並列オブジェクト指向言語 DinnerBell と、その実行モデル

河野真治 田中英彦

Department of Electrical Engineering,The University of Tokyo
7-3-1 Hongou, Bunkyo-ku, Tokyo 113, Japan

データフローマシン、コネクションマシンのようなファイングレイン高並列計算機上の知識処理向きプログラム言語は、どのようのなものが良いのだろうか。このような言語は、真に並列実行を実現するものである必要がある。ここで要求されるのは、高並列実行に対する表現力、効果的な同期機構、明確な実行モデルである。 *DinnerBell*は、単一代入、メッセージパッシングにより、高並列計算機上の知識処理、システム記述を可能にしようという言語である。ここでは、まず高並列実行を表現する*DinnerBell*の言語仕様を示す。次に、メッセージの待ち合わせによるシンプルな同期機構が、効果的であることを示す。最後に高並列プログラミングをおこなう上で重要なプログラムの実行モデルの構築法を提案する。*DinnerBell*では、メッセージの待ち合わせ機構と単一代入という特徴を利用して、実行モデルをある種のペトリネットの構築により実現できる。この実行モデルを用いて、実際にいくつかのプログラムのバグを発見できることを示す。

1 高並列実行向きの言語とその実行モデル

これまで、さまざまなプログラム言語が、並列計算機向けに提案されてきた。その中で、高並列記号処理を目指すものが、ここでの目標である。このタイプの言語は、いままで次の 3 種類があった。

1. 同期型 fp, CM-lisp[6], Tokio[15]
2. 論理型 Parlog, Conurrent Prolog, GHC, fleng
3. オブジェクト型 ABCL[1], ACT3[4], ORIENT84/K[11], ConcurrentSmalltalk[12]

また、論理型とオブジェクト型を統合した Vulcan, A'UM もある。

これらの言語には、以下のようなことが要求される。

- プログラムに内蔵される並列性を十分表現できること
- 並行プログラムの同期関係を十分表現できること
- 並行プログラムの検証、デバッグに使える実行モデルがあること

二つ目の問題は、並列実行環境で、どのように状態を作るかという問題に帰着される。

Connection Machine では、 Xector という SIMD 計算機に適したデータ構造を処理する CM-Lisp [6] が使われている。この言語は基本的には、関数型言語の構造を持っている。SIMD でなく、 MIMD 型の言語が、より柔軟な並列処理と同期関係の記述が可能だといわれているが、関数型言語に、 MIMD 型の並列処理を導入しつつ、同期機構なしに状態を持つ変数を導入することは無理がある。なぜなら、同期機構なしに、その状態の整列可能性を維持することはできないからである。 そこで、並列プログラミング言語には、通常、なんらかの同期機構を入れることになる。

ここで、特に興味のある MIMD 型の言語は、その同期機構と状態の作り方によってさらに分類することができる。

- 状態を持つプロセスの入口で、入力を直列化してしまう (ABCL, ACT3)
- Guard を同期機構に使い、状態は stream で実現する (Parlog, etc.)

- Port を同期機構に使い、変数は状態を持たない (μ-Calculus[9])

このうち、最初の並列オブジェクト指向言語は、プロセスの中で逐次動作するものが多い。これらのオブジェクト指向言語は、アクティブなオブジェクトの数だけ並列性がある。 これらの言語の実行モデルは、通常メッセージの到着順とオブジェクトの履歴に基づき構築される。残念ながらこのモデルはその性質上、 procedural なものになってしまう。

二つ目のものは、 Commit and Choice 型言語と呼ばれる。しかし、これらの言語には、共通の問題として、基になった pure な論理言語の持つ明解な実行モデルが、非決定性を実現するために導入したガードや状態により、損なわれてしまうという欠点がある。論理変数の代入の因果関係に着目した transaction という考え方が有効であることが知られている。

三つ目のものは、 pure なオブジェクト指向言語に対するモデル[9] であるが、これは、同期機構として、ポートを用いている。これは、明らかな欠点である。なぜなら、ポートは、メッセージパッシングで統一されたオブジェクト指向言語の統一性を損ねるし、ポートでつながれたオブジェクトは、ポートの方向性により、対称性が崩れてしまうからである。

ここでは、高並列知識処理言語のもう一つの方向として、単一代入則に基づくデータ駆動実行と MessageJoin を使った同期機構を中心としたオブジェクト指向言語 *DinnerBell* を導入する。高並列を目指す MIMD 計算機は、何らかのかたちでデータフロー[7] の概念を入れているのが普通である。データフローに基づくオブジェクト指向言語はいろいろあるが [13, 2, 5]、*DinnerBell*は、もっとも単純なセマンティクスを持ち、より効果的なを実行モデルを作ることができる。ここで使うモデルは、 Petri-Net に基づくもので、若干の拡張と制限が加わっている。このモデルの特徴は、*DinnerBell* のプロセスに、対応した、 Petri-Net ができることである。このモデルを実際に並列プログラムのデバックに用いることができることを示す。

2 高並列知識処理のためのプログラム言語*DinnerBell*

ここでは、*DinnerBell* の構文規則と意味について述べる。*DinnerBell* の基本要素は、受け付けるメッセージの定義と、それに引き続いて起きるメッセージ送信の定義、変数のスコープと寿命からなる。答の返却については、Actor モデルの Continuation の考え方を採用している。さらに *DinnerBell* では、各要素の省略がプログラムの意味を決定するようになっている。

2.1 クラス定義

図1は、最も簡単な *DinnerBell* のプログラムである。最初の class Add がクラス名を表わす。*DinnerBell* のクラスは、 itBlock と呼ばれ 3 種類の括弧ではさまれたメッセージパターンとメッセージ送信の組からなる。クラスの定義には、この他に変数の宣言とインヘリタンスの宣言を付けることができる。

```
class Add [
 inc:X □ ↑(X +:1)
]
```

図 1: Simple Program

メッセージパターンとメッセージ送信は、ネックと呼ばれる四角で区切られる。ネックの前をヘッドと呼び、下線を引いたメッセージによりメッセージパターンを表わす。ネックの後ろは、ボディと呼ばれるメッセージ送信の集まりである。このプログラムは、inc:1 と言うメッセージを受け取ると、 1 を加算しろと言うメッセージを 1 に送る。*DinnerBell* の変数は 2 種類あり、クラス変数に相当する itBlockVariable (itvar) と一時変数に相当する methodVariable からなる。クラス名の後に itvar を前置して必要なクラス変数の宣言を行う。 methodVariable の宣言は行わないで、代わりにクラス名の前方参照のための宣言を itBlock 部のないクラス宣言で行う。

