成蹊大学理工学研究報告
J. Fac. Sci. Tech., Seikei Univ.
Vol.44 No.1 (2007) pp.25-33

強マイグレーションモバイルエージェントを実現するコード変換手法

田久保　雅俊*[1]，櫻井　康樹*[1]，加藤　史彬*[1]，甲斐　宗徳*[2]

A Code Transformation Method Implementing Strong Migration Mobile Agent

Masatoshi TAKUBO*[1], Yasuki SAKURAI*[1], Fumiaki KATOH*[1], Munenori KAI*[2]

ABSTRACT：The authors have been developing a Java-based, strong migration mobile agent system called AgentSphere. It transforms strong migration agent source codes, written by users, into weak migration agent source codes automatically, and then executes the transformed agents on any Java Virtual Machines. The first version of AgentSphere has a defect of its performance, because it transformed source codes at each statement level. In this paper, a new source code transformation method is proposed and is integrated into AgentSphere. The new version of AgentSphere can transform strong migration source codes at each program control structure level including a migration function, and improves the performance of processing agents by reducing much overhead.

Keywords：Mobile Agent , Strong Migration , Source Code Transformation , Autonomic Distributed Processing System

(Received March 26, 2007)

１．はじめに

並列分散処理とは，ネットワーク上に接続された複数のマシンに処理を分けることで，並列的に処理を行い，さらに一台のマシンにかかる負荷を分散させることで処理の高速化や耐故障性の向上ができる手法である。

しかし，分散処理システムでは，変化し続けるシステム全体の状況把握や管理が必要となるため，様々な状況を考慮してプログラムを記述しなければならない。これには分散処理に対する高度な知識や経験が必要となる上，処理対象のプログラム記述に加えて，状況を制御するプログラムの記述をしなければならず，開発に時間がかかることが多い。

そこで我々は状況の判断・制御を自律的に行ってくれる自律分散処理システムを，モバイルエージェント技術を用いて試作してきた。

しかし，試作に用いてきたモバイルエージェントシステムは，多くの Java ベースのモバイルエージェントシステムと同様に，弱マイグレーションのモビリティを利用しているため，移動先で移動前の処理を継続するエージェントを自由に記述することは困難であった。そこで，利用してきたモバイルエージェントシステムのモビリティを強マイグレーション化して，移動前の処理を再開する記述が可能なエージェントの実現方法を提案する。

２．開発目的とする自律分散処理システムの概要

2.1　自律分散処理システムとモバイルエージェント

自律分散処理システムとは，分散処理で難しいと言われているシステムの状況把握やその制御をシステムに組み込み，システムやその周囲の状況の変化に応じて，システム自身が自律的に判断，対応できるようにしたシステムのことである。これにより，分散処理アプリケーションを記述したいプログラム記述者は，分散させたい計算処理のみを記述するだけで済み，開発にかかる時間を大幅に減らすことが可能となる。システムの自律性を満たすためには，システム内で動く各ソフトウェアが，他からのメッセージに応じた行動と，自ら取得した外部の情報に応じて判断を行い，その結果によって行動を計画し，その計画に基づき行動できる必要がある。そこで筆者らは，自律分散処理システムを実現するにあたって，モバイルエージェントを利用することにしてきた。

*1：工学研究科情報処理専攻修士学生
*2：工学研究科情報処理専攻教授（kai@st.seikei.ac.jp）

エージェントとは，人間の代理としてシステム上で自律的に行動するソフトウェアであり，モバイルエージェントとは，ネットワークを通じてマシン間を移動しながらタスク処理を行うことができるエージェントのことを言う。そして，モバイルエージェントシステムとは，モバイルエージェントの動作（生成，消去，移動，保存，複製，通信など）を提供するシステムのことである。既存のモバイルエージェントシステムとしては，AgentSpace（佐藤一郎氏）[1]や Aglets（日本 IBM 社）[2]，JavaGO（東京大学米澤研究室）[3]，MOBA（首藤一幸氏）[4]等が挙げられる。

