記　号　処　理 72－3
（1994．1．14）

PaiLispを用いたペトリネットの解析と実行

川本 真一　伊藤 貴康

東北大学 情報科学研究科

ペトリネットの解析と実行を行うSATPN(Simulation and Analysis Tool for Petri Net)というペトリネットツールを並列Lisp言語PaiLispを用いて試作した。SATPNは、ペトリネットのビジュアルエディット、シミュレーション、可達グラフの自動生成の各機能を持つ。可達グラフの自動生成及びペトリネットの実行はPaiLispの並列構文を用いて並列化されている。ツール全体としては、PaiLispの子プロセスとしてwishを動作させることによって、GUI(Graphical User Interface)を実現している。本稿では、試作したシステムとその使用例について述べる。

Analysis and Execution of Petri net using PaiLisp

Shinichi KAWAMOTO　Takayasu ITO

Depertment of Computer and Mathematical Sciences
Graduate School of Information Sciences
Tohoku University

We implemented Petri net tool named SATPN (Simulation and Analysis Tool for Petri Net) using PaiLisp. SATPN support to edit and execute Petri nets and generate reachability graphs.
Generation of reachability graphs are parallelized using PaiLisp parallel construct. Execution of Petri nets are achieved by executing PaiLisp processes associated with transitions in parallel. GUI(Graphical User Interface) is also implemented with using wish(window shell) from PaiLisp. This paper reports how SATPN is implemented using PaiLisp.

1 はじめに

並列モデルの1つであるペトリネットは、同時進行性の表現が可能、視覚的でわかりやすい、解析手法があるなどの優れた特徴があり、並列システムのモデル化や解析に使用されている[1]。しかし、対象とするシステムの規模が大きくなると、手作業によるペトリネットの実行や解析は困難となる。そこで、ペトリネットの解析や実行を計算機上で行うペトリネットツールと呼ばれるサポートシステムが試作されている[2)3)]。並列Lisp言語PaiLisp[4]を用いてSATPN(Simulation and Analysis Tool for Petri Net)と呼ばれるペトリネットツールを試作したので、その概要と使用例について報告する。

2 SATPNの概要

SATPNは、Net Editor、Net Analyzer、Net Simulatorの3つのサブシステムから構成される。

Net Editor グラフィカルなペトリネットを見たままの形で対話的に入力、修正する機能を提供する。

Net Simulator トランジションを次々と発火させトークンを移動し、その様子を観測し、それをウィンドウ上に描かれたペトリネット上でアニメーションする。

Net Analyzer ペトリネットの可達グラフを生成し、それをグラフィカルに表示する。

2.1 SATPNの構成

ペトリネットは図的なモデルであり、その入力や表示などはグラフィカルに行う必要がある。PaiLispから直接X Windowを扱うことができないので、Tcl/Tk[6]インタプリタwish上で入力や表示プログラムを動作させ、PaiLispとwishが通信することによってGUI(Graphical User Interface)を実現している。SATPNのシステム構成は図1のとうりである。