2.2 メッセージの送信

DinnerBell のメッセージは、 "!" 後づけした引き数なしのメッセージと、 ":" を後づけした引き数を持つメッセージである。これをそれぞれ、イベント、キーワードと呼ぶ。送信先のオブジェクトの後にメッセージを書くことにより、メッセージの送信を表す。 また複数のメッセージを後置することにより、同じ送信先への複数のメッセージの送信を記述する事ができる。 複数の送信先への並列送信は、メッセージ送信式をピリオドつなげることにより表す

(図2)。 *DinnerBell* では逐次送信の記述は特にない。順序性が要求される場合は、後で例題で示すようにデータの依存関係によって明示する。

DinnerBell は、データ駆動で実行される。つまり、メッセージ送信は、送信先が確定すれば発火し、引き数の値はそれが確定した時点で暗黙に送信先に転送される。

```
Destination event!
Destination keyword:2

Destination arg1:1 arg2:2

One one! . Two two!
```

図 2: Message Passing

2.3 メッセージの受信

どのメッセージを受信するかを表わすヘッドはメッセージパターンからなり、メッセージに下線を引いた形をしている。ここでキーワードの引き数は、変数でなければならない。メソッドは、メッセージパターンのどれか一つでも来たらメッセージが不足していても発火する。(図3)

```
event!
argument:X

arg1:A arg2:B

one: A . two: B
```

図 3: Message Pattern

また、ピリオドで区切られたメッセージパターンは、複数のメッセージの Join を表わす。Join とは送信者の違うメッセージの待合せである。この時すべてのメッセージがそろうまでボディは実行されない。このように Join は、構文的にも意味的にも並列メッセージ送信の双対である。

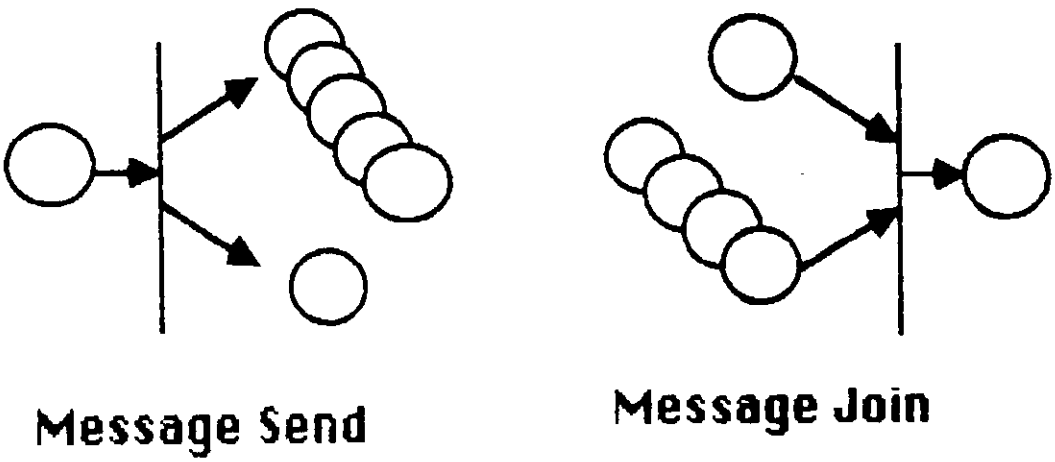

図 4: Symmery of MessageJoin

通常特定のメッセージパターンは、一つのクラス定義に一つしか許されないが、 Join を含む場合には、二つのメッソドが一つのメッセージを共有してもよい。このときには、待合せの用意ができている方が選択される。これに

より言語の表現力が変るわけではないが、簡潔に記述できる例がある (図5)。この例ではオブジェクトを状態を持つ変数として使うものである。もちろん、並列プログラミングでこのようなものを使うことは望ましいことではないが、割り込みなどの実現に使うことができる。複数のメッセージの両方が揃っているときにはランダムに選択される。この選択の仕方は、 Fairness の問題で、ここでは立ち入らない。

```
class Register [
 read!· state:X □
        ⇑X.
        state:X;
 write:X state:Y □
        state:X
]
```

図 5: Register in DinnerBel

2.4 Continutaion

*DinnerBell*では Actor モデルと同様に、答の返却は、答の送り先を指す変数である Continuation を使って実現される。特にメッセージの Join を使ったメソッドの場合には、メッセージの送信者は複数存在するので Continuation も複数存在する。そこでそれぞれのメッセージパターンの順に 1 から番号を付けて区別する。ここで、Continuation などの特別な変数は、疑似変数と呼び sender#1、sender#2 などと記述する。通常は、⇑ を用いて ⇑#1、特に、単に⇑ とした場合には、 sender#1 を指すものとする。

メッセージ送信時の Continuation の指定は、⇐ を使って行う。 Continuation としては、オブジェクトまたはメッセージパターンが使える。メッセージパターンは、答えのメッセージを受け取るオブジェクトであり、そのクラスにローカルなヘッドだけからなるオブジェクトである。単一代入変数への代入は、このようにローカルなクラスへのメッセージパッシングで実現される。

```
X ⇐ Add src: 1 dst: 2
ret: X ⇐ Add src: 1 dst: 2
high: H low: L ⇐ split of: List

NextObject ⇐ FirstObject do!

FirstObject do! continuation: NextObject
```

図 6: Continuation

2.5 スコープとループ

ロカールなオブジェクトを作るために*DinnerBell*では、3 種類のスコープとして、大きい順に、 itBlock, methodBlock, statementBlock が用意されている。それぞれ、 [ ]、{ }、( ) で表す。itBlockVariable は、itBlock の中でのスコープを持つ。 methodVariable は、methodBlock の中でのスコープを持つ。 statementBlock の中からは、 methodVariable, itBlockVariable の両方が参照できる。この時、当然、methodVariable なら、もっとも内側の methodScope に属するものが参照される。 *DinnerBell*のクラス定義は、全体で一つの itBlock を構成し、method は、一つの methodBlock を構成する。itBlock の中を ";" で区切ると、それぞれは、 methodBlock となり、一つのクラスの中の複数の method を表すことができる。ネックのない statementBlock は、通常のカッコと同じで、メッセージ送信式のネストを表す。これは、自動的に発火する statementBlock である。ヘッドとボディとも省略可能で、ヘッドが省略されたときには、eval! というメッセージが仮定される。ローカルなオブジェクトと、ヘッドの省略を使って条件式を実現したのが次の例である。

itBlock 変数はそのオブジェクトとおなじ寿命を持ち、method 変数はメッセージ受信の度に新しく作られる。Continuation は特別なスコープを持つ method 変数として扱われる。

```
class TRUE [
        yes: aBlock □ aBlock eval! ;
        no: aBlock
]

class FALSE [
        no: aBlock □ aBlock eval! ;
        yes: aBlock ;
]
```