2．2　AgentSpace を用いた自律分散処理システムの概要

本研究では，オープンソースであるためモバイルエージェントシステムの動作原理の解析やシステムの改造が容易で，エージェント移動時に情報を圧縮して送るため移動が速いという理由から，AgentSpace を用いて自律分散処理用のプラットフォームの開発を行ってきた。この自律分散用プラットフォームの構成の一例を図１に示す。

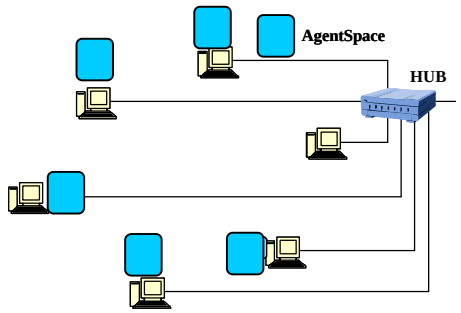


図１　システムの構成例

図１に示すように，システムはネットワークに接続された複数のマシンからなる。エージェントは AgentSpace 上にしか存在できないので，分散処理に参加するマシンでは AgentSpace を起動させておく必要がある。AgentSpace を起動すると，自律分散処理に参加する全マシンの性能値と IP アドレスが，マシンリストとして各 AgentSpace に保持される。

この自律分散処理用プラットフォームの目的は,「分散処理の手法について詳しくない人でも簡単に分散処理の恩恵を受けられるシステムを構築すること」と,「様々な種類のプログラムを実行可能な汎用性の高いシステムにすること」である。

一つ目の目的を充たすために，ユーザのサポートを行う様々なエージェントやシステムを作成した。これらのうち，まず Mediator というエージェントを起動することで分散処理を行うことができる。この Mediator は GUI を使って作られていて，実行するプログラムのクラスファイルやデータファイルを入力すると，分散処理を開始してくれる。この時，ネットワーク上に存在する各マシンの性能値に比例するように，処理量を分散するようになっているため，ユーザはどのマシンにどのくらいの処理を請け負わせるかを考えないですむ。また，処理が終わった部分解も取得し，ユーザ指定の解の形式で出力してくれるようになっている。さらに，システム内を巡回しながら，システム全体の動的な情報を取得・更新し，AgentSpace のダウンなどに対してエージェントの再生成などの対処を行ってくれる管理エージェント[6]や，Mediator の監視･サポートを行う MediatorSupporter[7]エージェントなど，不慮の事故に対処するエージェントを導入することで，ユーザにシステムの耐故障性の面で気を使う必要をなくしている。この他にも，システムの状態を把握しやすくするための分散管理モニタ[8]や，エージェント間のデータのやりとりをサポートする通信用エージェント[9]なども作成し，できるだけユーザは分散処理対象の計算処理部のプログラム記述だけに集中できるように努めている。

二つ目の目的である「様々な種類のプログラムを実行可能な汎用性の高いシステム」を充たすために，SPMD 形式のプログラムと，MPMD(Multiple Program Multiple Data)形式のプログラムのどちらでも自由に記述できるようにした[5]。現在，SPMD 形式のプログラムの実行開始用のエージェントとして Mediator がある。MPMD 形式のプログラムは，エージェント間の通信を含めてユーザがすべてを詳細に記述する必要がある。そのため，各エージェント間の通信関係を簡単に把握しながら記述ができる本研究用の分散処理エディタを作成しており，これを用いることによって SPMD，MPMD 形式どちらのプログラムも記述が比較的簡単にできるよう支援している。

2．3　強マイグレーションの必要性

ネットワークに繋がれたコンピュータの性能は全て同じではなく，各コンピュータは分散処理以外の処理も行っているため発揮できる性能は常に一定ではない。そこで本システムでは，分散処理開始時直近における各コンピュータの性能値に合わせて自律的に各コンピュータに割り当てる仕事量を調整する機能を構築している。

