人　工　知　能　106－1
（1996．10．30）

複数知的主体の相互作用の時間発展に関する理論モデル記述言語とその実装

石田 和成

電気通信大学 大学院 情報システム学研究科

概要: 本論文は、複数知的主体の相互作用の時間発展モデルの記述言語とその実装について報告するものである。このモデル記述言語は、モデル構築者が必要とする基本的なモデル動作方法として、組合せパターン処理による代替案生成、選択のメカニズムを提供する。また、この組合せパターン処理は、通常の逐次処理言語で生じる不必要なデータ間の逐次的依存性を排除し、データ間の並行性を提供する。さらに、この組合せパターン処理によって、シミュレーションモデルにおいて致命的な欠陥となるモデルを表現するプログラムの意味的な誤りの検出が可能であることを、スケジューリングの例を用いて示す。このような特徴を持つ本モデル記述言語は、マルチエージェントなどの複数知的主体のモデルを構築、検証するための有効なツールとなると考えられる。

A Modeling Language for Simulating Collective Behavior among Intelligent Performers

Kazunari Ishida

Graduate School of Information Systems,
University of Electro-Communications

Abstract: This paper is concerned with a modeling language for simulating collective behavior among intelligent performers. The language provides a mechanism of alternative creation and selection with combinatorial pattern processing for model builders who explore complexity among intelligent performers. The mechanism eliminates unnecessary sequential data dependency and provides data concurrency among the intelligent performers.

Hence, a model builder with the mechanism is able to detect a semantic error on model program. The error should be detected and be eliminated by the model builder, because the error makes a result of simulation meaningless.

This paper also contains a part of scheduling program on the language to show that the language is an effective tool for developing and testing a model of collective behavior among intelligent performers such as multi-agent model.

1 はじめに

本論文は、複数知的主体の相互作用の時間発展モデルの記述言語とその実装について報告するものである。このモデル記述言語は、モデル構築者が必要とする基本的なモデル動作方法として、組合せパターン処理による代替案生成、選択のメカニズムを提供する。また、この組合せパターン処理は、通常の逐次処理言語で生じる不必要なデータ間の逐次的依存性を排除し、データ間の並行性を提供する。さらに、この組合せパターン処理によって、シミュレーションモデルにおいて致命的な欠陥となるモデルを表現するプログラムの意味的な誤りの検出が可能であることを、スケジューリングの例を用いて示す。このような特徴を持つ本モデル記述言語は、マルチエージェントなどの複数知的主体のモデルを構築、検証するための有効なツールとなると考えられる。

これまでのモデル記述言語は、Lisp や Prolog などで用いられる逐次的なメカニズムを処理系の動作原理として採用し、その上に複雑なモデル記述機構を用意している [1] [2] [3] [4] [5] [6] 。

しかし、これらの言語は逐次的処理系を採用しているために、複数主体の相互作用をモデル化するのに適していない。さらに、モデルの複雑な記述機構が、モデル構築者の負担を増大させている。

それに対し本モデル記述言語では、複数主体モデルの動作において本質的な組合せパターン処理をモデル構築者に提供し、それ以外の不必要な機能の提供を極力抑えることによって、モデル構築者が言語に関して覚えるべき事柄を削減し、理論モデル記述に集中することを可能にしている。

以降、第2節では、モデル記述言語 TD-NCPP を説明する。第3節では、モデル世界を、第4節では、モデル世界の構成要素であるクラスをそれぞれ説明する。第5節では、処理系の実装について述べ、第6節では、プログラム例を示す。

2 時刻駆動型ネスト構造化組合せパターン処理記述言語

時刻駆動型ネスト構造化組合せパターン処理記述(Time Driven Nested Combinatorial Pattern Processing,TD-NCPP) 言語は、複数知的主体の相互作用を記述を目的としたモデル記述言語である。このモデル記述言語は、組合せ処理によるパターンの生成・選択を基本的なモデル動作方法として提供することによって、従来の逐次言語でプログラミングに課せられていた不必要なデータ間の逐次的依存性を排除し、データ間の並行性を提供する。ここで、時刻駆動 (TD) とは、モデル構築者が指定したタイミングでネストを許した組合せパターン処理 (NCPP) を駆動する機構である。

