社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

大規模マルチエージェントシステムにおける並列処理法の考察

齋藤　健司† 　三浦　克宜† 　齋藤　一† 　前田　隆†

† 北海道情報大学
〒 069-8585 北海道江別市西野幌 59 番 2
E-mail: †{ksaito,miura,hajime,maeda}@do-johodai.ac.jp

あらまし　通常並列処理を実現する方法としてマルチスレッドプログラミングを使用することが多い。この手法のマルチエージェントシステムでの適応方法について考察する。いくつかの代表的なエージェントシステムでの実例と、我々の開発している ELM-VE システム [1], [2] で採用される手法を比較し検討する。

キーワード　マルチエージェント、スレッドプログラミング、並列処理、モバイルエージェント、強いモビリティ

Parallel Proceccing Methods in a Large Scale Multi Agent Systems

Kenji SAITO†, Katsunori MIURA†, Hajime SAITO†, and Takashi MAEDA†

† Hokkaidou Information University
Nishi-Nopporo 59-2, Ebetsu, Hokkaido JAPAN 066-8585
E-mail: †{ksaito,miura,hajime,maeda}@do-johodai.ac.jp

Abstract　Usually, multithread programming is used in many cases as a method of realizing parallel proceccing. We consider the adaptation method of this technique in the multi-agent system. We compare and examine methods adopted by some typical agent systems, and ELM-VE system which we are developing [1], [2].

Key words　Multi Agents, Thread Programing, Parallel Processing, Mobile Agents, Strong Migration

1. はじめに

現在、実用的なプログラムを作成するためには並列処理は必須の項目となってきている。通常は、この並列処理を行なうための手法としてスレッドプログラミングを使用することが多い。まったく意識していない場合でも、GUI を用いたプログラムの場合にはメインとなるプログラムのスレッド以外にもマウスやキーイベントをキャッチするためのスレッドが使用されている。通信を行なうプログラムでも、通信の要求を待ち受けるためのスレッドと実際の処理を行なうスレッドを分けるのが一般的である。

このように一般的に使用されるスレッドプログラミングであるが、このような処理で必要とされる並列性は小規模なものであり、数個から数十個のスレッドがあれば十分である場合が多い。しかし、近年研究される多くのアルゴリズムでは、もっと大規模な並列性を仮定したものが増えてきている。例えば「遺伝的アルゴリズム」や「ニューラルネットワーク」、「人工生命」などは多量のコンポーネントが相互に作用しあうことにより動作する。これらのアルゴリズムの研究の背後には、高度で複雑な処理を単一の機構で実行する従来の方法の限界を打開しようという試みがあり、エージェントと呼ぶ自律的なプロセスの集合体で問題を解決する手法と言うことができる。通常、上記のアルゴリズムの研究ではエージェント技術を用いていると明示されることは少ないが、ミンスキーなどが最初に提案したエージェントという概念に相当する技術を用いていると言うことができる [3]。

通常エージェント技術に焦点が当てられている場合には、ある程度高度な機能を持つエージェントが取り上げられ議論されることが多いが、この論文では特に低機能のエージェントを大量に使用する場合を想定している。遺伝的アルゴリズムやニューラルネットワークなどの理論的研究では、個体数やニューロンの数を数百程度とするものが多いが、より複雑なシステムを想定するならば、これ以上のエージェントを必要とする場合も考えられる。よってこの論文ではエージェントの数は少なくとも数百、多くは数万程度を想定している。

現在、これほど多量のエージェントを使用するシステムとしては上記の遺伝的アルゴリズムやニューラルネットワークの研究などが主なもので、それ以外の応用ソフトウェアでは一部を除いてあまり見られない。しかし我々は、現行の複雑なシステムの多くは多量のエージェントを使用するシステムに置き換える価値があると考える。これにより、システムの設計の面でも、機能の面でもまったく新しい可能性が開かれる。例えば、オペ

レーションシステムやゲームソフトウェア、仮想環境システムなどである。これらのシステムが数万ものエージェントを必要としなかったとしても、多数のエージェントをストレスなく動作させることや、効率的に管理する機能が必要となる。

