ICOT Technical Memorandum: TM-0886

TM-0886

ペトリネットを利用した
並列プログラムの解析

長谷川春朗、田口　毅、
中島俊介、本城　啓(沖)

May, 1990

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

ペトリネットを利用した並列プログラムの解析

An Analysis of Parallel Program by utilizing Petri Nets

長谷川晴朗*1、田口毅*1、中島俊介*2、本城哲*2

Haruo HASEGAWA, Takeshi TAGUCHI, Shunsuke NAKAJIMA and Akira HONJO

*1 沖電気工業(株)　*2 沖通信システム(株)

*1 Oki Electric Industry Co., Ltd.　*2 Oki Telecommunication Systems Co., Ltd.

概要: 本論文では並列プログラミング言語GHC(Guarded Horn Clauses)で記述されたプログラムをペトリネットを用いて解析する手法について述べる。並列プログラムをその実行前に検証しかつ並列性を抽出するため、GHCプログラムをペトリネットに変換するが、その変換方法には2種類考えられる。一つは述語論理の導出原理に従ったものであり、もう一つはプロセッサでの制御の流れに従ったものである。並列性の解析の点から後者の方が優れることを述べ、ペトリネットの構造的な性質を利用してプログラムの検証法と並列性の解析法に関して、幾つかの例をあげながら論じる。

1. はじめに

通信システムについてのユーザ要求が高度化･多様化するに従って、それを制御するコンピュータの処理能力に対する要求が飛躍的に増大している。これに対する解決策として、これまでにシングルプロセッサ制御からマルチプロセッサ制御(並列ではない)となっており、これを延長すると並列化ということが素直に考えられる。しかし、並列プログラミングではその特性として処理が並列に進行し、互いに干渉し合い、また実行順序に再現性がなく特にデバッグにおいて困難さ･複雑さが顕著である。また、並列コンピュータを利用して処理能力をあげるには、ソフトウェア自体の並列性を大きくすることが必要であり、そのため現時点でのソフトウェアに内在する並列性を抽出することが望まれる。それらに対処するべく、並列ソフトウェアの開発環境構築に向けて各所で研究がなされているが、ここではその一助として並列プログラムの動的なバグを検出し、また並列性を抽出し設計者に提示することによって最大限の並列性のあるプログラムを作成し得るための情報を提供することを考える。

一方、並列でない一般のプログラムに非同期･並行系システムのモデル化及び解析用として使用されるペトリネットを利用した例が見られる[10]。また、逐次の論理型言語の一つであるPrologで記述されたプログラムをペトリネットでモデル化した研究も見受けられる[4]。そこでここでは並列論理型言語で作成されたプログラムの並列性抽出及び正当性の検証等にペトリネットを利用する。なお、取り扱う言語としてはGHC(Guarded Horn Clauses)を考える。

本論文は以下の構成からなる。まず、2章でペトリネットを使用する上での基本用語について触れる。3章でPrologとの相違点をあげつつGHCの概要を説明し、4章ではGHCで発生し得るバグを分類する。5章でGHCからペトリネットへの2種類の変換ルールの紹介とそれらの比較を行う。6章でペトリネットによる、バグの検出及び並列性の抽出という解析手法について実例をあげながら記す。最後の7章はまとめである。

2. ペトリネットの利用

2.1 利用の目的　ペトリネット[5]は、非同期･並行系システムをモデル化し解析するものであり、各方面で使用されているが、その数学的な解析能力及びその記述性から初期の頃には考えていなかった分野にも適用されている。例えば、プログラムは一般にロジック(論理)とコントロール(制御)からなる[2]が、後者の制御部をペトリネットでモデル化して正当性の検証を行おうとする研究が行われている。また、一階述語論理のモデル化にも利用されており、従って論理型言語によるプログラムの検証も行われている。ここでは、Prologのような論理型言語に並列性を持たせたGHCで記述されたプログラムの並列度の抽出とデッドロック等の検出を行う試みについて述べる。

2.2 用語の定義　ここで使用するペトリネットの用語の定義を行う。ペトリネットを、4項組 N=(P, T, A, M)で表す。ここで、P、T、A、Mはそれぞれプレースの集合、トランジションの集合、接続行列、マーキングである。ここで、トランジションからプレースへのアークの多重度を示す接続行列、