TD-NCPP では、データ構造の表現方法として、Prolog などで用いられている述語形式を用いる。この述語によるデータ構造表現は、構造体やレコードのような柔軟なデータ表現を可能にする。

また、プログラムの制御方法としては、マクロレベルではTD, ミクロレベルではNCPP を用いる。つまり、理論モデルの記述には、NCPP によって複数主体の相互作用や意思決定のメカニズムを記述し、TD によってそのメカニズムの動作する順序関係を記述する。

2節の構成を以下に説明する。2.1 では、モデルのマクロなアルゴリズムを記述するための時間駆動 (TD) について、2.2 では、モデルのミクロなアルゴリズムを記述するためのネスト構造化組合せパターン処理 (NCPP) について、2.3 では、モデルのデータ構造を格納するアクセス権管理機構付き黒板について、2.4 では、モデルのデータ構造を表現する述語について、2.5 では、モデルのデータに関する基本的な操作を提供する関数について、2.6 では、パターン生成のメカニズムを提供する変数についてそれぞれ説明する。

2.1 時刻駆動

モデル世界における大まかな処理の流れは、時刻駆動 (TD) によって記述される。これは、タイムラベルの順序リストによって表現される。これらのタイムラベルは、NCPP の起動条件として働くものである。タイムラベルリストは、それを構成するタイムラベルの挿入や削除、順序の入れ替えなどを柔軟に行なうことができる。また、各タイムラベルには、条件分岐が設定可能である。分岐先はタイムラベルで指定し、分岐条件は組合せパターン処理 (CPP) で記述する。

それに対して、細かなアルゴリズム記述は、次節で説明する NCPP の部分で記述する。このようなマクロレベルとミクロレベルのアルゴリズム記述方法は、モデル記述者の柔軟なモデル記述を可能とする。

```
\enter{TIMELIST}
    タイムラベル1 { 分岐条件1 } 分岐先1
                  :                :
                { 分岐条件n } 分岐先n
        :
\exit{TIMELIST}
```

分岐条件は分岐条件1から評価され、その条件が充足されれば、残りの分岐条件は評価せず、分岐先のタイムラベルに処理が移る。

2.2 ネスト構造化組合せパターン処理

ネスト構造化組合せパターン処理(NCPP)は本モデル記述言語のミクロレベルのプログラム制御方法を提供する。複数要素間の相互作用の処理において本質的である組合せパターン処理は、モデル構築者が考慮したい要素間の相互作用を述語の組合せで記述する方法を提供する。

このNCPPテンプレートをモデル世界に適用することによって、可能な限りのパターンが生成される。このパターンの生成がモデルの動作原理である。

NCPPでは操作部内で、ネストした条件部の記述を許すことによって、詳細なアルゴリズムを記述を可能をした。

以下では、NCPPの記述に関する基本的構造を説明する。

トップレベルテンプレート

トップレベルテンプレートはTDによって起動される。トップレベルテンプレートの基本的構造を示す。

```
COND {TIME=タイムラベル}   {ループ制御部}
    条件部
OPER {パターン選択制御部}
    操作部
```

ここで、タイムラベル、ループ制御部、そしてパターン選択制御部は必要に応じて指定する。トップレベルテンプレートのタイムラベルを指定しない場合には、TDが示すいかなるタイムラベルにおいても、そのテンプレートは黒板に適用される。また本言語では、このテンプレートによって生成された複数のパターンの操作部(RHS)の実行結果の間には並行性が保たれる。この処理によって、複数主体の個々の動作が1つのテンプレートで記述できる。

条件部

条件部は0個以上の述語、真偽関数、副作用をもつ関数、そしてネスト条件部によって構成される。

NCPPの条件部に記述できる関数は、真偽関数と、副作用有りの関数である。副作用有りの関数を条件部に記述する場合、関数の副作用の効果を利用することを目的としているので、副作用関数の戻り値は常に真として、条件判断に影響を及ぼさないようにしている。

ネスト条件部

ネスト条件部は0個以上の条件部、ネスト条件部によって構成される。

ネスト条件部においては、通常の条件部では記述できない条件のOR接続が可能である。

操作部

操作部は0個以上の述語、副作用を持つ関数、ネストテンプレートによって構成される。本言語では、このテンプレートのネストによって、詳細なアルゴリズムの記述を可能にしている。

