[27] 河野真治, 青柳龍也, 藤田昌宏, 田中英彦. 時相論理型言語 Tokio の検証. In *Logic Programming Conference '86*, 1986.

[28] 河野真治, 立川江介, 渡部眞幸, 田中英彦. 並列オブジェクト指向システム ORAGA - 単一代入則に基づくオブジェクト指向プログラミング -. 情報処理学会第 *33* 回 *(*昭和 *61* 年後期*)* 全国大会, number 5D-2, 1986.

[29] 橋爪進, 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 *(*昭和 *62* 年前期*)* 全国大会, number 6U-2, 1987.

[30] 神田陽治. 並列オブジェクト指向計算機システム *ORAGA* - プログラミング支援環境からの考察. PhD thesis, The University of Tokyo, 1986. 東京大学大学院工学系研究科情報工学専門課程博士論文.

[31] 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. Technical Report SF-11-3, 情報処理学会ソフトウェア基礎論研究会, 1984.

[32] 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 *(*昭和 *62* 年後期*)* 全国大会, number 3R-6, 1987.

[33] 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 のデバック環境. 情報処理学会第 *36* 回 *(*昭和 *63* 年前期*)* 全国大会, number 1H-4, 1988.

[10] Kenneth Kahn, Eric Dean Tribble, Mark S. Milller, and Daniel G. Bobrow. Objects in Concurrent Logic Programming Languages. In *OOPSLA 86*, pp. 242–257. ACM, 1987.

[11] G.E. Kaiser. MELDing Data Flow and Object-Oriented Programming. In *OOPSLA 87*, pp. 254–267. ACM, 1987.

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

[13] J. E. B. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. Technical Report LCS TR-260, MIT Laboratory for Computer Science, Jun. 1981.

[14] T. Moto-oka, H. Tanaka, H. Aida, K. Hirata, and T. Maruyama. The architecture of a parallel inference engine -PIE-. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*. ICOT, 1984.

[15] M. Murakami. An Axiomatic Verification Method for Synchronization of Guarded Horn Clauses Programs. Technical Report TR-339, ICOT, 1988.

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

[17] A. Takeuchi. Algorithmic Debugging of GHC Programs and Its Implementation in GHC. Technical Report TR-185, ICOT, 1986.

[18] Kazunori Ueda. Guarded Horn Clause. Technical Report TR-103, ICOT, Jan. 1985.

[19] Kazunori Ueda and Koichi Furukawa. Transformation Rules for GHC Programs. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*. ICOT, 1988.

[20] S.A. Ward and R.H. Halstead. A syntactic theory of message passing. *J. ACM*, Vol. 27, No. 2,, 1980.

[21] Daniel Weinreb and David Moon. Flavors: Message passing in the lisp machine. Technical report, Nov. 1980.

[22] Xerox. The smalltalk-80 system. *Byte*, Vol. 6, No. 8,, Aug. 1981.

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

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

[25] Kaoru Yoshida and Takashi Chikayama. A'UM - a stream-based concurrent object oriented language -. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*. ICOT, 1988.

[26] Y. Zhong and M. Sowa. Towards an Implicitly parallel Object Oriented Language. In *Procs. of Compsac 87*, pp. 481–485, 1987.

成された Program に反映されない。*DinnerBell* の持つ MessageJoin による同期は、このような非対称性が生じないという利点がある。実際 Petri-Net から *DinnerBell* のプログラムを合成することは、μ–Calculus に対する変換よりも容易であると期待される。これは、並列プログラミング言語として、*DinnerBell* が優れている点である。

8 最後に

DinnerBell は単一代入により Fine Grained Parallelism を目標に設計されたオブジェクト指向言語である。以上述べたように、*DinnerBell* からプロセスの同期関係をループをなす Petri-Net で表現することができる。これを用いて、Algorithmic Program Debugging の方法が適用できないような停止しないプログラムの同期関係についても系統的なデバッグを行なうことができる。

DinnerBell 自身に関しては、*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 64 台までの並列プロセッサシステムのシミュレータが動作している。これらは、コンパイラ、ランタイムルーチンとも 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。これらは現在、改良中である。

参考文献

[1] G. Agha and C. Hewitt. Concurrent programming using actors. In *Object-Oriented Concurrent Programming*. The MIT Press, 1987.

[2] E. A. Ashcroft. Dataflow and eduction: data-driven and demand-driven distributed computation. In *Current Trends in Concurrency, LNCS 224*. Springer-Verlag, 1986.

[3] J.D. Brock and W.B.Ackermann. Scenario: A model of nondeterminate computation. In *Formalization of Programming Concepts*. Springer-Verlag, 1981.

[4] Keith Clark and Steve Gregory. Parlog: Parallel progaming in logic. Technical Report Research Report DOC 84/4, Imperial College of Science and Technology, April 1984.

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

[6] A.S. Grimshaw and J.W.S. Liu. Mentat: An Object-Oritented Macro Data Flow System. In *OOPSLA 87*, pp. 35–47. ACM, 1987.

[7] Carl Hewitt. Control Structure as Patterns of Passing messages. In *Artificial Intelligence: An MIT Perspective*, volume 2, pp. 435,465. 1979.

[8] Carl Hewitt and Russell R. Atkinson. Specification and Proof Techniques for Serializers. *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 1,, Jan. 1979.

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

デバッグに対する利点となる。

7 他のモデルとの比較

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グ方法により、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グ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。

Algorithmic Program Debugging に対する本手法の明らかな利点は、

- 永続プロセスのデバッグ、
- 非同期通信のデバッグ、
- 特定の実行履歴の定常状態の一般的解析、

である。最初の二点は、Algorithmic Program Debugging では実例がないし、実際困難である。最後の特徴により、再現性のない並行プログラムのバグの発見のために、並行実行の可能なスケジューリングをすべて調べる必要がない。特定の例でも、ある程度、同期関係を調べることができる。これは、Algorithmic Program Debugging を並行プログラムへの応用する際に有効である [17]。

これは、並行プログラムのプロセスの同期関係を元のプログラムから分離して、解析することに相当する。ただし、任意の個数のプロセスをユーザから与えたパラメータで作るものに対しては、ここで与えた方法では有界な Petri-Net を生成することはできず、比較的困難な非有界な Petri-Net の解析を必要とする。しかし、このような場合でも、実行履歴によりモデルを作るこの方法では、パラメータが決まってトークンの数が有界となった時点の解析は行なうことができる。またタイムアウトのパラメータがあるような同期関係がメッセージの状態に強く依存するようなプログラムは、部分的な解析にしか使えない。

もちろん、この方法は、あくまでデバッグであり、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る。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り、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。

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計算量的に難しいものである以上、あまり利益はなかった。ここでは、有界な Petri-Net 上の可達問題、または、Petri-Net の有界性の検出問題へ問題を限ることにより、この計算量の問題を避けている。したがって、非有界なプログラムに関しては、計算量的には困難なままである。

また、Petri-Net の表現力には限界があることがしられている。*DinnerBell* 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、メッセージに乗せた「状態」を使って実現できる。しかし、メッセージの状態と同期が結び付いた問題は、ここで用いた方法では解析できない。

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 [29] は、ここで用いたモデリングのちょうど逆のプログラム合成に相当する。しかし、μ–Calculus では、非決定性の導入が、port を使って行なわれている。すなわち、元の Petri-Net が持つ対称性が、生

