自律的エージェントのための制約論理型言語RXFにおけるリフレクション機構の設計とその実装

大囿忠親† 新谷虎松†

本論文では，リフレクティブな制約論理型言語RXFにおけるリフレクションの実装と応用について述べる．本研究は，マルチエージェントシステムを容易に構築するための言語の実装を主眼とする．RXFにおいて柔軟なマルチエージェントシステムの実装のためにリフレクション機構を実現した．RXFのリフレクション機構は，手続き的リフレクションとオブジェクト指向言語におけるメタオブジェクトの実現技法に基づいて実装される．さらに，知識表現利用の視点から，メタな処理を実現するために自己参照機構のカスタマイズ機能を提供する．RXFは，自己参照機構のカスタマイズ機能によって，エージェントが，自分自身の構造について定義する機能を提供する．また，リフレクションの機能を用いることによって，複数のエージェントを組織化された1つのエージェントとして動作させることが可能である．最後に本研究の有用性と，問題点について議論し，今後の課題について述べる．

Implementing a Reflective Mechanism in Constraint Logic Programming Language RXF

TADACHIKA OZONO† and TORAMATSU SHINTANI†

In this paper, we describe a method for implementing a reflective mechanism in the reflective constraint logic programming language RXF. We have developed the language RXF for multiagent programming. The reflective mechanism is based on a procedural reflection and meta-object in object-oriented languages. An agent in RXF has an agent port. By using the port, the agent can refer the information of itself. The agent can control itself by the agent port. And more, it can customize itself by using the agent port for solving problems. An agent is called an united agent, which consists of some primitive agents. An united agent has highly abilities compared with that of a primitive agent. We show that RXF is an effective language for developing multiagent systems.

1. はじめに

近年，エージェント[1]が，計算機科学の分野において注目を集めている．計算機科学において，エージェントは，自律したソフトウェアとして扱われている．自律という観点から，エージェントは，エージェント自身の状態をエージェント自身で管理しながら，目的を遂行する必要がある．つまり，エージェントは，目的を遂行するために，(1) 目的を遂行するための計算，(2) (1) の計算を実行するための計算機資源の管理に関する計算，の2つの計算を行う必要がある．自律したエージェントの実現には，(2) の計算をエージェント自身で実行することが必要である．(2) の計算を行うために，エージェントは，エージェント自身の状態について知る能力が必要である．さらに，エージェントは，エージェント自身の状態を変更できる能力も必要である．前者を自己参照といい，後者を自己改変という[2]．

本研究の目的は，知的な自律したエージェントから構成されるマルチエージェントシステムを容易に構築可能にするリフレクション機構の実装である．本研究では，知的で自律した拡張性の高いエージェントを実現するための言語として，リフレクティブな制約論理型言語RXFを実装した[3]．RXFの主な特徴は，(a) 述語を用いた知識表現機能，(b) 複数エージェントの並行実行機能，(c) エージェントのエージェント自身に関する計算機能，の3点が特徴である．(a) は，論理を基礎とした自己参照形式[4]の利用が目的である．また，エージェントを構築するうえで，論理に基づく実装が

† 名古屋工業大学工学部知能情報システム学科
Department of Intelligence and Computer Science, Nagoya Institute of Technology

数多く研究され，その有用性が示されている[5]．エージェント内で線形方程式などの制約を扱う必要があることも指摘されている[6]．これらの要望から，論理と制約を同時に扱うことが可能な制約論理型言語[7]は，エージェントの効率的実装に対して強力な手段となる．(b) の実現のために，1台のワークステーション上で複数のエージェントを実装するためのマルチスレッド処理とネットワーク上に分散した複数のエージェントが通信するための機構を実現する．(c) の実現のために，近年ソフトウェア技術として注目を集めているリフレクション[2]を利用する．

人工知能研究におけるリフレクションの例[4),8)]として，知識表現の立場から，論理を基礎とした自己参照形式の研究がある．また，プロダクションシステムや定理証明システムにおいて，メタ規則という形での自己参照・改変が積極的に利用されている[2]．リフレクションの関連研究として，3-LISP[9]，ABCL/R2[10]，等がある．3-LISP は手続き的リフレクションの初めての実装である．ABCL/R2 はリフレクティブなオブジェクト指向言語である．

リフレクションは，手続き的リフレクション，オブジェクト指向言語におけるリフレクション，および，並行・分散システムにおけるリフレクションとして分類できる[2]．リフレクションを用いることによってシステムに拡張性と動的適応性を与えることが可能となる[2]．RXF は，これらのリフレクションを利用する機能を提供することにより，より柔軟なエージェントの実現を可能にする．リフレクティブなオブジェクト指向言語と RXF との相違点は，リフレクティブなオブジェクト指向言語がリフレクションを利用することによって，処理の効率化やプログラムのモジュール化を目指しているのに対し，RXF はメタな記述を可能にすることによって，エージェントが自己の機能を拡張する手段を提供することを目的とする．RXF では，エージェントが自己の機能を拡張する手段を提供するために，エージェントにポートと呼ばれる入出力機構を実現する．