ネストテンプレート

ネストテンプレートの基本的構造を示す。

```
[   COND {ループ制御部}
        条件部
    OPER {パターン選択制御部}
        操作部
]
```

トップレベルテンプレートとは異なり、タイムラベルの記述はない。それ以外の記述についてはトップレベルテンプレートと同じである。

ループ制御部

ループ制御は、テンプレートを条件を変えながら適用する方法を提供する。そのような意味で、ループ制御は、ミドルレベルのアルゴリズムの記述方法を提供するものと考えることができる。

現在のモデル記述言語では、ループ制御の方法として、WHILE制御とFOREACH制御の2つが利用可能であるが、紙面の都合上WHILE制御のみ説明する。

WHILE制御

WHILE制御は、C言語などのwhile文と同様の機能を持つ。WHILE制御の書式を以下に示す。

```
{ WHILE,TARGET(ループ変数,初期数値設定部),
  COND(真偽関数),ACTION(副作用有り関数) }
```

ループの終了はCONDの真偽関数が真を返した場合である。

パターン選択制御部

パターン選択制御部では、テンプレートから生成されたパターンを選択する方法を柔軟に設定できる。また、パターン選択において、パターンのサブグループ化という概念を導入することによって、サブグループ毎のパターン選択を可能とした。この機能を利用すると、1つのテンプレートで、各知的主体が個々に意思決定を行なうメカニズムを容易に記述できる。

{PIV<基点条件位置>,SELECT<パターン選択数>,
　選択用評価値,選択方法}

詳細については、「6 プログラム例」において説明する。

2.3 アクセス権管理機構付き黒板

複数要素の相互作用に関する理論モデルを記述する場合、複数主体間の情報伝達の方法の1つとして、各主体の間の2者間のメッセージパッシングが考えられる。しかし、TD-NCPPにおいては、全主体の間で同期のとれた相互作用を記述することを目的としているので、ブロードキャストの方が適している。

しかし、モデル全域的なブロードキャストのみでは、処理系の実行効率の低下が生じ、モデルを構成するクラスの間の予期せぬ相互作用によるバグの発生の可能性が高くなる。

そこで、TD-NCPPでは、他のクラスからの黒板の情報を読み書きの権利を制限し、モデル全体で一様にアクセスできるデータ空間を細かな部分データ空間に区切ることによって効率低下を防ぎ、バグ発生の可能性を削減する。このようなアクセス権の設定はオブジェクト指向言語における情報隠蔽の機能を果たす。アクセス権の設定方法の詳細は省略する。

\enter{BLACKBOARD}[黒板識別子]{アクセス権}
　述語宣言
\exit{BLACKBOARD}

TD-NCPPにおけるデータ構造である述語は、黒板上で宣言する。述語宣言によって、黒板の述語管理データベースにその述語の構造が記録される。データベースに登録された述語は、いかなる場所でインスタンスが生成されても、所属する黒板で一元的に管理できる。

述語宣言においては、述語識別子、そして述語を構成する属性識別子、そして属性デフォルト値を指定する。

\def 述語記述子(< 属性記述子 > 属性デフォルト値,...)

NCPPテンプレートにおいて、述語のフィールドを参照する場合、属性を指定して記述すれば、他の参照する必要のないフィールドに関する記述を省くことができる。また、全てそのような記述をしておくならば、述語を構成するフィールドを拡張する場合には、NCPPテンプレートの修正は必要ない。

2.4 述語

TD-NCPPにおいてデータは述語インスタンスとして生成され、アクセス権制御機能付き黒板にストアされる。ここで記述子とは、数字や記号から始まらず、空白を含まない文字列のことをいう。述語は述語識別子と丸括弧で囲まれたアトム列から構成される。アトムは整数、実数、記号、リストから構成される。

整数：使用可能範囲はOSによって定まる。
実数：使用可範囲や精度はOSによって定まる。
記号：数字や記号から始まらない間に空白を含まない文字列。
リスト = [アトム列]
アトム = 整数 | 実数 | 記号 | リスト
アトム列：カンマで区切られたアトムのならび。
述語 = 述語識別子(アトム列)
属性識別子：記述子を'<','>'で囲んだもの。
デフォルト値付き属性識別子：属性識別子の後ろにアトムを付加したもの。
属整列：カンマで区切られた属性識別子やデフォルト値付き属性識別子のならび。

