奈良女子大学大学院人間文化研究科
修士論文

BDIロジックに基づく
エージェント実行系の構築

学籍番号 10840031
片山 寛子

2012年1月31日

1 はじめに

BDI モデル [Singh 99] では、合理的エージェントは、明示的に信念・願望・意図の 3 種類の心的状態を持ち、これら心的状態を保持・更新することで意思決定を行い、目的を達成するように振る舞う。BDI モデルの特徴の 1 つは、これら 3 種類の心的状態やその時間的変化を明示的に記述できる BDI ロジック [Rao 91] という様相論理体系を持ち、これによって合理的エージェントの心的状態や振る舞いに関する形式的な議論や証明が行えることが保証される点である。この点が合理的エージェントの設計上大きな利点と考えられ、BDI モデルが受け入れられる一要因となってきた。BDI モデルに基づいて実現されるエージェントは、BDI エージェントと呼ばれる。

しかし、BDI モデルに基づいて多くのシステムが開発されたり研究が行われたりしてきた一方で、それら実際のシステムと BDI ロジックによる表現の間にギャップがあることはかねてから指摘されてきた [Fisher 97, Móra 99]。それは主に、実際のエージェントシステムは手続き的な記述によって実現され、その意味論は操作的に定義されることが、BDI ロジックの公理的意味論とのずれを招くことによるとされている。せっかくエージェントの振る舞いに関する形式的な議論を行える手段が用意されていても、それを生かしにくいことになるのである。

例えば、BDI エージェントの実装プラットフォームとして知られる Jason[Bordini 07] は、プランの前提条件の部分は Prolog ライクに書けるが、プラン本体の実行は、そこに書かれている基本行為や副目標を順に達成していくという手続き的な記述であり、また信念など心的状態の更新も、それらを加除するという手続きによる。

このようなギャップを解消するには、論理式による記述をそのまま実行可能なプログラムとして扱う手法が考えられる。そのような提案もいくらかあるが [Fisher 97, Móra 99, Dastani 03]、意味論が複雑であったり論理外の要素で動作が既定される部分があったりするため、論理的な記述とエージェントの動作を直接対応させて理解したいという要求からは不満が残る。

そこで本論文では、時相論理による記述を直接実行可能なプログラムとする Temporal Prolog[桜川 87] と同じ手法を用いて、BDI ロジックによってエージェントの性質を記述し、それを直接実行する形態のエージェント実装システムを実現するアイデアについて述べる。これにより、記述されたエージェントの性質は論理で記述されたものと同じになることが期待でき、形式的議論の有用性が上がると考えられる。

2 関連研究

論理による記述を用いるエージェントシステムの提案としては、本研究の他には Móra らの X-BDI (eXecutable BDI)[Móra 99]、Concurrent METATEM[Fisher 93] と BDI モデルのハイブリッドというアプローチ [Fisher 97]、Dastani らの 3APL[Dastani 03] などがある。

X-BDI は、Event Calculus[Shanahan 99](の拡張) を基にし、イベントがある時刻にある性質 P を「成り立たせ始める」・「成り立たなくする」などを表す述語 *initiates*, *terminates* などを用意し、それらを用いて心的状態の変化を表現する。これだと通常の論理プログラムと異なり、何かを成り立たせることと成り立たせなくすることとが両方導かれて衝突することが起きうるので、その対処として非単調論理の導入や、否定の概念を複数通り導入することなどを要する。このことと、イベントの成立時刻の記述に絶対時刻を用いること、心的状態に優先度の記述を設けることなどから、論理式の記述も意味論もかなり複雑なものとなる。これらが複雑だと、形式的議論に論理体系を用いることが難しくなる難点がある。

例えば、合理的エージェントの重要な性質の 1 つとされる「コミットメント戦略」[Rao 91] のうちの 1 つ、open-minded コミットメント戦略 (“エージェントが ϕ の達成を意図すれば、ϕ の達成を信じるかあるいは達成の可能性を信じなくなるまで、その意図を保持する”) は、本来の BDI ロジックでは $INT\phi \rightarrow INT\phi$ $until$ $(BEL\phi \vee \neg BEL \Diamond\phi)$ のように書けるが、X-BDI では絶対時刻を表す変数を引数に加える必要などがあるため、かなり煩雑な式となってしまう。我々の提案は、論理体系そのものは本来の BDI ロジックのままであるため、上の式を (プログラムとしては直接書けないものの) そのまま用いて、意味論的議論によってあるエージェントがこの性質を満たしているかどうかを議論することができる。

Fisher のものは、Concurrent MetateM と BDI モデルの推論サイクルの類似に着目し、前者による記述での直接実行に、BDI モデルの構成要素を取り込む形をとっている。特に、BDI モデルでのプランの選択 (目的–手段推論)、実行する意図の選択 (熟考) の部分は、論理的には非決定的に書かれ、別途選択関数を記述する形を取る。このため、意味論はこの選択関数でパラメタライズされることになり、プログラムの意味に論理外のもので規定される部分ができるため、やはり論理体系を用いた形式的議論が難しくなる。その例として、$B\phi_i \rightarrow \Diamond\xi_i (i = 1, 2)$ と書いた場合、「ϕ_i を信じれば未来のいつか ξ_i を達成する」の意味になるが、もし、ξ_1 を達成するプランおよび ξ_2 を達成するプランのうち、実行するものを選択する熟考が選択関数で実装され、それが後者のみを選ぶように作られると、$B\phi_1$ が成り立っても ξ_1 がいつまでも成り立たず、式の論理的な意味に合わないケースが起こりうることになる。我々のものではこうしたことは起こらない。

3APL は、信念と願望 (目標) は宣言的に記述されるが、意図の記述部分はプランニングシステムとして実現され、手続的に書かれる。従ってシステムとしては Jason と同様のメカニズムを持ち、意味論も操作的である。このため、我々のもののように、論理式をそのまま実行可能なプログラムとして扱うというものではない。特に、論理体系との意味論のずれという観点からは、Jason と同様の問題点がそのまま残る。

3 実装の提案

BDI モデルにおける心的状態は、時間の流れの中で保持されるものである。しかし BDI ロジックに基づく証明システムを作る際に、単に各時刻で証明を行うような実現では、時刻毎に証明されたことのみが成り立ち、時刻間における状態の関係はないので、心的状態が保存されないことになる。

BDI ロジックに基づいたシステムを作るためには、このような心的状態のあり方を再現する必要がある。すなわち、時刻が経過しても、論理式の成立が持続するような機構が必要となる。