及びその逆の向きの行列をそれぞれA^+, A^-とおくと、Aは次式で示される。

$$A = A^+ - A^- \quad (1)$$

マーキングMにおいて、トランジションkが発火するための条件を次式に示す。ここでr_kは、k番目の要素のみが1でそれ以外がすべて0であるベクトルである。

$$M \geqq A^{-T} \cdot r_k \quad (2)$$

各トランジションの発火回数を集計したベクトルを発火回数ベクトルという。初期マーキングをM_0、目的マーキングをM、その間の発火回数ベクトルをrとすると次式が成立する。

$$M = M_0 + A^T \cdot r \quad (3)$$

特にそれらが発火することによりもとのマーキングに戻るもの、つまり(3)式で$M=M_0$として次式が成立する非負整数ベクトルxをTインバリアントと呼ぶ。

$$A^T \cdot x = 0 \quad (4)$$

特に、和分解不能なTインバリアントを初等的Tインバリアントと言う。

3. GHCとは

GHC[1]は、Prologと同じく一階述語論理に基づく論理プログラミング言語であるが、同時に並列実行を基本としている。本章ではGHCの説明を行う。

3.1 ホーン節　一階述語論理の言語をLとし、$A_1, \cdots, A_n, B_1, \cdots, B_m$をLの原子式とする時、節は原子式を次式のように$\vee$で接続したものである。(ここで、$\neg, \wedge, \vee, \rightarrow$はそれぞれ否定、論理積、論理和、含意を示す。)

$$A_1 \vee \cdots \vee A_n \vee \neg B_1 \vee \cdots \vee \neg B_m$$

これは次式と等価である。

$$B_1 \wedge \cdots \wedge B_m \rightarrow A_1 \vee \cdots \vee A_n$$

ここで、$n \leq 1$の時これをホーン節と呼び、特に$n=1$の時確定節という。
$m \geqq 1$かつ$n=1$の時、IF-THEN形式のルールに相当し、$m \geqq 1$かつ$n=0$の時はデータベースに置かれるデータと考えられ、これと前記のルールと組み合わせて別のデータを導出していく。
純粋なPrologは、ホーン節に限定したものであり、一般には次の形式で表現する。

$$A_1 :- B_1, \cdots, B_m.$$

3.2 GHCの形式　GHCプログラムは、次に示すガード付き節の集合として表される。

$$\underbrace{H :- \overbrace{G_1, \cdots, G_n}^{\text{ガードゴール}} \mid \overbrace{B_1, \cdots, B_m}^{\text{ボディゴール}}.}_{\text{ヘッド}} \quad n, m \geqq 0 \quad (5)$$

ガード部　　ボディ部

ここで、H, G_i, B_jはそれぞれヘッド、ガードゴール、ボディゴールと呼ばれる。また、オペレータ"|"はコミット演算子と呼ばれ、これに先立つ部分をガード部、これに続く部分をボディ部と呼ぶ。一般に、プログラムのアルゴリズムは論理と制御の和で表されるが、その論理つまり宣言的意味、及び制御つまり手続き的意味について以下に述べる。

3.3 GHCの意味論

(A) 宣言的意味　コミット演算子は宣言的には","と同じであり、従って式(5)は「節に含まれる変数のすべての値に対して、$G_1, \cdots, G_n, B_1, \cdots, B_m$が真の時、Hが真になる。」と読むことができる。

(B) 手続き的意味　Prologと同じように、ゴール(複数)が与えられた時その中の一つのゴールについて、プログラム中の節のヘッドと同一化(ユニフィケーション)を行う。複数のゴールの同一化、及び一つのゴールと複数ある節中の一つのヘッドとの同一化は、それぞれAND並列、OR並列と呼ばれ、基本的にすべて並列で実行される。そのための同期の概念を導入するべく、ガードを採用している。

式(5)の形をした節の集合Cがあって、今Gを解くためにはGとHの同一化、及び$G_1, \cdots, G_n$の実行を行う必要がある。Gと同一化する節の選択については、Prologのような逐次ではなく、完全にフェアである。わかりやすく言うと、どのような順序で実行されるかは実行してみなければわからず、また実行する度に異なり得るということである。簡単な例を図1に示す。図1(a)は6個の節からなるGHCプログラムである。ここで、ゴールaを実行した時の実行順序を考えてみよう。コミット演算子以降に並べられたゴールを実行する順序はプログラム中に記述された順番とは無関係であり、もし並列に実行可能であれば並列に実行され得る。つまり、図2(b)のゴール木に示すような制約(b, cはaの後、d, eはbの後、f, gはcの後)を満足さえすれば、どのような実行順序でもよいのである。