述語宣言 = **\def** 述語識別子(属整列)

2.5 関数

TD-NCPPにおける関数は、副作用無しの関数と、副作用有りの関数がある。

副作用無しの関数には、真偽関数、整数関数、実数関数、リスト関数、アトム関数がある。真偽関数は評価の結果が真の場合は1、偽の場合は0を返す。整数関数は評価の結果を整数で返す。実数関数は評価の結果を実数で返す。記号関数は評価の結果を記号で返す。リスト関数は評価の結果をリストで返す。評価不可能の場合、整数関数、実数関数、記号関数、リスト関数はそれぞれ、0、0.0、NULL文字、NULLリストを返す。アトム関数は引数の種類に応じて、評価の結果としてを整数、実数、記号、リストのいずれかを返す。アトム関数が評価不可能の場合にはNULLリストを返す。

現在実装している全ての関数のリストは紙面の都合上省略する。

2.6 変数

TD-NCPP において変数は複数パターン生成メカニズムを提供する。変数は、テンプレート、モデルの終了条件、そしてモデルの一時停止条件の条件部分において、述語の属性部分や、関数の引数として利用される。

通常のプログラミング言語における変数が、宣言、代入を行なうことを基礎としているのと対照的に、TD-NCPP における変数は、パターンマッチによって処理系が変数に要素をバインドすることを基礎としている。この変数には述語の属性部分に記述されるいかなる要素もバインド可能である。

また変数のスコープに関して、LISP やC 言語などでは、既に上位レベルで宣言されている変数と同じ名前の変数がローカルレベルで宣言されると、同じ名前の上位変数は無効となり、ローカル変数のスコープが有効になる。しかし、TD-NCPP では、パターンマッチを動作原理としており、パターン生成の制約機構として変数を用いているので、下位レベルの変数が上位レベルの変数と同一の名前の場合には、下位レベルの変数は上位レベルの変数と同一のものとしている。

3 モデル世界

TD-NCPP において理論モデルは、モデル世界とそれを構成するクラス群によって構成される。これらクラス群はTD によってモデル全体で同期的に動作する。また、データの格納方法としては、アクセス権管理機構付き黒板を用いる。モデル世界は0個以上の黒板を持つことができる。この黒板は、アクセス権管理機構を持たない。この黒板では通常、モデルを構成するクラス群全体で参照可能としたい述語の宣言やインスタンスを記述しておく。

本節では、モデル世界の記述に関して、(1) クラスファイルリスト、(2) ワールドタイムリスト、(3) モデルの一時停止条件を説明する。それ以外のモデル世界の記述に関しては紙面の都合上省略する。

3.1 クラスファイルリスト

モデルを構成する主体は、「4 クラス」で説明する方法で、種類毎に1つのクラスファイルに記述する。世界モデルを構成するとき、必要となるクラスファイルを以下のように指定する。

```
\enter{CLASSFILES}
    クラスファイル名
\exit{CLASSFILES}
```

3.2 ワールドタイムリスト

ワールドタイムリストは、モデル世界全体のマクロなアルゴリズムの流れを記述するものである。この基本的な動作は、「2.1 時刻駆動」で説明した通りである。

```
\enter{GTLIST}
    タイムラベル1 { 分岐条件1 } 分岐先1
                      :              :
                  { 分岐条件n } 分岐先n
          :
```

\exit{GTLIST} ワールドタイムリストにおいて記述可能なタイムラベルは、「4.1 ローカルタイムリスト」において説明するモデル世界に属するクラスのローカルタイムリストラベルや、それを構成する個々のローカルタイムラベルである。このTD によってプログラム制御の階層的構造化が可能となる。

3.3 モデルの一時停止条件

モデルの一時停止条件は述語によるパターンによって表現する。モデルの一時停止条件が満たされると、処理系の動作は、通常、シミュレーションを行なう連続実行モードから、モデルのデバッグを行なうステップ実行モードになる。この一時停止条件は、モデルの意味的デバッグに用いることができる。どのようなプログラミング言語でも、構文の誤りは検出するけれども、ほとんどのプログラミング言語は、意味の誤りについて検出する簡単な方法は備えていない。