分散処理を開始するときに Mediator によって生成されるタスクエージェントは，SPMD 形式のプログラムと

して記述されているため同じ仕事量を持っていて，このタスクエージェントの数が各コンピュータに割り当てられる仕事量に相当する。タスクエージェントを生成したMediator は性能値の高いコンピュータにより多くのタスクエージェントを割り当てていく。

ただしこの割り当ては，分散処理開始時の各コンピュータの性能によるものであり，一度分散されたタスクエージェントがその後の状況に応じて自由に移動できるわけではない。例えば，分散処理中のコンピュータに分散処理とは別の要因で非常に大きな負荷がかかり，タスクエージェントの処理の完了が大幅に延期されるというような事態においてもエージェントはただ待つしかない。そこで，そのような場合にシステムが自動的に性能の良いコンピュータへタスクエージェントを移動させ，大幅な処理の遅延を防ぐ機能が必要となる。

この機能実現には，エージェントが任意の時点で処理を中断して実行途中のデータを保持し，移動先のコンピュータ上で中断時点からの処理の再開ができることが必要となる。

しかし，現在利用しているモバイルエージェントシステム AgentSpace のモビリティは弱マイグレーションであるため移動時に持っていく実行時データでは，中断した時点からの再開を行うには不十分である。そこで，本研究では，中断時点での再開が可能な実行時データを持って移動する強マイグレーション方式のモバイルエージェントシステムを実装することにした。

３．エージェントの強マイグレーション化

3．1　強マイグレーション化に必要な情報

Java においてプログラムが使用するメモリ領域には，実行コード領域，スタック領域，ヒープ領域がある。

Java を用いてモバイルエージェントを実装する際に，予め Java に備わっているシリアライズ機能を用いることによって，実行コードに加えてヒープ領域内の情報の保存が可能となっている。しかし，スタック領域内の情報やプログラムカウンタを実行状態として保存する機能は現状の JavaVM ではサポートされていない。

プログラムカウンタを取得・復元できれば移動直前の位置からのプログラムの再開が可能となる。また，スタック領域内に保持されているローカル変数の値を取得・復元できれば，移動前の計算結果を無駄にせず，実行を再開することができるようになる。そのため強マイグレーション方式のモバイルエージェントシステムを実現するためには，スタック領域内の情報とプログラムカウンタに相当する情報が必要となる。

しかし，これを Java で実装するのは容易ではない。これを満たすために, Java を用いた既存の強マイグレーション方式のモバイルエージェントの MOBA では JavaVM の変更やネイティブメソッドの追加によってローカル変数やプログラムカウンタを取得している。JavaGO ではソースコードを変換することで実行状態の保存と復旧の機能を付加している。しかし，MOBA の JavaVM を変更する手法の場合，JavaVM のバージョンアップごとに修正を行わなければならず，システム開発の負担が大きい。また，JavaGO では，エージェントの移動を実現するためにランタイムシステムを拡張している。そのため，拡張されたバイトコードインタプリタや特定の JIT コンパイラ上でしか動かないものになっていて移植性が失われるという欠点があった。

3．2　強マイグレーション化のための手法

本研究では，移植性や JVM のバージョンアップに対するシステム開発者の負担を減らすために，JavaVM に手を加えることなく強マイグレーション方式のモバイルエージェントシステムを実現することを目指している。しかも，記述上は，プログラム中に単に migrate(移動先ホスト名);のように記述すれば，マイグレーションが可能で，この続きから実行できるようにしたい。

そのための手法として，スタック領域内のローカル変数の取得・復元には後述する JPDA を用いた手法を，プログラムカウンタ相当の情報の取得・復元にはソースコード変換手法を用いて実現した。以下に，それぞれの手法について説明する。

3．3　スタック領域の取得・復元

スタック領域の中身を取得するために，JPDA(Java Platform Debugger Architecture)を使用した。これは，サン・マイクロシステムズ社が提供する JavaVM に標準的に実装されており，Java アプリケーションのデバッグに使用されるもので，実行中のスレッドや実行情報へのアクセスが可能となっている[10]。本研究では，この機能を用いることによって，スタック領域のデータを取得することにする。

