it into a Coloured Petri Net.

1. はじめに

ソフトウエアの開発工程において、テストと解析の工程は必要不可欠である。リリース後に発見したエラーやバグを修正するための費用は、リリース前に発見した場合よりも4倍のコストが、そして、設計段階では60倍とさえ言われている[6]。最近普及しつつあるオブジェクト指向言語で記述されたプログラムは、オブジェクトの自動生成、ポリモフィズム、ダイナミックバインディング、そして継承など、実行時に決定される性質を数多く含む。そのため、プログラムの実行による挙動の確認するテストの重要性が高まっている。さらに、Javaなど最近のオブジェクト指向プログラミング言語では、並行動作も重要な要素の一つとなってきた。

このようなテストはプログラムを実行し、その挙動を観察することによりある程度テスト可能である。ところが、プログラムを実行するだけでは、デッドロックなど並行動作に関する誤りを検出できないという問題がある。

従来のテスト方法の中で、並行性の誤りを発見するために、並行性を記述できる形式記述法であるペトリネットを利用する方法がある[5]。ペトリネットは、属性と階層化の概念を持たないため、大きなシステムを表現できないという問題があった。これらの問題をカラーペトリネットは解決している[3]。カラーペトリネットは、属性と階層化の概念を持ち、デッドロックなどの並行性の誤りを検出でき、ツールも整備されている。しかし、カラーペトリネットは、オブジェクト

指向の概念を十分に表現する方法が確立されていないため、オブジェクト指向の概念を含んだシステムを解析が難しいという問題がある。オブジェクト指向では実行時に決定される性質を含むため、従来の方法では解析時にカラーペトリネットを動的に書き換えたり、発火規則と呼ばれるペトリネットの状態遷移の表現規則に変更を加えたり、他の形式記述言語を補捉的に用いているため、カラーペトリネットほどの解析能力を得ることができないという問題があった。

本稿では、並行性を含むオブジェクト指向プログラミング言語をテスト可能にするために、オブジェクト指向プログラミング言語を既存のカラーペトリネットツールで解析可能なカラーペトリネットへ変換する方法を提案する。提案する方法により変換したカラーペトリネットは、解析時にカラーペトリネットを動的に書き換える必要はなく、発火規則の変更もいらない。

以下、まず2で、本稿で提案するオブジェクト指向ソフトウエアのテストと解析方法の概要について述べ、3では、テストと解析に利用するカラーペトリネットについて簡単に説明する。4で、提案する方法について述べ、5でその評価を行なう。そして6で提案した方法が目的を達成できたかどうかについて検討する。

2. カラーペトリネットによるオブジェクト指向プログラムのテスト方法

本稿で提案するテストと解析方法は以下のように行なう。

(1) オブジェクト指向プログラムをCPN変換器に入力する。
(2) 変換器は、オブジェクト指向プログラムを入力すると、それを変換し、カラーペトリネット(CPN:Coloured Petri Nets)を出力する。
(3) 変換したCPNを、CPNの既存のツールであるDesign/CPN[7]に入力しテストと解析を行なう。

Design/CPNは、CPNをグラフィカルに記述でき、その挙動をシミュレーションし、そしてデッドロックなどの並行性に関する誤りを発見できるツールである。従って、オブジェクト指向プログラムをCPNへ変換できれば、CPNの解析能力の範囲で、誤りを発見することができる。

3. カラーペトリネット

本論文では、K. Jensenが提案したカラーペトリネット(CPN:Coloured Petri Nets)を用いる。本章ではこのCPNについて、図1を用いて説明する。

3.1 CPNの構成

CPNは、グラフとテキストの2つの領域からなる。グラフ領域は、図1の上半分であり、文字と、丸、四角、矢印により、グラフを記述する。丸、四角、矢印は、それぞれプレース、トランジション、アークと呼ばれるカラーペトリネットの要素である。プレース、トランジションの中に記述されている文字列は、それぞれの要素を識別するためのラベルである。プレースの外側の文字列で、下線が添付されていないものはカラー、下線が添付されている文字列は初期マーキングと呼ばれる。カラーはプレースが持つことのできるトークンの型のことであり、初期マーキングは始めにプレースに置かれるトークンのことである。