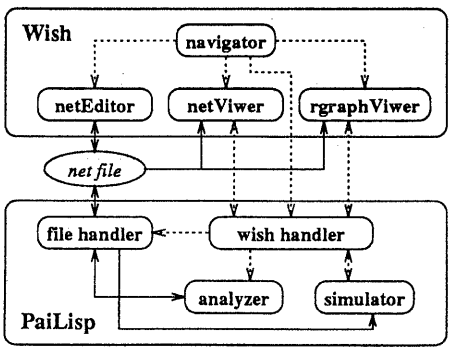

図1: SATPNの構成

図の実線はデータの流れを示しており、点線は命令の流れを示している。

netEditor ペトリネットの入力や修正のためのビジュアルエディタ。

netViwer ペトリネットをウィンドウ上に表示し発火動作をアニメーションする。

rgraphViwer 可達グラフをウィンドウ上に表示する。

navigator SATPNの操作をボタンやメニューによって行うためのGUI。

simulator ペトリネットの実行を行う。各トランジションを1つのPaiLispプロセスとして並列に動作させ、その様子を別のプロセスで監視する。

analyzer 可達グラフを生成する。並列化によって処理を高速化している。

wish handler PaiLispとwishとの通信を制御。

file handler ファイルの入出力を行う。

Net Editorの機能は、netEditorにより実現されており、Net Simulatorの機能は、simulatorとnetViwerによって実現されている。また、Net Analyzerの機能はanalyzerとrgraphViwerによって実現されている。

3 PaiLispとペトリネット

SATPNはPaiLispを用いたペトリネットの解析と実行を行うツールであるので、PaiLispとペトリネットの解析と実行について簡単に説明しておく。

3.1 PaiLisp

PaiLispは、共有メモリアーキテクチャを前提としてSchemeに様々な並列構文を導入して設計された並列Lisp言語で、現在その処理系は8台のプロセッサを持つAlliant FX/80上で稼働している[4]。PaiLispの主な並列構文を以下に示す。

(spawn e) 式eを評価する子プロセスを生成し、親プロセスはそれと並列に実行を続ける。

(suspend) 適用したプロセスの実行を停止させる。

(exlambda $(x_1 \cdots x_m)$ $e_1 \cdots e_n$**)** 排他的関数クロージャを作る。PaiLispプロセス間の相互排除を行う。

(call/cc *proc***)** Schemeのcontinuationを並列拡張したP-continuationを生成し、それを引数にして関数*proc*の適用を行う。

(par $e_1 \cdots e_n$**)** 式$e_1, \cdots, e_n$の評価を並列に行う。

(pcall f $e_1 \cdots e_n$**)** 式$e_1, \cdots, e_n$の評価を並列に行った後、関数fを適用する。

(future e**)** future値と呼ばれる式eの仮の値を即座に返し、eの値を計算する子プロセスを生成する。

3.2 ペトリネットの解析と実行

ペトリネットによるシステムのモデル化、解析及び実行について例を用いて説明しておく。

次のような簡単な通信システムを考える。

『データの送受信を行うシステムで、送信側はデータの送信 (sd) とそのデータに対する ACK の受信 (sa) を繰返し行う。受信側は、データの受信 (rd)、そのデータに対する ACK の送信 (ra)、受信準備処理 (pro) を繰返し行う。伝送路はデータは消失 (drop) する可能性があるが、ACK は消失しない。』

ペトリネットは、システムを条件 (プレース、○) と事象 (トランジション、|) によって表し、関係のある条件と事象をアーク (→) で結ぶ。現在成立している条件はプレースの中にトークン (•) を入れる。システムの状態はトークンの分布によって表され、マーキングと呼ばれる。データの送信 (sd)、ACK の受信 (sa)、データの受信 (rd)、ACK の送信 (ra)、受信準備処理 (pro) 及びデータ消失 (drop) はトランジションとして表され、それらの動作の起る前提条件と後提条件をプレースとして表現する。初期マーキングを送信可能かつ受信可能であるとすれば、この通信システムを表すペトリネットモデルは図2となる。このようにモデル化されたペトリネットの解析や実行を行う事によってシステムの解析やシミュレーションが行える。

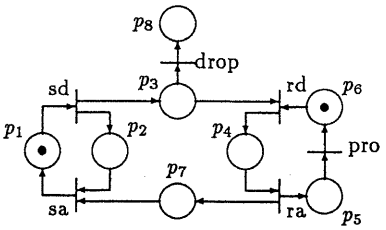

図 2: 簡単な通信システムのペトリネットモデル

マーキングを、$p_1 \sim p_8$のトークン数をこの順に並べたベクトル $(to_1, to_2, \cdots, to_8)$ として表せば、図2の初期マーキングは $(1,0,0,0,0,1,0,0)$ となる。事象の生起は、アクションを表すトランジションの発火によって表される。トランジションはその入力プレースにトークンが存在すれば発火でき、発火すると入力プレースからトークンが取り去られ出力プレースにトークンが入れられる。図2の場合、初期マーキングから発火可能なのは sd で、sd が発火すると $(0,1,1,0,0,1,0,0)$ という新しいマーキングに遷移する。これは、送信側からデータが送信されたことを示す。