図２は，ローカル変数を取得し，持ち出し可能にするまでのプロセスの概要を示したものである。

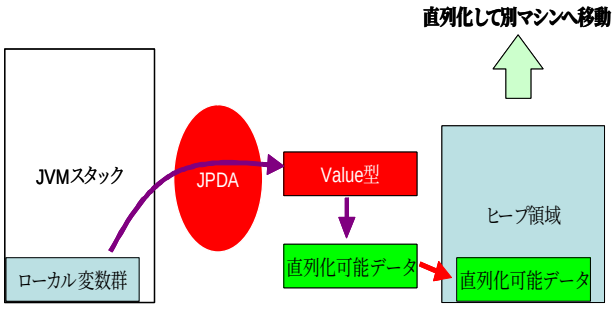


図２　ローカル変数持ち出しまでのプロセス

エージェントは，スレッド上で実行される。そこで，JPDA を用いてエージェントが実行されているスレッドにアクセスし，スタック内のローカル変数の値を取得する。しかし，このローカル変数の値は JPDA 特有のクラス(Value 型)となっているため，外部への持ち出しが許可されていない。そこで，この値をシリアライズ（直列化）可能な変数に変換し，ヒープ領域内に格納する。そうすることで，JVM を変更することなく，Java に備わっているシリアライズ機能のみを用いて実行コード領域・ヒープ領域そしてスタック領域内のローカル変数を保存して別マシンに送信することができるようになる。

ローカル変数の復元は，移動前のローカル変数の値を対応するローカル変数に代入することで実現する。この代入文は後述のソースコード変換時に挿入する。

図３に示すのは，実行再開位置にソースコード変換時に挿入するローカル変数復元代入文の一例である。

```
create(){
  int x ;
  double d;
        :
        :
  migrate();
  //実行再開位置
  x = ac.getIntegerValue(“x”);
  d = ac.getDoubleValue(“d”);
        :
        :
}
```

図３　ローカル変数復元代入文

JPDA にはスタックフレームにアクセスし，そのスタックフレームに直接値をセットする別の機能がサポートされている。しかし，その方法では，スタックフレームにアクセスするために対象となるスレッドを一定時間停止させておかなければならず，代入以上に時間がかかる。そのオーバヘッドをかけないために，ここでは移動前のローカル変数の値を代入することでローカル変数の復元を実現している。

3．4　ソースコード変換

実行状態を完全な形で保存するためには，スタックの中身だけでなく，プログラムカウンタも必要となってくる。JPDA では，このプログラムカウンタに類似したデータを扱っており，これを取得することは可能となっている。しかし，この取得したプログラムカウンタを用いてプログラムを途中から実行する機能は，JPDA でもサポートされていない。そこで，本研究ではソースコードの変換を行うことにより，同等の機能をサポートすることにした。

従来のソースコード変換は，処理の一行一行をステートメント関数に分けたステートメントコードと，そのステートメントフローを制御するランタイムコードの二つにユーザコードを変換する方式だった。この方式では，ステートメントを一行一行関数に区切る再構成を行い，どの関数を移動後に実行すれば良いかを制御することで，移動直前の処理から再開することが可能となっていた。

しかし，この方式では移動直前のスタックフレームの構造を復元していないためユーザ関数からのマイグレートや，繰り返し回数の判明していない繰り返し文中のマイグレートなど，変換できない制御構造があった。また，ステートメント関数中に実行処理文は１行しか記述されていないが，そこで使われている変数を宣言し，移動直前の値へと復元するために何行ものローカル変数復元代入文を挿入する必要があった。このため，一行の処理を行うのにオーバヘッドが大きくなってしまうという問題点があった。

本論文では，これらの問題点を解決するために新たなソースコード変換手法を提案する。

3．4．1　ソースコード変換手法の概要

新しいソースコード変換手法では，プログラムコード中に記述された移動用関数である migrate 関数の位置で実行時データを取得して移動し，到着後に移動直前の状態に復元するように変換する。これにより，ステートメント一行一行に対して，ローカル変数の取得・復元処理を行わないですむので，オーバヘッドを減らすことができる。