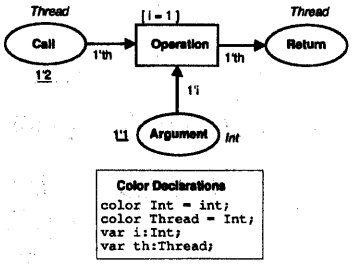


図1 An Example of CPN

アークに添付されている式はアーク式と呼ばれ、トランジションの外側に添付されている式はガードと呼ばれる。アーク式によって許可されたトークンのみがそのアークを通ることができ、ガードの式を満たした場合のみ、トランジションの発火、すなわち遷移が起きる。カラー、アーク式、ガードで用いる変数の宣言は、図1の下半分に示したカラー宣言と呼ばれるテキスト領域で行う。

図のCPNは、Call、Argument、Returnプレースと、Operationトランジションを持つ。CallとReturnプレースはカラーThreadのトークンを持つことができる。そして、初期マーキングはCallプレースとArgumentプレースに施されている。Callプレースの初期マーキングは、1'2で表されている。これは、Callプレースが値2のトークンを1個でマーキングされていることを示している。このトークンのカラーはプレースに添付されたカラーよりThreadであることがわかる。アークに添付されたアーク式1'thはThreadカラーのトークを1つだけ通すことができることを表している。変数thがカラー宣言部分でThreadカラーであることを宣言しているためである。トランジションに添付された[i=1]は、iの値が1ならばOperationトランジションが発火可能であることを示している。

3.2 CPNの挙動

CPNは、トークンによるプレースのマーキングにより、システムの状態を表現し、トランジションの発火によって遷移を表す。発火可能な状態になるとトランジションは発火することができる。発火が起きると、そのトランジションへ入力しているプレースのトークンが、トークンが通るアークのアーク式によって各々処理され、出力しているプレースへ移動する。発火可

OO concepts	CPN constructs
Class	a set of CPNs
Method (Service)	CPN
Instance (Object)	Token with a pair of an object identifier (incl. class name) and an an identifier of a thread
Thread	the same as above
Attribute(Instance variable)	Place
Value (inputs, outputs of methods and attribute values)	Token with corresponding colour
Type of Values	Colour
Action	Transition

表1 Relationship of OO-Concepts and CPN-Concepts

能かどうかは、トランジションへ入力している全てのアークのアーク式とトランジションに添付されているガード条件を満たしているかどうかによって判断される。アーク式を満たしている場合とは、アーク式で表されているトークンを、そのアークを出力しているプレースが有している場合である。

4. オブジェクト指向概念のカラーペトリネット概念への変換

本節では、提案するオブジェクト指向概念のカラーペトリネットへの変換方法について説明する。表1にオブジェクト指向概念を構成する要素とカラーペトリネットの構成要素との対応づけを示す。変換の処理は以下のように行なう。

(1) メソッドとコンストラクターを、表1に基づきカラーペトリネットに変換する。

(2) メソッドをカラーペトリネットへ変換しただけでは、各々のメソッドを表したサブネットは、独立しており、システム全体の挙動を表すことができない。そこで、インスタンスの生成、メソッドの呼び出し、継承、並行性の概念をカラーペトリネットに変換する。

(3) 上記二種類のカラーペトリネットを繋げることで、一つのカラーペトリネットにする。

次節以降では、図2のJavaプログラムを用いて、メソッドのCPNへの変換する方法について述べ、続いて、インスタンスの生成、メソッドの呼び出し、継承、並行性について説明する。そして、最後に一つのカラーペトリネットにする方法について述べる。図2は、文字型の値を持ったメッセージを伴ったメッセージにより、ObjBが始めに生成することで始まる。そのメッセージの文字が、'a'かどうかで、ObjまたはObjAのインスタンスを生成し、さらに、各々インスタンス毎にスレッドが生成されるプログラムである。