TD-NCPP では、理論モデルにおいて、モデルのプログラムに誤りが無ければ生成されるはずのない状態を、述語のパターンとして記述することによって、モデルを表現するプログラムの意味的デバッグが可能となる。

```
\enter{BPATTERNS}
    モデルの一時停止条件
\exit{BPATTERNS}
```

詳細なモデルのバグレポートをファイルなどに残したいならば、デバッグ用のクラスを記述する。そのデバッグクラスではモデルの意味的な誤りを検出したときに、その内容をファイルに記録し、それと同時にモデルの一時停止条件を発行するようにしておく。この意味的デバッグの例は、「6 プログラム例」で示す。

モデルを表現するプログラムの意味的誤りは、シミュレーション結果を無意味なものにしてしまうので完全に排除されなければならない。本モデル記述言語では、通常のプログラミング言語において困難なこの意味的誤りの検出が非常に簡単に行なうことができる。

4 クラス

本モデル記述言語では、モデル世界に属するクラス群はTDによってモデル全体で同期的に動作する。また、データの格納方法としては、アクセス権管理機構付き黒板を用いる。個々のクラスはそれぞれ0個以上のアクセス管理機構付き黒板を持つことができる。

本節では、クラスの記述について、(1)ローカルタイムリスト、(2)標準書き込み黒板、(3)NCPPテンプレートを説明する。それ以外のクラスの記述については紙面の都合上省略する。

4.1 ローカルタイムリスト

ローカルタイムリストは、クラス内のTD-NCPPの実行の順序を考慮して、それらの実行を指定するタイムラベルをリストにしたものである。

```
\enter{LTLIST} [ローカルタイムリストラベル]
    ローカルタイムラベル1 { 分岐条件1 } 分岐先1
                :                :          :
                                 { 分岐条件n } 分岐先n
                :
\exit{LTLIST}
```

この記述方法において、ワールドタイムリストと異なる点は、ローカルタイムリスト全体を表現するものとしてローカルタイムリストラベルを記述することである。

ローカルタイムリストや分岐先に記述できるタイムラベルは、そのリストを持つクラスのタイムラベルか、そのクラスのベースクラスやモジュールのタイムラベルのみである。

このようなタイムラベルの参照関係に従って、ワールドタイムリストの中に、このローカルタイムリストラベルを貼り付けることによって、プログラム制御の階層的構造化が可能となる。

4.2 標準書き込み黒板

モデルにおけるどの黒板にも登録されていない述語のインスタンスを生成する場合に、デフォルトで書き込む先の黒板を指定する。

```
\enter{DEFAULTBB}
    標準書き込み先黒板識別子
\exit{DEFAULTBB}
```

4.3 TD-NCPP テンプレート

TD-NCPPに基づいてモデルの動作を記述する部分である。TDの詳細については、2.1節で、NCPPの詳細については、2.2節で説明した。

```
\enter{NCPPTEMPLATE}
    TD-NCPPテンプレート
\exit{NCPPTEMPLATE}
```

5 処理系の実装

処理系の実装は、C言語を用いて行なった。実装に用いたOSは、PC/AT互換機上のLinux、Sun Workstation上のSolaris1.x、Solaris2.5である。また、複数のクラスや、それらに属する複数の黒板の状態を同時に表示するために、X-WindowsのAthena Widgetを使用した。

6 プログラム例

知的主体の行動の例として、スケジューリングのプログラムの主要な部分を以下に示す。作成されるスケジュールのデータは、黒板に以下のようなデータ構造で格納される。

```
\begin{BLACKBOARD}[ScheduleBoard]
  \def SCHEDULE(<Man>,<Machine>,<JobIdx>,<Start>,<End>)
\end{BLAKBOARD}
```

