

TM-0595

ペトリネットと時制論理による
並列プログラミング言語

内平直志, 藤原睦, 飯島正
本位田真一(東芝)

June, 1988

©1988, ICOT

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

ペトリネットと時制論理による 並列プログラミング言語

内平直志¹, 藤原睦², 飯島正³, 本位田真一¹

1 株式会社東芝 システム・ソフトウェア技術研究所

2 株式会社東芝 総合研究所

3 慶応義塾大学 理工学部

並列プログラムの自動生成および検証を目的とした言語MENDEL/88の言語仕様について報告する。MENDEL/88はペトリネットをベースとしたストリーム通信型並列プログラミング言語である。MENDEL/88のプログラムの骨格はMENDELネットと呼ぶ限定ペトリネットで見えられ、プロセス間の同期がトランジションの発火制御という形態で実現される。すなわち、同期の問題をトランジションの発火手順の問題に置き換えることができた。手順が有限の場合の手順生成には時制命題論理が有効である。トランジションの発火を原子命題とみなすと、発火手順に対する仕様が時制命題論理で記述できる。MENDEL/88の同期部は、時制命題論理に基づく仕様記述言語TSLから自動生成することができる。

CONCURRENT PROGRAMMING LANGUAGE USING PETRI NET AND TEMPORAL LOGIC

Naoshi Uchihira*, Mutumi Fujihara**, Tadashi Iijima***, and Shinichi Honiden*

*Systems & Software Engineering Lab., TOSHIBA Corporation

Yanagicho 70, Saiwai-ku, Kawasaki 210, JAPAN

PHONE: (044) 548 - 5465

** R & D Center, TOSHIBA Corporation

*** KEIO University

A concurrent programming language, called MENDEL/88, are proposed. MENDEL/88 is designed for the program synthesis. The MENDEL net, which is a restricted Petri net, automatically generated from MENDEL/88 source programs. The MENDEL net provides a semantics of MENDEL/88. The MENDEL/88 program has the body parts and the synchronization parts. The synchronization part is synthesized from temporal logic specifications.

1. まえがき

本報告では、ペトリネットをベースにした並列プログラミング言語MENDEL/88を提案する。MENDEL/88は、並列プログラムにおける自動プログラミングを目的とした言語である。

設計の動機

我々は、並列プログラムのプログラミング支援環境を研究、開発している。現在、とくに並列プログラムの自動生成、検証に注力している〔内平87〕。

プログラムの自動生成の目的には次の2つがある。

①生産性の向上

②正しいと保証されたプログラムの作成

目的によって、そのアプローチの方法も異なってくる。①に対しては、部品やパターンの再利用が主流であり、②に対しては、仕様記述言語から変換、生成する方法が多い。

また、生成の対象が、逐次プログラムであるか、並列プログラムであるかによって目的①と目的②の比重が異なってくる。

逐次プログラムの場合、現在まで多くの研究、開発が行われてきた。部品やパターンの再利用によるアプローチの中には、十分実用化レベルに達しているものがある〔小宮88〕。成功例の多くは、事務処理のように、定型パターンが存在するような分野が対象である。これらは、人間のプログラミング作業における、あまり本質的でない部分の労力を軽減することが目的であるといえる。すなわち、①の目的を達成している。

並列プログラムの場合、同期部分とそれ以外の部分（本体部分と呼ぶ）を分離して考えるべきである。本体部分の生成は逐次プログラムの手法が応用できるが、同期部分の自動生成は逐次プログラムの場合と本質的に異なる。同期部分のプログラミングは、人間にとって非常に困難である。なぜならば、全てのタイミングの可能性を考慮しなければならないからである。また、再現性のないバグが多く、テスト、デバッグが難しい。すなわち、同期部分の自動生成の最大の目的は②であるといえる。

タイミングを扱う同期部分の自動生成には、時制論理が適している。特に、時制命題論理は決定可能であり、種々の決定手続きが提案されている。そこで我々は、同期部分の仕様記述言語として時制命題論理を用

