動的環境下における反射的行動を可能とするマルチエージェントシステムの基本設計と実装

奈良女子大学 大学院人間文化研究科 情報科学専攻
新出研究室　学籍番号 I07-16　野口 真理子

1　はじめに

BDI モデルは Rao らによって提案された BDI 論理 [5] に基づいた、自律エージェントのモデルの一つである。BDI モデルを使用したエージェントを実装する枠組は [6] で示されるように一般に使用されている。しかしマルチエージェントシステムに関する一般的な枠組のようなものは与えられていない。本研究では、BDI によるマルチエージェントシステムの設計指針を一般的枠組として与えることを目指す。

マルチエージェントシステムにおいて重要であるのがエージェント同士の協調である。マルチエージェントシステムを構築する場合、個々のエージェントの振る舞いの他にエージェント同士の相互作用や、単体のエージェントでは成し得ない目標を複数エージェントの共同作業によってどのように達成させるかを考えなければならない。BDI モデルにおいては、この共同作業を行なうために他のエージェントの心的状態を持つことが必要となる。しかし Rao らによる元来の BDI 論理では単体のエージェントの心的状態しか表せない。そこで、複数エージェントの心的状態を記述できるように BDI 論理を拡張する $\mathcal{LORA}$ が提案されており、これによってエージェントの相互心的状態や協調の形式的な記述が与えられている。しかし $\mathcal{LORA}$ で与えられているのは形式的な記述のみであり、実装に使用できる一般的な枠組は与えられていない。そのため BDI エージェントによるマルチエージェントシステムを構築する際は、一からシステムを設計する必要がある。本研究では、$\mathcal{LORA}$ の記述に対応した複数エージェントの協調に関する一般的な枠組を構築する。

エージェントシステムの扱う問題によっては、環境の変化に対して反射的行動が求められる。マルチエージェントシステムで反射的行動を実現するためには、通信や交渉ではなく即決した行為決定が求められる。本研究で提唱する枠組みでは、このような反射的行動を必要とする問題にも適応できるようにする。

2　BDI モデル

BDI モデルは合理的エージェントの行為決定過程をモデル化したものである。BDI モデルでは意図に沿った行動を行なうことにより、エージェントに目的を達成するための一貫的な行動をとらせることができる。BDI モデルによって実装されるエージェントを BDI エージェントと呼ぶ。本節では BDI モデルについて述べる。

2.1 BDI 論理

BDI 論理とは、Bratman の「意図の理論」に基づいた心的状態を表すオペレータを持つ時相論理体系の一つである。BDI 論理は時相論理体系 CTL*[3] に信念 (BEL), 願望 (DESIRE), 意図 (INTEND) といった様相オペレータを導入し、さらに述語論理に拡張したものである。φ を論理式とすると、エージェントの「φ を信じている」という心的状態は論理式 BEL(φ) で表される。同様に「φ を願望している」は DESIRE(φ)、「φ を意図している」は INTEND(φ) と表すことができる。例えば、BDI 論理では「ψ が成り立つまで φ である、と信じる」は論理式 BEL(Aφ U ψ) で記述できる (φ, ψ は論理式、A は「すべての未来で」を表す時相オペレータ)。

2.2 BDI アーキテクチャ

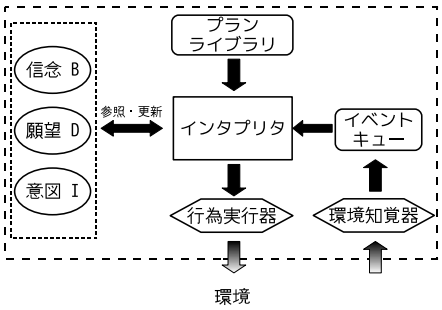

図 1: BDI アーキテクチャ

BDI アーキテクチャは BDI 論理に基づく、合理エージェントの実装方法の一つである。BDI アーキテクチャで実装されたエージェントの行動は、動的環境下における目標の達成のために推論・熟考して選択したプランに基づく。意図の理論に基づいた行為決定により、人間に近いエージェントの行為決定を可能とする。

BDI アーキテクチャは構成要素として、信念 (B), 願望 (D), 意図 (I) の 3 つの心的状態の他に、プランライブラリ、イベントキューとこれらを参照・更新するインタプリタ、更に環境から情報を得る環境知覚器と環境にはたらきかける行為実行器を持つ (図 1)。

2.2.1 心的状態

BDI アーキテクチャは BDI 論理で表すことのできる 3 つの心的状態を状態として持つ。

信念 (B) は、そのエージェントが真であると信じている情報である。BDI アーキテクチャではこの信念を参照して熟考し、プランを選択する。願望 (D) はエージェントが達成したいと望む目標、意図 (I) は達成したいと望むプランである。

これらの心的状態はインタプリタによって更新される。

2.2.2 プランライブラリ

プランライブラリはエージェントの持つプランの集合である。プランは「ある行為を実行すれば、結果としてあることが起こる」というエージェントの信念であり、これらはエージェントの心

的状態の信念とは別に保持されている。プランは意図形成条件、実行前提条件、プラン本体などによって構成される。意図形成条件はそのプランが意図として追加される条件、実行前提条件は意図を開始する条件、プラン本体は行為列である。

2.2.3 イベントキュー

BDI エージェントは環境知覚器から入力を受けると、その環境からの知覚をイベントとしてイベントキューに取り込む。

2.2.4 インタプリタ

インタプリタは心的状態の更新と、環境に対してエージェントが実行する行動を決定する。インタプリタは以下を繰り返し、エージェントの願望を達成しようとする。

1. プラン選択 プランライブラリから願望を達成するプランを選択し、意図とする。

2. 意図選択 次に実行すべき意図を選択する。

3. 行為選択 プラン内容を参照し、実行する行為を決定する。

4. 行為実行 決定された行為を実行する。

5. イベントの取り込み イベントキューからイベントを取り込む。

6. 心的状態の更新 成功した意図の削除など、心的状態を更新する。

3 BDI によるマルチエージェントの基本設計

マルチエージェントシステムにおいては、単体のエージェントでは達成できないことに対して、複数エージェントが共同作業で達成を試みることができなければ良いパフォーマンスは得られない。そのため、マルチエージェントシステムを構成するエージェントには他のエージェントと協調することが求められる。BDI エージェントに共同作業を実現させるためには、エージェント間の通信の実装と他のエージェントの心的状態の保持が必要となる。

ここではまず *LORA* について、および *LORA* による共同作業に関する記述について述べ、これに基づいて我々が提案するエージェントの基本設計について述べる。