図 7: TRUE and FALSE

itBlock の it は iteration を表していて、メッセージ送信式において送信先を省略することによりループを構成する。このときにループは、一番内側の itBlock に対して行われる。ループを含めて、ローカルなオブジェクトに対する送信は、通常のメッセージ送信よりも軽い関数呼び出しに相当する。図 8は、これを利用した階乗の計算である。

```
class Factor [
 fact: N □
        (N =: 0)
         yes: (□ ⇑ 1)
         no: (□ ⇑ (
                 (fact: (N −: 1)) *: N) )
]
```

図 8: Factorial

*DinnerBell*の単一代入則は、itBlock 変数、method 変数への再代入禁止のことである。オブジェクトへのメッセージ送信は、method 変数が新しく生成されるので、何回でも可能である。

2.6 stream と要求駆動の実現

図9は、ストリームによるプログラミングを示している。クラス Integer は、NotMardedInteger を生成する。生成されたオブジェクトは、整数をそのインスタンス変数として持っている。それらは、複数の Filter プロセスと Sift プロセスを通過する。ここで、wait: のメッセージは実際には、すべてのインスタンスに自動的に付加されるメソッドで、要求駆動を実現するために使われている。*DinnerBell*では、メッセージの送信は、転送先が確定するまで延期される。これを使って同期のための待ち行列を作ることができる。つまり、値を待つ必要のある変数にメッセージ送信式をブロックとして、wait: でその変数に送り付ける。すると値が確定するとブロックに eval! のメッセージが送られブロックがアクティブとなる。このサスペンドと値の転送によって要求駆動のためのハンドシェイクを行う。

3 *DinnerBell*の実行モデル

あるプログラミング言語で書いたプログラムがある。このプログラムを実行することによりいろいろなことが起きる。ここでいう実行モデルは、このプログラムの実行の様子を表したデータ構造である。あるプログラムのすべての可能な実行モデルの集合は、そのプログラムの意味と呼ばれる。pure な関数型言語では、実行モデルは、あるプログラムの実行において、そのプログラムに現れるすべての関数の入力と値の組であり、ロジックプログラムでは、すべての述語に対する Answer Substition の集合が実行モデルに対応する。ただし、どちらの場合でも、並列実行または、非決定性を含む様な言語拡張に対しては、実行モデルにならなくなってしまう。これは、Block-Ackerman Anormally として知られている。

DinnerBell は、message join を使った同期機構を持つ言語で、その実行は、Petri-net の生成とその上での token game と考えることができる。これを、phased Petri-net と呼ぶ。これは、以下で述べるように、状態遷移を時間軸方向に接続したようなものになっている。このモデルは、*DinnerBell* のある特定の実行に対して生成される。このモデルは、[3] などとことなり、実際のプログラムから、Petri-Net で表されるプロセスの挙動を抜き出したものである。したがって、ここで作るモデルは、元のプログラムから、いろいろな情報を捨ててプロセス間の関係だ

```
class Integer [
 N to: D □
        ND ⇐ D ret: (NotMarkedInteger make: N).
        ND wait: (□ ret: (N +: 1) to: ND)
]

class Sift [
 to: ~Out □ ⇑ {
        N □
         (N isMarked!)
                yes: ( □ ⇑ ret: (Sift to: ~Out) )
                no: ( □
                 NOut ⇐ ~Out output: N.
                 ⇑ (Filter cut: (N content!)
                        count: 1 to: (Sift to: NOut))
                )
 }
]

class Filter [
 cut: ~N count: ~I to: ~D ;
 M □
        (~I =: ~N)
         no: (□
                ND ⇐ ~D ret: M.
                ⇑ ret: (Filter cut: ~N count: (~I+:1) to: ND))
         yes: (□
                ND ⇐ ~D ret: (MarkedInteger make: M).
                ⇑ ret: (Filter cut: ~N count: 1 to: ND)
         )
]

class test [
 □      Integer ret: 2 to: (Sift to: Output)
]

class waiter
 [      wait: B □ B eval! ]
```

図 9: Demand Driven

けを抜き出したものになっている。例えば、元のプログラムでは、メッセージは値を運ぶものであるが、生成された Petri-Net では、色のないトークンとなる。

実際のメッセージパッシングの戻り値(関数の値)を問題にするデバックは、Shapiro の Algorithmic Program Debugging [10] などがあり、この結果を直接間接に利用することができる。ここでは、それでは解決できないような並列プログラムに基づく問題、デッドロックや順序の乱れに対して有効なモデルを作ることが目標である。

3.1 Phased Petri-net

*DinnerBell*には、2種類の変数があり、一つは一時変数で、もう一つはインスタンス変数である。*DinnerBell*は、単一代入であるから、インスタンス変数の値は、オブジェクトの生成以来変わらない。一方、一時変数の方は、メッセージ送信ごとに値が変わる。これに注目して、インスタンスを生成するメッセージと、そうでないメッセージの二つにメッセージを分類する。この分類は、メッセージがクラスに送られるか、インスタンス(変数)に送られるかによって分かる。

ある実行状態のメッセージの列を考え、それに(適当に)時間の若い順に番号をつける。並列実行を想定しているので、いくつかの実行には、時間順序はつけられない。この列を、インスタンス生成のメッセージを基準に分割する。

この時、分割されたフラグメントは、インスタンスを生成しないメッセージだけからなっている。このフラグメントのメッセージの流れと、インスタンスから、Petri-net を構成することができる。この時、message join のみが、transition に対する複数入力を表す。この時、Method ごとにあらかじめ生成した Petri-net の部品を結合していく方法をとる。

このとき、この Petri-net に対して、Token game がいくら進んでも、$token = message$ が増えない。(つまり、単調増加になるような game 系列がない。) という制限をつける。この制限は妥当である、何故なら、そのような系(あるいは、そのような*DinnerBell*の Program) では、同じメッセージを実行するインスタンスの(次の phase での) 生成数が並列実行の非決定性によって変わるか、無駄な message が出てるかどちらかであるからである。

さらに、この Petri-net は、メッセージが単調減少になることもあり得ない。何故なら、その時は、系は、この場で停止してしまうことがあり得るからである。

この場合、この Petri-net は、有界であり、この Petri-net に対する可達問題は Place の Token の分布のパターンに対して状態遷移図を作ることによりとける。これは、この Petri-net に関するほとんどすべての問題がとけることを意味する。

これは、*DinnerBell* Program 全体の解析に使える訳ではないが、少なくとも Program 内部のプロセスの相互作用の解析方法を提供する。また、*DinnerBell* 全体は、有界 Petri-net の列としてモデリングすることができる(図10)。

3.2 モデルの定義

ここで定義する モデルは、Petri-Net を若干変更したものになっている。モデルは、以下の要素からなる。

- Place
- Object Making Place
- Tokens in Place
- Transition
- Arrow from Place to Transition
- Arrow from Transition to Place

これをここでは、名前と行き先のついた Place $p(PlaceName, [Output1, Output2, ...])$ と、Transition の集合、$t(TransitionName, [Output1, Output2, ...])$ の集合というような形で表す。