この新しい変換手法を実現するためにまず，エージェントが持って移動する情報の中に，マイグレーション直後かどうかを示すフラグ MigFlag を用意した。このフラグが true のときには移動直後であることを示し，初期値は false である。

この MigFlag を使い，移動直後には実行再開位置までの計算処理を全てスキップするようにソースコードを変換する。こうすることで，移動前のプログラムカウンタ

を保存して続きが実行できるのと同じ状態を実現することにした。

変換は移動関数である migrate()が存在するスコープを中心に考える。まず，変換の前準備として，そのスコープ内で定義している宣言文をスコープの前半にまとめるように変換する。

こうすることでできた「宣言部・処理群・migrate 関数・処理群」という流れが変換対象の基本形となる。

図４は，この基本形をどのように変換するか，という変換の基本方式を示したものである。

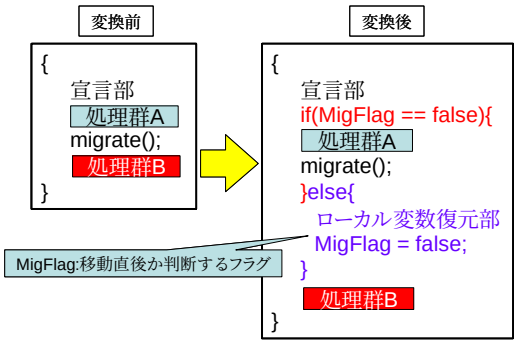


図４　変換の基本方式

具体的にどのように基本形を変換したかを以下に示す。

① migrate 関数と，それ以前の処理群（処理群 A）を内包した条件文 if(MigFlag == false)を挿入

② ①の if に対する else 文を作り，そこに宣言部の復元を行う代入文と MigFlag のリセット処理を挿入する

変換前に，宣言文をまとめていたのは，宣言文と計算処理文が混在していると，①の条件判断回数が増えてしまい，実行再開までのオーバヘッドが増加する可能性があるからである。

このようにして変換が行われたソースコードが実際にどのように動き，いかにして途中実行が実現されるかを説明する。

まず，エージェントは生成時か到着時かを問わず常に同じ決まった関数から実行が開始される。そして，処理に必要なローカル変数の宣言が行われる。次の，MigFlag を調べる条件文では，MigFlag の初期値は false であるため処理群 A が実行される。そして migrate 関数で別マシンへと移動する。この時，migrate 関数の内部処理で MigFlag が true に切り替わる。

エージェントが目的のマシンへ到着すると，プログラムの先頭から実行が開始される。そして，ローカル変数の宣言が行われる。そしてフラグの判定では，今度は MigFlag が true のため else 文へ入る。この else 文の中で，ローカル変数に移動前に取得しておいたローカル変数の値を代入し，MigFlag を false に戻す。

これにより，ローカル変数の復元が行え，migrate 関数直後の位置から実際の計算処理を再開することができるようになる。

3．4．2　ソースコード変換手法の詳細

ソースコード変換の基本方式・実行再開までの流れを説明したが，この変換・実行再開方式を，どのような状況，どのような制御構造をとるプログラムにおいても利用可能にするためにいくつかの変換仕様を決めた。以下にその仕様について説明する。

(1) 複数のスコープが関係する場合

前節で説明した変換概念は，単一のスコープ中での変換であったが，これが多重スコープになっても変換の基本は変わらない。図５は多重スコープが存在するソースコードの変換の流れを示したものである。