このように多量のエージェントを作成する状況において、並列処理の最も一般的な実現方法であるスレッドをどのように使用すれば良いのかについて考える。

2. エージェントシステムにおける並列処理

現在、多数のエージェントシステムが提案されているが、それぞれが特徴ある並列処理の方法を採用している。多くのシステムがオペレーションシステムの提供するスレッドを利用する並列処理をそのまま採用しているが、まず、スレッドがどのように実装され、どのような特徴を持ちどのような問題点が考えられるかについてまとめてみる。さらに代表的なエジェントシステムについて、それぞれどのような手法で並列処理を実現しているかを見ていくことにする。

2.1 スレッドの実装と問題点

スレッドがどのように実現されているかについては、コンピュータのハードウェアとオペレーティングシステムに依存しており、細かな点に関してはシステムごとに異なっているが、ここでは共通する仕組みについて考える。基本的にスレッドは、ソフトウェア科学におけるタイムシェアリングシステム (TSS) の一つの形態でありごく基本的な概念であるが、通常意識しなくても使用することができるためにその実装がどのようであるかは忘れられがちである。

CPU(中央演算装置) はプログラムを実行するために (1) プログラムカウンタ、(2) レジスタ、(3) メモリそして (4) スタック領域を必要とする。通常のプログラミング言語ではメモリとスタックを変数を記憶領域として使用する。特にスタック領域は一時的に使用するローカル変数と関数呼出しに関する情報を記憶する場所となっている。プロセスという言葉があるが、これはコンピュータ上で独立して動作する一つのプログラムであり、プロセスは上記の 4 つの要素を独立して持っている。通常スレッドはこのプロセスの中に複数作成することができ、プロセスで使用するメモリを共有するが、他のプログラムカウンタ、レジスタ、スタック領域の部分を独立に持つ。これにより、並列処理を実行する。

ここで問題となるのがスタック領域である。スタック領域はプログラムが実際に実行される段階まで、どの程度の領域を確保しておくべきかが判断できない。スタック領域を使い尽してしまうと、プログラムはスタックオーバーフローなどのエラーをおこし終了してしまう。そこで、システムは通常使用する領域よりも多めの領域を確保するようになっている。Linux システムでスレッドを使用する場合で考えると、デフォルトで確保されるスタック領域の大きさは 2 メガバイトとなっている。そのために、単純計算で 100 個のスレッドを生成すれば 200 メガバイト、1000 個では 2 ギガバイトのメモリを必要とすることになる。この結果、現在の通常のパソコンで 1000 個以上の並列処理をスレッドを用いて実現するのは難しいと言える。よって、大量のエージェントを作成しなければならないシステムでは、単純にスレッドを用いてエージェントを実行する方法は実現困難であると言える。

また、コンピュータ間を自由に移動して処理を行なうモバイルエージェントの研究では、モビリティという性質が重要な項目としてあげられる。(マイグレーションと呼ばれることもある。) これは、実行中のエージェントを一旦停止して、他のコンピュータに移動してから再び実行を開始するための仕組で、弱いモビリティと強いモビリティと呼ばれる分類が存在する。弱いモビリティでは、プログラムコードとデータを移動させるが、実行状態を移動させないために移動後のエージェントプログラムの実行は常に同じ場所から始まる。これに対して強いモビリティでは、実行状態も移動させるために、移動後のエージェントプログラムの実行は移動前に実行していた場所から開始される。このような強いモビリティを実現するために多くのシステムでは、スレッドの仕組自身に改良を加えることが多いようである。

また別の問題であるが、スレッドを用いたプログラミングは高度なプログラミングスキルを必要とする。スレッドを生成するための手法、システムがデッドロックに陥らないための手法、複数のスレッドからアクセスされるデータの整合性を維持する手法についてマスターしなければならないことに加えて、プログラムのバグを見つけ出すための労力は、スレッドを使用しない場合に比べて格段に増えてしまう。