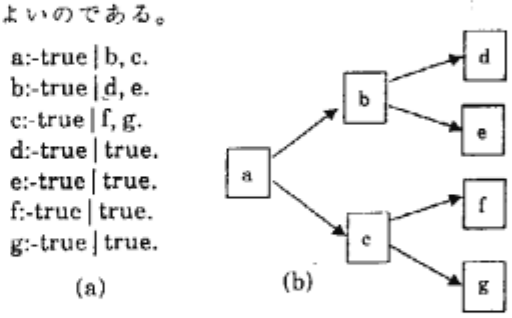


図1 GHCプログラム例とそのゴール木

なお、実際の実行はハードウェア環境や処理系等に依存する。例えば、プロセッサが1個の場合のGHCプログラムの実行は当然シーケンシャルとなるが、その実行順序は完全に処理系に依存することを意味している。因みに図1(a)のプログラムを逐次に実行する場合、上記の制約を満足する

その実行順序の組み合わせの数は全部で80通りある。

なお、GとHの同一化及びガードゴールの実行にあたって決してGの中の変数を具体化してはならない。具体化しようとするときは、できるまでその同一化は中断される。このように、ガードを導入することにより同期を自然な形で表現できる。例えば、図2に示すようなプログラムの場合、ゴールa(X)を与えると、必ず節2は節3の後に実行される。それは節2はガード部でXを具体化しようとするため、節3でXがmにユニファイされるまで中断されるからである。

```
a(X):- true|b(X), c(X).          節1
b(X):- X=m|true.                 節2
c(X):- true|X=m.                 節3
```

図2　引数を持ったGHCプログラムの例

3.4 PrologとGHCのプログラム例　GHCをよりよく知るために、Prolog[7]との簡単な比較を行う。例えば、[1, 2]と[3, 4]のリストを結合して[1, 2, 3, 4]とするような2つのリストを結合するappendプログラムを両言語で記述してみる。これを図3に示す。

```
append([], Y, Y).                                   (a1)
append([A|X], Y, [A|Z]) :- append(X, Y, Z).         (a2)
```

(a) Prolog プログラム

```
append([], Y, Y) :- true | true.                    (b1)
append([A|X], Y, [A|Z]) :- true |
                        append(X, Y, Z).            (b2)
```

(b) 間違ったGHCプログラム

```
append(X, Y, Z) :- X=[] | Z=Y.                      (c1)
append(AX, Y, AZ) :- AX=[A|X] |
                AZ=[A|Z], append(X, Y, Z).          (c2)
```

(c) 正しいGHCプログラム

```
append(X, Y, Z) :- true | X=[], Y=Z.                (d1)
append(AX, Y, AZ) :- AZ=[A|Z] |
                AX=[A|X], append(X, Y, Z).          (d2)
```

(d) 分割のためのGHCプログラム

図3　Append機能のプログラム

図3(a)はPrologで記述したプログラムであるが、その動きを簡単に追いかけてみる。[1, 2]と[3, 4]の結合、つまりappend([1, 2], [3, 4], Z)を考える。ヘッドがユニファイするものを上の節から探していく。まず、[1, 2]は[]とユニファイしないため、(a1)はスキップする。次に、(a2)の[A|X]とユニファイしてA=1, X=[2]となる。従って、Y=[3, 4]、Z=[1|Z']となる。つまり、次式のようになる。

append([1, 2], [3, 4], Z) :- append([2], [3, 4], Z').

次にappend([2], [3, 4], Z')とユニファイするものを探すとやはり(a2)がある。上記と同様に行うと、次式のようになる。

append([2], [3, 4], Z') :- append([], [3, 4], Z")

append([], [3, 4], Z")とユニファイするものは(a1)であってZ"=[3, 4]、従ってZ'=[2, 3, 4]、Z=[1, 2, 3, 4]が得られる。

次にGHCプログラムを考えてみる。図2(b)は(a)にガードとしてtrueを挿入しただけであるが、このプログラムは間違いである。なぜなら(b1)とユニファイできないのは当然として、(b2)とユニファイする時にZを[1|Z']にバインドしようとするからである。つまり、(b2)ともユニファイできずに中断したままとなることを意味する。Fig3.(c)のプログラムでは、それを解消してガード部でゴール中の変数を具体化しないようにしている。

次に2つのリストの連結ではなく、append(X, Y, [1, 2, 3, 4])のような、あるリストの2個のリストへの分割を考える。図3(a)を使用すると、X=[], Y=[1, 2, 3, 4]; X=[1], Y=[2, 3, 4]; X=[1, 2], Y=[3, 4]; X=[1, 2, 3], Y=[4]; X=[1, 2, 3, 4], Y=[]の5つの全部の解を求めることができる。ところが、(c)のプログラムでは(c1), (c2)ともXを具体化しようとしてできず中断するだけである。