3.1 プログラムに記述できるオペレータ

BDI ロジックでは、現地点から未来方向に無限に伸びた時間木が複数存在する離散構造をモデルとして扱う。その時間構造を図 1 に示す。

BDI ロジックの時間構造内でのそれぞれのノードで成り立つ論理式をプログラムで表現するために、時相オペレータを導入する。

- 未来に関するもの
  - $\bigcirc\, p$; 次の時刻で p が成り立つ。

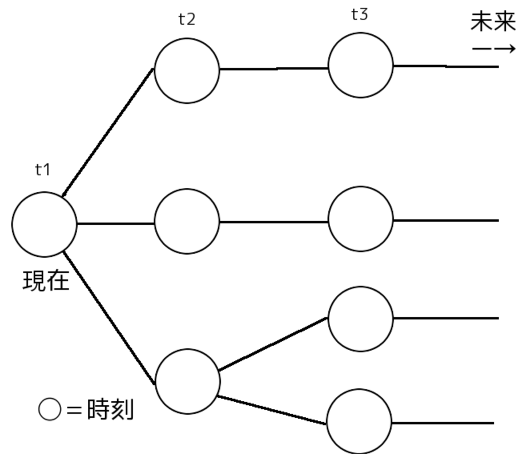

図 1: 時間構造

- □ p ; 未来永劫 p が成り立つ。
- p *until* q ; q が初めて成り立つ直前の時刻まで p が成り立つ。
- p *atnext* q ; q が初めて成り立つとき、p も成り立つ。

- 過去に関するもの
  - ■ p ; 現在まで p は常に成り立っている。
  - ◆ p ; 現在までに p が成り立ったことがある。
  - p *since* q ; 最も最近に q が成り立って以来、その時刻も含めて現在まで p は常に成り立っている。
  - p *after* q ; 最も最近に q が成り立って以後、その時刻も含めて現在まで p が成り立ったことがある。
  - p *for* n; 現在も含めて過去 n 回連続して p が成り立った。

- 心的オペレータ
  - BEL(p) ; 「p を信じる」
  - DES(p) ; 「p を願望する」
  - INT(p) ; 「p を意図する」

これらのオペレータを用いて、エージェントの性質や振る舞いを論理式で表現し、それをプログラムとして走らせることで、その論理式の宣言的意味と一致するような行動を行わせる。

過去に関するオペレータは、本来の BDI ロジックにはないが、記述の便宜のために導入した。これらは、後述の変換過程で全て◯オペレータに変換されるため、導入しても支障はない。

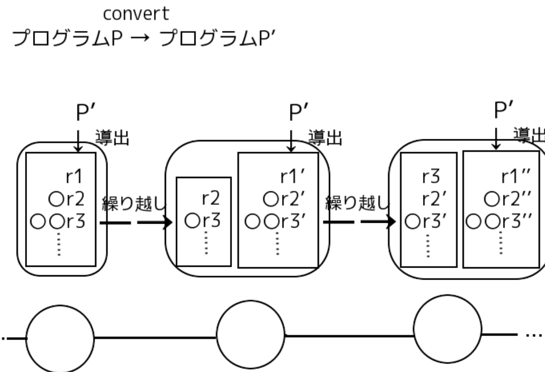

図 2: システム内部

3.2 システムのモデル概要

時相オペレータを含む論理式をそのまま扱おうとすると、過去に起こったこと全てを記憶しておかねばならないことがある。たとえば、■ $p \rightarrow q$ のような論理式がプログラムにあるとすると、p が始点からずっと成り立つ必要がある。すると、p が過去に成り立ったことは以後永久に記憶しておかなければならない。これでは、システムの実行開始後、時間がたつほど記憶すべきことが増えてしまう問題がある。

Temporal Prolog[桜川 87] では、時相に関するオペレータを全て●オペレータに変換することによって、一時刻前に導出したもののみを記憶しておけばよいようにして、この問題を解決している。我々のシステムもこのアイデアを用いている。

我々の提案するシステムにおいては、図 1 で時刻がノードからノードへ移り変わる度に、その時刻で成り立つ論理式を割り出す計算が行われる。ノードとは、離散構造における点時刻のことであり、図 1 では、線で繋がれた◯がそれにあたる。

その計算法は、まずプログラムを、確定節[1]に近い論理式の形、つまり 3.4 節で述べる特別な形の論理式の集合に変換してから、時間の経過にそって変換後のプログラム節を各時刻で評価し、時刻毎に成り立つ論理式を導出させるもので、[桜川 87] における Temporal Prolog のアイデアを用いている。この手法の利点は、計算の過程そのものがエージェントの振る舞いにおける証明と一致することである。

変換後のプログラム節は、体部には時相オペレータはなく、また頭部にはたかだか◯ (次の時刻で成り立つ) というオペレータのみがつく。よって、動作させるために記憶しておかなければならないものは、現時刻において一時刻後に成り立つとわかっているもの、つまり一番外側に◯のオペレータの付いているもののみで十分となる。

ステップ毎のノードの状態は、

1. その時刻に導出された論理式の集合

[1] 1 つの肯定形のリテラルをもつ節を確定節と呼ぶ。$p_1 \wedge p_2 \wedge ... \wedge p_k \rightarrow r$、($p_j, r$ はリテラル) というような形をしている。

2. 前の時刻で導出された論理式の中で◯が一つ以上ついたものを、今の時刻に繰り越し◯を一つ除去したものの集合

これらを合わせたものから成る。この様子を図 2 に示す。

3.3 変換のアルゴリズム

ここでは、プログラムを確定節に近い形の集合に変換するためのアルゴリズムについて説明する。変換の過程を step1 と step2 に分割し、step1 では主に未来時相オペレータの一部を除去するための変換を行い、step2 では過去時相オペレータを除去するための変換を行う。除去の対象となるオペレータは、過去時相オペレータ全てと、◯（次の時刻で成り立つ）以外の未来時相オペレータ全てである。時相オペレータの持つ意味を変えないような変換を段階的に行うことで、最終的に得られる結果は、元のプログラムと意味が変わらないことが保証される。図 3 と図 4 では、$\Rightarrow$ の左側は変換前の論理式で、$\Rightarrow$ の右側は変換後の論理式の集合を表している。