4.1 メソッドの変換

メソッドの変換方法は、従来の方法の多くと同様に、プレースを変数、トランジションを命令、そして、トークンは変数の値として表す。メソッド呼び出しは、トランジションによって表し、引数、戻り値のための変数はプレースによって表す。インスタンスの識別は、クラス名と識別子の組のカラーを持ったトークンで表

```
 1: // Obj.java (super class)
 2: import java.lang.*;
 3: public class Obj extends Thread{
 4:   public Obj(char c) {
 5:     Method(c);
 6:   }
 7:   public void run() {
 8:     System.out.println("Obj.run()");
 9:   }
10:   public void Method(char c) {
11:     System.out.println("Obj(). c=" + c);
12:   }
13: }
14:
15: // ObjA.java (sub class)
16: public class ObjA extends Obj{
17:   public ObjA(char c) {
18:     super(c);
19      Method(c);
20:   }
21:   public void run() {
22:     System.out.println("ObjA.run()");
23:   }
24:   public void Method(char c) {
25:     System.out.println("ObjA.c=" + c);
26:   }
27:
28: // ObjB.java (Initial class)
29: public class ObjB {
30:   public ObjB(char arg) {
31:     Obj obj;
32:     if (arg == 'a') {
33:       obj = (Obj)(new ObjA(arg));
34:     } else {
35:       obj = new Obj(arg);
36:     }
37:     obj.start();
38:     obj.Method(arg);
39:   }
40: }
```

図2 Javaプログラムの例

現し、スレッドの識別もスレッドの識別を表すカラーを持ったトークンで表す。

メソッドは、Javaでは図3に示す形式で表す。図3は、メソッドMethodの引数argの型がcharであり、戻り値の型がintであることを表している。これをCPNで表したのが、図4である。引数は、この場合argなので、argでラベル付けしたプレースで表される。戻り値は、Returnプレースで表される。

InプレースとOutプレースには、インスタンスの参照を表すカラーとスレッドを表すカラーの組のトークンが入る。この説明は、4.2と4.5で説明する。引数argの型はcharであるが、図4に示すように、インスタンス(Instance)とスレッド(Thread)そしてchar(Char)の組でカラーを表す(図6のカラー宣言参照)。この方法により、インスタンスとスレッドを識別して変数を扱うことが可能になる。詳細については、4.2と4.5で述べる。

メソッドの処理は、表1に従って変換する。図2のコンストラクタObjBを変換した例を図5に示す(基

```
int Method(char Arg) {
    :
    : Method process
    :
}
```

図3　Javaでのメソッドの表現

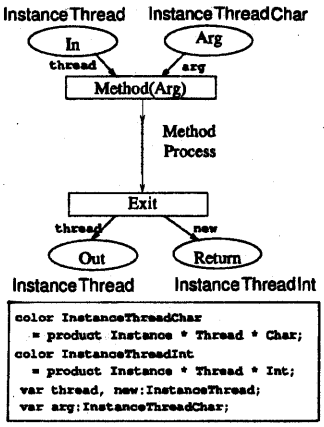


図4　メソッドの表現

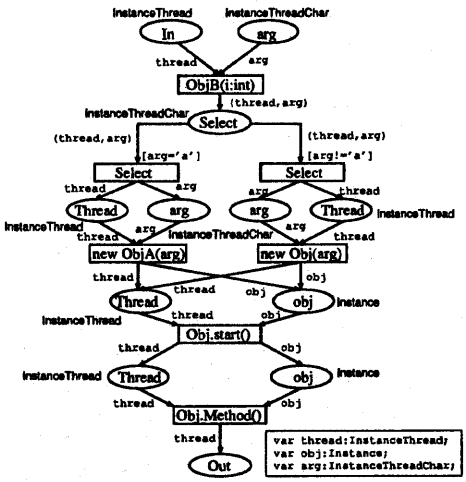


図5　変換したメソッドの例

本的なカラー宣言は、図6に示したものを用い、図5では省略する)。

4.2　インスタンスの生成

インスタンスの生成機能を表現するためには、単にコンストラクタをトランジションによって呼び出すだけではなく、新にインスタンスのための識別子を生成する機構が必要となる。この部分は、図6に示すCPNで表現する。この図は、図2の35行目の`obj = new Obj(arg)`をCPNに変換した場合の図である。表1に表すように、インスタンスはクラス名を表すカラーとインスタンスのIDを表すカラーの組で表されるトークンによって表現する。以下に、図6のCPNの処理について説明する。