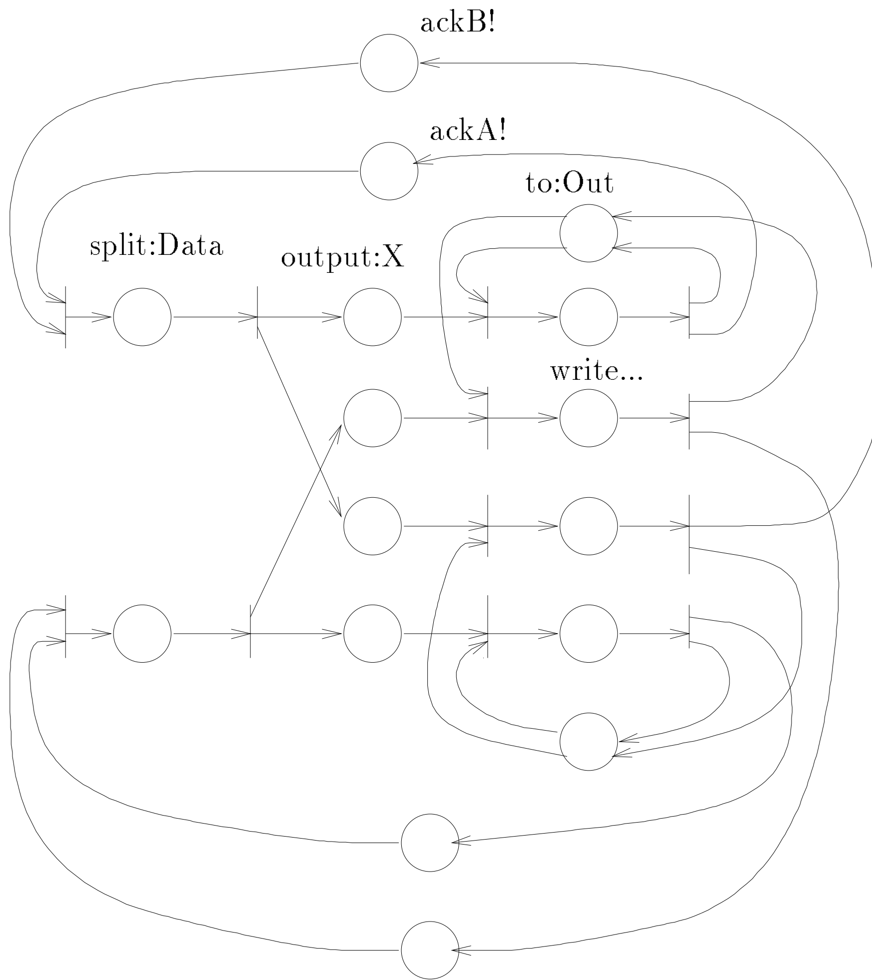

図 25: 正しい Talk のプログラムのペトリネット

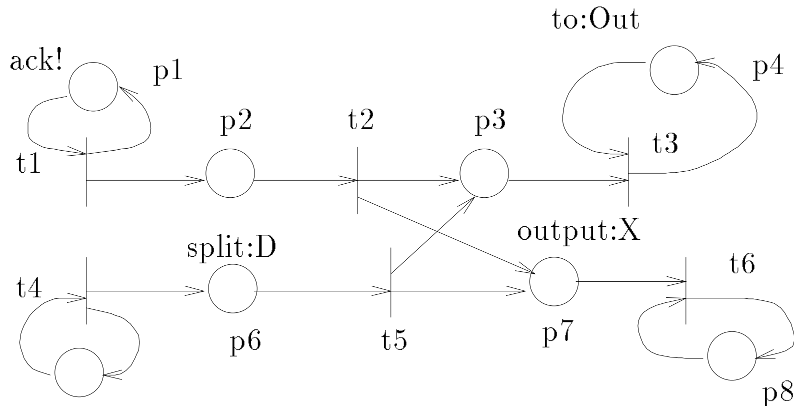

図 23: 誤りのある Talk のプログラムのペトリネット

```
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

```
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

メインのプログラムである **Talk** に、between:and: というメッセージを送ることにより、二つのキーボードと二つのディスプレイの間で、一文字ごとの通信が始まる。**Talk** は、**Display**, **Split**, **KeyBoard** の 3 種のオブジェクトを二つずつ作り、それぞれを接続する。**KeyBoard** のオブジェクトは、**StdIn** のインスタンスを引数として持つ from: を周回してプロセスを作る。このメッセージの一周に対応して output: というメッセージで入力が **Split** に送られる。そこで、output: は二つに分けられ、別々の **Display** に送られる。**Display** では状態を持つ **StdErr** を to: というメッセージで周回させて、来た入力にユーザ名を付けて出力している。

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、**KeyBoard** 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、**keyBoard** がメッセージを複数発生することにより、そこに接続された Place にいくらでもトークンを増やすことができる。その Place にトークンがたまると **KeyBoard** から読みとったメッセージの順序性はなくなってしまい、**Display** と、**KeyBoard** の間のメッセージの順序性もなくなってしまう。

このプログラムは、モデルの非有界性が、直接バグを引き起こす例になっている。つまり、プログラマの考えている仕様として有界なモデルを考えているのに対して、プログラムは非有界なモデルを生成してしまう。これ自身がバグの存在示している。

この間違いをなおしたものが、図 24である。この時は、**KeyBoard** が、**Display** からの ackA!, ackB! 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 は、*DinnerBell* 有界な実行モデルになっている。

Petri-Net を使ったモデルの特徴の一つは、モデルの分離合成が容易な点である。この例でも、**StdIn**、**StdErr** は、この Petri-Net の外に分離されている。ここでは、Petri-Net に流入するインタフェースが、トークンを一つ送るとトークンが一つ入るような形になっていると考えている。つまり、外部からのトークンが Petri-Net の有界性を破壊してない。実際の Input/Output は、複雑な別の Petri-Net により表現されるはずだが、ここでは、単純な Petri-Net により、その基本的な性質だけが表現されている。

この種のバグは、 Commited Choice 型言語で記述した Talk では現れにくい。なぜなら、それらの言語では同期通信が中心であり、ストリームのマージが直接に非有界なプログラムを作ることがないためである。これは、同期通信に限る場合は、有界な Petri-Net のみが生成されるためである。実際の Unix などでは、 Talk でも通信路の確定までの非同期通信が非常に難しい部分である。同期通信である相互通信はむしろやさしい部分である。非同期通信をストリームとマージで実現した、より複雑な場合に、このようなバグが発生することがある。

この例でも、対象は永続プロセスであり、デバッグが難しいプログラムの一つである。 Alorthmic Program Debugging などでは、このような非同期通信を扱うことはできない。実際マージを含むだけで状況は十分に複雑になってしまう。非同期通信の持つ難しさは、オブジェクト指向言語やサーバ / クライアントモデルなどに特有の問題であり、このデバッグ手法のユニークな点である。

このような非同期プロトコルに関しては、デバッグよりも、むしろ検証に重きがおかれてきた。検証の場合は、ある特定のプロトコルに関する仕様を決めて、その仕様とプロトコルが反しているかどうかをプロトコルの可能な状態すべてに関して調べなくてはならない。この検証は特にプロトコルに含まれるプロセスの数が不定の時には特に難しい。しかし、ここで示した方法は、ある特定の実行履歴に基づいているために、ある特定の数のプロセスが含まれた時点の Petri-Net を解析している。これによる相対的な解析の容易さが利点の一つである。この点で、プロトコルから Petri-Net を生成して検証を行なう方法とは異なる。

プロトコルの正しさは、ここでの方法では保証できないが、正しくないプロトコルのデバッグと考えると、定常状態になった時の同期関係の性質を調べられることが、トレースに基づいた