初期マーキングからマーキングの遷移を求め、それを描いたグラフが可達グラフである。図2のペトリネットの可達グラフは図3となる。

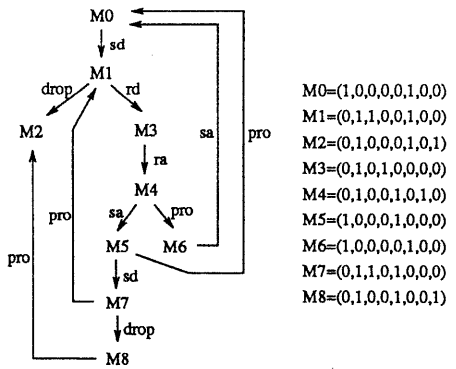

図 3: 図2のペトリネットの可達グラフ

可達グラフにおいて、そのノードから遷移のないノードは終端ノードと呼ばれ、ペトリネットの動作がそのノードの示すマーキングでデッドロックになることを表している。図3の場合は M_2 が終端ノードで、このノードに至る系列には必ず drop 動作が含まれており、データが消失すればこの通信システムがデッドロックになることを示している。

ペトリネットではアクションの並列動作 (すなわちトランジションの同時発火) が可能である。Net Simulator はペトリネットのトランジションを次々と自動的に発火させてマーキングを更新すると共に、その振舞いを観測し動作の様子をアニメーションとして表示させることにより、システムの並列動作をユーザが知る事ができるようにするツールである。Net Analyzer は、図3のような可達グラフを生成するもので、これを見る事によってユーザがデッドロックの有無やそれを引き起すアクション系列などを視覚的に解析できるようにするツールである。

4 Net Analyzer

可達グラフを逐次的に求める方法を図3を用いながら述べる。まず、初期マーキング M_0 からどのトランジションが発火可能か調べる。sd,drop,rd,…について順次発火判定し、最終的に sd のみ発火可能と判る。そこで sd を発火させ新しいマーキング M_1 を得る。M_1 に対して同様に発火判定を行い、発火可能トランジションが drop と rd であることが判るので、これらを発火させて M_2、M_3 を得る。新しいノードが複数できた場合は、その中

の1つを選んで処理を続け、もうこれ以上新しいノードが生成されなくなったら、べつの未処理ノードを選んで処理を続ける。M_2, M_3という2つのノードの対して、まずM_2を選ぶ。M_2からは発火可能なトランジションが存在しないので、M_2は終了しM_3に移る。M_3に対して処理を続けM_5まで進んだとする。新しいマーキングはM_0, M_7であり、まずM_0を選ぶが、これは既に初期ノードとしてグラフ中に存在するのでM_0に関する処理はそこで終り、M_7について続ける。以上のような操作を行い、処理すべき新しいノードが生成されなくなった時点で図3のような可達グラフが生成され完了する。

4.1 可達グラフの生成の並列化

可達グラフの生成コストは、プレースやトランジションの数に比例して指数関数的に増大するので、生成時間の短縮を行うために並列化を考える。最も簡単なのは、ノードの枝分れが生じたらそれぞれを並列に処理することである。図3の例では、M_1からM_2以降の処理(図4A)とM_3以降の処理(図4B)を並列に行う。M_3はさらに、M_4からM_5以降の処理(図4C)とM_6以降の処理(図4D)を並列に行う。