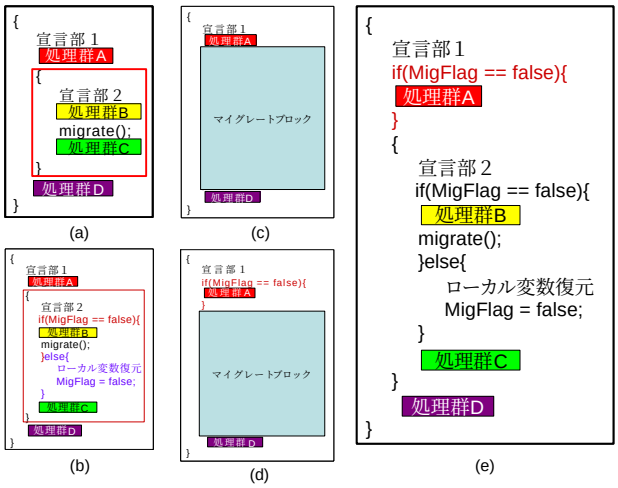


図５　多重スコープが関係するコードの変換

(a)が変換対象となるコードの初期状態で，多重スコープとなっている。まず，一番内側の，migrate 関数が存在するスコープに着目する。すると，そのスコープは基本形となっている。これを基本方式で変換すると(b)のようになる。

ここで，変換したスコープをマイグレートブロック呼ぶことにする。すると，(c)のように，外側のスコープは「宣言部・処理群・マイグレートブロック・処理群」と，基本形と似た形となっている。これも基本方式に則って，(d)のように変換することができる。ただし，マイグレートブロックの中でローカル変数の復元処理を行ってしまったので，外側のスコープでは復元処理を必要としない。

そのため，基本方式の①の手順のみを行えば良い。すると(e)のように変換される。これにより，何重にスコープがある構造であろうと，変換することができる。

図６のように，migrate 関数が関係していないスコープに対しては，そのスコープそのものを処理群の一部としてまとめることができるので，スコープごとスキップするように変換を行うことができる。

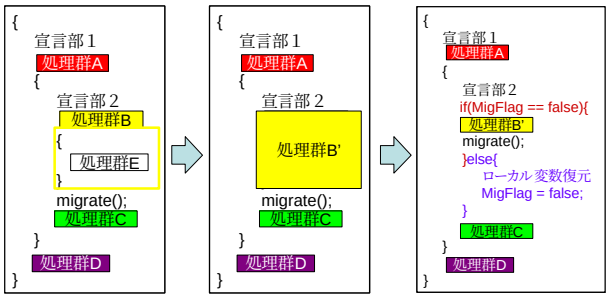


図６　migrate 関数が関係していないスコープの変換

ところで，実行状態を復元している最中には，migrate 関数やマイグレートブロックが含まれるスコープには無条件に入る必要がある。スコープに入るかどうかは，if 文や for 文などの条件文で判断する場合が多い。そこで，この条件文に「MigFlag ==true」という条件式を追加し，これを元の条件文との論理和に変換した。MigFlag が true の時には，復元を行うために無条件でスコープに入り，false の時には元の条件文の真偽でスコープに入るかが決まることとなる。図７は if 文や for 文に，どのように条件式が挿入されるかを示している。

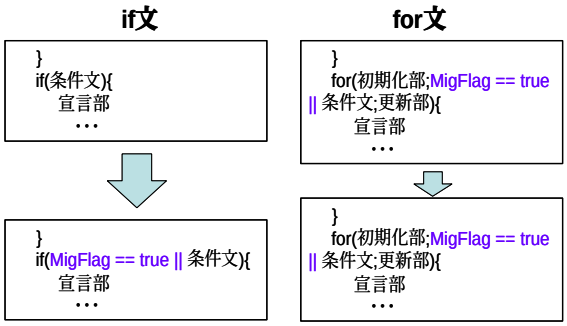


図７　条件文に追加する条件式

(2) ユーザ関数からマイグレートする場合

次に，migrate 関数がユーザの作成した関数の中に存在した場合に，どのように変換するかを説明する。図８はユーザ関数からマイグレートする場合のソースコード変換方法を示したものである。