まず、条件として使われる condition formula の定義を行う。

- $CF(conditionformula)$ の定義
  1. $a \in BDI_AF$ ならば、$a \in CF$
  2. $c, d \in CF$ ならば、$\blacksquare\, c$, $\blacklozenge\, c$, c *since* d, c *after* d,c *for* $n \in CF$, $c \wedge d \in CF$
- $AF(atomic\ \ formula)$ の定義
  1. 述語記号 p の arity が k で、$t_1, ..., t_k$ が項ならば、$p(t_1, ..., t_k) \in AF$
- $BDI_AF(BDI\ \ atomic\ \ formula)$ の定義
  1. a が atomic formula ならば、$BEL(a)$,$DES(a)$,$INT(a) \in BDI_AF$
  2. $a \in BDI_AF$ ならば、$BEL(a)$, $DES(a)$, $INT(a) \in BDI_AF$
- $R(result)$ の定義
  1. $a \in BDI_AF$ ならば、$a \in R$
  2. q, $r \in R$, $c \in CF$ ならば、$c \rightarrow r$, $q \wedge r$, $\bigcirc\, q$, $\square\, q$, q *until* c,q *atnext* $c \in R$

以下、$a, b \in CF$、$q, s \in R$、$p \in AF$ とする。
変換の過程を step1 と step2 に分割して述べる。

3.3.1 step1

step1 では、プログラム中の □、*until*、*atnext* などを消す。
後述の停止条件が成り立つまで以下の変換を繰り返す。
$\Rightarrow$ の左の形をしたものがあれば、$\Rightarrow$ の右のものに置き換える。ただし p は $p_i(x_1, \cdots, x_k)$ の形で p_i は新しい述語記号。$x_1 \sim x_k$ は a に現れる全ての自由変数。step2 でも同様。
停止条件は、プログラム内の論理式全てが、$a \rightarrow r$ または r という形をしていることである。ここで r は、◯...◯ p の形で $p \in BDI_AF$ である。

- $q \wedge s \Rightarrow q$
  s
- $a \rightarrow q \wedge s \Rightarrow a \rightarrow q$
  $a \rightarrow s$
- $\Box\, q \Rightarrow q$
- $a \rightarrow \Box\, q \Rightarrow a \rightarrow p$
  $p \rightarrow \bigcirc\, p$
  $p \rightarrow q$
- $q\ \mathit{until}\ b \Rightarrow \neg b \rightarrow q$
- $a \rightarrow q\ \mathit{until}\ b \Rightarrow a \rightarrow p$
  $p \wedge \neg b \rightarrow \bigcirc\, p$
  $p \wedge \neg b \rightarrow q$
- $q\ \mathit{atnext}\ b \Rightarrow b \rightarrow q$
- $a \rightarrow q\ \mathit{atnext}\ b \Rightarrow a \rightarrow p$
  $p \wedge \neg b \rightarrow \bigcirc\, p$
  $p \wedge b \rightarrow q$

図 3: step1 の変換規則

3.3.2 step2

step2 では、過去の時相オペレータを消す。ここで、start は始点でしか成り立たない atom とする。step2 の変換規則を図 4 に示す。

次の停止条件が成り立つまで、図 4 の変換を繰り返す。

停止条件は、プログラム内の論理式全てに、過去に関するオペレータが現れないことである。

3.4 プログラムとして実行可能な論理式の形

プログラムとして記述された論理式は、その意味を変えないように段階的に変換が施され、変換後にある条件を満たすプログラムだけを legal なプログラムとする。その条件は、プログラム内の論理式全てがある特別な形に変換されていることであり、その形の特徴を下記に示す。

$$c_1 \wedge c_2 \ldots . \wedge c_k \rightarrow r$$
$$c_j \in BDI_AF$$

r については、$\bigcirc \ldots \bigcirc\, a$, ここで $a \in BDI_AF$ という形をしたものである (ただし○はなくてもよい)。$c_1 \wedge c_2 \ldots . \wedge c_k$ は、現在に成り立つ論理式の連言であり、r は、特定の未来に成り立つ論理式を表している。

この特殊な形というのは、直感的にいえば、現在に起きたことを条件として、現在から未来において起きることを規定している。

Temporal Prolog も同様に時相オペレータについての変換を行うが、変換後の論理式の集合は、特定の過去から現在に成り立った論理式を条件に、現在成り立つものを導出するような形になっている。

- $...\blacksquare\, a... \to r \quad \Rightarrow \quad ...p... \to r$
  $a \wedge start \to p$
  $p \to \bigcirc q$
  $a \wedge q \to p$
- $...\blacklozenge\, a... \to r \quad \Rightarrow \quad ...p... \to r$
  $a \to p$
  $p \to \bigcirc p$
- $...a\ since\ b... \to r \quad \Rightarrow \quad ...p... \to r$
  $a \wedge b \to p$
  $p \to \bigcirc q$
  $q \wedge a \to p$
- $...a\ after\ b... \to r \quad \Rightarrow \quad ...p... \to r$
  $b \to q$
  $q \to \bigcirc q$
  $q \wedge a \to p$
  $p \to \bigcirc s$
  $s \wedge \neg b \to p$
- $...a\ for\ n... \to r \quad \Rightarrow \quad a \to \bigcirc p_1$
  $p_1 \wedge a \to \bigcirc p_2$
  $p_2 \wedge a \to \bigcirc p_3$
  ....
  $p_{n-2} \wedge a \to \bigcirc p_{n-1}$
  $p_{n-1} \to r$

図 4: step2 の変換規則

BDI モデルの場合、3.1 の図 1 にも示したように、時間構造は現在から未来方向に伸びているため、変換後の論理式の形として、現在に成り立つものから、現在から未来にかけて成り立つものを導き出すものに限定した。

ここでの論理結合記号 $\wedge$ は binary でなく、いくつでも引数をとるものとして扱う。つまり、$a \wedge (b \wedge c)$ も $(a \wedge b) \wedge c$ も $a \wedge b \wedge c$ と書いて区別しない。また、$a \to b \to c$ は、$a \wedge b \to c$ と同じ意味であるとする。

変換後、論理式内に残った時相オペレータ$\bigcirc$や BDI の心的オペレータについては、これを述語の性質の一種と捉えることで、論理式を一階述語として扱うことができる。また、変換後の形が確定節として扱えることから、これらを用いた推論を行う過程を、Prolog の処理系を使うことによって実現できる。

前述したように、プログラム変換後の論理式は、Prolog で処理可能な形になっている必要があり、そのためにはプログラム中に記述する論理式の形も制限をしなければならない。3.3 に示した集合 R の要素である論理式ならば、少なくとも確実に legal なプログラムに変換できる。

3.5 変換の正当性

上記で述べた変換は、主に時相オペレータを消すためのものであり、それらが持つ意味を変えないように変換していると前に述べた。ここでは、変換によって、プログラムの持つ意味が結果的に変わっていないこと、また意味通りの動作をすることが保証されていることを直観的な議論によって示す。

