マルチエージェントシステム構築環境に関する研究

2000年

大囿忠親

論文要旨

知的で自律した問題解決主体である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は，広く利用されており，今後もさらに利用されることが予想される．マルチエージェントシステムは，従来のソフトウェアには無い性質をもつ新しい種類のソフトウェアなので，その効率的な開発のためには，新たな手法が必要である．本研究では，マルチエージェントシステムの実装のための効率的な手法を実装した開発環境としてRXF（Reflective Familiar）を試作した．本研究の成果として，エージェントの特徴である自律性を実現するための機能であるリフレクションにおけるオーバヘッドの低減手法，エージェント構築機能を述語によって統一的に操作するための手法，マルチエージェントシステムのための新規なデバッグ手法，そして動的な環境に適応可能な事例ベース推論システムである階層的事例ベース推論システム，が挙げられる．本論文では，マルチエージェントシステム構築に必要な機能を明らかにし，それらの実装方法を提案する．本論文の構成を以下に示す．

1章では，研究の動機と目的について述べる．エージェントは，将来のコンピューティングを担う技術であり，さらなる研究が必要である．

2章では，本研究の背景について説明する．本研究の背景として，マルチエージェントシステム，制約論理型言語，リフレクション，デバッグ，そして事例ベース推論について説明する．

3章では，まず始めにRXFを概観する．RXFは，エージェント記述言語，エージェントオペレーティングシステム，そしてエージェントフレームワーク，の3つから構成される．エージェント記述言語は，エージェントを記述するためのプログラム言語である．エージェントオペレーティングシステムは，エージェントの基本特性を実現するための基盤となるソフトウェアである．エージェントフレームワークは，エージェントを実装するためのライブラリであり，推論システムなども含む．本章では特に，エージェント記述言語について述べ，RXFのリフレクション機能について説明する．ここでは，メタレベル表現を動的に生成することによって，メタレベル表現生成のオーバヘッドを減少するための手法を示す．実験により本手法によってオーバヘッドを軽減できること

を示す.

4 章では，エージェントオペレーティングシステムの設計と実装について述べる．本エージェントオペレーティングシステムにより，自律性や社会性などのエージェントの性質の実現が可能になる．ここでは，エージェントの外界へのアクセスは，パートと呼ばれるモジュールを介して実行される．エージェントオペレーティングシステム上では，パートに基づくリフレクション機能により，エージェントの自己改変などを実現する．パートによって，レガシーアプリケーションとの連係機能や，日本語の形態素解析機能などが提供されている．さらに，エージェントオペレーティングシステムの機能を述語を用いてプログラム上から統一的に扱うための機能を実装した．本機能により，評価器や節データベースの階層化機能やパートに基づくリフレクション機能をプログラム上から統一的に扱うことが可能になる．本機能により，モバイルエージェントの実装も可能になる.

5 章では，マルチエージェントシステムのためのデバッグ手法を提案する．マルチエージェントシステムのデバッグは，逐次実行プログラムのデバッグに比べて困難である．本デバッガの開発に於いて，マルチエージェントシステムのデバッグ時に発生する probe effect の回避を試みた．本デバッガは，デバッグ時に複数エージェントの実行速度を調整することによって，probe effect を軽減する．probe effect の現象を示すために，実験により実際に実行速度が調整されることを示す．これにより，マルチエージェントシステムのデバッグにおけるプログラマの負担を軽減できる.

6 章では，エージェントフレームワークの一部として提供される推論システムについて説明する．始めに，ルールに基づく協調プロトコル記述システムについて述べる．本システムにより，協調プロトコルをプロダクションルールとして記述可能になる．次に，階層的事例ベース推論システムについて述べる．メタ事例を用いた事例検索手法とメタ事例の構築手法を提案する．本手法では，メタ事例を用いて Nearest-Neighbor 法における評価軸に重み付けすることにより，動的適応を実現する．本手法によって，環境の動的な変化やユーザの好みに基づく推論機構の調整などが可能になる.

7 章では，本研究で得られた成果及び，今後の課題について述べる.

目 次

図目次

表 目 次

第1章　はじめに

将来，多数の機器が通信機能を持つようになるという．現在でも通信機能を持つ多くのコンピュータがネットワークに接続されているが，コンピュータ以外の機器も通信機能を持ち，ネットワークに接続されるようになる．文献 [1] によると，SUN Microsystems 社の技術である JINI を用いることによって，さまざまな種類の機器が通信機能を持ち，ネットワークに接続することが可能になる．JINI とは，ネットワークにさまざまな種類のハードウェアやソフトウェアを接続して有効活用するためのアーキテクチャである．JINI を用いることによって，プリンタを買ってきてネットワークに接続するだけで，ネットワークにつながれた他の機器からそのプリンタが使えるというようなことが可能になると言われている．JINI によって，携帯電話，PDA，ヘッドフォン・ステレオ，ディジタル・カメラ等の携帯型情報端末は言うに及ばず，家庭内のあらゆる情報機器がリンクしてコンテンツのやりとりを行う．さまざまなモノがネットワークに接続される時代が来る．

ネットワークへの接続というと，機器間をケーブルで接続する様子を想像してしまうが，無線による接続も個人の手に届くところにある．現在でも PHS を使えば 64kbps の接続速度でインターネットに無線でアクセスすることができる．次世代の移動通信，IMT-2000 (Iternational Mobile Telecommunications-2000) では，1.5Mbps の通信速度を実現するという [2]．IMT-2000 では，インターネットとの接続が最初から考慮されている．すなわち，移動体もあたりまえのようにインターネットに接続する時代が来る．移動通信技術の発達により，相当数の機器が移動通信によりインターネットに接続されるようになるだろう．現在でも，移動通信機器の普及には目を見張る物がある．1998 年 9 月末現在における自動車電話，携帯電話，そして PHS を合わせた普及率は 33.9 % である [3]．さらに，将来は，すべての移動体に移動通信機能が装備される．例えば，車やペットのように動くものすべてが無線による通信手段を持つのである．

情報家電などの新しい種類の情報機器のインターネットへの接続や，移動通信によるインターネット接続などにより，ありとあらゆるものがインターネットに接続され，インターネットに接続されるものの数は，想像を絶する物になるだろう．これらの物が効率よく連携して機能すれば便利になると考えられるが，物を効率よく連携させるにはどうすれば良いのだろうか．ありとあらゆるものがインターネットに接続されるような状況において，世界中に分散した機器を一極集中管理するのは不可能である．よってそれらの機器を適切に分散して管理する必要がある．

通信経路などのインフラストラクチャの管理は，将来も専門家によって実施されると考

えられるが，すべての通信機器の管理に専門的な知識が必要であるいうのは非現実的である．自動車の例に例えると，ドライバーは，自動車を運転するために自動車のに関する専門的な知識を必要としない．道路の管理や，自動車の定期点検は専門家が行うが，自動車の運転や日常的な管理はドライバーが行っている．もし運転や日所的な管理に高度に専門的な知識が必要だったならば，自動車はこれほど普及しなかったと考えられる．情報家電などの新しい種類の情報機器が普及し人々の生活をより豊かにするためには，家庭内における管理や，移動体の動的な振る舞いへの対応などは，自動化されなくてはならない．現在ネットワークの管理には，専門知識が必要である．このような状況では，通常のユーザにとって複数の通信機器をネットワーク接続することが困難である．次世代の情報機器が通信機能を備えていたとしても，それが有効活用されず，個々の通信機器ががバラバラの状態では通信機器全体としての能力を最大限に発揮できない．よって，専門知識を必要とする管理の必要無しに，複数の機器が連携できるような枠組みが必要となる．例えば，複数の情報機器の自動的な組織化機能が必要である．ユーザが情報機器を買ってきてネットワークに接続するだけでそれが組織に組み入れられ，有効活用されることが好ましい．物が有機的に動作し，自動的に組織化することによって，組織が個々の能力の和以上の能力を発揮できれば良いと考えるのは自然である．自動的な組織化の実現のためには，通信機器自体が，自律的に動作し組織を自動的に構成できる必要がある．マルチエージェントシステムは，このような課題に対する本質的なアプローチとして研究されている．

多数の情報機器による問題だけでなく，1台のコンピュータにおいてもエージェントの活躍する場面が存在する．コンピュータ技術の発展により，安価で高性能なコンピュータを個人で使える時代になった．コンピュータが高機能になるにつれて，コンピュータの操作は複雑になっていった．GUI（Graphical User Interface）は，ある程度の成功を収めたと考えられているが，完全な解決策ではない．あらゆるユーザが，現在の複雑なコンピュータを自由に扱えるようにするためには，GUI だけでは不十分である．そのため，コンピュータによる知的なユーザの支援の必要性が，次第に増してきている．コンピュータがユーザの意図を理解することによって，ユーザの支援や作業の自動化を行うのである．Norman は，Information Appliance と呼ばれる従来の計算機とは異なるアプローチの情報機器を提案している [4]．このような新しい種類の情報機器でも，複数の機器を効率よく連携させるためには，エージェント技術が必要である．

ユーザインタフェースの問題だけでなく，インターネット上の情報の扱いに関してもエー

ジェントが有効である．インターネットが世間に認知されるに従って，しだいに普通の人がコンピュータをメディアとして使うようになった．すなわち，コンピュータを，ただの計算機ではなく，情報を共有するための道具と見なすようになった．今までは，普通の人にとってコンピュータは，ワープロや表計算をしたり絵を描いたりするための道具としてしか見られていなかった．しかし，今では文書を作成したりするだけでなく，情報を得たり提供するためにコンピュータを使うようになった．特に WWW の普及によって，個人が低いコストで情報を公開できるようになったので，大量の情報が世に出回るようになった．しかし，WWW 上に存在する大量の情報の中から，有益な情報を見つけるのは手間のかかる作業であり，エージェントによる代行が望まれている．

また，上記のようなアプリケーションからのニーズだけでなく，科学的なニーズによってエージェントの研究が行われている．エージェントは，世の中の複雑でモデル化しにくい問題を解決するための手段としても考えられている．従来の人工知能研究との差は，個々の問題解決主体による問題解決ではなく，個々の問題解決主体が構成する社会による問題解決に焦点が当てられている点にあると考える．または，動的に変化する現実社会において実際に利用可能な問題解決主体としての焦点もある．

以上のように，マルチエージェントシステムの研究は，様々な観点から行われている．これらの研究の発展や実際に有益なマルチエージェントシステムの実装のための優れたマルチエージェントシステム開発環境が望まれている．

本研究では，自律した合理的なエージェントの実装のための効率的な開発環境の実現のために，RXF（Reflective Familiar） [5, 6] を試作した．本開発環境は，エージェント記述言語，エージェントオペレーティングシステム，そしてエージェントフレームワーク，の 3 つから構成される．エージェント記述言語は，エージェントを記述するためのプログラム言語である．エージェントオペレーティングシステムは，エージェントの基本特性を実現するための基盤となるソフトウェアである．エージェントフレームワークは，エージェントを実装するためのライブラリであり，推論システムなどを含む．RXF は，実装言語として C++ を用いてコンパクトに実現されている [7].

本開発環境は，エージェント記述言語として，制約論理型言語 [8, 9] による，エージェントの開発を支援する．本研究では，制約論理型言語は，論理式と数式を同時に効率良く扱うことが可能と言う利点のため，他の手続き型言語，関数型言語，そして論理型言語よりも有利であると考えている．論理式は，エージェントが用いる知識を容易に表現するこ

とが可能である．また，エージェント間のメッセージ通信において用いられる KQML [10] や KIF [11] は，メッセージの表現に論理式を用おり，論理式が扱える言語はエージェントの実装において有利といえる．さらに，制約論理型言語は，Prolog に準拠した論理プログラムを扱うことができるので，論理プログラムを生成する学習機構との連携が簡単に行え，エージェントの学習機構の実装という観点からも有利である．エージェント記述言語上では，自律エージェントの実現に利用されるメタレベルプログラミング [12] ためのリフレクション [13] 機構を追加した．本リフレクション機能の特筆すべき点は，メタレベル実行時のオーバヘッドの少なさである．これは，エージェントポートを用いてメタレベル表現を動的に生成することによって実現した．

エージェントオペレーティングシステムは，制約論理型言語による，エージェントの開発を支援するために，並行処理の実現のためのマルチスレッド機能そしてメッセージ通信機能などを実現する．エージェントの外界へのアクセスは，パートと呼ばれるモジュールを介して実行される．エージェントオペレーティングシステム上では，パートに基づくリフレクション機能により，エージェントの自己改変などを実現する．パートによって，レガシーアプリケーションとの連係機能や，日本語の形態素解析機能などが提供されている．エージェントオペレーティングシステムの機能をプログラム上から統一的に扱うための機能を実装した．本機能により，評価器や節データベースの階層化機能やパートに基づくリフレクション機能をプログラム上から統一的に扱うことが可能になる．本機能により，モバイルエージェントの実装も可能になる．

マルチエージェントシステムのデバッグは，逐次実行プログラムのデバッグに比べて困難である．本研究では，マルチエージェントシステムのデバッグを支援するためのデバッガを開発した．本デバッガは，複数エージェントの実行速度を調整することによって，マルチエージェントシステムのデバッグを支援する．本デバッガは，デバッグ自に複数エージェントの実行速度を調整することによって，マルチエージェントシステムのデバッグ時に問題となる probe effect を軽減し，プログラマの負担を軽減する．

マルチエージェントシステムの実装において，エージェントに適した推論機能の実現が必要である．RXF のエージェントフレームワークでは，エージェントの実装に適したルールベースシステムと事例ベース推論システムを推論機能として提供する．本ルールベースシステムは，割込処理が可能なように拡張してある．本事例ベース推論システムは，環境の変化に基づいて適切な類似性基準を構築するための機能を実現している．

以上のような機能を用いることで，マルチエージェントシステムを効率よく構築することが可能になる．

1.1 本論文の構成

図 1.1 を用いて本論文の構成について説明する．2 章では，本研究の背景について説明する．本研究の背景として，マルチエージェントシステム，制約論理型言語，リフレクション，デバッグ，そして事例ベース推論について説明する．3 章では，まず始めに RXF を概観する．RXF は，エージェント記述言語，エージェントオペレーティングシステム，そしてエージェントフレームワーク，の 3 つから構成される．エージェント記述言語は，エージェントを記述するためのプログラム言語である．エージェントオペレーティングシステムは，エージェントの基本特性を実現するための基盤となるソフトウェアである．エージェントフレームワークは，エージェントを実装するためのライブラリであり，推論システムなども含む．本章では特に，エージェント記述言語について述べ，RXF のリフレクション機能について説明する．ここでは，メタレベル表現を動的に生成することによって，メタレベル表現生成のオーバヘッドを減少するための手法を示す．実験により本手法によってオーバヘッドを軽減できることを示す．4 章では，エージェントオペレーティングシステムの設計と実装について述べる．本エージェントオペレーティングシステムにより，自律性や社会性などのエージェントの性質の実現が可能になる．ここでは，エージェントの外界へのアクセスは，パートと呼ばれるモジュールを介して実行される．エージェントオペレーティングシステム上では，パートに基づくリフレクション機能により，エージェントの自己改変などを実現する．パートによって，レガシーアプリケーションとの連係機能や，日本語の形態素解析機能などが提供されている．さらに，エージェントオペレーティングシステムの機能を述語を用いてプログラム上から統一的に扱うための機能を実装した．本機能により，評価器や節データベースの階層化機能やパートに基づくリフレクション機能をプログラム上から統一的に扱うことが可能になる．本機能により，モバイルエージェントの実装も可能になる．5 章では，マルチエージェントシステムのためのデバッグ手法を提案する．マルチエージェントシステムのデバッグは，逐次実行プログラムのデバッグに比べて困難である．本デバッガの開発に於いて，マルチエージェントシステムのデバッグ時に発生する probe effect の回避を試みた．本デバッガは，デバッグ時に複数エージェントの

実行速度を調整することによって，probe effect を軽減する．probe effect の現象を示すために，実験により実際に実行速度が調整されることを示す．これにより，マルチエージェントシステムのデバッグにおけるプログラマの負担を軽減できる．6 章では，エージェントフレームワークの一部として提供される推論システムについて説明する．始めに，ルールに基づく協調プロトコル記述システムについて述べる．本システムにより，協調プロトコルをプロダクションルールとして記述可能になる．次に，階層的事例ベース推論システムについて述べる．メタ事例を用いた事例検索手法とメタ事例の構築手法を提案する．本手法では，メタ事例を用いて Nearest-Neighbor 法における評価軸に重み付けすることにより，動的適応を実現する．本手法によって，環境の動的な変化やユーザの好みに基づく推論機構の調整などが可能になる．7 章では，本研究で得られた成果及び，今後の課題について述べる．

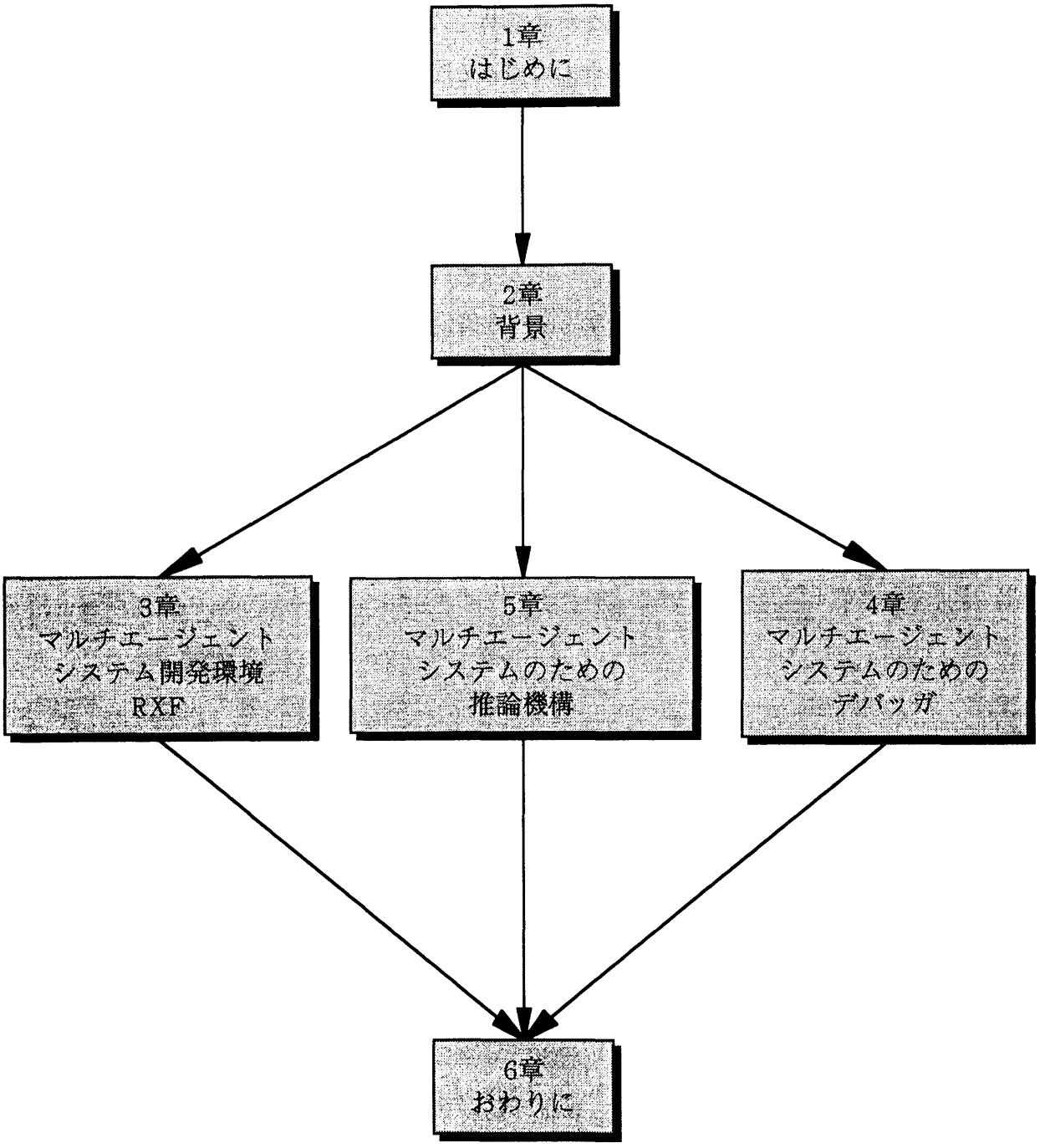
1章
はじめに
2章
背景
3章
マルチエージェント
システム開発環境
RXF
5章
マルチエージェント
システムのための
推論機構
4章
マルチエージェント
システムのための
デバッガ
6章
おわりに

図 1.1: 本論文の構成

第2章　研究の背景

2.1 マルチエージェントシステム

マルチエージェントシステムとは，相互作用する知的なエージェントから構成されるシステムである [14]．マルチエージェントシステムは，以下のような問題領域における有望な手段として研究されている．

- 電子商取引
- インターネットにおける情報検索
- 電話回線ネットワークにおける実時間制御
- 運送システムのモデル化及び最適化
- 航空管制の改善
- 自動的な会合計画の立案
- 製造プロセスの最適化
- ビジネスプロセスの分析
- エンターティメント
- 巨大組織における制御フローの設計及び再設計
- 知性の社会的側面の調査や，複雑な社会現象のシミュレーション

以上の問題領域における特徴をまとめると， エージェントが要求される背景は，文献 [15] のようにまとめることができる．

- オープン性
  インターネット上における情報・サービスはユーザの権限の及ばないところで生成し，修正され，移動し，陳腐化し，消滅する．従って，ネットワーク上の情報に関するどんな記録も常に不完全なものとなる．ネットワーク上の情報はこの意味でいつも外に向かって開いている．

- 多様性
  インターネット上の情報は，提供者の意図，用途の違いにより，多様な形式，内容，質を持つ非正規情報になっている．

- 分散性
  インターネット上では，情報源が互いに独立に存在，制御された系である．

- 広域性
  インターネット上の情報は広域に分散しており，単体での計算に比べネットワーク上での通信が著しく遅いことから，通信時間や通信量といった物理量を意識せざるをえない．

- 大量性
  インターネット上では，原理上利用可能な情報の量が，現実的に処理可能な量を超えて提供されている．

- 非専門性
  計算機が一部の専門家のものでなくなったように，ネットワーク社会が計算機の専門外のユーザにも開放されたものとなっている．

これらの特徴を持つ問題を解決するためには，分散人工知能などの従来の枠組みだけでは不十分であり，そのためマルチエージェントシステムが研究されているのである．

2.1.1 マルチエージェントシステムとは

マルチエージェントシステムとは，相互作用する知的なエージェントから構成されるシステムである [14]．ここで文献 [14] では，3 つのキーワード “エージェント”，“知的な”，そして “相互作用する” について，以下のように述べられている．

> “エージェント（Agent）” とは，自律的な計算主体である．エージェントは，センサーを通じて環境を認識し，その環境に作用する能力を持つ．エージェントが計算主体というのは，エージェントは，コンピュータ上で実行するプログラムという形態で物理的に存在することを意味する．エージェントが自律的であるというのは，エージェントがエージェント自身の行動を制御するための何

らかの仕組みを持ち，人間や他のシステムによる仲介無しに活動することができることを意味する．エージェントは，目的を満たすために，目標をたてタスクを実行する．一般的には，それらの目標やタスクは，矛盾していることもあれば補完的なこともある．

“知的な（intelligent）”とは，エージェントがある与えられた評価基準に対して最適になるように目標を達成したりタスクを実行することを指す．エージェントが知的というのは，エージェントが全知全能であるとか，絶対に失敗しないという意味ではない．むしろ，エージェントがさまざまな環境において，柔軟に又は合理的に動作できることを意味している．それゆえに分散 AI は，エージェントが柔軟で合理的に振る舞うことを可能にするために，問題解決，プランニング，探索，意思決定，そして学習のような処理に大きな焦点を当てている．そして分散 AI は，マルチエージェントシステムにおけるこれらの処理の実現にも焦点を当てている．

“相互作用する（interacting）”とは，エージェントが目標の遂行やタスクの実行に関して，エージェントが他のエージェントや人間によって影響されることを意味する．相互作用（interaction）は，環境を通して間接的に発生したり，共通の言語を用いて直接発生する．初期の分散 AI は，相互作用の形態として協調（coodination）に焦点を当てていた．協調は，目標の達成やゴールの遂行において特に重要である．協調の目的は，複数のエージェントによって好ましいことを実行し，好ましくないことを避けることである．エージェントは，エージェントの目標やタスクに関して協調するために，エージェントの活動を考慮しなければならない．協同（cooperation）と競争（competition）という，2 種類の基本的な協調があり，これらは対照的である．共同において，複数のエージェントは，一緒に作業を行い，共通の目標を達成するために彼らの知識や能力を収集する．それに対して，競争の場合は，複数のエージェントは，対立して作業を行う．なぜならば，彼らの目標は，競合しているからである．共同するエージェントは，個々で達成できないようなことをチームとして達成する．そして一緒に失敗又は成功する．競争するエージェントは，他のエージェントの損失によって，エージェント自身の利益を最大化しようとする．よって，

あるエージェントの成功は，その他のエージェントの失敗を意味する．

マルチエージェントシステムを構成するエージェント群は，知的に相互作用することによって，協調して問題解決を行う．ここで，単に個々のエージェントの能力の和以上の能力をシステム全体で持つことを目標とする．三人よれば文殊の知恵，というのが協調の真骨頂である．三人よれば文殊の知恵のことわざの意味は，多人数で協力して知恵を出し合えば，文殊が考えるような良いアイデアがでることを意味している．ここで重要な点は，個々の人間の能力では太刀打ちできないような能力を持つ存在である文殊の知恵を，個々が協力することによって達成できるという点である．これは，全体が部分の和以上になるということを別の面からとらえると，部分の属性ではないものが，全体の属性（文殊の知恵）として現れるということであり，これを創発（emergence）と呼ぶこともある．

2.1.2 エージェントの性質

エージェントの全世界において共通の定義は存在しない [14]．本研究では，エージェントは，柔軟な自律的行動能力を持つソフトウェアする．ここで，自律的とは，エージェントが人間やその他のシステムからの干渉無しに，エージェント自身の行動や内部状態を制御できるという性質を持つことを意味する．ここで柔軟とは，文献 [16] で述べられている (1) 反応性 (2) 自発性，そして (3) 社会性，の 3 つの特徴を意味することとする．(1) の反応性は，エージェントが，環境を認識する能力を持ち，環境の変化に適切に反応することができる性質を意味する．(2) の自発性は，エージェントが，目標を満たすように，能動的に活動する性質を意味する．(4) の社会性は，他のエージェントと相互作用する性質を意味する．エージェントは，これらの性質を知的に実現する能力を持つことが望ましい．これらの性質を以下にまとめておく．

- 自律性（autonomy）

  エージェントは，人間やその他のシステムからの干渉無しに，それらの行動や内部状態を制御する能力を持つ．エージェントが，知識などを利用して自ら判断して行動する点が重要である．エージェントは，調子が悪くなったときに，自分が調子の悪いことを認識し，ユーザの助けを借りずに解決することが好ましい．なぜなら，エージェントの調子が悪くなっても，一般的なユーザがそれを直すことができないからである．

- 反応性（reactivity）

  エージェントは，環境を認識する能力を持ち，それらの設計目的を満たすように環境の変化に適切に反応する．環境には，人間や他のエージェントも含まれている．ある種のエージェントには，未知の環境に対する適応も期待されている．

- 自発性（pro-activity）

  エージェントは，ある目的を満たすように，能動的に活動する．単に反射的に動作するだけや，外部からの作用によって受動的に動作するだけではなく，何らかの合理性基準などに従って能動的に活動する点が重要である．

- 社会性（social ability）

  エージェントは，それらの設計目的を満たすように，他のエージェントと相互作用する能力を持つ．共通の通信言語を用いるなどして多数のエージェントが協調などを行う点が重要である．異種エージェント間の相互作用には，共通の通信言語や知識表現形式が欠かせない．エージェント間は，相互作用し合うことで協調したり，組織を形成したりする．

以上の性質以外にも信念・意図・責務といった心理状態の概念を導入したり，人間のように振る舞うための情緒や感性などが議論されることもあるので，エージェントの性質といってもさまざまな視点や考え方があり，これがエージェントの概念を一層複雑にしている一因ともなっている [17].

2.1.3 エージェントの利用目的

一般的なエージェントの利用目的は，大きく分けて以下の 8 つに分類できる [18].

- 個人利用（Personal Use）

  パーソナルコンピュータの処理性能や能力の向上につれて，パーソナルコンピュータの管理支援をエージェントによって行う必要があると考えられる．エージェントは，スケジュール管理やファイル管理を支援することができる．

- ネットワーク管理（Network Management）

  インターネットが大きく複雑になるにつれて，エージェントによってネットワークの

アーキテクチャやトラフィックを管理する必要があると考えられる.

- 情報・インターネットアクセス（Information and Internet access）
  大量の情報を効率よく扱うために，エージェントによって情報のフィルタリングや優先度付けなどの情報の整理が行われている．さらにエージェントは，良質な情報を素早く，そして信頼できる方法で得るための方法をユーザに提案する.

- 移動管理（Mobility Management）
  今の社会では人々は，さまざまな移動手段を用いて動き回ることができる．そこでユーザが，どこにいても遠隔地から便利なソフトウェアやサービスを受けるためのインフラストラクチャが必要である．エージェントは，そのような遠隔地からのソフトウェアへのアクセスや，ネットワークにおけるそのような資源へのアクセスを支援する.

- 電子商取引（E-commerce）
  インターネットに広がりによって電子商取引の有望性が大きくなってきている．商品やサービスの売買の電子化に自動化により，取引が素早く行われる．エージェントを用いて，製品を選択し，仕様を決定し，最適な価格を交渉し，消費者から集金をすることができる．電子商取引は，大企業だけでなくインターネットにアクセスできるだれでもが参加できる.

- ユーザインタフェース（Computer User Interface）
  エージェント技術によって，ユーザインタフェースは使いやすくそして簡単になる可能性がある．エージェントによるユーザインタフェースは，簡単に使え，簡単に理解でき，ユーザの混乱を避け，そして直感的でなくてはならない．エージェントによるユーザインタフェースは，重要な分野である．エージェントは，ユーザの好みを理解し，ユーザインタフェースを変化させたり，ショートカットを追加したりするような知的な決定を行う．将来のエージェントは，ユーザが心地よいと思うように柔軟に適応するだろう.

- アプリケーション開発（Application development）
  エージェント技術は，商用ソフトウェアの設計・開発において，有望であると考えられている．エージェントは，プログラム，コード，モジュール，設計，仕様，そして

他に利用可能なアプリケーション開発ツールを探すことができる．従来の動的で複雑で分散したアプリケーションが，ソフトウェアエージェントによる開発の最初の対象になる．

- 軍事利用（Military applications）
  軍事組織は，知的なものに大きな興味がある．人間のようなエージェントは，機密情報の収集を極秘に行うことができる．軍や人工知能研究者は，情報収集や情報分析のためのエージェント技術の研究を行っている．このようなソフトウェアスパイは，消耗品として扱え，水も食料もいらず，信頼でき，さらに無休で働くことできる．

2.1.4 エージェントの研究

エージェントを実装するためのプログラミングパラダイムとして，エージェント指向プログラミング [19] が提案されている．例えば，AGENT 0[20] は，エージェント指向プログラミング言語である．エージェント指向プログラミングの特徴として，エージェントの様相の管理を提案．組込みの様相管理機構を持ち，様相管理機構によって，プログラムのコンテキストを制御することが挙げられる．

RETSINA（Reusable Task Structure-based Intelligent Network Agents）[21] は，エージェントを構築するための再利用可能なマルチエージェント計算基盤である．RETSINA では，インタフェースエージェント，タスクエージェント，そして情報エージェントの 3 種類のエージェントを定義している．これらのエージェントは，KQML を用いたエージェント間通信によって協調する．RETSINA は，分散環境やインターネットを利用した意思決定支援システムの実装に用いられている．RETSINA を用いて実装されたアプリケーションにより，インターネットにおけるマルチエージェントシステムの有効性が示された．

論理型言語に基づくエージェントプログラミングの関連研究として，I.C.Prolog[22] の実装研究がある．I.C.Prolog は，マルチエージェントシステムを構築可能とする論理型言語処理系の実現という主眼から，複数プログラムの並行実行機能，プログラム間の非同期通信，ネットワークを介した通信機能の 3 点を Prolog に付加した論理型言語である．一方，本研究における RXF は I.C.Prolog の主目標であるマルチエージェントシステムを構築するための機能に加えてさらに 2 つの拡張，すなわち，リフレクションおよび制約論理型プログラミング機能を実現した言語である．また，I.C.Prolog とは異なり，ユーザに対して，

オペレーティングシステムや通信プロトコルに依存しないメッセージ通信機構を提供する．さらに，エージェントの構築という観点から，メッセージに対して割込み処理を定義することができる．これにより，問題解決中のエージェントに対する推論の停止処理を容易に実現できる．

ABShell [23] は，エージェント構築のための再利用可能な言語とサービスを提供可能なエージェント構築シェルである．ABShell では，基礎的な通信機能や協調機能を提供することによって，プログラマがエージェントを一から記述する労力を軽減する．RXF では，エージェントフレームワークによって同様の機能を実現している．さらに，RXF では，メタレベルプログラミングに基づく柔軟なエージェントの構築が可能である．

AKL（AGENTS Kernel Language）[24] は，並行制約プログラミング言語である．AKL における計算は，制約を持つエージェント間のインタラクションによって実行される．

TACOMA（Tromsy And COrnel Moving Agents）[25][26] はエージェント構築を支援するためのオペレーティングシステムである．TACOMA では，エージェントを構築するために必要な機能を提供することによって，エージェントの構築を支援する．RXF の，エージェントオペレーティングシステムは，記述系として論理型言語を用いている．Clause Mapped Part によって，エージェントの内部状態を述語として統一的に扱えるという利点がある．

JATLite（Java Agent Template, Lite） [27, 28] は，インターネットにおける頑健な通信が可能なエージェントの実装のための Java 言語で記述されたプログラムパッケージである．JATLite は，共通の高レベル言語とプロトコルを用いるエージェントのためのテンプレートを提供する．JATLite の特徴は，Agent Message Router を利用した頑健なメッセージ通信機能である．RXF では，頑健性については，プログラム依存なので，このような支援の追加も必要である．

INTERRRAP [29] は，動的なマルチエージェントシステム環境において自律的で資源に限りがあるエージェントをモデル化することを目指して開発された．INTERRRAP は，BDI アーキテクチャに基づいて実装されている．BDI アーキテクチャでは，エージェントの心理状態は階層化されている．

マルチエージェントのための協調アーキテクチャの研究として，有機的プログラミング [30] に基づく言語処理系である GAEA[31] の研究が挙げられる．GAEA は，包摂アーキテクチャ（Subsumption Architecture）の論理型言語における実装である．Subsumption Architecture では，プログラムの集合をセルと呼ぶ．セルは複数存在して，セル間には上

位下位関係を持つものが存在する．下位のセルのプログラムは，上位のセルのプログラムによって制御される．GAEA ではこの機能をリフレクションと呼んでいる．

RXF では，BDI アーキテクチャや包摂アーキテクチャのような特定のエージェントアーキテクチャを提供しないが，エージェントフレームワークの階層化機構を用いることで BDI アーキテクチャや包摂アーキテクチャのような階層構造を用いるシステムを容易に実現できる．

MadKit [32] は，組織モデルに基づくマルチエージェントシステムのためのプラットフォームである．MadKit では，異種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ことを目的として，通信言語などの汎用的なエージェントの機能を提供している．

JAFMAS [33] は，マルチエージェントシステムにおける協調知識と協調プロトコルを表現・開発するためのフレームワークであり，Java 言語上で構築されている．JAFMAS は，発話-行為に基づくマルチエージェントシステムの開発のための汎用的な手法を提供する．

RXF では，プロダクションシステム KORE/IE に基づく協調プロトコル記述言語である抹茶によって協調プロトコルを記述することが可能となる．

文献 [34] では，エージェントの役割モデル（role model）と Aspect Oriented Programming [35] の関係について議論されており，UML を用いたエージェントの役割モデルの記述が試みられている．Aspect Oriented Programming は，新しいプログラミングパラダイムであり今後の発展が期待される．Aspect Oriented Programming では，基本となるアルゴリズムを実装したプログラムと，アルゴリズムをある局面（aspect）に最適化するためのプログラムを合成することによって，さまざまな局面へのプログラムの最適化を少ない開発工程で行えるようにする．このような考え方は，まさにエージェントに適していると考えられるので Aspect Oriented Programming のエージェントの適用は大変興味深い．

文献 [36] では，ユーザにとってのエージェントの理解しやすさを考慮したエージェントアーキテクチャである The Expressivator について述べられている．The Expressivator の特徴は，narrative psychology に基づいた，説明に基づくエージェントの理解しやすさの実現にある．実現のために，メタレベル制御を用いている．The Expressivator におけるメタレベル制御を用いた説明では，ユーザにとって理解が困難な用語を用いて説明するのではなく，ユーザにとって理解しやすい言葉で説明するための機能を用いている．このような機能は，RXF におけるデバッガの開発においても重要である．デバッガが，エージェントに関する情報を最も簡単な表現で最も的確に説明できれば，デバッグの効率があがると考

えられるからである.