そこで、リストの分割を行えるようにしたものが図3(d)である。ユニフィケーションの順序は節の並びに依存しないため、もしゴールappend(X, Y, [1, 2, 3, 4])がただちに(d1)とユニフィケーションするとX=[], Y=[1, 2]となり、逆に(d2)とばかりユニファイしていると4回の後に初めて(d1)とユニファイしてX=[1, 2, 3, 4], Y=[]が得られる。つまり、一見変数を(ボディゴールで)具体化するとバックトラックすることがないため、基本的に全解探索を行うことができず、上記5個の解のうちのどれかが得られるだけである。

4. GHCプログラムの解析

本章では、ペトリネットへの変換によりGHCプログラムをどのように解析するかについて記す。

4.1 GHCのバグの分類　GHCのプログラムに含まれるバグは2つに分類される[8]。一つは、基本的に実行前にチェックできるものであり、これを静的バグと呼ぶ。もう一つは、基本的に実行時までエラーが判明しないものであり、これを動的バグと呼ぶ。

静的バグについては、例えばシンタックスに関するものや述語記号のスペリングミス・引き数の数の不一致等であり、これらは比較的容易に訂正可能である。一方動的バグについては、以下の4つに分けられる。

(1)並列性と非決定性による、プログラマの意図に反する動作の可能性
(2)決して具体化されない変数を参照したり、どの節の条件部も成功しないことによるデッドロックの発生
(3)有限で停止すべき手続きの無限ループ
(4)値の具体化した変数への異なる値の再度の代入
(5)組み込み述語の誤った使用

ここでは、GHCプログラムをペトリネットに変換することにより、動的なバグをプログラムの実

行前に少しでも検出しようとすることを目的としている。

4.2 GHCプログラムの並列性の解析　第1章で述べたとおり、並列プログラムを使用する最大の理由は処理能力をあげることである。GHCプログラムの非決定性のためにどのようにプロセッサに分散され、どのような順序で実行されるかは実行する度に異なるものである。これらに対処するため、プログラムの実行時にメタインタプリタ等により情報を収集し、それをリアルタイム又は終了後にCRTに表示してプログラマにプログラム変更のための情報を提供することが考えられる[9]。しかし、ここではペトリネットの段階で並列性の解析を行う。従って、プロセッサの数にもよるが、プログラムの並列性を抽出してできればそれの最適な負荷分散を与えることが望まれる。例えば、図4はGHCプログラムをある方法によりペトリネットに変換したものである。ここでプロセッサ数が4個の時、点線で囲まれたものを同一のプロセッサで実行すれば、充分に並列度も高くまたプロセッサ間の通信量も少なくなると考えられる。このように最適な負荷分散を与えることを可能とするための適切な情報をプログラマに提供しようとするものである。

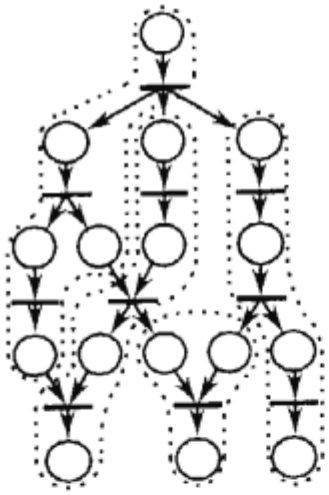


図4 ペトリネットによる並列度分析例

5. ペトリネットとの対応

5.1 変換ルール　GHCのプログラムとペトリネットとの対応を表1に示す。

一階述語論理の節をトランジションに、原子式をプレースに、また基本的にはカレントのゴールの引数をトークンに対応させて、プログラムをペトリネットに変換する。従って、節の実行はトランジションの発火に対応する。但し、ガードボディの位置とアークの向きについては、2種類の考え方がある。1つは、述語論理における導出原理に従ったものであり、ガードゴール及びボディゴールを入力プレースに、ヘッドのみを出力プレースとするものである。もう1つはプロセッサでの制御の流れに従ったものであり、ヘッド及びガードゴールを入力プレースに、ボディゴールを出力プレースに対応させるものである。

表1 GHC－ペトリネットの変換ルール

ルール	トランジション	入力プレース	出力プレース	トークン	発火
(a)	節	・ガードゴール ・ボディゴール	ヘッド	カレントのゴールの引数	節の実行
(b)	節	・ヘッド ・ガードゴール	ボディゴール	カレントのゴールの引数	節の実行

簡単な例を図5~7に示す。図5はGHCで書かれたプログラムであり、ここではまず簡単のために引数を持たない述語のみから成り、かつガードがすべてtrueであるプログラムを取り上げる。図6(a)は、それを表1の変換ルール(a)に従って作成したペトリネットであり、ここではpとqの述語が与えられれば、ペトリネット上ではそれらを示すプレース上にトークンが置かれることになる。この後は、各ゴールとヘッドがユニファイすることでペトリネット上でのトランジションが発火する。また、図6(b)は、表1の変換ルール(b)によって作成したペトリネットである。これらのペトリネットの接続行列は図7のように表される。