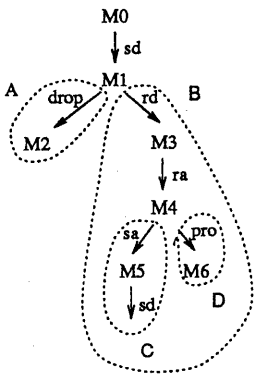

図 4: 可達グラフの生成の並列化

4.2 可達グラフ生成の PaiLisp による実現

並列化された可達グラフ生成アルゴリズムの PaiLisp による実現について述べる。

あるペトリネットにおいてマーキングがMであるときトランジションtが発火しマーキングがM'に変化したことを$M \xrightarrow{t} M'$と表しマーキングのプリミティブ遷移と呼ぶ。可達グラフは、初期マーキングM_0から始まるすべてのマーキングのプリミティブ遷移$M_0 \xrightarrow{t_1} M_1, M_1 \xrightarrow{t_2} M_2, \cdots$を求めそれをビジュアルに表示したものである。このすべてのプリミティブ遷移を先の並列アルゴリズムに従って求める関数を pmake-trans とすると、それは初期マーキングを引数として取り、プリミティブ遷移$M_i \xrightarrow{t_k} M_j$を$((M_i.M_j).t_k)$として表現したものを要素とするリストを返す。

先に示したとうり、可達グラフの並列化はあるマーキングMから枝分れがあると、そのそれぞれを並列に実行する。あるマーキングのプリミティブ遷移をすべての発火可能トランジションに対し並列に求める処理を1ステップの並列処理と呼べば、それを初期マーキングから生成されるすべてのマーキングに対し再帰的かつ並列に適用することによって実現される。そこで、まず1ステップの並列処理を実現する。

4.2.1 1ステップの並列処理

1ステップの並列処理は、与えられたマーキングに対し発火可能なトランジションをそれぞれ並列に発火させて新しいマーキングを生成し、それを値として返す。例えば、図3の例でマーキングM_4に1ステップの並列処理を適用すると図5のように処理され、M_5とM_6が並列に生成される。ただし、図のFは発火可能性の判定、Nは次段マーキングの計算処理である。

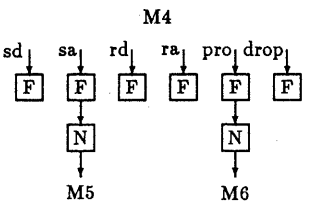

図 5: 1ステップの並列処理の実現法

1ステップの並列処理は、その計算結果としてマーキングのリストを返す。しかし、本来求めるべきものはプリミティブ遷移の集合である。そこで、1ステップの並列処理を実行することによって並列に得られるプリミティブ遷移は、排他キューに入れるものとする。

1ステップの並列処理はPaiLispの関数pmake-a-transとして実現されている。

```
1: (define (pmake-a-trans M trl)
2:  (map
3:   (lambda (tr)
4:    (future
5:     (if (firable? tr)
6:         (let ((Mn (next_marking M)))
```

```
7:          (add ans (cons (cons M Mn)tr))
8:          (if (noexist? Mn que2) Mn)))))
9:   trl))
```

図5のような、トランジション毎の処理の並列化はmapとfuture構文によって表されている。futureを用いた理由は、発火判定などの一連の処理(5-8行目)の結果として得られるマーキングを値として返す必要があり、またpmake-a-transの終了時にすべての結果が求まっている必要はないからである。さて5-8行目の処理の結果としては、もしそのトランジションが発火不能であったり可能であっても得られたマーキングが既にグラフ中に存在するものであった場合は、そのマーキングを値として返す必要がないためダミー(#f)を返すようにして並列実行している。