2.1.5 エージェントの機能

本研究において想定されているエージェントは,(1) 自律性 [37],(2) 永続性, の 2 つの特徴を持つソフトウェアである.(1) のエージェントの自律性とは, エージェントが自己管理をおこなえることを意味する. エージェントは, 突然のエラーや例外にも自分自身の状態にあわせて適切に対応しなければならない. 複数の自律したエージェント同士が, 独立した計算機資源を持ち, 並行動作することも含まれる.(2) のエージェントの永続性とは, エージェントが永続的に存在できることを意味する. 永続的に存在できるというのは, 1 次記憶装置上のエージェントの状態を 2 次記憶装置上に待避でき, かつ, 2 次記憶装置上から 1 次記憶装置上へ復帰できることを意味する. すなわち, エージェントを実行中のコンピュータの電源を切っても電源をいれればエージェントが復帰できるこという意味である.

2.1.6 エージェントの設計指針

本研究における一般的なエージェントは, 図 2.1 で示される構成を持つ. エージェントは, エージェント自身 (図 2.1 における真ん中の大きな円で示される) がその外側から隔離されている. エージェントの外側をエージェントの外界と呼ぶ. エージェントは, 外界へ干渉するための機能を持ち (図 2.1 における黒い円で示される), この機能を持つ部分をポートと呼ぶ. エージェントは, ポートを用いることによって外界に干渉でき, また外界からの干渉もポートを用いることによって受ける. エージェントは, ポート以外の外界への干渉手段を持たない.

エージェントの内部は, 主に複数のスレッドと複数の記憶領域で構成される. 図 2.1 では, スレッドと記憶領域しか描かれていないが, 実際には, これ以外のものがあっても良い. スレッドは, プログラムを実行する. エージェントは, 自己管理や, タスクの遂行のために複数の実行ストリームを持つので, このように複数のスレッドを実行できなければならない. 記憶領域は, データを記憶する. 複数の記憶領域は, それぞれ独立してデータを記憶する. 独立した複数の記憶領域を持つことによって, 自己管理のためのデータと, タスク遂行のためのデータを分離でき, データの管理が容易になる.

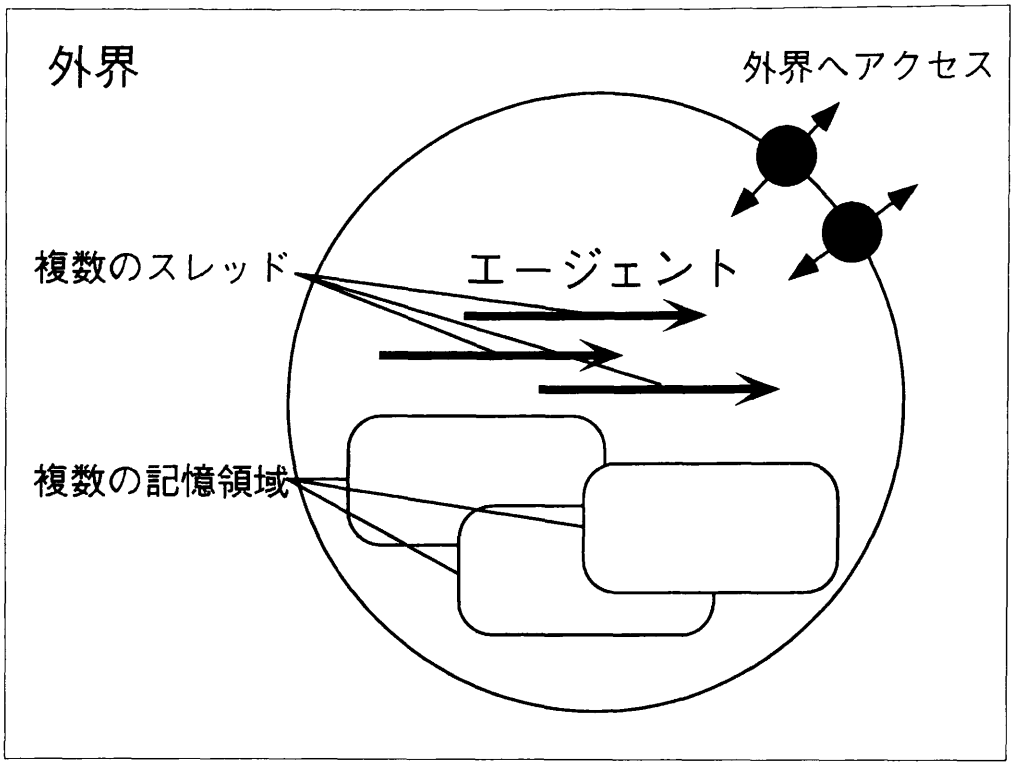

図 2.1: 本研究におけるエージェント

本エージェントの設計方針は，エージェントの基本的な構成を単純にすることである．すなわち，様相の管理方法などはここでは定義しない．エージェントの基本的な構成を単純に保つことによって，エージェントモデルの設計を容易にするのがねらいである．

このようなエージェントを実現するために，どのような機構を実装しなければならないかについて考察する．2.1.5 節における（1）の自律性を実現するためには，(a) 複数のエージェントが並行動作するための機構，(b) エージェントの状態を他のエージェントが勝手に変更できないようにする機構，(c) エージェントが自己管理できるようにするための機構，の 3 つの機構が必要である．(a) の機構は，複数の自律したエージェントが，並行動作しなければマルチエージェントにする意味が無いので必要である．(b) の機構は，エージェントの状態を任意のエージェントが変更できると，エージェントプログラミングにおいて他のエージェントからの状態の変更を考慮にいれる必要があり，エージェントプログラミングが困難になるので必要である．例外的な手段を用いてエージェントが他のエージェントの状態を変更することは認める．(c) の機構は，自律性を実現するために一番必要な機構である．

(a) の機構を実現するためには，多重プログラミングの技術を用いる．(b) は，エージェントが他のエージェントの状態を変更できるような手段を提供しないことによって実現する．(c) の機構を実現するためには，エージェントが自己の状態を参照・変更できるようにするための機構が必要である．この機能を利用するために，リフレクションための機構を用いる．

2.2 制約論理型言語

制約論理型プログラミング (CLP: Constraint Logic Programming) は形式的な基礎をもったプログラミング手法である [38]．制約論理型言語としては，[38][39] 等が挙げられる．これらの言語は，問題を制約として与えることによって，これらの言語の持つ制約ソルバーが問題を解く．本節では制約論理型プログラミングについて説明する．

2.2.1 制約論理型プログラミングとは

情報数学は，眼の前にあるさまざまな問題を解決するためにコンピュータをいかに使うかという問題を解決するために発展してきた．最適化（optimization）あるいは数理計画（mathematical programming）とよばれる分野は，対象とする問題に対する最適な解決策を求めるための定式化と手法を扱う．すなわち，解決すべき問題を「与えられた制約条件の下で，ある目的関数を最大（少）にする解を見出す」という最適化問題の形に記述し，数学的に厳密なアルゴリズムを用いて解くというアプローチをとる．やや広くは，オペレーションズリサーチ (OR) の一部とも考えられている．いわゆる問題解決手法には，オペレーションズリサーチや数理計画の他にも，人工知能 (AI)，エキスパートシステム，システム理論，ファジィ集合，ニューロ，遺伝的アルゴリズム (GA) いろいろなパラダイムが提案され，その実用化が計られてきた．これらのアプローチはそれぞれ独自の特徴をもち，問題解決のある面を受け持っているが，決して数理計画の役割を否定するものではない．解決すべき問題が最適化問題に記述されているとき，数理計画ほど深く対象を解析し，精密な解を提供してくれるものは他にないからである．数理計画という分野の誕生は，1974 年，G. B. Dantzig が線形計画法の基本アルゴリズムであるシンプレックス法を提案したときとされている．すでに半世紀近くの歴史があり，学問的にも成熟を見せているが，一方まだ着実に成長を続けている生きた分野であることも述べておきたい．この意味で最近大きな

話題となったのは，1984 年の N. Karmarkar の発表に始まる内点法である．内点法に刺激されたシンプレックス法の進歩もあって，現在数万から数十万の変数をもつ線形計画問題が実用的に解かれるようになっている [40]．制約論理型プログラミング：CLP(Constraint Logic Programming) は，宣言型の論理型言語と効率的な制約解決器が結合した新しい種類のプログラミング言語である．CLP の言語機能を用いることによって，より見易く，より柔軟なプログラムを作ることができる．論理型言語において制約が導入されたとき，制約解消系が導入された．トラディショナルな論理型言語における制約は，項どうしの同値関係だけである．そして単一化アルゴリズムが項どうしの同値関係という制約を解消するのに使われた．単一化には２つの見方がある．１つは，等式が解かれたかどうかを私達に伝える．２つめは，解が存在するときに，元の項との論理的な一致関係より一般的な解を得ることである [41]．

一般的には，以下のような領域が扱われる．

- 論理式

  論理における制約には項どうしの同値関係がある．評価中の項とデータベース内のプログラムのヘッドとのマッチングが主な制約になる．また，明示的に項の同値関係を表わすこともある．項どうしの同値関係はユニフィケーション (unification) によって求められる．

- 数式

  制約として扱われる領域に数に関する領域がある．数といっても整数，実数，虚数など多々あるが特に実数を扱う制約解消系が多い．なぜなら実数の扱いが最も簡単だからである．制約として扱われるのは方程式と不等式である．制約は大きく分けて線形な制約と非線形な制約に分けられる．線形な制約にはシンプレックス法を代表する高速にかつ精度良く問題を解くためのアルゴリズムがよく知られているので，計算精度を高めるために線形な制約だけを扱うのが一般的である．非線形の制約は線形になるまで評価を遅らせるのが一般的である．評価できるようになるまで評価を遅らせることを遅延評価 (defered evaluation) といい OS における資源管理などで良く用いられるプログラミング技法である．

  – 線形な制約

線形な制約とは一次以下の項からなる方程式，または不等式のことをいう．線形な制約からなる問題を線形制約問題という．線形制約問題 (Linear Program) という．線形制約問題に関する解法は良く知られている．線形制約問題の代表的な解法はシンプレックス法 (Simplex Method) である．シンプレックス法を用いることによって線形制約問題を効率良くかつ精度良く解くことができる．シンプレックス法のほかには内点法がある．

- 非線形な制約

  線形な制約でないものを非線形な制約という．非線形な制約に関する解法は線形な制約に関する解法よりも一般的に精度が悪い．よって非線形な制約は線形な制約になるまで遅延評価することが多い．

2.2.2 制約解消系

制約論理型言語における制約解消系 (Constraint Solver) は制約論理型言語のバックトラックメカニズムに対応するためにスタックを用いた制約の管理を行う必要がある．インターフェースは論理型言語のインタープリタとのインターフェースである．制約変換プログラムは論理型言語のインタープリタから渡された制約を制約解消プログラムに適したデータ形式に変換するプログラムである．スタックには制約が格納される．スタックを用いることによって論理型言語のバックトラックに対応している．制約解消プログラムが実際に制約を解く部分である．

2.3 リフレクション

エージェントの自律性を実現するためには，環境に適応するための機能をエージェントが持つ必要がある．そのような機能を実現するために，エージェント自身が自己の状態に関する情報を得て，自己の状態を適した状態にするための機能が必要である．RXF ではエージェントの自律性の実現のためにエージェントのためのリフレクション機能を実現した．リフレクション（自己反映，または自己反映計算とも言う）とは，計算システムが，自分自身の構成や計算過程に関する計算を行うことである [42]．また，一般の計算システムにおけることを強調する意味で計算的リフレクション (computational reflection) と呼ぶこともある．リフレクションの能力を持つシステムをリフレクティブなシステムと呼ぶ．リ

フレクティブなシステムは，システム自身の挙動を観察したり，変更することが可能である．一般に自分自身の構造や行動を観測する能力は内省 (intorospection) と呼ばれる．内省の能力とは，システムが自己の構成・計算過程をモデル化した表現を内部に持ち，その表現を参照できること（自己参照）といえる．リフレクティブなシステムでは，自己の構成・計算過程の表現を参照することだけでなく変更することもできる．そしてそのような表現に対する変更は，自己の構成・計算過程そのものに反映される．後者（自己改変）を，狭い意味でのリフレクションと呼ぶこともある．

もともとリフレクションの概念は，AI 分野において研究されていた．雨の中道を歩いている人がいたとすると，その人は自分の体が濡れ始めてきたのを観測し，傘を差すべきであると考えるだろう．この例は，人が自分自身の状態を観察することによって自分自身の行動を変更することを表している．例えば，自己の状態を認識し，自己の行動を変えることのできるシステムの実現のために研究されてきた．このような人間の行動を実現するために，AI 研究では自己の状態を認識し，自己の行動を変えることのできるシステムの実現のために研究されてきた．研究例として，(1) 知識表現の立場から，主にロジックを基礎とした自己参照形式について研究が行われてきたことや，(2) 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・自己改変が積極的に利用されてきたことが挙げられる [42]．前者は Perlis による自己参照言語の研究 [43] が，後者は FOL[44] や TEIRESIAS といったシステムが有名である．

1982 年に B.Smith は一般的な計算システムにおけるリフレクションを手続き的リフレクション（procedural reflection）という形で定式化し，その実例としてプログラミング言語 3-LISP を定義した．その後，P.Maes によってオブジェクト指向プログラミング言語における（手続的）リフレクションの有用性が示され，以降，ソフトウェアの構成原理としてのリフレクションが注目されてきている．これは以下のように，プログラミング言語やオペレーティングシステムといった記述系をリフレクティブなシステムとして構成することが，柔軟なソフトウェアの構築に有用であることが認識されているためである．リフレクションを用いることによって得られる特徴として，(1) 拡張性，(2) 動的適応性，の 2 つが挙げられる．(1) の拡張性とは，言語や OS を，アーキテクチャと実現方式の両方の面でカスタマイズできるようにすることである．アーキテクチャのカスタマイズにより，アプリケーションの持つ論理構造と言語・OS の提供する抽象化機構とのギャップを少なくすることができる．一方，既存のデータ表現によって対象の持つ論理構造を素直に表現できるに

もかかわらず，実行効率が著しく悪くなる場合がある（例えば巨大で疎な行列は，普通は２次元配列としては表現しない）．特定のデータ表現の実現方式をカスタマイズすることによって，このような効率の低下を避けることができる．(2) の動的適応性は，計算機の状態を動的に変更できるようにする性質である．計算資源の最適配置や制御方式を，静的解析によって求めることが困難な問題（例えば分散離散事象シミュレーションにおいて，シミュレート対象の挙動が予測困難な場合），マルチエージェントシステム，そしてモーバイルコンピューティング環境のように構造が動的に変化するような計算過程にたいしては，自己の構造や計算過程を適切に操作し，計算環境に動的適応できるようなソフトウェアの構成方式が求められる．リフレクティブな言語やOSは，このようなソフトウェアを自然な形で記述できる．

(1) (2) の２点に共通するのは，本来目的とする計算 (subuject matter) のレベル（ベースレベル）と，その計算・記述方式を制御またはカスタマイズするレベル（メタレベル），そしてその間のインタラクションが同一の記述系に基づいていることである．リフレクティブな言語やOSのアーキテクチャが適切に設計されていれば，これらのレベルを独立したモジュールとして分割し，再利用することが可能となる．このような（ベースレベルとメタレベルの分離という観点での）モジュール化方式は，従来のプログラミング言語の設計やプログラミング方法論においてはほとんど意識されることはなかった．しかしこれによって，従来 ad hoc な方法で導入されてきた機能（資源管理，例外処理，デバッグ機能，外界とのインターフェースなど）を，言語の提供するアーキテクチャに基づいた整合的な方式で導入できる．つまりリフレクションとは，新しいモジュール化手法を与えるソフトウェアの構成原理とみなすことができる．

2.3.1 リフレクションの原理

計算システムの本質は，対象とする問題領域（problem domain）に含まれるさまざまな具体化・概念的実態（entitiy）を，データや手続き，オブジェクトなどの形式 - 実態の表現（epresentation）- によってモデル化し，それら実体のふるまいをシミュレートすることにある．実体の表現はある言語に基づいて行われる．ここで言語とは，いわゆるプログラミング言語を含む，より広い意味でのソフトウェアの記述体系を指す．ある計算システム（S とする）において，実体の表現が言語 L に基づいている場合，S を L に基づく計算システムとよび，S における計算を L に基づく計算と呼ぶ．言語 L に基づく計算システムは，L

のプロセッサPLとプロセッサの状態を表わす項TLの組み¡PL,TL¿として表現することができる．計算はTLの書き換えとしてモデル化される．¡PL,TL¿における実体eに関する計算とは，TL内部におけるeの（言語Lによる）表現Reに関する書き換え操作にほかならない．Lに基づく計算システムS(=¡PL,TL¿)がリフレクションの能力を持つということは，以下のように述べることができる．

1．Sは自分自身をモデル化したLによる表現RsをTLに含む．
2．一般にシステムの状態は計算の進行に応じて変化するため，RsもSの状態を適切に表現するように変化する（内省の能力）．さらに，S内でRsに対する（変更を伴う）操作が行われたとき，このことがSに反映することが必要である（狭義のリフレクション）．

特に2のようなSとRsの関係は因果的結合 (causal connection) と呼ばれる．Sのリフレクションの能力は，RsがSのどの部分をどのような形式で（どの程度精密に）表現しているか，またSでRsに対してどのような操作が許されているかに依存する．RsがSをある程度精密に表現すれば，Rs中には必然的にRs自身を表現する部分が含まれることになるため，これによってSは自分自身のリフレクションの機能に関するリフレクションの機能を持つことが可能になる．

2.3.2 手続的リフレクション

ある計算システムMの問題領域が他の計算システムSを含んでいるとする．さらにMがその内部にSの表現RSを持ち，これによってSのふるまいをシミュレートしているとき，これによってSのふるまいをシミュレートしているとき，MはSのメタシステムと呼ぶ．M，Sがそれぞれ言語LM，LSにもとづくシステムであり，MがSのメタシステムであるとき，Mにおける計算は次のように2通りにとらえることができる．

1．M 本来の， LM にもとづく計算．
2．シミュレートされる S としての（LS にもとづく）計算．

1を2のメタレベルの計算，2を1のベースレベルの計算と呼ぶ．Mにおける Sの表現RSは，Mでは自由に参照・変更が可能であり，変更はシミュレートされるSに反映される．MとSが同一の言語Lにもとづくシステムである場合を考える．このときMはLのメタインタプリタと呼ばれる．このようなMを ↑S と表記することにする．Sは ↑S

によって解釈実行されているが，同一の言語 L を用いているため，TL の一部を↑S のレベルで実行することも可能である．このことをメタレベル実行 (metalevel execution) と呼ぶ．ベースレベルの計算の一部分をメタレベルで実行するような枠組みが言語 L に備わっているとする．これによって間接的にではあるが，S から RS を参照・操作するためのメカニズムを構成できる．S は RS を通して↑S によって解釈実行されているので，RS に対する変更は，S 自身に反映する．すなわち，S と因果的に結合した RS を得ることができるのである．↑S においても言語 L にもとづく計算が行われている．つまり，↑S においてさらにメタレベル実行をすることができることを意味していりる．↑S におけるメタレベル実行を実現するには，↑S のメタシステム↑↑S を導入する．同様にして，言語 L のメタインタプリタの無限の階層を構成することにより，任意のレベルの表現を参照・操作できるようになる．このようなメタインタプリタの無限の階層をリフレクティブタワーと呼ぶ．解釈実行の手続きとしてメタレベルを構成し，リフレクションを実現する方法を手続的リフレクションと呼ぶ．現在この方法はリフレクティブなシステムの実現方法として一般的に用いられている．実際に，計算機上に実装する場合は，リフレクティブタワーを有限の計算資源によって実現する必要がある．これには必要なレベルを動的に生成する方法がある．また，実際の応用には有限の階層ですむことが多いため，モデル上は無限のタワーが存在しても有限の階層だけを実現することもある．

リフレクションの具体例として，B. Smith による 3-Lisp について説明する．3-Lisp では，メタレベル実行の枠組みとしてリフレクティブ手続き (reflective procedure) と呼ばれる特別な形式を提供している．また quote の扱いも普通の Lisp とは異なり，式の評価はリダクション規則として厳密に定義される．リフレクティブ手続きの構文は，次のように Lisp のλ-式に REFLECT というキーワードを付加した形をしている．

```
(lambda REFLECT (args env cont) body)
```

リフレクティブ手続きは，普通の関数と同様に，任意の引数に対して適用することができる．上のリフレクティブ手続きが呼び出されると，body はメタレベル実行される．このとき，引数 args, env, cont には，リフレクティブ手続きを呼び出したレベルの，未評価の引数 (S-式)，環境，継続がそれぞれバインドされる．3-Lisp では，リフレクティブ手続きを呼び出すことによって，3-Lips プロセッサ自身の状態を表わすデータ (式，環境，継続) を参照できる．

2.3.3 オブエジェクト指向におけるリフレクション

オブジェクト指向型言語には，クラス，メッセージ，メソッド，総称関数などを，オブジェクトとして統一的に実現しているものがある．このような，言語・実行モデルの諸概念を実現しているオブジェクトをメタオブジェクトと呼ぶ．メタオブジェクトは言語のメタインタプリタの実現であり，メタオブジェクトの導入によって，オブジェクト指向言語は手続的リフレクションを実現しているといえる．

CLOS では，メタオブジェクトのインターフェースの使用をメタオブジェクトプロトコルという比較的使いやすい形で提供している．メタオブジェクトプロトコルでは，メタオブジェクトのクラス定義そのものは与えずに，それらのクラスに関係する総称関数および付随するメソッドの (自然言語による) 外部使用を与えている．つまり具体的な実装からは独立にメタオブジェクトの使用や利用方法を与えている．また，メタオブジェクトのカスタマイズは，継承を用いることにより局所的に行うことができる．これによって，カスタマイズしたシステムの安全性や再利用性を確保している．

2.3.4 並行・分散計算とリフレクション

並列計算機や分散システムを効率よく利用するためには，適切に計算資源管理を行うようにソフトウェアを構成する必要がある．ここで計算機資源管理として，プロセッサ配置・通信機構・記憶管理・入出力などの管理が挙げられる．通常これらの資源は，オペレーティングシステムによって管理される．しかし，より高い実行効率を要求する場合，分散システムにおける資源管理は逐次型計算システムと比較すると一般に複雑なのでオペレーティングシステムによる管理では不十分な場合があり，ソフトウェアに最適化された管理が必要になる．計算資源管理はアプリケーションの計算過程の制御をおこなうわけであり，アプリケーションからみてメタレベルの計算を行っていると考えられる．この点から，アプリケーションにあわせた計算資源管理機構をモジュラーな形で提供する枠組みとして，並行・分散計算システムにおけるリフレクションが研究されている．計算資源管理に関するリフレクションをおこなうためには，まず計算資源をアプリケーションと同じ言語にもとづいてモデル化する必要がある．一般に計算資源は，言語のモデルには陽に表れず，また多くの異なった形式化が可能であるため，抽象化能力の高いオブジェクト指向パラダイムに基づくモデル化が行われることが多い．

2.4 デバッグ

ここでは，デバッグを（1）逐次プロセスに対するデバッグ，(2）並行プロセスに対するデバッグ，そして（3）マルチエージェントシステムに対するデバッグ，の 3 つに分けて概観する．逐次プロセスとは，命令が 1 つずつ順番に実行されるプロセスである．並行プロセスは，複数の逐次プロセスが同時に並行動作するシステムである．マルチスレッドシステムや分散システムは，並行プロセスである．

逐次プロセスや並行プロセスのプログラミングと同様に，マルチエージェントシステムの開発においてもデバッグは必要不可欠である．一般的に，逐次実行するプログラムのデバッグよりも，並行実行するプログラムのデバッグの方が困難である．マルチエージェントシステムは，複数のエージェントが並行動作するので，並行プロセスと考えられる．マルチエージェントシステムにおけるデバッグでは，並行実行するプログラムをデバッグする必要がある．さらに，マルチエージェントシステムのデバッグにおいて，知識処理などの並行プロセスとは違う分野も扱う必要がある．しかしながら，マルチエージェントシステムのデバッグに対する系統だった研究はあまり知られておらず，今後の研究が望まれる．

2.4.1 逐次プロセスのデバッグ

プログラムを正常に動作させるための作業を大きく分けると，プログラム試験，プログラム検証，そしてデバッグに分けられる．プログラム試験とは，仕様に対するプログラムの誤りを発見することを意味する．プログラム試験とは，エラーを見つけるつもりプログラムを実行する過程である [45]．プログラム検証は，形式的な技法を用いてプログラムの正当性を証明することを意味する．プログラム検証とは，エラーが無いことを証明することである．デバッグは，正しく動作しないプログラムの不具合を修正することを意味する．デバッグとは，エラーのあるプログラムからエラーを取り除く作業である．プログラム検証は，高価であり，大規模なプログラム全体に適用することは困難なので，実用的な技法としてはソフトウェア試験が用いられている．

プログラム検証では，プログラムの形式的意味を定め，それに対応した推論規則を用いてプログラムの正しさを証明する．プログラム試験では，プログラムに適当な入力を与え，正しく動作するかを調べる．プログラム試験とデバッグの関係を簡単に説明すると，プログラム試験はバグの存在を確認するための作業で，デバッグはバグを取り除くための作業

であるといえる．

デバッガは，ソフトウェア開発に欠かせないツールである．デバッガは，コンパイラなどの開発ツールに比べると，あまり研究されていない[46]．しかし，デバッグには，多くの時間が費やされているのも事実である．デバッガとは，ソフトウェアプログラムのバグを追跡し，特定し，排除する作業を支援するためのツールである．バグとは，ソフトウェアの欠陥である．ソフトウェアの欠陥は，Harvard Mark I コンピュータのプログラムに誤った振る舞いを生じさせた悪名高き蛾にちなんで，それ以来バグと呼ばれている[47]．実際には，デバッガはプログラムの動的な性質を明らかにするツールであり，欠陥を見つけて修正するだけでなく，プログラムの挙動を理解するためにも用いられる．

デバッグには，静的デバッグと動的デバッグがある．静的デバッグとは，ソースプログラムの構造や使っている名前に着目してバグの手がかりを得るデバッグ方法である．静的デバッグ手法として，プリティプリンタやクロスレファレンスが挙げられる．プリティプリンタは，ソースプログラムを見やすく整形するためのツールである．プリティプリンタは，プログラムの構文に従って字下げしたり，キーワードや予約後を太字にするなどの機能を持つ．クロスリファレンスは，プログラム中の相互参照を出力するプログラムである．相互参照とは，プログラム中に出現する記号について，それを定義している場所と参照している場所の関係を意味する．動的デバッグとは，実際にプログラムを実行しながらバグの除去を支援するデバッグ方法である．動的デバッグ手法として，トレーサ，ステッパ，そしてインスペクタなどが挙げられる．トレーサは，プログラムの実行過程を追跡するためのツールである．ステッパは，ステッパは，プログラム言語の構文単位，すなわち文や関数ごとに実行を中断しながらプログラムの実行過程を追跡するシステムをいう．トレーサとの違いは，ステップは単に実行の軌跡を追跡するだけでなく，中断したときにプログラマが指示すればプログラムの状態（変数や引数の値や関数の返値など）を表示する機能を備えている．インスペクタは，プログラムの実行を中断したときに，データ構造やデータの内容を対話形式で調べたり，変数の値を変更して実行状態を変更するシステムである．これらの動的デバッグ手法を実現するための手法として，ブレークポイントが挙げられる．ブレークポイントは，特定の条件を満たした逐次プロセスの実行を停止するために用いられる．条件として，特定の命令を実行したときや特定の変数の値を変更したときなどが挙げられる．ブレークポイントについては，文献 [46] に詳しく述べられている．これらの動的デバッグ手法を利用した動的デバッグシステムとして，ソースプログラムと対応付けなが

らデバッグすることを可能にするシンボリックデバッガもある.

2.4.2 並行プロセスのデバッグ

並行プロセスとしてプログラムを記述することには，逐次プロセスとしてプログラムを記述することに比べて,(1) 処理の高速化，そして (2) 記述の簡潔さ，の 2 つの利点がある.(1) の処理の高速化は，順次行う処理を同時に並行して行うことが可能なので，全体としての処理時間を短縮できる可能性があることを意味する.(2) の記述の簡潔さは，プログラムの記述において実行の条件を並行に行われる処理ごとに記述するので記述すべき条件が少なくなるということを意味する．一方，並行プロセスを適切に動作させるためには，並行に動作するプロセスを適切に制御するための機構が必要である．制御が適切でないと逐次プロセスよりも処理が遅くなる可能性がある．さらに，個々のプロセスは並行に動作する他のプロセスとの間での相互作用があり，その相互作用が適切でなければならない．相互作用が適切でないと，プログラムそのものが動作しなくなったり，計算結果が矛盾するという誤りを生じる可能性がある．すなわち，プログラムを正しく動作させるという点において逐次プロセスよりも考慮すべき点が多く，並行プロセスを記述したプログラムを正しく動作させるのは逐次プロセスを記述したプログラムを正しく動作させるより困難である.

並行プロセスの誤りは,(1) 計算誤り,(2) 通信誤り，そして (3) 同期誤り，の 3 つに分類される [48].(1) の計算誤りは，逐次プロセスにおける計算に関する誤りである.(2) の通信誤りは，2 つのプロセス間でのデータの受け渡しに関する誤りである．逐次処理プログラムでは，手続きや関数の呼び出し時に発生し，並行処理プログラムでは，プロセス間の通信において発生する.(3) の同期誤りは，プロセスの実行順序が不適切であることによって発生する誤りであり，並行処理プログラム特有の誤りである．並行プロセスの誤りには，逐次プロセスにはない特徴があるので，並行プロセスを仕様通りに正しく動作させるための支援ツールが望まれている.

並行プロセスを正しく動作させるための支援ツールの分類は，基本的に逐次手プロセスを正しく動作させるためのツールと変わらない．すなわち，プログラム検証，プログラム試験，そしてデバッグのためのツールに分類される．プログラム検証は，逐次プロセスの場合と同様に，計算量の高さから実用的な状況が限られる．並行プロセスのプログラム試験は，困難である．理由は,(1) プログラムの状態数の増大さ，そして (2) プログラムの

正しさあるいは誤りの定義が困難，の2点が挙げられる [48]．(1) のプログラムの状態数の増大とは，並行プロセスの状態は逐次プロセスの状態の組み合わせなので，並行プロセスの状態はプロセスの数やプロセスの状態数に対応して組み合わせ論的に増大する．すなわち試験すべき状態の数も増大し，効果的な試験を行うためのデータ集合の生成が困難になる．(2) のプログラムの正しさあるいは誤りの定義が困難は，並行プロセスの状態数が膨大なために，正しい状態と誤り状態を明示的に定義することが困難であるという問題である．これは，試験結果の正誤の判定を困難にする．また，特定の誤りを発見するための試験データの作成も困難になる．並行プロセスのデバッグも，逐次プロセスのデバッグに比べて困難である．並行プロセスのデバッグにおける主な問題は，(1) 複雑さ，(2) "probe effect"，(3) 再現性の欠如，そして (4) 大域時刻の欠如，の3点が上げられる [49]．(1) の複雑さは，並行プロセスの状態数の多さに関係している．デバッグにおいて，プログラムの実行状態の把握が必要である．並行プロセスの状態数の多さは，プログラムの実行状態の把握を困難にし，効率的なデバッグを妨げる．また，並行プロセスにはメッセージ通信や資源共有などで発生するデッドロックなどの並行プロセス特有の問題も存在し，並行プロセスのデバッグを困難にしている．(2) の "probe effect" [50] とは，システムの振る舞いを観察することが，その振る舞いを変えてしまうことを意味する．すなわちデバッグすることがシステムの動作を変えてしまい，非デバッグ中に出現したバグが，デバッグ中に出現しなかったりすることもある．(3) の再現性の欠如とは，同じプログラムが同じデータを用いて実行した場合に，異なる結果得られる可能性があることを意味する．逐次処理プログラムでは，同じ入力データに対して出力データは基本的に一定であるけれども，並行処理プログラムではプロセスの実行のタイミングや実行環境，順序付けアルゴリズムによって出力データが異なる可能性が高い．これは，デバッグにおいて重大な問題となる．並行プロセスの非決定性により，一度バグを発見しても，そのバグが再現されるかどうか解らない．すなわちバグの特定が困難になり，デバッグの妨げとなる．(4) の大域時刻の欠如は，システム全体で同一な時計を仮定することができないという問題である．大域時刻を実現する直接的な方針は，書くプロセスが保持する局所時刻をメッセージ通信によって同期させることであるが，実際には，完全な同期は不可能である [51]．大域時刻の欠如により，(2) の "probe effect" を発生させずに並行プロセスの振る舞いが観察できたとしても，観察結果を解釈することが困難であるという問題を生じさせる．例えば，観察された2つの事象の時間に関する前後関係を特定することが困難になる．

並行プロセスの記述方法およびデバッグ方法については，これまで多くの議論がなされている [49]．文献 [49] で述べられているデバッグ技術の研究では，並行プログラムの非決定性や “probe effect” の回避のための研究が行われてきた．並行プログラムのためのデバッグ技術は，(1) 逐次プロセスのデバッグ技術を応用したデバッグ技術，(2) イベントに基づくデバッグ技術，(3) 並行プロセスの制御やデータの流れを表示する技術，そして (4) 並行プロセスの静的解析，の 4 種類に分類される．(1) の逐次プロセスのデバッグ技術を応用したデバッグ技術は，逐次プロセスで適用されてきたデバッグ技術を並行プロセスに適用することを意味する．これには，実装の容易さ，計算コストの低さ，そして理解のしやすさなどの利点があるが，“probe effect” や再現性の欠如などの並行プロセスのデバッグにおける本質的な問題に対するアプローチとしては不十分である．しかしながら，プログラムの実行制御や状態調査には役に立つ．(2) のイベントに基づくデバッグ技術では，並行プロセスの動作をイベントの系列として抽象化し，イベントの系列を解析することによってバグを発見する．このアプローチは，並行プロセスのデバッグの本質的な解決を目指して研究されている．並行プロセスの挙動をイベントとして完全に記録できれば，プログラムの実行を再現することができる．すなわち，非決定的な並行プロセスのデバッグに，決定的な再実行機能を実現することができる．(3) 並行プロセスの制御やデータの流れを表示する技術では，並行プロセス全体の実行状態やデータの依存関係をプログラマにわかりやすく表示することによってデバッグを支援する．(4) 並行プロセスの静的解析では，プログラムを静的に解析することによって，プログラムを動作させずにデバッグを行う．このアプローチでは，プログラムを実行しないので，“probe effect” や再現性の欠如などのプログラムの実行時に発生する問題を完全に回避することが可能になる．しかし，解析アルゴリズムの計算コストが高いために，応用にはさらなる研究が必要である．

2.4.3 マルチエージェントシステムのデバッグ

現在，マルチエージェントシステムのデバッグとして系統だった研究は知られていない．ここでは，RXF を例にしてマルチエージェントシステムのデバッグにおいて必要と考えられる機能について考察する．マルチエージェントシステムは，並行システムである．よって，並行システムと同様なデバッグの問題がある．例えば RXF の場合，複数のエージェントが並行実行するのに加えて，1 つのエージェント内でも複数のスレッドが並行動作している．RXF におけるデバッグにおいて，1 つのエージェント内の複数のスレッドの同期に