(1)　新しいインスタンスを生成する。
(2)　継承があれば、継承のコンストラクタを呼び出す。
(3)　コンストラクタを処理する。

インスタンスの生成は、図6の枠で囲まれた部分で行なう。ClassNameプレースには、クラスネームを表すトークンが入っている。IDプレースへトークンが入って出ると、インスタンスIDが1プラスされる。従って、InstanceCreationトランジションが発火すると、クラスネームとIDを持つ新しいインスタンスが生成される。そしてそのIDは、すでに生成されたインスタンスのIDに1プラスされる。

なお、この枠で囲まれた部分は、インスタンスの生成をするときに共通に呼ばれる。すなわち、同じクラスの全てのインスタンスを生成する際に、この部分のCPNが共有される。この仕組みにより、インスタンス識別子をグローバルに管理することができる。

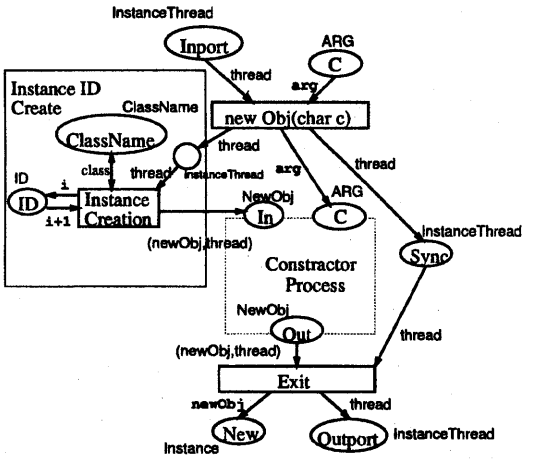
継承の前に、新しいインスタンスのトークンを生成しているため、もし、親クラスのコンストラクタ内で、子クラスと親クラスの両方に共通なメソッドを呼び出している場合、子クラスのメソッドがオーバーライドすることができる。

4.3　メソッド呼び出し

従来のCPNを用いたメソッド呼び出しの表現方法は、呼び出す側をトランジションで、呼び出されるメソッドをCPNで表す。すなわち、呼び出す側のトランジションと呼び出される側のメソッドが一対一に対応している。オブジェクト指向プログラミング言語の場合、呼び出されるメソッドが実行時に決定されるという問題がある。すなわち、メソッドを呼び出すトランジションと呼ばれたメソッドの処理を表すCPNを一対一に対応付けることはできない。オブジェクト指向プログラムでのこのようなメソッドの動的な選択は以下の3種類に分類できる。

(1)　引数の型は異なるがメソッド名が同じメソッドが複数あり、引数の型によってメソッドが選択される場合。
(2)　異なるクラスに同名のメソッドがあり、メソッドをバインディングしているインスタンスのクラスによりメソッドが選択される場合。
(3)　メソッドをバインディングしているインスタンスのクラスに呼び出すメソッドが存在せず、そのクラスが親クラスを持つ場合。

上記3は次節で説明する。まず始めの問題は、メソッドを呼び出す際に、メソッド名のみで識別するのではなく、引数を含んだ形でメソッドを選択すればよい。こうすることにより、メソッドを呼び出すトランジションと呼び出されるメソッドのCPNは一対一に対応付けることができる。




```
color ClassName = string;
color ID = int;
color Instance = product ClassName*ID;
color Thread = ID;
color InstanceThread = product Instance*Thread;
color NewObj = product Instance*InstanceThread;
color ARG = product InstanceThread*Char;
var thread:InstanceThread;
var arg:ARG;
var i:ID;
var class:ClassName;
var newObj:Instance;
```

図6　インスタンスの生成方法

次に二番目だが、この問題を解決するためには、図7に示すように、メソッド M1 はクラス C1 のメソッドかもしれないし、クラス C2 のメソッドかもしれない。そのため、図中の疑問符で表されている部分に何か仕掛けが必要となる。本論文では、この疑問符の部分もカラーペトリネットで記述することにより、一つの CPN で表現する方法を提案する。疑問符の部分を図8に示すようなカラーペトリネットで表すと解決することができる。この方法では、同名のメソッド (クラスは異なるが引数の型は同じメソッド) を全て集め、CPN を実行するときにメソッドを選択できる仕組みを CPN で与える。