ユーザ関数であっても，その関数内は単一あるいは複数のスコープで構築されている。そのため，ユーザ関数内の変換は可能である。

ここで，変換されたユーザ関数全体をマイグレーションブロックとおくと，ユーザ関数を呼び出しているスコープも基本形と似た形となるので，変換することができる。この時，ユーザ関数は必ず呼び出さなければならないため，スキップする条件文の中には含めず，変数の復元等が含まれた else 文を抜けた後に記述される。こうすることで，移動直前の JavaVM スタックの構造まで復元することが可能となっている。

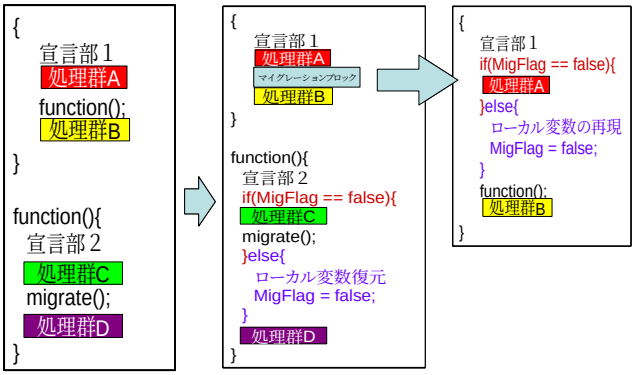


図８　ユーザ関数からマイグレートする場合の変換

(3) プログラム中に複数の migrate 関数がある場合

ソースコード変換時に，プログラムコード中に migrate 関数がいくつ存在するかはわかるので，その数だけ MigFlag を用意する。現在，MigFlag は boolean 型の Vector となっている。

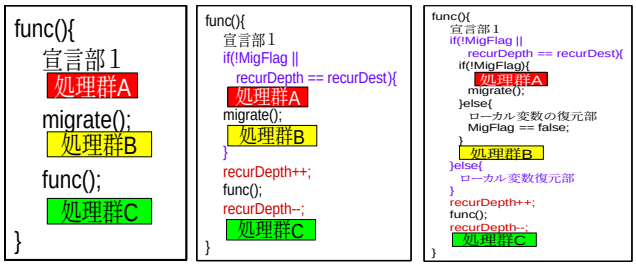
そして，ソースコード変換時に，各 migrate 関数にシーケンスとなる番号を付け，その番号に対応する MigFlag を使って処理をスキップするか判断する。

ある migrate 関数から見ると，他の migrate 関数によって挿入された条件文などは，その migrate 関数に関係のないスコープとなっている。そのため，他の migrate 関数や挿入した条件文はすべて処理群の一部として考え変換することができる。このように，migrate 関数の変換は別の migrate 関数の変換に対して不干渉なので，それぞれの migrate 関数ごとに独立して，変換を行うことができる。

(4) 再帰関数からマイグレートする場合

関数中からのマイグレートは変換が可能で，移動前のスタックフレームの構造も復元が可能であると説明した。再帰関数からのマイグレートも可能性としては考えられる。しかし，再帰関数は再帰の深さだけスタックフレームが積まれていくので，再開時にもまったく同じスタックフレーム構造を復元する必要がある。

図９は，再帰関数の中からのマイグレートを可能とする変換手順を示したものである。



初期状態　　再帰復元用の処理挿入後　　変換完了時

図９　再帰関数からマイグレートする場合の変換

移動する時には，その時の再帰の深さを保持しておき，移動先で目的の深さになるまで再帰処理をスキップしながら関数呼び出しを行い，スタックフレームを復元する。復元が完了したら，migrate 関数の位置から実行を再開する。このような流れとなるように，再帰関数を変換する。

そのために，再帰の深さをカウントし記憶しておくための変数として recurDepth と recurDest を用意した。

変換の手順は次のようになる。

I. 再帰関数の直前に recurDepth++を挿入
II. 再帰関数の直後に recurDepth--を挿入
III. 再帰関数以前の処理を全て if (!MigFlag || recurDepth == recurDest)でスキップ
IV. else 文中に，そのスタックでのローカル変数を復元する処理を挿入