関するデバッグや，複数エージェントの相互作用に関するデバッグを行う必要がある．さらにエージェントの自律性を実現するための機能（例えばリフレクション機能）を用いたプログラムのデバッグも困難である．なぜならば，リフレクションにおいてはプログラムの実行が階層化されるが，デバッグ時にプログラマがデバッグ対象のプログラムの動作に注意しつつ，そのような階層の状態を意識することは大変な負担になり，デバッグの妨げとなる．結果として，リフレクションを用いたプログラムは記述する時点では有用であるが，いざデバッグとなると障害になる可能性もある．

マルチエージェントシステムのデバッグにおいて必要なデバッガを以下のように定義する．

- 命令レベルデバッガ（Instruction level debugger）
  逐次プロセス用のデバッガである．変数の値の内容や，実行状況に関するエラーの発見を支援する．

- 知識レベルデバッガ（Knowledge level debugger）
  知識レベル [52] におけるデバッガである．推論機構や学習機構が正常に動作しているかを調べ，不具合があればその原因を発見するために用いる．エージェントの心理状態（mental state）の整合性のチェックなども行う．

- 通信レベルデバッガ（Communication level debugger）
  並行プロセス用のデバッガである．並行プロセス間の資源競合や，メッセージ通信に関するエラーの発見などを支援する．

- 協調レベルデバッガ（Cooperation level debugger）
  マルチエージェントシステムにおける協調や，交渉プロトコル，チームにおける役割などの整合性を調べるために用いられる．

命令レベルのデバッガは，2.4.1 節で述べたデバッグを支援する．通常の開発環境では命令レベルのデバッガは，提供されている．知識レベルのデバッガにおいては，推論エンジンの動作の不具合を発見することや，エージェントモデルを用いてエージェントの不具合を発見することなどが考えられる．知識レベルのデバッガを開発終了後のエージェントに組み込むことにより，想定外の環境の変化による不具合の修正を支援できることが望ましい．通信レベルのデバッガは，2.4.2 節で述べたデバッグを行う．デッドロックなどの並行プロセス特有の問題の修正を支援する．協調レベルのデバッガは，エージェントの協調に

おける不具合の修正を支援する．交渉プロトコル，チームの意図などのマルチエージェントシステムに特有の機能が正常に動作していることの確認の支援や，不具合箇所の発見を支援する．

RXF におけるデバッガは，命令レベルデバッガと通信レベルデバッガである．知識レベルデバッガや協調レベルデバッガには，それぞれエージェントモデルやチームモデルなどが必要と考えている．RXF では特定のエージェントモデルやチームモデルを提供しないので，現状において RXF ではこれらのレベルにおけるデバッグを支援しない．

2.5 事例ベース推論

事例ベース推論とは，過去に解いた問題とその解を問題解決事例として保存し，後の問題解決に役立てる推論手法である [53, 54]．図 2.2[54] を用いて事例ベース推論の処理過程について説明する．図 2.2 の中央の四角は，知識ベースを表している．この知識ベースには，汎用の知識と過去の問題解決事例が格納されている．過去の問題解決事例を格納している部分を事例ベースと呼ぶ．知識ベースの周りの円が事例ベース推論の実行サイクルを表している．実行サイクルは，検索，再利用，修正，そして格納の 4 段階から成る（図 2.2 の弧上の網掛け部分）．円上の四角は事例を表している．最初に問題が与えられる．与えられた問題は，新規事例として適切な形式に変形される．検索段階では，新規事例の解決に適切な過去の問題解決事例（検索事例）の検索が行われる．通常，新規事例に類似した事例が事例ベースから選択される．ここでいくつの事例を事例ベースから選択するかは，システム依存である．単一の事例だけでは問題解決が困難な場合があり，そのような問題に対して複数の事例を用いた問題解決が行われる [55]．検索事例は，再利用段階によって，新規事例の解決に適するように変形される．再利用段階において得られた事例を問題解決事例と呼ぶ．事例ベース推論システムは，問題解決事例をユーザに提示したり，システム内で検証する．その結果，問題解決事例が適切であれば，次の修正段階を飛ばして，実際の問題に問題解決事例を適用し，格納段階へと飛ぶ．適切でないときは，修正段階により問題解決事例が修正される．ここで得られた事例を，修正された事例と呼ぶ．修正事例は，再び検証され，適切になるまで修正と検証を繰り返す．問題解決に用いられた事例は，格納段階において，事例ベースに格納される．格納段階により，事例ベース推論機構における学習が達成される．この段階における事例は，学習された事例と呼ばれる．

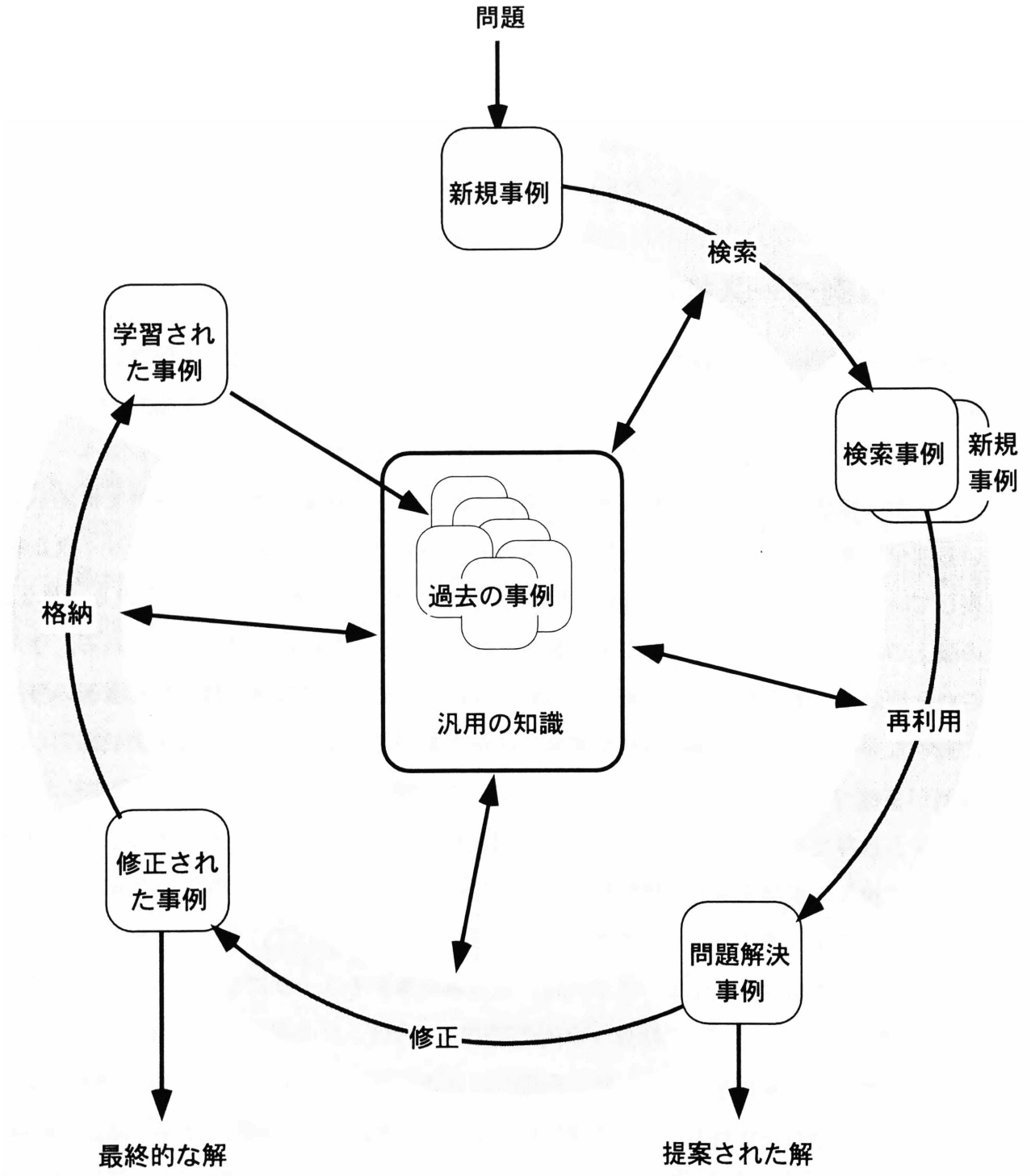

図 2.2: 事例ベース推論における推論過程

本研究では，階層的事例ベース推論システムの研究を行う．階層的事例ベース推論システムとは，事例ベース推論システムを階層的に適用することによって，事例ベース推論システムの処理能力の向上を目指す物である．

階層的事例ベース推論システムの例としては，Leake 等の研究 [56] や Bonzano 等 [57] の研究が挙げられる．

Leake 等は，複数の領域が重なり合った問題に適用された事例ベース推論システムにおいて，それぞれの領域について独立に学習することを可能にするための手法を提案した．ここでは，事例適用方法を学習するという事例ベース推論処理に対してメタな処理が行われている．さらに，事例適用の学習に従って類似性基準を調整するというメタな処理により事例適用の学習の効率を高めている [56]．

Bonzano 等は，航空管制支援システムの実装に階層的事例ベース推論システムを適用している [57]．航空管制支援システムでは，航空機のコンフリクトを解消することを支援する．ここで，3 機以上の航空機に関連したコンフリクトの事例を保存することは，その組み合わせの多さより困難であるという問題がある．Bonzano [57] は，3 機以上の航空機に関連したコンフリクトを，2 機に関連したコンフリクトの解消のための事例とメタ事例を用いて解消している．ここでのメタ事例は，2 機に関連したコンフリクト解消事例の利用の仕方に関する事例である．これらの事例を階層的に用いることによって，事例保存のための空間と時間を大量に節約しつつ，同じ事例ベースを 2 機の航空機のコンフリクトの解消と多数の航空機のコンフリクトの解消の両方で共有可能になる．これによって，3 機以上の航空機のコンフリクトに関する事例ベース構築に必要なコストを軽減している．

事例ベースの管理に関する話題も盛んに研究されている．例えば，Portinale は，ADAPtER における utility problem に関連したいくつかの見解について述べている [58]．ADAPtER とは，事例ベース推論とモデルベース推論を結合した，診断問題の解決のためのマルチモーダル推論システムである． utility problem とは，事例メモリに事例を追加しても，システムの高性能化は実現ができないばかりか，逆にシステムの性能を落とすこともあるという問題である．ADAPtER を用いた実験により，事例メモリの増加が，問題をスクラッチから解かなければならない必要性を減らすことを確認した．それとと同時に，事例メモリのサイズの増加は，ADAPtER における utility problem の発生に大きく関係していることも解った．解決のために，事例メモリを動的に制御するための 2 つの学習戦略を提案している．2 つの学習戦略は，(1) 事例は忘れることができ，よりよい適用を保存すべきである，

(2) 最も高価な事例だけが，事例ベースに保存されるべきである，という 2 つのアイデアに基づいている．これら 2 つの戦略により，システムは，事例メモリのサイズと内容の両方を制御し続けるために，動的に事例を事例メモリへ追加または事例メモリから除去したりできる．

最近は，事例ベース推論のモデル化に関する研究も盛んである．例えば，Bergmann 等は，事例ベース推論における適用の，一般的で形式的モデルを提案してる．本モデルは，適用を事例の変形としてモデル化している．本モデルの特徴は，問題に対する解の質という考えに基づいている点である．適用知識は，ある事例を次の事例に変形する関数として定義されている．解の質を定義することによって，適用知識に意味を与え，健全性，正しさ，そして完全性のような用語を定義できるようになる．本モデル化により，適用もしくは事例ベース推論のプロセス全体でさえも，最適化問題の特殊な例と見なすことが可能になる [59]．Kontkanen 等 [60] の研究では，ベイズ確率論を用いた事例のマッチングに関するモデル化が行われている．Gómez 等 [61] の研究では，Description Logic を用いた事例ベース推論のモデル化が行われている．Fuchs 等は，領域に依存しない一般的な適用について論じている [62, 63]．

また，Bergmann 等のように，事例ベース推論の新しい処理を提案するような流れもある [64]．Bergmann 等は，Reformulation と呼ばれる事例適用の精度向上ののための処理を提案している．Reformulation では，事例を改変するだけでなく，事例改変の知識も改変することで事例適用の精度向上を達成する．

第3章　マルチエージェントシステム開発環境 RXF

RXF [5, 6, 7, 65, 66] 実装の目的は，マルチエージェントシステムの構築の支援である．RXF は，マルチエージェントシステムを構築するために必要な機能を提供する．また，RXF は，必要ではあるが繁雑なメモリ管理などの処理を行う．

RXF 上におけるプログラミング言語として，プログラミングの負担を軽減するために，制約論理型言語を実装した．制約論理型言語を用いることによって，問題を制約として記述することによって問題を解くことができる．論理型言語特有のデータベース機能によってデータの管理も容易におこなうことができ，単一化を利用することによって容易にプログラミングできる．RXF では，制約として扱える領域として，ロジック，数式，リスト，文字列が挙げられる．

プログラムを実行する方式として，コンパイル方式と，インタプリタ方式がある．コンパイル方式の利点は，実行速度である．インタプリタ方式の利点は，コンパイル方式のようなコンパイル作業が必要ないので扱いが容易であるという点である．RXF は，インタプリタ方式でプログラムを実行する．理由は，インタプリタ方式の扱いの容易さを重視したからである．プログラムの実行方式にインタプリタ方式を用いることによって，コンパイル作業を省略でき，プログラムの修正とテストを効率良くおこなうことができる．

RXF は，リフレクション機能を持つ．RXF のリフレクション機能によって，RXF 自体に柔軟性を持たせることができ，RXF のシステムの拡張をおこなうことができる．

筆者等は，Macintosh 上と Java 上で動作する RXF を実装した．Macintosh 上で動作する RXF の実装は C++言語を用いた．Macintosh 上で動作する RXF の実装は HyperCard や FileMaker などの Macintosh 上のアプリケーションと簡単に通信するための機能をも備えている．HyperCard は Macintosh 上で利用可能なオブジェクト指向プログラミング環境である．

3.1 RXF におけるエージェント

3.1.1 エージェント間の関係

RXF におけるエージェント間の関係を図 3.1 に示す．

すべてのエージェントは名前をもつ．エージェントには，標準的なエージェントと，標準的なエージェントの機能の一部を実現するためのエージェントに分類される．本論文では，後者をメタエージェントと呼ぶ．エージェントを生成したエージェントをそのエージェン

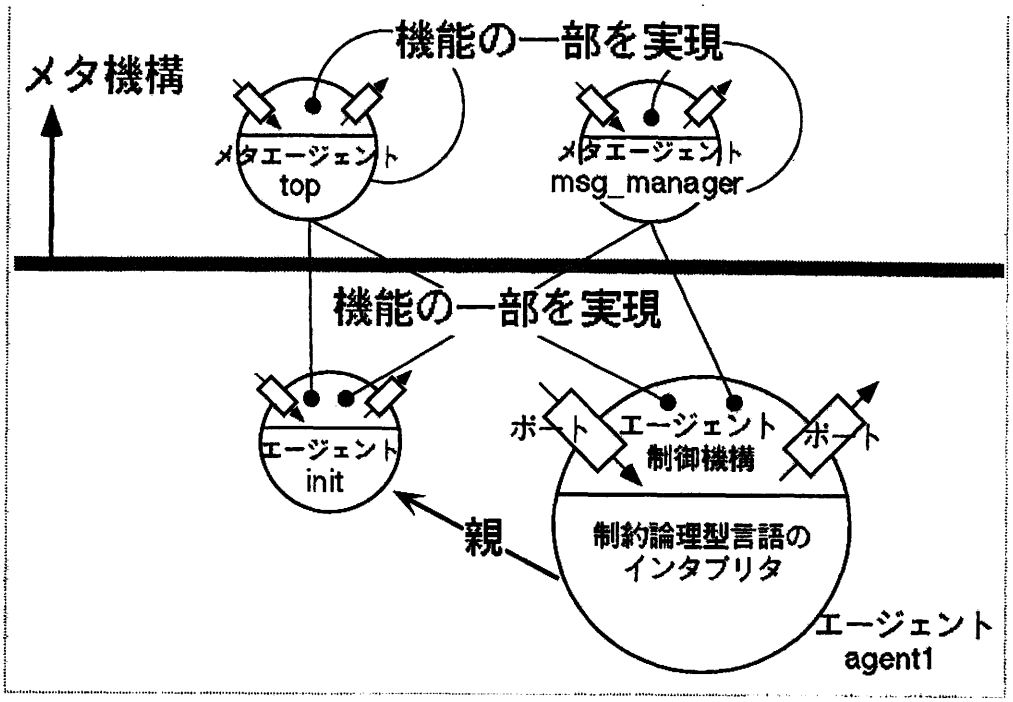

図 3.1: エージェント間の関係

トの親と呼ぶ．標準的なエージェントは，システム組込みの init と呼ばれるエージェントを親とする．例えば，図 3.1 中のエージェント agent1 は，init を親とする．標準的なエージェントとメタエージェント間には親子関係が存在しない．

システム組込みのメタエージェントとして，top エージェントと msg_manager エージェントがある．top エージェントと msg_manager エージェントをあわせてシステムエージェントと呼ぶ．システムエージェントは，標準的なエージェントに対してメッセージ通信やファイル入出力などの機能を提供するためのエージェントである．リフレクションに関連して，システムエージェント以外にもユーザが標準的なエージェントに対するメタエージェントを定義することもできる．リフレクションに関連して，問題解決のためのメタな推論をおこなうメタエージェントを定義することによって問題解決のための計算と問題解決のための計算を制御するための計算を分離することができる．

init エージェントは RXF が起動したときに 3 番目に起動される．最初に起動されるエージェントは top エージェントと呼ばれる．標準ではユーザインタフェースは top エージェント上で稼動する．2 番目に起動されるエージェントは msg_manager である．msg_manager エージェントは，メッセージ通信機構に相当し，メッセージ通信を実現する．

これら複数のエージェントは，単一 RXF 上では，複数のスレッドとして並行実行される．標準ではエージェントはそれ自身がインタプリタを持ち，他のエージェントと独立な節データベースを持つ．エージェント生成時のオプションによって，親エージェントと子エージェントが同じ節データベースを共有することを指定することが可能である．デフォルトでは，子エージェント生成時に，親エージェントの持つ節データベースの内容は，子エージェントの持つ節データベースにコピーされる．また，オプションとして，エージェントの生成時に，子エージェントの節データベースに，親エージェントのプログラムをコピーしないことも指定できる．

3.1.2 エージェントの動作状態

RXF におけるエージェントのインタプリタの動作状態として，RUN，WAIT，SUSPEND，SLEEP の 4 種類がある．これらの動作状態は節データベースやメッセージの相互排除や，デバッガによって使われる．RUN 状態は，インタプリタがプログラムの解釈・実行中であることを表す．WAIT 状態と SUSPEND 状態は，インタプリタがプログラムの解釈・実行を一時停止している状態を表す．WAIT 状態はイベント待ちや，共有メモリの相互排除などに利用される．SUSPEND 状態と WAIT 状態との相違点は，WAIT 状態のときはインタプリタは新規のプログラムを解釈・実行することはできないが，SUSPEND 状態のときは新規のプログラムを解釈・実行できる点である．このとき一時中断されたプログラムはスタックに保存される．SUSPEND 状態のとき，resume イベントが発生するとエージェントはスタックの先頭にあるプログラムの実行を再開する．SLEEP 状態は解釈・実行中のプログラムがない状態を表す．インタプリタは，プログラムの実行を終了したときに，SLEEP 状態に移行する．インタプリタは，SLEEP 状態のときに新規のプログラムの解釈・実行を依頼されると，RUN 状態に移行しプログラムの解釈・実行を開始する．

3.1.3 メッセージ通信

アプリケーションとして実行中の RXF 上では並行動作するエージェント群は互いに通信できる．もし，2 つの RXF が異なる計算機上で起動されていても，すべてのエージェントはお互いに通信可能である．Macintosh 上における RXF のエージェントは，アプリケーション間通信のための AppleEvent[67]，および計算機間の標準的な通信プロトコルである

TCP/IP を用いた通信が可能である．NEXTSTEP 上の実装では Mach の提供する IPC を用いた高速な通信も可能である．プログラマが簡単に利用できる通信機構を実装するためには，プログラマが実際にどのような通信機構を使っているかわからなくても通信できるようにする機構が必要である．異なるオペレーティングシステム間の場合 (例えば，Macintosh と NEXTSTEP 間) は，標準的に利用されている TCP/IP を標準の通信プロトコルとして利用する．

RXF で使われている次の通信用プリミティブについて説明する．

```
send(To, Message)
receive(From, Message)
```

send はエージェント To にメッセージ Message を送信するプリミティブである．To が単にアトムのときは，同一 RXF 上のエージェントを意味する．To が引数を一つ持つとき，その引数は計算機名を表す．例えば，

```
send(agent1,ok)
```

は，同一 RXF 上のエージェント agent1 へメッセージ ok を送信する．また，

```
send(agent2(host2),ok)
```

は，計算機名 host2 における RXF 上のエージェント agent2 へメッセージ ok を送信する．現状では，同一計算機上の複数の RXF を区別することはできない．これは，エージェント名と計算機名だけで目的のエージェントを指定できるという簡潔なエージェント指定方法をプログラマに提供するためである．すなわち，単一計算機上の並行動作を実現するためには，RXF を複数起動する必要はなく，単一の RXF 上でエージェントの並行動作が記述可能である．

receive は Message と単一化可能なメッセージを受信し，その送り手として From を得る．例えば，From として agent2(host2) が得られた場合，計算機名 host2 における RXF 上の agent2 が送り手となる．条件に適合するメッセージがなかった場合，receive のデフォルトの動作は失敗である．もし，条件に適合するメッセージが到着するまでプログラムの実行を中断したいときは，以下の形式で receive を用いる．

```
receive(From, Message, [wait])
```

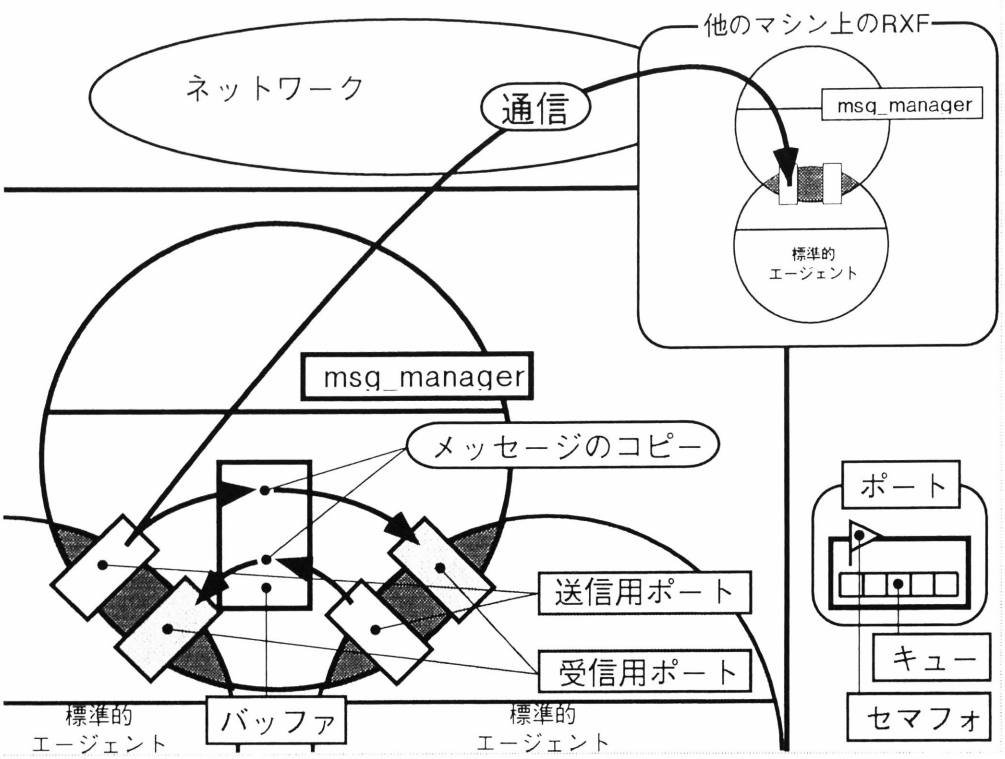

図 3.2: RXF におけるメッセージ通信

ここでは，メッセージの到着待ち時において，インタプリタは WAIT 状態になる．図 3.2 は RXF におけるメッセージ通信機構を表す．

msg_manager エージェントは，標準的なエージェントのエージェント管理機構の一部として動作する．msg_manager エージェントと標準的なエージェントは，メッセージポートを共用する．メッセージ通信は，msg_manager と標準的なエージェント間で共用されたメッセージポートによって実現される．メッセージポートには，送信用ポート，受信用ポートの 2 種類がある．送信用ポートと受信用ポートは，キューとセマフォを構成要素とする．メッセージポートへの書き込みは，メッセージポートに含まれるキューへの追加を意味する．メッセージポートからの読み込みは，メッセージポートに含まれるキューの先頭要素を取り出すことを意味する．メッセージポートは，msg_manager エージェントと関連するエージェント間の共有メモリなので，相互排除をおこなう必要がある．メッセージポートにおける相互排除は，セマフォに基づく相互排除処理によって実現している．

エージェントはメッセージポートに受け手のエージェント名とメッセージの対を送信用ポートに書き込む．msg_manager エージェントは標準的なエージェントの送信用ポートを

監視している．msg_manager エージェントは，送信ポートに対する書き込みを検出すると，その送信ポートに対して書き込みをしたエージェント，つまり，メッセージの送り手のエージェントを WAIT 状態にする．msg_manager エージェントは，メッセージを msg_manager エージェントの管理するバッファにコピーし，送り手のエージェントを RUN 状態にする．このとき受け手のエージェントが，送り手のエージェントと同一 RXF 上に存在するときと，そうでないときがある．同一 RXF 上に存在するとき，かつ，受け手のエージェントが RUN 状態のとき，msg_manager エージェントは，受け手のエージェントを WAIT 状態にする．その後，msg_manager エージェントは，送り手のエージェント名とメッセージの対を受け手のエージェントの受信用ポートに書き込む．ここでは，送り手のエージェント名とメッセージの対が，受信用ポートのキューにコピーされる．メッセージのコピー処理前の受け手のエージェントの状態が RUN 状態だったならば，受け手のエージェントの状態を RUN 状態に戻す．ここでのメッセージのコピーに関する冗長な処理は，論理型言語の持つバックトラックの機能に対応するために行われる．すなわち，送り手のエージェントがバックトラックによってメッセージのデータを破壊しても，msg_manager エージェントや受け手のエージェントに影響のないように冗長にコピーする．実際には送り手がバックトラックによってメッセージを破壊してしまうまではコピーする必要がないことに着目することにより本メッセージ通信の高速化を実現する．

指定された計算機が送り手のエージェントの存在する計算機と同一の計算機でないときは，指定された計算機上に存在する RXF の msg_manager エージェントに対して送り手のエージェント名，受け手のエージェント名，そしてメッセージの 3 つ組を送信する．指定された計算機上で実行中の RXF が存在しなかったときは，送信を中止する．受け手の存在する RXF 上の msg_manager エージェントは，メッセージを受け取ると，送り手のエージェントと同様にメッセージの送信処理を実行する．

RXF におけるメッセージ通信の特徴は，単一 RXF 上におけるエージェント間通信だけではなく，簡潔なエージェント指定方式や，msg_manager エージェントを用いてネットワークを利用するための手続きの隠蔽を実現することにより，ネットワーク上に分散した計算機上で動作する RXF におけるエージェント間のメッセージ通信も容易に実現する点にある．

3.2 設計

本章では，RXF の設計とその実装について述べる．RXF は，(1) マルチエージェントシステムの実行系，(2) マルチエージェントシステムを構築するための開発環境，(3) の 2 つの部分から構成されている．(2) の開発環境は，(1) の実行系上で動作している．よって，開発支援エージェントに開発の支援をさせるということも可能である．本論文では，(1) のマルチエージェントシステムの実行系について述べる．

自律したエージェントを開発するための開発支援機構の実現のために，(a) エージェントオペレーティングシステム（Agent Operating System : AOS），(b) エージェント記述言語（Agent Description Language : ADL），(c) エージェントフレームワーク（Agent FrameWork : AFW），の 3 つを実装した．(a) のエージェントオペレーティングシステムは，エージェントの実行形である．エージェントの性質を実現するための機能を提供する．(b) のエージェント記述言語は，エージェントを記述するためのプログラム言語である．(c) のエージェントフレームワークは，エージェント構築のためのライブラリである．図 3.3 は，これら 3 つの関係を表している．(a) のエージェントオペレーティングシステムは，特定のプラットフォーム（図 3.3 では MacOS 上）において，エージェントの性質を実現するための機能を提供する．(b) のエージェント記述言語は，(a) 上で動作する．(c) のエージェントフレームワークは，(b) を用いて (a) の機能を利用することで実現される．(a) (b) が，(1) の実行系を構成する．(b) (c) が，(2) の開発環境を構成する．(b) は，言語処理系なので，実行系にも開発環境にも属する．本章では，エージェント記述言語について述べる．エージェントオペレーティングシステムは，後述する．

(a) ～ (c) を以下にまとめる．

- (a) エージェントオペレーティングシステム（AOS: Agent Operating System）
  AOS は，エージェントの性質を実現するための機能を実現する．例えば，エージェントの内部と外部を分離するための機能を提供する．例えばこの機能は，エージェントの自律性を実現するために必要である．

- (b) エージェント記述言語（ADL: Agent Description Language）
  ADL は，エージェントの仕様を記述するために用いられる．ADL は，エージェントの宣言的または手続き的な使用を記述するための能力が必要である．

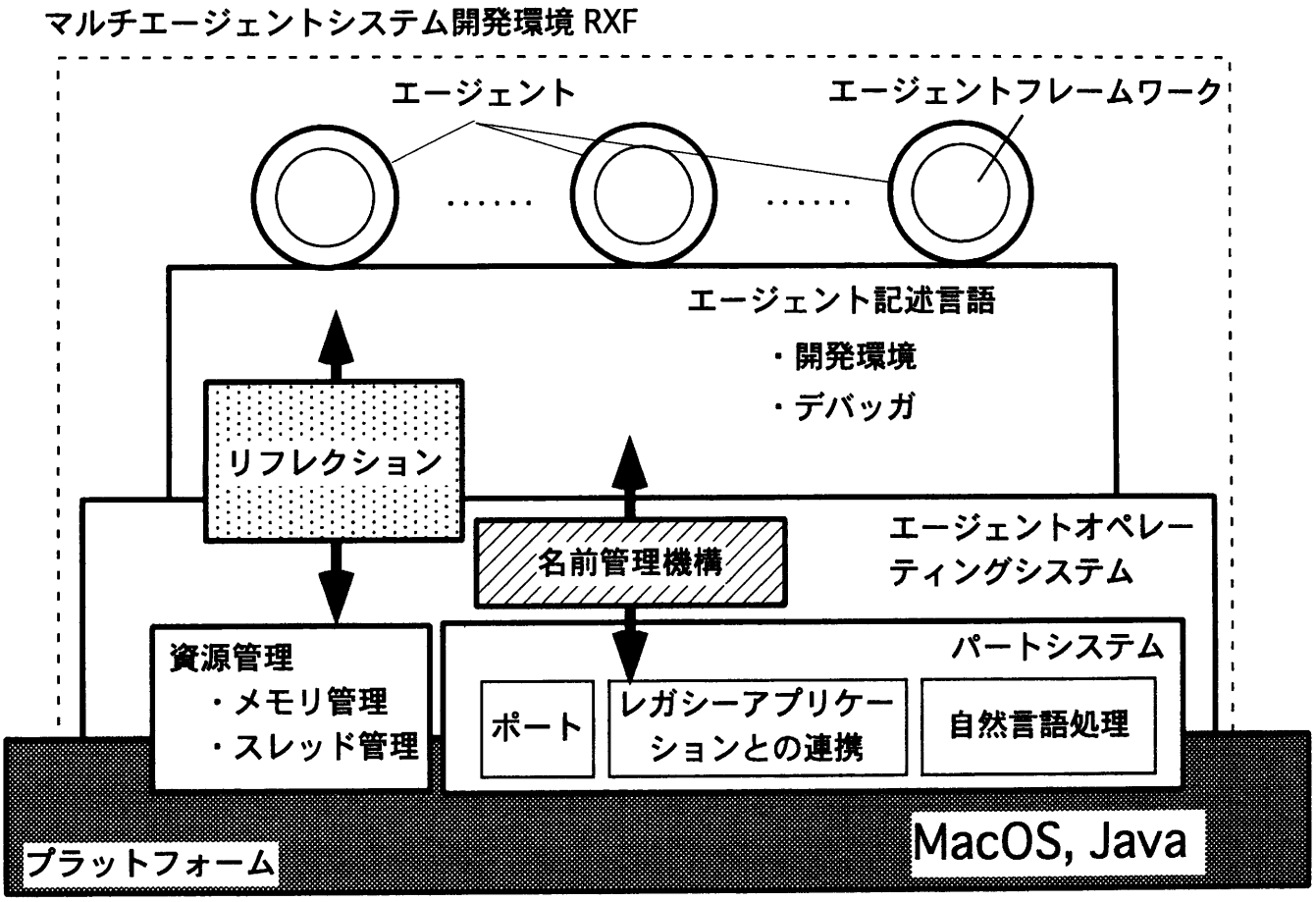
マルチエージェントシステム開発環境 RXF
エージェント
エージェントフレームワーク
エージェント記述言語
・開発環境
・デバッガ
リフレクション
名前管理機構
エージェントオペレーティングシステム
資源管理
・メモリ管理
・スレッド管理
パートシステム
ポート
レガシーアプリケーションとの連携
自然言語処理
プラットフォーム
MacOS, Java

図 3.3: RXF の構成

- (c) エージェントフレームワーク (AFW: Agent FrameWork)
  AFW は，特定のエージェントアーキテクチャに依存しないエージェントの枠組みを提供する．例えば，AFW の提供する枠組みには，エージェント内のスレッドやメッセージの管理等が含まれる．またスレッド，メッセージ管理機構を応用して，動的環境に対応するための割り込み処理機構を実装している．

RXF におけるエージェント記述言語は，制約論理型言語である．ここでは，RXF における制約論理型言語インタプリタを，図 3.4 を用いて説明する．図 3.4 は，RXF における制約論理型言語インタプリタ (以降インタプリタと略す) が，(1) メモリ管理機構，(2) スレッド管理機構，(3) 入出力管理機構，の 3 つの機構と協調して動作する様子を図式化したものである．(1) のメモリ管理機構は，インタプリタからのメモリ割当要求に対してメモリを割り当てるための機構である．メモリを割り当てていくうちに，メモリが足りなくなるが，メモリ管理機構は，GC (Garbage Collection) によってメモリ不足を解消しようと試みる．(2) のスレッド管理機構は，スレッドに関する処理をおこないスレッドを管理する．(3) の入出力管理機構は，入出力のバッファリングや，入出力のための同期処理などをおこなう．インタプリタ上では，入出力機構をポートと呼ばれる機構によって利用する．

図 3.4 で示すように，インタプリタは，その内部に制約解消系を含む．制約解消系は，制約をあらかじめあたれられたアルゴリズムを用いて解く．

3.3 エージェント記述言語

近年，エージェントを構築するうえで，論理に基づく実装が数多く研究されている．これらの研究は，エージェントの様相 [19] の管理に用いられ，その有用性が示されている [68]．また，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 [69]．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．

制約ソルバーは，与えられた制約をあらかじめ定められたアルゴリズムを用いて解く．Prolog は単一化に基づく変数の束縛を行っているが，制約論理型言語において，単一化は，単一化する項同士を項同士の制約 (等式 項 = 項) とみなし単一化アルゴリズムを制約解消アルゴリズムとして解く．

RXF における制約ソルバーの設計と実装について説明する．RXF における制約ソルバー

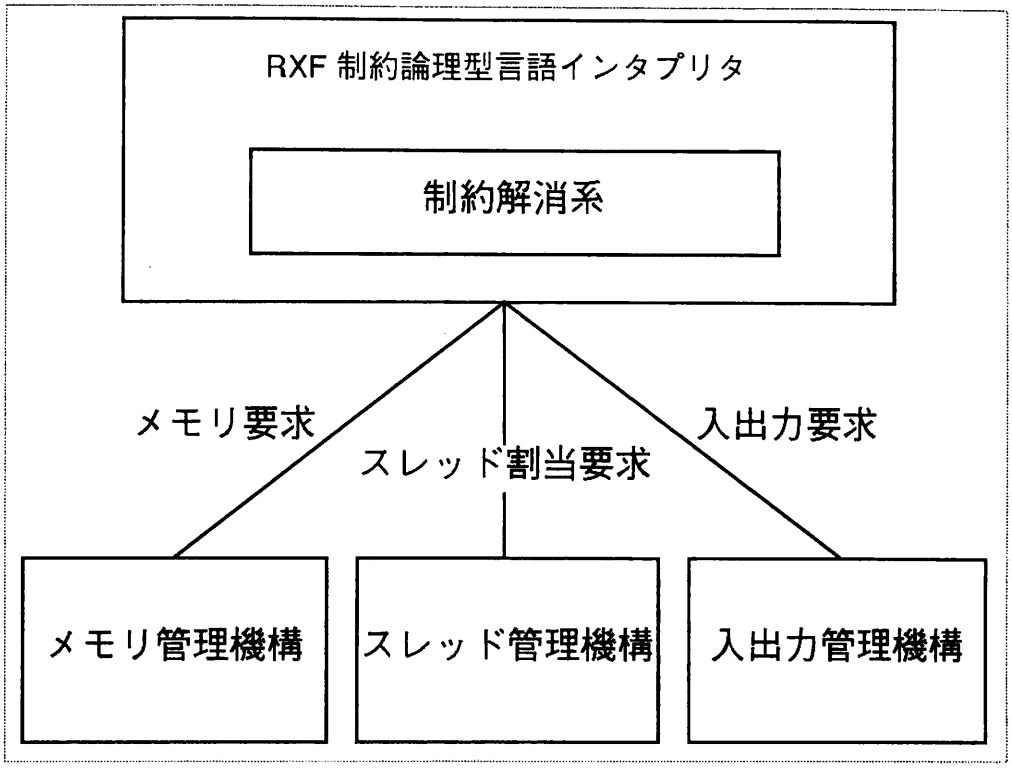

図 3.4: RXF インタプリタの構成

(以降制約ソルバーと略す) の特徴は, (1) 制約として, 論理式, 線形方程式, リスト式, そして文字列式が扱える, (2) 拡張可能, の 2 つの特徴がある. (1) の特徴は, 制約ソルバーによって, 単一化をおこなったり, 線型方程式を解いたり, リストの足し算を処理したり, 文字列の引き算を処理できることを意味する (全バージョンでは, 線型不等式も扱えたが現在はサポートを中止). (2) の拡張可能というのは, 制約ソルバーが, 動的に新しい種類の式を扱えるという意味である. (2) によって, ユーザ定義の式を解く制約ソルバーを拡張できる.

