4J−10

並行プログラムからその拡張ペトリネットモデルへの変換

後藤正智　程京徳　牛島和夫
(九州大学 工学部 情報工学科)

1. はじめに

並行処理プログラムのデバッグは一般的に難しい。何故なら、処理の流れが一意に定まらないからである。その上、逐次処理プログラムにはないデッドロック、ライブロックが発生してデバッグがより困難となっている。そのために何か並行処理のデバッグに役立つツールが必要となってくる。

デバッグには静的解析方法と動的解析方法がある。静的解析方法は対象となるプログラムをその数学的モデルに変換し、それを解析することによってエラーなどを発見することである。動的解析方法とは実際にプログラムを動作させて、その振舞いを監視し、内部に発生したエラーを発見することである。

本研究では静的解析に役立つツールとしてＡｄａ並行プログラムからその数学モデル拡張ペトリネットへの変換ツールを作成する。

以下２章に今回使用する拡張ペトリネットの概要を述べ、３章にＡｄａ並行プログラムの拡張ペトリネットモデルの例を挙げる。４章では拡張ペトリネットの内部表現法について述べ、５章にＡｄａ並行プログラムからその拡張ペトリネットへの変換方法を記し、６章にまとめをしるす。

2. 拡張ペトリネットについて

2.1 ペトリネットとは

ペトリネットとは、並行動作システムを研究する上での数学的道具の一つである。ペトリネットを用いて、並行システムをモデル化し、これを解析すれば、モデル化した並行システムの挙動について重要な情報を得ることができる。[1]

ペトリネットは以下の要素で構成されている。

- プレースの集合
- トランジションの集合
- 入力関数
- 出力関数
- マーキング

モデル化に際してはプレースは状態をトランジションは事象と条件をあらわしている。入力関数と出力関数はトランジションとプレースとを関係づけるものである。ペトリネットはトランジションとプレースが交互に現れるが、トランジションに着目すると、この入力となるプレースを入力プレース、出力となるプレースを出力プレースという。マーキングとは、プレースにトークン（ペトリネットの実行を定義するのに用いられるもの）を割り当てることである。トークンの位置および数はペトリネットの実行に伴い変化する。次節でシステムを解析するときに必要なペトリネットの実行について述べる。

2.2 ペトリネットの実行

ペトリネットの実行とはトランジションを発火させることである。発火とは入力プレースからトークンを取り去り、新しいトークンを生成し、出力プレースに分配することである。発火するためには発火可能でなければならない。発火可能とはそのトランジションへの全ての入力プレースがそのプレースからトランジションへのアーク数以上のトークンをもっているときである。アークとはペトリネットグラフにおいてプレースとトランジションを結合するもので、多重入出力はアークの本数であらわされる。トランジションの発火により入力プレースからアーク数だけのトークンが取り去られ、出力プレースにトランジションから出力プレースへのアーク数だけのトークンが投入される。[1]

2.3 拡張ペトリネット

ペトリネットではＡｄａ並行プログラムのselect文をモデル化できない。そこでそのselect文をモデル化できるように抑止アークを付加した拡張ペトリネットを使用する。抑止アークは発火規則を次のように変更する。通常の入力プレースの全てにトークンがあり、抑止入力プレースのすべてにトークンがないトランジションは発火可能であるとし、発火したトランジションは通常入力プレースの全てからトークンを取り去る。[1]

3. Ａｄａ並行プログラムの拡張ペトリネットモデル

我々はすでに拡張ペトリネットを用いてＡｄａ並行プログラムをモデル化した。[2]

ここに簡単なＡｄａプログラム（ fig.1 ）を例にとりその拡張ペトリネットモデルのグラフ表現（ fig.2 ）をしるす。このプログラムは、タスクを２つ発生し、各々のタスクが互いに受付状態に入りその後相手のタスクを呼び出してデッドロックを発生させるプログラムである。

```
procedure CAW is
  task T1 is
    entry E1;
  end T1;
  task T2 is
    entry E2;
  end T2;
  task body T1 is
  begin
    accept E1;
    T2.E2;
  end T1;
  task body T2 is
  begin
    accept E2;
    T1.E1;
  end T2;
begin
  null;
end CAW;
```

fig.1 例題プログラム

→ 抑止アーク
— アーク
— トランジション
○ プレース

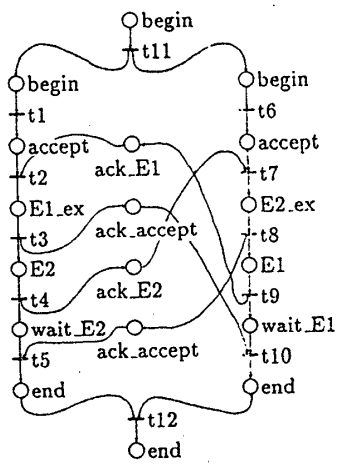

fig.2 fig.1の拡張ペトリネットグラフ

4. 拡張ペトリネットの内部表現法

前章でＡｄａプログラムの拡張ペトリネットモデルを紹介した。この章では、この拡張ペトリネットを変換ツール内部でどのように表現

Transforming Concurrent Programs into Their Extended Petrinet Models
Masatomo GOTOU, Jingde CHENG, Kazuo USHIJIMA
Department of Computer Science and Communication Engineering, Kyushu University.