あるプログラムであるデータつまり知識が与えられた時に、求めたい結果が得られるかを調べるには、ペトリネット上で初期マーキングから求めるマーキングに至る発火系列があるかどうかを調べればよい。この初期マーキングと求める目的マーキングはプログラマが与えるものである。例えば、図6(a)では、ガードゴールとボディゴールの両方がtrueとなっている節であるpとqの両プレースにトークンを与えたものを初期マーキング、rとsのプレースにトークンを与えたものを目的マーキングとすれば、(3)式を満足する非負整数ベクトルγの存在は発火系列の存在のための必要条件である。逆にいうと、そのようなベクトルγが存在しなければ失敗することを意味する。ここで両者とも初期マーキング、目的マーキング共にトークンを0にするようなトランジションを追加する(図6の点線で示す)ことにより、その間の発火系列をTインバリアントで表現することができる。なぜなら、初期マーキングと目的マーキングの差が0となるからである。但し、初期マーキングを生じさせるトランジション及び最終の目的マーキングからのトランジションは発火しなければならない。

5.2 両変換ルールの比較　一般にあるマーキングからそのマーキングに戻る発火系列が存在するための必要条件は、そのネットでTインバリアントが存在することである。さらに、任意のトランジショ

```
p :- true | true.   (T1)
q :- true | true.   (T2)
r :- true | p, q.   (T3)
s :- true | q, r.   (T4)
p :- true | s.      (T5)
r :- true | s.      (T6)

?- s, r.            (T7)
```

図5 GHCプログラム例

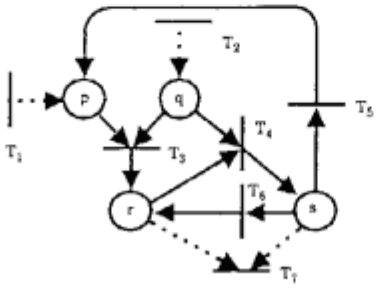
$$A = \begin{array}{c} \\ T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \\ T_6 \\ T_7 \end{array} \begin{array}{c} \begin{array}{cccc} p & q & r & s \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & -1 & 1 & 0 \\ 0 & -1 & -1 & 1 \\ 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & -1 \end{pmatrix} \end{array}$$

(a)接続行列

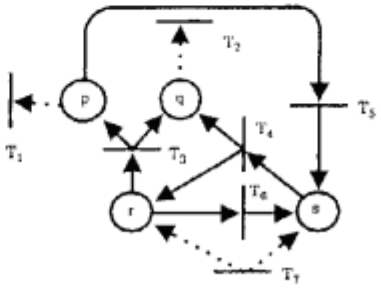
$$A = \begin{array}{c} \\ T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \\ T_6 \\ T_7 \end{array} \begin{array}{c} \begin{array}{cccc} p & q & r & s \end{array} \\ \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 1 & -1 & 0 \\ 0 & 1 & 1 & -1 \\ -1 & 0 & 0 & 1 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

(b)接続行列

図7 図6に示すペトリネットの接続行列



(a) 変換ルール(a)によるネット



(b) 変換ルール(b)によるネット

図6 図5に示すプログラムのペトリネット

ンから出力プレースに至るアークの総数が1以下の時はそれが十分条件にもなることが知られている[4]。従って、トランジションからの出力プレースが高々1個であってその多重度が1である(a)の変換ルールではそれが適用できて望ましいと考えられる。しかし、ガード部に変数があってそれがある定数に等しいかを調べるような時、それを示すプレースのトークンは発火によって消滅してはならないため、自己ループとする等の処置が必要となり、必ずしも上記の前提が成立するとは限らない。さらに、GHCのプログラムでは、ストリームというリストの概念でデータを取り扱うことが多く、そのため再帰を頻繁に使用するため(a)の変換方法をうまく使用できる場合が多くない。また、並列性の解析にあたっては、制御の流れを忠実に追いかけることが必要であり、その面からも(b)の変換方法が望ましい。

なお、GHCからペトリネットへの変換アルゴリズムをAppendixに示す。

6. ペトリネットによる解析

前章で示した変換ルールによって以下の2つの解析を行う。前章では、Prolog的なプログラムのみを扱ったので、ここでは実質的なガードゴールが存在し、かつ引数を持つプログラムも取り扱う。なお、変数は大文字で、それ以外の述語や定数は小文字で表す。