インスタンスが生成されるたびに、選択される可能性のあるメソッドは増えるが、インスタンスはクラス名を表すカラーとインスタンス ID(インスタンス毎に付けられる整数値) を表すカラーの組のトークンであるため、そのトークンの値を計算することで、どのクラスのどのオブジェクトかを計算することができる。同じクラスのメソッドの処理は、異なるインスタンスでも全て同じ CPN であり、その違いはトークンのカラーによって表される。従って、図の上の部分にあるように、クラス名を判断して、トランジションを発火させればよい。図ではインスタンスを表すトークンのクラス名が Obj なら ClassName=Obj が真となるた

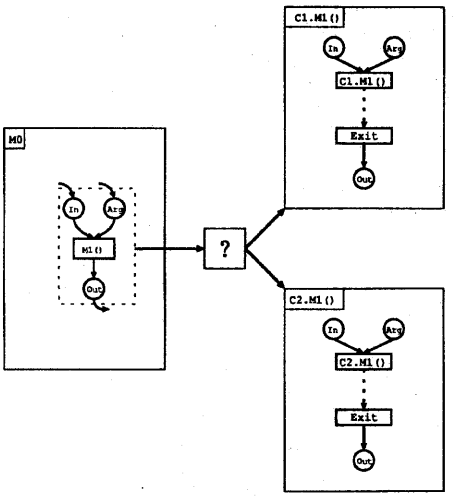


図7　ダイナミックバインディング問題

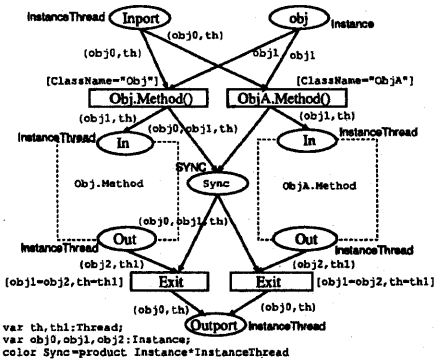


図8　メソッド呼び出しの CPN

め、左のトランジションが発火し、Obj のメソッドを実行する。

メソッドは、図 8の中央付近にある In プレースにトークンが入ると同時に、呼び出すメソッドの In プレースがマーキングされ、呼び出すメソッドの実行が始まる。そして、呼び出したメソッドの処理が終わるとメソッドの Out にトークンが入るのと同時に、図 8 の Out プレースにマーキングされる。

4.4 継　　承

前節で述べたように、メソッドをバインディングしているインスタンスのクラスに呼び出すメソッドが存在せず、そのクラスが親クラスを持つ場合、親クラスのメソッドを継承する。さらに、何段階も継承している場合でも、正しくメソッドが呼ばれなければならない。

すなわち、メソッドをバインドしているインスタンスのクラスに呼ばれるメソッドがある場合は、そのメソッドが呼ばれ、もしない場合は、そのメソッドを持

```
public class A{
  public ObjA() {....}
  int func() {....}
}
public class B extends A{
  public ObjB() {....}
}
public class C extends B{
  public ObjC() {....}
  int func() {....}
}
public class D extends C{
  public ObjD() {....}
}
```