3.1 BDI 論理の拡張

BDI エージェントが協調して行為決定を行うためには、そのエージェント自身だけでなく、システム内の他のエージェントの信念・願望・意図をといった心的状態を自身の心的状態として持つ必要がある。こうした複数エージェントの相互心的状態を扱えるように BDI 論理を拡張したものとして *LORA*[8] がある。

LORA は BDI 論理に更にオペレータを追加した論理体系である。オペレータの追加の他、エージェントやエージェントの集合を要素として表すことが可能となった。そのため、BDI 論理では書き表せなかった自分以外のエージェントの心的状態を持つことが可能となる。例えば、「エージェ

ント i が φ を信じている」という心的状態を $\mathcal{LORA}$ では、 Bel(i φ)[1]と書くことができる。ただし BDI 論理の BEL, DESIRE, INTEND などのオペレータに代わり、$\mathcal{LORA}$ では Bel, Des, Int を使用する。$\mathcal{LORA}$ では新たにオペレータ "$\in$", "Agts" が定義として追加される。"$i \in g$" はエージェント i がエージェントの集合 g に属することを表し、Agts(α g) は g が行為 α に必要であることを表す。

また補助的な定義として、E-Bel(g φ)(g はエージェントの集合), E-Des(g φ), E-Int(g φ) といった略記が追加される。E-Bel(g φ) は $\forall i \cdot (i \in g) \Rightarrow$Bel($i$ φ) の略記であり、E-Des, E-Int についても同様である。

この他に、$\mathcal{LORA}$ は相互心的状態を表す M-Bel, M-Des, M-Int といったオペレータを持つ。例えば、エージェントの集合 g が φ としう相互信念を持っていることを、M-Bel(g φ) と書く。これは前述の E-Bel を用いて、以下のように書ける。

$$
\begin{aligned}
\text{M-Bel}(g\ \varphi) \quad := \quad & \forall i \cdot (i \in g) \Rightarrow \\
& \text{Bel}(i\ \varphi) \wedge \\
& \text{Bel}(i\ \text{E-Bel}(g\ \varphi)) \wedge \\
& \text{Bel}(i\ \text{E-Bel}(g\ \text{E-Bel}(g\ \varphi))) \wedge \\
& \cdots
\end{aligned}
$$

M-Des, M-Int についても同様に書くことができる。

3.2 で述べる共同作業を記述するには、相互心的状態の概念が必要である。これは、エージェントの集合が共同での信念が成立したことを信じるには、信念の無限のネストを要する場合があるからである。例えば、エージェント a とエージェント b が、それぞれ相手のエージェントと「地点 X で合流する」と信じているとする。ここで a が待ち合わせ場所を地点 Y に変更したいとしたとき、a は b へ「地点 Y で合流する」と変更することをメッセージとして伝えたとする。しかし、a は実際に b が「地点 Y で合流する」ことを信念として追加したかどうかは把握することはできないため、待ち合わせ場所が変更されたという信念を持つことができない。一方、b は「地点 Y で合流する」というメッセージを受けた時点では、上記の a と同様の理由で待ち合わせ場所の変更の信念を持たない。そこで b は a に「地点 Y で合流することを承知した」という、「承知」のメッセージを送る。それでも b は、a が「承知」のメッセージを受け取り損ねる可能性があるという理由で、まだ待ち合わせ場所が変更されたという信念を持つことができない。同様のやりとりが a と b の間で繰り返されたとしても、それが行なわれるのが有限回である限り、双方で待ち合わせ場所の変更の信念は成り立たない。そのため、a と b がこの信念を持つためには「相手のエージェントが待ち合わせの変更を承知した」という信念の無限のネストが要求される。

3.2 共同作業

ここでは、[8] で述べられている共同作業のメカニズムと、その条件となる $\mathcal{LORA}$ による論理式の記述を示す。エージェントが協調し、共同作業を行なうまでの過程は以下の通りである。

1. 発生 : 願望とエージェントの能力の認識から共同作業の試みが発生
2. チーム形成 : 共同作業に携わるエージェントの集合を作成
3. プラン形成 : 願望を達成するための行為列を決定

[1]LORA では (Bel i φ) と記述するが、本論文では他文献に揃えて Bel を前置オペレータとし Bel(i φ) と記述する。他のオペレータについても同様。

4. チーム行動 : 形成されたプランに沿った行為の実行

上記 (1) では、システム内のエージェント i が φ に対して、

- i が φ を願望している
- i が現在 φ が真でないと信じている
- i が φ を達成可能なエージェントの集合 g が存在すると信じている
- i が φ を達成することができないか、i が φ を自力で達成することを望まない

とした場合に共同作業の可能性を認識し、それを試みる。この「エージェント i が φ を共同で達成できる可能性を認識する」ことをオペレータ PfC を使用して PfC(i φ) と表す。

$$
\begin{aligned}
\text{PfC}(i\ \varphi) \quad := \quad & \text{Des}(i\ \varphi) \wedge \\
& \text{Bel}(i\ \neg\varphi) \wedge \\
& \exists g \cdot \text{Bel}(i\ \text{J-Can}(g\ \varphi)) \wedge \\
& \left[\begin{array}{l} \text{Can't}(i\ \varphi) \vee \\ \text{Bel}(i\ \forall\alpha \cdot \text{Agt}(\alpha\ i) \wedge \text{Achvs}(\alpha\ \varphi) \\ \quad \Rightarrow \text{Int}(i\ \text{Doesn't}(\alpha))) \end{array}\right]
\end{aligned}
$$

J-Can(g φ) はエージェントの集合 g が φ を達成可能であることを示すオペレータである。

$$
\begin{aligned}
\text{J-Can}^0(g\ \varphi) :=& \exists\alpha \cdot \text{M-Bel}(g\ \text{Agts}(\alpha\ g) \wedge \text{Achvs}(\alpha\ \varphi)) \\
& \wedge \text{Agts}(\alpha\ g) \wedge \text{Achvs}(\alpha\ \varphi) \\
\text{J-Can}^k(g\ \varphi) :=& \text{J-Can}^{k-1}(g\ \text{J-Can}^0(g\ \varphi)) \quad \text{for } k > 0 \\
\text{J-Can}(g\ \varphi) :=& \bigvee_{k \geq 0} \text{J-Can}^k(g\ \varphi)
\end{aligned}
$$

Can't(i φ) は i が φ を達成できないこと、Achvs(α φ) は行為 α によって φ が達成されること、Agt(α i) は i が α を実行できる唯一のエージェントであること、Doesn't(α) は α が起こらないことを表す。