4.2.2 pmake-trans関数の実現

pmake-trans関数すなわち初期マーキングから始まるすべてのプリミティブ遷移を求める手続きは、pmake-a-transを初期マーキングから得られるすべてのマーキングに対して、再帰的かつ並列に適用すれば良い。これは次のように実現されている。

```
1: (define (pmake-trans M)
2:   (apply-trans (pmake-a-trans M trl)))
3: (define (apply-trans Ms)
4:  (cond ((null? Ms) #t)
5:        ((eq? (car Ms) #f)
6:         (pmake-trans (cdr Ms)))
7:        (else
8:         (par (pmake-trans (car Ms))
9:              (apply-trans (cdr Ms))))))
```

pmake-transは、マーキングMに対し1ステップの並列処理pmake-a-transを適用し、その結果に、apply-transを適用する。apply-transは、pmake-a-transの返したダミー(#f)ではない要素つまり新しいマーキングの各々に対してpmake-transを並列に適用する。並列適用はparによって表現している(PaiLispのpar構文が有効に使える)。apply-transは再帰によって実現されているが、例えば、マーキングM_iにpmake-a-transを適用した結果が$(M_j\ M_k\ M_l)$であったとすれば、実質的に実行されるのは

```
1: (par (pmake-trans Mj)
2:      (par (pmake-trans Mk)
3:           (pmake-trans Ml)))
```

という式である。

4.3 並列化の評価

並列化の評価として、図6の3人の哲学者を表すペトリネットをシステムに入力しその可達木グラフ(図7)を生成し、pmake-connection関数の実行時間を計測した。

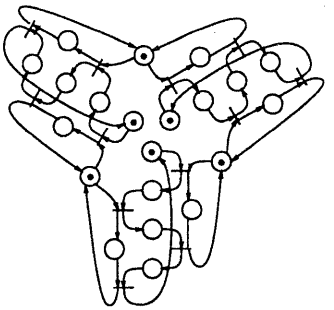

図 6: 3人の哲学者のペトリネットモデル

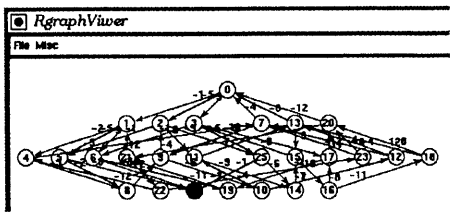

図 7: 図6のペトリネットの可達グラフ

実験は、並列化を行わない場合(A)、pmake-a-transのみ並列化(B)、apply-transのみ並列化(C)、両方とも並列化した場合(D)の4通りについて行った。Aはプロセサ数を2台に、B,C,Dは1台から8台にして行った。結果は表1のとうりである。単位はsecである。

表 1: 実行結果

PN	A	B	C	D
1		10.29	9.28	10.41
2	9.22	6.21	4.91	5.29
4		5.08	3.10	2.83
6		5.23	3.36	2.23
8		5.38	3.34	2.14

BやCは余り高速化されないが、Dの場合は最高で約5倍と良い結果が得られている。

5 Net Simulator

Net Simulator はペトリネットのトランジションを次々と自動的に発火させマーキングを更新すると共に、その振舞いを監視し動作の様子をアニメーションとして表示する。本稿ではトランジションを発火させるだけでなく、動作の様子をユーザに提供することも含めてペトリネットの実行またはシミュレーションと呼ぶ。Net Simulator は図1の simulator と netViwer から構成される。simulator は PaiLisp 関数 `simulate` として実現されている。

(`simulate` *net n*)
: *net* のトランジションを発火させマーキングを更新すると共に、その振舞いを監視し変化があればその情報を netViwer に送る。トランジションの発火回数が *n* になると停止する。netViwer 上の実行開始ボタンが押されるとこの関数が呼び出される。実際にペトリネットを実行することから、これを実行エンジンと呼ぶ。