以上の点から、エージェントシステムにおけるスレッドの使用はには様々の側面がある。

2.2 現行のエージェントにおけるスレッドの使用法

現在、存在するエージェントシステムの多くは一万個ものエージェントを作成することを意識して作られているものは少ないと思われるが、スレッドの使用方法としては様々な工夫が見られる。ここでは、いくつかの特徴的なエージェントシステムについてスレッドの使用方法に注目しまとめておく。

2.2.1 Aglets

IBM で開発された著名なエージェントシステムであり、Java を使用したモバイルエージェントシステムとして早くから開発が始められ現在でもオープンソースのライセンスで公開されている。

仕様書の説明によるとエージェントに対して Java のスレッドをそのまま使用しているようである [4]。ただし、メッセージの送信に関する同期のためのメソッドが用意してあり、手軽に使用できるように工夫されている。

モビリティに関しては弱いモビリティをサポートしている。

2.2.2 Voyager

Voyager は ObjectSpace 社が開発し、現在では Recursion Software 社で開発されているモバイルエージェントシステムである [5]。エージェントの送信時には実行状態を転送しないので、弱いモビリティのサポートにとどまっているが、転送後にどのメソッドからスタートするかを指定することができるので、実行状態を転送しない欠点をある程度おぎなっていると考えられる。さまざまな通信仕様に対応している。

2.2.3 NOMADS

West Florida 大学の IHMC 研究ユニットで開発されている NOMADS は強いモビリティを実現するモバイルエージェントシステムとして開発されている [6]。このシステムは Java 言語で開発されているが、通常の Java 実行環境でなく独自に開発した Aroma と呼ばれる Java バーチャルマシンを使用している。この Java バーチャルマシンを使用することにより、実行中のスレッドを停止しスタック領域なども転送することで強いモビリティを実現している。

2.2.4 picoPlangent

以前は単に Plangent と呼ばれていたが、PDA や携帯電話での使用もサポートするようになり、picoPlanget と名称変更し現在も東芝で開発が続けられているモバイルエージェントシステムである [7]。picoPlanget の特徴は、プランニング技術を用いてゴール (目標) を達成するためのプランを Prolog 形式の導出法を用いて自動生成し、実行することである。

今回調べたモバイルエージェントシステムの中では、唯一スレッドをそのまま使用しないタイプのシステムだと思われ、他のシステムと比べて多くのエージェントを生成することが可能のようである。携帯電話などのデバイスを視野に入れているからではないかと思われる。

2.2.5 AgentSpace/MobileSpaces

国立情報学研究所の佐藤一郎氏の作成したモバイルエージェントシステムで、データとコードをまとめて転送することにより、エージェント移動を高速化していることが特徴の一つとなっている [8], [9]。また、MobileSpaces のほうでは、エージェントの中にエージェントを入れ子状に作成することが可能で、階層的なエージェントシステムを構成することができる。この機能は大規模なマルチエージェントシステムを作成する上で非常に有用な機能であると言うことができる。エージェントが自律的に動作し続けるためには、独自のスレッドを作る必要がある。残念ながら開発は中止されたようである。

2.2.6 MOBA

MOBA は早稲田大学の首藤一幸氏が開発しているモバイルエージェントシステムである [10]。JavaVM の拡張法の一つであるネイティブメソッドを使用してスタック領域も完全に移動させることができるようにしてあり、強いモビリティを実現している。スレッドは強いモビリティを実現するために拡張されているが、やはり一つのエージェントに対して一つのスレッドを使用するようである。

2.2.7 Swarm

もとは Santa Fe Institute で開発されていたものであるが、現在は Swarm Development Group という独立した組織でオープンソースとして開発されている [11]。これは他のシステムとは異なるタイプのエージェントシステムで、大量のエージェントを用いて様々なシミュレーションを行うことを目的としており、本論文の主旨に最も近いシステムである。主に Objective-C を使用するエージェントシステムだが、Java への対応も進んできているようである。スレッドは複数のエージェントで共有する形式をとっており、独自のスケジューラがスレッドを管理している。現在のバージョンではインストールがかなり複雑である。