これらは、それぞれ有限の集合である。さらに、オブジェクトを新しく生成するようなメッセージ(このようなメッセージは Petri-Net の性質を変えてしまう) のために、そのような Place は、new(PlaceName) のような形にして、特別扱いする。ここでは、そのような例は取り扱わない。

さらに、Petri-Net の生成のために、Place に、*DinnerBell* の message の入出力の印をつけた Petri-Net のフラグメントを用意する。入出力には、input, output の2種類がある。これは、*DinnerBell* のヘッドとボディに相当する。

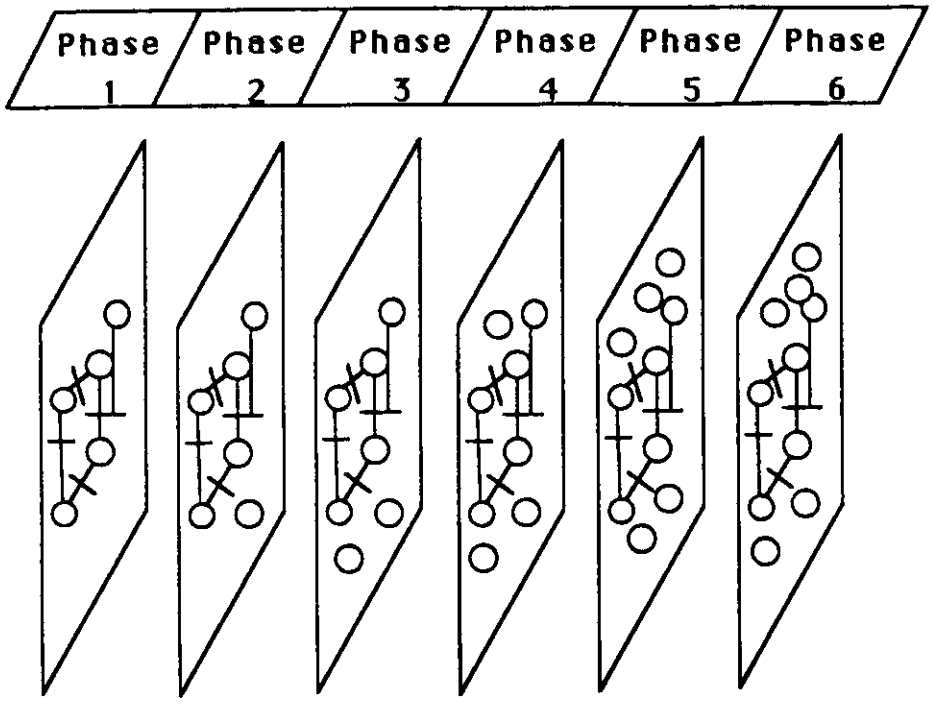

図 10: Phased Petri-net

$input$: $(message1!, Place1)$
$output$: $(message2!, Place2), output(message3!, Place3)$
net : $p(Place1, [T1]), t(T1, [Place2, Place3])$

このフラグメントは、図11のような Petri-Net になる。

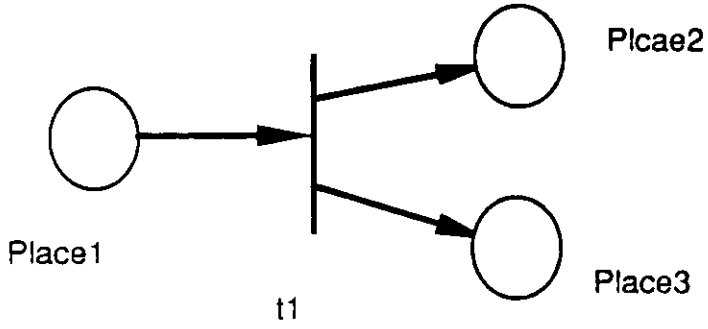

図 11: Petri-Net Fragment

3.3 モデルの生成

モデルの生成は3 段階になっている。まず、各メソッドに対してフラグメントを生成し、それから、メッセージの実行シークェンスに基づいて、それを接続していく。最後に冗長な部分を取り除く。

変換は、メッセージパッシングと Petri-Net の対応を重視しておこなわれる。まず、メソッドのメッセージパターンは、一つの Place に変換される。その後に、一つ Transition を置く。それからは、そのメソッドが起動された時に送信されるメッセージに対応する Place への矢印が出る。例えば、図7 は、以下のようなフラグメントに変換される。(図12)

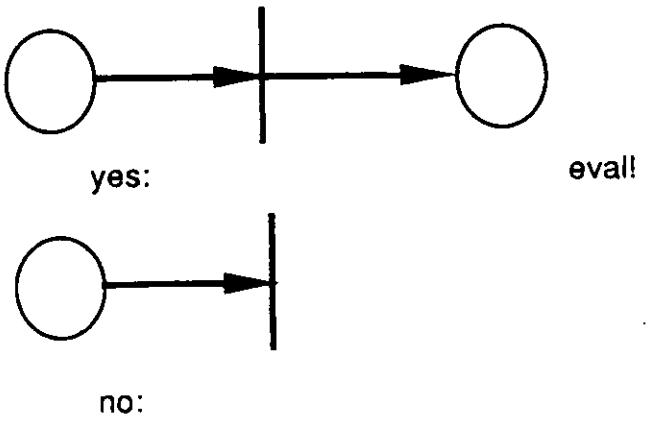

図 12: TRUE のフラグメント

MessageJoin の場合は、待ち合わせるメッセージの各々に対して、Place 作り、その後に一つ Transition をおく。図13は、*DinnerBell*で記述した、Dining Philosopher の、Fork の部分である。

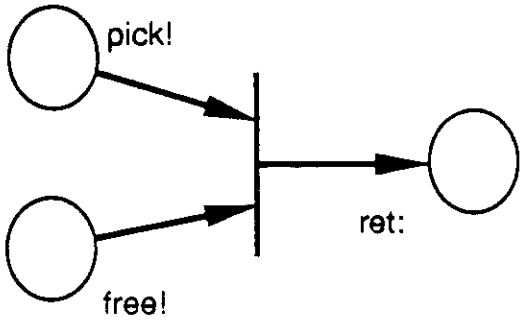

図 13: Fork のフラグメント

Petri-Net の生成は、メソッドごとに生成したフラグメントを実行の履歴に沿って接続することによっておこなう。接続は、実際には、メッセージの送信に対応しておこなわれるものと、変数の代入に対応しておこなわれるものがある。図14のようなメッセージ送信がおこなわれたとする。

(~Lfork pick!) yes: (□ pickR!);

図 14: Fork の送信

この時、Fork のメッセージが発火したとすると、TRUE の値が返されて、それに、yes: というメッセージが送られる。まず、Fork の戻り値である TRUE に、yes: が送られる。このとき、フラグメントが接続される。さらに、TRUE が、送られた itBlock に、eval! を送る。この接続の様子は、図15になる。