6.1 論理的な誤りの検出

4.1節でGHCプログラムのバグの分類を行ったが、表1(b)の変換によりプログラムの論理をペトリネットでモデル化することで、あるゴールからあるゴールに至る系列の存在のための必要条件を調べることができる。このため単にネットの構造的な性質に基づいて解析することになり、従って実際にプログラムを動作させることなく、またペトリネットにおいても実行させることもないため、解の存在の検出・解の導出を高速に行うことができる。すなわち、初期マーキングと目的マーキングを設定した上でその間の発火系列が存在する必要条件を求めることができる。例えば、図6(b)でT_1, T_2, T_7を追加する前のネットにおいて、初期マーキング(0, 0, 1, 1)から目的マーキング(1, 1, 0, 0)に至る発火可能ベクトル、つまりT_1, T_2, T_7を追加したネットでT_7を発火させる初等的Tインバリアントを求めると、(6)式より$x = [2\ 3\ 2\ 1\ 0\ 0\ 1]$となる。

これよりトランジションT_5及びT_6に相当する節はs・rを得るには必ずしも必要ではないといえる。逆にもしここで、図5のプログラムでトランジションT_4に相当する節がない時、$A^T \cdot x = 0$を満足する解がないため発火系列が存在せず、s・rが成立しないことがわかる。

$$A^T \cdot x = \begin{pmatrix} -1 & 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & -1 & 1 \\ 0 & 0 & 0 & -1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 2 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = 0 \quad (6)$$

なお、トランジションの追加方法には幾つかのものが考えられる。例えば、図6ではT1、T2とT7のみを追加したが、これら以外にも初期マーキング又は目的マーキングとなるであろう、プレースの入出力となるトランジションを追加しておくのが望ましい。そして、ある初期マーキングからある目的マーキングに至る発火可能ベクトルを求める時には、すべてのトランジションを追加した接続行列について初等的Tインバリアントを求め、その中で前記のトランジションが発火するものが必要とするTインバリアントとなる。

例えば、図5のプログラムでプログラマの意図としてrをゴールに設定する可能性のある時は、図8に示すように図6のペトリネットにさらにT8のトランジションを追加する。この時、その接続行列は図9に示すようになる。

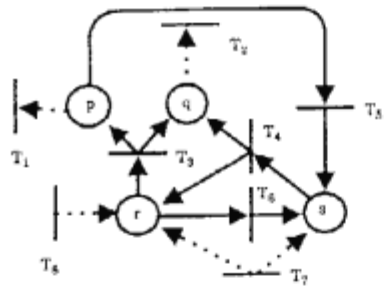


図8　図6(b)にT8を追加したペトリネット

$$A = \begin{array}{c} \\ T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \\ T_6 \\ T_7 \\ T_8 \end{array} \begin{array}{c} \begin{array}{cccc} p & q & r & s \end{array} \\ \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 1 & -1 & 0 \\ 0 & 1 & 1 & -1 \\ -1 & 0 & 0 & 1 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

図9　図8の接続行列

この初等的Tインバリアントを求めると以下の4個の解が得られる。

$\mathbf{x}_1 = [1\ 1\ 1\ 0\ 0\ 0\ 0\ 1]$
$\mathbf{x}_2 = [0\ 1\ 0\ 1\ 0\ 1\ 0\ 0]$
$\mathbf{x}_3 = [0\ 2\ 1\ 1\ 1\ 0\ 0\ 0]$
$\mathbf{x}_4 = [2\ 3\ 2\ 1\ 0\ 0\ 1\ 0]$

もし、r·sのゴールを解くためには、そのTインバリアントの中からT7が発火するものを選んでx4が得られる。同様に、rのゴールを解くためにはT8が発火するもの、つまりx1を選べばよい。このように、予め解くべきゴールを決めておくことにより、その追加トランジションを加えたペトリネットで1回だけTインバリアントを計算するだけでよい。

図10(a),(b),(c)にそれぞれGHCのプログラム、そのペトリネット変換及びその接続行列を示す。ここで、「X=m」は、述語unify(X,m)と見なす。プログラムからペトリネットへの変換については、同じ述語毎に1つのプレースに対応させるが、同じ述語であっても引数が異なる場合はトークンの色によって識別するハイレベルネットを使用する。図10(b)ではアーク上にトークンの色を表示している。従って、例えばプレースuとトランジションT3との間で自己ループを形成しているように見えるが、そのアークを通るトークンの色が異なるため接続行列の対応要素は0ではない。この時スタート:p(X)からゴール:X=bに至る発火系列は、接続行列のTインバリアントから求めることができ、$A^T \cdot x = 0$より$x = [1\ 1\ 1\ 0\ 1\ 1]^T$となる。つまりこのプログラムでは、T3に相当する節が冗長なことがわかる。