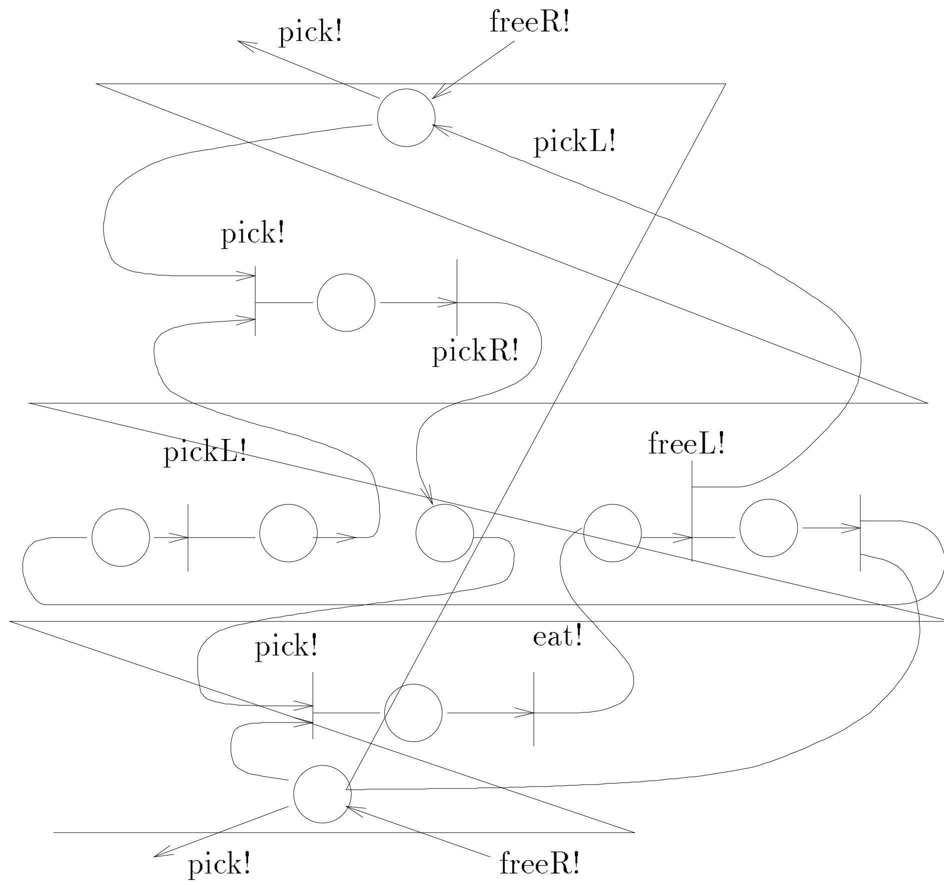

図 21: Dining Philosopher の Petri-Net

```
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

この例では、まず五人の哲学者と五つのフォークが作られ、それぞれ初期化される。哲学者は、自分自身にメッセージを送ること(メッセージの周回)によりプロセスを構成している。自分の状態は、周回するメッセージにより決まり、次々に遷移していく。

diningPhilosopher は、5つのフォークと5人の哲学者を初期化をする。**Fork** は、ここで free! が送られ解放状態にある。哲学者は left:right: のメッセージにより、それぞれが知っているフォークが配られる。それから、think! が送られ **Philosopher** のプロセスが始まる。各人のプロセスは、pickL!, pickR!, eat!, freeL!, freeR!, think! の6つのメッセージによる状態遷移からなる。各プロセスは **Fork** に対して、pick! を送ることでフォークを獲得し、free! を送ることで解放する。**Fork** では、free! と pick! の待ち合わせにより一つのフォークの資源管理を行ない、pick! できたものに確認の **TRUE** を返している。ここでの **Fork** からの戻り値 **TRUE** へ送られる yes: は、wait: と同じ働きをしている。

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。この Petri-Net は有界なので、デッドロックするかどうかは、状態グラフを作るなどの直接的な Petri-Net の解析で調べることができる。具体的には、すべての **Philosopher** が pickR! の状態に入ると、待ち合わせる free! のトークンがなくなり、したがって発火可能なトークンがない。すなわち、デッドロックする。

Comitted Choice 型言語などのデータ駆動に基づく言語では、ある変数への代入を忘れたり、返すべき答えを送らなかったりする場合に、デッドロックがよく起こる。そのようなデッドロックは変数やメッセージ送受信の入出力のモード解析や、Algorithmic Program Debugging などでも見つけることができる。この Dining Philosopher のデッドロックは、フォークの資源管理を原因とするバグであり、入出力のモード解析では見つけることができない。なぜなら、このデッドロックはプロセスの相互作用によってはじめて起きるものであり、ここのモード解析自身は正しいためである。

また、Algorithmic Program Debugging では、このようなバグを検出することは難しい。一つはこのプログラムは永続的なプロセスからなっているために、完全な Answer Substitution を求めることができない。また、このデッドロックは全体のプロセスのある特定の場合に起きる。したがって、このデッドロックを起こす実行履歴を見つけるためには、ある程度の量の可能な実行履歴を調べなければならない。

ここで示す方法では、Dining Philosopher が定常的な通信をはじめた状態の Petri-Net を作ることができると、その状態の同期関係を詳しく調べることができる。したがって、可能な実行をすべて調べることなく、定常状態に行き着くことさえできれば良い。実際この場合は、Algorithmic Program Debugging のようなデッドロックを起こすまでの試行錯誤を行なう必要はなく、一回の実行履歴で必要な情報を得ることができる。

このデッドロックは、フォークと哲学者に関する資源管理グラフのループにより検出することができる。ここで構成した Petri-Net は、同期機構に関する資源管理グラフを一般化したものである。実際、デッドロックした状態の Petri-Net で、その発火条件をたどることにより、資源管理グラフのループに相当するものを得ることができる。

6.2 Talk の例

図22は、Talk のプログラム例である。**KeyBoard** 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り、MessageJoin を使って同期をとっている。このプログラムには、メッセージの順序性が保証されないというバグがある。これについて、今まで述べてきた方法がどう使えるか調べてみよう。

この二つの場合の Petri-Net を見ると、(場合 1) の双対になっている。これは、MessageJoin の持つメッセージの受信送信の双対性のもう一つ別な形である。この Petri-Net のように、セマフォとして MessageJoin を使うプログラムの場合には、もしプログラムが有界ならば、必ずこの対の形が出てくる。これは、有界なプログラムを作る時の基準の一つである。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、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、*DinnerBell* のプロセスに相当する。

5.4 元のプログラムとモデルの関係

このようにして生成したモデルは、新しくインスタンスが生成されるまで、つまり itBlock 変数のリンクで構成したデータフローグラフの構成が一定である間の、並行プログラムの挙動の同期関係を表している。このモデルでは、

- 固定された itBlock 変数のリンクによるループ、
- 状態を作るためにそのループを周回する method 変数を持ったメッセージ

の二つの要素がプロセスを表している。同じようにループを作る場合でも、前回のループ上の method 変数が次の回でも参照される場合、例えば **Factorial** のようなものは、このモデルでは、どんどん大きくなる Petri-Net を作る。このモデルには以下のような性質がある。

[保存される性質] itBlock 変数によるオブジェクトの間の接続は保たれている。また、周回するトークンの数は、*DinnerBell* でのプロセスの数である。

[抜け落ちる性質] 同期関係にのみに着目する時、周回するメッセージの持つ状態には忘れられてしまう。メッセージの運ぶ値に基づく条件分岐は、Place から出る複数の矢印としてまとめられる。

[Petri-Net の大きさ] Petri-Net 自身の大きさは、Place、Transition、矢印の数で決まる。生成される Petri-Net は、最大はメッセージの履歴全体となる。しかし、メッセージの周回によるプロセスは、一つの Petri-Net にまとることができるので、定常的なプロセスをコンパクトに表現できる。

このモデルは並行プログラムを、「周回するメッセージによるプロセスを表現する」ものである。

6 デバッグ例

ここでは、Phased Petri-Net を利用したデバッグの例を二つあげる。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例である。もう二つ目は、メッセージ送信の順序の保存に関するバグを見つける例で、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 “Talk” である。

6.1 Dining Philosopher による例

Dining Philosopher 問題のプログラムを図 20に示す。五人の哲学者が、五つのフォークをはさんで輪になって座り、それぞれが自分の両方にあるフォークを時々とって食事をするというものである。一つのフォークが二人の哲学者によって共有されていて、排他制御を行なっている。五人の哲学者が同時に、左のフォークをとるとデッドロックが起きる。

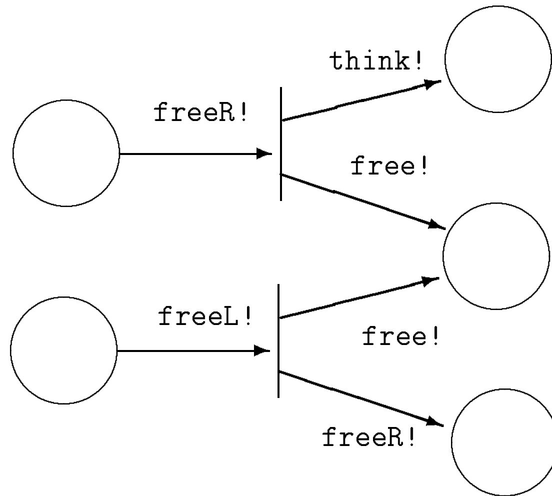

図 18: 複数のメッセージが一つの Petri-Net を共有する

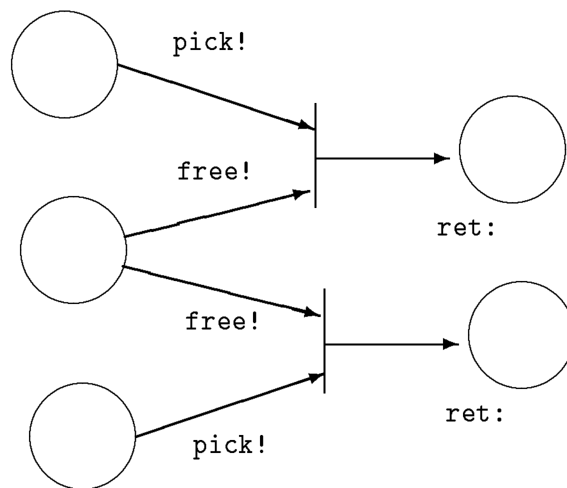

図 19: 複数のメッセージが複数の Petri-Net を生成する

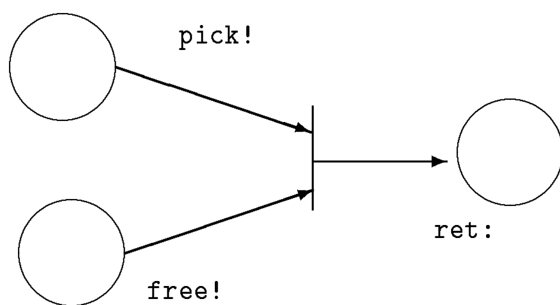

図 15: Fork のフラグメント

信 (パターンマッチによる変数の代入) に対応しておこなわれるものがある。今、図16のようなメッセージ送信がおこなわれたとする。 この時、 ˜Lfork pick! が発火したとすると、 **TRUE** と