単一代入則に基づく*DinnerBell*では、同じオブジェクトへの同じメッセージの送信は、通常同じ Petri-Net をもう一つ生成する。このとき、MethodVariable の値だけが異なって実際に、それぞれメッセージを送ったオブジェクトの Continuation に向かって戻り値を返すからである。戻り値を返さない場合は、同じ Petri-Net を共有することができる。Fork の例の場合は、この二つの両方の場合が出てくる。Fork のオブジェクトには、二つの Philosopher オブジェクトからの送信がありえるが、まず、free! のメッセージには、戻り値がないので、このメッセージは一つの Place への送信となる。

pick! のメッセージは値を返すので、二つ別々の Petri-Net を作る。MessageJoin の場合の接続は特別で、free! のメッセージは、二つの pick! のどちらとも Join する可能性がある。これは、同じオブジェクトに、複数のオブ

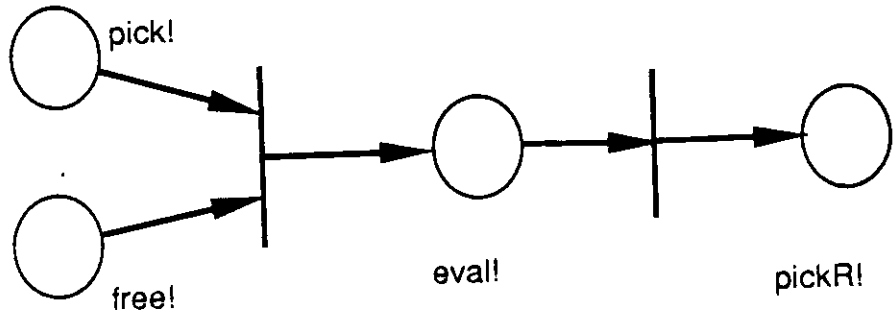

図 15: フラグメントの接続

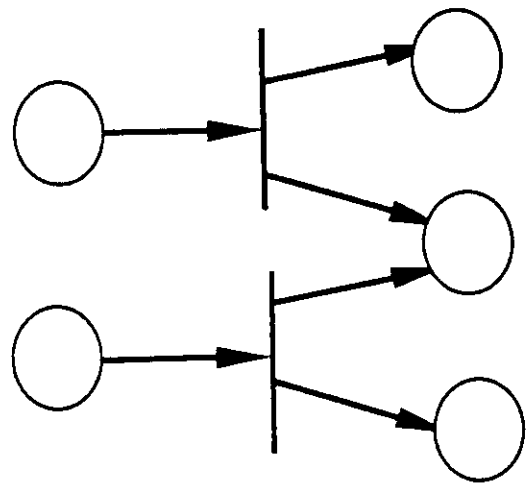

図 16: 同じオブジェクトに複数のメッセージが来る場合

ジェクトからメッセージが来る場合の双対になっている。これは、 MessageJoin の持つもう一つの双対性である。(図17)

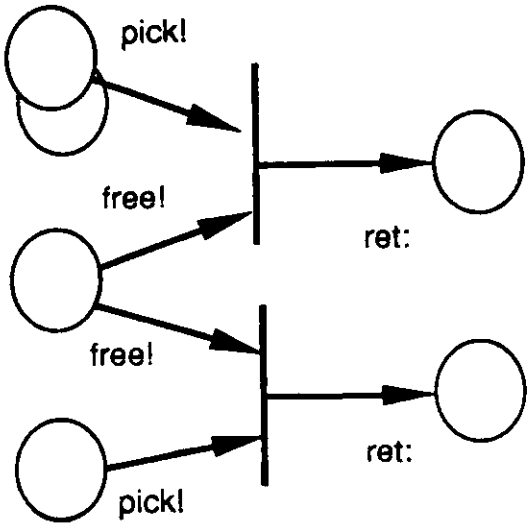

図 17: MessageJoin の場合

また、あるメッセージが生成した Petri-Net から、そのメッセージと同じメッセージを自分自身に送る場合は、特別で、この場合は、これまでに作った、 Net への接続となる。このような、 Petri-Net 上のループが、実際に、*DinnerBell* のプロセスに相当する。

DinnerBell の if 文は、あるメッセージの答えが、 TRUE、 FALSE の二通りあることにより実現されている。このような時には、さらにこれに対して、 yes: no: の形のメッセージにより二つの itBlock が引き続いているのが普通である。このような場合は、この二つの itBlock のどちらかに eval! のメッセージが送られることに帰着される。(図18)

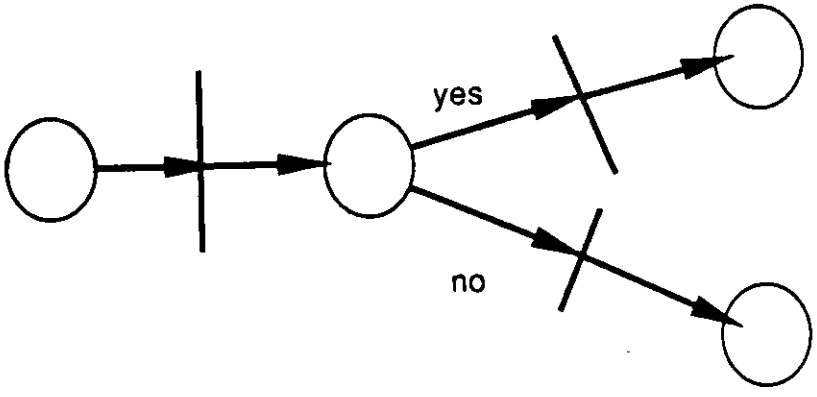

図 18: if 文

このようにして生成した Petri-Net には、いくつか冗長な部分がある。例えば、 if 文の時に生成された eval! メッセージは、通常は省略できる。ここでは、冗長な部分の除去はそれほど重要ではないので、以下のような時だけ、余分な、 Place と、 Transition をとりのぞく。(図19)

4 デバッグへの応用

ここでは、実際に、*DinnerBell* のプログラムのデバックに、この実行モデルを応用してみよう。ここで対象とするプログラムは、一つは、 Dining Philosopher 問題で、プログラムのデッドロックを見つけるものである。もう一つ、メッセージ送信の順序が保証されないことによるバグを見つける例として、 "Talk" 二つの Display と、二つの KeyBoard を使って、互いに話しをするシステムを考える。