RXF のリフレクション機能は，オブジェクト指向でいわれてきたリフレクションで実現されてきた機能に基づいている．しかし，エージェントは，プログラミングにおけるオブジェクトのような処理やデータの単位ではなく，あるタスクを遂行するための問題解決主体である．オブジェクト指向におけるリフレクションだけではなく，エージェントの様相管理のための枠組みの提供も必要である[11]．

本研究では，モデルに基づくエージェントを実装するのではなく，エージェントのモデルを設計したときにそのモデルを実装し，評価することも目的の一部であるので，エージェントの形式的なモデルを特に持たない．モデルは，論理で記述されることが多く，本研究では，制約論理型言語に基づいて実現・実装される．

本論文は，2章において RXF の特徴について説明し，3章で RXF におけるリフレクションについて論じる．4章で RXF におけるリフレクションの実装について説明する．5章では RXF におけるリフレクションの使用例を示す．6章では，終わりとして，まとめと今後の課題を述べる．

2. RXF の概略

2.1 特　　徴

リフレクティブな制約論理型言語 RXF[12]は，C++ を用いて実現され，効率的にマルチエージェントシステムを開発するという観点から，制約論理プログラミング，マルチスレッド処理，エージェント間通信，および，リフレクションのための機能を提供する．

マルチスレッド処理は，マルチエージェントの記述を可能にする．RXF によって記述されたエージェントを利用することによって複数プログラムの並行実行が可能である．RXF は並列論理型言語のようにプログラムを並列に動作させることによる処理の高速化を目指すものではなく，並行動作するプログラムの記述を可能にすることが目的である．マルチスレッドを用いた Prolog は，I.C. Prolog II[13]によって，その有用性が示されている．

RXF におけるリフレクションの機能はエージェントを述語で表現，操作することを可能とする．また，RXF のリフレクション機能は，複数のエージェントを組織化することにより，1つのエージェントのように振る舞わせる機能を提供する．

2.2 記 述 例

RXF は，CLP(R) に基づく構文を持つ．CLP(R)[14]は，制約解消の対象として線形制約問題を扱う制約論理型言語である．**図1** は，RXF のプログラム例である．p は，RXF の組込み述語 **receive** によってエージェント **AG** から6つの整数を受信し，制約 $AX+BY+C=0$ かつ $DX+EY+F=0$ かつ

```
p :-
  receive(AG,[A,B,C,D,E,F],[wait]),
  A*X+B*Y+C=0, D*X+E*Y+F=0, X>0, Y>0,
  maximize(X+Y),
  send(AG,[X,Y]).
```

図1　RXF プログラム
Fig. 1　Program in RXF.

$X > 0$ かつ $Y > 0$ を満たす変数 X, Y の組の中で $X+Y$ が最大になるものを RXF の組込み述語 send によってエージェント AG に送信するプログラムである．本例での receive は，オプション [wait] の指定により，メッセージを受信するまで処理をブロックする．図1の3行目が，本制約の RXF における記述である．4行目の maximize(X+Y) によって，$X+Y$ が，最大になるように制約解消系が制約を処理する．

2.3 エージェントとメタエージェント

RXF におけるエージェントは制約論理型言語のインタプリタとエージェント制御機構から構成される．インタプリタは，制約論理型言語のプログラムを実行する．本インタプリタは，線形制約問題のための制約解消系が含まれ，与えられた制約を解く．エージェント制御機構は，メモリの管理，ファイルやメッセージなどの入出力管理，イベント管理，そして割込み管理を行う．イベント管理は，イベントの発生と処理を管理する．割込み管理は，特定のイベントに対する割込み処理を管理する．

RXF において，エージェント A がエージェント B のインタプリタの実行環境を参照・変更してエージェント B のプログラムを評価・実行できるとき，A は B のメタエージェントと呼び，B は，メタエージェント A に対するベースエージェントと呼ぶ．このとき B の実行環境を参照・変更しながら実行された B のプログラムは A によってメタレベル実行されたという．

メタエージェントは，エージェントに対してイベントを発生させることができる．たとえば，RXF におけるメッセージ通信機構は，メタエージェントとして実装される．イベントは，メッセージの到着やインタプリタの終了などをエージェントのエージェント制御機構に知らせるための信号である．また，エージェントの機能をそのエージェントのメタエージェントが処理することもできる．