3. ELM-VE におけるスレッドの使用

大規模なマルチエージェントシステムを作成するためには、現状のコンピュータの限界により、全てのエージェントにスレッドを割当てることは難しい状況である。我々の開発している ELM-VE システムでは、この問題を回避するために新しい処理方法を採用している。基本的に ELM-VE システムでは、一つのスレッドを複数のエージェントで共有する形式を取る。ELM-VE でのスレッド共有の方法を説明する前に、現行のシステムでもスレッドを共有して動作する例がある。最初にこのような例について考察する。

3.1 現行システムのおけるスレッド共有の方法

一つのスレッドで複数のオブジェクトやエージェントを駆動させる最も簡単な方法は、for ループや while ループを用いて、そのループの中でそれぞれのオブジェクトやエージェントのメソッドを呼出す方法である。これは、特別な方法ではなくデザインパターンの最も基本的な Iterator パターンに対応する方法となる。簡単なゲームプログラミングから、ニューラルネットワーク、遺伝的アルゴリズム、人工生命の研究においてシミュレーションプログラムを作成する場合においても、最も手頃で一般的だと思われる。事実、それほど複雑な処理を行わせない場合であればこれで十分な場合が多い。この方法を取れば、使用するスレッドは一つであり、スタック領域に大量のメモリを消費してしまうこともない。しかし、我々はこの方法には 2 つの問題点があると考える

一つは、すべてを 0 から作り上げなければならない点である。単純な処理の場合は問題ないが、少々複雑な処理を行う場合、例えば、プログラム実行中にエージェントの数が増減するような場合、全てのエージェントを動作させるのでなく必要なエージェントだけを動作させ、残りのエージェントを休眠状態にする場合、エージェント同士でメッセージを交換する方法を考える場合など、独自の仕様を定めて実装しなければならない。

もう一つは、感覚的な問題でもあるが、必ずエージェント全体を管理するプログラム (for ループや while ループを含むプログラム) が必要であり、本質的にはそのプログラムがエージェントを動かしているという形態を取るということである。自律的なエージェントを作成するためのモデルとしては、あまり適当ではないと思われる。単純なエージェントを作成する場合でも、システムを記述する時に必ず全体を管理する要素を加えて考えなければならなくなる。

スレッドを共有するもうもう少し高度な例としてイベントドリブンモデルがある。デザインパターンの用語では Observer パターンと呼ばれる。今現在、オブジェクト指向が普及したことにより、多数の独立したコンポーネントを集めて一つのソフトウェアとして機能するような実例がいくつか存在する。このようなソフトウェアでは、それぞれのコンポーネントは独自の機能を持っており適切なタイミングでその機能が実行されなければならない。このような例としてユーザインタフェースのプログラムが挙げられる、近年の GUI を用いたアプリケーショ

ンはメニューバー、ツールバー、ボタン、チェックボックス、テキストエリアなどのコンポーネントを持っているが、これらは部品として作成され時には大量に使用される。このようなコンポーネントの一つ一つにスレッドを使用すると大量のメモリ (スタック領域) を消費してしまうために、複数のコンポーネントが小数のスレッドを共有して使用するように設計されている。通常のプログラミングでは、このスレッドを意識せずにプログラムできるが、高度なプログラミングを行うためには、どのようにスレッドを共有しているかについて深い理解が必要で、時として非常に複雑である。3D のコンピュータグラフィックスで、複数のキャラクタをリアルタイムに動作させるような場合でも同様の問題がある。

このようなユーザインタフェースにおける、プログラムの設計ではイベントドリブンというモデルが採用されることが多い。イベントドリブンのモデルを用いたプログラムでは、各コンポーネントは「ある条件が成り立った時に自分を呼出して欲しい」ということを上位のプログラムに伝えておき、呼出されたら処理を行うという手法を取る。この仕組により、コンポーネントの動作は受動的なものとなる。厳密な意味では全てのプログラムは受動的なものであるが、エージェントの特徴である自律的、能動的な機能を実現するためには、他のプログラムから呼出されて駆動するという仕組以外に、自分を駆動させる仕組を自分自身の中に持っていることが必要と考える。さらにこれらの仕組は GUI などの特定の目的のためだけに設計されているもので、他の用途に使用することができないという問題もある。