い、その仕様から同期部分を自動生成するというアプローチを選択した。また、本体部分は部品再利用によって生成するという方針を取った。

このような、自動プログラミングを実現するにおいて、従来の並列プログラミング言語では、ギャップが大きかった。とくに、時制命題論理でそのまま記述可能な同期機構を持った言語は少ない。Manna と Wolper〔MW84〕は、時制命題論理を用いたCSPプログラムの自動生成を行なった。この方法では、メッセージの送受を原子命題とみなし、各プロセスが中央のスケジューラプロセスを介してお互いにメッセージ交換するというCSPプログラムの同期部分が生成される。時制命題論理を使う以上、スーパーバイザによるスケジューラ機能は必要であるが、これがプロセスとして前面にでるのは、本体部分を作成しなければならないユーザにとって気持ちが悪い。スーパーバイザが前面に出ず、時制命題論理で記述可能な同期機構を持つ言語を新しく設計する必要がある。

また、本体部の自動生成を考慮したとき、ストリーム通信による並列プログラム（Occam, GHC, Stella〔久世88〕など）が有望である。つまり、並列オブジェクト指向言語ABCL〔柴山88〕のような自由自在にプロセス間のメッセージ交換が可能な言語は、自由度が高すぎて自動生成は困難である。ストリーム通信のように、あらかじめ通信路が限定されているほうが扱いやすい。従来から我々は、再利用部品としてのプロセスを部品ライブラリから検索し、プロセス間のストリーム通信路を設定することは、部品検索、結合による並列プログラムの自動生成であるという観点から研究を進めてきた〔本位田85, 内平86〕。Barstovも、仕様からプログラム変換により実行可能なプログラムを生成する方法を、ストリームによる並列プログラムに適用して成功している〔Barstov88〕。

以上のような背景から、今回提案する並列プログラミング言語MENDEL/88は次の方針で設計された。

①時制命題論理で記述可能な同期機構を持つ。

②固定の通信路によりプロセス間の通信を行なう。

設計の概要

並列システムのモデル化には、ペトリネットが広く使われている。MENDEL/88は、このペトリネ

ットをベースとする並列プログラミング言語である。すなわち、プログラムの骨格がベトリネットで表現される。これを、MENDEL ネットと呼ぶ。プログラムの同期（の一部）は、ベトリネットにおけるトランジションの発火制御として実現できる。そして、トランジションの発火順序に対する制約条件を時制命題論理で記述し、定理証明の手法によって具体的手順を生成する。

また、ベトリネットのアークを通信路とみなせば、プロセス間通信もプロセス間の通信路をトークンが移動するという形で表現される。ベトリネットにおいて、通信路は固定である。

このように、ベトリネットと時制論理をうまくジョイントさせることにより、目的の並列プログラミング言語を設計することができた。

本報告では、2章でプログラムモデル、すなわちMENDEL/88におけるベトリネットと時制論理の関係を説明し、3章で言語仕様を与える。4章、5章で時制命題論理に基づく同期部分の仕様記述言語（TSL）を定義し、TSLで書かれた仕様から同期部分を自動生成されることを示す。再利用部品による本体部の生成については、文献〔伊藤88〕にあるので省略する。最後に6章で哲学者の食事の記述例を示す。

2. プログラムモデル

MENDEL/88におけるプログラムの基本モデルを説明する。

データフローと制御フロー

並列システムは、いくつかのプロセスとプロセス間のデータフローおよび制御フローによって表現できる。最近話題のリアルタイムSAもこの原則に基づいている。

ベトリネットでは、データも制御も全てトークンの移動によって表現している。しかし、データとコントロールは区別されたほうがよいし、複雑な制御を記述する場合はフローが複雑になって読みにくい。ところが、制御フローの多くはデータフローのトランジションの発火を制御することで代替可能である。この発火を制御するプログラムを、並列プログラムの同期部分とみなすことができる。この発火制御を導入することにより、ベトリネット自体はデータフロー中心になり

読み易くなる。

時制論理

何も制御しなければトランジションはトークンの存在の有無によって発火し、トークンは自由に流れる。ゆえに、発火制御の要求のほとんどは、制約や禁止である。このような場合、制約や禁止を実現する具体的な発火手順を求めるのは人間にとって容易ではない。

一方、時制論理は制約や禁止を記述するのに適している。そこで、トランジションの発火を時制論理における原子命題とみなし、発火制御プログラムの仕様を記述する。そして、発火制御プログラムは仕様から自動生成される。ベトリネットと時制論理の関係を図1に示す。