(˜Lfork pick!) yes: (□ pickR!);

図 16: Fork の送信

いうオブジェクトが返されて、それに、 yes: というメッセージが送られる。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17になる。

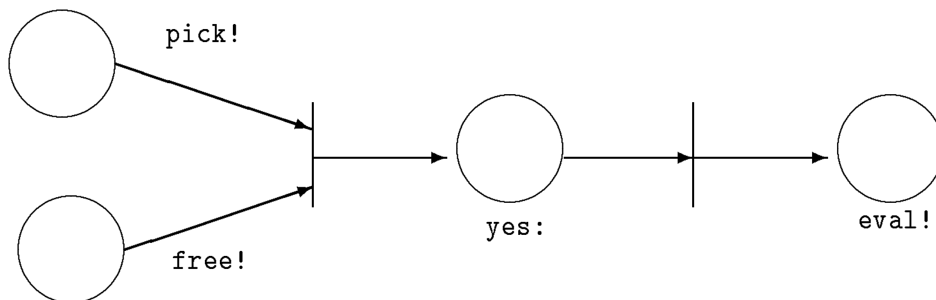

図 17: フラグメントの接続

同じオブジェクトへ同じキーワードのメッセージを複数送信する場合、同じ Petri-Net が複数個生成される。これは Method 変数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それぞれの Continuation に向かって異なる戻り値を返すからである (場合 1)。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 (場合 2)。 **Fork** の例の場合は、この二つの両方の場合が出てくる。

(場合 1) **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 (図 18)。

(場合 2) pick! のメッセージは値を返すので、二つ別々の Petri-Net を作る (図 19)。 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。

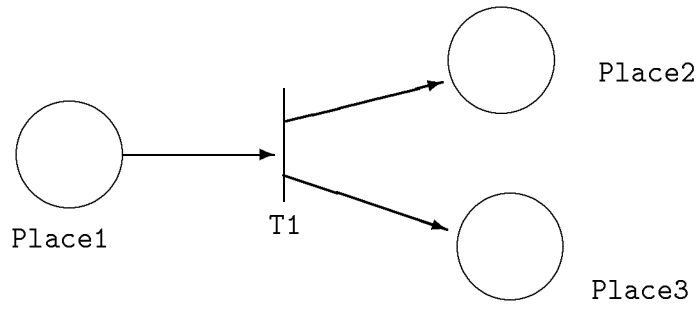

図 12: Petri-Net フラグメント

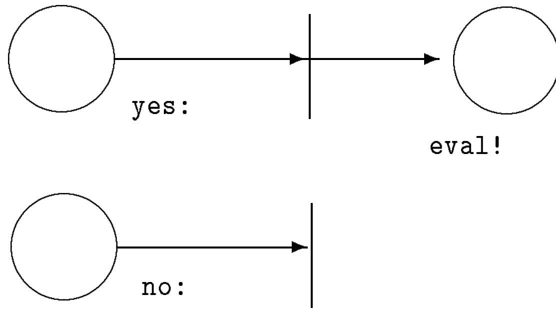

図 13: TRUE のフラグメント

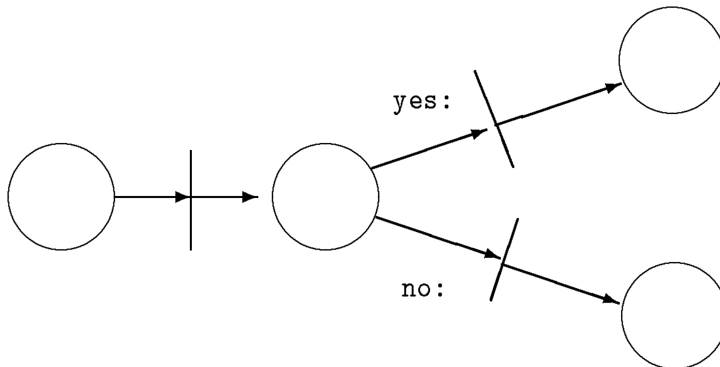

図 14: 条件文

などがある。このようなプログラムの場合には、まず、その時に増加するプロセス (= トークン) に対して、新しくオブジェクトを生成するようにすることにより、同じ働きをする有界なプログラムを作ることを考える。それができなければ、トークンの数を制限する、非有界な部分を切り出すなどの方法がある。これらの方法でも有界にできないプログラムは、非有界な Petri-Net の解析を必要とする。

5.2 モデルの定義

ここで定義する モデルは、プログラムの終了と次の状態への遷移を表す Place を付け加えた Petri-Net を使う。モデルは、以下の要素からなる。

- Place
- 状態遷移する Place (プログラムの停止も含む)
- Token
- Transition
- Place から Transition への矢印
- Transition から Place への矢印

これを、ここでは、名前と行き先のついた Place 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) があるが、ここに到達するとこの Petri-Net の役割は終了する。

5.3 モデルの生成

Petri-Net の生成のためには、*DinnerBell* の各メソッドごとに Place に、メッセージの入出力の印をつけた Petri-Net を用意する。 これをフラグメントと呼び、入出力を表す Place には、input, output の 2 種類がある。 Transition は、 *DinnerBell* のネックに相当する。例えば、図 4の output: を二つに分ける **Split** の例のフラグメントは図 12のような Petri-Net になる。 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ケンスに基づいて、それを接続していく。最後に冗長な部分を取り除く。

メッセージパッシングは次のようにして Petri-Net の対応づけられる。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。その Transition から、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 へ矢印を出す。

ここでは条件文を例にして基本的な Petri-Net の生成を説明する。条件文では、あるメッセージの答えとして、**TRUE**、**FALSE** の二通りある。図 5のプログラムは、図 13に示すフラグメントに変換される。 こらのオブジェクトに対して、 yes:, no: のキーワードにより二つの itBlock を送ると、どちらかに eval! のメッセージが送られる。これが、条件文の動作である (図 13)。この場合の Petri-Net の接続は以下のようになる (図 14)。

また MessageJoin の場合は、待ち合わせるメッセージの各々に対して、 Place を作り、その後に一つ Transition をおく。図 20のプログラムは、図 15のフラグメントに変換される。

このように、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と Petri-Net が生成できる。接続は、メッセージの送信に対応しておこなわれるものと、メッセージの受

クトへの送信が起きる。このように、通常 place から transition にはメッセージの受信を表す一つの矢印があり、transition からはメッセージの送信を表す複数の矢印が他の place へのびる。この時、MessageJoin には、transition に対する複数入力が対応する。

このようにして、メッセージの履歴を Petri-Net の列で表すことができる。これが、ここで使う実行モデルである。この Petri-Net を解析することにより、プログラムの同期関係を調べることができる。

とくにインスタンス生成の区切りの間の定常状態について考えよう。このようなインスタンスの生成がない間の Petri-Net のトークンゲームには以下のような結果がありうる。

- トークン が消滅して系が停止する (正しいプログラムの終了)
- トークン が発火できなくなって停止する (デッドロック)
- 新しくインスタンスを作る transition に到達する (次の phase への移行)
- 上記いずれにも到達せず、永久にトークンゲームが続く (永続プロセス)

最初の 3 つの状態に必ず行きつくようなものは、最後の状態だけ問題にすれば良い。この時はプログラムは逐次実行でのトレースによるデバッグなどにより解析することができる。並行プログラムの興味のある場合は、最後の永久にトークンゲームが続く可能性がある時である。例えばオペレーティングシステムや、サーバクライアントモデルのサーバ、通信ドライバ、プロセスのマネージャなどの有用なプログラムがこのクラスに属する。これらのクラスでは、非決定的実行や再現性のないバグがあり、トレースなどの既存の方法ではうまくデバッグできない。ここで述べた Petri-Net によるモデルは *DinnerBell* プログラムの部分的な定常状態、つまり、プロセス間の相互作用の解析方法を提供する。プログラム全体の意味は、複数の有界 Petri-Net の間の非決定的な状態遷移列として与えることができる。

永続プロセスを表す Petri-Net の解析は、その困難さにより、そのトークンゲーム系列において、Petri-Net 内のトークンの数に上限がある場合 (有界) とそうでない場合に分けられる。

[有界な Petri-Net] Petri-Net に対する可達問題を Place のトークン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解くことができる。可達問題が解けることは、Petri-Net に関するほとんどすべての問題が解けることを意味する。

[非有界な Petri-Net] 非有界であるということは、ある特定の Place のトークンをいくらでも増やせるということであり、非決定的なプロセス数の増加を意味する。非有界な Petri-Net に対する可達問題は解決可能ではあるが、計算量的には困難である。

ここでの Petri-Net 可達問題とは、初期状態から、ある可能なトークンの分布状態に、到達できるかどうかを調べる問題である。

有界なプログラムにより必要な並列プログラムのほとんどのクラスを表現することができる。とくに非有界性自身が、(後で示すように) 必要な順序性を破壊しているような場合があり、このような場合は非有界性の検出だけを行なえば良い。問題はバグを特定できるような、解析性の良いモデルを持つプログラムを作ることであり、有界性は並行プログラミングの一つの評価基準と考えることができる。