ここで<Man>は担当者名、<Machine>は使用機械、<JobIdx>はジョブ番号、<Start>はジョブ開始時刻、そして<End>はジョブ終了時刻である。

スケジューリング処理においては、処理すべきジョブに関する全ての順列を作成して、それぞれのジョブの順列について、ガントチャートを作成し、

最も処理時間が短いものを本スケジュールとして採用する。

ここでは、各ジョブ順列リストに関するガントチャートを作成する場合に、すでにスケジュール済みのジョブとの無矛盾性のチェックを行なう条件部を示す。ガントチャート処理を施す順列リストは、$pb によって、そのリスト内の各ジョブは、$num によって制御される。これらの変数は、上位レベルでループ制御されていることとする。そして、$tm、$localtime、$nexttime、そして$basetime には、スケジュール開始時刻がバインドされていることとする。$localtime はスケジュール実行可能時刻探索ループ制御用変数、$nexttime はスケジュール実行可能候補時刻バインド用変数、$basetime はスケジュール実行可能時刻バインド用変数である。

```
IF {WHILE,TARGET($localtime,$tm)),
          COND(#noteq($nexttime,FINISH)),
          ACTION(@set($localtime,$nexttime))}
^[ // 現在注目するジョブについて整合性を調べる
 JOBTEMPLATE(<Number>$num,<Machine>$mc,
             <Duration>$nt,<Man>$who)
  [ // 前段階までの仮スケジュールと整合性チェック
  VIRSCHEDULE(<Machine>$mc,<Man>$ag,<PermNum>$pb,
              <Start>$sta,<End>$end)
   [  #range($sta,@add($localtime,$end)
   || #range($sta,
             @add($st,@sub($nt,1.0,$localtime),
             $end)
   ]
  @set($nexttime,@add(1.0,$end))
  ]
  ||
  [ // 本スケジュールと整合性チェック
  SCHEDULE(<Machine>$mc,<Start>$sta,<End>$end)
   [  #range($sta,@add($st,$localtime),$end)
   || #range($sta,
             @add($st,@sub($nt,1.0),$localtime),
             $end)
   ]
  @set($nexttime,@add(1.0,$end))
  ]
 ]
THEN
 !@set($nexttime,FINISH)
 !@set($basetime,$localtime)
```

このテンプレートでは、ジョブ順列リストを構成する各ジョブについて、$localtime がスケジュール実行可能時刻となるまでループ制御が適用されることによって、スケジュール実行可能時刻が$basetime に検出される。

ここで #range は第2引数の値が第1引数の値以上、第3引数の値以下の場合に真を返す真偽関数である。また、@set は第1引数の変数に第2引数の値を代入する副作用のある関数である。JOBTEMPLATE は、ジョブの情報が格納された述語である。VIRSCHEDULE はスケジューリング処理中に作成されたスケジュール案の情報が格納された述語である。関数 @set の前の記号'!'は、関数の逐次的な処理を指定するものである。この指定によって、ループの終了条件が機能する。

次に、作成されたそれぞれの仮スケジュールの必要とする処理時間を集計するプログラムを示す。

```
IF
  PERMDAT(<Name>$ag,<PermNum>$pb,<Finish>$ftime)
  VIRTUAL(<Machine>$mc,<Man>$ag,
          <PermNum>$pb,<End>$end)
THEN {PIV<1>,SELECT<1>,$end,BIG}
  @set($ftime,$end)
```

ここでPERMDAT はジョブ順列のデータを保持する述語である。

パターン選択制御部における PIV<1>は、各ジョブ順列毎にスケジュール終了時刻を求めるために、PARMDAT を基点としたパターンのサブグループ化を行なうためのものである。SLELECT<1>によるパターン選択は、各サブグループ毎に行われる。

それらの各ジョブ順列にに関するスケジュール終了時刻は、その順列に属する仮スケジュールの終了時刻$end のうちの最も遅いものである。したがって、その選択方法としては、評価値$end、その評価値が大きいパターンを選択する選択方法 BIG をパターン選択制御部で指定する。

このパターン選択制御部によって、PARMDAT の <Finish>$ftime にそのスケジュール終了時刻が格納される。

次に、各主体が自分の仮スケジュールのなかで最も処理時間が短い仮スケジュールを選択するプログラムを示す。

```
IF
 AGENT(<Name>$ag)
 PARMDAT(<Owner>$ag,<Finish>$ftime)
THEN {PIV<1>,SELECT<1>,$ftime,SMALL}
 @set($ftime,Select)
```