制約ソルバーは, あらかじめ定められた制約解消アルゴリズムを用いることによって, 制約を解くプログラムである. 制約ソルバーが扱える制約は, (1) 等式, (2) 不等式, (3) not, の 3 種類である. それぞれの仕様ついては後程説明する.

制約ソルバーは, 複数の領域依存ソルバーによって構成される. 領域依存ソルバーは, 特定の型を持つ式を解くためのソルバーである. 例えば, 線型方程式は, 線型方程式ソルバーと呼ばれる領域依存ソルバーによって解かれる. 制約ソルバーが扱える制約と, 領域依存ソルバーの対応関係は, (a) 論理式ソルバーと論理式, (b) 線型制約ソルバーと線型方

```
プログラム ::= ファクト | ルール
ファクト ::= ヘッド
ルール :: ’ヘッド :- ボディ’
ヘッド ::= アトム | ファンクタ
ボディ ::= 節
節 ::= AND 節 | OR 節 | リテラル
AND 節 ::= ’節, 節’
OR 節 ::= ’節 | 節’
リテラル ::= アトム | ファンクタ | 制約
アトム ::= {a-z}.+
ファンクタ ::= ’ファンクタ名( 引数 )’
ファンクタ名 ::= {a-z}.+
引数 ::= 項 | ’項, 引数’
項 ::= アトム | ファンクタ | 数 | 変数 | リスト | 文字列
制約 ::= ’式 = 式’ | ’式 > 式’ | ’式 >= 式’ | ’式 < 式’ | ’式 =< 式’
式 ::= 項 | 項 オペレータ 項
オペレータ ::= {+, -, *, / }
数 ::= {0-9)* | {0-9}*’.’{0-9}*
変数 ::= ’{A-Z}.+’ | ’_.+’
リスト ::= ’[]’ | ’[項, 項]’ | ’[項 | リスト ]’ | ’[項 | 変数 ]’
文字列 ::= ".+"
```

図 3.5: RXF におけるプログラム

程式, (c) リストソルバーとリスト, (d) 文字列ソルバーと文字列, の 4 つである. (a) の論理式ソルバーは, 単一化アルゴリズムを用いる. (b) の線型制約ソルバーは, シンプレックス法を用いて線型方程式, および線型不等式を解く. (c) のリストソルバーはリストの加算をリストの結合として定義することによって, リスト間の制約を解く. (d) の文字列ソルバーは, 文字列を文字のリストとみなすことによって, リストソルバーと同様の処理によって制約を解く.

3.4 RXF プログラミング

図 3.5 で示すように, ほとんどが, 標準的 Prolog に準拠する. 標準的 Prolog との違いは制約である. RXF は, 制約として等式と不等式を扱う. 例えば, 制約は, X+Y=0 のように記述される. 制約中では, 関数を利用できる. 関数は, ファンクタの前に '@' をつけ

```
(1) 普通の append
   append([],L,L).
   append([H|L1],L2,[H|L3]) :-
     append(L1,L2,L3).

(2) 制約処理を利用した append その 1
   cappend(L1,L2,L1+L2).

(3) 制約処理を利用した append その 2
   fappend(L1,L2,L3) :-
     L3 =
       @if(@is_nil(L1),
         L2,
         @cons(@car(L1), @fappend(@cdr(L1), L2))
       ).
```

図 3.6: RXF におけるプログラム例：append

た形をしている．図 3.6 で，実際の RXF におけるプログラムを用いて説明する．

図 3.6 における（1）の append は，標準的な Prolog 上で実行可能な append である．図 3.6 における（2）の append は，RXF におけるリストの制約処理を利用した append である．この例では，cappend の第 3 引数の L1+L2 がリストの制約に相当する．これは，cappend の第 3 引数が L1 と L2 を append したリストであることを宣言している．(3) の append も（2）と同様に RXF における制約処理を利用した append である．これは，RXF の制約処理における関数処理を利用した例である．RXF は，関数が出現するとそれが組込み関数か調べる．もし組込み関数ならばそれを実行する．組込み関数でないときは，以下の手順で関数の形式を変形しゴールとして評価する．

1. 制約 C 中の関数を $@f(A_1, A_2, .., A_n)$ とする．C 中に出現しない変数 X を用意する．
2. C 中の $@f(A_1, A_2, .., A_n)$ を X で置き換える．
3. 制約 $X = @f(A_1, A_2, .., A_n)$ を新たな制約とする．
4. ゴール $f(A_1, A_2, .., A_n, X)$ を評価する

例えば，X+Y=@f(1) の場合は，(1) X でも Y でもない変数 Z を用意，(2) X+Y=Z に変形，(3) 制約 Z=@f(1) を追加，(4) ゴール f(1,Z) を評価，のようになる．本処理によっ

(1) 普通のハノイの塔

```
hanoi([[From,To]], 1, From, Via, To).
hanoi(Sol, N, From, Via, To) :-
  hanoi(Sol1, N-1, From, To, Via),
  hanoi(Sol2, 1, From, Via, To),
  hanoi(Sol3, N-1, Via, From, To),
  append(Sol1, Sol2, Sol12),
  append(Sol12, Sol3, Sol).
```

(2) 制約処理を利用したハノイの塔

```
hanoi(1,F,V,T,[[F,T]]).
hanoi(N,F,V,T,@hanoi(N-1,F,T,V) + (@hanoi(1,F,V,T) + @hanoi(N-1,V,F,T))).
```

図 3.7: RXF におけるプログラム例：ハノイの塔

て特別な関数定義のための手段を用意する必要がなく，かつ，従来の Prolog プログラムを再利用できる．

次に RXF におけるハノイの塔のプログラムを示す．図 3.7 の (1) が普通に定義したハノイの塔で，(2) がちょっと変わったハノイの塔である．(2) のハノイの塔では，RXF のリストソルバーと関数処理機構を利用している．

これらのように，プログラムを制約として記述することによってプログラムをコンパクトに記述することができ，プログラムの可読性を高めることができる．

3.5 エージェント記述言語におけるリフレクション

3.5.1 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，例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．例えば，3.7.1 節で示すような実行制御を可能にする．さらに，AGENT 0[20] 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．例えば，3.7.2 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．例えば，3.7.3 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．

3.5.2 メタレベル表現

3.5.1 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

型	メタレベル表現
アトム X	atom(X)
空リスト []	nil
リスト $[X_1\|X_2]$	list(M_1,M_2)
ファンクタ f(X_1,X_2,...,X_n)	functor(f,n,[M_1,M_2,...,M_n])
数値 X	number(X)
文字列 X	string(X)
変数 X	var(N_X,ID_X) N_X：X の名前 ID_X：変数 ID

(注：X_i のメタレベル表現は M_i)

表 3.1: メタレベル表現一覧

例		
型	ベース	メタレベル表現
アトム	a	atom(a)
空リスト	[]	nil
リスト	[a,b]	list(atom(a),list(atom(b),nil))
ファンクタ	f(a,b)	f(a,2,[atom(a),atom(b)])
数値	3.14	number(3.14)
文字列	"hello"	string("hello")
変数	X	var('X',1)

表 3.2: メタレベル表現例一覧

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．

表 3.1 は，RXF における述語のメタレベル表現の一覧表である．述語 p のメタレベル表

現は，p の型や内部情報を反映している．アトム X のメタレベル表現は，atom(X) となる．アトムのメタレベル表現は，データの内部表現という観点から，アトムの名前の文字列を表現するという考え方もあるが，ここでは，組み込み述語 atom/1 との整合性を重視した結果このようになった．同様に，ファンクタにおけるファンクタ名や，文字列においても同様の理由で 表 3.1 のようになっている．空リスト [] のメタレベル表現は，“nil” である．リスト [X_1|X_2] のメタレベル表現は，list(M_1,M_2) となる．ここで M_1 と M_2 は，それぞれ X_1 と X_2 のメタレベル表現である．ファンクタ名 “f” でアリティ n のファンクタ f(X_1,X_2,...,X_n) のメタレベル表現は，functor(f,n,[M_1,M_2,...,M_n]) となる．ファンクタのメタレベル表現のそれぞれのは，順番にファンクタ名，アリティ，そして引数のメタレベル表現のリストを表す．数値 X のメタレベル表現は，number(X) である．これも組み込み述語 number/1 との整合性のために，ベースレベルとメタレベルの間で数値の表現形式に差はない．文字列 X のメタレベル表現は，string(X) である．メタレベルにおける文字列の表現を，文字列を構成する文字のリストとすることも考えられるが，前述の理由によりベースレベルにおける文字の表現とメタレベルにおける文字の表現は同じにした．最後に変数 X のメタレベル表現を説明する．変数 X のメタレベル表現は，var(N_X,ID_X) である．ここで，N_X は X の名前を表すアトムである．ID_X は，変数につけられた固有の ID 番号である．変数への値の束縛・解放を行うときは，この ID で操作対象の変数を指定する．メタレベル表現を用いることにより，言語の機能を詳細に制御することが可能になる．

3.5.3 リフレクション述語 meta

リフレクション述語 meta は，アトムやリストなどのデータのメタレベル表現を得るためと，メタレベル表現からデータを生成するための組み込み述語である．図 3.8 は，リフレクション述語 meta の実行例である．meta には，2 引数の meta/2 と 3 引数の meta/3 がある．meta/2 は，第 1 引数のメタレベル表現が第 2 引数であることを表す．例えば 図 3.8 中の meta/2 の最初の例 “`meta(X,var('X',1))`” は，第 1 引数の変数 “`X`” のメタレベル表現が第 2 引数の “`var('X',1)`” であることを表している．ここでメタレベル表現 “`var('X',1)`” は，変数 “`X`” の型が変数（“`var`” が表す）で，変数名が “`'X'`”（“`var`” の第 1 引数が表す），変数の ID が “1”（“`var`” の第 2 引数が表す）であることを意味している．

meta/3 と meta/2 の違いは，meta/3 の第 3 引数にある．meta/3 の第 1，2 引数は，

```
meta/2
  meta(X,var('X',ID))
  meta(1,number(1))
  meta(a,atom(a))
  meta(f(1,2),functor(f,2,[1,2]))
  meta([1,2],list(1,list(2,[])))
  meta([],[])
  meta((h:-t),clause(h,[t]))
meta/3
  ?- append([a,b],[c,d],A),meta(A,B,C).
  A = [a,b,c,d]
  B = var('A',1)
  C = [bound(1,list(var('H',4),var('L3',7))),
       bound(7,list(var('H',8),var('L3',11))),
       bound(11,list(atom(c),list(atom(d),nil))),
       bound(8,atom(b)),bound(4,atom(a))]
```

図 3.8: リフレクション述語 meta の実行例

meta/2 と同じである．meta/3 の第 3 引数は，meta/3 を評価した時点での環境のメタレベル表現を表している．例えば 図 3.8 の meta/3 の例では，meta/3 の第 3 引数には append を実行した後の環境が束縛されている．

meta/2 と meta/3 を用いることによってデータや環境のメタレベル表現を容易に得ることが可能になり，リフレクティブなプログラムの記述が可能になる．

3.5.4 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．

3.5.5 リフレクション述語 up_call

計算論的リフレクションにおいて，言語 L のインタプリタ I_L が言語 L_{I_L} で記述されているとき言語 L_{I_L} のインタプリタ $I_{L_{I_L}}$ をメタレベルインタプリタ（以降メタレベルと呼ぶ）と呼び，I_L をベースレベルインタプリタ（以降ベースレベルと呼ぶ）と呼ぶ．メタレベルにおけるプログラムの評価をメタレベル実行と呼ぶ．メタレベルにおけるベースレベルの情報はメタレベル表現と呼ばれる．論理型言語における計算論的リフレクションの問題として，メタレベルにおけるベースレベルの変数の扱いがある [70]．RXF ではメタレベルにおけるベースレベルにおける変数への処理は，メタレベルにおけるベースレベルにおける変数への処理をおこなうための組込み述語 bind によって処理することによって解決した．

RXF ではメタレベルをベースレベルにおける言語インタプリタとして利用するだけでなく，問題解決におけるメタな処理をメタレベルで実現することによるプログラムのモジュール化にも利用する．これは，リフレクション述語 reflect によって可能になる．このような処理をおこなう場合，bind による変数の束縛処理はプログラムを複雑にする．RXF ではこのような処理をおこなうためのメタレベル実行機能を提供することによってこの問題を解決した．RXF において reflect/1 を用いたメタレベル実行では，メタレベルにおけるベースレベルの変数の扱いは，プログラマによって適切に行われる必要がある．問題解決におけるリフレクションの利用においてこのような処理はプログラムを複雑にし，バグの原因になる．up_call/1 はメタレベルにおけるベースレベルの変数の扱いをプログラマに対して隠蔽する役割を持つ．

RXF におけるメタレベル実行に関連したリフレクション述語は meta(Obj,Meta), reflect(Q), up_call(Q) の 3 つである．meta(Obj,Meta) は Obj のメタレベル表現 Meta を生成する．逆にメタレベル表現 Meta から Obj を生成することも可能である．reflect(Query)

はメタレベル上で，Query を評価するための組込み述語である．この時，メタレベル上では，Q = foo(a1,a2,...) のとき foo(AP,a1,a2,...) が評価される．AP はエージェントポート [6] と呼ばれるエージェントのメタレベル表現を得るための機構である．up_call(Q) はメタレベルにおいて Q を評価する．meta/2, reflect/2 は計算論的リフレクションのための基本的な組込み述語である．up_call/1 はメタレベルにおけるベースレベルの変数の処理を隠蔽するための組込み述語である．

以下のプログラムは up_call/1 の実装概要を示す．

```
up_call(Q) :-
    meta_rename(Q, Q1),
    call(Q1),
    meta(Q1,Q2),
    meta_unify(Q,Q2).
```

up_call/1 はメタレベルで定義されている．ここで，Q はメタレベル表現である．meta_rename(X,Y) はメタレベル表現 X から meta_rename/2 を呼び出したレベルにおけるベースレベルのオブジェクトを生成する．meta_unify(X,Y) はメタレベル表現 X, Y を単一化し単一化による変数への代入をベースレベルに反映する．up_call/1 は，meta_rename/2 によってメタレベル表現を現在のレベルで評価可能なオブジェクト（ここでは Q1）に変換する．次に，Q1 メタレベルで評価する．Q1 の評価が成功したら meta/2 によって Q1 のメタレベル表現 Q2 を得る．最後に，meta_unify/2 によって Q1 の評価の結果をベースレベルに反映する．以上の処理により up_call/1 を用いることによってベースレベルの変数の束縛処理をプログラム中でおこなう必要がなくなる．

3.6 リフレクション機構の実装

3.6.1 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．リフレクション機構の観点からの RXF の構成図は，図 3.9 で示される．図 3.9 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 3.9 の斜線部分）は，マ

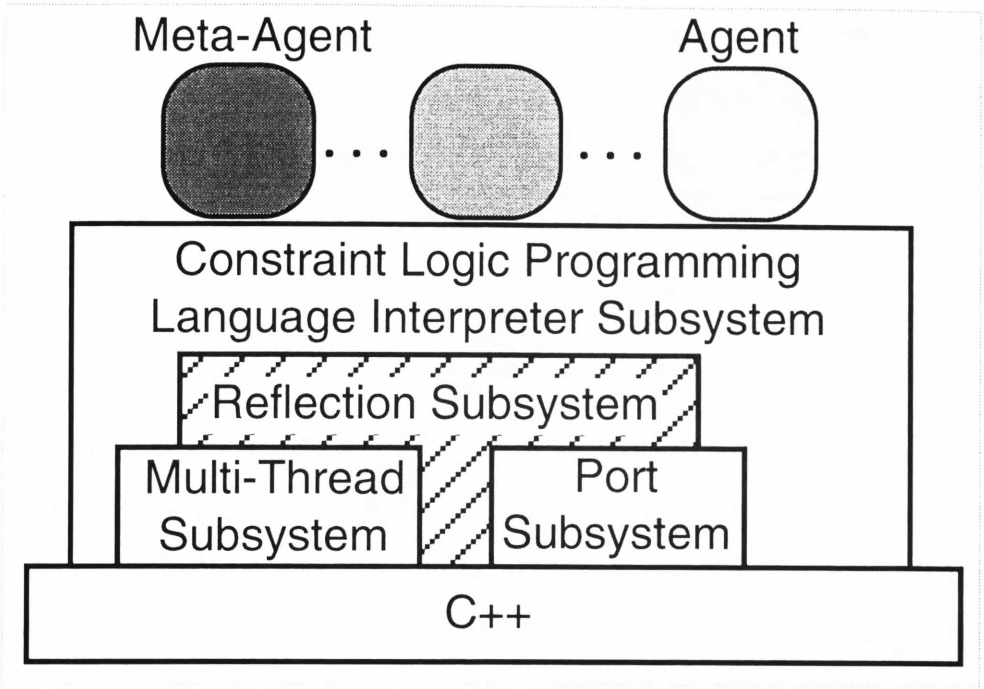

図 3.9: RXF におけるリフレクション機構の構成図

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 3.9 の制約論理型インタプリタは，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 3.9 における灰色部分がエー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[71] 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．

3.6.2 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も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．例えば，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わす．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わす．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こむ．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．

3.6.3 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 3.9 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において，RXF の制約論理型言語における制約解消系を利用したメタレベル表現の遅延生成機構が実装されている．メタレベル表現の生成要求を，制約として制約解消系で管理することによって，メタレベル表現の遅延生成が実現されている．

エージェントポートの仕組みを，図 3.10 に示す．ここでは特にエージェントのインタプ

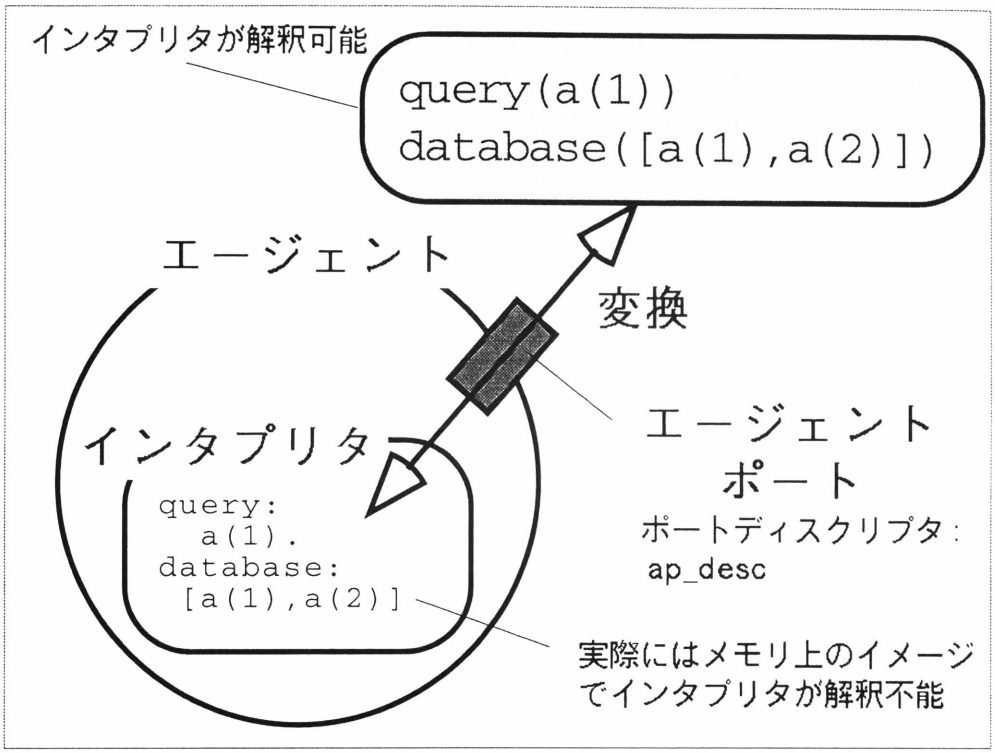

図 3.10: メタレベル表現とエージェントポート

リタのメタレベル表現の取得に焦点をあてる．ここで，図 3.10 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 [42] 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

表 3.3: reflect ベンチマーク

	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）

ことである．

3.6.4 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（例えば組込み述語）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・実行と同様におこなわれる．

表 3.3 に call 述語を用いたときの実行速度と，reflect 述語を用いた時の実行速度，そして vcall（バニラインタプリタ [70]）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程効率の悪いものではないことを示している．

・**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 3.11: "catch & throw" の定義と使用例

3.7 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 [5] 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．

3.7.1 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[72] 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 3.11 に catch & throw のプログラム定義とその使用例の概略を示す．図 3.11 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．例えば，

```
:- 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 3.11 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（3.6.2 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 3.11 の例では，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．

3.7.2 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 本の手をもつ 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．ここでおこな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挙げ下げるするのを，a が b の状態を観察して b と同じように手を挙げ下げする．概要を図 3.12 で示す．

図 3.12 中の丸はエージェントを表わす．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

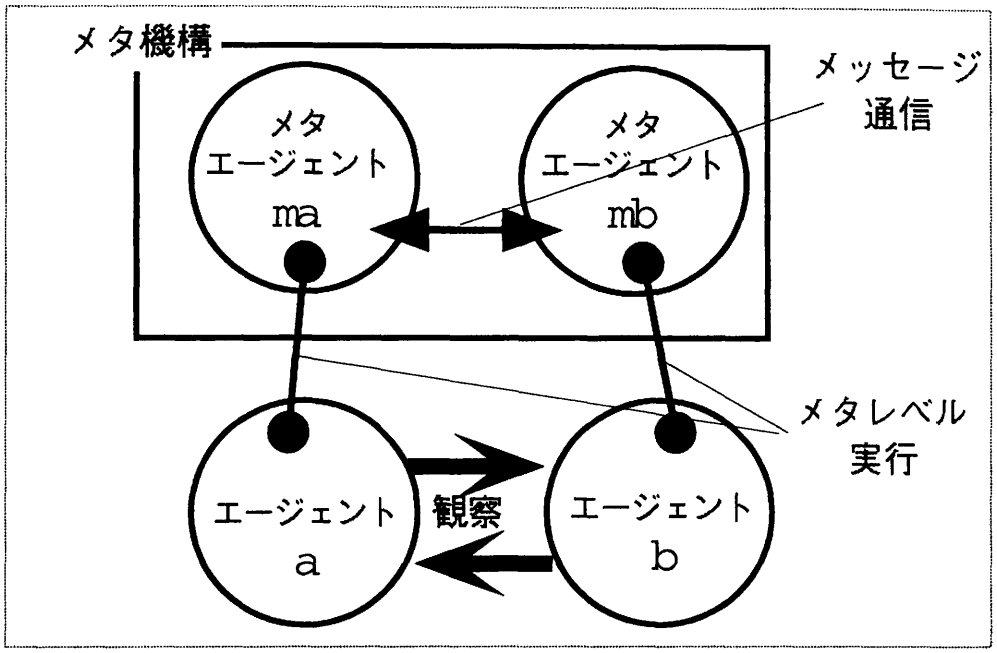

図 3.12: RXF におけるリフレクションの例

(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 3.13: reflect 使用例

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 3.13 はプログラムの概略を示している．図 3.13 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．

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．例えば，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．

3.7.3 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[73] 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 [74]．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．

図 3.14 は，BDI を管理するメタエージェント m のプログラム（図 3.14 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 3.14 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．

```
・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 3.14: 組織化エージェントプログラム例

図 3.14 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 3.14 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 3.14 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 3.14 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 3.14 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 [7] 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．

3.8 マルチエージェントシステム開発環境

RXF におけるユーザインタフェースは，エージェントプログラムを効率的に実装するためのプログラミング環境を提供する．新たに生成されたエージェントには，標準的なユーザインタフェースが暗黙的に用意される．

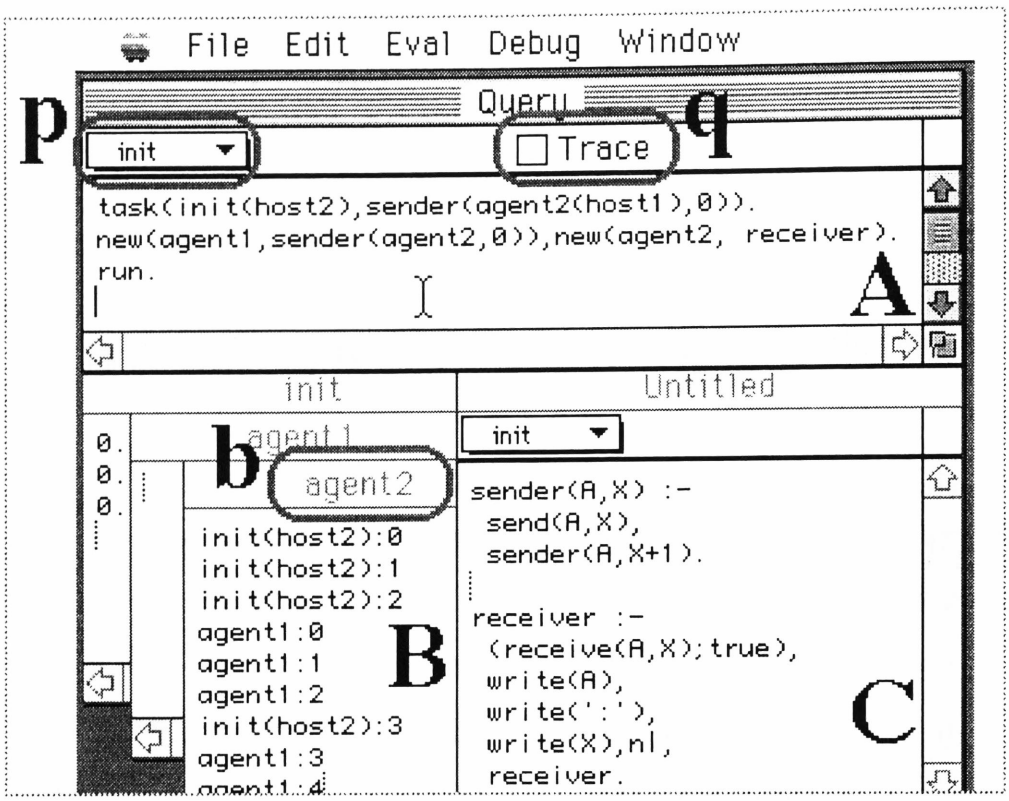

図 3.15: RXF のユーザインタフェース

図 3.15 は，Macintosh 上に実装された RXF のユーザインタフェースを表している．

本ユーザインタフェースにおいて，ウィンドウの種類は，コンソールウィンドウ（図 3.15 の A），エージェントウィンドウ（図 3.15 の B），プログラムウィンドウ（図 3.15 の C）の 3 種類がある．

コンソールウィンドウは，RXF のユーザインタフェースにおいて，必ず 1 つ存在する．ユーザは，コンソールウィンドウを用いることによって，システムエージェントを除く全てのエージェントに対して質問できる．コンソールウィンドウの内容は，ポップアップメニュー（図 3.15 の p）で選択されたエージェントによって評価される．チェックボックス（図 3.15 の q）は，デバッガ使用の有無を指定する．

エージェントウィンドウは，各エージェントに対する標準的なユーザインタフェースである．例えば，図 3.15 の B のエージェントウィンドウは，図 3.15 の b で示されたエージェントのインタフェースである．ユーザは，エージェントウィンドウを使ってエージェントに質問することもできる．エージェントは，出力をエージェントウィンドウへ出力する．

プログラムウィンドウは，プログラムを編集するためのウィンドウである．プログラム

ウィンドウにもコンソールウィンドウと同様なエージェント選択用のポップアップメニューがある．ポップアップメニューによって，プログラムウィンドウ中で定義されたプログラムをどのエージェントで利用するかを指定することが可能である．プログラムウィンドウのタイトルには，プログラムを保存するファイル名が記されている．

3.9 成果

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型プログラミング，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．

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 [19] 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 [70] 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 [70] 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めざしている．

RXF におけるリフレクション機能の利用のための 3 つの組込み述語について述べた．meta/2 と reflect/1 はリフレクションのための基本的な組込み述語である．up_call/1 はリフレクションの機能を利用しやすいようにメタレベル表現の扱いを隠蔽した組込み述語である．up_call/1 においてメタレベル表現の扱いを隠蔽するために，メタレベルではベースレベルからの質問のコピーを評価する．これによりメタレベルにおける変数の扱いに関す

る問題を回避した.

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挙げられる．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[71] 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.

第4章　RXF におけるエージェントオペレーティングシステム

オペレーティングシステムは，計算機ハードウェアとユーザの間に位置するソフトウェアである．オペレーティングシステムは，計算機ハードウェアの資源を仮想化することによって，ユーザにとって使い勝手の良い計算機環境を提供する．さらに，オペレーティングシステムは，ユーザに使い勝手の良い計算機環境を提供するだけで無く，計算機ハードウェア資源や，ソフトウェアの効率的な管理や制御も目的とする [75].