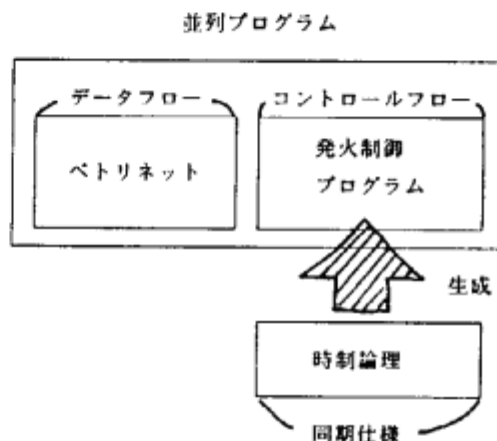


図1 ベトリネットと時制論理の関係

オブジェクト

並列システムをベトリネットで表現した場合、プロセスは、ベトリネットの部分ネットとして表されている。この部分ネットをオブジェクト（原子オブジェクト）と呼ぶ。オブジェクト間の通信は、1入力1出力のトランジションによって行われる。これは1対1の1方向通信路である。オブジェクト間を通信路により結合することを配管と呼ぶ。

発火制御プログラムはオブジェクト間のトランジションに対して1つ与えられる。配管されたオブジェクトと発火制御プログラムで新しいオブジェクトを階層的に構成できる。このようなオブジェクトを複合オブジェクトと呼ぶ。オブジェクトの階層性は、図2のようなイメージとしてとらえられる。

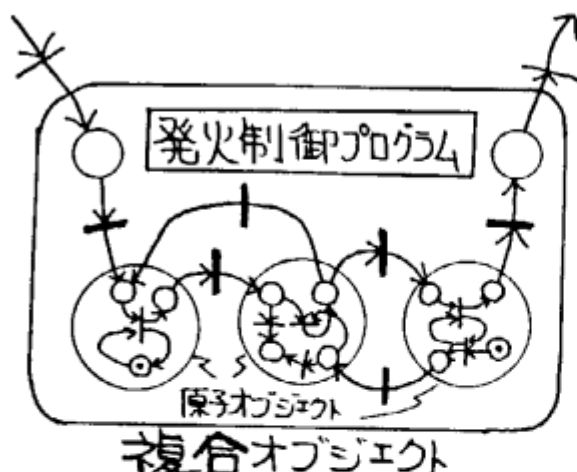


図2 オブジェクト間の階層的構成

3. MENDEL/88の言語仕様

MENDEL/88 (以下、MENDELと呼ぶ)の言語仕様を示す。構文および意味の説明をするともに、プログラムの骨格が限定されたベトリネット (以下、MENDELネットと呼ぶ) で表現されることを示す。

3.1 プログラム

(構文)

<プログラム> ::=

<原子オブジェクト> | <複合オブジェクト>

(説明)

MENDELプログラムは、オブジェクトである。オブジェクトには原子オブジェクトと複合オブジェクトがある。

3.2 原子オブジェクト

(構文)

原子オブジェクトは宣言部、メソッド部、ジャンク部から構成される。

```
atomic object <オブジェクト名> : {
    dec: [ <宣言部> ] ;
    meth: [ <メソッド部> ] ;
    junk: [ <ジャンク部> ]
}
```

1.

宣言部

宣言部は4つの項目から成る。

1) 外部入力ポート

inport (<ポート名>: <属性> , ...) ;

2) 外部出力ポート

outport (<ポート名>: <属性> , ...) ;

3) 外部データ変数

exdata (<ポート名>: <属性> , ...) ;

4) 内部状態変数

openstate(<ポート名>: [状態 , ...] <初期状態> , ...) ;

state(<ポート名>: [状態 , ...] <初期状態> , ...) ;

5) 内部データ変数

data(<ポート名>|<初期値> , ...) ;

メソッド部

メソッド部には、0個以上のメソッドから成る。各メソッドの構文を次に示す。

method(<ポート名> ? <項> , ... ,

<ポート名> ! <項> , ...)

← <ガード部> | <Prologのゴール列> ;

または、method(<ポート名> ? <項> , ... ,

<ポート名> ! <項> , ...);

ジャンク部

ジャンク部の中はPrologプログラムである。これらはメソッドから呼び出される。

(説明)

宣言部

1, 2を外部入出力ポートと呼び、4, 5を内部変数と呼ぶ。外部入出力ポートを通して、外部とのメッセージの送受信を行なう。外部データ変数は、その変数と配属された外部の共通変数の値を参照できる。外部入出力ポートとの相違点は、外部入出力ポートはバッファであり、メッセージがFIFOで送受されるのに対し、外部データ変数は時々刻々と変化する外部変数を参照するだけである。この機能により、リアルタイムシステムにおける、温度や圧力などの外界からのデータの参照を記述できる。内部状態変数は有限の状態を持ち、その状態を示すアトムをリスト