(2) では、共同作業を試みたエージェント i が他のエージェントに呼びかけ、φ を達成するためのチームを形成を試みる。i がエージェントの集合 g にチーム形成の呼びかけを行なうには、g が φ を達成できるという相互信念と、g が φ を達成するという相互意図を持つことが前提条件となる。この前提条件は、PreTeam(g φ) と書き、以下のように定義される。

$$
\begin{aligned}
\text{PreTeam}(g\ \varphi) \quad := \quad & \text{M-Bel}(g\ \text{J-Can}(g\ \varphi)) \wedge \\
& \text{M-Int}(g\ \varphi)
\end{aligned}
$$

i がこの条件を満たしているならば、i はチーム形成の呼びかけを行う。チーム形成の呼びかけは {FormTeam i g α φ} と書く。この呼びかけは、i が g のエージェントに対して「g が φ を達成可能である」と知らせ、g に「φ を達成する」ことを要求することで行う。チーム形成は以下のように定義される。

$$
\begin{aligned}
\{\text{FormTeam}\ i\ g\ \alpha\ \varphi\} \quad := \quad & \{\text{Inform}\ i\ g\ \alpha\ \text{J-Can}(g\ \varphi)\}; \\
& \{\text{Request}\ i\ g\ \alpha\ \varphi\}
\end{aligned}
$$

{} で囲まれたものを発話行為と呼ぶ。Inform は通知、Request は要求の発話行為である。";" は連続しての実行を表す。この呼びかけにより、φ を達成するためのチームが形成される。

(3) では、形成されたチームが φ を達成するための行為列を決定する。行為列の決定は、主にチーム内のエージェントの話し合いによって行なわれる。

(4) では、(3) で決定したプランに沿って g の各エージェントが行動する。この過程は常に成功するとは限らず、失敗した場合、また願望の達成が困難となった場合に、チームの各エージェントが意図を放棄しなければならない。

3.3 基本設計

3.2 節で述べた $\mathcal{LORA}$ の論理式による共同作業の記述に基づき、共同作業を行なうエージェントの基本設計を与える。

3.3.1 $\mathcal{LORA}$ からの変更点

3.2 節では相互心的状態の定義として 3.1 節で述べたように無限のネストの心的状態を要求しているが、実際のシステムではその検査は難しい。現実的な問題を扱う場合、通信路が著しく信頼できないなど特別な場合を除き信念の無限のネストも必要ないと考えられる。そのため実装にあたり、$\mathcal{LORA}$ の心的状態の定義を変更した。3.1 節の相互信念を示すオペレータ M-Bel の定義を以下のように変更する。

$$\text{M-Bel}(g\ \varphi) := \forall i \cdot (i \in g) \Rightarrow \text{Bel}(i\ \varphi) \wedge \text{Bel}(i\ \text{E-Bel}(g\ \varphi))$$

M-Des、M-Int の定義も同様に変更する。この定義であれば、後で述べる Jason などでオペレータの定義が可能となる。

3.3.2 論理型言語による記述

共同作業を行なうアルゴリズムは、発生 (recognition), チーム形成 (team_formation), プラン形成 (plan_formation), チーム行動 (team_action) の繰り返しである。論理ベースのプログラムでは、それぞれ以下のように表せる。

```
recognition(Desire) :-
        desire(Agent, Desire),
        belief(Agent, not(Desire)),
        belief(Agent, j-can(Group, Desire)),
        (can't(Agent, Desire);
        belief(Agent, not(agt(Actions, Agent),
                achvs(Actions, Desire),
                intend(Agent, does(Actions))))
        ).
team_formation(Desire) :-
        pre_team(Desire, Group),
        form_team(Agent, Group, Actions, Desire).
```

```
plan_formation(Group, Desire, Actions) :-
        negotiation(Group, Desire, Actions).
team_action(Actions) :- execute(Actions).
```

recognition(Desire) は 3.2 節の発生、team_formation(Desire) は 3.2 節のチーム形成の論理式をプログラムに対応させたものである。

plan_formation(Group, Desire, Actions) は 3.2 節のプラン形成にあたり、negotiation(Group, Desire, Actions) は、共同作業を行うエージェントの集合 Group 内のエージェントで願望 Desire を達成するための行為列を話し合って決めたものが Actions であることを表す。negotiation の定義は、扱う問題によって内容が異なるのでここでは説明を省く。

execute(Actions) は、plan_formation によって得た行為列 Actions を実行するものである。しかし、集合 Group 内のエージェントの実際の行動はそれぞれ異なっている場合がある。このような場合の個々のエージェントの行動は、チーム形成の過程で決定される場合と、プラン形成以降の過程で決定される場合が考えられる。前者はチームを形成するとき、特定のエージェントに何かさせることによって Desire が達成できるとわかっている場合にエージェントの果たす役割が決定される。したがってエージェントはこの役割に従った行動をとる。後者は、集合 Group で Desire が達成できるとしながら、エージェントの役割が決定されていない場合に起こる。このとき、negotiation においてエージェントの役割も含めたプランについての話し合いが行なわれる。

3.4 意図の維持

BDI エージェントは意図の維持により、目的を達成に向けた一貫的行動を取る。この意図の維持について、マルチエージェントシステムのエージェント間では意図の持続 · 放棄の条件は統一されていなければならない。つまり、全てのエージェントが同じ戦略に基づいて意図を持続しなければならない。

エージェントの意図は、その意図が達成されたタイミングで破棄される。またエージェントの題材によっては、意図の達成が困難となった場合などにも、意図が破棄されることが望ましい場合もある。以下は Rao らが提案した意図の持続に関する戦略である [5]。

- Blind: 意図がすでに実現されていると信じるまで持続
- Single-minded: 意図がすでに実現されていると信じるか、意図の実現が可能であると信じなくなるまで持続
- Open-minded: 意図がすでに実現されたと信じるか、意図を形成した願望が取り下げられるまで持続

しかし、例えば、2 つのエージェントが共通の意図に沿って行動していたとき、片方のエージェントが意図の実現が不可能であると認識して意図を放棄して新しい意図を作成しようとしたのに対し、もう一方のエージェントが Blind 戦略に基づいて意図を継続していたのでは協調的な行動をとっているとは言えない。このような意図の同期の失敗は同じ戦略に基づいていても起こる可能性がある。例えば、全てのエージェントが Single-minded に基づいている、宝探しを行なうマルチエージェントシステムがあるとする。現在、すべてのエージェントは「宝を手に入れる」という願望を持ち、それを達成するための意図を持っている。ここで、あるエージェントが宝を発見してこの意図を実現したとする。そのエージェントは意図の実現によってこの意図を破棄するが、他の

エージェントは意図が実現されたと信じていないため、この意図を持続してしまう。共通の意図の維持を同期させるには、共通の戦略に基づいて意図を持続するとともに、意図の持続性が失われる条件を認識したとき、それを共同作業を行なう他のエージェントに伝えることが必要となる。