非有界な Petri-Net を生じるようなプログラムとしては、例えば、

- 引数 N を与えて N 個のメッセージを同じオブジェクトに対して送信するもの、
- アクノリッジを受けとらずにメッセージを無制限に生成するオブジェクト

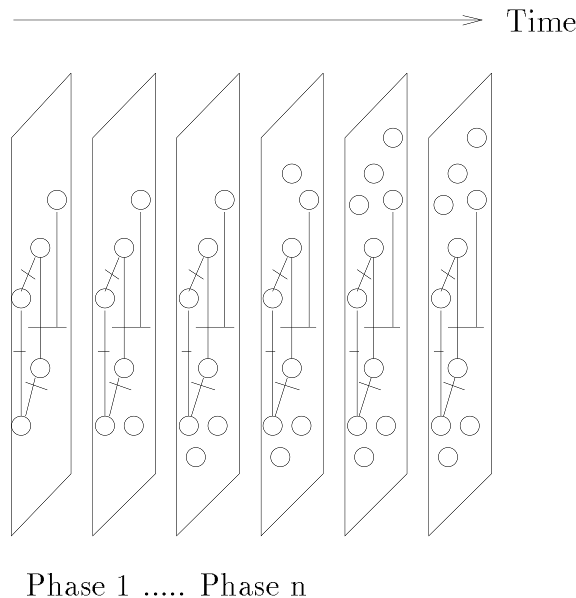

図 11: Phased Petri-Net

ここでの実行モデルとは、プログラムの各実行のインスタンスの構造を示すものである。プログラム自身は、プログラマの持っている仕様を満たさないこと (バグ) があり、プログラム自身を調べてもその不一致を調べることは難しい。なぜなら、それは一般的な検証問題であるからである。それよりも、個々の実行が、プログラマの考えている仕様と一致しているかどうかを調べる方が常にやさしい。しかし並列実行では実行モデルそのものが複雑になってしまうので、ここでは同期関係のみの実行モデルを定義している。

5.1 Phased Petri-Net

一回のプログラム実行の履歴を考える。*DinnerBell* では履歴はメッセージパッシングの集合であり、ACTOR モデル [7] と同様、各メッセージパッシングには、因果関係に基づく半順序関係がある。Petri-Net を生成していくアルゴリズムを考えるために、この半順序を含む適当な全順序を考える。半順序から全順序を生成する所に任意性があるので、実行履歴に対して一意に決まるモデルではないが、この任意性はここでは重要ではない。この全順序をたどってメッセージパッシングの集合を、インスタンス生成のメッセージを境に分割する。

DinnerBell には、2 種類の変数があり、一つは method 変数 (一時変数) で、もう一つは itBlock 変数 (インスタンス変数) である。*DinnerBell* は、単一代入であるから、itBlock 変数の値は、オブジェクトの生成以来変わらない。一方、method 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判別できる。

この時、それぞれの分割された部分は、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。メッセージの受信は、一組の place と transition となる。オブジェクトに送られたメッセージが来る場所が place であり、メッセージが受け付けられると transition を通過して、複数の他のオブジェ

のitBlockに向けて送られる。この時に前に説明したwait:を使って、出力のオブジェクトからのアクノリッジを待っている。

同じメッセージパターンは、一つのオブジェクトに一つしか許されないが、二つのメッセージパターンが同じ一つのメッセージパターンを共有してMessageJoinを構成することは許される。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。これを使って実際に状態変数を実現することができる(図10)。ここでは、state:Xというメッセージが周回する。このメッセージは、write:X、read!のどちらか一方とJoinする。ここでは二つあるContinuationのうちread!に対してのみ答えを送信している。この**Register**には、まず初期化としてstate:0のメッセー

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

図10: Register in DinnerBell

ジ送る。このオブジェクトにwrite:1が送られたとしよう。すると、itBlockのiterationによりstate:1が、自分自身に送られる。次にread!が来ると、state:1とJoinして値の1が、read!に対して返される。複数のread!が来ていたとしても、その中の一つしかJoinしない。それは、state:1が一つしかないからである。write:をいくらたくさん送ってもstate:メッセージは増えないことに注意して欲しい。

5 *DinnerBell*の実行モデル

逐次型のプログラミング言語では、実行結果を表すモデルとして、例えば、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を取ることができる。また、論理型言語では、すべての述語に対する論理変数の値の取り方の集合が実行モデルに対応する。しかし、どちらの場合でも、並列実行または、非決定性を含むような言語拡張に対しては、実行の様子を十分に表現した実行モデルにはならない。非決定性と同期機構の組合せが入出力や論理変数の値の取り方に現れない隠れた状態を作りだしていて、さらにこの隠れた状態を、別な同期機構により観測することができる。この隠れた状態は、プロセスの同期関係を表している。これは、Brock-Ackermann Anomaly[3]として知られている。しかし、このような精密な入出力を含むような実行モデルは非常に複雑になってしまう。

そこで、元のプログラムの実行履歴から、いろいろな情報を捨ててプロセス間の関係だけを抜き出したモデルを作ることを考える。*DinnerBell*は、MessageJoin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トークンゲームと考えることができる。そこで、あるPetri-Netで表された定常状態を時間軸方向に接続したようなものを考える。そこでこれをPhased Petri-Net (図11)と呼ぶことにする。

このモデルは、*DinnerBell*のある特定の実行履歴に対して生成される。つまり、同じプログラムを同じ入力データに対して実行させても、実行の一回ごとにことなるモデルができる。このモデルは、λ式に対応したモデル[12]などのように、プログラムの実行をそのままPetri-Netに変換するのではなく、実際のプログラムからPetri-Netで表されるプロセスの挙動だけを抜き出したものである。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を解析する時には、この値を無視している。

ワード wait: を使って送信すれば良いわけである。

Out2 wait: (□ to: Out2)

後の例で出てくるようにクラス **TRUE** をアクノリッジに使い、yes: を wait: の代わりに使うこともある。

4.4 MessageJoin の使い方

statementBlock と MessageJoin を組合わせて、さまざまな同期問題を *DinnerBell* でプログラムすることができる。ここで重要なことは、メッセージの待ち合わせのほかは、すべてメッセージパッシングによって成立している点である。これにより、モデルの構築が容易になっているのである。

4.4.1 セマフォ

ここでは、後で使う Dining Philosopher の例題の中で資源管理を行なうフォークを説明する。このフォークに対して、まず free! を送ることにより初期化する。フォークを使いたいプロセスは、pick! を送る。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ので、誰かが free! しなければ、pick! できない。これは、*DinnerBell* で、binary semaphore を実現する時の標準的な方法である (図 8)。この時、**TRUE** は、pick! に対応する

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

図 8: Fork in Dining Philosopher

Continuation に対して送信される。

4.4.2 マージ

複数のプロセスから来る複数のメッセージを一つのストリームにまとめることが、マージであり、例えば、入出力などに必要になる。ここでは、メッセージ対話プログラム Talk の一部の **Display** を見てみよう。まず、newFor: により、itBlock 変数 User にユーザ名を設定し、to: で Out の初期状態を指定する。そして、このオブジェクトを状態を持った to:Out というメッセージを自分に送る。つまり、この to:Out が **Display** を周回している。この周回しているメッセージと待ち合わせることによって、マージが実現される。このようにしないと Out に対して、write: と nl! が順不同に送られてしまう (図 9)。to:Out は、この中の statementBlock の中から一番外