問題点を整理すると、自分自身の機能を実行するための仕組を全て自分の中に持つことのできるスレッド共有のプログラミングモデルが無いということになる。ELM-VE では、使用領域を限定しないスレッド共有モデルを、できるだけ単純化して提案する。これにより、スレッドをそのまま使用する場合に比べて、格段に多量の自律的エージェントを作成することができるようになる。

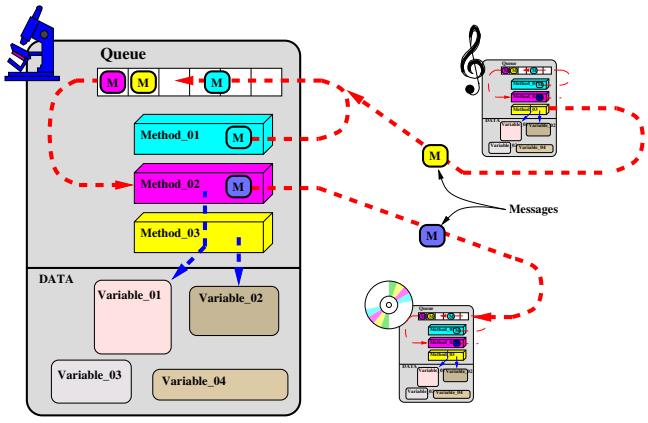

図 1　ELM-VE システムにおけるエージェント

3.2　ELM-VE で提案するスレッド共有方法

通常のエージェントシステムでは、エージェントどうしがメッセージを交換する目的は情報の交換にあるが、ELM-VE では情報交換のためだけでなく実行制御にもメッセージの交換を積極的に利用する。動作原理は非常に単純でメッセージが届いたら一旦待ち行列 (キュー) に保存し待ち行列の先頭からメッセージを処理してゆくだけである。

ELM-VE のエージェントの構成を図 1 に示す。全てのエージェントはメッセージの (1) 待ち行列、(2) メソッド、(3) 内部データを持っている。メッセージは他のエージェント、または自分自身が作り出し待ち行列に蓄積される。ELM-VE におけるメッセージは図 2 に示されるような内部構造を持つ。メッセージにはメッセージの (a) 送り手、(b) 受け手の情報の他に (c) メソッド名、(d) 引数が含まれる。

エージェントは待ち行列からメッセージを取り出し、メッセージに指定されたメソッドを呼出す。このメソッドの中でメッセージに含まれる引数を取り出し必要な処理を行い、場合によってはその処理の中でメッセージを作成して送信する。

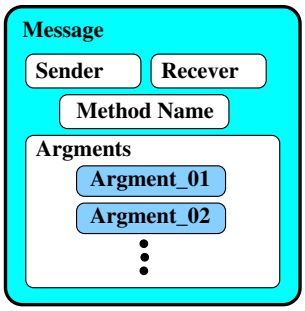

図 2　ELM-VE におけるメッセージの内容

上の説明では、エージェントが待ち行列からメッセージを取り出してメソッドを実行すると記述したが、正確には ELM-VE システムがそれを実行しており、厳密な意味では ELM-VE のエージェントも受動的に動作している。しかし、プログラマーにはその部分が完全に隠蔽されており実際エージェントがメッセージを取り出しメソッドを実行していると考えてプログラムを作成することができる。

ELM-VE のエージェントは、自分自身に対してメッセージを送ることができるが、この仕組を利用すれば、自分自身の機能を実行するための仕組を自分の中に持つことができる。

自律的な機能は、一つのスレッドを占有するエージェントシステムでは自然に実現される特徴であるが、そのようなシステムでも同時に複数の処理を並行して行うエージェントを作成するには、さらにスレッドを生成するか、一つのスレッドを分岐させるプログラムを作成しなければならず、繁雑なプログラムになってしまう。ELM-VE システムでは図 3 のようなコードを書くことで、それぞれの処理を独立して記述することができる。