意図の維持は 3.3.2 節の設計には盛り込まれていない。これは扱う問題ごとに、適する戦略に沿った意図の維持を実現される。Single-minded については、後述のオブジェクト収集エージェントの例で述べる。

4 実装

本章では 3.3.2 節で述べた基本設計に基づくエージェントの実装方針について示し、その有効性を示すため、それに基づいたマルチエージェントシステムの実装例について述べる。

本研究では BDI マルチエージェントシステムのテストベッドとして、3 次元マップでのオブジェクト収集エージェントの事例を選択した。また本章では、この基本設計によって反射的行動を必要とするシステムにも対応させることが可能であることを示すため、RoboCup サッカーの事例についても述べる。システムを構築するエージェントの実装には AgentSpeak を使用し、そのインタプリタに Jason ををを用いる。

4.1 AgentSpeak

AgentSpeak[4] はエージェント指向の論理型言語である。AgentSpeak は BDI モデルの概念に基づいており、エージェントの心的状態、イベント、行為列を一階述語論理式で記述する。ただし、心的状態の願望 D については、代替としてゴール G を使用する。AgentSpeak で書かれたエージェントプログラムは後述の Jason インタプリタによって実行することが可能である。

4.1.1 プログラム例

AgentSpeak によるプログラムは、エージェントの持つ信念とプランから構成される。

エージェントの信念は内容を表すオペレータとパラメータから構成される述語で表記される。例えば、「A は B の父親である」という信念を、オペレータ father とパラメータ A,B を用いて father(A,B) と定義したとする。すると「taro は ichiro の父親である」という信念は述語 father(taro, ichiro) と表すことができる。また、ある信念を持っている、ということが成立するための定義をルールとして記述することができる。「X と Y は兄弟である」という信念を持つことの条件を、信念「Z は X の父親である」を持ち、かつ信念「Z は Y の父親である」を持っている、とした場合を図 2 のように記述できる。

```
father(taro, ichiro).
father(taro, jiro).
brother(X, Y) :- father(Z, X) & father(Z, Y).
```

図 2: AgentSpeak プログラム:信念とルール

また、ゴールについても述語で表す。プログラム中でゴールを表す場合、述語の前に “!” をつける。

AgentSpeak のプランは「Trigger : Context <- Body.」のように記述する。プランの頭部にあたる Trigger はこのプランが呼び出されるためのトリガイベントである。トリガイベントの種類には、信念またはゴールの発生と、それらの取り下げがある。発生のイベントならば、信念やゴールを表す述語の前に “+” を、取り下げのイベントならば “–” をつける。例えば、信念 belief(B) の発生がトリガイベントならば+belief(B)、ゴール goal(G) の取り下げならば-!goal(G) となる。Context はこのプランが意図として選択されるための前提条件が記述される。Body はプラン本体である。プラン本体はアクションとサブゴールで構成される。プランの例を図 3 に示す。

```
+!go(X) : present_place(X)
          <- true.
+!go(X) : present_place(Y) & X>Y
          <- advance; !go(X).
+!go(X) : present_place(Y) & X<Y
          <- retreat; !go(X).
```

図 3: AgentSpeak プログラム:プラン

図 3 はゴール go(X) の発生をトリガイベントとするプランである。変数 X には整数が代入される。!go(X) は「地点 X へ行く」というゴール、present_place(X) は「地点 X にいる」という信念である。信念 present_place(X) をすでに持っている場合は、プラン本体が真であることを示す true のみである一番上のプランが選択される。エージェントの現在いる地点 Y が X と異なる場合、現在の地点が X に近づくようなプランが選択される。advance は前進するアクションで、present_place(X) の X の値を 1 増加させる。retreat は後退するアクションで、present_place(X) の X の値が 1 減少する。前進、または後退のアクションを実行した後、再度ゴール go(X) を発生し、地点 X を目指す。

4.2 Jason

Jason[1] はマルチエージェントシステム上のエージェント同士の通信を可能とした AgentSpeak のインタプリタである。Jason はエージェントに特定の動作を実行させる内部コマンドをもっており、デフォルトの内部コマンドにはエージェント間の相互通信を実装するものもある。通信を行なう内部コマンドを使用することにより、特定のエージェント、またシステム内のすべてのエージェントに心的状態の情報を伝えあって共通の意図の形成に役立てることが可能である。また通信で他のエージェントに指定したゴールの達成を要求することもできる。Jason の内部コマンドには通信を行なうものの他に、心的状態の参照 · 変更、リスト · 文字列の操作、条件分岐を行なうものも用意されている。

Jason ではシステムの環境や内部コマンドなどはすべて Java プログラムで作成する。また、Java プログラムによるエージェントの心的状態の更新手順やエージェントアーキテクチャのカスタマイズが可能である。例えば、イベント、オプション、意図の選択関数はクラス Agent 内の関数 selectEvent,selectOption,selectIntention にあたる。デフォルトの関数では、イベントと意図はキューの先頭のものが、オプションは条件に合うものの中からプログラムコードの一番上に記述されているものが選択される。これらの関数をオーバーライドすることで、それらの選択方法を変更するこ

とができる。例えば意図の実行に優先順位をつけたり、特定のイベントを常に最初に選択するようにできる。どのような選択関数が適正であるかは、扱う問題によって異なる。

4.3 共同作業

図 4 は 3.3.2 節の設計に沿った、一般的な場合の AgentSpeak によるエージェントプログラムである。ゴール Goal を達成するためにエージェントの集合 Group がチームとして形成され、そのチームが共通の意図を形成し、その共通の意図に沿った行為列が実行される。

```
%recognition
+!recognition(Goal) : not(Goal) & j_can(Group, Goal)
        <- !cannot(Goal); +pfc(Goal).
%team_formation
+!team_formation(Goal) : pfc(Goal)
        <- !preTeam(Group, Goal);
           !formTeam(Group, Goal);
           +group(Goal, Group).
%plan_formation
+!plan_formation(Group, Goal)
        <- .send(Group, achieve, negotiation(Group, Goal));
           !negotiation(Group, Goal).
%team_action
+!team_action(Actions) <- !execute(Actions).

+!preTeam(Group, Goal) : m_bel(Group, j_can(Group, Goal)) & m_int(Group, Goal).
+!formTeam(Group, Goal)
        <- .send(Group, tell, j_can(Group, Goal));
           .send(Group, tell, request(Goal)).
+!execute([]).
+!execute([Action|RestList]) <- Action; !execute(RestList).
```

図 4: 共同作業の実装 (a)