[]の中に明記する。!の後に初期値を書く。例えば、onとoffの状態を持つ内部状態変数switchは、

state(switch:[on.off]!off) ;

のように宣言される。特に、openstateで宣言された内部状態変数は後に述べる同期部から参照可能である。内部変数の取りうる値が有限でない場合は、内部データ変数として宣言する。この場合、状態集合を書く必要はなく初期値だけでよい。

例 data(temperature! 20) ;

外部入出力ポートおよび外部データ変数につけられた属性は、GARNET [伊藤88] による部品の自動検索/結合に利用される。属性は省略してもよい。

メソッド部

メソッドにおいて、<ポート名> は外部入出力ポート、内部変数のいずれかである。“?” は入力，“!” は出力をあらわす。“!” はコミットメント・オペレータであり、ガード部とそれ以降を区別する。<項> はPrologの項と同じである。ただし内部状態変数の後の項は宣言した状態アトムの子孫でなければならず、変数は許されない。

各メソッドは、外部入力ポートからメッセージを取り込み、あるいは内部変数を参照し、ガード部の起動条件が満たされればコミットメント・オペレータ以降のPrologのゴール列を実行する。そして実行の最後に外部出力ポートへのメッセージの送出、及び内部変数の更新を行う。ガード条件の評価は、上から順に行われる。

例 method(hour?H.minute!M)
 ← H>=0 | M is H#60.write(M).nl ;
 method(switch?off.temp?T.switch!on)
 ← T>100 | true ;
 (ここでswitchは内部状態変数)

<Prologのゴール列> の中のゴールが失敗した場合は警告を表示し、メッセージの出力及び内部変数の更新を行わずにそのメソッドを終了する。“←” 以下がない場合は、“← true | true ” が省略されたとみなす。

(MENDELネットでの表現)

原子オブジェクトにおけるメソッドは、プロダクションルールである。また、プロダクションルールとベトリネットは基本的に等価とみなすことができる。ゆえにメソッドはトランジションとして表現できる。

原子オブジェクトの各要素は、MENDELネット表現では次のように表現する。

- 内部変数 (プレース)
- 外部入出力ポート (プレース)
- | メソッド (トランジション)

外部入出力ポートと内部変数は、MENDELネッ

トにおけるプレースである。すなわち、メソッドの入出力プレースは外部入出力ポートおよび内部変数である。内部状態変数は、各状態アトム毎に別のプレースとみなす。内部状態変数の参照をせずに更新だけ行なっているメソッドは、全ての状態に対してそれを参照する等価なメソッドが存在すると解釈する。内部データ変数は1つのプレースで表わす。内部データ変数については値による区別はせず、必ずトークンを保持するように自分自身へのアークを作る。原子オブジェクトとそのMENDELネット表現の例を図3に示す。

```
atomic object エアコン :{
  dec:{
    inport(開始,温度変更);
    openstate(状態:(入,切)|切);
    data(温度!0)
  };
  meth:{
    method(開始?X,状態?切,状態!入);
    method(状態!入,温度?T,状態!切)
      ← T < 20 | write_message ;
    method(温度変更?T,温度!T)
  };
  junc:{
    write_message :-
      write('サーモスタット作動'),nl
  };
};
```

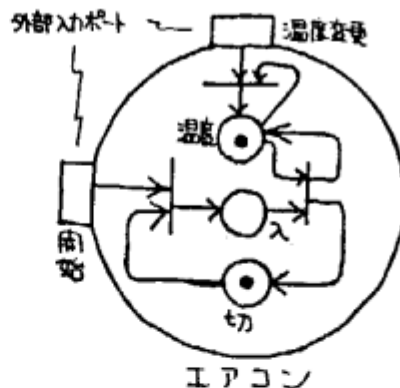


図3 原子オブジェクトの例

3.3 複合オブジェクト

(構文)

各オブジェクトのポートを通信路 (以下、パイプと呼ぶ) により結合することを配管と呼ぶ。複合オブジェクトは、宣言部、配管を行なう本体部、ゲートの発

火順序を記述する同期部から構成される。

```
object <オブジェクト名> : {  
  dec: { <宣言部> } ;  
  body: { <本体部> } ;  
  sync: { <同期部> }  
};
```

宣言部

宣言部は2つの項目から成る。

1) 入力ポート宣言

```
inport ( <ポート名>: <属性> , ... ) ;
```

2) 出力ポート宣言

```
outport ( <ポート名>: <属性> , ... ) ;
```

3) ゲート宣言

```
agate ( { <ゲート名>, <ゲート名>, ... } ) ;
```

```
sgate ( { <ゲート名>, <ゲート名>, ... } ) ;
```

本体部

本体部では、通信パイプでオブジェクト間を配管する。

```
innode( <ポート名> ? [ <ゲート名>, ... ], ... ) ;
```

```
outnode( <ポート名> ! [ <ゲート名>, ... ], ... ) ;
```

<オブジェクト名> (

<ポート名> : [<状態>, ...], ...