RXFにおけるエージェントオペレーティングシステムの目的は，(1) 計算機環境の仮想化，(2) 計算機資源の管理，の2つである．(1) の計算機環境の仮想化とは，エージェントが実行している計算機環境に関する情報を抽象化することである．これにより，エージェントが環境を認識するプログラムの記述が容易になる．(1) は，エージェントの実現に大きく関係している．例えば，エージェントの自律性の実現のためには，エージェント自身で環境に適応する能力が必要である．(1) が適切に実現されていれば，エージェントの自律性の実現のためのプログラムの記述は容易になる．(2) の計算機資源の管理は，スレッドやメモリなどの計算機資源を管理することである．エージェントは，永続的に動作することも求められるので，メモリ管理は特に重要である．本章では，RXFのエージェントオペレーティングシステムについて述べる．

4.1 エージェントオペレーティングシステム

AOSは，エージェント構築に必要な機能をプログラマに提供するためのシステムである．AOSでは，各種機能をパートと呼ばれる単位でプログラマに提供する．AOSは，複数存在するパートを管理するシステムである．ここで，パートの管理とは，(1) 生成・廃棄の管理，(2) メモリの管理，(3) 実行状態の管理，(4) 永続性の管理，(5) 名前の管理，(6) アクセス管理，の6つである．(1) 生成・廃棄の管理は，生成要求や廃棄要求に対して，適切なパートを生成・廃棄することである．(2) のメモリの管理は，パートの使用するメモリの管理をおこない，参照されていないパートの使用しているメモリの解放などを行う．すなわちごみ集め（GC: Garbage Collection）[76] 処理を行う．(3) の実行状態の管理は，パートの実行状態の管理を行う．(4) の永続性の管理は，永続的なパートの管理を行う．永続的なパートは，二次記憶装置にデータを格納し，永続的に状態を保存することができる．永続性の管理は，パートの移動に関連する．パートの移動とは，ホストA上で実行中のパートPが，Hとネットワークで接続されたホストB上に移動して実行をすることを意味す

る．概念的には，オブジェクトの migration ?? に相当する．(5) の名前の管理は，パートの名前を管理する．パートの名前は，パートを特定するために用いられる名．(6) のアクセス管理では，パートへのアクセスコントロールを行い不正なアクセスを禁止する．

エージェントはその外部と相互干渉をする必要がある．本 AOS では，外部との相互作用のためのパートをポートと呼ぶ．ポートは，外部に対する入出力を実現する．ポートの特殊な形式として，ストリームがある．ストリームは，ポートと異なり出力か入力の片方だけを実装する．出力用のストリームを出力ストリームと呼ぶ．入力用のストリームを入力ストリームと呼ぶ．エージェントの外部として，(1) コンピュータ環境，(2) エージェント，(3) ユーザ，の 3 種類に分類する．(1) のコンピュータ環境は，エージェントが実行しているコンピュータの状態である．例えば，エージェントが実行しているコンピュータ上に存在するファイルシステムが挙げられる．(2) のエージェントとは，自分以外のエージェントとエージェント自身である．(3) のユーザは，エージェントと相互作用するユーザである．ユーザとの相互作用は，重要なので，特別に分けた．(1) のコンピュータ環境との相互作用には，ファイルを操作するためのファイルポート等を用いる．(2) のエージェントとの相互干渉は，主にメッセージ通信ポートを用いる．メッセージ通信ポートは，他のエージェントとのメッセージ通信を実現するポートである．(3) のユーザとの相互作用は，インタフェースポートによって達成される．インタフェースポートは，ウィンドウシステムの場合，ウィンドウの操作等を実現する．

AOS は，エージェント構築のためのさまざまな機能を提供するためのシステムである．AOS では，さまざまな機能をパートと呼ばれる単位で管理する．すなわち，AOS は，エージェント実現のための機能を提供する，複数のパートを管理するためのシステムである．パートの管理として，(1) メモリの管理，(2) 実行の管理，(3) 永続性の管理，(4) 名前の管理，の 3 つが挙げられる．メモリの管理は，パートの使用するメモリの管理をおこない，参照されていないパートの使用しているメモリの解放などを行う．(2) の実行の管理は，パートをスレッドとして実行するための管理を行う．本 AOS では，1 つのパートが複数のスレッドを持つことを許す．(3) の永続性の管理は，永続的なパートの管理を行う．永続的なパートは，二次記憶装置にデータを格納し，永続的に状態を保存することができる．(4) の名前の管理は，パートを指定するための名前を管理する．名前の管理では，パートへのアクセス制御も行う．

エージェントはその外部と相互干渉をする必要がある．本 AOS では，外部との相互作

用のためのパートをポートと呼ぶ．エージェントの外部として，(1) 他のエージェント，(2) コンピュータ環境，(3) ユーザ，の 3 種類に分類する．(1) の他のエージェントとは，自分以外のエージェントである．(2) のコンピュータ環境は，エージェントが実行しているコンピュータの状態である．例えば，エージェントが実行しているコンピュータ上に存在するファイルシステムが挙げられる．(3) のユーザは，エージェントと相互作用するユーザである．ユーザとの相互作用は，重要なので，特別に分けた．(1) の他のエージェントとの相互干渉は，主にメッセージ通信ポートを用いる．メッセージ通信ポートは，他のエージェントとのメッセージ通信を実現するポートである．(2) のコンピュータ環境との相互作用には，ファイルを操作するためのファイルポート等を用いる．(3) のユーザとの相互作用は，インタフェースポートによって達成される．インタフェースポートは，ウィンドウシステムの場合，ウィンドウの操作等を実現する．例外的に，エージェントの外部を参照しないポートとして，リフレクションポートが挙げられる．エージェントは，リフレクションポートからデータを入力することにより，そのエージェントの内部状態のメタレベル表現を得ることができる．ここでは，メタレベル表現は，RXF の採用している言語である制約論理型言語で扱えるデータ形式，すなわち論理式を用いたエージェント自身の内部状態の表現を意味する．つまり，リフレクションポートからのデータの入力は，リフレクションにおける参照に相当する．エージェントのリフレクションポートを介したエージェントの内部状態の改変は，リフレクションポートに内部状態の改変を意味するメタレベル表現を出力することにより達成される．つまり，リフレクションポートへのデータの出力は，リフレクションにおける改変に相当する．

あるリフレクションポートとあるエージェントは，一対一の関係を持つ．あるリフレクションポートは，リフレクションポートの生成時に指定されたエージェントまたはリフレクションに対応したパートの参照・改変に用いられる．つまり，あるエージェントまたはリフレクションに対応したパートが複数のリフレクションポートを持つことは可能であるが，あるリフレクションポートが，複数のエージェントまたはリフレクションに対応したパートの参照・改変を行うことはできないという意味である．

4.2 スレッド管理機構

スレッド管理機構は，インタプリタに対するスレッドの割当管理をおこなう機構である．RXF 上では，複数のインタプリタが並行動作することが可能である．それぞれのインタプリタは，それぞれスレッドを割り当てられることによって並行動作する．スレッドの割当管理とは，プログラムを実行するインタプリタにスレッドを割当て，プログラムの実行を行っていないスレッドの割当てを解除することである．ここで，インタプリタに割当てられていないスレッドを破壊せずに，新たに割り当てるために保存することによってスレッド割当ての効率を改善する．一般的にスレッドの生成，破壊処理は，オーバーヘッドとなるので，スレッドの再利用をおこなうことは意味がある．

スレッド管理機構は，スレッドの同期，スレッドのブロック，そしてスレッドの実行順序の管理もおこなう．スレッドの同期とは，複数のスレッドの実行の同期をおこなうことである．スレッドのブロックとは，特定のスレッドをある条件を満たすまで実行を中断させることである．例えば，メッセージの受け取り処理で，メッセージがくるまで待つような場合に，スレッドのブロックを使う．スレッドの実行順序の管理は，複数のスレッドが効率良く処理をおこなうようにスケジューリングする．

4.2.1 スレッドに基づく並行処理機能

本節では，RXF インタプリタのマルチスレッドへの適応法について説明する．

1 つの CPU で複数のスレッドを並行動作させるには，実行するスレッドを次々に切替えればよい．スレッドの実行順序のスケジューリングとし実行権の横取りを許すか許さないかで，プリエンプティブスケジュールとノンプリエンプティブスケジュールに分けられる．RXF の Macintosh 上の実装は，Macintosh プログラミング環境の制約のためノンプリエンプティブスケジュールを使用している．

ノンプリエンプティブスケジュールにおいて，実行中のスレッドを切替えるためには，実行中のスレッドを切替えるための手続き (Yield) を呼ぶ必要がある．複数のスレッドを並行動作させるためには，必ず有限時間内に Yield を呼ばなければならない．さらにスレッドを切替えるオーバヘッドを考慮して，Yield をあまり頻繁に呼ばないことが望ましい．RXF では，以上の 2 点を考慮して，Yield を呼ぶタイミングを決定する必要がある．

本実装において，Yeild を呼ぶタイミングは，box モデル [77] に基づいて決定される．box

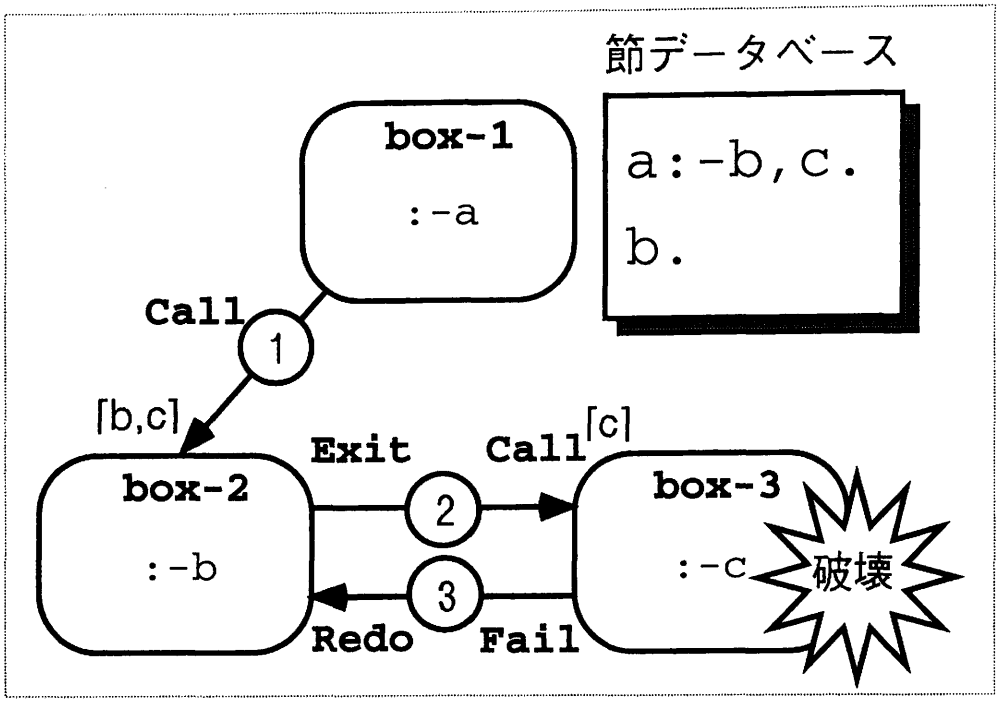

図 4.1: box モデルに基づくスレッド切替えタイミングの検出

モデルとは，Prolog インタプリタの実行モデルである．box モデルではプログラムの実行を 2 入力，2 出力の box の生成，消滅で表現する．入力端子には Call，Redo がある．出力端子には Exit，Fail がある．Exit は Call と接続されるための端子で，Fail は Redo と接続されるための端子である．

box モデルに基づくスレッド切替え処理は，box が制約論理型プログラミングにおけるインタプリタの処理の区切とみなせることを利用する．以下に box モデルに基づくスレッド切替方式を説明する．

図 4.1 の丸で囲まれた数字が Yeild を呼ぶタイミングを表す．図 4.1 の角の丸い四角は box モデルにおける box を表す．ここで節データベースは図 4.1 の節データベースと書かれた四角の中で示し，RXF への質問が"a"であるとする．ここで，ゴール":-a"はスレッド T で評価されるとする．始めに，ゴールを評価するために box が生成される．ここでは，box-1 が生成された．"a"を検索したところ"a:-b,c"が発見されたので"b"を評価するために box-2 が生成される．box-2 が生成された後に Yeild が呼ばれる (図 4.1 の 1)．スレッド T の実行権は，Yeild が呼ばれることによって他のスレッドに渡される．他のスレッドから再びスレッド T に実行権が渡されると，box-2 において"b"が評価される．"b"が成功し，"c"

を評価するためにbox-3が生成される．ここでもYeildが呼ばれ，スレッドTの実行権は他のスレッドへ渡される(図4.1の2)．スレッドTに実行権が戻ると，box-3では"c"が評価されるが節データベースに"c"は存在しないので"c"は失敗する．質問が失敗するとboxは破壊される．つまり，box-3が破壊される．破壊された後Yeildが呼ばれる(図4.1の3)．Yeildが呼ばれることによってスレッドTの実行権は他のスレッドに移る．他のスレッドからスレッドTに実行権が戻ると，この後box-2の"b"が再実行される．あとは"b"と"a"がそれぞれ失敗し質問"a"は失敗する．以上のようにboxの生成・破壊時にYeildを呼ぶことで，複数のプログラムの並行動作が可能になる．

4.3 メモリ管理機構

メモリ管理機構は，RXFにおけるメモリオブジェクトの管理をおこなうための機構である．メモリオブジェクトとは，アトム，ファンクタ，リストなどのRXF上で操作可能なデータの実メモリ上における実体である．本メモリオブジェクトマネージャの特徴は，(1) 可変サイズデータを扱える，(2) 世代型GC（4世代まで），(3) Installable Memory Object (IMOと略す)，(4) メモリをワード単位（Macintosh版では，4バイト）で扱う，の4つである．(1) の可変サイズデータを扱えるという特徴により，固定サイズのメモリオブジェクトを管理していたころと比較して，メモリ資源の有効活用と，処理速度の向上を達成した．(2) の世代型GCは，高速にGCをおこなうための技術である．(3) IMOは，新しい種類のメモリオブジェクトを動的に追加できる機能で，この機能によりオブジェクト指向拡張における新規メモリオブジェクトの追加を実現する．(4) のメモリをワード単位で扱うという特徴は，RISC CPUに最適化するための特徴であり，ミスアラインを防ぐことによってRISC CPUを効率良く活用する．ミスアラインとは，メモリアクセスの際に，CPU毎に定められた特定の値の整数倍の値を持つアドレス以外のアドレスにアクセスすることを意味する．RISC CPUは，ミスアラインによって処理速度が低下するのでミスアラインをなくすことを重要である．

4.3.1 メモリオブジェクト

メモリオブジェクトは，(1) ヘッダ，(2) ボディ，の2つの部分から構成される．(1) のヘッダは，すべてのメモリオブジェクトで同一のフォーマットになっている．ヘッダには，

```
XObjT   mLType;
XDataT  mDType;
uint16  mGCable:    1;
uint16  mFree:      1;
uint16  mIsMoved:   1;
uint16  mAge:       2;
uint16  mMagic:     2;
uint16  mIsConst:   1;
uint16  mWords:     8;
```

図 4.2: メモリオブジェクトのヘッダ部分

オブジェクトの種類やサイズなどに関する情報が格納される.(2) のボディは，メモリオブジェクトの種類毎に異なるサイズ，フォーマットをもつ．ボディ部分に，メモリオブジェクトの種類に依存するデータを格納する．ヘッダを固定サイズにし，ボディを可変サイズにすることによって，効率と柔軟性の両方を実現した．

ヘッダのサイズは，1 ワード（Macintosh 版では 4 バイト）である．ヘッダは，図 4.2 のような構成を持つ．

ここでの mLType は，プログラム言語としての型を表す．プログラム言語としての型とは，アトムやファンクタやリスとなどの型であり，プログラム上で直接操作可能なデータである．mDType は，メモリオブジェクトのフォーマットの種類，すなわちメモリ上における表現の種類を表す．例えば，節とオペレータは，mLType は異なるが，mDType が同じであるオブジェクトが存在し，プログラム言語上では異なる型を持つが，メモリオブジェクトとしてのフォーマットを共有しメモリオブジェクトの操作関数は共有する．mGCAble は，メモリオブジェクトがごみ集め処理の対象である場合は true となり，そうでない場合は false となる．mFree は，ごみ集め処理時に用いられる．ごみ集め時に解放可能と判断されたメモリオブジェクトは，mFree が true になる．mAge は世代型 GC におけるメモリオブジェクトの世代を表す．表現可能な世代数は，2 ビット，つまり，4 世代である．mMagic は，メモリオブジェクトの整合性チェックのために利用される．正常なメモリオブジェクトは，mMagic が 0x02 である．mMagic が 0x02 でないメモリオブジェクトは正常なオブジェクトではない可能性が高いのでエラー処理の対象となる．mIsConst は，メモリオブジェクトが定数，すなわち変数を含まないときに true となり，それ以外は false と

```
uint32  mName; 変数名を表す識別番号.
XCell*  mCellP; 変数セルへのポインタ.
```

図 4.3: 変数のフォーマット

```
uint32  mVarID; 変数を識別する識別番号.
XObj*   mBindP; 束縛へのポインタ.
```

図 4.4: 変数セルのフォーマット

なる．mIsConst が true（すなわち定数）であるオブジェクトのコピーは，実際にはする必要が無いことから，処理の高速化を実現できる．mIsMoved は，メモリオブジェクトが GC によって移動されたときに true となる．GC によって，mIsMoved が false であるオブジェクトは，不要なオブジェクトとして扱われる． mWords は，メモリオブジェクトの大きさを表す．単位は，ワード（Macintosh 版では，4 バイト）である．0 バイトから 16K バイトまで表現できる．16K バイト以上のメモリオブジェクトの場合，オブジェクトの大きさは mWords のアドレスの次のアドレスに保存される．

ヘッダ以外のデータフォーマットについてメモリオブジェクトの種類毎に説明する．特にことわりがない場合すべて Macintosh 版における実装について説明する．

以降の例で使用する変数の型について説明する．uint32 は，32 ビット符号なし整数型である．sint32 は，32 ビット整数型である．XObj* は，メモリオブジェクトへのポインタ型である．

図 4.3 は，変数を表すメモリオブジェクトのフォーマットを表す．ここでの，mName は変数名を表す．すべての変数は，ユニークな識別番号を持つ．mCellP は，変数セルへのポインタである．変数セルは，変数の実体である．mCellP には，変数の初期化時（例えば名前替え処理時）に変数セルへのポインタが代入される．変数に束縛されるオブジェクトへのポインタは，変数セル中で保存される．

図 4.4 は，変数セルを表すメモリオブジェクトのフォーマットを表す．ここでの mVarID は変数を識別する識別番号を表す．すべての変数は，ユニークな識別番号を持つ．mBindP は，束縛へのポインタで，変数が束縛されるとその束縛へのポインタが mBindP に代入される．

```
uint32  mName; アトムの名前を表す識別番号.
```

図 4.5: アトムのフォーマット

```
uint32  mName; ファンクタ名を表す識別番号.
uint32  mArity; アリティ
XObj*   mArgs; 引数へのポインタのリスト
```

図 4.6: ファンクタのフォーマット

図 4.5 は，アトムを表すメモリオブジェクトのフォーマットを表す．ここでの，mName はアトムの名前を表す．

図 4.6 は，ファンクタを表すメモリオブジェクトのフォーマットを表す．ここでの，mName はファンクタ名を表す．mArity は，アリティを表す．mArgs は引数へのポインタのリストへのポインタである．図 4.7 は，アリティ 2 と 3 のファンクタがどのようにメモリに配置されるかを図示したものである．図 4.7 からわかるようにファンクタは，アリティによってサイズが変化する．アリティ 2 のときは，アリティの後に引数へのポインタを 2 つ格納するための配列が続く．同様に，アリティ 3 のときは，アリティの後に引数へのポインタを 3 つ格納するための配列が続く．

図 4.8 は，リストを表すメモリオブジェクトのフォーマットを表す．ここでの，mCarP は，リストの car 部へのポインタであり，mCdrP は，リストの cdr 部へのポインタである．

図 4.9 は，整数を表すメモリオブジェクトのフォーマットを表す．ここでの，mValue は，32 bit 符号付き整数である．

図 4.10 は，浮動小数点数を表すメモリオブジェクトのフォーマットを表す．ここでの，mValue は，倍精度浮動小数点数である．

4.3.2 GC の実装

本メモリサブシステムの構成図を 図 4.11 に示す．本メモリサブシステムは，GC による自動メモリ管理を実現している．本メモリサブシステムでは，(1) 任意の大きさのセルに対応，と (2) 12 バイトのセルだけに対応，2 種類のメモリ空間を持つ．(1) は，従来の RXF でも実装されていた物である．GC の種類としては，複写式世代型の GC である．(1)

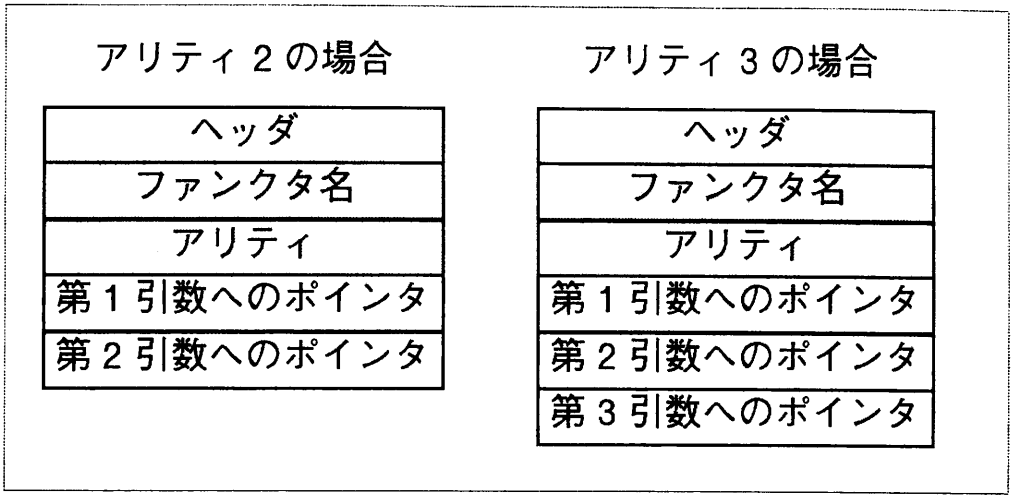

図 4.7: ファンクタの例

```
XObj*   mCarP; リストの car 部へのポインタ
XObj*   mCdrP; リストの cdr 部へのポインタ
```

図 4.8: リストのフォーマット

では，任意のサイズのセルを管理する．(2) は，新たに実装された部分である．(2) は，12 バイトのセルだけを管理する．GC の種類としては，目印付け法を用いている．(2) により，GC 一回あたりに必要な処理時間が平均で約 1/10 になり，最大メモリ使用量は，平均で約 1/4 になった．これは，RXF において 12 バイトのセルが，約 80 % を占めるという事実に基づいて最適化した結果である．12 バイトのセルとしてリストのノード，変数，そして引数 2 のファンクタが挙げられる．固定サイズのセルのメモリ管理には目印付け法が適しており，12 バイトのセルを複写式から目印付け法に変更することによって，GC の実行速度が改善されている．さらに，(1) の部分のメモリ割当量を減らすことができるので，最大メモリ使用量の減少が達成されている．複写式の GC では，GC 処理のために実際に必要なメモリサイズの 2 倍のメモリを必要とするが，(1) の部分が小さくなったため，全体のメモリ使用量の大きな削減が実現できたと分析している．

```
sint32   mValue; 整数値.
```

図 4.9: 整数のフォーマット

double mValue; 浮動小数点数値.

図 4.10: 浮動小数点数のフォーマット

4.4 パートシステム

RXF におけるエージェントは，パートの集合として構築される．すなわち，パートは，エージェントを実現するための部品である．パートは，メッセージ通信や，ファイル操作，そしてユーザインタフェースなどの機能を実現する．言語処理系もパートの一種である．

制約論理型言語処理系の構成要素もパートとして実現されている．すなわち，制約論理型言語処理系が必要とする内部データベース，制約解消系，変数管理機構，そしてフロー管理機構もパートとして実現されている．またデバッガもパートとして実現されている．

すべてのパートは，パートの機能を言語処理系から操作することを可能にするためのインタフェースを実装している．

マルチエージェントシステム構築支援機能は，パートとして実現される．パートは，メッセージ通信や，ユーザインタフェースなどの機能を実現する．言語処理系自身もパートの一種である．パートは，C++ 言語におけるクラスとして実装され，C++ による利用も可能である．表 4.1 で現在定義されているパートをまとめる．

名前管理

パートは，一意な名前を持つ．名前の表現法は，言語処理系依存である．RXF における制約論理型言語処理系，アトムを名前として用いる．すなわち，本制約論理型言語処理系は，アトムを用いてパートを特定する．

名前管理機構は，パートと名前の関係が一対一になるように名前の管理を行うためのパートである．名前管理機構が，別の名前管理機構を管理することによって，階層的な名前管理が可能になる．RXF では，トップレベルの名前管理機構の名前は，それ自身によって管理されている．

名前管理機構は，言語処理系が，パートを操作するために必要な機構である．名前管理機構によって，パートの生成，破壊，そして参照・操作が可能になる．名前管理機構は，パートにある言語処理形上でユニークな名前を与えることによってパートを特定する．

パートの生成は，@new(N,T) または @new(N,T,O) によって達成される．N は，パート

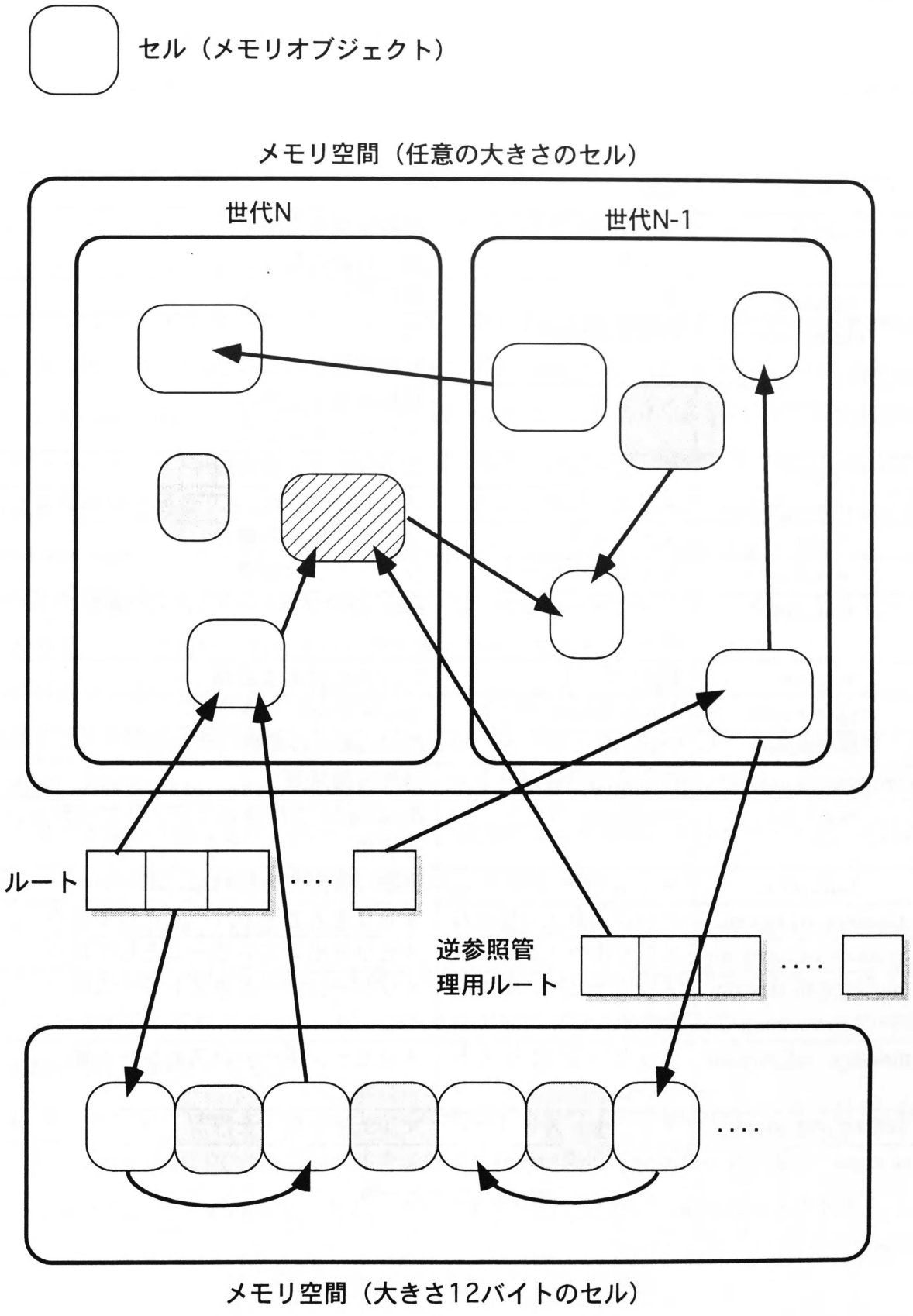

図 4.11: メモリサブシステム

名前	種類	
eval	制約論理型言語 インタプリタ	制約論理型言語の 実行をおこなう
database	節データベース	節のデータベース
namespace	名前管理機構	パートの 名前管理をおこなう
tracer	トレーサ	制約論理型言語の トレースをおこなう
agent	エージェント	エージェントを実現
message	メッセージポート	メッセージ通信
file	ファイルポート	ファイルの読み書き
window	ウィンドウポート	ウィンドウの操作
mac_app	アプリケーション ポート	Mac のアプリケーションを操作
tcp_port	TCP ポート	TCP/IP による通信
apple_event	Apple Event パート	Apple Event の操作
filemaker	FileMaker パート	FileMaker の操作
language_analyzer	自然言語処理パート	自然言語処理
realbasic	RealBasic パート	RealBasic で作成したアプリケーションとの 通信用
null_port	ヌルポート	実際に入出力を行わないポート
memory_in_stream	メモリ入力ストリーム	メモリを入力ストリームとして扱う
memory_out_stream	メモリ出力ストリーム	メモリを出力ストリームとして扱う
message_in_stream	メッセージ入力スト リーム	メッセージポートのストリーム版
message_out_stream	メッセージ出力スト リーム	メッセージポートのストリーム版
scatter_out_stream	マルチキャストパート	マルチキャストを行う

表 4.1: パート一覧

を識別するための名前，T は，パーとオブジェクトの種類である．O によって，生成時のオプションを指定することも可能である．

パートの破壊は，@del(N) によって達成される．N は，パートを識別するための名前である．

パートの参照・改変は，N::M によって達成される．N は，パートを識別するための名前，M は，パートのメソッドを表す．例えば，s がスタックを実現するパートのとき，X=s::pop によって s から 1 つ要素を取り出し，それを X に束縛する．

パートは，メタ機構において，ベースレベルの構成要素を操作するためにも利用される．例えば，ベースレベルのインタプリタは，(1) base_eval，(2) base_db，(3) base_ns，(4) base_message，の 4 つのパートから構成される．(1) の base_eval は，制約論理型言語インタプリタである．(2) の base_db は，節データベースである．(3) の base_ns は，名前管理機構である．名前管理機構は，ベースレベルのパートの名前を管理する．(4) の base_message は，メッセージポートである．これらのパートを操作することで，ベースレベルを参照・改変することが可能である．tracer は，ベースレベルで実行されるプログラムのデバッグに用いられるデバッガを表すパートである．

ベースレベルからは，直接これらのパートを操作することができない．ベースレベルに関係するメタ機構におけるパートは，ベースレベルでは，ベースレベルの名前管理機構（メタ機構における base_names）によって操作可能である．ただし，全てのパートを操作することはできない．表 4.2 に，ベースレベルに関係するメタ機構におけるパートと，それらがベースレベルでどのように扱われるかの対応関係を示す．

表 4.2 は，base_db，base_message，そして base_ns は，ベースレベルにおいてそれぞれ db，message，そして ns として操作可能であるが，base_eval を操作することはできないことを表す．これによって，ベースレベルのプログラムが間違えて base_eval の内容を破壊することがなくなる．

up は，ベースレベルにおいてメタレベルを参照するために利用可能なパートである．down は，メタレベルにおいてベースレベルを参照するために利用可能なパートである．in，out，そして err は，それぞれ入力，出力，そしてエラー出力に用いられるパートである．

表 4.2: メタレベルとベースレベルにおける名前の対応

メタレベル	ベースレベル	備考
base_eval	なし	
tracer	なし	
base_ns	ns	
base_db	db	
base_message	message	
なし	up	ベースレベルでの up_call に利用
down	なし	メタレベルでの down_call に利用
in	in	メタレベルとベースレベルで共有
out	out	メタレベルとベースレベルで共有
err	err	メタレベルとベースレベルで共有

4.4.1 パートとリフレクション

RXF の AOS におけるリフレクションの実現方式は，パートに基づいている．リフレクションにおいて，言語処理系が，パートをその処理系で処理可能な表現に，パートを操作できるようにするという方式である．

ベースレベル上で動作するプログラムは，そのベースレベルを構築しているパートを直接操作することはできない．ベースレベル上で動作するプログラムが，ベースレベルを構築するパートを操作するためには，メタレベル上で動作するプログラムを利用しなければならない．この制限は，プログラマに，プログラミングにおける制約を与えるために必要である．その制約とは，問題解決のためのプログラムをベースレベルで実行し，問題解決のためのプログラムに対するメタ機構をメタレベルで実行しなければならないという制約である．この制約は，問題解決プログラムからメタな処理を分割することによって，問題解決プログラムをシンプルにするために存在する．

パートにおけるリフレクションの実現は，宣言的な手法と手続き的な手法の両方で行った．ここで，宣言的な手法とは，ベースレベルの状態の取得・改変を，ベースレベルの状態を宣言的に記述するメタレベル表現を介してリフレクションを実現する手法である．宣言的な手法とは，メタレベル上でベースレベルの状態を取得・改変するための手続きを介してリフレクションを実現する手法である??．宣言的な実装により，汎用的な 2 つの手続きだけでリフレクションを実現できる．すなわち，メタレベル表現の取得のための手続き

と，メタレベル表現に基づいてベースレベルの状態を改変するための手続きである．リフレクションポートは，これらの 2 つの手続きを実現する．

リフレクションポートは，2 つのインタフェース in/1 と out/1 を持つ．それぞれ in/1 と out/1 は，メタレベル表現の取得と，メタレベル表現に基づいたベースレベルの状態の改変を行う．リフレクションポートの生成時に，リフレクションの対象を，引数として与える必要がある．

例外的に，エージェントの外部を参照しないポートとして，リフレクションポートが挙げられる．エージェントは，リフレクションポートからデータを入力することにより，そのエージェントの内部状態のメタレベル表現を得ることができる．ここでは，メタレベル表現は，RXF の採用している言語である制約論理型言語で扱えるデータ形式，すなわち論理式を用いたエージェント自身の内部状態の表現を意味する．つまり，リフレクションポートからのデータの入力は，リフレクションにおける参照に相当する．エージェントのリフレクションポートを介したエージェントの内部状態の改変は，リフレクションポートに内部状態の改変を意味するメタレベル表現を出力することにより達成される．つまり，リフレクションポートへのデータの出力は，リフレクションにおける改変に相当する．

あるリフレクションポートとあるエージェントは，一対一の関係を持つ．あるリフレクションポートは，リフレクションポートの生成時に指定されたエージェントまたはリフレクションに対応したパートの参照・改変に用いられる．つまり，あるエージェントまたはリフレクションに対応したパートが複数のリフレクションポートを持つことは可能であるが，あるリフレクションポートが，複数のエージェントまたはリフレクションに対応したパートの参照・改変を行うことはできないという意味である．

宣言的な手法は，実行速度と消費メモリ量の両方の点から非効率的である．さらに，些細なことを処理するためにもメタレベル表現を用いなければならないのが不便である．よって，部分的に手続き的な手法も実現している．

図 4.12 は，リフレクション述語 up_call と down_call のパート up と down を用いた実装のプログラムである．up は，ベースレベルにおいてメタレベルを参照するために利用可能なパートである．down は，メタレベルにおいてベースレベルを参照するために利用可能なパートである．メソッド up::quote_call(X) は，X の変数をパート up 中の変数に置き換えた後に（X' とする），X' を評価して，その評価結果を返す．すなわち，up_call(X) が評価されたときに，3.5.5 節で述べられている処理が行われる．ベースレベルにおいて，

```
up_call(X) :-
  call(X) = up::quote_call(X).

down_call(X) :-
  call(X) = down::quote_call(X).
```

図 4.12: up と down を使った up_call と down_call の実装

up はメタレベルを参照するために利用できるので，up_call によって，メタレベル実行が達成される．

4.4.2 パートシステムにおけるポート

入出力管理機構は，ファイルや通信機構などの入出力を管理する．例えば，入出力管理機構は，スレッド管理機構のスレッドのブロック機能を用いて入出力待ちのインタプリタをブロックする．RXF におけるインタプリタ上では，入出力は，ポートと呼ばれる機構によっておこなう．入出力管理機構は，ポートに対するアクセスをファイルや通信機構などの実際の入出力機構へのアクセスに変換する機構を持つ．

RXF ではエージェントの入出力のために，ポートと呼ぶ機構を実装している（図 3.1 参照）．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．逆に言えば，ポートを介することによってしか外界に干渉できない．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もつ．つまり，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．エージェントは，デフォルトでは，メッセージ通信用のポート，メタレベル表現生成用のポート，およびユーザインタフェース用のポートの 3 種類のポートをもつ．それぞれのポートは，入力用と出力用のポートがあるので，エージェントは初期状態で 6 つのポートをもつことになる．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．ポートに求めら

れるただ一つの機能は，ポートはエージェントに対して，そのエージェントのエージェントプログラムが解釈可能なデータ形式で情報のやり取りをおこなわなければならないという機能である．ポートはエージェントに情報を提供するときエージェントプログラムが解釈可能なデータ形式に提供する情報を変換することである．例えば，エージェントプログラムが，論理型言語で記述されているなら，ポートは情報を述語によって表現しなければならない．この場合，エージェントからの入力も述語でおこなわなければならない．言い換えると，ポートは，その実装を規定しない．例えば，ポートはキュー，スタック，データベース，通信機構のいずれであってもかまわないし，これ以外の実装でもよい．つまり，エージェントにとっての入出力機構をポートと呼び，それがファイルであってもキューであってもかまわない．ポートはエージェントプログラムにとっての入出力機構である．

4.5 Clause Mapped Part

Clause Mapped Part（以下 CMP と略す）は，RXF のリフレクション機能を論理型言語において透過に扱うことを可能にするための機能である．CMP は，従来の RXF において定義された各種機能が異なるインタフェースを持ったため，ユーザを混乱させたという教訓に基づいて設計された． 図 4.13 は，CMP の概念図である．図 4.13 は，CMP によってエージェントやエージェントを構成するコンポーネントの内部情報（図 4.13 中のパート p）が，特定の節データベース上（図 4.13 中の節データベース db）における節として述語表現されている様子を表している．リフレクション機能を論理型言語において透過に扱うことを可能にするために，CMP は，パートのメタレベル表現を特定の節データベースにおける特定の節に対応付けることを可能にする．言い換えるとエージェントの世界の情報（パートの内部情報）を，プログラミング言語の世界の情報（節）に対応付けることを可能にする．例えば 図 4.13 では，エージェントの内部情報が db 上の節 “agent(x(...” に対応付けられ，エージェントの評価器が db 上の節 “eval(active,...” に対応付けられている．プログラムからは，CMP によって生成された述語は，節データベース（ここでは db）上で定義されている述語と同様に扱うことが可能になる．すなわち，パートの内部情報を論理型言語のプログラムから自然に扱うことが可能になる．

図 4.14 は，節データベースを，CMP を使って別の節データベース上の節に対応付けた様子を表している．ここでは，節データベース db' が節データベース db 上の節 clause に