3.3.2 節で示した共同作業の各過程が達成すべきゴールとして発生した時に、これらのプログラムが呼び出される。recognition(Goal) はゴール Goal を共同作業で達成できるかどうか条件をチェックする。条件を満たしている場合は「Goal を共同作業で達成できる可能性がある」という信念 pfc(Goal) が追加される。

team_formation(Goal) では 3.3.2 節の設計にある pre_team と form_team をそれぞれサブゴール preTeam,formTeam とする。チームが形成されると、信念 group(Goal, Group) が追加される。group(Goal, Group) は「Goal を達成するためのチームが Group である」という信念である。

plan_formation(Group, Goal) で行なうプラン形成のための話し合いをサブゴール negotiation(Group, Goal) とする。negotiation(Group, Goal) を達成するためのプランは事例に合わせたプログラムとなる。.send は Jason の内部アクションで、他のエージェントに信念の追加やゴールの達成を要求す

るものである。.send(Group, achieve, negotiation(Group, Goal)) は Group 内のエージェントすべてにゴール negotiation(Group, Goal) の達成を要求する。

team_action(Actions) はリスト形式で渡される行為列 Actions をサブゴール execute(Actions) にで実行する。execute(Actions) ではリスト中の先頭のアクション Action を実行し、Action を除いた行為列について、再度サブゴール execute を発生させる。このような再帰処理を行為列が空リストになるまで行ない、行為列 Actions のすべてのアクションを実行する。

図 4 のプログラムで使用される achvs などの $\mathcal{LORA}$ のオペレータは図 5 に示す通り、ルールによってその定義を記述しておく。

```
j_can(Group, Goal) :- achvs(ActionList, Goal) & agts(ActionList, Group).
achvs(ActionList, Goal) :- .concat("+!", Goal, Trigger) & .relevant_plans(Trigger, ActionList)
        & .length(ActionList, Size) & Size > 0.
agts(ActionList, Group) :- .length(ActionList, N) & agts_check(ActionList, N).
m_bel(Group, Term) :- Term[Group] & e_bel(Group, Term).
e_bel(Group, Term) :- Term[Group].
```

図 5: 共同作業の実装 (b)

図 4 の共同作業の一連の流れを行なうのが図 6 である。

```
+!cooperation_check(Goal)
        <- !recognition(Goal);
           !team_formation(Goal);
           ?group(Goal,Group);
           !plan_formation(Group,Goal);
           ?action(Goal, Actions);
           !team_action(Actions).

-!cooperation_check(Goal)
        <- -pfc(Goal); -group(Goal, Group);
           !!Goal.
```

図 6: 共同作業の実装 (c)

共同作業の可能性をチェックしたい Goal を達成すべきサブゴールとするときに、!Goal ではなく!cooperation_check(Goal) として発生させる。cooperation_check(Goal) を達成するプランの途中で失敗した場合、ゴール cooperation_check(Goal) が取り下げられる。この取り下げのイベント -cooperation_check(Goal) から引き起こされるプランで共同作業の過程で追加された信念の取り下げを行ない、改めて達成すべきゴールとして Goal を発生させる。

この他にも、3.4 節で述べた意図の維持の戦略の実現が求められる。戦略ごとに、実装方法が異なり、問題によって適切と思われるものを使用する。

4.4 オブジェクト収集エージェント

3.3.2 節の基本設計に基づいて協調を行なうマルチエージェントシステムを作成した。エージェントは図 7 のような X,Y,Z 軸からなる 3 次元座標上のオブジェクト (object) を集めることを目的とする。エージェントプログラムは AgentSpeak で記述する。

エージェントが動作する 3 次元座標のマップは、以下の規則を持つ環境である。

- エージェントは定められた視界に入ったものの座標を得る
- エージェントは同一座標に存在するオブジェクトを取得することができる
- エージェントは X 軸、Y 軸方向に 1 だけ移動することができる

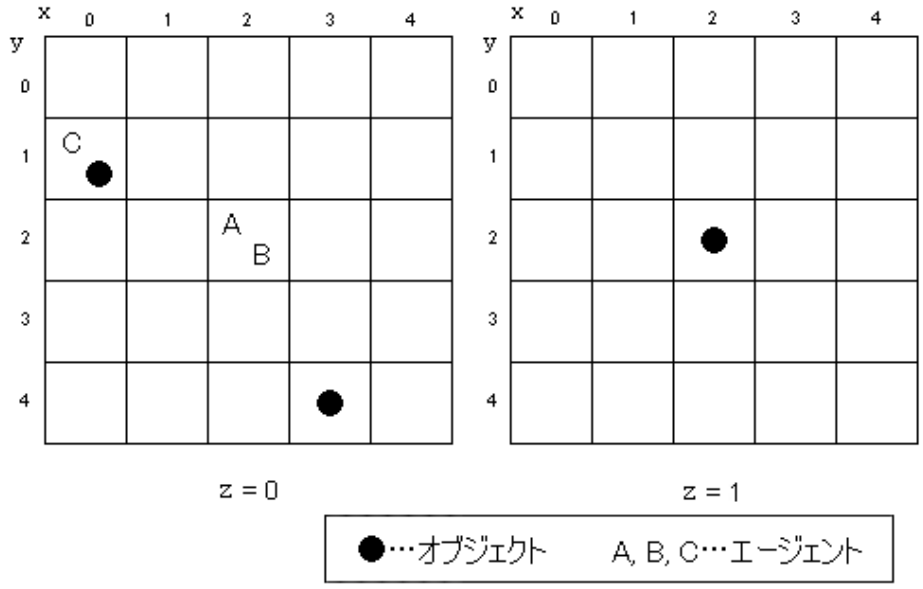

図 7: オブジェクト収集エージェント

例えば図 7 では、座標 (0,1,0) にいるエージェント C は同一座標にあるオブジェクトを取得することが可能である。

4.4.1 エージェントプログラム

本節では共同作業を行わないエージェントプログラムを示す。

オブジェクト収集を行なうエージェントは以下の過程でオブジェクトの取得を試みる。

1. オブジェクトの探索
2. オブジェクトのある座標まで移動
3. オブジェクトを取得
4. 1 に戻る

```
+!search(object) : remainder_obj(0).
+!search(object) : remainder_obj(N) & N>0 & locate(object,X,Y,Z)
        <- !aim.
+!search(object) : remainder_obj(N) & N>0 & not locate(object,_,_,_)
        <- move; !search(object).
+!aim: locate(object,_,_,_) & at(X,Y,Z)
        <- hunters.aim(List,[X,Y,Z],[X1,Y1,Z1]);
            !take(object(X1,Y1,Z1)).
+!take(object(X,Y,Z)) : at(X,Y,Z)
        <- take(object).
+!take(object(X,Y,Z)) : not at(X,Y,Z)
        <- go_to(X,Y,Z); take(object).
```