`netViwer`
: ペトリネットを表示するためのウィンドウを生成し、PaiLisp からの次のコマンドを受け取って動作する。

<`load` *filename*> ペトリネットファイル *filename* を読み込みウィンドウに表示する。

<`token` *pid mode*> プレース *pid* の中に *mode* に従ってトークン (正の整数) を表示する。

<`fire` *tid*> 発火動作のアニメーションとして、トランジション *tid* を一瞬反転させる。

以下ではペトリネットの実行エンジンについて述べる。

5.1 ペトリネットの実行モデル

従来のペトリネットの実行を自動的に行うシステムの多くは、実行を制御するシステムがペトリネットとは完全に分離して存在し、それがペトリネット全体を見て次に発火可能なトランジションを判別し、その中からいくつかをランダムに選んで発火したものとしみなし、マーキングを更新するという処理の繰返しによって実現されていた[5]。

実行エンジンは、ペトリネットのプレース及びそのプレースにおけるのトークンの個数を PaiLisp の大域環境に、トランジションを、条件を満せばその入力及び出力プレースに相当する大域環境を書き換えるという PaiLisp プロセスにそれぞれ対応させる。

例えば、図8(a) のペトリネットは、同図 (b) のように表される。プレース p_i はその識別子 i とトークン数とのペアを要素とする大域環境上のリストに、トランジション t_j は PaiLisp プロセス T_j に対応する。PaiLisp プロセス T_1,T_2 は並列に実行され、発火可能条件が満足されている T_1 が図8(b) の矢印のように大域環境にアクセスし、その結果として図 (d) のようになる。これは図 (a) のペトリネットのトランジション t_1 が発火して、図 (c) の様になったことを示している。

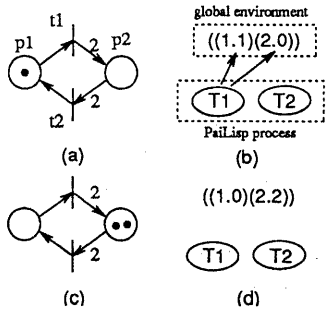

図 8: ペトリネットの PaiLisp 上での実行モデル

5.2 実行エンジンの構成

実行エンジンは、ペトリネットのトランジションを発火させトークンを移動すると共にその様子を監視する。トランジションの発火はそれに対応した PaiLisp が scheduler の制御の下に行う。一方、トランジションの発火のの監視は、トランジションに対応した PaiLisp プロセスと並列に動作する inspector によって行われる。実行エンジンの構成は図9のとうりである。

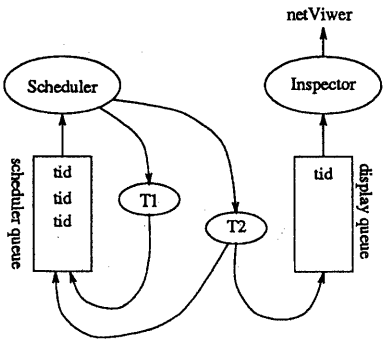

図 9: 実行エンジンの構成

scheduler は scheduler キューからトランジション *tid* を取り出し、それに対応した PaiLisp プロセスを生成する。生成された PaiLisp プロセスは発火判定を行い、発火可能であればその *tid* を display キューに入れ、環境を更新する。発火不可能であるか、発火の処理が終了

した場合には、*tid* を scheduler キューに入れ終了する。inspector は display キューに *tid* が入って来れば、その *tid* と現在のマーキングを netViwer に送る。

トランジション *tid* に対応した PaiLisp プロセスは、引数としてその *tid* などを取る関数 `transition` として実現されており、入力プレースに相当する大域環境を見て発火可能かどうか判定し、可能であれば発火して入力及び出力プレースに相当する大域環境を書き換える。`transition` 関数はトランジションの数だけ並列に実行されるので、大域環境に対する同時アクセスを禁止するため、各プレース毎にバイナリセマフォを用意し、これを確保してから処理を行っている。