3.5.1 変換例

後述の図 4.3 で示すプログラム図 10 内の論理式 (6) について変換を行う。

$$INT(\text{ソーダを飲む}) \rightarrow INT(\text{ソーダを所有}) \;\; until \;\; BEL(\text{ソーダを所有})$$

このプログラムを変換すると以下のようになる。

$$INT(\text{ソーダを飲む}) \;\rightarrow\; p \tag{1}$$

$$p \land \neg BEL(\text{ソーダを所有}) \;\rightarrow\; \bigcirc p \tag{2}$$

$$p \land \neg BEL(\text{ソーダを所有}) \;\rightarrow\; INT(\text{ソーダを所有}) \tag{3}$$

この変換後の論理式がどのように解釈されるのかを以下に述べる。

式 (1) では、INT(ソーダを飲む) が成り立つと、新しく生成された述語記号 p が成り立つことを意味する。また式 (2) では、現時刻で p が成り立ち、かつ BEL(ソーダを所有) が成り立っていなければ、次の時刻で p が成り立つことを示している。つまり p が成り立つには、その時刻に INT(ソーダを飲む) が成り立つか、前の時刻に p かつ $\neg BEL$(ソーダを所有) が成り立てばよい。再帰的に追っていくと、p が成り立つことは、INT(ソーダを飲む) が成り立ってから一時刻前までずっと BEL(ソーダを所有) が成り立っていないことを示していることがわかる。

また式 (3) は、過去における条件 (p) かつ現時点での条件 ($\neg BEL$(ソーダを所有)) を満たしていれば、次の時刻に INT(ソーダを所有) が成り立つと解釈できる。

以上からこれらの式は、INT(ソーダを飲む) が成り立ってから現時刻までずっと BEL(ソーダを所有) が成り立っていないならば、現時刻で INT(ソーダを所有) が成り立つことを表している。言い換えれば、INT(ソーダを飲む) が成り立ってから BEL(ソーダを所有) が成り立つ直前までは INT(ソーダを所有) が成り立つことになり、変換前の式と同じ意味を持つことが分かる。

3.5.2 状態の時間的変化の例

次に、3.5.1 の変換によって得られたプログラムを実行した場合の、状態の時間的変化の具体例として、図 5 にあるような INT(ソーダを飲む), BEL(ソーダを所有) の遷移に対し、INT(ソーダを所有) がいつ成り立つのかを述べる。

図 5 では、時刻 t2 で INT(ソーダを飲む) が成り立ち、これが until 式のトリガーの役割を果たす。p は、トリガー INT(ソーダを飲む) が発生すると無条件に成立し (式 (1))、そこから BEL(ソーダを所有) が成り立つまで (t2-t5) ずっと成り立つ。この p は、出力結果には直接関係しないが実行系内部でフラグのような役割を果たし、過去における、INT(ソーダを所有) が成り立つための必要条件が満たされているかを判定するのに使用される。図にある黒文字の論理式の遷移の中で、式 (3) が評価された結果、INT(ソーダを所有) は時刻 t2-t4 で成り立ち、t5 では成り立たないことが保証される。

図 5: until の遷移

変換後の論理式の宣言的意味を考えると、until の性質を満たしていることが明らかである。またそれらを評価して、それぞれの時刻で出力された結果をみると、until の持つ意味の通りに動いていることがわかる。

変換によって消されるオペレータすべてについて示すのは冗長になるため省略するが、他の時相オペレータも同様のことがいえる。

4 実装と評価

4.1 実行系の制御

本処理系は、Prolog で実装されている。処理系を実行開始するための述語が用意されており、この述語を実行することで開始される。

この述語には、第一引数に最初のステップ t1 に与える外部述語のリスト、第二引数に変換前のプログラム (論理式のリスト) を与える。

Prolog を起動しこの述語を実行すると、第一引数で与えたリストを元に時刻 t1(最初の時刻) で成り立つ論理式のリストが表示される。

その直後に、データ入力のプロンプトが表示されるので、そこに時刻 t2 で得られた外部述語のリストを入力すると、時刻 t2 で導かれる論理式のリストが表示される。tn 以下も同様にして得られる。

それぞれの時刻で出力される結果は、◯のついた論理式と心的オペレータのついた論理式のみのリストとなる。これによって、主要な論理式の遷移の流れを見ることができる。単にエージェントを動作させるだけならば、INT(does(述語)) という形の基本行為 (これ以上分解できない意図) のみを参照すればよい。

実行系内部では、まず与えられたプログラム (論理式のリスト、これを A とする) を 3.3 で説明した手順で変換し、そのリストを A' とする。この変換は初めの一回しか行われず、ループは以下の手順で行われる。

次に示す手順は、初めのステップでは行われない。まず、入力で与えられた外部述語のリストを節に変換した後、リスト A' に追加する (これをリスト B とする)。次に、前時刻で導出されたものの中で、◯の付いた論理式を抽出し、◯を取り外し節に変換したものを、リスト B に追加する。

このリストを元にその時刻での評価を行うが、すべての節を評価するのではなく、心的状態を表す論理式または、未来に持ち越される論理式が導出される可能性のある節のみを対象にする。つまり、プログラムの頭部に該当するような論理式に、心的オペレータまたは、◯の付いた論理式が見つかれば、その節は評価の対象となる。

節を評価した後、導出された論理式の集合をリストとして出力して、またループの先頭に戻り、このような手順を繰り返す。

4.2 性能評価

本処理系が、実際にどれくらい実用性があるのかを検証するため、性能評価を行った。性能評価は、プログラムの変換の速度、および変換後のプログラムの実行速度の計測を行った、その結果をここに示す。この性能評価は、以下のような環境下で行った。

CPU... intel Celeron 420 1.600GHz

メモリ... 500MB

OS... Linux 2.6.16

ソフト... SWI-Prolog 5.6.64

まず、最初に行う変換処理の計算量について検証を行う。検証は、一つの原始論理式に同じオペレータを入れ子状に何個か付けたもの (例えば、$q \rightarrow \blacksquare\blacksquare\, a$ や、$q \rightarrow (a\ \mathit{until}\ b)\ \mathit{until}\ c$ など) を変換し、その計算量を計測した。その結果が図 6 と図 7 で、x 軸が重ねて付けたオペレータの数で、y 軸が inference(Prolog の処理において単一化した回数) の数である。2 つ引数があるオペレータについては、一番計算量のかかる組合せの検証をしたが、$a\ \mathit{until}\ b$ や $a\ \mathit{atnext}\ b$ にあるような未来時相オペレータについては、b の位置に未来時相オペレータの付いた論理式を置くと legal な論理式ではなくなるため、a の位置に未来オペレータがすべて収まるような論理式についての変換を行った。