2.4 ポート

エージェントは，状態が動的に変化する．それに対して，論理型言語は，静的なプログラムとデータを用いて計算を行う．論理型言語でエージェントのような動的なソフトウェアを管理するには，特別な機構が必要であり，RXF では本機構をポートと呼ぶ．すなわち，ポートは，動的なエージェントを論理型言語上で扱うためのインタフェースである．RXF において，ポートは，エージェントの入出力機構も実現する．エージェントはポートを用いることによって，他のエージェントや，ファイルなどのエージェントにとっての外界に干渉できる．ポートは，ファイルや通信プロトコルなどのオペレーティングシステムに依存するような低レベルな部分を RXF の言語仕様にあわせるための機構を持つ．たとえば，ポートは，バイト列や C++ におけるクラスや構造体などの低レベルなデータを RXF の扱える述語形式に変換する機能を備える．すなわち，本ポートは，ファイルアクセス機能，エージェント間のメッセージ通信機能，そしてエージェントの内部データ変換機能を提供する．

エージェントは，デフォルトでは，メッセージ通信用，メタレベル表現生成用，およびユーザインタフェース用の3種類の入出力用ポートを持つ．メッセージ通信用のポートをメッセージポートと呼ぶ．メタレベル表現生成用のポートは，エージェントのメタレベル表現の生成と，メタレベル表現に基づくエージェントの状態変更を実現するためのポートである．メタレベル表現生成用のポートをエージェントポートと呼ぶ．ユーザインタフェース用ポートは，ユーザからの入力を得るためのポートである．その他のポートとして，ファイルポートがある．ファイルポートは，ファイルの読み書きに使われる．

3. RXF におけるリフレクション

3.1 機　　能

RXF におけるリフレクションの実装目的は，(1) エージェントに自分自身の状態を知る手段を提供する，(2) 言語のカスタマイズを可能にする，(3) 問題解決に関する計算と資源管理に関する計算を分離する，(4) エージェントのカスタマイズを可能にすることである．

(1) の自分自身の状態は，さらに2つに分けられる．1つは，問題解決中におけるエージェントの状態である．もう1つは，計算の実行主体としてのエージェントの状態で，たとえばメモリの使用状況やメッセージの到着状況などである．エージェントが自分自身の状態を知る機能を提供することによって，本エージェントの特徴である，自律性を実現する．

(2) の言語のカスタマイズは，RXF の制約論理型言語のインタプリタをカスタマイズする機能を提供する．たとえば，5.1 節で示すような実行制御を可能にする．さらに，AGENT 0[11] のように，プログラミング中でエージェントの様相を扱う言語を RXF 上に実装することも容易になる．

(3) の計算の分離は，プログラマが問題解決プログラムの中に明示的に資源管理プログラムを記述する必要性をなくし，問題解決に関する計算と資源管理に関する計算をそれぞれ独立なプログラムとして実行する環境を提供する．たとえば，5.2 節で示すようなメ

タエージェントを利用することにより，プログラマは，問題解決プログラムの開発に集中することができる．

(4) のエージェントのカスタマイズは，エージェントの機能を実現するソフトウェアをメタエージェントとして実現することにより実現する．たとえば，5.3節で示すように，メタエージェントをカスタマイズすることによってエージェントの機能をカスタマイズすることが可能になる．エージェントのカスタマイズ機能によって，計算機環境に適したエージェントの実装が可能になる．

本研究では，(1) を実現するためにエージェントポートを実装する．(2) を実現するためにメタレベル実行機能を実現する．(3) の実現は，メタレベル実行機能とエージェントポートに関係する．(4) を実現するためにエージェントの機能を実現するためのメタエージェントを導入する．以上の4つの機能によって，エージェントは，問題解決を行いながら，同時に計算機資源に関する計算を自分自身の状態に基づいて実行することが可能になる．また，問題や計算機環境に対して自分自身をカスタマイズすることによって処理の効率化や高機能化を実現することが可能になる．

3.2 メタレベル表現

3.1節で示した (1), (2), (3) のリフレクションに必要な機能として，メタレベル表現生成機能がある．メタレベル表現は，ベースレベルの状態をメタレベルで扱えるようにする．RXFにおいて，メタレベル表現の生成は，エージェントポートによって実現される．RXFにおける制約論理型言語インタプリタにおいて，論理型言語の性質から変数に対する破壊的代入をすることはできない．メタレベル実行時に，メタレベル表現を変数に束縛した場合，変数への破壊的代入を行うことができないという制限から，メタレベル表現が表すエージェントの状態を変更するために特別な手段を提供する必要がある．RXFでは，変数を介したメタレベル表現を扱わずに，ポートを用いてメタレベル表現を扱うことによってこの問題を解決した．メタエージェントにおいて，エージェントポートからあるデータを読み込むことが，そのデータに対応したベースエージェントのメタレベル表現の取得に相当する．さらに，エージェントポートへのメタレベル表現の書き込みは，それに対応するベースエージェントの状態を変更することに相当する．

エージェントのメタレベル表現をエージェントポートとして渡すことによってエージェントのカスタマイズと，エージェントのインタプリタの制御，つまりメタインタプリタとしての機能の実現の両方が可能になる．