<ポート名> ! [<ゲート名>, ...], ...

<ポート名> ? [<ゲート名>, ...], ...) ;

<オブジェクト名> (

<ポート名> : [<状態>, ...], ...

<ポート名> ! [<ゲート名>, ...], ...

<ポート名> ? [<ゲート名>, ...], ...) ;

.....

同期部

ゲート発火規則を次の有限オートマトンの形式で与える。ゲート発火の系列は無限系列であり、終了状態を持たない。

<同期部> : = <ゲート発火規則>

<ゲート発火規則> : =

(<状態遷移規則の集合>, <初期状態>)

<状態遷移規則の集合> : =

[<状態遷移規則>, ...]

<状態遷移規則> : =

trans (<現在の状態>, <遷移後の状態> ,

<ガード条件>, <ゲート名>)

(説明)

宣言部

入出力ポートの定義は原子オブジェクトと同様である。ゲート宣言で、各ゲートがメッセージゲート (agate) あるいはシグナルゲート (sgate) として宣言される。このゲートは、本体部および同期部で使われる。

本体部

本体部はinnode, outnode, およびオブジェクト間の配管を記述したものである。オブジェクトの記述で、各オブジェクトの参照可能な内部状態変数とその状態集合を“:”でつなげて記述する。“!”および“?”はオブジェクトの外部入出力ポートの記述であり、パイプはゲート名で同定され、同じゲート名をもつ入出力ポート間は配管される。1つのポートに複数のパイプが配管可能なのでリスト形式で表す。配管部に書かれた各オブジェクトはAND並列に動く。Innodeは入力ノードであり、実行時に外部からの入力メッセージを<ポート名>?の後のゲートのうち発火したところへ送信する。outnodeは出力ノードであり、実行時に<ポート名>!の後の発火したゲートからメッセージを受信し外部に出力する。

同期部

MENDELではもともと、メソッドの選択機構に基づく簡単な同期制御が可能である。つまり、必要なメッセージを受信するまでメソッドは起動できないので、オブジェクトへのメッセージの送信を停止すれば、そのオブジェクトは待ち状態になる。この同期機構は簡単である反面、複雑な同期を実現する場合には、同期用の余分な配管が必要になる。2章でのべたように、基本的にパイプはデータフローを実現するものであり、制御用のパイプが複雑に交錯することは望ましくない。そこで、ゲートを用いた新しい同期機構を導入する。入出力の一方しかないゲートをシグナルゲートと呼び、それ以外をメッセージゲートと呼ぶ。

ゲートは各パイプに1つ存在する。ゲートが1回発火すると1個のメッセージがゲートを通過する。メッセージがポートに溜まっていない場合は発火できない。

シグナルゲートには入力シグナルゲートと出力シグナルゲートの2種類がある。前者を1回発火すると1つのシグナルメッセージが生成される。シグナルメッセージはメソッド起動のタイミングを与えるための信

号である。一方、後者を1回発火するとオブジェクトからシグナルメッセージが1つ吸収される。メッセージがないとゲートは発火しない。

状態遷移規則は、ガード付きゲート発火規則であり、たとえば、`trans(s1, s2, G, g)` は、状態 `s1` において、ガード条件 `G` が満たされるならば、ゲート `g` を発火して、状態 `s2` に遷移するという規則である。

ガード条件としては、次の記述が可能である。

(1) オブジェクトの内部状態の照合およびその否定

例: `switchobj:state=on`
`~(switchobj:state=on)`

(2) ゲートの状態の照合およびその否定

`empty`: メッセージが空
`eos`: メッセージが全て終了
`full`: メッセージの受信側ブレースが満杯
`<メッセージ>`: 次にゲートを通過しうるメッセージの内容
`others`: その他

例: `g1=empty`
`g2=full`
`~(g3=10)`

(3) 1または2の記述を論理積で結合したもの(リストで表わす)