まず図 6 を見ると、$\wedge, \Box, \blacklozenge, atnext, until$ についてはこの順で inference の数が多く、どれも直線を描いており、計算量は $O(n)$ と考えられる。

計算量は同じでも、係数に多少差が生じるのは、変換処理に、新しい述語記号が生成される手順が含まれているかなどの違いが、inference の数に影響していると考えられる。

次に、図 7 を見ると、$for, \blacksquare, since, after$ の順で inference の数が多く、指数関数的に増大している。

図 6 の直線を描いているオペレータと計算量が違う原因として、変換の際、論理式の一部分が複製されることにより計算量が倍増することが考えられる。(例えば、$a\ \mathit{atnext}\ b \rightarrow r$ の a が論理オペレータを含む場合、その式が変換により 2 つに増える) 複製された論理式の中に、さらに複製されたようなオペレータが含まれる場合、計算量は 4 倍に膨れあがり、それが n 回続くと、計算量は $O(2^n)$ となる。

次に、変換後のプログラムから時刻毎に成り立つものを導出するのに、どれだけ時間がかかるのかを検証した。

先ほどの実験結果より、1 つの論理式の中に、$after$ を重複して付けたものが一番負荷がかかることがわかり、また、変換後のプログラム量も必然的に一番大きくなる。そこで、検証には、$after$ を 4 個から 10 個まで付けた論理式を変換したものを用意し、それぞれについて 4.1 で述べたルー

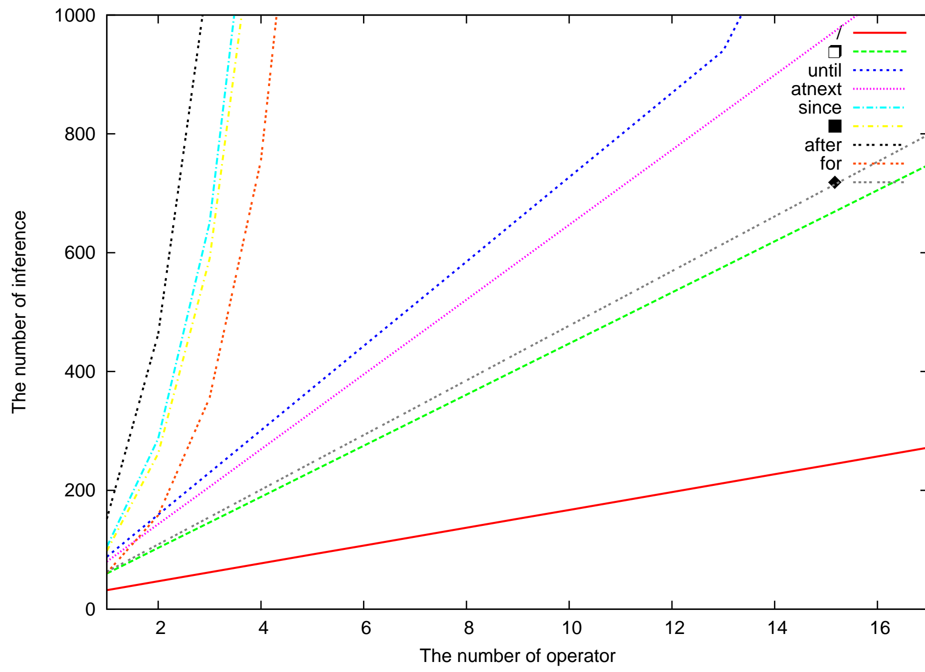

図 6: 変換の計算量 (xrang[0:17])

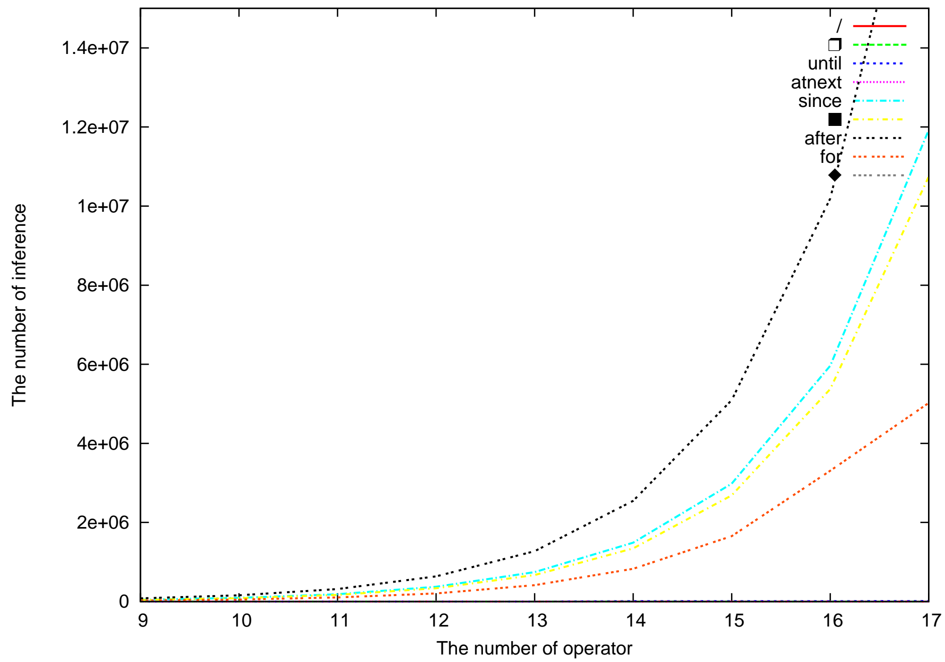

図 7: 変換の計算量 (xrang[9:17])