図 8: エージェントプログラム

この動作を行なうプログラムは次のように記述する。

search(Target) は変数 Target に代入されたものを見つけるというゴールである。ゴール search(object) を達成することで、オブジェクトを発見する。図 8 の上から 3 番目までの節は、ゴール search(object) を達成するためのプランである。remainder_obj(N) は、マップ上にある残りのオブジェクト数が N 個であるという信念である。残りオブジェクトが 0 である場合、search(object) は無条件に達成され、エージェントはその動作を終了する。locate(Name,X,Y,Z) は Name が座標 (X,Y,Z) にあるという信念である。変数 Name には object または player が代入される。例えば locate(object,0,3,1) は、座標 (0,3,1) にオブジェクトが存在するという信念である。エージェントが locate(object,X,Y,Z) に単一化可能な信念を一つ以上持っている場合、aim を達成すべきゴールとして発生する。エージェントが、このオブジェクトの座標に関する信念を持っていない場合は行為 move を実行し、ゴール search(object) を発生する。行為 move はマップ上を移動する行為である。移動することで視界に入るマップの座標を変化させ、オブジェクトの探索を行っていく。

aim は取得すべきオブジェクトの狙いをつけるためのゴールである。aim を達成するプランにある hunter.aim(List,[X,Y,Z],[X1,Y1,Z1]) は hunter ライブラリの内部アクションである。変数の List にはエージェントの持つオブジェクト座標に関する信念のリストが、X,Y,Z にはエージェントの現在の座標の X 値、Y 値、Z 値が代入される。内部アクション aim はこれらの渡された値を参照し、取得すべきオブジェクトの座標を X1,Y1,Z1 で返す。オブジェクトに狙いをつけたら、ゴール take(object(X1,Y1,Z1)) を発生する。

take(Target) は変数 Target に代入されたものを取得するゴールである。take(object(X,Y,Z)) は座標 (X,Y,Z) のオブジェクトの取得を目指す。エージェントが座標 (X,Y,Z) にいる場合、アクション take を実行する。アクション take(Target) はマップ上にある Target を取得する行為である。エージェントがオブジェクトと同一座標に存在していない場合、go_to(X,Y,Z) でオブジェクトのある座標まで移動する。go_to は X,Y,Z で指定された座標まで移動するアクションである。オブジェクトの座標までの移動が完了したら、take(object) でオブジェクトを取得する。

以上のようにして、図 8 のエージェントプログラムではオブジェクトの取得を行なっている。

4.4.2 エージェントの共同作業

4.4.1 節ではエージェントが単独でオブジェクトを取得する場合のエージェントプログラムを示した。本節では、4.3 節のプログラムを使用し、エージェント同士で協調してオブジェクトを取得するエージェントプログラムを示す。

エージェントの共同作業として、二つのエージェントが協調して、現在エージェントのいる座標から Z 軸方向に 1 高いところに存在するオブジェクトを取得することを可能とした。これは同一座標に二つのエージェントが存在している場合に、一つのエージェントがもう一つのエージェントの上に乗ることで実現する。例えば図 7 の状態では、座標 (2,2,0) にいるエージェント A,B が協力することにより、座標 (2,2,1) のオブジェクトを取得することができる。この「土台とそれに登るエージェントでオブジェクトを取得する」共同作業 (以降ではこれを共同作業 φ と呼ぶ) を行なうエージェントがオブジェクトを収集するプログラムは図 9 に示す通りである。

```
roles_action(rider,
      [go_to(X,Y,Z-1), wait(support, go_to(X,Y,Z-1)), ride, take_object(X,Y,Z), get_off])
roles_action(support, [go_to(X,Y,Z-1), wait(rider, get_off)])

+!search(object) : remainder_obj(0).
+!search(object) : remainder_obj(N) & N>0 & locate(object,X,Y,Z)
            <- !aim.
+!search(object) : remainder_obj(N) & N>0 & not locate(object,_,_,_)
            <- move; !search(object).
+!aim: locate(object,_,_,_) & at(X,Y,Z)
            <- hunters.aim(List,[X,Y,Z],[X1,Y1,Z1]);
                !cooperation_check(take(object(X1,Y1,Z1))).
+!take(object(X,Y,Z)) : at(X,Y,Z)
            <- take(object).
+!take(object(X,Y,Z)) : not at(X,Y,Z)
            <- go_to(X,Y,Z); take(object).

+!negotiation(Group, take(object(X,Y,Z)))[source(self)]
            <- !roles(leader); ?role(Role); ?roles_action(Role, Actions);
                +action(take(object(X,Y,Z)), Actions).
+!negotiation(Group, take(object(X,Y,Z)))[source(Leader)] : Leader\==self
            <- !roles(other); ?role(Role); ?roles_action(Role, Actions);
                +action(take(object(X,Y,Z)), Actions).
```

図 9: 共同作業エージェントプログラム

オブジェクトの探索から取得すべきオブジェクトの決定までは図 8 のプログラムと同じである。図 8 との相違は、ゴール aim を達成するプランの最後でゴール take(object(X1,Y1,Z1)) を発生するのではなくゴール cooperation_check(take(object(X1,Y1,Z1))) を発生させている点である。狙うべきオブジェクトを決定した後で、そのオブジェクトを取得するのに共同作業が必要かどうか、その可

能性を検査している。4.3 節の plan_formation のサブゴールである negotiation は、この事例では図 9 のように定義している。

negotiation を達成するプランは二つ存在する。Jason 上で動くエージェントは、その心的状態を示す述語には、図 8 のプランの頭部に示すように末尾にリストがつく。このリストには、リストの直前に示した心的状態を対象とするアノテーション[2]が格納される。これらのアノテーションはプログラムの操作の条件として参照することができる。図 8 にあるトリガイベント+!negotiation(Group, take(object(X,Y,Z))) の末尾にあるリスト中の source(Source) という述語は、対象を発生させたものが変数 Source であることを示すアノテーションである。この Source の値を参照し、negotiation の発生元によって異なるプランが選択される。

イベント+!negotiation(Group, take(object(X,Y,Z)))[source(self)] は、発生元がこのエージェントであるゴール negotiation(Group, take(object(X,Y,Z))) の発生である。ゴール negotiation を自ら発生させたエージェント、つまり plan_formation のサブゴールとして negotiation を達成しようとするエージェントの場合は、これをトリガイベントとしている上のプランが選択される。+!negotiation(Group, take(object(X,Y,Z)))[source(Leader)] をトリガイベントとしているプランは、前提条件の Leader\==self と合わせて、ゴール negotiation を発生させたのがこのエージェント自身ではない場合に選択される。つまり、このエージェントが.send(Group, achieve, negotiation(Group, Goal)) で他のエージェントから negotiation の達成を要求された場合に選択され、変数 Leader には.send を行ったエージェント名が入る。