3.3 リフレクション述語 reflect

組込み述語 reflect によって，明示的にメタレベル実行を開始させることが可能である．reflect には，1引数の reflect（reflect/1 と略す）と2引数の reflect（reflect/2 と略す）がある．reflect/1 は，第1引数をメタレベル実行する組込み述語である．このとき，メタエージェントが新たに生成されメタレベル実行を行う．基本的にメタエージェントは必要なときにだけ生成される．reflect/2 は，第1引数にメタエージェントを指定し，第2引数にメタレベル実行するゴールを指定する．以下に reflect/1 を用いた例と，reflect/2 を用いた例を示す．

```
reflect(ma,task)
reflect(task)
```

ここでは，task をメタレベル実行する例を示している．本例における，reflect/2 が評価されると，第1引数（ここでは ma）で示されたメタエージェントにおいて，第2引数（ここでは task）がメタレベル実行される．メタレベル実行が終了するとベースレベルでの実行が再開する．

4. リフレクション機構の実装

4.1 リフレクション機構の構成要素

本節では，リフレクション機構の実装について示す．RXFの構成図は，**図2** で示される．図2のC++は，RXFが，C++ 言語を用いて実装されていることを表す．C++ の上の“マルチスレッド”，“ポート”，“リフレクション”，そして“制約論理型言語インタプリタ”の各構成要素は，それらがC++ を用いて実装されたことを表す．特に，ここでの，リフレクション機構（図2の斜線部分）は，マルチスレッド機構とポート機構の機能を用いてC++で実現されていることを表している．図2の制約論理型言語インタプリタは，マルチスレッド，ポート，そしてリフレクション機構を用いて実装され，かつそれらをエージェントが利用するための機能を持つ．図2における灰色部分がエー

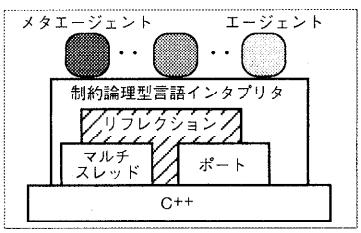

図2 RXFの構成図
Fig. 2 The structure of RXF.

ジェントとメタエージェントを表す．

RXF におけるリフレクション機構実現のために，(a) エージェントポート，(b) メタレベル実行機構，(c) メタエージェント，の3点の構成要素を実装する．エージェントポートはリフレクションにおける内省と causal connection [10] を実現する．causal connection は，ベースレベルのメタレベル表現に対する操作が，ベースレベルの状態に影響を及ぼすことを意味する．メタレベル実行機構は，プログラムの実行環境をベースレベル，メタレベルのように階層化する．メタエージェントは，エージェントの機能を実現するために用意される．

4.2 ポートディスクリプタ

RXF インタプリタは，プログラム中でポートを表現するためにポートディスクリプタを生成する．ポートディスクリプタは，ポートを操作する組込み述語への引数として利用される．RXF において，ポート操作用の組込み述語として，new_port，del_port，write，read の4つが利用可能である．new_port は2引数の述語であり（new_port/2 と略す），新規にポートを生成する組込み述語である．new_port/2 を評価することによって，第2引数で表現される特徴を持つポートが新規に生成される．新規に生成されたポートのポートディスクリプタは，new_port/2 の第1引数に束縛される．たとえば，new_port/2 は，次のように記述される．

```
new_port(PortDesc,file("fname",w))
```

ここで，第2引数である file("fname",w) のファンクタ名が，ファイルに対して入出力するポートであることを表す．本ファンクタの第1引数の "fname" が新規に生成されるポートが操作するファイル名を示し，第2引数の w が，本ポートが出力ポートであることを表す．本例を実際に評価することによって，ファイル名 fname のファイルを読み込むためのポートのポートディスクリプタが変数 PortDesc に束縛される．del_port は，1引数の組込み述語であり，第1引数に与えられたポートディスクリプタの示すポートを使用不能にする．

write は，2引数の組込み述語であり（write/2 と略す），出力用ポートに対して出力を実行させるための組込み述語である．write/2 の第1引数には，出力用ポートのポートディスクリプタを指定する．write/2 の第2引数には，出力するデータを指定する．read は，2引数の組込み述語であり（read/2 と略す），入力用ポートからデータを読み込む．第1引数には，入力用ポートのポートディスクリプタを指定する．第2引数は，実際に入力されたデータが束縛される．

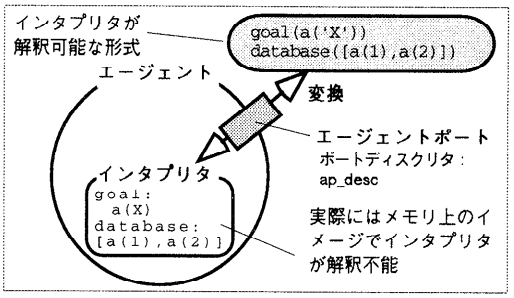

図3 メタレベル表現とエージェントポート
Fig. 3 An agent port for meta-level expressions.