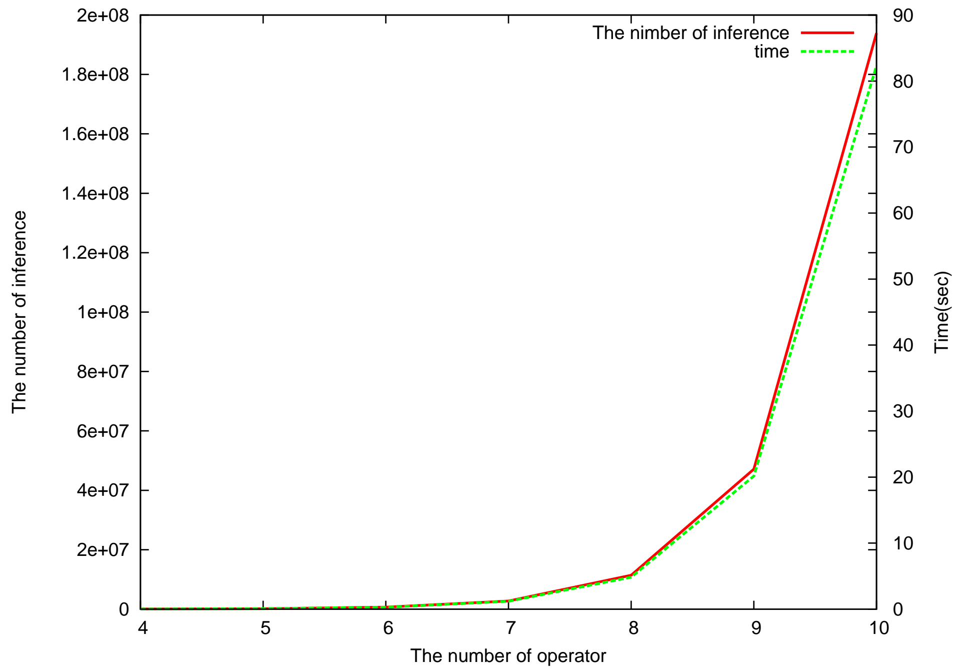

図 8: 一時刻分の計算量 (xrang[4:10])

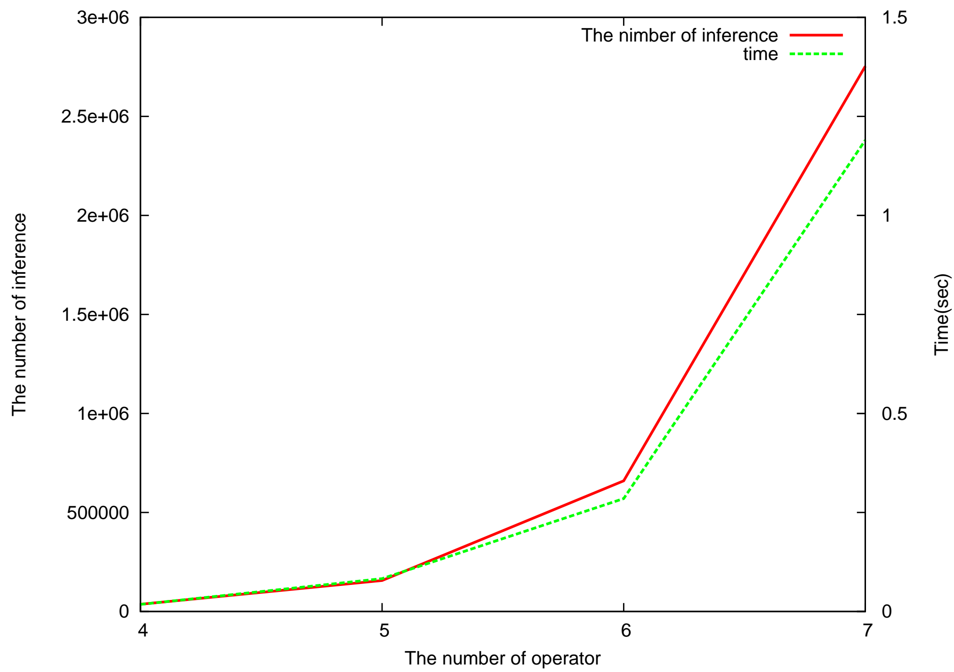

図 9: 一時刻分の計算量 (xrang[4:7])

プを自動で回した。またプログラム量が、評価の処理にかかる負荷にそのまま反映されるように、変換したすべての式が評価されるような外部述語を与えた。

図 8, 図 9 には、それぞれの一時刻の評価にかかる、時間と inference 数の平均を載せている。

図 8 を見ると、評価を行う処理も、変換処理と同様に $after$ の数 n に対して計算量 $O(2^n)$ が必要となることがわかる。時間に関しては、用途によって必要とされる速度はまちまちであるが、熟考型エージェントの場合、一時刻 0.1 秒 ~1.0 秒くらいであれば十分と考えると、一つの論理式に最大オペレータ 7 つ (1.1sec) 付けることが可能となる。しかし、プログラムの論理式は一つとは限らないので、実際に記述する場合を考えると、一つの論理式にオペレータを 4 つ (0.018sec) 付けたものを、60 ほどまで記述することが実用的に可能であると考えられる。

実用的に記述されるエージェントプログラムは、この範囲内に十分収まると考えられる。よって、本処理系には十分な実用性があるといえる。

4.3 プログラム例

プログラム例を図 10 に示す。

この例は、エージェントが喉の渇きを癒すために、冷蔵庫からソーダを取り出し飲むといった一連の流れをプログラムで記述したものである。

なお、エージェントが行う基本行為は全て成功することを前提としており、失敗した時の対処に関する記述は省いた。

まず、エージェントは喉の乾きを感じて、それを癒すために「喉が潤う」という信念を得るまで「喉を潤う」という願望を保持する (式 (4))。このとき信念として、「蛇口に水がある」という事実式 (5) と、「ソーダを取り出す」という信念を得るまで「ソーダが冷蔵庫にある」という信念を保持する (式 (6))。

これらの結果、式 (7) の X と Y にソーダと冷蔵庫が代入され、「喉が潤う」という願望と「ソーダが冷蔵庫にある」という信念をもつならば、「ソーダを飲む」ことが信念に加わるまで「ソーダを飲む」願望を保持するという規則を使い「喉が潤う」願望を達成しようとする (式 (7))。ここで「ソーダを飲む」という願望をまず達成するための目的とし、「ソーダを飲む」信念が得られるまで「ソーダを飲む」ための意図を形成する (式 (8))。

この規則における副目標として、「ソーダを飲む」という意図を形成するため、「ソーダを所有」するまで「ソーダを所有」するための意図を形成する。式 (9) も同様に解釈できる。

さらに、「冷蔵庫を開ける」という意図を形成するために、足が駆動可能な状態「available(足)」ならば、次の状態で「冷蔵庫まで移動」するという意図を形成する (式 (11))。そして冷蔵庫まで移動できて「succeed(冷蔵庫まで移動)」が信念に加わり、かつ手が駆動可能な状態「avalable(手)」であるならば、次の時刻に「冷蔵庫を開ける」ことを実行する (式 (12))。