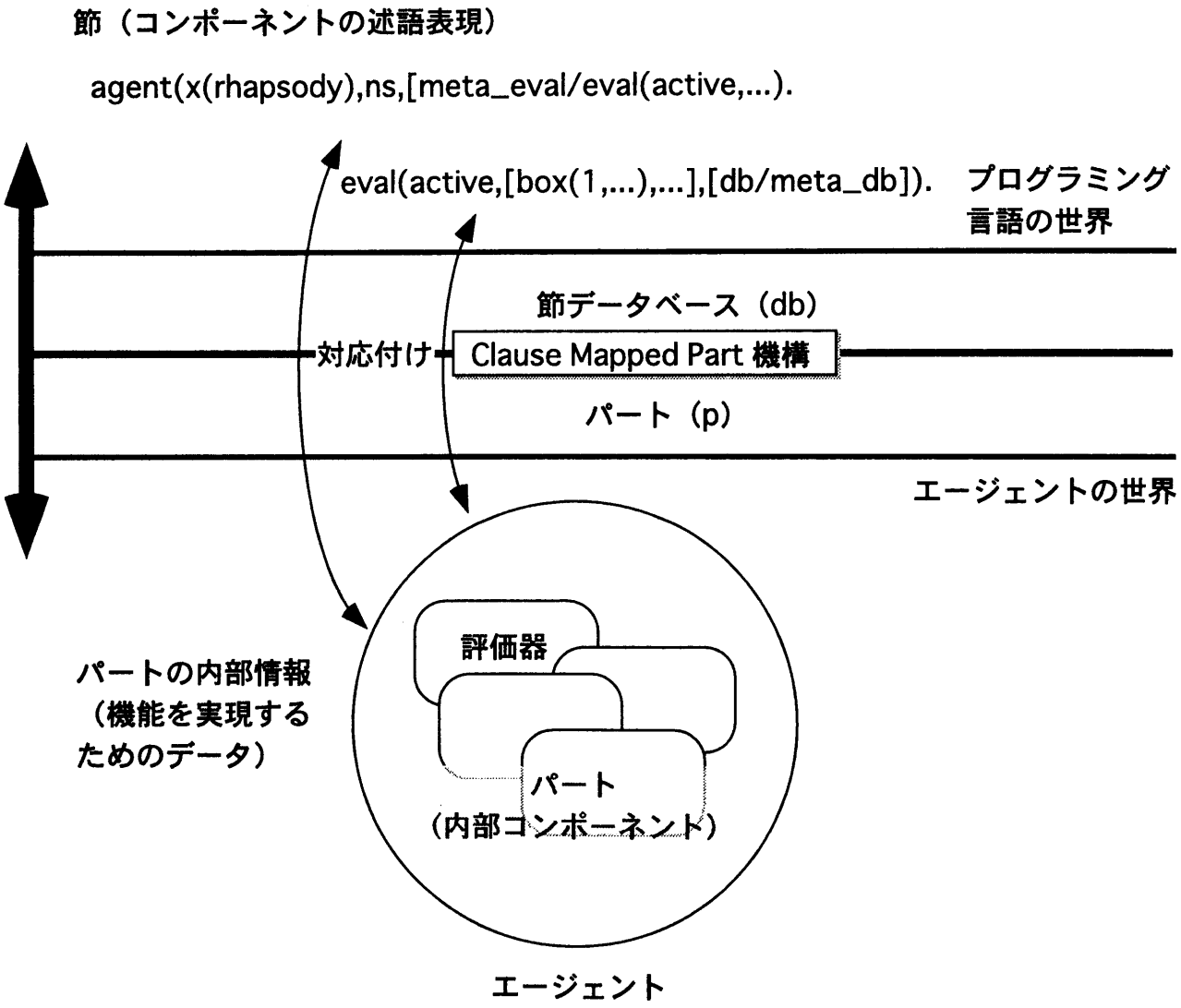

図 4.13: Clause Mapped Part

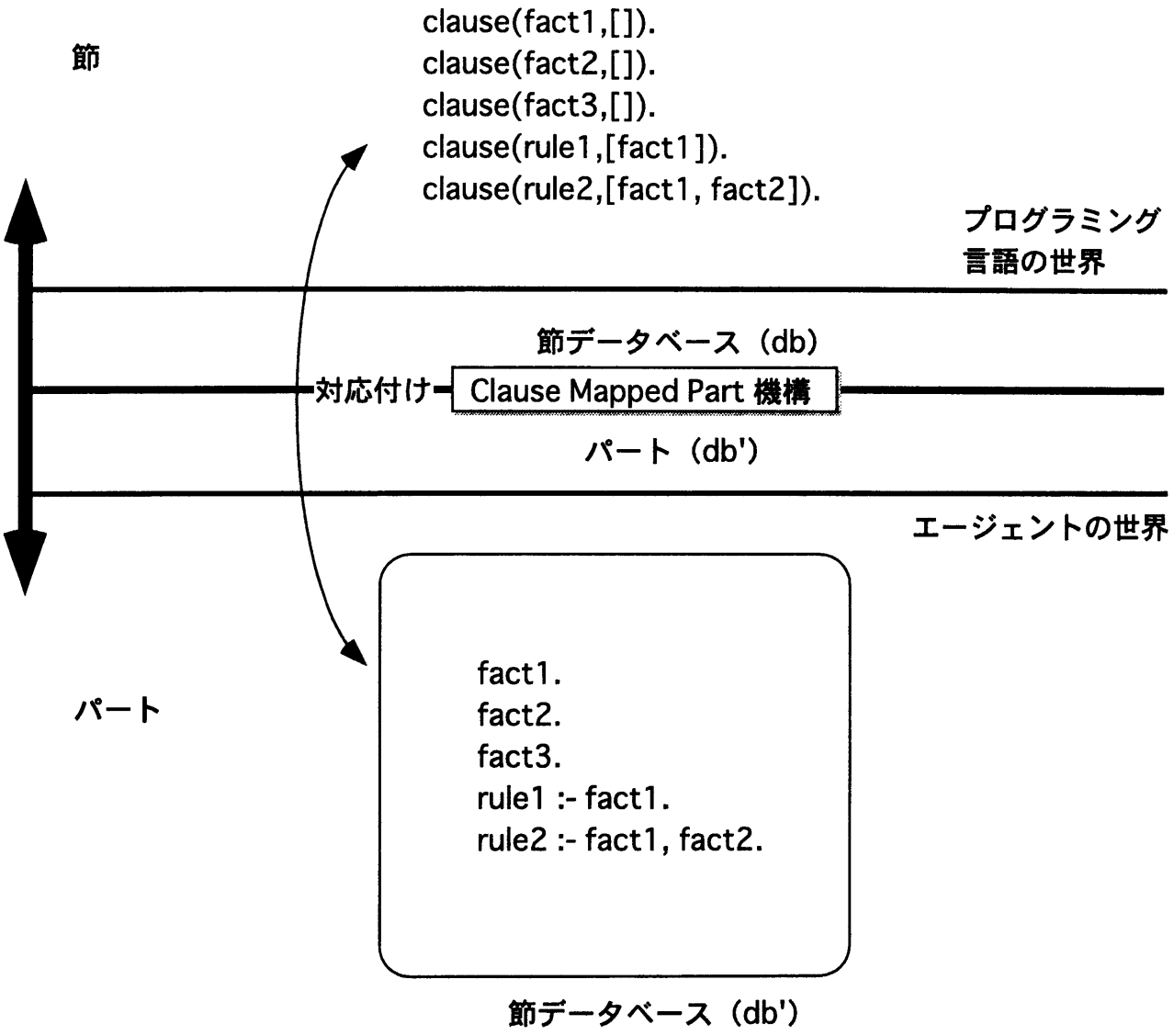

図 4.14: Clause Mapped Part の節データベースへの適用例

```
(1) ?- db::map(clause,part).
(2) ?- db::unmap(clause).
```

図 4.15: map と unmap の使用例

対応付けられている．db' 上の節はファンクタ名 clause を用いることによってアクセスすることが可能になる．このように CMP を用いることによって簡単にモジュールシステムが構成できる．さらに CMP で節データベースを用いることによって階層的な節データベースを構築することが可能になる．これは，包摂アーキテクチャなどの階層的なデータベースが有効なエージェントアーキテクチャの実装に活用できる．

パートを CMP を使って節データベース上の節に対応づけるには，節データベースパートのメソッド "map" を用いる．逆に，対応を取り消すには節データベースパートのメソッド "unmap" を用いる．図 4.15 は，map と unmap の使用例である．図 4.15 の (1) では，節データベース db 上に，CMP で対応付けられた節が定義される．図 4.15 の (1) の `map(clause,part)` により，節 clause にパート part が対応付けられる．図 4.15 の (2) では，節データベース db 上に，CMP で対応付けられている節の対応が解消される． 図 4.15 の (2) の `unmap(clause)` により，節 clause に対応付けられているパートとの関係が解消される．

4.5.1 Clause Mapped Part の実装

CMP の実装は，RXF における節データベースの実装に依存しているので，CMP の実装について説明する前に RXF における節データベースの実装について説明する．RXF におけるパート（主に評価器パート）は，ある述語にマッチする節を節データベース中から取り出すときに，節データベースにマーカの作成を依頼する．概念的にはマーカは，ある述語にマッチする可能性のある節のリストである．マーカから次々と，ある述語にマッチする可能性のある節を取り出すことができる．実際には，マーカに節のリストが格納されているわけではなく，マッチする可能性のある節を検索するためのヒント情報が納められている．マーカには，ヒント情報だけではなく，そのマーカ自身を作成した節データベースへの参照が保存されている（実装上はポインタ）．

図 4.16 は，CMP の実装に必要な節データベースのクラスの定義を表している．CMP に

```
class DataBase {
  virtual XObj* CreateMarker( uint32 inName, uint32 inArity );

  virtual XObj* Next( XRef& inMarker );
  virtual bool HasMoreClause( XRef& inMarker );
  virtual void Retract( XRef& inMarker );

  virtual XObj* Asserta( const XRef& inClause );
  virtual XObj* Assertz( const XRef& inClause );
  virtual void Abolish( uint32 inName, uint32 inArity );
}
```

図 4.16: CMP 実装のためのクラス定義

対応したパートは，図 4.16 に示される (1) CreateMarker, (2) Next, (3) HasMoreClause, (4) Retract, (5) Asserta, (6) Assertz, そして (7) Abolish, の 7 つのメソッドを実装する必要がある. (1) の CreateMarker メソッドは，マーカを作成するためのメソッドである. (2) の Next メソッドは，次候補を得るためのメソッドである. (3) の HasMoreClause メソッドは，次候補が存在するかどうかを調べるためのメソッドである. (4) の Retract メソッドは，情報を削除するためのメソッドである. (5) の Asserta メソッドと (6) の Assertz メソッドは，情報を追加するためのメソッドである．Asserta メソッドは，追加された情報をデータベースの先頭に追加し，Assertz メソッドは，追加された情報をデータベースの末尾に追加する. (7) の Abolish メソッドは，情報を削除するメソッドである．Retract メソッドとの違いは，Abolish は同じファンクタ名を持つ情報をすべて削除する点にある.

CMP は，パートの内部情報を節データベース状の節に対応づけるために，パートが節データベースのメソッドを継承する必要がある．パートを CMP に対応させるためには，節データベースの (1) ～ (7) メソッドをすべて実装する必要がある.

4.5.2 Clause Mapped Part の実行例

Clause Mapped Part の実行例として，図 4.17 を用いて節データベースの階層化の例を示す.

図 4.17 の (1) では，新規節データベース db1 が生成されている．図 4.17 の (2) では，

```
(1) ?- X = @new(db1,db,[]).
(2) ?- db::map(test,db1).
(3) ?- listing.
    なにも表示されない
(4) ?- assert(test(hello)).
(5) ?- listing.
    test(hello,[]).
(6) ?- db::unmap(test).
(7) ?- listing.
    なにも表示されない
```

図 4.17: CMP の使用例

既存の節データベース db に db1 を節 test に対応付けている．図 4.17 の（3）で 組み込み述語 listing によって，節データベースの内容を表示させるが，現在 db と db1 の両方とも空なので何も表示されない．図 4.17 の（4）では，test(hello) を db に追加することを試みているが，test は db1 に対応付けられているので db1 が assert(test(hello)) の処理を行う．assert の処理内容は，パート依存である．もし db1 がエージェントポートだったら，ここでエージェントの内部情報の改変が行われることもある．db1 は，節データベースなので，assert の処理が行われる．図 4.17 の（5）で listing してみると，確かに節が追加されていることを確認することができる．図 4.17 の（6）では，db1 と test の対応を解消している．その結果図 4.17 の（7）で listing を実行してもなにも表示されない．

4.5.3 Clause Mapped Part に基づくモバイルエージェント

CMP を用いることによって，強いマイグレーションに基づくモバイルエージェントを容易に実装できる．マイグレーションの実現のためには，(1) エージェントを述語にエンコードする機構，(2) 述語としてエンコードされたエージェントを転送する機能，そして (3) 述語としてエンコードされたエージェントをデコードして元に戻す機能，の 3 つが必要である．(1) と (3) は，Clause Mapped Part によって，達成され．(2) は，メッセージ通信機能よって達成される．(1) と (3) は，それぞれエンコーダエージェントとデコーダエージェントによって実現される．これらのエージェントは，通常のエージェントにとってメ

タエージェントである.(2) の通信は，これらのエージェント間のメッセージ通信である.

4.6 エージェントフレームワーク

マルチエージェントシステム構築環境において，階層化されたエージェントアーキテクチャが多い．本 AFW は，階層化されたエージェントを容易に構築するためのフレームワークを提供する． AFW では，標準的なエージェントの枠組みを定義する．ここでは，階層構造，割込み処理，そしてメッセージ通信処理をエージェントの基本的な枠組みとする．AFW に基づくデフォルトのエージェントは，2 つのスレッドから構成される．図 3 において，スレッドは，点線の四角で囲まれている．それぞれのスレッドでは，インタプリタが動作し，それぞれ，meta，base1 と名付けられている．meta は，エージェントの持つポートを管理し，エージェントのメッセージ通信処理と割込み処理を実現するスレッドである．base1 は，タスクを実行するためのスレッドである．meta は，base1 の管理も行う．すなわち，デフォルトでは，エージェントの機能を実現するスレッドと，タスクを実行する 2 つのスレッドから構成される階層を持つ．図 3 は，低レベルエージェントアーキテクチャに基づくエージェントを示している．図 3 におけるエージェントは，デフォルトの状態から，base2 を追加した状態を表す．エージェントの階層化には,(1) エージェントの処理の階層に基づく階層化,(2) エージェントの実装に関する階層化，の 2 つがある.(1) は，様相等に基づくエージェントの階層化が挙げられ，一般的なエージェントアーキテクチャにおける階層は，この階層化に基づく.(2) は，エージェントの機能の実現に関する階層化が挙げられ，オブジェクト指向におけるリフレクションに基づく階層化である．本アーキテクチャは,(1) (2) の両方の階層化を支援する．RXF では,(1) (2) の両方の階層化を支援することが可能である．デフォルトでは,(2) の階層化を実現している.

エージェントフレームワークに基づくエージェントの利点は，他のエージェントによってその構成を推論できる点にある．この特長により，本フレームワークに適したエージェントデバッガの実装も可能になる．本エージェントフレームワークに基づくエージェント視覚化ツールの実装を検討中である.

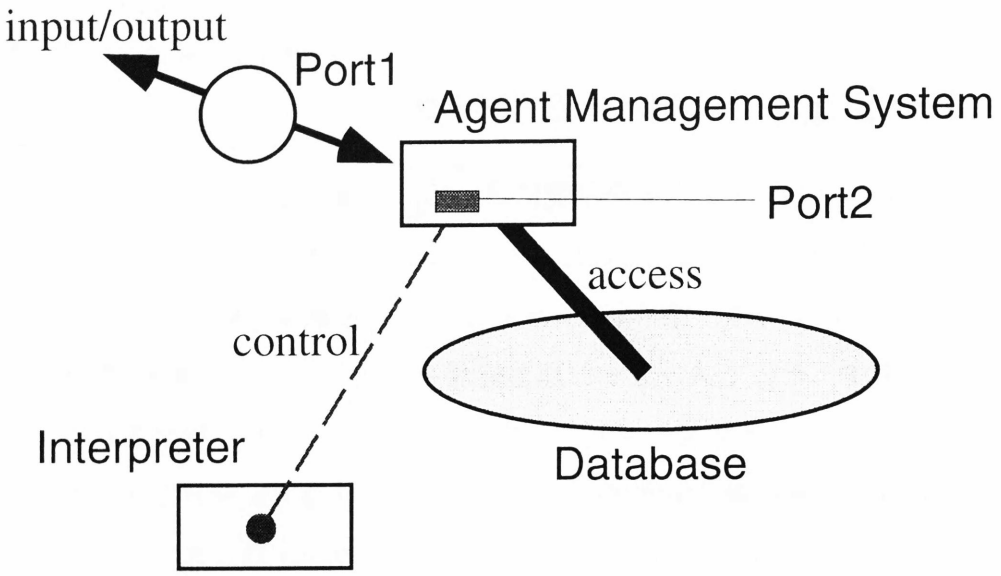

図 4.18: RXF エージェントの内部構造

4.6.1 デザイン

RXF におけるエージェントは，制約論理型言語インタプリタとエージェント管理機構から構成される．制約論理型言語インタプリタは，制約論理型言語のプログラムを実行する．エージェント制御機構は，メモリ，ファイル，メッセージ，イベント，そして割込み等を管理する．RXF の実装において，システムを単純にするため，エージェント制御機構も，制約論理型言語によって記述されている．エージェント制御機構と制約論理型言語インタプリタは，異なるスレッドで並行動作する．本実装方式によって，柔軟なエージェントの構築が可能になる．

図 4.18 は，RXF におけるエージェントの内部構成を示している．図 4.18 中において，インタプリタ，エージェント制御機構，データベースそして 2 つのポートがある．データベースは，インタプリタによって用いられるプログラム節を格納する．ポートは，インタプリタのための入出力機構を実現する．エージェント制御機構は，プログラムをデータベース，ポート `Port1` からの入力，そしてポート `Port2` への出力を用いてプログラムを実行する．`Port2` は，インタプリタを制御するための特殊なポートである．インタプリタの状態は，`Port2` を用いることによってインタプリタの状態を操作できる．すなわち，エージェント制御機構は，`Port2` を用いることによってインタプリタを制御する．

エージェントアーキテクチャを示す．RXF におけるエージェントは，複数の基本的なコ

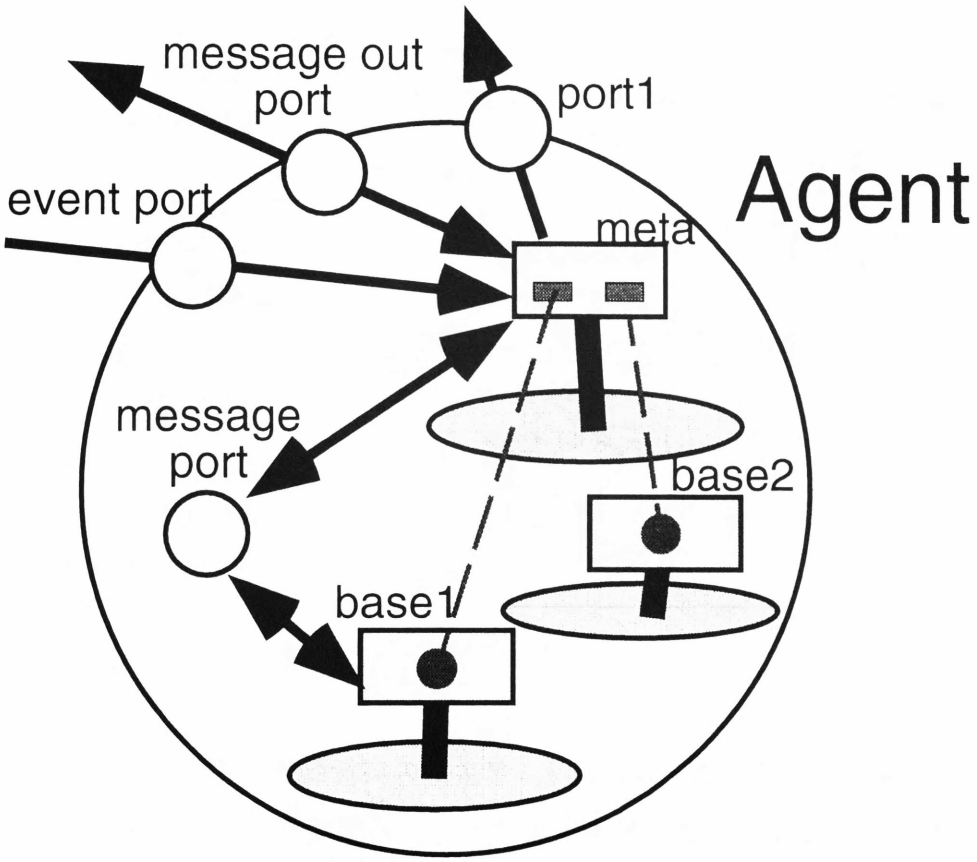

図 4.19: 基本的なエージェントアーキテクチャ

ンポーネントから構成される．図 4.19 は，RXF におけるエージェントアーキテクチャの基本的な概念を表している．図 4.19 は，3 つのインタプリタ（meta，base1 そして base2），3 つのデータベース（meta，base1 そして base2 で用いられる），そして 6 つのポート（図中で 4 つの円と 2 つの小さな四角で表される）があることを表している．図 4.19 における meta は，エージェント管理機構を実現する．エージェント管理機構は，ポートを用いることによって base1 と base2 を管理する．meta は，図 4.19 中の event port，message out，message port そして port1 も管理する．イベントポートとメッセージポートは，エージェントを実現するための基本的なポートである．meta は，イベントポートからイベントを取り出し，そして処理する．例えば，meta がイベントポートからメッセージを取り出したとき，meta は，イベントを base1 がメッセージキューとして利用しているメッセージポートにいれる．message out ポートは，他のエージェントへのメッセージの送信に利用される．すなわち，message out ポートは，meta，base1 そして base2 によってメッセー

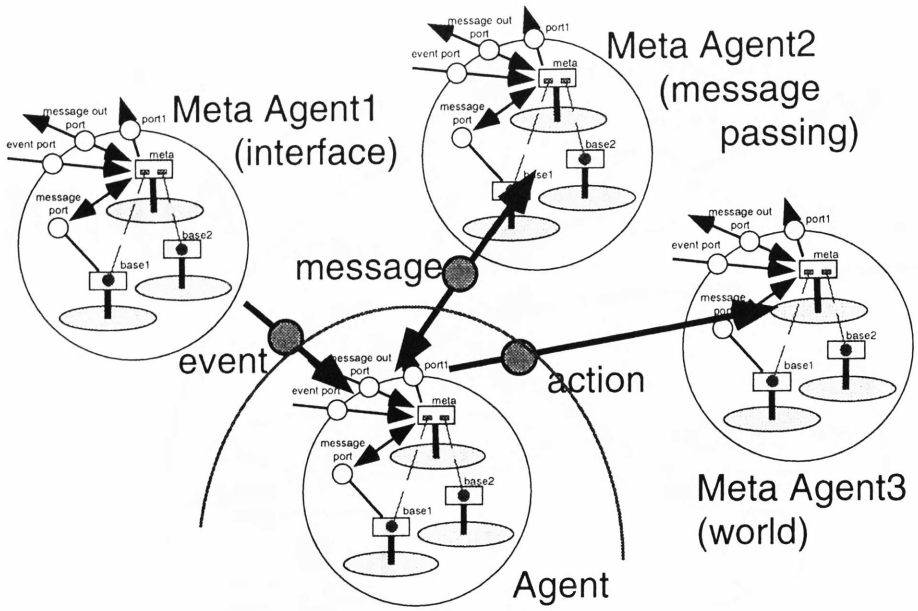

図 4.20: エージェントと環境

ジの送信のために用いられる．port1 は，タスクの実行制御に用いられる．

図 4.20 は，エージェントの世界を実現するための環境を表す．図 4.20 は，オブジェクトレベルのエージェントと 3 つの基本的なメタエージェントがあることを表す．メタエージェントは，インタフェース，メッセージ通信，そして世界の実現を行う．基本的なメタエージェントは，最初から RXF 上で動作している．オブジェクトレベルのエージェントは，メタエージェントと協調してタスクを遂行する．

例えば，ユーザからの入力は，meta-agent1（interface）によってイベントとして管理される．メッセージは，meta-agent2（message passing）によって管理される．meta-agent2（message passing）は，他のエージェントへのメッセージの送信と，他のエージェントからのメッセージの受信を実現する．エージェント内のスレッド間のメッセージとエージェント間のメッセージが異なることに注意．meta-agent3（world）は，問題解決に用いられる．例えば，問題空間におけるエージェントの行動のシミュレートを実現する．

4.6.2 実装

図 4.21 は，エージェントのメタ機構のプログラムの概略を示している．エージェントのメタ機構は，エージェントの低レベルな処理を実現する．エージェントの低レベルな処理

```
(1) run_agent :-
      loop,
        event::in(Event,[wait]),
        do_attention(Event),
        fail.
(2) do_attention(message(F,T,M)) :-
      !,
      (message(F,T,M)
      ;base_message::put(message(F,T,M))),
      !,
      true.
(3) do_attention(Event) :-
      !,
      call(Event),
      !,
      true.
(4) query(X) :-
      base_eval::status = stop,!,
      base_eval::query(X).
(5) query(X) :-
      base_eval::thread_query(X).
```

図 4.21: メタ機構プログラム

とは，メッセージ通信処理，割込み処理，そしてユーザからの質問に答えるための処理を開始するための処理である．本プログラムは，(a) イベントポートから，イベントを取り出す，(b) 取り出したイベントに応じた処理をする，の 2 つの処理を繰り返す．(a) のイベントポートは，エージェントが，イベントを受け取るための機構である．ここで，イベントとは，メッセージ通信機構や，ユーザインタフェースなどのエージェントのメタ機構からエージェントへのメッセージである．イベントには，ユーザの質問，他のエージェントからのメッセージ，そしてウィンドウなどのユーザインタフェースにおけるマウスのクリックなどが挙げられる．(b) では，(a) で取り出したイベントに応じた処理をおこなう．イベントが，メッセージのときは，エージェントのメッセージポートにメッセージを配送する．ユーザからの質問のときは，その質問を処理するためのスレッドを生成し，質問を処理する．

図 4.21 における (1) は，処理 (a) (b) を繰り返すプログラムである．`event::in(Event,[wait])` が処理 (a) に相当し，`do_attention(Event)` が処理 (b) に相当する．`event::in(Event,[wait])` は，イベントポート `event` からイベントを得ることを表す．ここでの，`[wait]` は，イベントが得られるまで，処理をブロックすることを表す．

図 4.21 における (2) (3) は，処理 (b) を実現するプログラムである．(2) は，メッセージを

処理し,(3) はそれ以外を処理する.(2) は,メッセージにマッチするプログラム (アテンションハンドラに相当する) を評価し,その評価に失敗したときに base_message::put(message(F,T,M)) によってベースレベルのメッセージポート base_message にメッセージを追加する.(3) は,イベントにマッチするゴールを評価する.

図 4.21 における (4) (5) は,query(X) の形式のイベントを評価するプログラムである.(4) の base_eval::status = stop ベースレベルの言語処理系の実行状態が stop のとき成功する.つまり,(4) は,ベースレベルの言語処理系の実行状態が stop,すなわちなにも評価していないときに base_eval::query(X) によって X を評価するプログラムである.(5) は,(4) が失敗したとき,すなわちベースレベルの言語処理系がプログラムの実行中のとき,base_eval::thread_query(X) によって新たなスレッドを生成して,X を評価する.

パートと名前管理機構によって,エージェントの低レベルな処理をこのように簡潔に記述することが可能になった.

本章では,RXF のマルチエージェントシステム構築環境について説明する.はじめに,マルチエージェントシステム構築環境について説明する前に,RXF におけるエージェントについて説明する.次に,マルチエージェントシステム構築環境について説明し,最後に,マルチエージェントシステムの開発と関連して,RXF におけるエージェントとリフレクションの関係について説明する.

RXF におけるエージェントの特徴は,(a) 自律性 [37] を持つ,(b) 他のエージェントに対して独立である,(c) 複数のエージェントは並行動作する,の 3 点によって特徴づけられる.(a) の自律性はエージェントがそれ自身で外部に対して自ら主体的に行動することを表す.(b) の独立性はエージェントのデータが他のエージェントから保護されていることを表す.(c) の並行性は複数のエージェントが,概念的には同時に実行していることを表す.RXF では,(a) を実現するために,エージェントが自分自身の状態を,参照・変更するための機能としてリフレクションを提供する.

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の 2 つの部品から構成される (図 3.1 参照).インタプリタは言語 RXF のプログラムを実行する部品である.本インタプリタに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部分である.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．インタプリタによるプログラムの実行と，制御機構は，異なるスレッド上で並行動作する．制御機構がインタプリタと並行処理しているので，制御機構による割込み処理が可能になる．

4.7 成果

RXF のパートを用いることにより，レガシーアプリケーションとの連携を容易に行えるようになった．実際に，HyperCard を用いたインタラクティブな GUI の開発や，FileMaker を用いたデータベースの操作に応用している．

RXF のメモリ管理機構は，特定のサイズのセルに対する最適化を行うことによって，10 倍の実行速度の改善と，メモリ使用量 1/4 を達成した．具体的には，12 バイトのセルに対しては，世代型複写式 GC を適用せずに，目印付け法を適用した．

Clause Mapped Part によって従来の RXF における様々な機能（階層化インタプリタ，階層化データベース，リフレクション）を統一された方法で記述することが可能になった．さらに Clause Mapped Part によって，エージェントの永続化が可能になり，モバイルエージェントの実装が可能になった．

第5章　RXF におけるデバッグ機能

本研究は，マルチエージェントシステムの開発支援環境 RXF の実装研究である [5, 6]. マルチエージェントシステムは，エージェントと呼ばれる知的で自律したソフトウェアから構成されるシステムである．それぞれのエージェントは，並行動作し互いに協調しあいながらタスクを遂行する．

マルチエージェントシステムの開発において，デバッグが問題となる．マルチエージェントシステムは並行プロセスにより構成されるので，マルチエージェントシステムのデバッグは，並行プロセスのデバッグと同様な課題を持つ．一般的に，並行プロセスのデバッグは，困難であり，さまざまな研究が行われている [49].

本論文では，RXF を用いて実装されたマルチエージェントシステムにおけるエージェントをインタラクティブにデバッグするためのトレーサに関して述べる．トレーサは，プログラムの実行を追跡 (トレース) し，プログラムの実行状態に関する情報を得るために用いるプログラムである．トレーサによるデバッグは，“probe effect” [50] と呼ばれる問題を発生させる．“probe effect” とは，デバッガによるプログラムへの干渉が，プログラムの挙動を変えてしまうことを意味する．“probe effect” は，日本語で “プローブ効果” またはデバッグにおけるハイゼンベルグ効果と呼ばれることもあるが，本論文では “probe effect” と呼ぶ．トレーサが行う，プログラムの実行状態に関する情報の取得も，プログラムへの干渉である．“probe effect” は，逐次プロセスのデバッグにおいても問題となるが，並行プロセスのデバッグにおいても，大きな問題となる．例えば，トレーサによるプログラムの干渉が，並行プロセス間の同期を狂わせる可能性がある．すなわち，トレーサが，本来はなかったはずのバグを作ってしまう．

本論文で提案するトレーサは，エージェントへのトレーサの適用によって発生するエージェントの実行遅延を原因とする “probe effect” の回避を目指して実装された．本トレーサは，“probe effect” の回避のために，デバッグ対象のエージェントの実行遅延にあわせて，デバッグ対象ではないエージェントの実行速度を調整する．

本トレーサは，ステッパの機能も備えているが，従来から Prolog 上のステッパ機能付きのトレーサはトレーサと呼ばれていたので，便宜上トレーサと呼ぶことにする．

本研究で扱っているトレーサは，逐次プロセスのデバッグ技術を応用したものである．本トレーサは，プログラムのステップ実行機能を持つ．ステップ実行機能とは，プログラムを 1 命令ずつ実行させるための機能である．トレーサは，プログラムを 1 命令実行したあと，そのプログラムの実行を停止させ，プログラムの実行過程を調査し，そしてプログラ

ムの実行を再開させるという動作を繰り返す．トレーサを用いることによって，プログラマは，プログラムの実行過程を詳細に調べることができる．

5.1 マルチエージェントシステムのデバッグ

本章では，マルチエージェントシステムのデバッグについて考察し，マルチエージェントシステムをインタラクティブにデバッグ可能なトレーサの実現に関して論じる．

RXF におけるマルチエージェントシステムの開発環境は，エージェントの記述を簡単にするが，デバッグの支援環境が不十分である．例えば，マルチスレッド機能は，並行プロセスの記述を簡単にするが，そのデバッグを困難にする．効率の良いエージェント開発のために，RXF の機能を効率良くデバッグするための支援が必要である．

5.1.1 box モデルに基づくトレーサ

本トレーサは，制約論理型言語インタプリタのためのインタラクティブなトレーサとしてデザインされている．本トレーサは，box モデル [77] に基づいて実装されている．box モデルは，Prolog プログラムのデバッグのためのモデルである．box モデルは，Prolog プログラムの実行過程を表現しており，Prolog を代表する論理型言語のトレーサの実現に利用できる．box モデルに基づくトレーサは，論理型言語用で記述された逐次実行プログラムのためのデバッグ機能を提供する．RXF における制約論理型言語のデバッガは，box モデルに基づくトレーサとして実現できる．

box モデルではプログラムの実行を 2 入力，2 出力の box の生成，消滅で表現する．入力端子には Call と Redo がある．出力端子には Exit と Fail がある．Exit は Call と接続されるための端子である．Fail は Redo と接続されるための端子である．ゴールの評価は，Call 端子からゴールを box へ入力することとして表現される．ゴールの成功は，Exit 端子からの次に評価すべきゴールを出力することで表現される．ゴールの失敗は，Fail 端子からの出力によって表現される．バックトラックは，Redo 端子への入力として表現される．それぞれの端子を用いて，制御対象の box を変更する．

図 5.1 は，box モデルに基づく Prolog プログラムの実行過程を示している．図 5.1 中の box1 および box2 とラベル付けされた 2 つの長方形は，box モデルにおける box を表している．節データベースは，図 5.1 中の点線枠で示される．質問は，“?- a,b.” であると

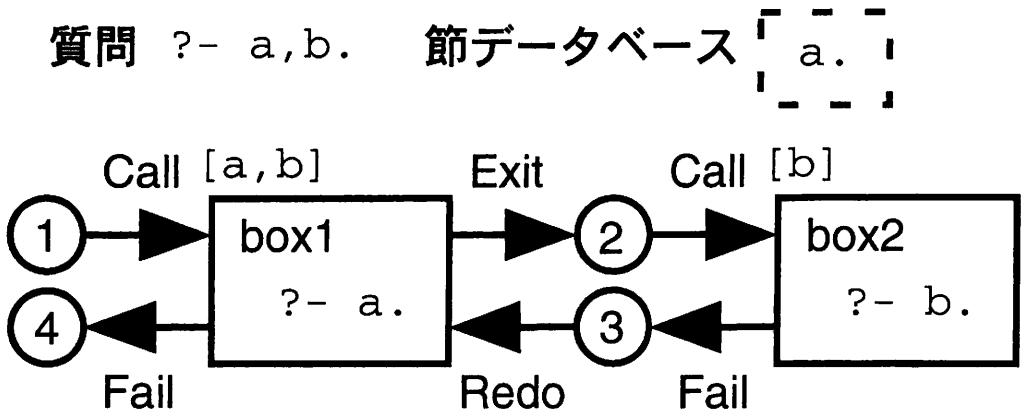

図 5.1: box モデルに基づく Prolog の実行過程

する．Prolog プログラムにおける質問は，いくつかのサブゴールにより構成される．ここでの質問は，サブゴール “a” およびサブゴール “b” が同時に成立するかどうかのチェック (もしくは，証明) するためのものである．

最初に，box1 が生成される．box1 の Call 端子にリスト “[a,b]” が渡され (図 5.1 中の 1)，box1 に制御が移る．このリストは，評価すべき連言節 “a,b” をリストで表現したものである．ここで box1 は，“[a,b]” の先頭の要素 “a” を評価する．節データベースに “a” が存在するので，“a” の評価は成功する．その結果，box1 の Exit 端子から，評価すべき残りの要素を表すリスト “[b]” が出力される．このとき box2 が生成され，box2 の Call 端子に “[b]” が渡される (図 5.1 中の 2)．そして box2 に制御が移る．box2 は，“b” を評価するが，節データベースに “b” は存在しないので，失敗する．失敗すると，box2 の Fail 端子から box1 の Redo 端子に制御が移り (図 5.1 中の 3)，box2 は消滅する．その後，box1 で “a” が再評価される．しかし，“a” は決定的なので失敗し，box1 の Fail 端子から移り，質問が失敗する．

本研究で実装したトレーサは，box の入力各端子で，プログラムの実行を停止させる．具体的には，図 5.1 中の 1，2，そして 3 でプログラムを停止させる．プログラムを停止させた後，トレーサは，プログラムの実行状況を表示する．例えば，図 5.4 における　とラベル付けされたウィンドウには，box モデルに基づくトレーサによって出力されたプログラムの実行状況例が表示されている．

本トレーサにおける 1 ステップは，1 つの box の Call または Redo 端子から，Exit または Fail 端子までの実行を意味する．例えば 図 5.1 では，i から $i+1$ までの間 $(1 \leq i \leq 3)$

が，それぞれ 1 ステップに相当する．

5.1.2 マルチエージェントシステムへのトレーサの適用における問題点

並行プロセス中の逐次プロセスのデバッグに，トレーサを用いることは有効であると考える．しかし，単純に逐次プロセス用のトレーサを，並行プロセスのデバッグに適用することには問題がある．問題として，“probe effect” が挙げられる．この問題の原因は，トレーサによってデバッグ対象のプロセスが一時的に停止されてしまうことである．デバッグ対象のプロセスが一時停止している間も，非デバッグ対象のプロセスは，実行し続けている．結果として，デバッグ対象のプロセスの実行速度だけ遅くなってしまう．これにより本来は無かったはずの誤りが発生する可能性も出てくる．並行プロセス中の逐次プロセスのデバッグに，トレーサを用いるためには，この問題を解決する必要がある．

並行プロセスへのトレーサの適用には，並行プロセス中の複数の逐次プロセスを対象にする方法，および並行プロセス中の単一逐次プロセスを対象にする方法，の 2 種類がある [49]．

並行プロセスへのトレーサの適用において，並行プロセス中の複数の逐次プロセスを対象にする場合，複数のプロセスに対してそれぞれ独立のトレーサを用いる手法がある [78]．このようなトレーサは，複数のトレーサによって生成される大量のデバッグ情報の分析の困難さ，マルチウィンドウを用いてデバッグ情報を表示する際のユーザへの負担が問題となる．単一の逐次実行プロセスをトレースするだけでも，ユーザには大きな負担がかかる．なぜなら，トレーサによって生成されるデバッグ情報を理解するためには，ユーザはプログラムの実行過程を詳細に把握しておく必要があるからである．複数のプロセスのデバッグ情報をマルチウィンドウを用いて表示したとしても，それらを瞬時に理解することは困難である．結果として，そのようなトレーサは，ユーザへの負担も大きくインタラクティブなデバッグの効率改善につながりにくい．よって，本研究では，インタラクティブなデバッグという観点から，並行プロセス中の単一逐次プロセスを対象にする．

並行プロセス中の単一の逐次プロセスに対してトレーサを適用した場合，その逐次プロセスが実行を一時停止している間に，並行プロセス中の他の逐次プロセスの実行をどうするかによって，(a) トレース対象の逐次プロセスだけを一時停止する，そして (b) トレース対象の逐次プロセスだけでなくすべての逐次プロセスの実行を一時停止する，の 2 つのアプローチがある [49]．アプローチ (a) では，“probe effect” が問題となる．なぜならば，

デバッグ対象のエージェントの実行が，トレーサによって変化してしまうからである．アプローチ (b) では，一時停止によるタイムアウトが問題となる．実際には並行動作する複数のプロセスを十分に短い時間で停止させるのは困難である [49].