4.3 エージェントポートの仕組み

エージェントポートは，計算機のメモリ上におけるエージェントの表現を，あらかじめ定義された方法に従って図2の制約論理型言語インタプリタ（以降，単にインタプリタと略す）の解釈可能な表現である述語に変換する機能を持つ．

エージェントポートの仕組みを，**図3** に示す．ここでは特にエージェントのインタプリタのメタレベル表現の取得に焦点をあてる．ここで，図3中のエージェントは，ゴール a(X) を評価中で，節データベースには，a(1) と a(2) の2つの事実が定義されているとする．このときのエージェントポートディスクリプタは ap_desc とする．エージェントが現在評価中の質問を得るためには，メタレベルで，以下のようにする．

```
:-read(ap_desc,goal(Y)).
```

組込み述語 read/2 によって第1引数として与えられたポート（ここでは ap_desc が示すエージェントポート）から，第2引数（ここでは goal(Y)）と単一化可能なデータと第2引数が単一化される．その結果変数 Y にベースエージェントに与えられた質問が a('X') としてメタレベルで束縛される．ここで得られた goal(a('X')) はエージェントポートによって述語に変換されたベースエージェントの質問のメタレベル表現に相当する．ベースレベルにおける変数 X は，メタレベルにおいてクォートされる．ここで特筆すべき点は，エージェントポートによって，モデル上は，メタサーキュラ [2] な定義をされたインタプリタ（すなわち，インタプリタと同じ言語で実現されたメタインタプリタ）を定義したインタプリタを実際にはメタサーキュラな定義をすることなく効率的に実装することが可能になることである．

4.4 メタレベル実行

組込み述語 reflect と組込み述語 call によってメタレベルにおける計算とベースレベルにおける計算の切

表1 reflect ベンチマーク
Table 1 Benchmark of reflect.

	1000	2000	3000	4000	5000
call	0.133	0.166	0.233	0.350	0.483
reflect	0.250	0.266	0.350	0.433	0.583
vcall	4.38	11.0	18.4	27.9	36.8

(単位は秒)

替えが実現される．メタレベル実行は，(a) メタレベル実行の検出，(b) メタレベル実行するゴールとベースエージェントのエージェントポート（具体的には，そのポートを表すポートディスクリプタ）をメタエージェントへ送信，(c) メタエージェントにおける評価・実行，の3ステップで構成される．ステップ (a) は，ベースレベルでメタレベル実行するように定義された述語（たとえば組込み述語）を検出する．ステップ (b) は，メタレベル表現を生成するために，メタレベル実行するゴールとベースエージェントのエージェントポートを送信する．メタエージェントはメッセージを受信し，メタレベル実行するための準備をする．メタレベル実行するための準備は，ベースレベルのメタ表現を生成するためのエージェントポートを利用可能にすることである．メタレベルにおける評価・実行はベースレベルにおける評価・実行と同様に行われる．

表1 に call 述語を用いたときの実行速度と，reflect 述語を用いたときの実行速度，そして vcall（バニラインタプリタ[15]）を用いたときの実行速度を示す．バニラインタプリタは，RXF 自身で定義された RXF インタプリタであり，reflect 述語と同様な機能を実現する．ベンチマークとしてリストの結合処理 (append) を行った．リストの長さが1000，2000，3000，4000 そして 5000 の場合をそれぞれ3回ずつ実行し，その平均を示す．実行速度の単位は秒である．本事例では，メタレベル実行がベースレベルの実行に比べてそれほど効率の悪いものではないことを示している．

5. リフレクションの記述例

RXF におけるマルチエージェントシステムの記述例は，文献12) で詳細に論じている．本章では，特にリフレクションの記述例に焦点をあて，具体的なマルチエージェントプログラム例を示す．

5.1 メタレベル実行制御例

ここで，reflect 述語の応用として LISP で用いられる catch & throw[16] の定義例を示す．LISP における catch & throw は，例外処理に用いられる関数である．

catch & throw を用いることによって標準的な Prolog において提供される機能であるカットとは異なる

・**catch & throw の定義**

```
% ベースレベルにおける定義
catch(X) :-
  reflect(tcall(X)).
% メタレベルにおける定義
tcall(Port,throw) :-
  call(Port).
tcall(Port,X) :-
  clause(Port,X,B),
  tcall(Port,B).
tcall(Port,X) :-
  is_constraint(Port,X),
  solve(Port,B).
tcall(Port,(C1,C2)) :-
  tcall(Port,C1),
  tcall(Port,C2).
tcall(Port,(C1;C2)) :-
  tcall(Port,C1)
  ;tcall(Port,C2).
```

・**catch & throw の使用例**

```
% ベースレベルで評価
catch_test:-catch(goal).
% メタレベルで評価
goal:-sub_goal, goal; throw.
sub_goal :- ...
```

図4 "catch & throw" の定義と使用例
Fig. 4 An example for realizing "catch & throw".