冷蔵庫を開けることに成功「succeed(冷蔵庫を開ける)」したことが信念が加わったならば、「冷蔵庫が閉まっている」を信念に持つまでは「冷蔵庫が開いている」ことを信念に持つ。これらの結果、「ソーダを所有」する意図を保持しかつ「冷蔵庫が開いている」信念を持つならば、「ソーダを取り出す」まで「ソーダを取り出す」意図を形成する (式 (13))。

これより、「ソーダを取り出す」意図を形成したならば、手が駆動可能な状態「available(手)」であることとソーダが知覚できる「percept(ソーダ)」ならば次の時刻に「ソーダを取り出す」意図を形成する (式 (15))。

以下式 (16)、式 (17)、式 (18) も同様にして解釈することにより「ソーダを飲む」ことが信念に加わることにより「喉が潤う」という願望を満たすことが可能となり、エージェントの願望が達成

$$DES(\text{喉が潤う}) \quad until \quad BEL(\text{喉が潤う}) \tag{4}$$

$$\Box\, BEL(\text{蛇口に水がある}) \tag{5}$$

$$BEL(\text{ソーダが冷蔵庫にある}) \quad until \quad BEL(\text{ソーダを取り出す}) \tag{6}$$

$$DES(\text{喉が潤う}) \wedge BEL(X\text{が}Y\text{にある}) \rightarrow DES(X\text{を飲む}) \;\; until \;\; BEL(X\text{を飲む}) \tag{7}$$

$$select_goal(DES(X\text{を飲む})) \rightarrow INT(\text{ソーダを飲む}) \;\; until \;\; BEL(\text{ソーダを飲む}) \tag{8}$$

$$INT(\text{ソーダを飲む}) \rightarrow INT(\text{ソーダを所有}) \;\; until \;\; BEL(\text{ソーダを所有}) \tag{9}$$

$$INT(\text{ソーダを所有}) \rightarrow INT(\text{冷蔵庫を開ける}) \;\; until \;\; BEL(\text{冷蔵庫が開いている}) \tag{10}$$

$$INT(\text{冷蔵庫を開ける}) \rightarrow INT(does(\text{冷蔵庫まで移動})) \;\; atnext \;\; BEL(available(\text{足})) \tag{11}$$

$$INT(\text{冷蔵庫を開ける}) \rightarrow INT(does(\text{冷蔵庫を開ける})) \;\; atnext \;\; BEL(available(\text{手})) \wedge BEL(succeed(\text{冷蔵庫まで移動})) \tag{12}$$

$$BEL(succeed(\text{冷蔵庫を開ける})) \rightarrow BEL(\text{冷蔵庫が開いている}) \;\; until \;\; BEL(\text{冷蔵庫が閉まっている}) \tag{13}$$

$$INT(\text{ソーダを所有}) \wedge BEL(\text{冷蔵庫が開いている}) \rightarrow INT(\text{ソーダを取り出す}) \;\; until \;\; BEL(\text{ソーダを取り出す}) \tag{14}$$

$$INT(\text{ソーダを取り出す}) \rightarrow INT(does(\text{ソーダを取り出す})) \;\; atnext \;\; BEL(percept(\text{ソーダ})) \wedge BEL(avairable(\text{手})) \tag{15}$$

$$BEL(succeed(\text{ソーダを取り出す})) \rightarrow BEL(\text{ソーダを所有}) \wedge BEL(\text{ソーダを取り出す}) \tag{16}$$

$$INT(\text{ソーダを飲む}) \rightarrow INT(does(\text{ソーダを飲む})) \;\; atnext \;\; BEL(\text{ソーダを所有}) \tag{17}$$

$$BEL(succeed(\text{ソーダを飲む})) \rightarrow BEL(\text{ソーダを飲む}) \tag{18}$$

図 10: プログラム例

t1	t2	t3
DES(喉が潤う)	DES(喉が潤う)	DES(喉が潤う)
BEL(蛇口に水がある)	BEL(蛇口に水がある)	BEL(蛇口に水がある)
BEL(ソーダが冷蔵庫にある)	BEL(ソーダが冷蔵庫にある)	BEL(ソーダが冷蔵庫にある)
DES(X を飲む)		
DES(水を飲む)		
DES(ソーダを飲む)	DES(ソーダを飲む)	DES(ソーダを飲む)
select_goal(DES(X を飲む))	INT(ソーダを飲む)	INT(ソーダを飲む)
	INT(ソーダを所有)	INT(ソーダを所有)
	INT(冷蔵庫を開ける)	INT(冷蔵庫を開ける)
	INT(does(冷蔵庫まで移動))	BEL(succeed(冷蔵庫まで移動))
		INT(does(冷蔵庫を開ける))

t4	t5	t6
DES(喉が潤う)	DES(喉が潤う)	
BEL(蛇口に水がある)	BEL(蛇口に水がある)	BEL(蛇口に水がある)
BEL(ソーダが冷蔵庫にある)	BEL(ソーダを取り出す)	
DES(ソーダを飲む)	DES(ソーダを飲む)	BEL(ソーダを飲む)
INT(ソーダを飲む)	INT(ソーダを飲む)	BEL(ソーダを飲む)
INT(ソーダを所有)	BEL(ソーダを所有)	
BEL(冷蔵庫が開いている)	BEL(冷蔵庫が開いている)	BEL(冷蔵庫が開いている)
BEL(succeed(冷蔵庫を開ける))		
INT(does(ソーダを取り出す))	BEL(succeed(ソーダを取り出す))	
	INT(does(ソーダを飲む))	BEL(succeed(ソーダを飲む))

図 11: 状態の時間的変化

される。
図 11 は図 10 を実行し、時刻毎に成り立つものを、それぞれ列挙しまとめたものである (t_1 がプログラムの開始時刻)。○のついた論理式については、基本的にある性質が成り立つための条件のような役割を果たしており、それ自体に特別意味はないので、この図 11 では省略した。

5 まとめ

本論文では、プログラムを節に似た形に変換し、時間の経過とともに各時刻で成り立つことを求めていく Temporal Prolog の手法を、BDI エージェントの実装システムに応用し、BDI ロジックの論理式をプログラムとして直接実行できるような実行系を構築した。これによって、エージェントの論理的性質をプログラムに直接記述できるようになった。

実装における今後の課題としては、心的オペレータについての変換方法の提案、導入が挙げられる。本研究の実装では、心的オペレータは単なる一階述語論理式として扱われており、それについての変換は行われていない。例えば、$BEL(INT(a))$ や、$DES(\Box\, a)$ といった、外側に心的オペレータがネストしている形の場合、現時点でそれ以上の変換はできない。BDI エージェントの実装システムとしては、この変換は必須であり、BDI ロジックの論理体系に即した変換の手法を導入する必要がある。