図 9　Java プログラムの継承の例

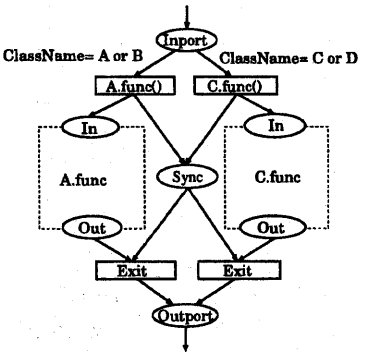


図 10　継承の CPN

つ最も近い親クラスのメソッドが呼ばれる。

この仕組みは、図 10のような CPN へ変換する。図 10の左側にあるような継承関係がある場合、図の右側の CPN へ変換される。左側の図は、クラス D はクラス C を継承し、クラス C はクラス B を継承し、そして、クラス B はクラス A を継承していることを表している。そして、クラス A とクラス C に同名のメソッド func() がある。クラス A とクラス B のインスタンスにバインドされているメソッドが呼ばれるとクラス A のメソッドが呼ばれる。クラス C とクラス D のインスタンスにバインドされているメソッドが呼ばれるとクラス C のメソッドが呼ばれる。この選択の機構は、図 10の CPN の Inport プレースから出力しているアークのアーク式で表される。Inport プレースには、スレッドとインスタンスを表すカラーの組のトークンが入ることができる。そして、インスタンスは、前で述べたように、クラス名とインスタンス ID の組からなるため、そのクラス名を用いてアーク式を表現できる。

4.5 並 行 性

本論文では図 12に示すようなスレッドによる並行処理の概念を考慮した。すなわち、インスタンスと独立に、スレッドを定義することができる。ことにより、Java 以外のオブジェクト指向プログラミング言語にも適用できるようにした。本論文では、並行性を表現

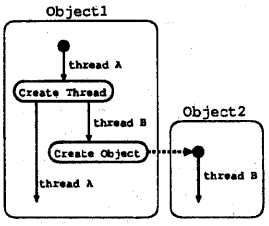


図 11　スレッドのモデル

するためにスレッドを表すカラーを導入した。スレッドを表すカラーはスレッドを識別するための整数値であり ThreadID と呼ぶ。インスタンスの生成と同様の機構でスレッドを生成し、そしてスレッドのグローバルに管理される。この概念を実装するには以下の 2 つの問題がある。

(1)　スレッドやインスタンスが正しいデータへアクセスできること。

(2)　スレッドやインスタンスが正しく制御されること。

一番目の問題の解決法は、全ての変数を表すプレースのカラーを変数の型だけではなく、スレッドを表すカラー、インスタンスを表すカラー、そして変数の型を表すカラーの三つのカラーの組のカラーで表す。これにより、どのスレッドのどのインスタンスのための値のトークンでマーキングされているかが分かる。

二番目の問題は、図 8の中央にあるプレース Sync によって解決できる。このプレースにより同期を取る。Sync は In と同じスレッドのカラーとインスタンスのカラーの組のカラーのトークンでマーキングする。これで、In プレースをマーキングしたスレッドとインスタンスの組のトークンと同じトークンが Out プレースにマーキングすることができる。

図 12に、スレッドを生成する処理を CPN へ変換した様子を示す。図の new Thread トランジションで、整数型のカラーを持ったトークンを新たに生成する。そして、Thread Start トランジションが発火すると、もとのスレッドは、Outport プレースを通してもとの処理へ戻る。一方新しいスレッドは、特別なメソッド run() に渡される。従って、Thread Start トランジションの発火により、二つのスレッドが並行に動き出す。

4.6 一つの CPN へ展開

メソッドとコンストラクタ毎に変換した CPN 分離されており、シミュレーションを行うことはできない。これらのメソッドは、上記で示したオブジェクトの生成、メソッド呼び出し、継承のために表した CPN を用いることにより、ばらばらな CPN を一つの CPN にする。図 13のプログラムを例に、その変換の様子を図 14〜図 15に示す。図 13は、Method1() が Method2() を 2 回呼び出し、Method2() は Operation() を処理

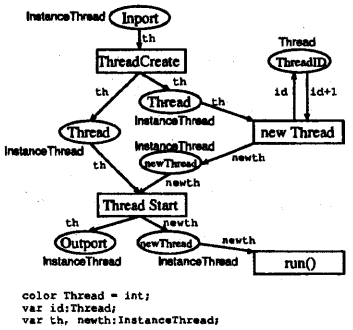


図12　スレッドの生成

```
Method1(){          Method2(){
  Method2();          Operation();
  Method2();        }
}
```

図13　Method 呼び出しの例

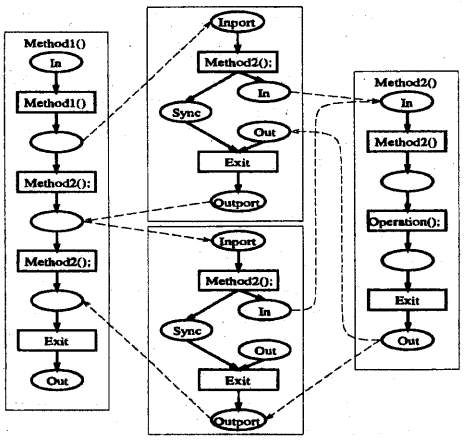


図14　メソッド呼びだし(1)