6.2　並列性の抽出　並列性の抽出については、節の同一化の過程を調べることによって、各時点において並列に実行できる節の数を求めるものである。つまり、ペトリネットに変換した後で、節の実行をトランジションの発火ととらえて発火し得るトランジションの数をその時点における並列度と考える。従ってその結果、複数の節による中断によってデッドロックになることも検出することができる。

今、話を簡単にするためにすべてのアークの多重度を1とする、ピュアなペトリネットを考える。無論このように仮定したからと言って一般性を失うことはない。そのようなペトリネットでは、トランジションの入力プレースのすべてにトークンがあるとき、そのトランジションは発火する。あるマーキングのもとで(2)式を満足するkの総数、つまり発火し得るトランジションの数がその時点での並列度である。次にそれらのトランジションの中のすべてが発火したとしてその時のマーキングで発火し得るトランジションの数を求める。このように順次発火し得るトランジションの数を求めていくことにより並列度を求めることができる。例えば、図6(b)のネットでは、Tインバリアントから各トランジションの発火回数が求められる。つまり、全体の発火回数は、各トランジションの発火回数の総計であり、それは2+3+2+1+0+0+1=9回である。その発火順序については、幾つかのトランジション間での時間的な制約さえ満足すればそれ以外の指定はない。例えば、$T_7 \rightarrow T_4 \rightarrow T_3 \rightarrow T_3 \rightarrow T_2 \rightarrow T_2 \rightarrow T_2 \rightarrow T_1 \rightarrow T_1$のように完全にシーケンシャルであってもよい。しかし、トランジションの同時発火を最大限に認めると、その発火系列は表

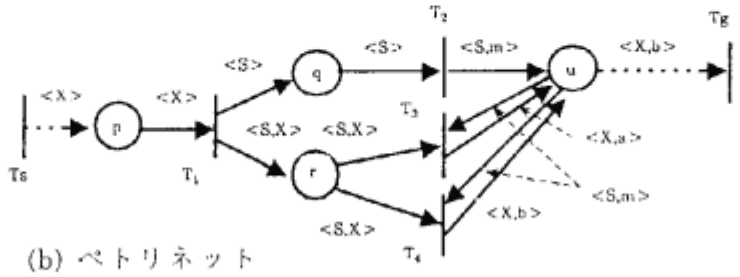
p(X) :- true | q(S), r(S,X). (T_1)
q(S) :- true | S = m. (T_2)
r(S, X) :- S = m | X = a. (T_3)
r(S, X) :- S = m | X = b. (T_4)

スタート: p(X) (Ts)
ゴール: X=b (Tg)

(a) GHC プログラム

$$A = \begin{array}{c} \\ Ts \\ T_1 \\ T_2 \\ T_3 \\ T_4 \\ Tg \end{array} \begin{pmatrix} p & q & r & u \\ <X> & 0 & 0 & 0 \\ -<X> & <S> & <S,X> & 0 \\ 0 & -<S> & 0 & <S,m> \\ 0 & 0 & -<S,X> & -<S,m> \\ & & & +<X,a> \\ 0 & 0 & -<S,X> & -<S,m> \\ & & & +<X,b> \\ 0 & 0 & 0 & -<X,b> \end{pmatrix}$$

(c) 接続行列



(b) ペトリネット

図10 引数を持ったGHCプログラムとそのペトリネット

2に示すとおりである。その最大瞬間並列度(同時に発火できるトランジションの数)は4、また最短ステップ数が4であることから平均並列度(各ステップでの並列度の総和÷最短ステップ数)は2.25(=9÷4)であることがわかる。

表2 図6のプログラムの並列度

ステップ	発火トランジション	発火後の各プレースのトークン数			
		p	q	r	s
1	T_7	0	0	1	1
2	T_3, T_4	1	2	1	0
3	T_1, T_2, T_2, T_3	1	1	0	0
4	T_1, T_2	0	1	0	0

なお、Tインバリアントの存在は発火系列存在のための必要条件であって十分条件ではない。Tインバリアントが存在しても解がないことがある。また、図6(a)のペトリネットは、図6(b)のそれとはガード部がすべてtrueであってTインバリアントとしては符号が逆となるだけであるが、その実行過程は大きく異なる。例えば、もし図6(a)で同じように並列度を求めると、$T_1,T_1,T_2,T_2,T_2 \rightarrow T_3,T_3 \rightarrow T_4 \rightarrow T_7$と4ステップは同じであるが、逆にしてもプログラムの実行過程を表さない。