6 Net Editor とユーザインタフェース

wish 上に実現された Net Editor や GUI について述べる。

6.1 Net Editor

Net Editor は、ビジュアルなペトリネットを見たままの形で対話的に入力修正するための機能を提供する。マウスによって、プレース、トランジション、アーク、トークンの配置、修正及び削除が行える。入力、修正したペトリネットはファイルにセーブされ、Net Simulator や Net Analyzer がこれを利用する。このようなシステムを実現するには、X Window のプログラムを作成する機能が必要であるが、現在のところ PaiLisp にはまだこの機能はない。しかし、PaiLisp は UNIX OS 上で動作するプログラムを子プロセスとして生成し、標準入出力を用いてそのプロセスと通信する機能を持つ。SATPN はこの機能を利用し、OS 上で Tcl/Tk[6] インタプリタ wish を動作させ、それと PaiLisp とが通信することによって GUI を実現している。Net Editor は、wish プログラム netEditor によって実現されている。

SATPN を構成する wish プログラムは、Net Editor を実現する netEditor、ペトリネットの表示を行う netViwer、可達グラフを表示する rgraphViwer および navigator の 4 つである (図 1)。navigator は、システムへのペトリネットの読み込みや実行の開始などの SATPN 上の操作を、PaiLisp インタプリタのトップレベルにおける関数実行の代りに、メニューやボタン操作によってできるようにするものである。

6.2 PaiLisp と wish との通信

PaiLisp から wish の起動は次の式を評価する。

```
1: (*sys:exec ''/usr/bin/wish'' eventh)
```

値として wish の標準入出力に継ったポート (*WishPort) が返る。PaiLisp から wish へコマンドを送る場合は、*WishPort に対し `write` 関数を適用する。一方、navigator や netViwer 上のボタンやメニューの呼び出し (イベント) に伴なって、wish から PaiLisp へコマンドが送られるが、これは`*sys:exec` の第2引数に指定された引数無し関数 `eventh` によって処理される。`eventh` は送られてきたコマンドを判別し、コマンドそれぞれに対応した PaiLisp の処理関数を呼び出す。コマンドはペトリネットファイルを読み込む Load、ペトリネットの実行を始める SimS などである。Load コマンドに対する処理は PaiLisp の load 関数を用いるため、トップレベルプロセスで実行されなければならない。これは、システムの起動時にトップレベルプロセスで次のプログラムを実行し、load を実行する P-Contiunation を toploaad に取っておく。

```
1: (define toploaad '())
2: (let ((filename '#f))
3:  (set! fn (call/cc
4:             (lambda (c)
5:              (set! toploaad c) #f)))
6:  (if fn
7:      (load fn)))
```

PaiLisp の P-continuation はそれを生成したプロセスが実行を行うので、`eventh` がどのプロセッサで実行されていても、toploaad の適用によりトップレベルで load が実行される。

以上のように、子プロセスとして wish を用いることによって C 言語などで作成したものと同等の機能を有する GUI を実現している。

7 SATPN の使用例

SATPN の使用例を示す。対象を図 2のペトリネットとする。ます、システムを起動すると Navigator ウィンドウが表れる (図 11右上)。Navigator ウィンドウから Editor ボタンを押して、Net Editor を起動する (図 10)。画面の左上のボタンは左から、プレース (○)、トランジション (□)、アーク (→)、トークン (•) をそれぞれ入力するモードに入るためのものである。まず○ボタンを押しプレース入力モードにし、$p_1 \sim p_8$ を描く。次にトランジションとアークを書き、最後にトークンを入れる (図 10)。

セーブは File メニューから Save を選びファイル名を指定する。Net Editor を終了し Navigator の Simulator と Analyzer ボタンを押して NetViwer と RgraphViwer を起動する。Navigator の File メニューからセーブし

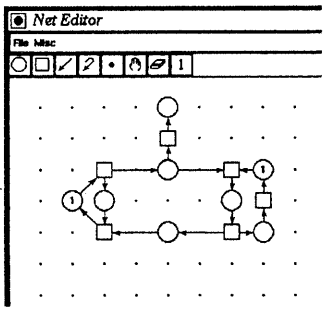

図 10: Net Editor によってペトリネットを入力

たファイル名を選択すると、そのペトリネットがシステムに読み込まれ、可達グラフが生成され、ペトリネットと可達グラフがそれぞれ NetViwer と RgraphViwer 上に表示される (図 11)。