この事例ではオブジェクトを共同作業で取得するプランは一つしかないものとし、negotiation では.send を実行したエージェントを Group のリーダーとした役割分担の話し合いを行なう。

リーダーとなったエージェントはどのエージェントが何の役割を担うかを決定し、それを Group の各エージェントに通知する。これの役割を決定するやりとりはゴール roles の達成によって行う。リーダーとなったエージェントはサブゴール roles(leader) で、自らを含めた共同作業の役割を Group 内の各エージェントに振り分ける。リーダー以外のエージェントは negotiation のサブゴール roles(other) でリーダーから役割が振り分けられるのを待ち、その役割を自分で達成可能か判断し、引き受けるか否かをリーダーに通知する。リーダーは Group のエージェントすべてから役割を引き受けると返事が来た時点で、その振り分け方をこの共同作業の役割分担にすると決定する。

役割の振り分けが終了したら、リーダーが Group の全てのエージェントに役割を表す信念 role(Role) を追加する。変数 Role には役割が代入され、この値によって、take(object(X,Y,Z)) を達成するための行為列が決定される。ここでは、エージェントの役割は rider と support の二つがあるとする。このシステム内のエージェントは役割 Role に対応する行為列が Actions であるという、信念 roles_action(Role, Actions) を持っている。この信念 roles_action を参照することにより、Group 内のエージェントは 4.3 節の team_action(Actions) で実行すべき行為列、Actions を得て、信念 action(take(object(X,Y,Z)), Action) を追加する。

team_action で実行する行為列は、信念 role(rider) を持つエージェントはリスト [go_to(X,Y,Z-1), wait(support, go_to(X,Y,Z-1)), ride, take_object(X,Y,Z), get_off]、信念 role(support) を持つエージェントはリスト [go_to(X,Y,Z-1), wait(rider, get_off)] である。リスト中のアクションはリストの順番通りに実行される。

役割 rider が実行する行為列は、始めにアクション go_to(X,Y,Z-1) で取得すべきオブジェクトの真下の座標まで移動する。wait(Role, Action) は役割 Role を担うエージェントがアクション Action の実行を終了するまで待機するアクションである。rider は support がアクション go_to(X,Y,Z-1) を

[2]Jason の構文要素としての Annotation を、本論文ではそのまま「アノテーション」と表記する。これはプログラム中で無視される「注釈」と区別するためである。

```
-object(X, Y, Z) : group(take(object(X,Y,Z)), Group);
                .send(Groop, untell, group(take(object(X,Y,Z)), Group);
                    -group(take(object(X,Y,Z)), Group).

-group(Goal, Group) <- !init(Goal).
```

図 10: Single-minded

終了するまで、つまり役割 support を持つエージェントが座標 (X,Y,Z-1) にいる状態になるまで待機する。ride は役割 support を持つエージェントの上に乗るアクション、get_off は現在乗っているものから降りるアクションである。ride の実行後は座標 (X,Y,Z) のオブジェクトを取得することが可能となるので、アクション take(object(X,Y,Z)) でオブジェクトを取得し、その後 get_off を実行し support から降りる。

役割 support を持つエージェントが実行する行為列は、オブジェクトの真下の座標まで移動と、rider がアクション get_off を終了するまでの待機のみである。

二つのエージェントがそれぞれの行為列を実行することで、ゴール take(object(X,Y,Z)) を達成する。二つのエージェントが実行する行為列はそれぞれ異なるが、take(object(X,Y,Z)) を共同作業 φ によって達成する、という共通の意図に基いている。

以上のように、3.3.2 節で与えた設計に基いた、オブジェクト収集問題のマルチエージェントシステムのエージェントの共同作業を実装した。

また、このオブジェクト収集問題では意図の維持の戦略として Single-minded を選択した。図 9 のプログラムの共同作業では、共同作業 φ によってオブジェクトを取得したとき、または他のエージェントによって目的のオブジェクトが取得されるなどして、目的のオブジェクトが消失した場合に、共通の意図を破棄する。共同作業 φ でオブジェクトを取得した場合は、ゴールを達成するための行為列が終了した時点でそれぞれのエージェントが意図を破棄する。目的のオブジェクトが無くなった場合は、そのことを認識したエージェントが他のエージェントに共同作業の中止を知らせる必要がある。図 10 は 3.4 節の Single-minded の例である。
座標 (X,Y,Z) にあるオブジェクトの消失のイベントは、信念 object(X, Y, Z) の取り下げである。このとき信念 group(take(object(X,Y,Z)), Group)、つまり座標 (X,Y,Z) のオブジェクトを共同作業で取得するチーム Group の一員であるという信念を持っている場合、Group 内のエージェントに共通の意図の取り下げを通知する。.send(Group, untell, group(take(object(X,Y,Z)))) で信念 group(take(object(X,Y,Z))) の取り下げを要求する。信念の取り下げのイベント-group(Goal, Group) で、ゴール Goal を達成するための共通の意図を取り下げるプランが呼び出されている。このプランのサブゴール init(Goal) はゴール Goal を達成する意図を取り下げる他、必要であれば意図を達成するための行為列の実行により変化した状態を通常の状態に戻す。

4.5 RoboCup サッカー 2D シミュレーションリーグ

扱う問題によっては、マルチエージェントシステムを設計する上で実装に制限がかけられる場合がある。また、エージェントが反射的行動を要するような応用もある。そのような場合においても、3.3.2 節で示した設計で対応できることを示すため、RoboCup サッカー [7] のプレイヤーエージェントを作成した。

RoboCup サッカーには複数のリーグが存在し、本研究では 2D シミュレーションリーグを選択した。

4.5.1 RoboCup サッカーにおける通信

2D シミュレーションリーグでは実機を使わず、サッカーサーバという仮想フィールドでプレイヤーエージェントが試合を行なう。サッカーサーバとプレイヤーエージェントの通信は UDP/IP ソケットを通じて行なう。ルール上の制約でプレイヤーエージェント同士が直接通信を行なうことは禁じられており、プレイヤー同士のコミュニケーションはサッカーサーバ上で行なわなければならない。そのため、図 4 の実装で使用している内部アクション.send を RoboCup サッカーで使用することができない。

サッカーサーバの通信では、(コマンド名 値1 値2 $\cdots$) というフォーマットのコマンド形式で情報が伝えられる。Jason 上で動作するエージェントがサッカーサーバからコマンド形式で送られてくる情報を読み取れるようにするため、またエージェントの行為をサッカーサーバのコマンド形式に変換して送信できるようにするため、内部アクションのライブラリ soccer を作成した。soccer ライブラリには、サッカーサーバのコマンドに対応する内部アクションが含まれる。

