ICOT Technical Memorandum: TM-0881

TM-0881

並列プログラムの動作解析の一方式

長谷川春朗，安藤津芳，
前田賢一(神)

April, 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

講演番号 401

並列プログラムの動作解析の一方式

沖電気工業(株)[+)] ○長谷川晴朗、安藤津芳、前田賢一

A Method of Analyzing Performance of Parallel Program

Haruo HASEGAWA, Tsuyoshi ANDO and Ken'ichi MAEDA

Oki Electric Industry Co., Ltd.

Abstract: *This paper describes a method of analyzing a program written in a parallel logic programming language GHC(Gurarded Horn Clauses) by utilizing Petri Nets. Two methods of converting program written by GHC into Petri Nets are proposed in order to analyze correctness and parallelism of programs and they are compared with each other. The former is more familiar to reduction of predicate logic than the latter. But, the latter is superior, because it is effective for analysis of the execution of programs on processors. Then, by adopting the latter, this paper describes the method of verifying software and analyzing parallelism, giving several examples.*

1. はじめに

コンピュータの高速実行の解として並列化ということが素直に考えられるが、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。それを解決するべく、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。

一方で、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。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。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目的及び基本用語について触れる。3章でGHCの概要を説明し、次章で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。5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6章はまとめである。

計測自動制御学会 第4回離散事象システム研究会
平成2年4月6日 沖縄

+)〒108 東京都港区芝浦4-11-22
電話:03-454-2111 内5240
Fax: 03-452-4124
E-mail: hasegawa@sot.telcom.oki.co.jp

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)から成る[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おうとする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。

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

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。

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

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 $M=M_0$であって次式が成立するベクトルxをTインバリアントと呼ぶ。

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

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、含意である。)

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

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、$n \geqq 1$の時、次式のようになりこれをホーン節と呼び、特に$n=1$の時これを確定節という。

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

$m \geqq 1$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。質問を表すのに使用される。

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。

$$H :- G_1, \cdots, G_n \mid B_1, \cdots, B_m. \quad n, m \geqq 0 \qquad (5)$$

ヘッド　ガードゴール　ボディゴール

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ユニフィケーションを行う。これらはそれぞれ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(ユニフィケーション)、及び$G_1, \cdots, G_n$の実行を行う必要がある。但し、そ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。また、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Fig.1に示す。Fig.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、Fig.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。

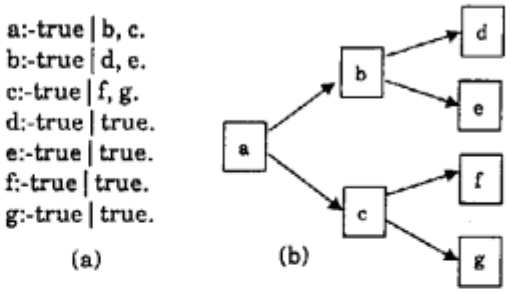


Fig.1 GHCプログラム例とそのゴール木

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Fig.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。

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Fig.2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) 分割するGHCプログラム

Fig. 2 appendプログラム

Fig.2(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。Fig.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']にバインドしてしまうからである。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。Fig2.(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。Fig.2(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Fig2.(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、一旦変数を(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. ペトリネットとの対応

4.1 変換ルール GHCのプログラムとペトリネットとの対応をTable 1に示す。

Table 1 GHCプログラムとの変換ルール

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

一階述語論理の節をトランジションに、述語をプレースに、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。

簡単な例をFig.3~5に示す。Fig.3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。Fig.4(a)は、それをTable 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、Fig.4(b)は、Table 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

```
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)
```

Fig.3 プログラム例

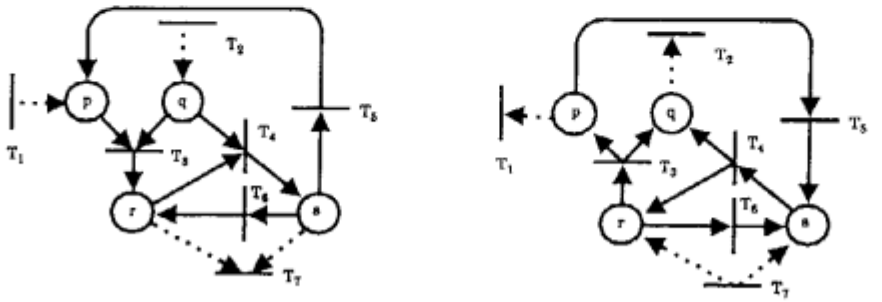
$$A = \begin{array}{c} \\ T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \\ T_6 \\ T_7 \end{array} \begin{array}{c} \begin{array}{cccc} p & q & r & s \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & -1 & 1 & 0 \\ 0 & -1 & -1 & 1 \\ 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & -1 \end{pmatrix}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

(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

Fig.5 Fig.4のペトリネットの接続行列



(a)変換ルール(a)によるペトリネット　(b)変換ルール(b)によるペトリネット

Fig.4. Fig.3のプログラムのペトリネット

ペトリネットの接続行列はFig.5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。例えば、Fig.4(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(Fig.4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、初期マーキングも目的マーキングも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。

4.2 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。

5. 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。

5.1 論理的な誤りの検出

Table 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、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・解の導出を高速に行うことができる。例えば、Fig.4(b)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、次式より$x = [2\ 3\ 2\ 1\ 0\ 0\ 1]$となる。

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

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、Fig.3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。

Fig. 6 (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。Fig. 6 (b)ではアーク線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T_3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$x = [1\ 1\ 1\ 0\ 1\ 1\ 1]^T$となる。つまりこのプログラムでは、$T_3$に相当する節が冗長なことがわかる。

5.2 **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幾つか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、Fig. 4 (b)のネットでは、Tインバリアントから各ト

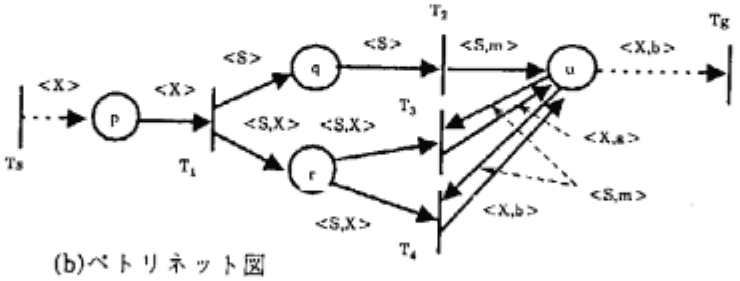
```
p(X) :- true | q(S), r(S,X).  (T1)
q(S) :- true | S = m.         (T2)
r(S, X) :- S = m | X = a.     (T3)
r(S, X) :- S = m | X = b.     (T4)

スタート: p(X)                (Ts)
ゴール: X=b                   (Tg)
```

(a) GHCプログラム

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

(c) 接続行列



(b)ペトリネット図

Fig. 6 引数のあるGHCプログラムとペトリネット

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Table 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。

Table 2 Fig. 4 (b)の各ステップにおける並列度

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	u
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、Fig. 4(a)のペトリネットは、Fig. 4(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Fig. 4(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。

6. 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った。簡単な例についてバグの検出が可能であることを示した。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。日頃有益な御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, 22-7, 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, 15-2(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, 209/260(1989)
7)中島秀之:Prolog、産業図書(1983)