プログラムの実行制御を行うことが可能になる．カットを利用した実行制御は，呼び出しのレベルとして1レベルに限定した実行制御を可能にする．一方，catch & throw は，任意のレベルにおける実行制御を可能にする．メタレベル実行の例として**図4** に catch & throw のプログラム定義とその使用例の概略を示す．図4 において，tcall は，RXF における制約論理型言語インタプリタを実現する．catch はベースレベルで評価され，tcall をメタレベル実行させるプログラムである．tcall はメタレベルで評価され，catch の第1引数をメタレベルで評価する．たとえば，

```
:- catch_test.
```

がベースレベルで評価されると，catch によって，

```
tcall(ap_desc,goal)
```

がメタレベル実行される．ここで，すなわち，図4 に示した tcall の第1引数の変数 Port は，メタレベル実行時に，自動的に ap_desc に束縛される．ap_desc は，システムにより自動的に与えられる．ap_desc は，メタレベル実行時に生成されるベースエージェントのエージェントポートのポートディスクリプタ（4.2 節で示した）である．このとき throw が他プログラム中（本例では goal）に出現すると tcall(ap_desc,throw):-call(ap_desc) が評価され，ベースレベルに戻る．call(ap_desc) は，ap_desc の示すエージェントのインタプリタとして動作する．図4 の例で

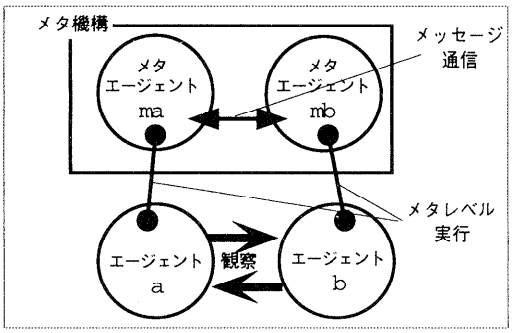

図5 RXF におけるリフレクションの例
Fig. 5 The example of reflection on RXF.

は，goal がメタレベル実行中に，sub_goal が失敗したときに throw が評価される．

5.2 メタエージェントの利用

リフレクションを用いたメタエージェントの利用例として，エージェントが他のエージェントを見るという行為の RXF におけるプログラミング例を示す．具体例として，右手と左手の2本の手を持つ2つのエージェントがいるとする．2つのエージェントをそれぞれ a，b と呼ぶ．ここで行われるエージェントの行動は，b が右手と左手をランダムに上げ下げするのを，a が b の状態を観察して b と同じように手を上げ下げする．概要を図5で示す．

図5中の丸はエージェントを表す．エージェントとして，a，b，ma，mb の4つを考える．ma は a のメタエージェントで，mb は b のメタエージェントである．a が b を観察することによって得られるデータを b の外見と呼ぶ．ma は a の外見を管理し，mb は b の外見を管理するようにプログラミングされている．以下に a が b の外見を得る過程を示す．a が b を観察するには，観察をするための述語（ここでは look）が定義されている必要がある．look は ma に b の外見を得させるようなメッセージを送るようにプログラミングされている．a 上で look が評価されると，look は ma に対して b の外見を得させるようなメッセージを送る．このときの look の仕様は，1）b のメタエージェントである mb に b の外見を得るためのメッセージを送信，2）mb から送られてきた b の外見を a に返す，である．

図6 はプログラムの概略を示している．図6の（1）が，ベースレベルのエージェント a で実行されるプログラムであり，（2）がメタエージェント ma で実行されるプログラムである．（1）は，reflect(ma,look) によって，look をメタレベル実行し，reply(X) によって結果を受け取るプログラムである．（2）は，reflect(ma,look) によって呼ばれる．（2）は，b の外見を得るために send(mb,get_appearance) によって mb に get_appearance メッセージを送信する．その後，receive(mb,X,[wait]) によって mb からの返事を待つ．mb からの返事を受信した後 read(A,database(D)) によって，エージェント a の節データベースを読み込み，write(A,datebase([reply(X)|D])) によって a の節データベースに reply(X) を追加する．

```
(1) エージェント a のプログラム
    look(X) :-
      reflect(ma,look),
        reply(X).
(2) メタエージェント ma のプログラム
    look(A) :-
      send(mb,get_appearance),
      receive(mb,X,[wait]),
      read(A,database(D)),
      write(A,database([reply(X)|D])).
```

図6 reflect 使用例
Fig. 6 A program using reflect.