本論文では，次に示す新たなアプローチ (c) を提案する．アプローチ (c) は，トレース対象のプロセスの実行は一時停止するが，そのときトレース対象ではないプロセスの実行速度をトレース対象プロセスの実行速度の低下に合わせて調整するというアプローチである．ここで調整とは，並行プロセス中の逐次プロセスの実行速度の比 (実行速度比と呼ぶ) を一定に保つことを意味する．本アプローチ (c) は，アプローチ (a) の問題点である “probe effect” の回避を試みながら，かつアプローチ (b) の問題点であるタイムアウトを避けるための新たなアプローチである．

5.2 “probe effect” を回避可能なマルチエージェントシステム用トレーサの実装

本章では，box モデルに基づくトレーサを並行システムに適用したときの “probe effect” の回避方法について述べる．本トレーサは，5.1.2 節で述べたような性質をもつマルチエージェントシステムのデバッグを支援する．本トレーサは，5.1.2 節で述べた新たなアプローチ (c) を実現したトレーサである．すなわち，トレース対象のプロセスの実行は一時停止するが，そのときトレース対象ではないプロセスの実行速度をトレース対象プロセスの実行速度の低下に合わせて調整することが可能なトレーサである．ここでの調整には，並行プロセス中の逐次プロセスの実行速度比を一定に保つ必要がある．ここで，RXF におけるエージェントは，複数のスレッドから構成されているが，簡単のために，以降エージェントは，単一のプロセスにより構成されているとする．

5.2.1 並行プロセスの速度比調整機構

本研究における速度比調整機構とは，マルチエージェントシステム中のエージェント間の実行速度比を一定に保つためのシステムである．本トレーサにおける速度比調整機構は，デバッグ対象のエージェントの実行速度の低下に関する情報を，システム中のその他のエージェントにブロードキャストして，デバッグ対象のエージェントにあわせてその他のエージェントの速度を調整することによって，速度比調整を行う．

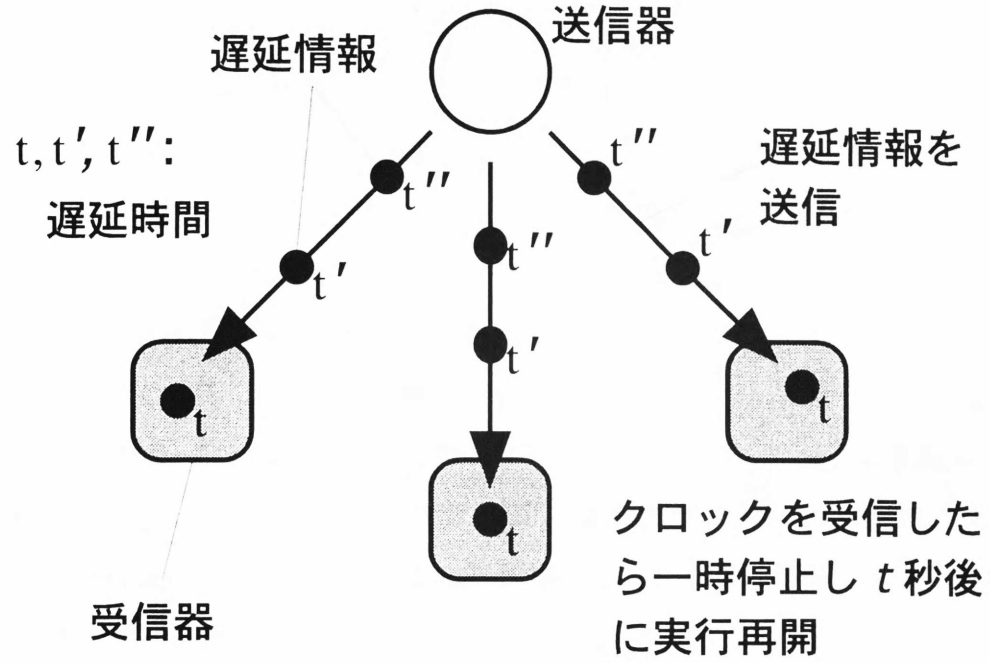

図 5.2: 並行プロセスの速度比調整機構

本手法の利点は，異種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にも適用可能，前もってエージェントの実行速度を知る必要がない，通信における遅延時間を考慮する必要がない，大域時刻を必要としない，そして処理が簡単である，の 5 点である．

さらにインタラクティブなデバッグという観点から，デバッグに必要な処理は，短時間かつ少ないメモリ使用量で実行可能でなければらない．デバッグに必要な処理が，短時間かつ少ないメモリ使用量で達成できれば，デバッグ対象のプロセスに対する干渉を減少させることが可能になる．結果として，“probe effect” が可能になる．

図 5.2 は，本トレーサのために実装した，並行プロセスの速度調整機構の概念図である．本機構は，遅延信号送信器 (以降，送信器と呼ぶ) と遅延信号受信器 (以降単に受信器と呼ぶ) から構成される．送信器 (図 5.2 中の円) は，トレーサに内蔵されている．受信器 (図 5.2 中の灰色の四角) は，全てのエージェントに内蔵されている．受信器は，エージェントに対して独立に存在している．受信器は，エージェントの実行を制御できる．エージェントが，受信器の動作に支障を与えることはできない．

送信器は，デバッグ対象のエージェントの実行速度に合わせて，非デバッグ対象のエージェントを速度調節するのため情報 (遅延信号と呼ぶ) を受信器に送信するための機構であ

る．受信器は，遅延信号に基づいて非デバッグ対象のエージェントの実行速度を制御するための機構である．

デバッグ対象のエージェントは，直接トレーサにより実行制御されている．ここではトレーサは，デバッグ対象エージェントの実行の一時停止・再実行を 1 命令 (本システムの場合 1 述語) ごとに繰り返す．一時停止をするタイミングは，1 命令が終了した直後である．実行再開をするタイミングは，ユーザによって与えることができる．

送信器は，(1a) トレーサによるデバッグ対象エージェントの実行速度低下を検出，(1b) 受信器に遅延情報を生成，そして (1c) 遅延情報を受信器に送信，の 3 つの処理を繰り返す．(1b) の遅延情報は，デバッグ対象エージェントがトレーサによって一時停止されていた時間 t_1 とユーザによって与えられた時間 t_2 のうちの小さい方である．t_2 を最大遅延時間と呼ぶ．これは，t_1 があまりにも長いと，非デバッグ対象エージェントの動作に支障が出てくる場合があるからである．例えば，エージェントが，ユーザと相互作用している場合に，エージェントが完全に停止することは不都合である．

受信器は，(2a) 受信器からの遅延情報を受信，(2b) 遅延情報に基づいてエージェントの一時停止・実行再開，の 2 つの処理を繰り返す．(2b) の実行制御では，基本的には一時停止と実行再開までの間隔は，遅延情報をそのまま利用する．タイムアウトを引き起こす可能性のある処理を行っている場合は，その処理の終了後に通常の処理を行う．受信器は，エージェントが，タイムアウト付きの処理待ちを行っている間タイムアウトを引き起こす可能性があると判断する．

図 5.2 では，送信器は，遅延信号 t，t'，そして t'' を送信した状態で．それぞれの受信器が，t を受信した状態である．このとき，t を受け取った受信器は，その受信器の制御対象のエージェントを停止させ，t 経過後に，実行再開させる．これにより，デバッグ対象エージェントで発生した実行速度の低下が，非デバッグ対象エージェントでも再現される．

本手法では，エージェントに受信器さえ装備しておけば良いので，実行速度が不確定な異種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にも適用可能である．さらに，通信において遅延時間があったとしても，受信する遅延信号の数は変化しないので，通信遅延を考慮する必要がない．また，実装において，大域時刻も必要としない．しかも，送信器と受信器は，少ないメモリで高速に処理できる．

本機構により，マルチエージェントシステム全体の実行速度比を保ちながらエージェントのデバッグを行うことが可能になる．

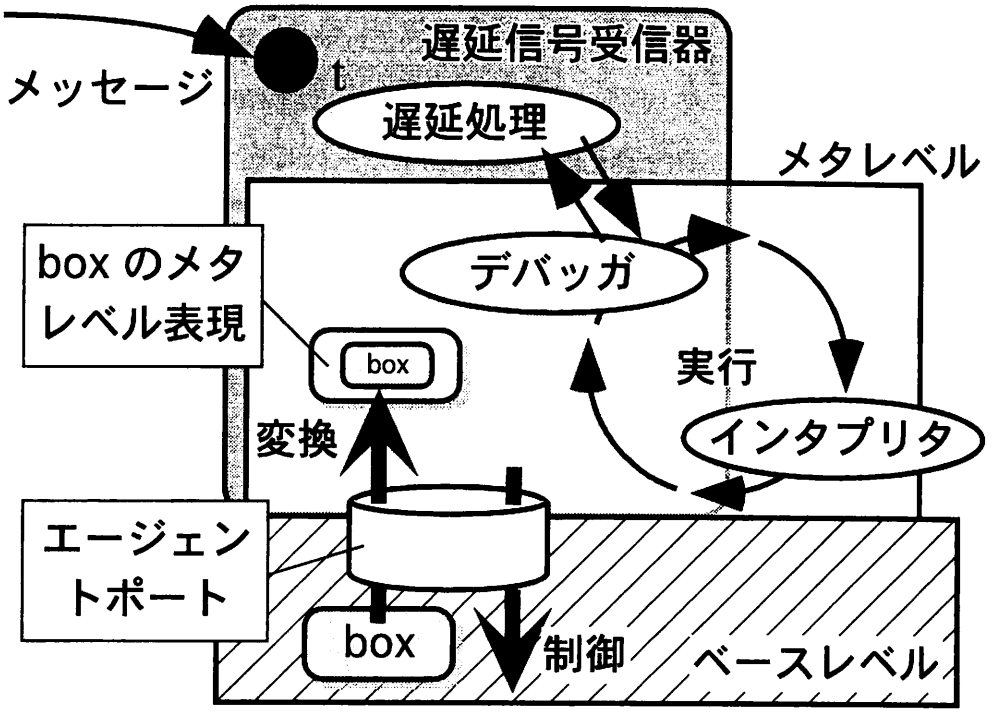

図 5.3: 遅延信号受信器の構成

5.2.2 遅延信号受信器の実装方式

図 5.3 は，受信器の構成図である．図 5.3 は，RXF のリフレクション機能 [6] を利用した，遅延信号受信器の実装を表している．受信器は，メタレベルにおいて動作しているデバッガに機能追加することで実現される．図 5.3 のメタレベルと書かれた四角が，図 5.3 のベースレベルと書かれた四角が表すベースレベルのメタレベルである．ここでベースレベルは RXF のプログラムの実行状態を意味し，メタレベルはそのプログラムのインタプリタを意味する．メタレベルでは，エージェントポートを利用してベースレベルでのプログラムの実行を観察・制御する．エージェントポートは，RXF でリフレクションを利用するために用いられる [6]．ベースレベルの状態の取得は，ベースレベルの情報をメタレベルで扱える表現形式（メタレベル表現）に変換するエージェントポートの機能によって実現される．ベースレベルの制御は，メタレベル表現に基づいてベースレベルの状態を変更するエージェントポートの機能によって実現される [6]．図 5.3 では，エージェントポートを用いて，ベースレベルにおける box を，box のメタレベル表現に変換している．メタレベルでは，インタプリタの実行とデバッガの実行を交互に行っている．受信器の機能は，デバッガの実行時に実行される．この機能拡張は，デバッガを実現する部分のプログラムを，デ

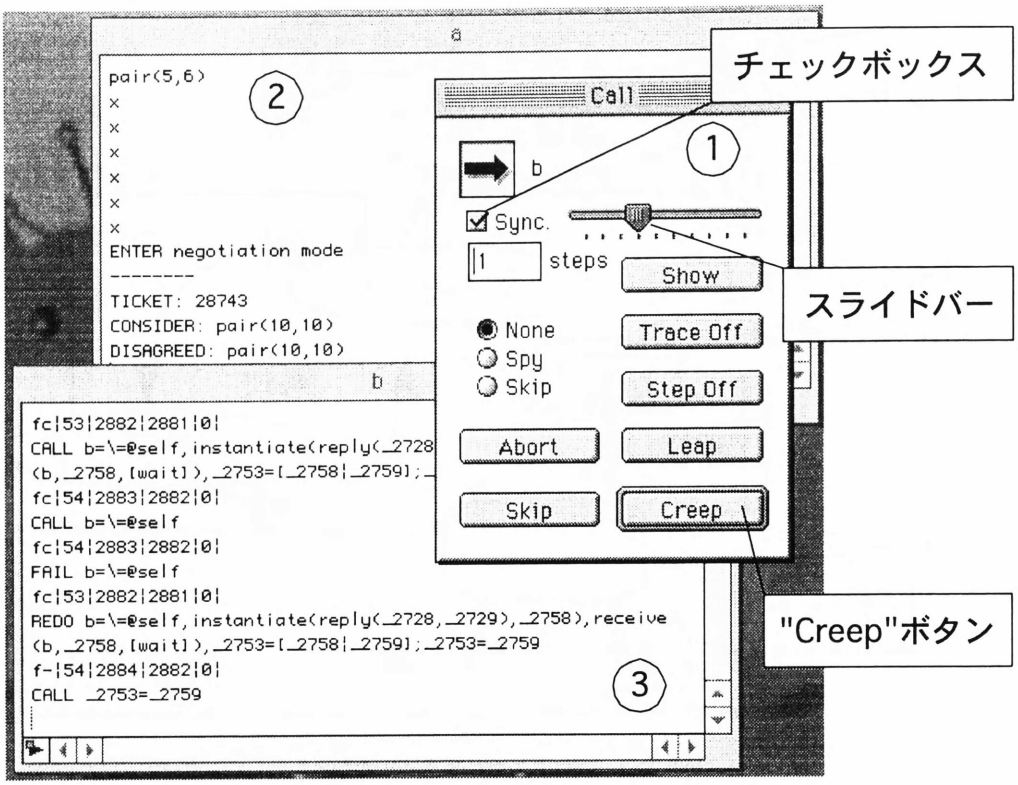

図 5.4: トレースダイアログ

バッガと受信器を実現するプログラムに変更することによって達成される．受信器を実現するプログラムは，エージェントポートを利用して，ベースレベルにおけるプログラムの実行を制御する．このようにリフレクション機能は，デバッガの機能拡張にも利用できる．

5.2.3 インタフェース

トレーサの制御は，専用のダイアログウィンドウで行う (トレースダイアログと呼ぶ)．図 5.4 には，エージェント “a” と “b” が存在し，“a” は非デバッグ対象で，“b” がデバッグ対象である状況を表している．トレースダイアログは，図 5.4 中の 1 とラベル付けされたウィンドウである．本トレースダイアログ上の “Creep” ボタンは，トレーサに一時停止されたデバッグ対象エージェントが実行を再開するタイミングを決定するためのボタンである．すなわち，ユーザが，“Creep” ボタンを押すたびに，デバッグ対象エージェントのステップ実行が進む．本トレースダイアログ上のチェックボックスは，本トレーサの速度比調整機能のスイッチである．本チェックボックスによって，本トレーサの速度比調整機能を有効にしたり無効にしたりすることができる．速度比調整機能が無効なときは，非デ

バッグ対象エージェントは実行制御されない．本トレースダイアログ上のスライドバーは，5.2.1 節で述べた最大遅延時間の設定に用いられる．本スライドバーを，左右にスライドすることによって，最大遅延時間を調整することができる．

2 および 3 とラベル付けされたウィンドウは，それぞれエージェント “a” 用のインタフェース (ウィンドウ 2)，および エージェント “b” 用のインタフェース (ウィンドウ 3) である．ウィンドウ 3 には，トレーサによって出力された “b” に関するデバッグ情報が表示されている．このように，トレーサによって得られたデバッグ情報は，デバッグ対象エージェントのもつウィンドウに出力される．

本トレースダイアログを用いることによって，本トレーサの速度比調整機能を制御することが可能になる．

5.2.4 評価

本実験の目的は，本トレーサにおける速度比調整機構が，デバッグ中のマルチエージェントシステムの実行速度比を一定に保つことができることを示すことである．本トレーサが，マルチエージェントシステムの実行速度比を一定に保つことを示すことができれば，トレーサによるデバッグ対象エージェントの速度遅延が回避されたと考えられる．すなわち，トレーサによるデバッグ対象エージェントの速度遅延に基づく “probe effect” が回避できると考える．

実験では，3 台の異なる実行速度を持つ CPU を持った計算機を用いる．それぞれの計算機は，実行速度の速いものから並べると，PowerMacintosh G3 DT233，PowerMacintosh 7300/180，そして PowerMacintosh 7300/166 である．すべての計算機は，10Mbps のイーサネットでネットワーク接続されており，互いに通信可能である．それぞれの計算機上でエージェント A，B，そして C が動作している．

本実験では，エージェント A，B，そして C に “ハノイの塔” を解くプログラムを実行させる．ここでは，A，B，そして C は，互いに実行の同期を取らない．ここでは 3 つの状況における計測を行った．それぞれ，(1) トレーサを使わずに実行，(2) 速度比調整機構を無効にして，トレーサを使いながら実行，そして (3) 速度比調整機構を有効にして，トレーサを使いながら実行，という状況で計測した．計測 (1) は，結果を評価するための基準を得るための計測である．計測 (1) の結果と計測 (3) の結果が類似しているのが，良い結果である．それぞれの計測では，300 秒間，各エージェントの実行速度を計測

	trace	ctrl	A	B	C
計測 (1)	off	off	1.00	0.603	0.554
計測 (2)	on	off	1.00	0.979	0.922
計測 (3)	on	on	1.00	0.603	0.553

表 5.1: 実行速度比の比較

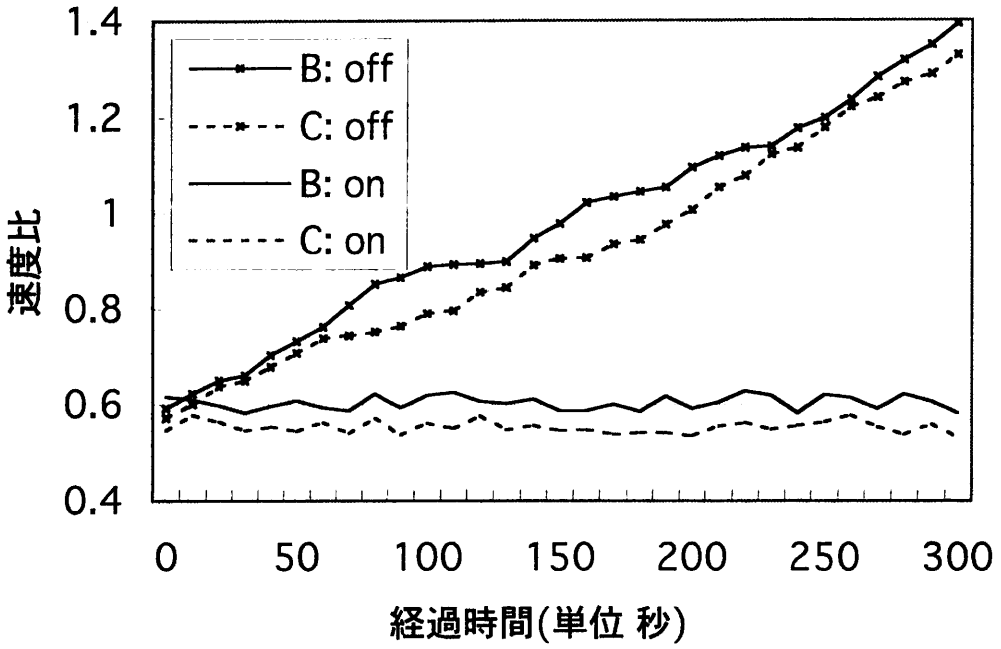

図 5.5: 実験の結果

した．本実験では，エージェント A の実行速度を 1 として結果を集計している．

それぞれの計測結果を表 5.1 にまとめた．表 5.1 の 2 列目（trace 列）及び 3 列目（ctrl 列）は，それぞれトレーサが有効（on）か無効（off）か，そして速度比調整機構が有効（on）か無効（off）かを表している．A，B，そして C 列は，それぞれの計測における，A の実行速度に対する比を表している．計測（1）により，トレーサ未使用時の，エージェント間の実行速度比は，“A : B : C = 1.00 : 0.603 : 0.554” （表 5.1 の “計測（1）” 行） であることがわかった．計測（2）により，トレーサを使用して，速度比調整機構を使用しないときの，速度比が，”A : B : C = 1.00 : 0.979 : 0.922” （表 5.1 の “計測（2）” 行） になることがわかった．A の実行速度が，遅くなっているのがわかる．計測（3）により，トレーサを使用して，速度比調整機構を使用したときの，速度比が，“A : B : C = 1.00 : 0.603

: 0.553” （表 5.1 の “計測（3）” 行） であることがわかった．計測（3）の結果は，計測（1）の結果に比べて，かなり類似していると言える．

図 5.5 は，計測（2） と 計測（3） における，各エージェントの実行速度比の変化をグラフ化したものである．図 5.5 の “B: off” 及び “C: off” は，それぞれ計測（2）におけるエージェント B 及び C の速度比を表している．同じように，図 5.5 の “B: on” 及び “C: on” は，それぞれ計測（3）におけるエージェント B 及び C の速度比を表している．計測（2）では，速度比が変化するが（A が遅くなっている）が，計測（3） では，速度比はそれほど変化しないことがわかる．

表 5.1 と 図 5.5 から，本研究で提案したトレーサを適用しても，それぞれのエージェントの実行速度比はそれほど変化しないことがわかる．すなわち，本速度比調整機構は，デバッグ中のマルチエージェントシステムの実行速度比を一定に保つことができる．速度比を一定に保つことによって，実行速度に関して，デバッグをしないときの状況に類似した状況で，デバッグを行うことが可能になる．これにより，“probe effect” の軽減が見込まれる．さらに，プログラマが，デバッグによる実行速度比の違いを意識する必要が無くなるという利点がある．

5.3 考察

プログラムの品質を向上させるための方法として，プログラム試験 [48] がある．プログラム試験では，プログラムの誤りを発見することによって，プログラムの品質の向上を目指す．並行プロセスのプログラム試験は，困難である [48]．同様に，並行プロセスの一種であるマルチエージェントシステムにおけるプログラム試験も，困難であると考えられる．マルチエージェントシステムは，動的にシステム構成を変更できるようなシステムの構成を目指しているので，マルチエージェントシステムにおけるプログラム試験は，非常に困難になることが予想される．なぜならば，動的にシステム構成を変更することによって，試験すべき状態数が爆発的に増加するからである．プログラム試験だけでは，マルチエージェントシステムのバグを十分に発見することは困難だと思われるので，マルチエージェントシステムの新たなデバッグ技術が必要となる．

並行プロセスにおけるデバッガとしては，アルゴリズミック・デバッギング [79] のように自動的にバグを発見できるような手法の方が好ましい．なぜならば，トレーサなどを用

いたデバッグをするためには，プログラマがデバッグ対象のプログラムに関する十分な知識を持っている必要があるからである．一般的な，トレーサなどの逐次プロセス用のデバッガは，並行プロセスの非決定性や大域時刻の欠如などの問題に対して本質的な解決策を示すことができない．一方，並行プロセスを静的に解析することによって，並行プロセスのデバッグにおける問題点を解決することも可能である．例えば，Krinke は，マルチスレッド化されたプログラムを静的に解析する手法を提案している [80]．しかし，一般的にこれらの手法は，高い計算量という問題点を持つ [49, 80]．すわなち，これらの手法を用いても，現実的には完全にバグを発見できるという保証はない．よって，本システムのようなデバッグ支援も有効であると考える．

並行プロセスへのインタラクティブなトレーサを考えた場合，複数の逐次プロセスを同時にトレースするよりも，並行プロセス中の単一の逐次プロセスをトレースした方が良い結果が得られる．これは，トレーサ利用時のユーザへの負担の大きさが理由として挙げられる．複数のトレーサをマルチウィンドウを用いて制御するのは，経験上，決して効率的であるとは言えない．なぜならば，複数のトレーサによって生成される大量のデバッグ情報を瞬時に理解するのが困難だからである．本アプローチでは，プログラマは，基本的に 1 つのエージェントのデバッグ情報だけを理解すればよい．よって，本アプローチは，本トレーサのような，インタラクティブなデバッグに適している．

5.4 成果

RXF におけるマルチエージェントの開発支援のための機能は，開発を支援する一方でデバッグを困難にする場合があり，従来の RXF ではデバッグ機能が不十分であった．本研究では，マルチエージェントシステムのデバッグにおける重要な問題である “probe effect” の解決を試みた．本論文では，マルチエージェントシステムのような並行プロセスのインタラクティブなデバッグに有効なトレーサの実装方式を提案した．トレーサなどの逐次プロセスのためのデバッグ技術に基づくデバッガは，本質的に並行プログラムのデバッグにおける問題を解決できないが，本手法を用いたトレーサは，並行プログラムのデバッグにおける問題の 1 つである “probe effect” を回避することが可能である．本研究では，デバッグ時に分散プロセスの速度比を一定に保つことによって “probe effect” の回避を目指した．また RXF のリフレクション機能を利用した，速度比調整機構に基づくトレーサの実装を示

した．RXF のリフレクション機能を用いることにより，RXF 自体を変更することなく本速度比調整機構を実装することができた．本速度比調整機構は，遅延信号に基づく手法を用いて速度比の調整を行う．本手法は，対象とするプログラムや計算機環境を前もって知る必要なしに適用できるという特徴がある．実験によって，本手法がデバッグ時のマルチエージェントシステムの速度比調整を可能にすることを示した．本システムにより，“probe effect” が回避される．実験結果により，本システムが，マルチエージェントシステム中の単一エージェントのデバッグに有効であることを示した．本アプローチは，box モデルに対して非依存であり，一般的な並行プロセス中の逐次プロセスのトレースにも適用可能である．逐次プロセス用のトレーサを，本アプローチに基づいて拡張することは，容易である．本アプローチは，並行プロセス中の単一の逐次プロセスのトレーサの実装に容易に適用でき，かつ有効な手法である．

第6章　エージェントのための推論機構

6.1 プロダクションシステム KORE/IE

RXF では，Prolog 上高速動作可能なプロダクションシステムである KORE/IE [81] をプロダクションシステムとして実装している．本章では，KORE/IE をマルチエージェントシステム開発に適用するための 2 つの拡張について述べる．一つ目の拡張は，例外処理への対応である．エージェントの反応性を実現するために有効である．もう一つの拡張は，ルールによる協調記述機能である．ルールによる協調プロトコルの記述を可能にする．

エージェントにおける割込処理は，エージェントの反応性の実現という点において重要である．例えば，文献 [82] では，例外処理の重要性について述べられており，エージェントの例外処理機構を動的に生成するシステムについて述べられており興味深い．

RXF における，ルールによる割込処理の実現は，RXF の割込処理機構であるアテンションハンドラと階層的評価器を応用している．

6.1.1 ルールによる協調記述

本システムでは，SaHow（作法）システム [83] によるルールによる協調プロトコルの記述機能を利用できる．SaHow システムでは，GUI を用いて協調プロトコルを設計することが可能である．設計した協調プロトコルは，KORE/IE のルールとして得られる．SaHow システムの開発において，従来の協調プロトコル記述系の以下のような問題点の解決を目指した．

- (i) メッセージの集まりごとに独立のメッセージの意味を定義するのでは，プロトコルを複数のメッセージの集まりからなるメッセージの列 (シーケンス) として定義するのが難しい．
- (ii) プロトコルを段階的に拡張しようとする場合多くのメッセージハンドラを変更する必要が出てきて，拡張が容易でない．
- (iii) メッセージの意味を応用領域に依存する部分としない部分に切り分けること，つまり，プロトコルの応用が難しい．
- (iv) 一つのエージェントが同時に複数の交渉などに関わる場合，これらの交渉などが独立ではなく，その間に何らかの競合があると，これらの競合の解決をプロトコル記述に含めるのが難しい．

- (v) プロトコルを動的に変更したり、新たに追加していくことを許可しながらエージェントを交渉させるのが難しい.

SaHow システムでは，これらの問題点を克服するために，ルールベースシステムに基づく協調プロトコルの表現を実現した．ルール表現を用いることで，プロトコルの状態遷移を表現でき，ルール集合を一組のプロトコルのシーケンスとして表現することができる (問題点 (i))．また，ルールの独立性によってメッセージの意味を与えるメッセージハンドラを別のルール集合で再利用することができる．これによって段階的な拡張が容易になる (問題点 (ii))．エージェントはプロトコルを表現したルールから呼ばれる関数を提供する．エージェントごとに応用領域に適した関数を用意することにより，応用領域に特化したエージェント独自な関数とプロトコルを切り分けることができる (問題点 (iii))．本研究で構築したシステムでは，エージェントは複数のルール集合を持つことができる．また，あるルール集合のルールの実行部から他のルール集合を呼び出して推論を行うことができる．これによって複数の推論を同時に行うことができる．ルール集合間で変数を共有することによって複数のプロトコル間の競合解決メカニズムを表現することができる (問題点 (iv))．本システムでは，エージェントがルールを動的にロードする機構を用意しており，プロトコルを動的に拡張することができる (問題点 (v))．

6.2 階層的事例ベース推論

本研究は，自律エージェントのための階層的事例ベース事例機構シェルの実装研究である [84, 85, 86, 87]．階層的事例ベース推論システムとは，事例ベース推論機構を制御するための推論システムの構築に事例ベース推論機構を階層的に用いた事例ベース推論機構である．事例ベース推論機構を制御するための事例ベース推論機構をメタ事例ベース推論機構という．本研究では，階層的事例ベース推論機構を，自律エージェントの環境への適応を実現するために導入する．これにより変化するユーザの主観に適応可能なエージェントの実装を目標とする．応用として動的に変化するユーザの好みに適応可能な献立作成システムを実装している．本章では，階層的事例ベース推論におけるメタ事例を用いたユーザの好みの表現について述べる．

事例ベース推論 [53]（Case-Based Reasoning，CBR と略す）とは，過去に解決した問題とその解決方法を事例として貯え，それらの事例を用いて問題解決を行う推論手法である．

解決した問題を，新たな事例として保存することで，学習をすることもできる．事例ベース推論の実現には，問題解決に適した事例を検索するための事例検索機構が必要である．事例検索機構の性能は，事例ベース推論の性能に大きく影響する．すなわち，事例検索機構の高機能化は，事例ベース推論全体の高機能化につながる可能性があり，事例検索機構の高機能化が望まれている．

6.2.1 階層的事例ベース推論とは

図 6.1 は，階層的事例ベース推論機構の模式図である．通常の事例ベース推論（図 6.1 の CBR）では，問題を解決するための事例を用いて問題を解決する．メタ事例ベース推論機構は，事例ベース推論機構を動作させることを問題とする事例ベース推論機構である．すなわち，事例ベース推論機構を動作させるための事例（メタ事例）を用いて事例ベース推論機構を動作させる．このように事例ベース推論機構を階層的に用いる事例ベース推論を階層的事例ベース推論と呼ぶ．事例ベース推論機構が，新しい事例を獲得することによって学習するのと同様に，メタ事例ベース推論機構も新しいメタ事例を獲得することによって学習する．

階層的事例ベース推論の実装において，事例ベース推論を制御するための推論システムとしての事例ベース推論，すなわちメタ事例ベース推論を用いた事例ベース推論方式で必要ある．階層的事例ベース推論は，事例ベース推論 の高機能化に貢献するとう報告 [56, 57] もある．メタ 事例ベース推論は，事例ベース推論 を用いた 事例ベース推論の構成部分（事例検索，事例適用，事例評価など）である．本研究では，メタ事例を用いた事例検索機構の実装を行っている．

メタ事例ベース推論は，事例ベース推論の実行過程をデータとして扱い，事例ベース推論の処理の一部を実現する．事例ベース推論の推論処理は，事例検索，事例適用，そして事例格納の 3 つから構成される．事例検索は，問題解決に適した事例を事例ベースから検索する処理である．事例ベースには，過去に解決した問題とその解決方法の対が事例として格納されている．事例適用は，事例検索によって得られた事例に基づいて，問題解決手法を生成する処理である．最後の事例格納は，事例適用で得られた問題解決手法を新たな事例として事例ベースに格納する処理である．事例検索処理では，現在の問題の解決に適した事例を検索する．ここでは，事例の問題解決への適正を，問題とその事例において解決された問題の類似性によって評価する．

メタ事例は，事例ベース推論における問題解決過程を表現し，問題解決過程や解の正当性を説明するために用いられる [86]．事例検索機構において，メタ事例は，事例検索過程や事例検索結果の正当性を説明するために用いられる．

図 6.2 は，階層的事例ベース推論を事例検索に応用した場合を図示したものである．図 6.2 の中央付近の太線の上部が事例検索を扱うメタ事例ベース推論機構（図 6.2 のメタ CBR）であり，下部が事例ベース推論機構である（図 6.2 の CBR）．まず始めに，図 6.2 の ① で，事例ベース推論システムに対して問題が与えられる．事例ベース推論システムは，問題が与えられると事例検索を開始する．ここで，処理がメタ事例ベース推論機構に切り替わる（図 6.2 の ②）．図 6.2 の ③ では，メタ事例ベース推論機構における事例ベース推論エンジン（図 6.2 の ③ の CBR エンジン）が，事例検索を開始する．メタ事例ベースエンジンは，通常の事例ベース推論における処理を行う．すなわち，メタ事例メモリからメタ事例を取り出し，事例検索の手順を解として得る．得られた検索手順は，事例検索実行部によって，実行され事例検索が達成される．事例検索実行部は，事例ベース推論システムに対するメタ機構となっており，事例ベース推論システムが管理する事例メモリ（図 6.2 の例では，2 つの事例メモリによって事例ベースが構築されている）に自由にアクセスできる．

6.2.2 基本的な定義

以降，問題空間 P, 解空間 S, システムの状態の集合 C, n 種類のオブジェクトの属性 $A_1, A_2, ..., A_n$, そしてオブジェクト x であるとする．ここでのオブジェクト x は，特徴空間上におけるベクトルとして表現される．すなわち $x =< a_1, a_2, ..., a_n >$ と表現される．このとき，$a_1, a_2, ..., a_n$ は，それぞれ属性 $A_1, A_2, ..., A_n$ に対する属性値であり実数値として表現される．これは，オブジェクト x に関する Nearest-Neighbor 法を用いた類似度比較時に用いられる．