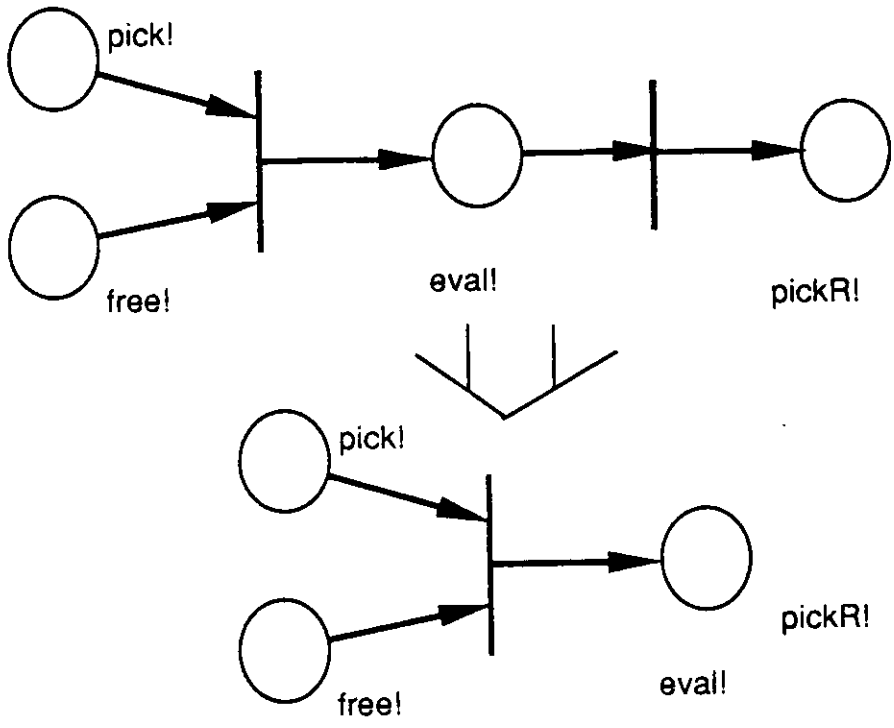

図 19: 冗長な部分の除去

4.1 Dining Philosopher による例

図20は、*DinnerBell* による Dining Philosopher 問題である。ここで、まず、五人の哲学者と五つのフォークが作られる。哲学者は、自分持しにメッセージを送ることにより、自分の状態を作り、順序良く遷移していく。フォークの方は、非決定的な状態を実現するために、 MessageJoin を使っている。このフォークは、free! と、pick! の二つのメッセージを待ち合わせるようになっている。これは、誰かが、free! しなければ、 pick! できないというように読むことができる。これは、*DinnerBell* で、 binary semaphor を実現する時の標準的な方法である。

この時生成される Petri-net は、すべてのオブジェクトが生成された後は、変化しない。図21は、その一人分のネットを表している。このネットの、デッドロックが、そのまま、プログラムのデッドロックに相当する。

4.2 Talk の例

図22 は、*DinnerBell* で記述した Talk のプログラム例である。このプログラムは、 Key Board から、互いに文字を読みとって、自分と相手の Display に送り出す。 Display は、状態を持つオブジェクトであるから、 Message-Join を使って同期をとっている。残念ながらこのプログラムには、バグがある。

このプログラムが生成する Petri-net が、図23である。このネットは、有界でない。これは、 Key Board が、発生するメッセージが有界でないことによる。実際このプログラムは、 Key Board から読みとったメッセージが、Display にたまって順序が変わる可能性がある。

この間違いをなおしたものが、図24である。この時は、 Key Board が、 Display からの ack! を待ち合わせることにより、同期をとっている。これから生成される Petri-net も、正しい *DinnerBell* の実行モデルになっている。

5 他のモデルとの比較

オブジェクト指向言語と Petri-net の関係は、古くは、 message system と Petri-net の関係として捉えられてきた。この時は、 Petri-net が、 message system を含むことが示されただけで、 Petri-net の可達問題がすでに難しいものである以上、あまり利益はなかった。*DinnerBell* の場合は、有界な Petri-Net、または、 Petri-Net の有界性の問題へ問題を限ることにより、これを避けている。

また、長岡科学技術大学で行なわれてる Petri-net から Ward の μ-Calculus への変換は、ここで用いたモデリングのちょうど逆に相当する。しかし、彼らの方法では、非決定性の導入が、 port を使って行なわれているため

```
class Fork [
        pick! . free! □                          % sync free and pick
                sender#1 ret:TRUE
]

class Philosopher
[
 left:~Lfork right:~Rfork id: ~N ;

 pickL!          □
        (~Lfork pick!) yes: (□ pickR! );
 pickR!          □
        (~Rfork pick!) yes: (□ eat! );
 eat!            □       freeL! ;
 freeL!          □       ~Lfork free! . freeR! ;
 freeR!          □       ~Rfork free! . think! ;
 think!          □       pickL!
]

class diningPhilosopher
[       □
        Fa      ⇐ Fork free!.
        Fb      ⇐ Fork free!.
        Fc      ⇐ Fork free!.
        Fd      ⇐ Fork free!.
        Fe      ⇐ Fork free!.
        Pa      ⇐ Philosopher new! . Pa left:Fa right:Fb id:1.
        Pb      ⇐ Philosopher new! . Pb left:Fb right:Fc id:2.
        Pc      ⇐ Philosopher new! . Pc left:Fc right:Fd id:3.
        Pd      ⇐ Philosopher new! . Pd left:Fd right:Fe id:4.
        Pe      ⇐ Philosopher new! . Pe left:Fe right:Fa id:5.
        Pa think!.Pb think!.Pc think!.Pd think!.Pe think!
]
```