```
class Display [
 newFor:˜User out:Out □ to:Out;
 output:X· to:Out1 □
        Out2 ⇐ ((Out1 write:˜User) write:X) nl!.
        Out2 wait: (□ to: Out2. ⇑TRUE)
]
```

図 9: Merge and Display

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

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

図 5: TRUE and FALSE

きにループは、一番内側のitBlockのselfに対して行われる。ここでは、fact: がそのようなメッセージになっている。これをメッセージの周回と呼ぶ。

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

図 6: Factorial

4.3.4 値の待ち合わせ

またブロックは、遅延評価に使うこともできる。*DinnerBell* のような単一代入型データ駆動の言語では、変数はまだ値を書き込まれていない未定の状態と、値が書き込まれた状態の二つの状態を持っている。値が書き込まれるということを値が確定するともいう。ある変数の値が確定することにより、特定のstatememntBlockの中を実行するという形で同期をとりたいとしよう。これはactive value[21] のようなものと考えて良い。*DinnerBell* では、デフォルトで図7のクラスを継承している。したがって任意の変数(最終的には、何かのオブジェクトに確定する)に、このメッセージを送ることができる。使い方は、待ち合わせたい変数に向かってキーワードwait: を使ってブロックを送信する。*DinnerBell*では、メッセージの送信は、送信先が確定するまで延期される。値が確定するとwait: メソッドが 起動され、ブロックにeval! のメッセージが送られブロックが動き出す。 このサスペンドと値の転送によって要求駆動 / 遅延評価 / 同期

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

図 7: 遅延評価ブロックを起動するメソッド

のためのハンドシェイクを行う。例えば、変数Out2の値が確定したら、to: Out2 というメッセージを自分に送りたいとする。この時には、次式のように、そのメッセージ送信をデフォルトのメッセージパターンを持つstatementBlockにつくって、待ち合わせたい変数に向かってキー

pick! · free!

というパターンならば、最初の pick! の Continuation が sender#1 であり、 free! の Continuation が sender#2 となる。 Continuation のスコープと寿命は特別に設定されている。これは次に述べる条件文では、各条件節はその親のメソッドの Continuation を共有した方が便利なためである。寿命は method 変数だが、スコープは itBlock 変数として取り扱われる。通常は受信側では以下のような形で答えが送信される。

sender#1 ret: 3

メッセージ送信時の Continuation の指定は、 $\Leftarrow$ を使う構文が用意されている。受信側の Continuation、sender#1 は、$\Uparrow$ を用いて $\Uparrow$#1、さらに、このように 1 番の場合は、単に $\Uparrow$ とすることができる。さらに、 Continuation についてはキーワードも省略を許す。メッセージパターンだけからなる statementBlock は括弧とネックを省略することができて、また、デフォルトのキーワードとして ret: が使われることになっているので、送信側は以下のようになる。 これにより、上の二つの式は以下のように省略できる。

X $\Leftarrow$ 1 +: 2
$\Uparrow$ 3

これで、 Smalltalk-80 と同様の構文に見えるようになった。またここで下線が単なる補助でなく、構文の一部を構成している。矢印自身は、 Continuation を切替えるための記号であり、その切替え先は、パターンでない、オブジェクトでもよい。同様に以下のように並列代入も実現できる。

high: H low: L$\Leftarrow$ **List split!**

答えの送信側では、次のように high: と low: の二つを同時に一度に sender#1 に送信しても良いし、

sender#1 high: H low: L

あるいは以下のように二度に渡って別々に送信しても良い。

sender#1 high: H. sender#1 low: L

また別々のオブジェクトが、 high: と low: を別々に送信しても良い。

4.3.2 条件文

DinnerBell では真理値もオブジェクトとして返される。 Smalltalk-80 と同様に、真理値に対してブロックを送りつけることにより、真の時には yes: で送られたブロックに、偽の時には no: で送られたブロックに eval! が送られて、そのブロックが動き出す。ブロックの中の変数と Continuation は、外と共有される (図 5)。

4.3.3 ループの記述

メッセージパターンが省略されたオブジェクトには、 eval! というメッセージパターンが仮定される。ローカルなオブジェクトとこの省略記法を使って、条件式を記述したのが次のこれを利用した階乗の計算の例である (図 6)。メッセージ送信式において送信先を省略すると、デフォルトとして、送信するオブジェクト自身である self が送信先となる。これがループを表す。このと

なデータフローグラフに相当する。ここでは、*DinnerBell* の2種類のことなる寿命を持つ変数が出てきている。もちろん、これらの変数は、その寿命の間、値が書き換えられることはない。Aのまえの“˜”は、AがSmalltalk-80のインスタンス変数に相当する変数で、itBlock変数であることを表す。この変数は、オブジェクト(itBlock)と同じ寿命を持ち、このクラスのように、定数や固定されたデータフローを表すことに使う。*DinnerBell*では、他にSmalltalk-80の一時変数に相当するmethod変数というものがあり、ここでのXがそれである。クラスとして宣言されていなくて、“˜”も前置されていない名前はmethod変数である。この変数の寿命はメソッド呼び出しのそれと同じであり、このオブジェクトにoutput: というメッセージが来る時に新しく作られる。したがって、単一代入変数しか持たない言語であっても同じオブジェクトに対して、output:1とoutput:2のようにメッセージを何回でも送ることができる。この例のように*DinnerBell*では、method変数で受けるメッセージパターンとitBlock変数への送信のリンクがデータフローグラフを構成することが多い。

itBlock変数を含むメッセージパターンへの複数のメッセージの到着や、変数を共有するオブジェクトでの複数のメッセージの到着などは、単一代入への違反となる。*DinnerBell*ではこのような場合エラーとしてしまうが、計算は継続する。

4.3 入れ子になったオブジェクト

()でくくることによりメッセージ送信式をネストさせることができる。例えば1 +: (2 +: 3)のような記述ができる。さらに、オブジェクトの中に()でくくった、ネックを持つ独立のオブジェクトstatementBlockを作ることができる。これは、Smalltalk-80のブロックに対応するもので、ブロックの中と外で変数は全く同じものを指す。したがってstatementBlockがその外のitBlockと同じ環境を持つということもできる。

このような環境を共有した入れ子のオブジェクトは、Lispでは、closureとして知られ、データベースの分野では、Nested Transaction[13]として研究されていている。書き換え可能な変数を使うプログラミング言語では、並列環境下で入れ子になった複数のオブジェクトを同時実行させつつ、それらを含むオブジェクトの状態の直列可能性を維持することは難しい。Smalltalk-80やABCL/1では、オブジェクトの中で実行が逐次型になるので、実装することができた。*DinnerBell*の場合はオブジェクトの中の変数が状態を持たないので、容易に矛盾なくブロックを導入することができる。

4.3.1 代入文とContinuation

*DinnerBell*ではブロックはさまざまな形で使われるが、最も良く使われるのは代入文として使うことだろう。*DinnerBell*では代入文でさえもメッセージパッシングにより実現される。*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。実際メッセージパターンだけからなるボディが空のstatementBlockをContinuationとして送ることにより代入文を実現できる。パターンマッチングが実際の代入の役割を果たしている。答えはこのcontinuation: で送られたオブジェクトに対してメッセージパッシングで返される。これは以下のように記述される。

1 +: 2 continuation: (<u>ret:X</u> □)

特にメッセージのMessageJoin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。

=:X
free! · pick!
to:A and:B
output: X

図 2: Message Pattern

非決定性が生じる。これは並列実行下の同期機構に基づく非決定性である。この場合、同期機構として使いやすくするため、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としている。図 3のように MessageJoin は、構文的にも意味的にも並列メッセージ送信と双対である。

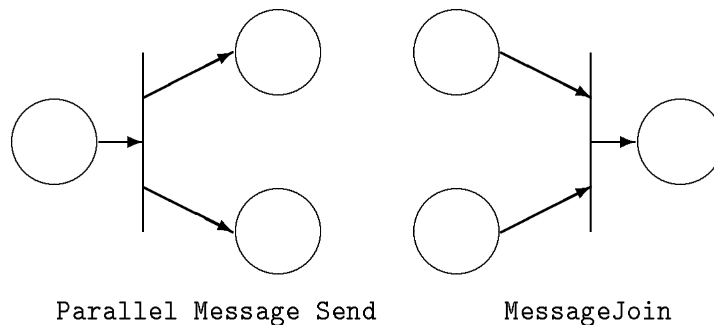

図 3: Symmetry of MessageJoin

4.2 クラス定義

図 4の **Split** は、*DinnerBell* のクラス定義の典型であり、また、*DinnerBell* のオブジェクトの典型的な使い方でもある。最初の **class Split** がクラス名を表わす。*DinnerBell* のクラスは、itBlock と呼ばれる四角括弧の中に、“;” で区切った複数のメソッドを並べたものである。メソッドはメッセージパターンとメッセージ送信の組で、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。ネックの後ろの送信式はボディと呼ばれる。

itBlock の it は iteration を表していて再帰的メッセージ送信の単位であることを表している。このオブジェクトは、A と B という二つのオブジェクトに、output:X というメッセージをコピー