このようなデータを設計者に提示することにより、設計者は高速に実行するために並列性を最大限に引き出すことのできるプログラムの修正を行うことができる。

7. あとがき

ここでは、並列論理型言語をペトリネットに変換してデッドロック等の検証を行う手法の考え方について述べた。簡単な例についてネットの構造的な性質を利用し、Tインバリアントを用いて解析することにより、バグの検出や並列性の解析を少ない計算時間で行えることを示した。一度バインドした変数はもう変更ができないという特徴を有するGHCで頻繁に使用されるストリームのペトリネットによる表現方法、更には並列性を抽出するスマートな方法など課題が残っており、これらについては今後検討を行っていく予定である。日頃有益な討論を行う当社ソフトウェア技術開発部の諸氏に感謝する。なお、本研究は第5世代コンピュータプロジェクトの一環として行っているものである。御指導を戴くICOTの長谷川隆三第一研究室長に深謝する。

[参考文献]

1)淵一博監修：並列論理型言語GHCとその応用、共立出版(1987)
2)R.A. Kowalski : Algorithm = Logic + Control, Comm. ACM, Vol. 22, No. 7, pp.424-436(1979)
3)前田賢一他：ペトリネットによる並列プログラムの動作解析に関する一考察、情報処理学会、第40回全国大会、2R-3(1990)
4)G. Peterka, T. Murata, : Proof Procedure and Answer Extraction in Petri Net Model of Logic Programs, IEEE, Trans. on SE, Vol. 15, No. 2(1989)

5)J.L.Peterson : Petri Net Theory and the Modeling of Systems, Prentice Hall(1981)
6)T. Murata : High-Level Petri Nets for Logic Programming and AI Applications, PNPM89, Pre-Workshop Tutorials, pp.209-260(1989)
7)中島秀之 : Prolog、産業図書(1983)
8)前田宗則 : 色効果を利用したGHCデバッガ、電子情報通信学会、ソフトウェアサイエンス研究会、SS89-14 (1989)
9)安藤津芳恒 : 並列プログラムの動作評価に関する一検討、情報処理学会、第40回全国大会、2R-2(1990)
10)S. M. Shatz, W. K. Cheng, : A Petri Net Framework for Automated Static Analysis of Ada Tasking Behavior, Journal of Systems and Software, 8, pp.343-359 (1988)

[Appendix]

GHC→ペトリネット変換:
ここでは、ガードゴールに対してユニファイのみ扱ったプログラムを対象とする。説明をわかりやすくするために図Aに示す例を取り上げる。

```
p(X):-true|q(Y), r(Y, X).
q(X):-true|X=m.
r(m, X):-true|X=a.
r(m, Y):-true|Y=b.
```

図A　プログラム例

GHCプログラムのペトリネット変換を以下のステップで行う。

[ステップ1]　前処理
[ステップ1.1]　プログラムの変数名を内部的に同一のものとする。

```
p(X):-true|q(Y), r(Y, X).
q(Y):-true|Y=m.
r(m, X):-true|X=a.
r(m, X):-true|X=b.
```

図B　変数名統一後のプログラム

[ステップ1.2]
ヘッドからのガードゴールを抽出することにより、プログラムのガードを明確にする。

```
p(X):-true|q(Y), r(Y, X).
q(Y):-true|Y=m.
r(Y, X):-Y=m|X=a.
r(Y, X):-Y=m|X=b.
```

図C　ガードゴール抽出後のプログラム

[ステップ1.3]
ユニファイ機能を原子式(たとえばu(X, Y))で表現する。

```
p(X):-true|q(Y), r(Y, X).
q(Y):-true|u(X, m).
r(Y, X):-u(Y, m)|u(X, a).
r(Y, X):-u(Y, m)|u(X, b).
```

図D　ユニファイを原子式で表現した後のプログラム

[ステップ2]　ペトリネット変換
[ステップ2.1]
ヘッド・ガードゴール・ボディゴールからプログラムの3次元ベクトルに変換する。

プログラムのベクトル表現{Head, Guard, Body}
Head:ヘッドの述語名リスト
Guard:ガードゴールの述語名(trueはnullとする)リスト
Body:ボディゴールの述語名リスト

```
[{[p], [], [q, r]},
 {[q], [], [u]},
 {[r], [u], [u]}
 {[r], [u], [u]}]
```

図E　プログラムのベクトルのリスト

[ステップ2.2]
下記の変換ルールにより、プレース・トランジション・アーク・トークンの集合を抽出する

変換ルール
(1)各節毎に一つのトランジションを対応させる。
(2)一つの原子式に一つのプレースを対応させる。
(3)ヘッド及びガードを入力トランジションに、ボディを出力トランジションに対応させる。
(4)アーク上を流れるトークンとして、述語の引数を設定する。
以上より本文中の図10(b)に示すペトリネット図が得られる。