図 20: Dining Philosopher

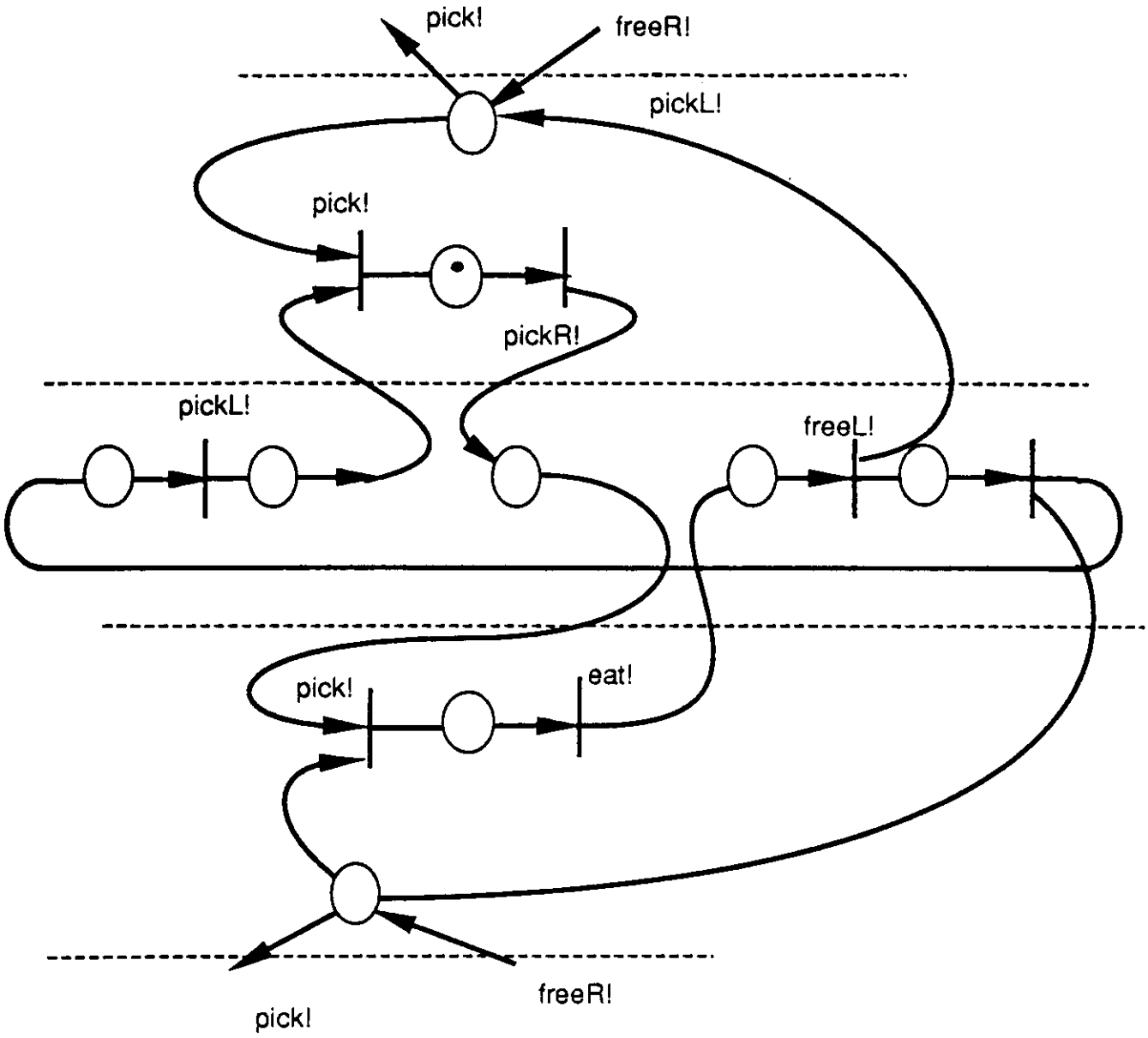

図 21: Dining Philosopher の Petri-net

```
class Talk [
 between:User1 and:User2 □
        Display1 ⇐ Display newFor:User1.
        Display2 ⇐ Display newFor:User2.
        Split1 ⇐ Split to:Display1 and:Display2.
        Split2 ⇐ Split to:Display2 and:Display1.
        KeyBoard newFor:User1 to:Split1.
        KeyBoard newFor:User2 to:Split2
]

class Display [
 newFor:~User □ to:StdErr;
 output:X· to:Out1 □
        Out2 ⇐ Out1 write: ~User write:X nl!.
        Out2 wait: (□ to: Out2)
]

class Split [
 to:~A and:~B;
 output:X □
        ~A output:X.
        ~B output:X
]

class KeyBoard [
 newFor:~User to:Out □
        from:StdIn to:Out;
 from:In to:Out □
        (X) ret: (In getC!).
        (X=:0)
         no:(□
                Out output:X.
                from:In to:Out)
]

class test [
 □ Talk between: "a" and: "b"
]
```

図 22: talk

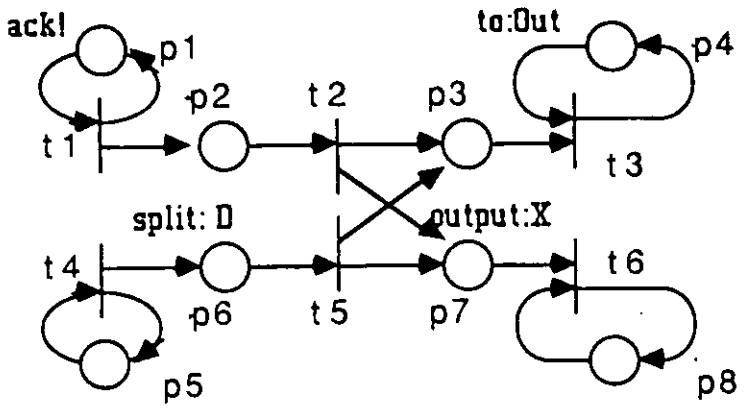

図 23: 誤りのある Talk のプログラムのペトリネット

```
class Talk [
 between:User1 and:User2 □
        DA ⇐ Display newFor:User1 out:StdErr.
        DB ⇐ Display newFor:User2 out:StdErr.
        KeyBoard from:StdIn to:DA and:DB.
        KeyBoard from:StdIn to:DB and:DA
]

class Display [
 newFor:~User out:Out □ to:Out;
 output:X· to:Out1 □
        Out2 ⇐ ((Out1 write:~User) write:X) nl!.
        Out2 wait: (□ to: Out2. ⇑TRUE) ]
class KeyBoard [
 from:~In to:~A and:~B □ ackA!.ackB!;
 ackA!·ackB! □ split: (~In getC!);
 split: Data □ (~A output: Data) yes:(□ ackA!).
 (~B output: Data) yes:(□ ackB!) ]

class Test [
 □ Talk between:"Rick " and:"Eliza "
]
```