図11の上のウィンドウがNavigator、右がRgraphViwer、左が NetViwer、下が PaiLisp のトップレベルである。表示された可達グラフのノードのうち、縞模様のノードは終端ノードを表している。ユーザはこのグラフを用いて可達性や活性を調べる。可達グラフの各ノードの表示位置は自動的に計算されるため、図3のような判りやすいグラフが得られるとは限らない。自動的に判りやすい配置を計算することは非常に困難なので、得られた可達グラフをユーザが手作業で修正するためのエディタが必要であろう。

ペトリネットの実行は、Net Simulator の右上にある start ボタンを押すことによって始まり、stop を押すと停止する。reset ボタンはマーキングを初期マーキングに戻す。トランジション□が一瞬反転し、そのトランジションが発火したことが判る。

8 おわりに

PaiLisp を用いてペトリネットツールを試作した。Net Analyzer は可達グラフの枝分れをそれぞれ並列に処理することによって高速化されているが、システムの状態に関する知識を用いた投機的実行などによる高度並列化・高速化の試みは今後の課題である。Net Simulator はトランジション 1 つずつの処理を PaiLisp プロセスとして並列に実行する。ペトリネットの実行に関してはペトリネットをそれに対応する PaiLisp プロセスに変換し実行する方法など様々なモデルが考えられるが、実行モデルに応じたペトリネットの実行とアニメーションの実現は今後の課題である。システム全体としては、PaiLisp が wish と通信を行うことによって GUI を実現した。今回 SATPN の試作を通して、GUI をも含めたアプリケーションの作成に対し、PaiLisp が十分適用可能であると考える。

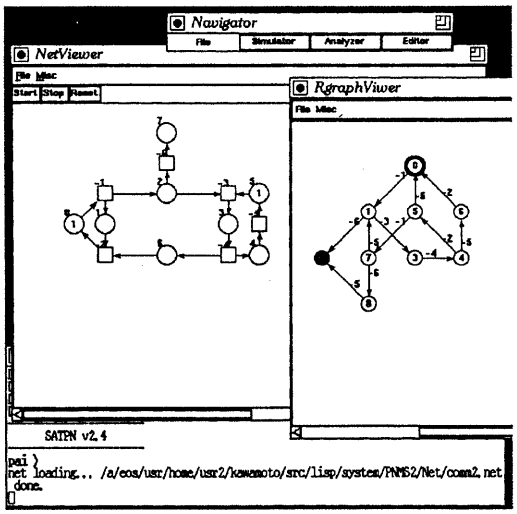

図 11: Net Analyzer と Net Simulator の画面

謝辞

本研究を進めるにあたり、PaiLisp の処理系の整備、PaiLisp のプログラミングに関し討論頂いた東北大学情報科学研究科伊藤研究室の小柳 明氏と田村清朗氏に感謝します。

参考文献

1) J.L. ピータースン: ペトリネット入門 –情報システムのモデル化–, p.293, 共立出版, (1980).

2) Kurt Jensen: *Computer tools for construction, modification and analysis of Petri nets.*, LNCS, Vol.255, pp.4-19, (1986).

3) 熊谷 貞俊: ペトリネットツール, 計測と制御, Vol.28, No.9, pp.26-30, (1989).

4) 清野 智弘 伊藤 貴康: Pailisp の並列構文の実現法と評価, 情報処理学会論文誌、Vol.34, No.12, (1993).

5) Kurt Jensen and Frits Feldbrugge: *Petri net tool overview 1986*, LNCS, Vol.255, pp.20-61, (1986).

6) John K. Ousterhout: *An Introduction To Tcl and Tk*, Addison-Wesley Publishing, (1993).