モバイルエージェントの研究で重要な項目として取り上げられるモビリティに関しては、ELM-VE システムは弱いモビリティと強いモビリティの中間に位置すると考えることができる。ELM-VE システムでは、全てのメソッドの実行は短い時間で終了するように記述しておく必要がある。そして、エージェントの移動はメソッドの実行が終了したタイミングで行なわれるので実行状態 (スタック領域) を移動させる必要が無いのである。

そのかわり、メッセージの待ち行列に次に実行すべきメソッドが保存してあり、メッセージの待ち行列は通常のデータ転送で実現できるために、エージェントの移動後も引き続き処理を続けることが可能である。プログラマがエージェントの移動をほとんど意識せずにプログラミングすることができるという意味では、強いモビリティのほうに近いと言える。

```
import ac.hiu.j314.elmve.*;
public class TestElm extends Elm {
    protected void init() {
        send(makeOrder("loop1",null));
        send(makeOrder("loop2",null));
    }
    public void loop1(Order o) {
        /* 処理 1 */
        send(makeOrder("loop1",null));
    }
    public void loop2(Order o) {
        /* 処理 2 */
        send(makeOrder("loop2",null));
    }
}
```

図 3　ELM-VE のサンプルプログラム

本論文では、単純なエージェントを大量に作成するための手法について論じているが、メソッドさえ追加すれば ELM-VE システムのエージェントはどのようにも拡張できるので、単体で複雑な動作を行なうエージェントを作成する場合でも問題なく使用できる。

また図 3 のプログラムは、単独で実際に動作するプログラムとなっている。他のエージェントシステムと比べて、非常にシンプルにプログラムを作成できる。システムのインストールに関しても簡便で、すぐに導入することができる。並列処理で問題となる、デッドロックとデータ整合性の問題も起りにくい仕組になっている。メッセージ送信の同期を取る簡単な方法が提供されている。プログラミング初心者でも楽にエージェントプログラミングができると考えている。

3.3　その他の ELM-VE の特徴

ELM-VE の最大の特徴は、上記のスレッド共有モデルを採用して自律的なエージェントを大量に作成できる点である。その他 ELM-VE の特徴としては以下の点があげられる。

ELM-VE システムは仮想環境意識して作成されており、各エージェントは x,y,z の 3 次元座標を持っている。ELM-VE のクライアントを用いてサーバ内のエージェントを観察すると、3 次元空間に実際にエージェントが存在していることがわかる。ビジュアルプログラミングとは、ユーザが 2 次元、またはそれ以上の次元を用いてプログラムを表示することを可能にするシステム [12] であるが、このようなシステムへの応用も考えている。このようなシステムでプログラムする場合にも、大量のエージェントが必要となるため ELM-VE の特徴が生かされる。

ELM-VE ではエージェントの中にエージェントを生成する機能を持つ、エージェントも一つの空間を表すことができるので、図 4 のような階層的な仮想空間を作成することができる。単に階層的な仮想空間というだけでなく、複数の単純なエージェントをまとめて一つの高機能のエージェントに見せるなどの応用が可能である。

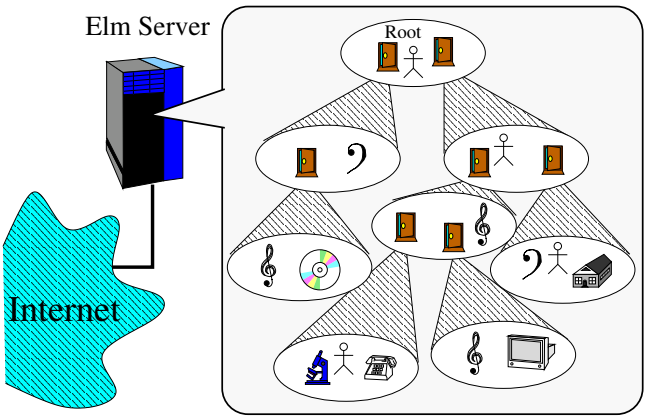

図 4　ELM-VE の仮想空間の構造

また、我々は仮想環境を用いて学習を行うためのシステムとして VESMA を開発している [13] が、このシステムは ELM-VE を用いて開発されている。

4.　他のシステムとの比較

他のエージェントシステムと ELM-VE の最も多きな違いは、スレッドを共有する新しい仕組を採用している点である。

大量のエージェントを作成できる。実際、単純なエージェントならば 1 ギガバイトのメモリを持つパーソナルコンピュータで 100 万以上のエージェントを作成することが可能である。