例: `[switchobj:state=off,g1=empty]`

(MENDEL ネットによる表現)

メッセージゲートは1入力1出力のトランジションと考えることができる。ゲートの入出力ポートは各オブジェクトの外部入出力ポートである。また、入力シグナルゲートは入力ブレースのないトランジション、出力シグナルゲートは出力ブレースのないトランジションである。MENDEL ネットではゲートを太棒で表現する。

□ 外部入出力ポート (ブレース)
 ■ ゲート (トランジション)

各オブジェクトは丸で囲み可読性を高める。各オブジェクトの中は省略して配管関係だけを表示してもよい(配管図)。

MENDEL では、ゲートの発火条件として次の制約があるといえる。

①受信外部入力ポートの安全性: ゲートに関しては、出力ブレースにトークンがない場合のみに発火可能と

する。これにより、外部入力ポートのトークンの数は高々1個であることが保証される。また、内部変数には高々1個のトークンしか存在しないことも明らかであり、複数トークンが存在しうるのは外部出力ポートのみとなる。

②発火の優先順位: ゲートよりメソッドの発火を優先する。これは、全てのオブジェクトが待ち状態になるまで、次のゲートを開かないことを意味する。

複合オブジェクトとその配管図の例を図4に示す。

```
object 伝票処理 : {
  dec: {
    inport(入力伝票);
    outport(出力伝票);
    mgate(g1, g2, g3);
    sgate(s1, s2);
  };
  body: {
    innode(入力伝票?lg1);
    outnode(出力伝票?lg3);
    処理1(開始!s1, 伝票a!lg1, 伝票b?lg2);
    処理2(伝票b!lg1, 伝票c?lg3終了!s2);
  };
  sync: {
    trans(s(0), s(1), [], s1),
    trans(s(1), s(1), [], lg1, g2, g3),
    trans(s(1), s(0), lg1=empty, s2),
    s(0)
  }
},
```

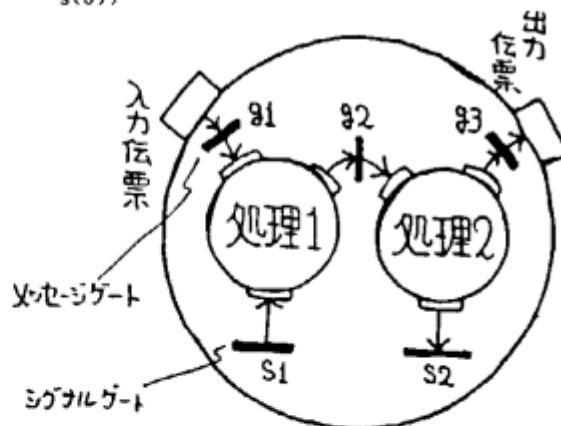


図4 複合オブジェクトの例

3.4 実行形態

任意のオブジェクトは実行可能な単位である。オブジェクトはMENDELインタプリタから入出力先の設定を付加したexecute文により実行される。オブジェクトを実行すると初期メッセージが入力ノードに送信され、出力ノードからゴールメッセージが受信される。MUEはMENDEL USER EXECUTIVEの意味のプロンプ

トである。

```
MUE?- execute(< オブジェクト名>,[
    <ポート名> ! <入力源>,...
    <ポート名> ? <出力先>,...
    ]).
```

[例]

```
MUE?-execute(keycount,
    [stream|file('stream.dat'),
    keyword|list([man,dog]),
    summary|list(X)],write(X),nl..
```

4. 同期部仕様記述言語

MENDELの複合オブジェクトの同期部の仕様を記述する言語TSL (Temporal Specification Language) を定義する。

定義

TSLは、線形時制命題論理 (PTL: Propositional Temporal Logic) に基づいており、PTLのマクロ言語である [川田87]。

(構文)

TSL式は、対象の有限集合OBJと対象の取りうる状態の全体集合Sから帰納的に定義される。

● $obj \in OBJ$, $s \in DC S$ のとき, $obj(s, D)$ は式である。ここで、Dを対象objのドメインと呼ぶ。

● f_1, f_2 が式のとき, $\sim f_1, f_1 \wedge f_2, f_1 \vee f_2, f_1 \supset f_2, \Box f_1, \Diamond f_1, \bigcirc f_1, f_1 \bigcup f_2$ は式である。

(意味)