ここでAGENT はジョブ処理のためのスケジューリングを行なう知的主体を表現する述語である。

パターン選択制御部における PIV<1>は、各知的主体が本スケジュールに関する意思決定を行なうため

に、AGENTを基点としたパターンのサブグループ化を行なうことによって、各エージェントの代替的選択肢をまとめるためのものである。SLELECT<1>によるパターン選択は、各サブグループ毎に行われるので、各エージェントのスケジュール選択に関する意思決定が可能となる。

スケジューリングにおいて、各エージェントは各自で作成した仮スケジュールの終了時刻$ftimeのうち、最も終了時間の速いものを本スケジュールとして選択する。したがって、その選択方法として、評価値$ftime、その評価値が小さいパターンを選択する方法SMALL をパターン選択制御部で指定する。

このパターン選択制御部の記述によって、PERMDATの<Finish>$ftime にスケジュールの採用フラグSelect がセットされる。

スケジューリングにおいて、プログラムの誤りによる実行不可能なスケジュールが生成されるならば、妥当なシミュレーションを行なうことができない。

これはシミュレーションモデルにおいては致命的なことであるが、通常のプログラミング言語では、このような状況を検出すること自体が困難である。しかし、本記述言語では、パターンマッチの機構によって処理系が動作するので、モデルにおいて存在が許されないパターンを記述しておけば、モデルを表現するプログラムの意味的な誤りが検出可能となる。

例えば、モデル上における意味的誤りとして、実際には実行できないにも関わらず、同一の機械を使用するスケジュールに時間的な重複がある場合を検出する場合には、以下のような条件部を持つテンプレートを記述すればよい。

```
IF
  SCHEDULE(<Machine>$mc1,<Start>$st1,<End>$ed1)
  SCHEDULE(<Machine>$mc2,<Start>$st2,<End>$ed2)
  #equal($mc1,$mc2)          // 同一の機械を使用
  [  #range($st1,$st2,$ed1) // 時間の重複あり
  || #range($st1,$ed2,$ed1)
  ]
THEN
  @write(_file(debug.log),
         $mc1,$ag1,$st1,$ed1,$ag2,$st2,$ed2)
  ScheduleError($mc,$ag1,$st1,$ed1,$st2,$ed2)
```

ここで#equal は第1引数と第2引数の値が等しい場合に真を返す真偽関数である。

このテンプレートは、モデルを表現するプログラムの意味的誤りによる、本来は意図していないスケジュールが生成された場合には、その状況をファイル debug.log と、述語 ScheduleError とに出力する。

モデル世界の一時停止条件において、以下のように ScheduleError を登録しておけば、シミュレーションの連続実行は中断され、ステップ実行モードに移行するので、原因の究明を行なうことができる。

```
\enter{BPATTERNS}
  { ScheduleError(*) }
\exit{BPATTERNS}
```

7 まとめ

複数知的主体の相互作用を、組み合わせ処理として捉え、モデル記述言語を考案した。また、その処理系を実装し、例題を示すことによって、モデル記述言語の記述能力を示した。

参考文献

[1] Fishwick, P. A. & R. B. Modjeski, : Knowledge-Based Simulation - Methodology and Application, Springer-Verlag, 1991.

[2] 木下 哲男, 菅原 研次, : エージェント指向コンピューティング, SRC, 1995.

[3] Y. Shoham, : Agent-oriented Programming, Artificial Intelligence, Vol. 60, pp. 51-92, 1993.

[4] Stelzner, M., J. Dynis, & F. Cummins, : The SIMKIT System : Knowledge-Based Simulation and Modeling Tools in KEE, Proceedings of the 1989 Winter Simulation Conference E.A. MacNair, K.J.Musselman,P.Heidelberger(eds.), pp. 232 - 234.

[5] 川村尚生, 斎藤善徳, 金田悠紀夫, : エージェント間の共有知識を実現したPrologに基づく協調処理言語, 情報処理学会 プログラミング-言語, 基礎, 実践-,pp. 59 - 65, 1992.

[6] Bosschere, K., P. Tarau, : Blackboard-based Extentions in Prolog, Software - Practice and Experience, Vol. 26(1),pp. 49 - 69, 1996.