するプログラムである。この**Method1()** のプログラムを変換した CPN が、図 14の左側である。そして、**Method1()** のプログラムを変換した CPN が図 14の右側である。そして、メソッドを呼び出す仕掛けの CPN が中央の二つである。この場合、同じメソッドが一つしかないので、このような形式になっている。

図 14で、点線で結ばれたプレースがある。これらのプレースが共有されていれば、ばらばらな 4 つのネットを一つのネットにすることができる。そこで、図 15でのプレースやトランジションで点線のものを削除し、太線のアークを追加する。以上のようにして、一つのネットに展開する。

5. 評　　価

提案した方法を評価するために、図 16に示した座席予約システムを開発した。座席予約システムは、3 つ

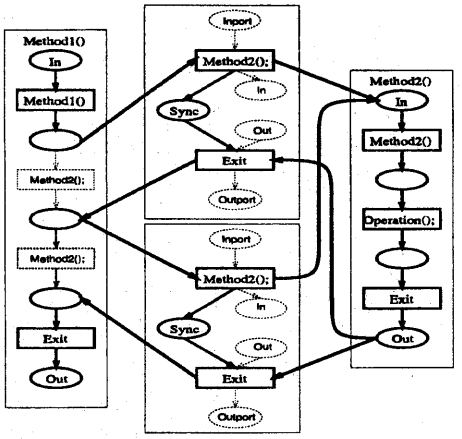


図15　メソッド呼びだし(2)

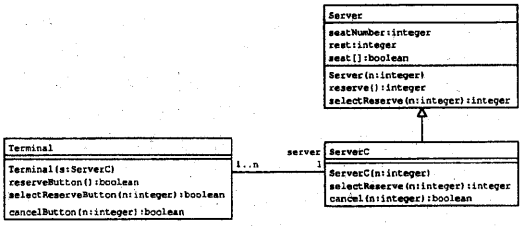


図16　座席予約システムのオブジェクトダイアグラム

Class	Contents
Server	Class of servers without the function of cancelling reservations
ServerC	Class of servers with cancelation function
Terminal	Class of terminals for user inputs and displaying results

表2　座席予約システムのクラス

のクラスからなり、各々のクラスの説明を表 2に示す。このプログラムは、プレースが 258 個、トランジション 74 個の大きさの CPN に変換された。そして、既存の CPN 評価ツールである Design/CPN で評価することができた。Design/CPN で実行している様子を図 17に示す。Design/CPN で評価した結果、ノードの数が 91、アークの数が 145 の full occurence graph を得ることができた。

以上から、変換した CPN が既存のツールである Desgin/CPN で評価できることが分かる。オブジェクト指向概念を含んだシステムをシミュレーションすることができ、その中にデッドロックなど並行性に関する誤りを含んでいるかどうかのテストが可能である。