$obj(s, D)$ は対象objの状態が $s \in D$ であるという命題である。ドメインDはobjの取りうる状態の集合である。 $\sim$ (否定), $\wedge$ (論理積), $\vee$ (論理和), $\supset$ (含意), $\Box$ (常に), $\Diamond$ (いつかは), $\bigcirc$ (次は), $\bigcup$ ($\sim$ までは $\sim$) などのオペレータについてはPTLと同じ意味を持つ。

TSL式の解釈とは、TSL式をPTL論理式とみなし、次の単一状態条件を付加して、PTL上で解釈することである。単一状態条件とは、各対象は各時刻にドメインの中のある1つの状態をとるということであり、単一イベント条件 [NW84] の拡張である。

(単一状態条件)

$$(\bigvee_{s \in D} obj(s, D) \wedge \sim (\bigvee_{\substack{s_1, s_2 \in D \\ s_1 \neq s_2}} (obj(s_1, D) \wedge obj(s_2, D))))$$

同期部の記述

TSLによる同期部の記述において、対象としては、次の3つがある。

対象名	状態
fire	発火するゲート名
<ゲート名>	ゲートの状態
<原子オブジェクト>: <状態変数>	状態変数の値

ここで、3章の同期部のプログラムとの対応を容易にするために、次の表記法の変換をおこなう。

```
fire(g1)      --->  g1
g1(empty)     --->  g1=empty
obj:flag(on)   --->  obj:flag=on
```

例

『ゲートがfullの場合以外は、ゲートg1とゲートg2を交互に発火させる』という仕様をTSLで記述すると次のようになる。

$$\Box (g1 \supset \bigcirc (\sim (g2 = full) \supset g2)) \wedge \Box (g2 \supset \bigcirc (\sim (g1 = full) \supset g1))$$

5. 同期部の自動生成

TSLによって記述された同期仕様から同期部を自動生成できる。この生成手法は、時制命題論理の定理証明手法の1つであるタブロー法をベースとしている [NW84, Plaisted86]。詳細は別の機会に報告する [内平88]。

6. プログラム例：2人の哲学者の食事

2人の哲学者の食事の問題をMENDELで記述する (付録)。このプログラムにおいて、トップレベルの複合オブジェクトphmainは4つの原子オブジェクト (哲学者2人、フォーク2本) を配管したものである。哲学者とフォークの間にはパイプが設定されており、このパイプを通して、フォークを掴む (s: seize)、あるいはフォーク放す (r: release) といった通信が行なわれる。ここではまだ同期部は記述されていない。このプログラムからMENDELネット (図5) が抽出できる。

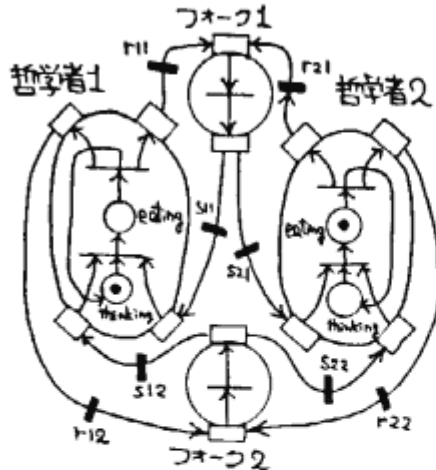


図5『2人の哲学者の食事』のMENDELネット
次に、同期部の仕様をTSLで記述する。
 $\square (r11 \Leftrightarrow \circ r12) \wedge \square (r11 \Leftrightarrow \circ r12) \wedge$
 $\square \diamond (s11 \vee s12) \wedge \square \diamond (s21 \vee s22)$
 このTSL式の中で、 s_{ij} は哲学者*i*がフォーク*j*を
 掴むこと、 r_{ij} は哲学者*i*がフォーク*j*を放すことを
 意味している。このTSL記述から phmain の同
 期部が生成できる(図6)。

```
sync: { ( [ trans(n(1).n(2).[],r11),
          trans(n(2).n(3).[],r12),
          trans(n(3).n(4).[],s22),
          trans(n(3).n(5).[],s21),
          trans(n(3).n(6).[],s12),
          trans(n(3).n(7).[],s11),
          trans(n(4).n(8).[],s21),
          trans(n(5).n(8).[],s22),
          trans(n(6).n(10).[],s11),
          trans(n(7).n(10).[],s12),
          trans(n(8).n(11).[],r22),
          trans(n(10).n(2).[],r11),
          trans(n(11).n(3).[],r22)).n(1)) }
```