サッカーサーバ上でのプレイヤー同士の通信は一般的に say コマンドを使用して行なわれる。say コマンドは (say Message) というフォーマットで、変数 Message で指定した 10 バイト以内のメッセージを周囲に伝達するサッカーサーバのコマンドである。この say コマンドを使用して、.send の代替とする。

soccer ライブラリで say コマンドに対応する内部アクションが.say である。say コマンドは特定のプレイヤーに通信を行なうのではなく、メッセージを聞くことのできる範囲にいるプレイヤーすべてにメッセージが配信される。そのため.say は.send と異なり、メッセージの送信先のエージェントを明示的に指定できない。図 4 のプログラムで.send を使用している箇所を soccer ライブラリの内部アクション.say に置き換え、図 11 に示すように変更した。

4.5.2 プレイヤーエージェント

図 11 のプログラムを使用して、RoboCup サッカーのプレイヤーのエージェントプログラムを作成する。プレイヤーが反射的行動を行うことが求められるため、イベントに対して熟考を行なわずに即座に行動を起こす必要がある。RoboCup サッカーでは多くの反射的な共同作業を必要としており、その一部を図 12 に示す。

図 12 はボールをとったエージェントが、他のプレイヤーを協力して相手ゴールから得点を奪うためのプログラムである。ball_have は自分自身がボールをキープしているという信念である。信念 ball_have の発生の発生で図 12 の 1 行目のプランが呼び出され、ゴール cooperation_check(score) を発生する。

サッカーのような動的な環境下では、プラン形成の過程での話し合いに充分な時間を使えないことに注意しなくてはならない。ボールを取得したというイベントに対して複数のエージェントを即座に対応させるため、ゴール score を達成するための行為列はボールをキープしているプレイヤー、すなわち信念 ball_have を持つプレイヤーが他のプレイヤーに指示を出すことで決定する。指示を出すプレイヤーは negotiation のサブゴール command(score) の達成によって、他のプレイヤーに実行すべき行為列を渡す。プレイヤーへの指示は信念 command(Goal, PlanNumber, RoleNumber) の通

```
%recognition
+!recognition(Goal) : not(Goal) & j_can(Group, Goal)
        <- !cannot(Goal); +pfc(Goal).
%team_formation
+!team_formation(Goal) : pfc(Goal)
        <- !preTeam(Group, Goal);
           !formTeam(Group, Goal);
           +group(Goal, Group).
%plan_formation
+!plan_formation(Group, Goal)
        <- soccer.say(negotiation(Group, Goal));
           !negotiation(Group, Goal).
%team_action
+!team_action(Actions) <- !execute(Actions).

+!preTeam(Group, Goal) : m_bel(Group, j_can(Group, Goal)) & m_int(Group, Goal).
+!formTeam(Group, Goal)
        <- soccer.say(j_can(Group, Goal));
           soccer.say(check(Goal)).
+!execute([]).
+!execute([Action|RestList]) <- Action; !execute(RestList).
```

図 11: 図 4 の変更プログラム

達で行なう。変数 Goal にゴール名、PlanNumber には Goal を達成するプランのうち、実行すると決定した行為列に対応する整数、RoleNumber には役割を表す整数が代入される。指示を出すプレイヤーはまず score を達成する複数のプランの中から PlanNumber を決定するが、プレイヤーエージェントは既に score を反射的行動で達成するための PlanNumber を信念として持っている。この信念は別途強化学習などで獲得した戦略によって決定されるものとする。この信念を参照し、チーム内のエージェントに信念 command を送信する。PlanNumber と RoleNumber は熟考して決定されないため、短時間で行なわれる。そのため、ボールの取得に対してエージェントのチームが反射的に行動できる。

このように、4.4.2 節のオブジェクト収集エージェントと同じ設計で、実装上の都合による制限がある場合や、反射的行動を要するようなエージェントの構築も行なえることが示せた。

5 まとめ

LORA で与えられた形式的な記述に基き、論理型言語による BDI マルチエージェントシステムの基本設計を与え、この基本設定を使用したマルチエージェントシステムのエージェントプログラムを作成した。形式的記述と対応がとれているため、本論文で提案する設計では論理によって性質の形式的な議論が可能なエージェントを構築することができる。意図の維持の戦略は設計から独立しており、扱う問題ごとによってどのように共通の意図を保持するかを決定できる。本論文ではこの基本設計による、オブジェクト収集問題を扱うマルチエージェントシステムを示した。また、実

```
+ball_have <- !cooperation_check(score).

+!negotiation(Group score) : ball_have
        <- !command(score); ?command(score, PlanNumber, RoleNumber);
            ?plan_num(score(PlanNumber), Actions);
            +action(score, Actions).
+!negotiation(Group score) : not ball_have & position(forward)
        <- !wait(score); ?command(score, PlanNumber, RoleNumber);
            ?plan_num(score(PlanNumber), RoleNumber, Actions)
            +action(score, Actions).
```

図 12: プレイヤーエージェントプログラム

装に制限のある事例でも基本的に同じ設計のもとでマルチエージェントシステムを実装できることを、RoboCup サッカーの例で示した。

本論文で示した事例ではエージェント間の通信によって共通の意図の形成を実現している。しかし扱う問題によっては、通信が不可能な環境での協調を求められることも考えられる。今後は通信路が信頼できない、もしくは頻繁に通信を行なうことができない状況での他のエージェントの心的状態の推測を目指す。また、反射的行動でより良いパフォーマンスを実現するため、強化学習との融合も行なっていきたい。

参考文献

[1] Bordini, R. H. and Hübner, J. F.: *Jason: A Java-based interpreter for an extended version of AgentSpeak*. http://jason.sourceforge.net/Jason.pdf.

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

[3] Emerson, E. A. and Srinivasan, J.: Branching Time Temporal Logic, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency* (de Bakker, J., de Roever, W. and Rozenberg, G.(eds.)), Springer-Verlag, pp. 123–172 (1989).

[4] Rao, A. S.: AgentSpeak(L): BDI agents speak out in a logical computable language, *Proc. of MAAMAW-96*, LNAI, Vol. 1038, Springer-Verlag, pp. 42–55 (1996).

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

[6] Singh, M. P., Rao, A. S. and Georgeff, M. P.: Formal Methods in DAI: Logic-Based Representation and Reasoning, *Multiagent Systems*, The MIT Press, pp. 331–376 (1999).

[7] The RoboCup Federation: RoboCup. http://www.robocup.org/.

[8] Wooldridge, M.: *Reasoning about Rational Agents*, The MIT Press (2000).