現行のエージェントシステムでは、強いモビリティを実現するためには JavaVM を新しく作成したり、ネイティブメソッドを使用して Java のポータビリティを失なうような改良が必要であったが、ELM-VE ではまったく Java 実行環境の拡張を行わずに強いモビリティに近い機能を実現している。

最も大きな欠点は、メッセージを処理するメソッドは短時間に処理を終了するように記述する必要があり、そのメソッドの中で無限ループを実行するような処理を許さないこと、そしてメッセージの同期を取るプログラムを書く場合にはメソッドを 2 つ以上に分割して記述する必要があるので、一連の処理が 2 つ以上のメソッドに分割されてしまいプログラムの可読性が落るという点である。この点は、ELM-VE のスレッド共有法の本質的な問題であるために、容易には解決できない問題である。根本的にこの問題を解決するためには、新しい言語を設計しその言語のコンパイラを開発することになるだろうと考えている。

また、ELM-VE の実行速度は他のシステムと比べて遅い。これはメッセージ作成や送信そしてスレッドの分配などの処理が重いためである。本格的なシミュレーションを行なう環境としてはあまり適していないが、プログラムミングが容易なので、試験的な実験には使用できると考えている。通常のエージェントの応用範囲であれば通常は問題無いと思われる。

広く使用されているシステムではないので、動作実績が無く

プログラムには不完全な部分が多い。ユーザインタフェースに関しても使いにくい部分が多々ある。

5. ま と め

大量のエージェントを作成する場面においてスレッドの使用方法に注目して、様々なシステムの考察を行った。スレッドをそのまま使用する現行の多くのエージェントシステムでは、パーソナルコンピュータで大量のエージェントを作成するのは困難であることを示した。多量の自律的なエージェントを生成するための手法としてスレッドを共有する新しい方法を提案した。新しい手法の利点・欠点を明かにした。

文 献

[1] Kenji Saito, Takashi Ohno,Hajime Saitoh and Takashi Maeda, *A Development Platform for an Educational Virtual Environment Supporting a Large Number of Autonomous Agents*, Proceeding of IS2000.

[2] *ELM-VE*, http://elm-ve.sf.net/, 齋藤健司.

[3] Marvin Minsky, *Society of Mind*, Simon & Schuster, Inc., 1985.

[4] *Aglets Specification 1.1 Draft*, http://www.trl.ibm.com/aglets/spec11-j.htm, IBM:Aglets, 1998.

[5] *Voyager Introduction, The Basic*, http://www.recursionsw.com/products/voyager/voyager.asp, Recursion Software, Inc., 2003.

[6] *An Overview of the NOMADS Mobile Agent System*, http://nomads.coginst.uwf.edu/, IHMC research unit.

[7] *picoPlangent* **の概要**, http://www2.toshiba.co.jp/plangent/, 東芝研究開発センター.

[8] *AgentSpace*, http://research.nii.ac.jp/ ichiro/agent/, 佐藤一郎.

[9] 佐藤 一郎, *AgentSpace:* **モーバイルエージェントシステム**, 日本ソフトウェア科学会 Workshop on Muti Agent and Cooperative Computation, (MACC'98), 1998.

[10] *MOBA*, http://www.shudo.net/moba/index.html, 首藤 一幸.

[11] *Swarm*, http://www.swarm.org/, Swarm Development Group.

[12] 岡村 寿幸, 田中 二郎, *3* **次元ビジュアルプログラミング環境における視覚化手法**, 日本ソフトウェア科学会第 19 回大会論文集, 2002.

[13] *VESMA*, http://vesma.sf.net/, 齋藤健司.