図6 MENDELプログラムの同期部

このプログラムをMENDELインタプリタで実行
 する場合は、入出力がないので次ように指定する。

```
MUE?-execute(phmain.[]).
```

7. まとめ

ベトリネットをベースとし、時制命題論理で記述可
 能な同期機構を持つ並列プログラム言語MENDEL
 /88を提案した。今後は、ベトリネットを基礎にし
 たMENDELの意味論を明確にしていきたい。また、

リアルタイムSA手法との融合により、要求定義段階
 (リアルタイムSA)からプログラム言語(MENDEL/88)
 までの一環したプログラミング支援環境
 も興味深いテーマである。

謝辞

本研究の一部は、ICOTの再委託研究の一環とし
 て行なわれた。ICOTの関係各位に感謝する。

参考文献

- [伊藤88] 伊藤, 内平, MENDELにおける意味ネットを
 用いた部品結合, 第2回人工知能学会大会4-4, 1988.
- [内平86] 内平 他, MENDELにおける並列プログラムの
 部品合成, 情報処理学会ソフトウェア工学研究会46
 -8, 1986.
- [内平87] N.Uchihira, etc., Concurrent Program Syn-
 thesis With Reusable Components Using Temporal L-
 ogic, COMPSAC87, 1987.
- [内平88] 内平 他, 時制論理からの同期部の自動生
 成, 第37回情報処理学会全国大会(発表予定), 1988
- [川田87] 川田 他, 時制論理ベースの形式的仕様記
 述言語PTS, ソフトウェア科学会第4回大会, 1987.
- [久世86] 久世 他, Stellaにおけるモジュールの部品
 化・再利用システム, 第32回情報処理学会全国大会,
 p705-705, 1986.
- [小宮88] 小宮, 部品合成による自動プログラミング
 ・システムの实现方法について, 情報処理学会ソフト
 ウェア基礎論研究会24-5, 1988.
- [柴山88] 柴山, 米沢, 並列オブジェクト指向言語AB
 CL/1.bit Vol.20, No.7, 共立出版, 1987.
- [本位田88] 本位田, 内平 他, 推論型システム記述
 言語MENDEL, 情報処理学会論文誌Vol.27, No.2, 1986.
- [Barstov88] D.Barstov, Automatic Programming Fo-
 r Streams II: Transformational Implementation, 10th
 ICSE, 1987.
- [MW84] Z.Manna and P.Volper, Synthesis of comm-
 unicating processes from temporal logic specific-
 ation, ACM TOPLAS, 1984.
- [Plaisted86] D.A.Plaisted, A decision procedure
 for combinations of propositional temporal logic
 and other specialized theories, J. Automated Rea-
 soning 2, 1986.

<<< 付録 >>>

※— 二人の哲学者の食事のMENDELプログラム —

※— メイン (複合オブジェクト) —

```
object phmain :{
  dec:{
    mgate(s11, s12, s21, s22, r11, r12, r21, r22)
  } ;
  body:{
    philosopher1(outrfork?[r11], outlfork?[r12],
      inrfork![s11], inlfork![s12] ) ;
    philosopher2(outrfork?[r21], outlfork?[r22],
      inrfork![s21], inlfork![s22] ) ;
    fork(answer?[s11, s21], question![r11, r21]) ;
    fork(answer?[s12, s22], question![r12, r22])
  } ;
  sync:{
    *** 同期部は後に生成される ***
  }
}.
```

※— 哲学者1 (原子オブジェクト) —

```
atomic object philosopher1 :{
  dec:{
    inport(inrfork, inlfork) ;
    outport(outrfork, outlfork) ;
    state(s:[thinking, eating]!thinking)
  } ;
  meth:{
    method(s?thinking, inrfork?ok, inlfork?ok, sleating) ;
    method(s?eating, outrfork!free, outlfork!free, slthinking)
  }
}.
```

※— 哲学者2 (原子オブジェクト) —

```
atomic object philosopher2 :{
  dec:{
    inport(inrfork, inlfork) ;
    outport(outrfork, outlfork) ;
    state(s:[thinking, eating]!eating)
  } ;
  meth:{
    method(s?thinking, inrfork?ok, inlfork?ok, sleating) ;
    method(s?eating, outrfork!free, outlfork!free, slthinking)
  }
}.
```

※— フォーク (原子オブジェクト) —

```
atomic object fork :{
  dec:{
    inport(question) ;
    outport(answer)
  } ;
  meth:{
    method(question?free, answer!ok)
  }
}.
```