この実行系の処理の問題点としては、プログラムの変換処理の際に、論理式が複製されることによって、現在の実装では同じ処理を複数回行うこととなり、計算量が倍増することが挙げられる。改善手法としては、すべてを再帰的に処理をするのではなく、一度行った処理の結果を記憶しておき、処理が同じところは省くように設計することによって計算量を減らすことが考えられ、今後の課題である。

次に、記述における問題点として、記述が複雑になる点が挙げられる。この実行系は、一時的に成り立つ性質については記述しやすいという利点があるが、BDI モデルの性質として基本的に保持しなければならない心的状態などの性質を記述する場合、いつまで成り立つかを明示的に記述する必要があり、その例としては、サブプランを順に実行するような自然な過程も、$INT(a) \rightarrow INT(b)\, \mathit{until}\, BEL(a)$ のように複雑な式になってしまう。対処の一つとしては、頻出する形の論理式はマクロで短く表記できるようにするなどの案が考えられる。具体的には、$INT(a) \rightarrow INT(b)\, \mathit{until}\, BEL(a)$ を、a;b と略記できることにすれば、一連のサブプランを実行する過程は、a1; a2; a3; .... のように簡潔に書ける。

もう一つの問題点として、記述の自由度が高い分、柔軟な記述はできるが、何をいつまで保持するかを全てユーザーが把握し、整合性の合うよう細部まで気を配る必要があり、記述の厳密性が求められるという点が挙げられる。

例えば、複数の意図を扱う場合、エージェントの振る舞いが、整合性を保ちかつ簡潔に記述できるかが課題となる。そのためには、意図を選択する時点で競合しないようなものを見極め、それらを並行に実行させたり、競合した場合の対処として、どちらかの意図を一旦中断し、実行可能な状態になったら再開するというような記述が必要となってくる。

その一環として、ここでは資源割当ての例を考える。一つの資源を複数の意図が、非決定的に相互排除を行いながら利用する場合、資源割り当てのコードは以下のように書ける。ここで、それぞれの意図には何らかの ID が割り当てられ、変数 X にはその ID が入るものとする。また、$assign(X)$ は、「X が資源割当てを要求している」、$assigned_to(X)$ は、「X に資源が割り当てられている」を

表す。

$$assign(X) \land previously_assigned_to(X) \rightarrow assigned_to(X) \tag{19}$$

$$assigned_to(X) \rightarrow \bigcirc\, previously_assigned_to(X) \tag{20}$$

$$assign(X) \land \neg assigned_to_another(X) \rightarrow assigned_to(X) \tag{21}$$

$$assigned_to(X) \land \neg(X = Y) \rightarrow assigned_to_another(Y) \tag{22}$$

変数 X にある意図の ID が代入された場合、X が資源を確保する十分条件は、

1. X が資源の使用を要求し、かつ一時刻前に X が資源を使用していた場合
2. X が資源の使用を要求し、かつ現在、資源が X 以外の意図に割り当てられていない場合

となる。これは、[桜川 87] の記述にあるプログラム例をもとに改変したものである。

このように記述することで、1 つの資源を 2 つの意図が同時に使用するというようなことは起こらず、整合性が保てる。しかし上記のプログラムは、論理プログラムとしては問題ないが、実際に本処理系で実行すると、資源を割り当てる意図を特定しようとする際に、ゴール $assigned_to(X)$ を解こうとして、式 (21) と式 (22) により無限ループに陥ってしまう。本処理系は Prolog で実装されており、Prolog の機構に制限を受けてしまうためである。

このように、プログラム記述上は Prolog の制約も考慮する必要があり、将来的にはこの部分に別な機構、例えば、並行論理プログラミングを用いて非決定性を表現するなどの手法を適用することで、柔軟さを向上する可能性も考えられる。ただし、実用上は Prolog のままでも、以下の記述によりランダムな資源割当てを行うことは可能である。ただし、資源を要求している意図の中から一つをランダムに選ぶ $randomly_assigned_to(X)$ という述語が別に実装されているものとする。

$$assign(X) \land previously_assigned_to(X) \rightarrow assigned_to(X) \tag{23}$$

$$assigned_to(X) \rightarrow \bigcirc\, previously_assigned_to(X) \tag{24}$$

$$\begin{aligned} &assign(X) \land \neg previously_assigned_to(X) \\ &\qquad \land randomly_assigned_to(X) \rightarrow assigned_to(X) \end{aligned} \tag{25}$$

今後このような手法を可能にすることで、複雑なエージェントの振る舞いを記述し、現実世界で使えるような有用なものにしていきたいと考えている。

参考文献

[Bordini 07] Bordini, R. H., Hübner, J. F., and Wooldridge, M.: *Programming multi-agent systems in AgentSpeak using Jason*, WILEY (2007)

[Dastani 03] Dastani, M., Riemsdijk, van B., Dignum, F., and Meyer, J.: A Programming Language for Cognitive Agents: Goal Directed 3APL, in *Programming Multi-Agent Systems*, pp. 111–130, Springer (2003), 1st International Workshop, PROMAS 2003. Selected Revised and Invited Papers

[Fisher 93] Fisher, M.: Concurrent MetateM — A Language for Modelling Reactive Systems, in *Proc. of PARLE '93*, pp. 185–196 (1993)

[Fisher 97] Fisher, M.: Implementing BDI-like Systems by Direct Execution, in *Proc. of IJCAI '97*, pp. 316–321 (1997)

[Móra 99] Móra, M. C., Lopes, J. G., Viccari, R. M., and Coelho, H.: BDI models and systems: Reducing the gap, in Müller, J. P., Singh, M. P., and Rao, A. S. eds., *Proc. of ATAL-98, LNCS 1555*, pp. 11–27, Springer Verlag (1999)

[Rao 91] Rao, A. S. and Georgeff, M. P.: Modeling Rational Agents within a BDI-Architecture, in *Proc, of International Conference on Principles of Knowlegde Representation and Reasoning*, pp. 473–484 (1991)

[Shanahan 99] Shanahan, M.: The Event Calculus Explained, in Wooldridge, M. J. and Veloso, M. eds., *Artificial intelligence today*, pp. 409–430, Springer-Verlag (1999)

[Singh 99] Singh, M. P., Rao, A. S., and Georgeff, M. P.: Formal method in DAI, *Multiagent systems: a modern approach to distributed artificial intelligence* (1999)

[桜川 87] 桜川 貴司：Temporal Prolog: A Programming Language Based on Temporal Logic, コンピュータ・ソフトウェア, Vol. 4, No. 3, pp. 199–211 (1987)