6.2.3 メタ事例を用いたユーザの好みの表現

本システムでは，ユーザの好みへの適応を，事例検索における類似度計算を制御することによって実現する．本システムは，一般的な好みとユーザ特有の好みの 2 種類を用いてユーザの好みを推論する．一般的な好みとは，デフォルト値として用いられる好みである．ユーザ特有の好みとは，ある状況におけるユーザの好みである．一般的な好みは，料理知

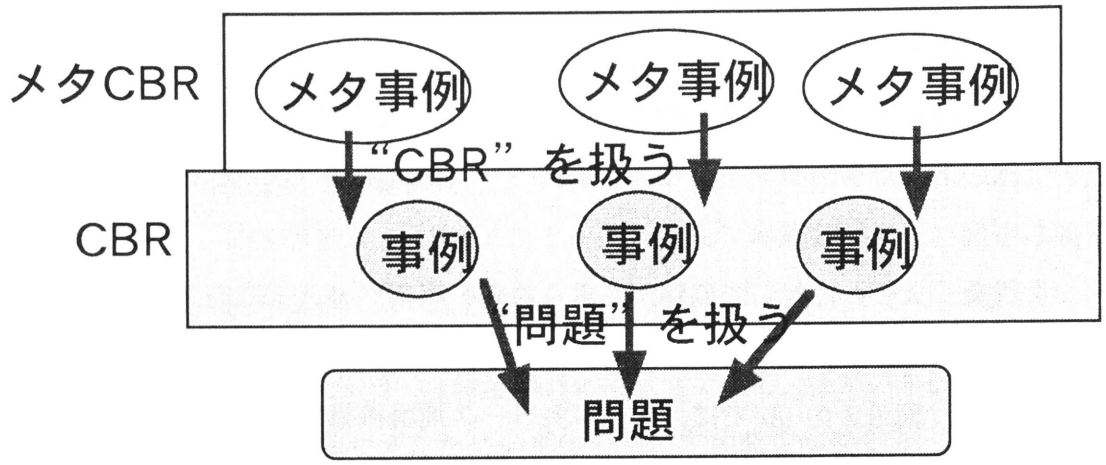

図 6.1: 階層的事例ベース推論機構

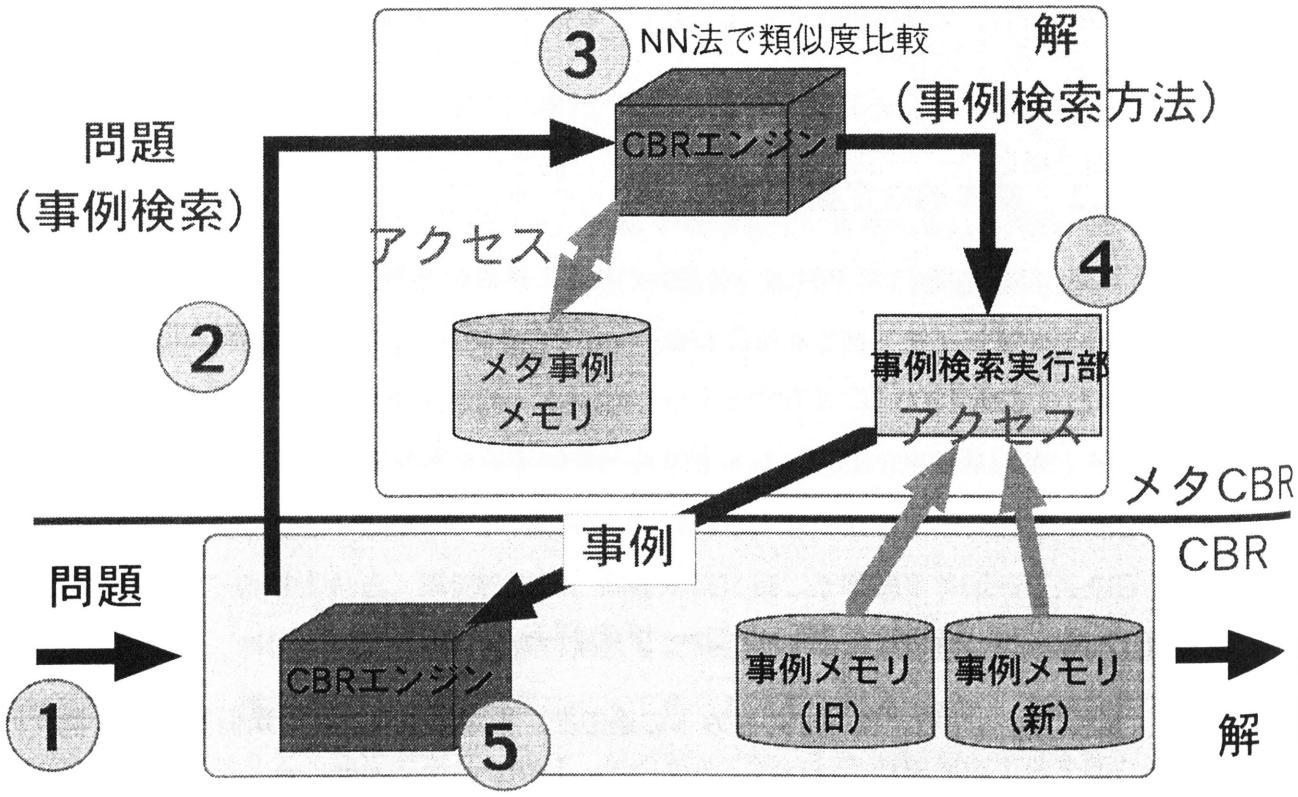

図 6.2: メタ事例を用いた事例検索機構

識ベース，好みルール，そして好み事例の3つの表現法を持つ．料理知識ベースは，料理に関する一般的な知識を格納している．好みルールは，好みを推論するための一般的なルールである．好み事例は，好みを推論するための事例である．好み事例に関しては後述する．ユーザ特有の好みの構成は，一般的な好みから料理に関する一般的な知識を除いたもの，すなわち好みルールと好み事例から構成される．一般的な好みは，すべてのユーザの好みの推論に利用されるが，ユーザ特有の好みは，ユーザ毎に構築される．

好み事例

好み事例には，属性値へのウェイトと2つのオブジェクト間の類似度の2つがある．属性値へのウェイトとは，Nearest-Neighbor法において用いる属性値に掛けるウェイトである．オブジェクト間の類似度とは，2つのオブジェクト間の類似度を直接数値で表現したものである．これらの事例は，両方とも，(1) 問題とシステムの状態，(2) 類似度と適用方針，(3) 事例の利用状況，の3つから構成される．(1) は，問題とその問題を解いた時のシステムの状態である．(2) は，得られた類似度と実行された事例適用操作である．(3) は，この事例が過去にどのように扱われたかである．(3) は，メタ事例の評価に利用される [86]．本メタ事例中の，c，t，そしてhはメタ事例特有であり，メタ事例ベース推論上で利用される．本システムでは，これらの事例を複合的に用いることによってユーザの好みを表現している．

ウェイトに基づく好み事例の定義は，$p \in P, c \in C, w_1, w_2, ..., w_n$がウェイト，$t$が問題解決時に用いられれた適用，そして$h$がこの事例の利用状況であるとき，以下のように定義される．

$$<< p, c >, << w_1, w_2, ..., w_n >, t >, h >$$

$w_1, w_2, ..., w_n$は，それぞれ属性$A_1, A_2, ..., A_n$に対するウェイトであり，$0 \leq w_1, w_2, ..., w_n \leq 1$の実数値で表現される．これは，特定の状況$< p, c >$においてウェイト$w_1, w_2, ..., w_n$を用いた類似度計算が有効だったことを表すメタ事例である．

x_1とx_2が類似していることを表わすオブジェクト間の類似度に基づく好み事例の定義は，$p \in P$でpはオブジェクトx_1を含む，$c \in C$，rがx_1とx_2の類似度，tが問題解決時に用いられれた適用，そしてhがこの事例の利用状況であるとき，以下のように定義される．

$$<< p, < x_1, x_2 >, c >, < r, t >, h >$$

これは，特定の状況$< p, c >$においてx_1とx_2が類似しているとみなすことが有効だったことを表すメタ事例である．

好み事例の例

実装は論理型言語上で行っているので，好み事例は述語として表現される．述語 case(1,w,[P,C],[[1,1,1],T],H) は，ウェイトに基づくユーザ好み事例の例である．第2引数の w は，この述語がウェイトに基づくユーザ好み事例であることを表している．述語 case(2,p,[P,C,[x1,x2]],[r,T],H) は，オブジェクト間の類似度に基づくユーザ好み事例の例である．第2引数の p は，この述語がオブジェクト間の類似度に基づくユーザ好み事例であることを表している．両事例とも，第1引数は事例のID番号である．また P は問題，C はシステム状態，T は適用，H は過去の利用情報を表わす．前者の [1,1,1] は，属性値へのウェイトのリストである．後者の r は，x1 と x2 の類似度である．

類似度計算

本システムでは，オブジェクト x_1 と x_2 の類似度関数 $sim(x_1, x_2)$ を用いて類似度計算を行う．$sim(x_1, x_2)$ は，以下のように定義される．

$$
\begin{aligned}
sim(x_1, x_2) &= sim_k(x_1, x_2) \\
&\quad + sim_r(x_1, x_2) + sim_c(x_1, x_2) \qquad (6.1) \\
sim_k(x_1, x_2) &: \text{料理知識ベースを用いた計算結果} \\
sim_r(x_1, x_2) &: \text{好みルールを用いた計算結果} \\
sim_c(x_1, x_2) &= sim_{c_w}(x_1, x_2) \\
&\quad + sim_{c_{pair}}(x_1, x_2) \\
sim_{c_w}(x_1, x_2) &= \sum_{i=1}^{n} w_i a_i \qquad (6.2) \\
sim_{c_{pair}}(x_1, x_2) &= r : x_1 \text{と} x_2 \text{の類似度} \qquad (6.3)
\end{aligned}
$$

式（6.1）が $sim(x_1, x_2)$ の定義である．本関数の特徴は，関数 $sim_c(x_1, x_2)$ である．関数 $sim_c(x_1, x_2)$ は，メタ事例を用いてオブジェクト x_1 と x_2 の類似度を計算する関数である．式（6.2）は，6.2.3節におけるウェイトに基づく好み事例に基づく類似度関数である．ここで $w_1, w_2, ..., w_n$ はウェイトに基づく好み事例を用いて計算される．式（6.3）は，6.2.3節におけるオブジェクト間の類似度に基づく好み事例に基づく類似度関数である．ここで r は，オブジェクト間の類似度に基づく好み事例を用いて計算される．

6.3 成果

エージェントの反応性を実現するために，ルールベースシステムに基づく割込処理機構を実装した．これにより，エージェントの反射的な動作を容易に記述可能になった．また，エージェントの協調動作の実装のためにルールベースシステムに基づく協調プロトコル記述系を実装した．これにより，エージェントの協調プロトコルをグラフィカルに設計できるようになった．

ユーザの好みに適応可能な献立作成エージェントの実装のために階層的事例ベース推論を導入した．ここでユーザの好みを表現するための2種類のメタ事例を用いた．ユーザの好みのような表現の困難な情報をメタ事例として表現し，推論に役立てることができた．このような事例検索機構は，自律エージェントの実現において，エージェントの環境によって動的に変化する可能性のある評価基準を扱う上で有効になる．

第7章　おわりに

RXF は，マルチエージェントシステム開発支援環境である．本開発環境は，エージェント記述言語，エージェントオペレーティングシステム，そしてエージェントフレームワーク，の 3 つから構成される．エージェント記述言語は，エージェントを記述するためのプログラム言語である．エージェントオペレーティングシステムは，エージェントの基本特性を実現するための基盤となるソフトウェアである．エージェントフレームワークは，エージェントを実装するためのライブラリであり，推論システムなどを含む．RXF は，実装言語として C++ を用いてコンパクトに実現されている．

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．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，手続き的リフレクションと，エージェントとメタエージェント間のリフレクション，パートに基づくリフレクションを利用している．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．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，問題解決をおこなうレベルや，問題解決のために実行するシミュレーションをおこなうレベル）に基づいた，プログラムの階層化をおこな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．

一般的にメタレベル実行には，メタレベル表現の生成のためのオーバヘッドが生じる．本研究では，エージェントポートを用いてメタレベル表現を動的に生成することにより，メタレベル表現の生成によって発生するオーバヘッドを抑えている．文献 [70] 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．また，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めざしている．

RXF のパートを用いることにより，レガシーアプリケーションとの連携を容易に行えるようになった．実際に，HyperCard を用いたインタラクティブな GUI の開発や，FileMaker を用いたデータベースの操作に応用している．

RXF のメモリ管理機構は，特定のサイズのセルに対する最適化を行うことによって，10倍の実行速度の改善と，メモリ使用量 1/4 を達成した．具体的には，12 バイトのセルに対しては，世代型複写式 GC を適用せずに，目印付け法を適用した．

Clause Mapped Part によって従来の RXF における様々な機能（階層化インタプリタ，階層化データベース，リフレクション）を統一された方法で記述することが可能になった．さらに Clause Mapped Part によって，エージェントの永続化が可能になり，モバイルエージェントの実装が可能になった．

RXF におけるマルチエージェントの開発支援のための機能は，開発を支援する一方でデバッグを困難にする場合があり，従来の RXF ではデバッグ機能が不十分であった．本研究では，マルチエージェントシステムのデバッグにおける重要な問題である “probe effect” の解決を試みた．本論文では，マルチエージェントシステムのような並行プロセスのインタラクティブなデバッグに有効なトレーサの実装方式を提案した．トレーサなどの逐次プロセスのためのデバッグ技術に基づくデバッガは，本質的に並行プログラムのデバッグにおける問題を解決できないが，本手法を用いたトレーサは，並行プログラムのデバッグにおける問題の 1 つである “probe effect” を回避することが可能である．本研究では，デバッグ時に分散プロセスの速度比を一定に保つことによって “probe effect” の回避を目指した．また RXF のリフレクション機能を利用した，速度比調整機構に基づくトレーサの実装を示した．RXF のリフレクション機能を用いることにより，RXF 自体を変更することなく本速度比調整機構を実装することができた．本速度比調整機構は，遅延信号に基づく手法を用いて速度比の調整を行う．本手法は，対象とするプログラムや計算機環境を前もって知る必要なしに適用できるという特徴がある．実験によって，本手法がデバッグ時のマルチエージェントシステムの速度比調整を可能にすることを示した．本システムにより，“probe effect” が回避される．実験結果により，本システムが，マルチエージェントシステム中の単一エージェントのデバッグに有効であることを示した．本アプローチは，box モデルに対して非依存であり，一般的な並行プロセス中の逐次プロセスのトレースにも適用可能である．逐次プロセス用のトレーサを，本アプローチに基づいて拡張することは，容易である．本アプローチは，並行プロセス中の単一の逐次プロセスのトレーサの実装に容易に適用でき，かつ有効な手法である．

RXF によって，制約論理型言語による命令レベルのまたは知識プログラミングが容易になり，さらにそれらのプログラムの並行動作やメッセージ通信による協調も容易である．

リフレクション機能を用いることで階層的なシステムの構築が容易に行える．エージェントオペレーティングシステムを，Clause Mapped Part を用いて操作することで，エージェントの性質の実現が容易に行える．エージェントフレームワークを用いることで，内部に階層構造を持つエージェントの実装が支援され，プロダクションシステムを用いることで割込処理や協調プロトコルの記述が可能になる．階層的事例ベース推論システムを用いることによって，動的な環境に適応可能な事例ベース推論システムの構築が可能になる．また RXF には，マルチエージェントシステムためのデバッグ機構もある．以上の RXF の機能を用いることによって，マルチエージェントシステムの構築が支援される．

7.1 今後の課題

RXF における，エージェントモデルやチームモデルなどの構築支援機能を強化する必要がある．形式的なモデルに基づいた，エージェントやマルチエージェントシステムの可視化や，知識レベルや協調レベルにおけるデバッグ支援に関する研究が必要である．これらのデバッグにおける研究は，単にデバッグのためだけではなく，未知の環境におけるエージェントの頑健性の向上にも役立つ．

マルチエージェントシステムのデバッグに関しては，モデルに基づく自動的な検証・試験・デバッグ機構の構築が今後の課題として挙げられる．今回は，あえて単一エージェントのデバッグを研究したが，今後はマルチエージェントシステムにおけるデバッグの問題に正面から取り組むつもりである．

本研究では，メタ事例を用いた事例検索について研究したが，メタ事例を用いた事例検索は今後の課題である．ここで重要になる点として，それぞれのメタ事例ベース推論システム同士の連携が挙げられる．これらを個々に性能向上させるだけでは，システム全体の性能の向上が望めないからである．よって，システム全体としての性能を改善させるためのメタ事例の利用が課題である．

謝辞

本研究を進めるにあたり，終始熱心な御指導，御鞭撻を賜わりました名古屋工業大学知能情報システム学科新谷虎松教授に心から御礼申し上げます．新谷教授には，学部生の頃から，修士，そして博士に至るまでさまざまな面で本当にお世話になりました．新谷教授の的確な御助言と適切な御指導，そして熱心な御討論をして頂きましたことは，研究を進めていく上でこれ以上無い程の励みになりました．さらに沢山の種類の Macintosh に触れる機会を与えて頂いたことは，学生生活において大きな収穫でした．もし私が新谷教授に指導される機会が無く Macintosh に触れることもなかったならば，現在の自分は無かったと思います．

また，本論文の作成にあたり，数々の御教示を賜りました名古屋工業大学知能情報システム学科石井直宏教授，北村正教授，そして伊藤英則教授に深く感謝致します．

さらに，発展的な御意見，積極的な御討論を頂きました新谷研究室の皆様をはじめ，多くの方々の御援助によって本研究を行うことができましたことを厚く御礼申し上げます．特に，RXF を使ってアプリケーションを作成して頂いた皆様，RXF の沢山のバグを発見して頂いた皆様に対しては，感謝の言葉もありません．心の中で感謝しながら，機能の追加やバグの修正をしていたことをここで告白します．

最後に，雨の日も風の日も夏も冬も終始温かい目で私を見守ってくれた家族に対して感謝の意を表したいと思います．

参考文献

[1] Jim Waldo. The JINI ARCHITECTURE for network-centric computating. *COMMUNICATIONS of the ACM*, Vol. 42, No. 7, pp. 76–82.

[2] 田原康生, 歌野孝法, 冲中秀夫, 丸山辰夫. Imt-2000 のサービスとシステム要求条件. 電子情報通信学会誌, Vol. 82, No. 2, pp. 108–115, 1999.

[3] 羽島光俊. 移動通信の変遷と展望. 電子情報通信学会誌, Vol. 82, No. 2, pp. 102–107, 1999.

[4] Donald A. Norman. *The Invisible Computer.* The MIT Press, 1998.

[5] 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.

[6] 大囿忠親, 新谷虎松. 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のその実装. 情報処理学会論文誌, Vol. 38, No. 7, pp. 1361–1369, 1997.

[7] Tadachika Ozono and Toramatsu Shintani. On a programming environment for building reflective agents. In *Proc. the IPSJ International Symposium on Information Systems and Technologies for Network Society*, pp. 303–309, 1997.

[8] Catherine Lassez. Constraint logic programming. *Computer Science*, 1987.

[9] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *14th ACM Symposium on Principles of Programming Languagesm*, pp. 111–119, 1987.

[10] F. Rich F. Tim and M. Don. A language and protocol to support intelligent agent interoperability. In *Proc. the CE & CALS Washington '92 Conference(1992)*, 1992.

[11] M.R. Genesereth and R. E. Fikes. *Knowledge Interchange Format Reference Menual.* Stanford University Logic Group, 1992.

[12] P. Maes. Issues in computational reflection, in meta-level architecture and reflection. pp. 21–35, 1988.

[13] B. C. Smith. Reflection and semantics in lisp. In *Proc. 11th ACM Symposium on Principle of Programming Languages*, pp. 23–35, 1984.

[14] Gerhard Weiss, editor. *Multiagent Systems A Modern Approach to Distributed Artificial Intelligence.* The MIT Press, 1999.

[15] 有馬淳. 複雑系としてのエージェント社会. 人工知能学会誌, Vol. 13, No. 1, pp. 5–6, 1998.

[16] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, Vol. 10, pp. 115–152, 1995.

[17] 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. ソフト・リサーチ・センター, 1995.

[18] Richard Murch and Tony Johnson. *Intelligent Software Agents.* Prentice Hall PTR, 1999.

[19] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, Vol. 60, No. 1, pp. 51–92, 1993.

[20] Y. Shoham. Agent0: A simple agent language and its interpreter. pp. 704–709, 1991.

[21] Anandeep Pannu Mike Williamson katia Sycara, Keith Decker and Dajun Zeng. Distributed intelligent agents. *IEEE Expert/Intelligent Systems & Their Applications*, Vol. 11, No. 6, pp. 36–46, 1996.

[22] Damian Chu. I.C.Prolog: a language for implementing multi-agent systems. In *the Special Interest Group on Cooperating Knowledge Based Systems*, 1993.

[23] Mihai Barbuceanu and Mark S. Fox. The architecture of an agent building shell. *Lecture Notes in Artificial Intelligence 1037, Intelligent Agents II*, pp. 235–250, 1996.

[24] Per Brand. *Enhancing the AKL compiler using global analysis.* Deliverable D.WP2.1.3.M2 in the ESPRIT project ParForce, 6707, 1994.

[25] Robbert van Renesse Dag Johansen, Fred B. Schneider, and Supporting Broad. Internet access to tacoma. In *the 7th ACM SIGOPS European Workshop*, pp. 55–58, 1996.

[26] Robbert van Renesse Dag Johansen and Fred B. Schneider. Operating system support for mobile agents. In *the 5th. IEEE Workshop on Hot Topics in Operating Systems*, 1995.

[27] C. Petrie. Agent-Based Engineering, the Web, and Intelligence. In *IEEE Expert*, December 1996.

[28] Stanford University CDR. *JATLite.* http://java.stanford.edu/.

[29] Markus Pischel Klaus Fischer, Jorg P. Muller. A pragmatic bdi architecture. *Lecture Notes in Artificial Intelligence 1037, Intelligent Agents II*, pp. 203–218, 1996.

[30] Itsuki Noda Hideyuki Nakashima and Kenich Handa. Organic programming lanugage gaea for multi-agents. In *ICMAS-96*, pp. 236–243, 1996.

[31] Itsuki Noda Hideyuki Nakashima and Kenich Handa. Organic programming lanugage gaea for multi-agents. In *ICMAS-96*, pp. 236–243, 1996.

[32] Jacques Ferber and Olivier Gutknecht. A meta-model for the analysis and design of organization in multi-agent systems. In *ICMAS98*, pp. 128–135, 1998.

[33] Deepika Chauhan and Albert D. Baker. Developing coherent multiagent systems using jafmas. In *ICMAS98*, pp. 407–408, 1998.

[34] Elizabeth A. Kendall. Agent roles and aspects. In *Aspect-Oriented Programming Workshop at ECOOP '98*, pp. 83–88, 1998.

[35] A. Mendhekar C. Maeda C. Lopes J. M. Loingtier G. Kiczales, J. Lamping and J. Irwin. *Aspect Oriented Programming.* Xerox Corporation, 1997.

[36] Phoebe Sengers. Designing comprehensible agents. In *IJCAI '99*, July 1999.

[37] 伊藤正美, 市川惇信, 須田信英. 自律分散宣言. オーム社, 1995.

[38] Catherine. Lasse. *Constraint Logic Programming.* Computer Science, 1987.

[39] T Chikayama. Esp - extended self-constrained prolog - as a preliminary kernel language of fifth generation computers. *New Generation Computing*, Vol. 1, No. 1, pp. 11 – 24, 1983.

[40] 茨木俊秀, 福島雅夫. 最適化の手法. 共立出版, 1993.

[41] hom Fruhwirth, et al. Constraint logic programming - an informal introduction. *ECRC technical report ECRC-93-5*, 1993.

[42] 渡辺卓雄. 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.

[43] C. S. Brian. Reflection and semantics in lisp. In *11th ACM Symposium on Principle of Programming Languages*, pp. 23–35, 1984.

[44] R. W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Aritificial Intelligence*, Vol. 13, pp. 133–170, 1980.

[45] Glenford J. Myers. *The Art of Software Testing.* Jhon Wiley & Sons, Inc., 1979.

[46] Jonathan B. Rosenberg. *How Debuggers Work.* John Wiley & Sons, Inc., 1997.

[47] Jack B. Rochester and John Gantz. *The Naked Computer.* William Morrow, New York, 1983.

[48] 古川善吾, 伊東栄典, 片山徹郎. 並行処理プログラムの試験. 情報処理学会論文誌, Vol. 39, No. 1, pp. 7–12, 1998.

[49] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, Vol. 21, No. 4, pp. 593–622, 1989.

[50] J. Gate. A debugger for concurrent programs. In *Proceedings of Workshop on Parallel and Distributed Debugging*, pp. 130–140. ACM, 1988.

[51] 亀田恒彦, 山下雅史. 分散アルゴリズム. 近代科学社, 1994.

[52] A. Newell. The knowledge level. *Artificial Intelligence*, Vol. 18, pp. 87–127, 1982.

[53] C. K. Riesbech and R. C. Schank, editors. *INSIDE CASE-BASED REASONING.* Lawrence Erlbaum Associates, Inc., Publishers, 1989.

[54] Mario Lenz, et al., editors. *Case-Based Reasoning Technology.* Springer, 1998.

[55] Friedrich Gebhardt, et al. *Reasoning with complex cases.* Kluwer Academic Publishers, 1997.

[56] Andrew Kinley David B. Leake and David Wilson. Learning to integrate multiple knowledge sources for case-based reasoning. In *IJCAI-97*, Vol. 1, pp. 246–251.

[57] Andrea Bonzano and Padraig Cunningham. Hierachical cbr for multiple aircraft coflict resolution in air traffic control. In *ECAI 98*, pp. 58–62, 1998.

[58] Pietro Torasso Luigi Portinale and Paolo Tavano. Dynamic case memory management. In *ECAI 98*, pp. 73–77, 1998.

[59] Ralph Bergmann and Wolfgang Wilke. Towards a new formal model of transformational adaptation in case-based reasoning. In *ECAI 98*, pp. 53–57, 1998.

[60] T. Silander P. Kontkanen, P. Myllym'aki and H. Tirri. On bayesian case matching. In *EWCBR-98, Lecture Notes in Artificial Intelligence 1488*, pp. 13–24. Springer-Verlag, 1998.

[61] Mercedes Gómez-Albarrán, et al. Modelling the cbr life cycle using description logics. In *ICCBR-99, LNAI 1650*, pp. 147–161, 1999.

[62] A. Mille B. Fuchs, J. Lieber and A. Napoli. Towards a unified theory of adaptation in case-based reasoning. In *ICCBR-99, Lecture Notes in Artificial Intelligence 1650*, pp. 104–117. Springer-Verlag, 1999.

[63] B. Fuchs and A. Mille. A knowledge-level task model of adaptation in case-based reasoning. In *ICCBR-99, Lecture Notes in Artificial Intelligence 1650*, pp. 118–131. Springer-Verlag, 1999.

[64] Ralph Bergmann and Armin Stahl. Reformulation in case-based reasoning. In *EWCBR-98, Lecture Notes in Artificial Intelligence 1488*, pp. 172–183. Springer-Verlag, 1998.

[65] Tadachika Ozono and Toramatsu Shintani. RXF: a java-based programming environment for building a multi-agent system. In *Proceedings of the 5th Pacific Rim International Conferences on Artificial Intelligence (PRICAI'98) Workshop on Java-based Intelligent Systems*, pp. 51–60, 1998.

[66] Tadachika Ozono and Toramatsu Shintani. Reflective agent programming environment rxf and its multi-agent tracer. In *IASTED International Conference, Robotics and Applications*, pp. 234–239, 1999.

[67] Apple Computer, Inc. *Inside Macintosh, Vol 6.* Addison Wesley, 1991.

[68] Y. Lesperance, et al. Foundations of a logical approach to agent programming. *Lecture Notes in Artificial Intelligence 1037, Intelligent Agents II*, pp. 331–346, 1996.

[69] T. Mullen and M. P. Wellman. Some issues in the design of market-oriented agents. *Lecture Notes in Artificial Intelligence 1037, Intelligent Agents II*, pp. 283–298, 1996.

[70] 田中二郎菅野博靖. メタ計算とリフレクション. 情報処理, Vol. 30, No. 6, pp. 694–705, 1989.

[71] 増原英彦, 松岡聰, 渡部卓雄. 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.

[72] Guy L. Steele. *Common Lisp the Language, 2nd Edition.* Digital Press, 1990.

[73] M. E. Bratman. *Intentions, Plans, and Practical Reason.* Harvard University Press, 1987.

[74] J. P. Muller. *The Design of Intelligent Agents.* Springer-Verlag, Berlin, 1996.

[75] 福田晃. 並列オペレーティングシステム. コロナ社, 1997.

[76] Richard Jones and Rafael Lins. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, 1997.

[77] W.F.Clocksin and C.S.Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[78] Jon E. Stromme. Integrated testing and debugging of concurrent software systems. *INDC96, http://www.kvatro.no/products/chipsy/pilot/paper-indc96/paper.html*, 1996.

[79] E Shapiro. *Algorithmic Program Debugging.* The MIT Press, 1983.

[80] Jens Krinke. Static slicing of threaded programs. In *PASTE '98*, pp. 35–42. ACM, 1998.

[81] 新谷虎松. Prologにおけるプロダクション照合フィルタの高速化. 情報処理学会論文誌, Vol. 32, No. 1, pp. 20–31, 1991.

[82] Mark Klein and Chrysanthos Dellarocas. Exception handling in agent systems. In *AGENTS'99. Proceedings of the third Annual Conference on Autonomous Agents*, pp. 62–68. ACM, 1999.

[83] 水口卓也, 水谷篤志, 大囿忠親, 新谷虎松. インタフェースエージェントにおける知識共有アーキテクチャについて. 第57回情報処理学会全国大会論文集. 情報処理学会, 1998.

[84] 大囿忠親, 新谷虎松. 事例ベース推論による事例検索機構の試作. 第56回 情報処理学会 全国大会, 第2巻, pp. 24–25, 1998.

[85] 大囿忠親, 新谷虎松. 事例ベース推論のためのメタ事例を用いた事例評価方式について. 第12回 人工知能学会全国大会論文集, pp. 152–153, 1998.

[86] 大囿忠親, 新谷虎松. アプリケーション独立なメタ事例の評価ルールについて. 第57回情報処理学会全国大会, 第2巻, pp. 339–340, 1998.

[87] 大囿忠親, 新谷虎松. 階層的事例ベース推論におけるメタ事例を用いたユーザの好みの表現について. 第58回情報処理学会全国大会, 第2巻, pp. 195–196, 1999.

論文リスト

査読有り

和文論文誌

- 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）
- 大囿忠親，新谷虎松：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のその実装，情報処理学会論文誌，Vol. 38，No. 7，pp. 1361-1369 (1997)

International Journal

- Tadachika Ozono, Toramatsu Shintani: On a Programming Environment for Building Reflective Agents, Information Systems and Technologies for Network Society, World Scientific, pp.303 - 309 (1997)

International Conference

- Tadachika Ozono, Toramatsu Shintani : RXF : a Java-based Programming Environment for Building Multi-Agent System, PRICAI'98 Java-Based Intelligent Systems Workshop, pp. 51 - 60 (1998)
- Tadachika Ozono, Toramatsu Shintani : Reflective Agent Programming Environment RXF and its Multi-Agent Tracer, IASTED International Conference, Robotics and Applications, pp. 234 - 239 (1999)

査読無し

- 大囿忠親, 柴田正弘, 新谷虎松 : プロダクションシステム KORE/IE の分散化とその応用について, 第 48 回 情報処理学会全国大会 講演論文集, Vol. 2, pp. 187-188(1994)
- 大囿忠親, 新谷虎松, リフレクティブな制約論理型言語 RXF を用いた事例ベース推論機構の試作について, 第 50 回 情報処理学会 全国大会 講演論文集 (2), pp.127-128(1995)
- 大囿忠親, 新谷虎松 : 事例ベース推論による事例検索機構の試作, 第 56 回 情報処理学会 全国大会 講演論文集 (2), pp. 24 - 25 (1998)
- 大囿忠親, 新谷虎松 : アプリケーション独立なメタ事例の評価ルールについて, 第 57 回 情報処理学会全国大会 講演論文集 (2), pp. 339 - 340 (1998)
- 大囿忠親, 新谷虎松 : 階層的事例ベース推論におけるメタ事例を用いたユーザの好みの表現について, 第 58 回 情報処理学会全国大会 講演論文集 (2), pp. 195 - 196 (1999)
- 大囿忠親, 新谷虎松 : アプリケーション間通信機構に基づくマルチエージェント制御機構の実現, 第 8 回 人工知能学会全国大会 講演論文集, pp. 295-298(1994)
- 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論文集, pp. 299-302(1995)
- 大囿忠親, 新谷虎松, 制約論理型言語 RXF におけるリフレクションについて, 第 10 回人工知能学会全国大会講演論文, pp. 123-126(1996)
- 大囿忠親, 新谷虎松 : 論理型言語 RXF におけるエージェント実装機能について, 第 11 回 人工知能学会全国大会, pp. 492-493 (1997)
- 大囿忠親, 新谷虎松 : 事例ベース推論のためのメタ事例を用いた事例評価方式について, 第 12 回 人工知能学会全国大会論文集, 人工知能学会, pp. 152-153 (1998)

- 大囿忠親，新谷虎松：自己反映型シナリオ管理機構を持つロールプレイングゲームの実装について，第6回 並列人工知能研究会 論文集（1995）
- 大囿忠親，新谷虎松：論理型言語 RXF におけるマルチエージェントシステム開発環境について，第28回 人工知能基礎論研究会，pp. 86-91 (1997)
- 大囿忠親，新谷虎松：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述語について，平成8年度 電気関係学会東海支部連合大会 講演論文集，p.280 (1996)
- 大囿忠親，新谷虎松：RXF におけるエージェントオペレーティングシステムについて，平成9年度 電気関係学会東海支部連合大会 講演論文集，p. 286 (1997)
- 大囿忠親，新谷虎松：エージェント開発環境 RXF における slicing に基づくデバッガについて，SWoPP 下関'99 プログラミング研究会, (1999).
- 副島大和，大囿忠親，新谷虎松：エージェント記述言語 RXF におけるエージェントプログラミング支援機構について”，第12回人工知能学会全国大会論文集，人工知能学会，pp.203-204，(1998).
- 水口卓也，永川成基，大囿忠親，川上義雄，新谷虎松：”ユーザーサポートエージェントの為の問題解決環境の実現，人工知能学会第10回全国大会論文集，pp.115-118，(1996).
- 大平峰子，大囿忠親，新谷虎松：階層的事例ベース推論を用いた料理デザインエージェントについて”，第58回情報処理学会全国大会論文集(2)，情報処理学会，pp.239-240，(1999).
- 副島大和，大囿忠親，新谷虎松：マルチエージェント開発環境 RXF におけるリフレクションを用いた分散トレー サについて”， 第58回情報処理学会全国大会論文集(1)，情報処理学会，pp385-386，1999.
- 水口卓也，水谷篤志，大囿忠親，新谷虎松：インタフェースエージェントにおける知識共有アーキテクチャについて”，第57回情報処理学会全国大会論文集，情報処理学会，(1998).

- 水谷篤志，水口卓也，大囿忠親，新谷虎松：インタフェースエージェント間の知識共有によるドキュメント管理システムの実現”，第 57 回情報処理学会全国大会論文集 (4)，情報処理学会，pp.217-218，(1998).
- 大平峰子，大囿忠親，新谷虎松：ユーザ意見を反映した事例による献立作成エージェントについて”，第 57 回情報処理学会全国大会論文集 (2)，情報処理学会，pp.337-338，(1998).
- 副島大和，大囿忠親，新谷虎松：リフレクションに基づくエージェントプログラミング支援機構について”，第 57 回情報処理学会全国大会論文集 (3)，情報処理学会，pp.545-546，(1998).
- 藤田隆久，大囿忠親，新谷虎松：ブラウジングモデルに基づく WWW ナビゲーションエージェント，第 56 回情報処理学会全国大会論文集 (3)，pp.179-180，(1998).
- 大平峰子，大囿忠親，新谷虎松：献立作成エージェントにおける献立の表現について，平成 10 年度 電気関係学会東海支部連合大会講演論文集，pp.289，(1998).