本例では，エージェント間の観察をメタエージェント間のメッセージ通信によって実現した．問題解決のための計算と，問題解決のための計算をするための計算を分離することによってプログラミングを容易にし，拡張性を高めることが可能になる．たとえば，エージェントの外見がメタエージェント上でシミュレートされるものではなく，実際のハードウェアでもメタエージェントだけをプログラミングしなおせばよい．

5.3 組織化エージェント

本節では，RXF におけるリフレクション機構の使用例として，複数のエージェントから構成されたエージェントである，組織化エージェントの実装概要を示す．プログラム例として BDI アーキテクチャ[17]に基づく組織化エージェントを例にする．BDI（Belief, Desire, Intentions）アーキテクチャの基本的なアイデアは，belief，desire そして intention の集合によってエージェントの内部状態を記述し，エージェントが内部状態に基づき合理的に行動を選択するための制御機構を定義することである[18]．belief は，エージェントの現在の状態を表し，desire はエージェントの未来における好ましい状態を表す．intention は，エージェントの目標の集合の部分集合として記述される．これは，エージェントの資源に関する制約から達成可能な目標のうち実際に評価する目標を選ばなければならない場合が存在することを表す．

本組織化エージェントは，問題解決を行うベースエージェント a，b と，BDI の管理に基づき a，b を制御するメタエージェント m から構成される．

```
・メタエージェント m
(1) main :-
        do_BDI,
        control(a),
        control(b),
        main.
(2) do_BDI :-
        BDI の管理.
(3) control(N) :-
        agent_port(N,A),
        A を用いてベース
        エージェント N を制御.
(4) register(A) :-
        read(A,name(N)),
        assert(agent_port(N,A)).
・ベースエージェント a, b
(5) register :-
        reflect(m,register).
```

図7 組織化エージェントプログラム例
Fig. 7 An example of an united agent program.

図7は，BDIを管理するメタエージェントmのプログラム（図7の（1）（2）（3）（4））とタスクを遂行するベースエージェントa，bのプログラム（図7の（5））から構成される．本論文では，便宜上，プログラムの詳細な記述は省略する．

図7の（1）で示すmainは，mのメインループを実現するプログラムである．mは，BDIの管理とa，bの制御を繰り返す．BDIの管理は，do_BDIによって実行され，a，bの制御方針を決定する．図7の（2）で示すdo_BDIは，BDIの管理プログラムである．ここでは，BDIの管理を行い，a，bの制御方針を決定する．図7の（3）で示すcontrol(N)は，ベースエージェントNを（2）で決定した制御方針に基づき制御するプログラムである．図7の（4）で示すregister(A)は，ベースエージェントを表すエージェントポートAを保存するためのプログラムである．read(A,name(N))によってAが表すエージェントの名前Nが得られる．図7の（5）で示すregisterは，ベースエージェントが，mに自分自身を表すエージェントポートを渡すために用いるプログラムである．

本プログラムは，RXFにおけるユーザインタフェース[12]を介して，エージェント（m，a，およびb）の生成とエージェントの起動（aとbにおいてregisterの実行，およびmにおいてregisterの実行）により，並行実行される．

6. おわりに

RXFは柔軟で拡張性の高いマルチエージェントシステムを構築するために制約論理プログラミング，リフレクション機構を実装している．本研究では，エージェントの自律性を実現するために，リフレクション機構を実装した．またエージェントのリフレクションの能力を用いることによってエージェントが自分自身を動的にカスタマイズできる．応用例として分散TMSに基づくマルチエージェントシステムの実装がある[19]．

RXFは，AGENT 0のようなエージェントを記述するためのエージェント指向言語[20]として設計されたのではなく，エージェントを実際に実装するための，いわばエージェントのオペレーティングシステムとして開発実装された．

メタエージェントに基づく，メタレベル実行によって，処理のレベル（たとえば，問題解決を行うレベルや，問題解決のために実行するシミュレーションを行うレベル）に基づいた，プログラムの階層化を行うことが可能になり，問題解決のために実行するシミュレーションを実現するプログラムを，問題解決のためのプログラムから隠蔽することが可能になる．

文献15)によってPrologにおけるデータのメタレベル表現が提案されている．現在のRXFでは，実行効率をあげるためにバニラインタプリタ[15]のようなPrologメタインタプリタを構成せずに，メタインタプリタを実装している．このため，メタインタプリタにおけるインタプリタのメタレベル表現を得るために，ポートを利用している．RXFにおける，Prologメタインタプリタの実現は，論理型言語におけるリフレクションのモデルよりも実行効率の高いメタレベル実行の実現を目指している．

RXFにおけるリフレクションの主目標である，問題解決におけるメタな問題解決機構の実装のために，エージェントの計算機上での状態を表現するエージェントポートに加えて，問題におけるエージェントの状態を表現するためのポートを実装中である．本ポートによって，問題解決におけるエージェントの状態を参照するための手段を提供するのがねらいである．