図 24: 正しい talk

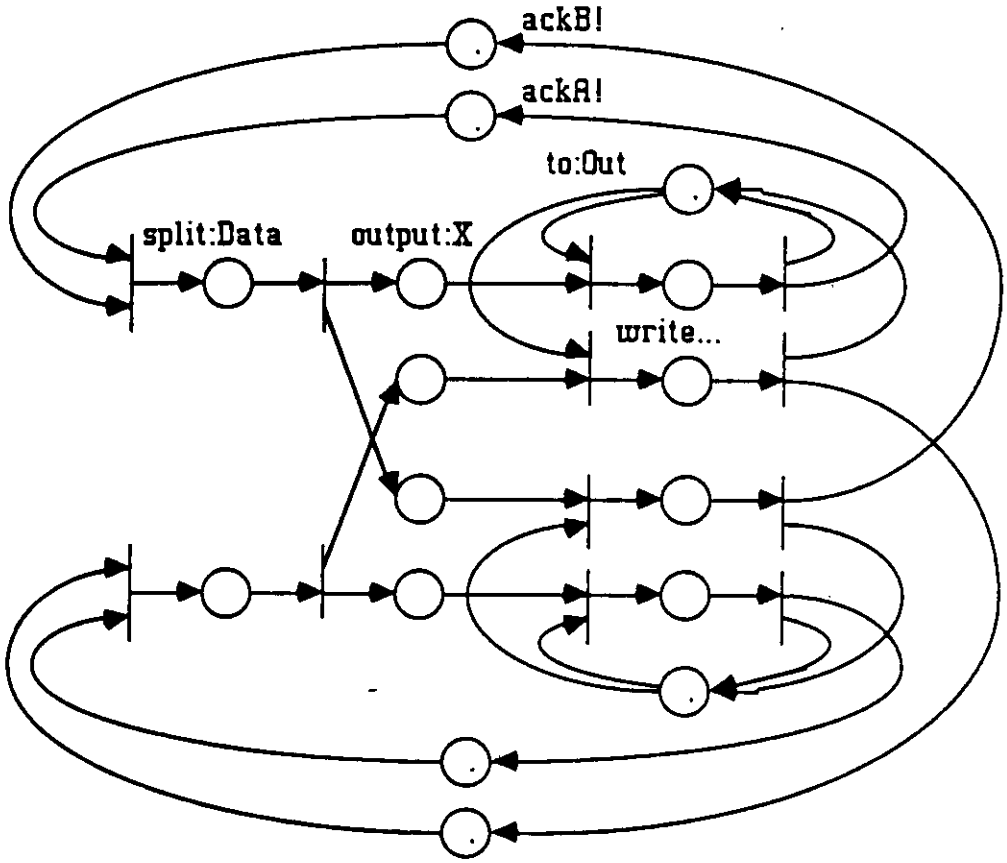

図 25: 正しい Talk のプログラムのペトリネット

に、元の Petri-net が持つ対称性が生成された Program に反映されない。*DinnerBell* 特有の Messages on 同期は、このような非対称性が生じないという利点がある。

また、Petri-Net の表現力には限界があることがしられているが、DinnerBell 自身には、そのような限界はない。DinnerBel では、Petri-net で表現できない並列システム(トークンの不在を検出するもの)を、message に乗せた「状態」により、これを実現できる。ただし、message の状態を含む問題は、ここでもちいたの方法では解析できない。

6 最後に

このオブジェクト指向言語から Petri-Net を得る方法により、Fine Grained Parallelism に基づいて動作する言語から、プロセスを Petri-Net でのループの形で抜き出すことができる。実際のプログラムの実行に基づいて、Petri-Net を構築し、さらにその Petri-Net の性質を調べるという並列プログラムのデバック方法は、他に例を見ない。これにより、並列プログラムでは、再現性のないような問題、例えば、Dining Philosopher のデッドロックや、通信の途中での順序の保存性などのデバックを可能にすることができる。これは、この時プログラム全体の検証という難しい問題を解かずに、ある実行の瞬間だけ取り出して問題にすることができるからである。

もちろん、実行にしたがってモデルを作るこの方法では、ある特定の実行に関連するバグしか対象にならないので、この点は限界がある。しかし、通常のプログラムでは、全体が定常的に動く、つまり、いくつかのプロセスが協調して動いている状態があり、このような状態でのプロセス相互の関係を調べることがもっとも重要である。このような場合には、ここで示した方法が有効となる。残念ながら、非常に頻繁にプロセスが生成消滅する場合には、ここで述べた方法を使うことはできない。

DinnerBell 自身は、*DinnerBell* を micro-message と呼ぶ引数単位のメッセージ送信の実行単位に展開するコンパイラと、micro-message を実行する仮想的な 1 から 64 台プロセッサのシミュレータが動作している。これらは、コンパイラ、ランタイムとも C で記述されており、VAX, Sun/3 上で動作している。

Petri-Net の生成系は、*DinnerBell* から Prolog へのコンパイラとして記述されており、トークナイザ、コンパイラ、インタプリタ、はすべて Prolog で記述されている。こちらの方は、ブロックないのブロックの取り扱いなど、改良中である。

参考文献

[1] A. Yonezawa, E. Shibayama, T. Takada, Y. Honda. Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1. In *Object-Oriented Concurrent Programming*, The MIT Press, 1987.

[2] A.S. Grimshaw, J.W.S. Liu. Mentat: an object-oritented macro data flow system. In *OOPSLA 87*, pages 35–47, ACM, 1987.

[3] E. Meijer. Petri net models for the λ-calculus. In *Advances in Petri Nets 1987, LNCS 266*, Springer-Verlag, 1987.

[4] C. Hewitt G. Agha. Concurrent Programming Using Actors. In *Object-Oriented Concurrent Programming*, The MIT Press, 1987.

[5] G.E. Kaiser. Melding data flow and object-oriented programming. In *OOPSLA 87*, pages 254–267, ACM, 1987.

[6] G.L. Steele, W.D. Hillis. Connection machine lisp: fine-grained parallel symbolic processing. In *1986 ACM Conf. on LISP and Functional Programming*, pages 279–297, ACM, 1987.

[7] J.B. Dennis. Models of data flow computation. In *CompCon*, IEEE, 1984.

[8] S. Kono and T. Aoyagi and M. Fujita and H. Tanaka. Implementation of temporal logic programming language Tokio. In *Logic Programming Conference '85*, pages 138–147, ICOT, 1985.

[9] S.A. Ward, R.H. Halstead. A Syntactic Theory of Message Passing. *J. ACM*, 27(2), 1980.

[10] E. Y. Shapiro. Algorithmic Program Debugging. M.I.T. Press, 1982.

[11] Y. Ishikawa, M. Tokoro. Orient84/K: An Object-Oriented Concurrent Programming Language for Knowledge Representation. In *Object-Oriented Concurrent Programming*, The MIT Press, 1987.

[12] Y. Yokote, M. Tokoro. Concurrent Programming in ConcurrentSmalltalk. In *Object-Oriented Concurrent Programming*, The MIT Press, 1987.

[13] Y. Zhong, M. Sowa. Towards an implicitly parallel object oriented language. In *Procs. of Compsac 87*, pages 481–485, 1987.

[14] 神田陽治, 金子誠司, 田中英彦, 元岡達, 並列オブジェクト指向言語 DinnerBell の概要 情報処理学会ソフトウェア基礎論研究会, SF-11-3, 1984.

[15] 河野真治, 立川江介, 渡部眞幸, 神田陽治, 金子誠司, 田中英彦, 元岡達, オブジェクト指向言語 DinnerBell のコンパイラ 電子通信学会技術報告, EC-85-72, 1985.

[16] 渡部眞幸, 河野真治, 田中英彦, 並列オブジェクト指向言語 DinnerBell – 非決定的な部分のデバッキング 情報処理学会第 35 回全国大会,3R-6,1987.

[17] 河野真治, 田中英彦, データ駆動に基づくオブジェクト指向言語 DinnerBell データフローワークショップ,1987.

[18] 橘爪進, 神保知余, 猪股俊光, 西村安行, μ式によるペトリネットの表現 情報処理学会第 34 回全国大会,6U-2,1987.