6. 検　　討

本節では、提案した方法が、オブジェクト指向概念を CPN へ展開した時に生じるネットの爆発的な成長を抑えることができること、また、Java 言語だけでは

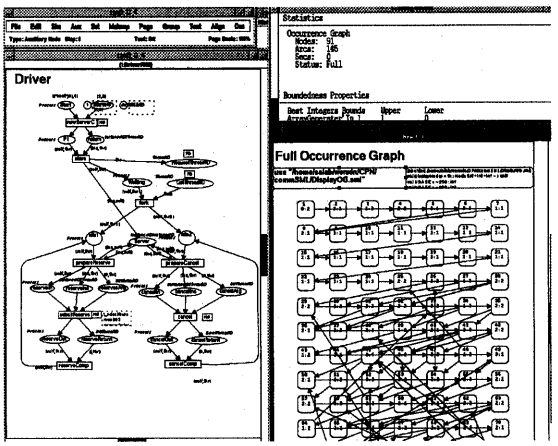


図 17　Design/CPN での実行の様子

なくオブジェクト指向概念を持った記述から CPN へ展開できる一般的な方法であることについて検討する。

従来の方法では、オブジェクト指向概念を CPN へ展開した時に生じるネットの爆発的な成長は、オブジェクトの生成、メソッドが呼び出される時、そしてスレッドが生成される時に生じる。オブジェクトの生成による場合、生成される前にあったネットに生成したネットを追加するためである。メソッド呼び出しの場合、呼び出されるメソッドが動的に決定されるため、メソッドを表す CPN をメソッド呼び出し時に追加するためである。これらの方法を、本稿では、インスタンスとスレッドをトークンで表し、同名のメソッドを一つのサブネットへ集めそれをインスタンスとスレッドのトークンを識別することにより選択するという方法を用いることで解決した。このようにすることで、CPN のグラフ部分は、プログラムから変換した時と変わらない。その理由は、インスタンスが生成されるとトークンが増え、値が変更されれば、トークンの値が変わり、メソッドが選択されても、あらかじめ変換してあるネットにマーキングされるだけであるためだ。

次に、本稿で提案した方法が、一般的なオブジェクト指向概念を持った記述に適用できることについて述べる。4で扱ったオブジェクト指向の概念、インスタンスの生成、メソッドの呼び出し、継承、並行性は Java 言語特有のものではない。Java 言語では、多重継承を許していないが、本稿で提案した方法では、多重継承も、継承するコンストラクタを全て並べることで可能である。並行性に関しても、インスタンスと独立なスレッドの概念を導入したため、一つのスレッド内に複数のインスタンスを逐次的に動かすこともできるし、インスタンス内に複数スレッドを持つ場合も表すことができる。インスタンス毎にスレッドが割り当てられるモデルも、インスタンス生成に続いて、スレッドの処理を行なえば良い。

また、メソッドという概念を持たないオブジェクト指向概念もある。この場合、オブジェクト間でのメッセージの送受信により関連を表現するが、送信の事象と受信の事象はトランジションで表現することができる。そして、事象により駆動するハンドラはメソッドと考えることができる。従って、この場合のモデルも単に送受信の事象をメソッドと考えれば提案した方法によりモデル化することが可能である。

7. おわりに

オブジェクト指向プログラムでは、挙動が実行時に決定されるため、テストすることが難しいという問題があった。本論文では、この問題を解決するために、オブジェクト指向プログラムを CPN へ変換し、CPN の解析機能を用いてデッドロックなどプログラムを実行しただけでは見つけることが難しい誤りを発見できるようにした。提案した方法が正しく動くかどうかを確かめるために、座席予約システムを開発し、テストした。その結果により、提案した方法で生成される CPN がプログラムと同じ挙動をシミュレートし、既存の CPN ツールで解析できることを示した。本論文では、Java を例に説明してきたが、本論文で扱った概念はオブジェクト指向プログラミング言語や形式的な仕様書にも適用できる。

参考文献

1) O.Biberstein,D.Buchs,N.Guelfi, "Modeling of Cooperative Editors Using CO-OPN/2", 17 th INTERNATIONAL CONFERENCE ON APPLICATION AND THEORY OF PETRI NETS Object-Oriented Programming and Models of Concurrency, pp.25-42,June 1996
2) J. Gosling, B. Joy, G. Steele, "The Java Language Specification", Addison-Wesley, 1996
3) K. Jensen, "COLOURED PETRI NETS – Basic Concepts, Analysis Methods and Practical Use – Volume 1-3", Springer-Verlag, 1992, 1994, 1997
4) C.Lakos, "The Consistent Use of Names and Plymorphism in the Definition of Object Petri Nets", Application and Theory of Petri Nets 1996 Lecture Notes in Computer Science 1091, pp.380-399,1996
5) J. L. Peterson, "Petri Net Theory and the Modeling of Systems", Englewood Cliffs, New Jersey, Prentice Hall Inc., 1981,
6) R. S. Pressman, "Software Engineering – A Practitioner's Approach Third Edition", McGraw-Hill, 1992
7) "Design/CPN Reference Manual for X-Windows Version 2.0", Meta Software Corporation, 1993