RXFでは，リフレクティブな制御構造を持つエージェントの実装のために，(a) 手続き的リフレクション，(b) エージェントとメタエージェント間のリフレクションを利用している．現段階では，(b) のエージェントとメタエージェント間のリフレクションの実現のために (a) の手続き的リフレクションのメタレベル実行を利用している．今後の課題として，手続き的リフレクションによって構成されるリフレクティブタワーと，エージェントとメタエージェント間の関係に基づくリフレクティブタワーの関係のモデル化がある．モデルを提供することによってエージェントの実装の枠組みを提供するのがねらいである．

さらに，今後の課題として，RXF におけるエージェントの資源管理のためのリフレクション機構の実装があげられる．エージェントによる自己の利用する資源の管理は，エージェントが自律的に存在するソフトウェアであるという観点から必要である．現段階では，エージェントの実装のための言語処理系として試作されたので，資源管理のためのリフレクション機構を持たない．現在，エージェントの資源管理のためのリフレクション機構を実装中である．組織化エージェントに関連して，組織化エージェントを構成するエージェント群の共有資源の管理のために，ABCL/R2[10]における Hybrid Group Architecture のようなグループにおける共有資源の管理のためのリフレクション機構も必要である．

参考文献

1) 石田 亨：エージェントを考える，人工知能学会誌，Vol.10, No.5, pp.663–667 (1995).
2) 渡部卓雄：リフレクション，コンピュータソフトウェア，Vol.11, No.3, pp.5–14 (1994).
3) 大囿忠親，新谷虎松：リフレクティブな制約論理型言語 RXF の設計とその実装，第 9 回人工知能学会全国大会講演論文集，pp.299–302 (1995).
4) Perlis, D.: Language with Self-reference I: Foundations (or: We Can Have Everything in First-Order Logic!), *Aritificial Intelligence*, Vol.25, pp.301–322 (1985).
5) Lesperance, Y. et. al: *Foundations of a Logical Approach to Agent Programming, Intelligent Agents II*, pp.331–346, Springer-Verlag (1996).
6) Mullen, T. and Wellman, M.P.: *Some Issues in the Design of Market-oriented Agents, Intelligent Agents II*, pp.283–298, Springer-Verlag (1996).
7) 淵 一博：制約論理プログラミング，共立出版 (1989).
8) Weyhrauch, R.W.: Prolegomena to a Theory of Mechanized Formal Reasoning, *Aritificial Intelligence*, Vol.13, pp.133–170, (1980).
9) Brian, C.S.: Reflection and Semantics in Lisp, *Proc. 11th ACM Symposium on Principle of Programming Languages*, pp.23–35 (1984).
10) 増原英彦，松岡 聡，渡辺卓雄：自己反映並列オブジェクト指向言語 ABCL/R2 の設計と実現，コンピュータソフトウェア，Vol.11, No.3, pp.15–32 (1994).
11) Shoham, Y.: AGENT0: A Simple Agent Language and its Interpreter, *AAAI-91*, pp.704–709 (1991).
12) 大囿忠親，新谷虎松：マルチエージェントシステムのための制約論理型言語 RXF の実現，情報処理学会論文誌，Vol.37, No.10, pp.1765–1772 (1996).
13) Chu, D.: I.C. Prolog : A Language for Implementing Multi-agent Systems, *Proc. Special Interest Group on Cooperating Knowledge Based Systems* (1993).
14) Jaffar, J. and Lassez, J.L.: Constraint Logic Programming, *Proc. 14th ACM Symposium on Principles of Programming Languages*, pp.111–119 (1987).
15) 菅野博靖，田中二郎：メタ計算とリフレクション，情報処理，Vol.30, No.6, pp.694–705 (1989).
16) Steele, G.L.: *Common Lisp the Language*, 2nd Edition, Digital Press (1990).
17) Bratman, M.E.: *Intentions, Plans, and Practical Reason*, Harvard University Press (1987).
18) Muller, J.P.: *The Design of Intelligent Agents*, pp.17–25, Springer-Verlag, Berlin (1996).
19) 川上義雄，伊藤孝行，新谷虎松：分散 TMS に基づくマルチエージェントシステムの試作，第 9 回人工知能学会全国大会講演論文集，pp.267–270 (1995).
20) Shoham, Y.: Agent-oriented Programming, *Artificial Intelligence*, Vol.60, No.1, pp.51–92 (1993).

(平成 8 年 2 月 7 日受付)
(平成 9 年 5 月 8 日採録)

大囿 忠親（学生会員）

1972 年生．1997 年名古屋工業大学大学院工学研究科博士前期課程卒業．同年同大学院工学研究科博士後期課程入学．エージェントの研究に従事．人工知能学会会員．

新谷 虎松（正会員）

1955 年生．1982 年東京理科大学大学院理工学研究科修士課程修了．同年富士通（株）国際情報社会科学研究所入所．知識情報処理，論理プログラミングなどの研究に従事．現在，名古屋工業大学知能情報システム学科助教授．工学博士．分散人工知能，意思決定支援システム，マルチエージェントシステムの研究に従事．電子情報通信学会，ソフトウェア科学会，人工知能学会など各会員．