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

図 4: Class Definition

して、その二つに送り込む働きをする。ちょうどこれは、A と B へトークンをコピーするよう

プログラムの中での、これらの要素の果たす役割はまったく異なる。これは、Smalltalk-80 が本質的に逐次型の言語であり、オブジェクトの状態を作るのに書き換え可能な変数を使うの対し、*DinnerBell* は、本質的に並列型の言語であって、オブジェクトの状態は再帰的なメッセージ送信により実現されるためである。

DinnerBell の全構文は、メソッド単位の分割コンパイル、3 種類のスコープ、多重継承などを含みかなり複雑である。ここでは、*DinnerBell* のプログラミングスタイルと結びつけながら、後の章で用いる例題を理解するのに必要なものだけを説明する。

4.1 メッセージ、メッセージ送信式、メッセージパターン

DinnerBell は、オブジェクト指向言語であり、その基本要素はメッセージ送信である。*Dinner-Bell* のメッセージは、イベントとキーワードの並びで表される。イベントは引き数なしのメッセージで、名前の後ろに "!" をつける。キーワードは引き数を直後に一つ持つメッセージで、名前の後ろに ":"、その後ろに引き数が一つ来る。(図 1)。名前は空白を含まない英文字の並びで、数字以外で始まるものである。またいくつかの特殊文字も使うことができる。例えば、オブジェクトが等しいかどうかを調べるメッセージは =: であり、加算を表すメッセージは +:、資源の解放には free! などを使う。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。送信先のオブジェクトとメッセージの組をメッセージ送信式という。メッセージの中のイベントとキーワードの順序性はない。つまり、to:10 from:1 と、from:1 to:10 は同じ意味を持っている。さらには to:10 だけ送るということもできる。このため、一つのオブジェクトのメソッドに to:out: と to:in: のようなキーワードの組合せが出てくるようなことは許していない。

複数の送信先への送信は、メッセージ送信式をピリオドでつなげることにより表す。*Dinner-Bell* では、クラス名、変数名には、表記上の区別はなく、別に宣言することによって区別する。ここでは見やすさのために、クラス名は太字で表す。**Split** は、二つのキーワードを同じオブジェクトへ送信している例である。クラスへの送信は、まず (暗黙の new! メッセージにより) インスタンスが作られてから、それに対して行なわれるので、*DinnerBell* にはいわゆるクラスメソッドは存在しない。output: のメッセージの例では、X を二つのオブジェクトに対して送信している。例えば A と B とが同じオブジェクトを指している変数であったとしても、*DinnerBell* では問題なく並列実行される。一つのオブジェクト自身に複数のメッセージの同時実行を許しているからである。これは、いわゆる Multi-Thread 実行である。

N =: 0
Fork free!
Split to:Display1 and:Display2
A output:X . B output:X

図 1: Message Passing

オブジェクトが、どのようなメッセージを受信するかを示すためにメッセージパターンを使う。メッセージパターンは、メッセージに下線を引いてあらわす。この下線は、他の言語にはない *DinnerBell* の構文の特徴の一つである。ここでキーワードの引き数は変数でなければならない。図 1 に対応したパターンが図 2 である。

この free! の例のような、ドットで区切られたメッセージパターンは、MessageJoin を表わす。MessageJoin とは、複数の送信者の送ったメッセージ同士の待合せである。もちろん同一の送信者が二つの待ち合わせのメッセージを送ってもよい。どのメッセージと待ち合わせるかによって

これにより、オブジェクトの持つ変数は書き込みの競合のないものとなり、入れ子になったオブジェクトや、変数のスコープに関連する問題も、並列実行を前提にしているにも関わらず、単純に参照をコピーすることにより取り扱うことができる。したがって、オブジェクトの入れ子、使い捨てのローカルなオブジェクトなどを導入している。これは、他の並列オブジェクト指向言語にはない特徴である。

変数はこの意味で常に同じものを指すので、他の並列オブジェクト指向言語のように、オブジェクトの状態を表す変数ではない。Smalltalk-80[22] のインスタンス変数、メソッド変数に対応するものはあるが、*DinnerBell* ではこれらの変数自身はオブジェクトの状態をつくり出すことができない。*DinnerBell* では、オブジェクトの状態を実現するためには、再帰的なメッセージ送信を用いる。つまり状態の値を持ったメッセージを自分自身に送ることで状態を作る。これは、データフロー言語では標準的な方法である。また、変数には必ずオブジェクトが格納され、その値も変化しないのでオブジェクトと変数を区別せずに扱っても問題ない。ただし、変数は常に同じオブジェクトを指しているが、それが状態を持つオブジェクトである場合は同じ変数に同じメッセージをおくっても、同じ値が返るとは限らない。

このような再帰的メッセージ送信で状態を作るのに必要な同期機構としては、複数のメッセージの待ち合わせ MessageJoin を使う。これは、通常の一対一の値の待ち合わせとことなり、非決定性を持つ同期機構である。データフロー言語ではマージを直接導入することも多いが、ここでは、メッセージ送信を直接、データフロートークンと対応させる。これにより、トークンの順序を保存するような静的データフロー [5] に対応する通常のメッセージ送受信と、トークンの順序を保存しない動的データフローに対応する MessageJoin を使い分けることができる。もちろん静的データフローといっても、実際にはオブジェクト指向言語であるので、全体のデータフロー自身は実行時に定まることになる。これにより、メッセージと、データフロートークンを対応させた言語を実現できたことになる。つまり、*DinnerBell* は、

- メッセージの待ち合わせを同期機構に使い、状態はストリームで実現する

言語である。この方法により、一つのプロセスは、(いくつかの) オブジェクトとそのオブジェクトを再帰的メッセージ送信により周回するメッセージからなる。この意味で、オブジェクトとプロセスは対応しているが、別な概念である。

また、計算モデルを簡単に保つために、すべてをメッセージパッシングで表し、for 文、do 文、while 文のような特別な制御構造も排除している。効率的に問題はあるが、**TRUE**、**FALSE** のような分岐を担当するオブジェクトもユーザ定義可能にしている。変数への値の代入はメッセージの受信の時のみ行なわれる。代入はスコープを共有するローカルなオブジェクトに対する送信として表す。ただし、Continuation をサポートする構文を用意し、構文的に手続き型言語に近いものを実現している。またクラスとインスタンスの区別は意図的になくしてある。これは、クラスメソッドとして、インスタンスを生成する new! しか存在しないと考えてもよい。これも、実行モデルを単純化するためである。したがって、この先、クラスとオブジェクトは、区別なく取り扱う。

DinnerBell でのプログラムの実行は、メッセージパッシングによるデータフローグラフの構築と、それにそって走るメッセージと変数の値の二重構造を持っている。

4 *DinnerBell* の構文と意味

ここでは、*DinnerBell* の構文規則と意味について述べる。*DinnerBell* は、構文的には Smalltalk-80[22] に近くなるように設計されている。例えばメッセージ送信やインスタンス変数のような従来のオブジェクト指向言語に対応するものがあり、その表面的な意味も似ている。しかし、並列

ことである。ただし、この様な例示を有限個積み重ねるのではプログラムの意味をすべて決めることはできず、これをプログラムの検証に用いることはできない。

本論文で用いる方法は、ある種の Petri-Net をモデルに使って、並列プログラムに対して系統的なデバッグを行なうものである。この Petri-Net により、コインとジュースのように単純にはいかないが、複雑な同期関係を視覚的に見ることができ、また、一見してわからないような条件 (例えばデッドロックや有界性) などについても適当な方法を用いて調べることもできる。

並列プログラムのデバッグの難しさはいろいろある。

- 非決定性から来るバグの事象の非再現性。
- トレースのような逐次型のデバッグを不可能にしている同時実行の性質。
- デバッグの対象として不定個数のプロセスが存在する。

ここでいうプロセスは、因果関係で結ばれた連続した並列処理の単位である。ある決まったアルゴリズムに基づいて、これらのプロセスは他のプロセスと相互作用を行なう。特に *DinnerBell* では、一つのプロセスはオブジェクトを周回するメッセージで表される。プロセスの間の定常的な同期関係がここでは知りたいことであり、これを関数の入力と出力の組で表すような単純な方法は使えない。そこで、定常的な同期関係を保存したモデルをつくって、それを解析する方法をとる。この定常的な同期関係を表した Petri-Net を調べることにより、再現性のないバグを見つけることを可能にする。

また、履歴を用いたデバッグ方法により、並行実行をむしろデバッグの有効な手段とすることができる。例えば、履歴を取りながら、同時にそれを調べることができる。このモデルでは、プロセスの数は Petri-Net 上のトークンの個数であり、 Petri-Net の形に依存しない形できる。プロセスが Petri-Net の形を変えないような定常的な状態ならば、不定個数のプロセスが存在する場合にも同期関係を調べることができるのである。

仕様と実装の関係を考える時にプログラム合成も重要である。合成では、実装と異なり、仕様 (この場合は Petri-Net) を与えて、それから、プログラムを自動生成する。

定義 3 (合成) 仕様を、その仕様を満たすプログラムに変換する。

もちろん、以下で述べる言語仕様をみれば、わかるように、Petri-Net から、*DinnerBell* への変換は容易で、 Petri-Net を用いたプログラムの検証 / 合成も考えられるが、それは、この論文では扱わない。このような合成は例えば、[29] に述べられている。

3 データフローとオブジェクト指向を調和させたプログラミング言語 *DinnerBell*