するかを示す。拡張ペトリネットを数学的に解析する際や、この拡張ペトリネットをfig.2のようなグラフとしてウインドウ上に表示するプログラムを作成する際の容易さを考慮して内部表現を決定した。

拡張ペトリネットの内部表現を次のようにT（トランジション）の集合として表現する。

T （アーク数、入力プレースの名前、行番号），・・・
→ （アーク数、出力プレースの名前、行番号），・・・

つまり拡張ペトリネットの要素の1つであるトランジションに着目し他のすべての情報を盛り込み、拡張ペトリネットをトランジションの系列として表現することにした。ここで矢印より前の括弧でくくられた要素は入力プレースを、矢印より後の括弧でくくられた要素は出力プレースを表している。アーク数は多重入出力の多重度を表すためのものであり、行番号はソースプログラム中において同じ名前の命令が別の場所で繰り返し行なわれたときに、それらが区別できるようにするために必要なものである。

抑止アークの表現方法は、アーク数が 0 のときとする。つまり、このプレースからトランジションへのアークが抑止アークとなる。

ここでfig.1のプログラムの拡張ペトリネットの内部表現をfig.3に示す。

```
t1  (1,begin,9) → (1,accept,10)
t2  (1,accept,10),(1,ack_T1.E1,16)
      → (1,T1.E1_ex,10)
t3  (1,T1.E1_ex,10)
      → (1,ack_accept,10),(1,T2.E2,11)
t4  (1,T2.E2,11) → (1,ack_T2.E2,11)
                   (1,wait_T2.E2,11)
t5  (1,wait_T2.E2,11),(1,ack_accept,15)
      → (1,end,12)
t6  (1,begin,14) → (1,accept,15)
t7  (1,accept,15),(1,ack_T2.E2,11)
      → (1,T2.E2_ex,15)
t8  (1,T2.E2_ex,15)
      → (1,ack_accept,15),(1,T1.E1,16)
t9  (1,T1.E1,16) → (1,ack_T1.E1,16)
                   (1,wait_T1.E1,16)
t10 (1,wait_T1.E1,16),(1,ack_accept,10)
      → (1,end,17)
t11 (1,begin,18) → (1,begin,9),(1,begin,14)
t12 (1,end,12),(1,end,17) → (1,end,20)
```

fig.3　fig.2の拡張ペトリネットの内部表現

5. 拡張ペトリネットモデルへの変換方法

拡張ペトリネットへの変換方法はShatz氏らの論文[3]を元にしてそれを変更・改良して作成した。以下の通り。

第一段階：

プレースとなり得る文をソースプログラムより抽出しプレース列を作成する。ソースプログラムの大部分はプレースとなり得るが宣言部や注釈文はプレースとなり得ないので、この段階でこれらの文は取り除かれる。

第二段階：

どのタスクがどのタスクを呼び出しているかを示すＥＣＴ（Entry Call Table）を作成する。ここにentry文,accept文への変換の際に必要な情報を与えており、このテーブルを見ることによって拡張ペトリネットへの変換を行なう。前述のプログラムのＥＣＴは次のようになる。

Calling Task	Called Task	Entry Name	Caller ID
T 1	T 2	E 1	11
T 2	T 1	E 2	16

第三段階：

if文やcase文,select文,loop文の範囲を示したＥＳＮＴ（Ending Statement Number Table）を作成する。このテーブルは上述の文に関してＥＣＴと同じような働きをする。前述のプログラムにはＥＳＮＴは必要なかったが、if文が10行から17行まであったとするとＥＳＮＴは次のようになる。

Statement	Begin ID	End ID
if	10	17

第四段階：

プレース列を行番号が小さい順に検索していき、procedure,task,function ごとにトランジションを作成していく。特殊な文（entry，accept など）以外のプレースは自分を入力プレースとし、次のプレースを出力プレースとしたトランジションを生成する。entry文,accept文のときはＥＣＴを参照し次のように変換される。

entry
t　(1,entry,ID) → (1,ack_entry,ID)，(1,wait_entry,ID)
t　(1,wait_entry,ID)，(1,ack_accept,ID) → 次のプレース

accept
t　(1,accept,ID)，(1,ack_entry,ID) → (1,entry_ex,ID)
t　(1,entry_ex,ID) → (1,ack_accept,ID)，次のプレース

6. おわりに

現在のツールでは単一procedureで、タスクの中にタスクを入れ子として持たないようなプログラムを変換の対象としている。

今後の課題として、次のようなものが挙げられる。

- いかなるＡｄａプログラムにも対応できるツールの作成
- ウインドウなどにペトリネットグラフを表示するツールの作成

参考文献

[1] J.L. ピーターソン: “ペトリネット入門 - 情報システムのモデル化”共立出版株式会社

[2] J.Cheng and K.Ushijima: “Analyzing Ada Tasking Deadloks and Livelocks Using Extended Petri Nets,” Proc. Ada-Europe 10th Annual International Conference. LNCS, Vol.499, pp.125-146, 1991.

[3] Sol M.Shatz et al. : “Design and Implementation of a Petri Net Based Toolkit for Ada Tasking Analysis,” IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS. Vol.1, No.4, pp.424-441, 1990.