ここでは、 *DinnerBell* [31, 30] の設計方針について述べる。 *DinnerBell* の基本的なアイデアは、オブジェクト指向とデータフローを結びつけることである。従来データフローに基づくプログラム言語では、複雑な構造を持つデータの処理という点では問題があった。これをオブジェクト指向に基づくプログラム言語の持つ豊富なデータ構造表現とモジュール化で解決できるだろうと期待される。最近では、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 [26, 6, 11]、 *DinnerBell* [31, 28] は、もっとも単純な言語仕様を持ち、より効果的な実行モデルを作ることができる。

DinnerBell では、単一代入則を導入して、データフローの概念にそった実行を行なう。つまり、ある変数には一回しか代入できず、もしまだ書き込まれていない変数を読み出そうとした時には、値の待ち合わせが起こる。これが、もっとも基本的な *DinnerBell* の同期機構である。

選択”という非決定性(don't know non-determinism)があるが、これを“不可逆的な可能な実行の選択”である非決定性(don't care non-determinism)に置き換えることにより、Commited Choice型言語が実現される。ここでどのような選択機構を使うかが並列プログラミング言語としての非決定的な動作を決める。つまり、この部分が論理型言語の実行モデルでは説明されない部分である。(本論文で使われる非決定性という用語は後者の意味で使っている。)これに対する一つの解決方法として、最近では変数の代入の因果関係や、suspend setを使ったモデル[17, 19, 15]も研究されている。

三つ目のものは、オブジェクト指向言語に対する計算モデル[20]である。これは同期機構としてポートを用いる方法を使っている。しかしこの方法はオブジェクト指向言語としては不自然である。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。

四番目のものは、ストリームを積極的にオブジェクトとしてとらえたもので、まだ、モデルに関する研究などはないが、並列オブジェクト指向言語のモデル、または、Committed Choice型言語のモデルを使うことが考えられる。

これらのモデルは一部の理論的な研究を除いて、直接的にプログラムのデバッグに用いることはできない。プログラムの実際の実行から、有用な情報を抜き出すということが実用的な面からは重要である。*DinnerBell*の同期機構は、プログラムの同期関係のみを表すような宣言的なモデルを作ることができるという利点がある。このモデルを使ってデバッグを行なうことができる[32, 33]。

2　検証、合成、デバッグ

プログラムのデバッグと検証は、仕様記述、実装の記述、そのモデルの関係、として表すことができる。仕様記述は、実際には何らかの言語で記述されるのが普通である。これ自身をプログラムとして考えることも、実行可能仕様記述として知られている。実装はまた、別なプログラミング言語を用いて行なわれる。この時、実装が仕様を満たすということは、

定義 **1** (検証) 実装によって生成されるモデルが、必ず仕様を満たすことを調べる

ことである。これは、例えば通常良く使われる論理式による仕様記述では、通常ある種の恒真式を証明すること(Validity)に帰着される。これがプログラムの検証である。Tokio[27]で使われたのは、仕様として時相論理式を使い、実装にも時相論理式を使う方法である。

しかし、実際には、仕様記述自身にも誤りが入ることは避けられず、デバッグが不要になることはない。しかし、もし宣言的なセマンティクスに対するモデルがあれば、そのモデルが正しいものかどうか判断することは、比較的容易である。なぜなら、そのようなものは、プログラムの実装から離れた「コインを入れると、ジュースが出てくる」のような具体的な例示であり、これに対して、プログラム作成者は、何らかの知見を前もって持っているからである。ここで宣言的という性質が、プログラムの実装から離れた判断を可能にしている。例えば、あるプログラムに対してturing machine上の実行モデルを考えたとする。この時、その個々の実行の良し悪しを判断することは難しい。なぜなら、その実行はturing machine上の実装に依存しているからである。

Algorithmic Program Debugging[16]は、このようにプログラムの実行にそって、モデル構築をし、その正否をユーザに問うことによって、プログラムをデバッグしていく方法である。つまりデバッグとは、

定義 **2** (デバッグ) 実装によって生成されたモデルを調べることにより、実装/仕様を修正する

は、複数のオブジェクトへのメッセージパッシングを並列実行の表現とする並列オブジェクト指向言語 ABCL[24], ACT3[1], ORIENT84/K[9], ConcurrentSmalltalk [23] があり、暗黙的なものとしては、Lucid[2] などのデータフロー [5] に基づく言語がある。これらの言語は、なんらかの明示的な同期関係記述を持っている。副作用のない並列関数型言語の場合は同期関係記述は暗黙的である。

DinnerBell は、暗黙的な並列実行表現と、明示的な同期関係記述を持つ言語であり、副作用のないメッセージパッシングに基づく実行モデルを持っている。このモデルを使い定常的なプロセス間通信をデバッグする方法を示す。

この論文は、*DinnerBell* 言語の設計方針に関する考察、モデルの提示、そのモデルを用いたデバッグ例という構成になっている。以下では、プログラミング言語の同期機構に関する考察を行なう。2 章では、デバッグと実行モデルの関係を考察する。3 章で *DinnerBell* の設計方針を述べ、4 章で言語仕様とプログラミング手法について述べる。5 章では同期機構を表現するための *DinnerBell* の実行モデルとその生成方法を述べる。6 章で、実行モデルを用いたデバッグ例を示し、7 章で他のモデルや手法などとの比較検討を行なう。

1.1 並列プログラミング言語の同期機構

同期機構と状態の作り方には、様々なものが考えられてきた。

1. 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)
2. Guard を同期機構に使い、状態はストリームで実現する (Parlog[4],GHC[18])
3. Port を同期機構に使い、変数は状態を持たない (μ-Calculus[20])
4. 状態はストリームで実現し、ストリームのマージを同期機構に使う。(Vulcan[10], A'UM [25])

並列プログラムの同期機構は、並列実行から生じる非決定性の制御にほかならない。基本的な同期機構があれば、他の複雑な同期機構を構成することはできるので、これらの同期機構に表現能力的には差はないといってよい。これらの評価は、並行プログラムの検証、合成、デバッグの手法に密接に結び付いている。とくに重要なのは実行モデルとの関係である。

あるプログラミング言語で書いたプログラムがあるとする。このプログラムを実行することにより関数呼び出しや代入などが起きる。このようなプログラムの実行の様子を表したデータ構造が実行モデルである。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。このモデルの中でプログラムの意味を実際のプログラムから離れて定義できるようなものを宣言的といい、より直接的にプログラムの実行を忠実にモデル化したものを操作的あるいは、計算モデルという。モデルとしては宣言的な方が望ましいのだが、並列プログラムでの有用な宣言的なモデルはまだない。

これらの同期機構の中で、一番目の入力を直列化してしまうものの代表は、いわゆる並列オブジェクト指向言語であり、特にオブジェクトの中では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、動作している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。このモデルはその性質上、操作的なものであり、これを用いて直列可能性などを議論する [8]。

二つ目のものは、Committed Choice 型言語 [18] などで使われていて、並列動作する再帰呼び出しにより変数のリスト (ストリーム) を作り、それにより通信を行なう。しかし、これらには共通の問題として、基になった論理型言語の持つ宣言的な実行モデルが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。論理型言語にはもともと "可逆的な可能な実行の

並列オブジェクト指向言語 *DinnerBell* の 実行モデルによるデバック手法

河野真治 * 青柳龍也 ** 田中英彦

Department of Electrical Engineering, The University of Tokyo
7-3-1 Hongou, Bunkyo-ku, Tokyo 113, Japan
* 現在 Sony Computer Science Laboratory Inc.
** 現在 電気通信大学情報工学科

デバッグを系統的に行なう手法として、プログラムの実行モデルを入出力関係の集合として構築し、そのモデルを調べることによりプログラムのバグを発見する Shapiro の Algorithmic Program Debugging[16] が知られている。しかし並列プログラミングでは、プロセスの同期関係を個々の入出力関係のみに基づいてプログラマがその適不適を判断することは難しい。

並列オブジェクト指向言語 *DinnerBell* [31, 30, 32, 33] では、高並列プログラミングに適した、メッセージの待ち合わせ機構と単一代入を同期機構として採用している。この同期機構の性質を使い、並列プログラミングでは難しいとされてきたデバッグに新しい手法を導入することができる。

DinnerBell の同期機構は、同期関係を Petri-Net として表現することができる。この Petri-Net を調べることにより、プログラマの求めた同期関係になっているかどうかを判断する。この方法を用いて、永続プロセスでのデッドロックやメッセージの順序非保存性など、再現性の低い並列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

知識処理とは基本的には記号処理のことであり、構造を持った情報である知識の表現とそれに対する処理 (ルール) の記述を、効果的に行なうことがプログラミング言語に要求される。この時、知識とルールは大量にあると考えられ、それに基づいて高並列処理が可能であると考えられてきた。しかし、知識処理を高並列処理をしていく際には、実は、 OR 並列 Prolog[14] のような単純なアプローチでは不十分で、実際の高並列を目指すプログラミング言語には、以下のようなことが必要であることがわかってきた。

- アルゴリズムに内蔵される並列性を自由にかつ容易に表現できること
- 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、デバッグの手法があること

この時に並列実行を表現する方法として、明示的なものと暗黙的なものが考えられる。同様に、同期関係の記述にも明示的なものと暗黙的なものが考えられる。ある。例えば、明示的なものに