これらの手順を踏んで変換された結果が図９の中央の変換結果となる。あとはこのコードを変換の基本方式で変換することで，変換が完了する。

変換されたコードを実行し，再帰関数に至り，再帰が行われると，recurDepth が再帰の深さをカウントしていく。そして，移動時に recurDest にその値をコピーする。そして，recurDepth を初期値の 0 に戻し，移動後には復元のために recurDepth のカウントを行う。

recurDepth＜recurDest の時には，スタックフレームの復元が済んでいないとみなし，再帰関数以前の処理を全てスキップする。

このように再帰関数からのマイグレートも実現できるが，実行効率上は，スタックフレームの復元が大きなオーバヘッドとなるので，利用することはあまり推奨できない。

４．強マイグレーション方式のモバイルエージェントシステム AgentSphere

４．１　AgentSphere の機能拡張

３章で説明した手法を用いた強マイグレーションモバイルエージェントシステムを AgentSphere と名づけ，我々が提案してきた自律分散処理のプラットフォームを AgentSphere に合わせて再構築することとした。従来の自律分散処理の機能の移植をより簡単にするため，AgentSphere のシステムを AgentSpace と類似したシステム構成にすることにした。

そのため，AgentSpace にも存在した以下のような機能を AgentSphere に追加した。

４．１．１　エージェント ID

ネットワークに複数存在するエージェントを一意に識別するためのエージェント ID の実装を行った。これは，ハッシュコード，生成された時間，アドレス，生成されたマシンでのシーケンスな番号を組み合わせている。この ID はエージェントが生成された時に割り振られる。

４．１．２　ユーザとシステム間のインタフェース機能

ユーザがエージェントの生成，移動，消去，情報開示などをシステムに指示できるコマンド機能を次のように追加した。

(1) エージェント起動コマンド load

『load　ファイル名』と入力すると，指定されたファイルを読み込み，エージェントを生成する。

(2) エージェントの情報開示コマンド list , info

移動や削除などの指示をエージェントに出すために，そのエージェントを特定する情報が必要となる。そのようなエージェントの情報を開示するコマンドとして次の二つを用意した。

『list』と入力すると，現在活動中の全てのエージェントの ID(識別子)を表示させる。

『info　ID』入力で自身のマシンの中から同じ ID のエージェント情報を引き出し，実行中のファイル名，エージェントのクラス名などを出力する。

(3) エージェントの転送コマンド go

『go　ID　送り先の IP アドレス』

エージェントの ID でエージェントを指定し，送信先のマシンの IP アドレスを指定することでエージェントの送信を行えるようにした。

(4) その他のコマンド

『destroy　ID』と入力することにより，指定したエージェントを削除する。

『help』により，以上のコマンド一覧が出力される。

４．１．３　複数のエージェントを生成・管理する機能

本研究の目指す自律分散処理システムでは，一つの AgentSphere に複数のエージェントが存在することが必要である。そこで，AgentSphere 内に存在する複数のエージェントの情報をリストとして管理する機能を作成し，

複数のエージェントが同時に存在することが可能となるようにシステムを拡張した。

4．1．4　エージェントとシステム間のインタフェース機能

エージェントが AgentSphere にアクションを行おうとする時には，システム内の対応するクラスやデータに直接アクセスしている。一方，ユーザが記述するような分散処理用のエージェントプログラムにおいても，エージェントからシステム側へのアクションを記述することが考えられる。

しかし，それを記述するためには，対応するクラスやデータについて詳しく知っている必要があり，本研究の目的とする「分散処理の手法について詳しくない人でも簡単に分散処理の恩恵を受けられるシステムを構築すること」に相反することとなる。さらに，エージェントプログラムから，AgentSphere のシステム全てにアクセスできるようでは，セキュリティ上非常に危険である。

そこで，エージェントとシステム間のインタフェースとなるクラスを設けた。ユーザが記述するエージェントプログラムでは，このクラスを利用するだけでシステムへのアクションをとることができると同時に，このクラスで許可されている事以上のアクションをシステムに行うことができないようにした。

4．2　AgentSphere の動作確認

今回このシステムの動作確認をするに当たって，データ数 10 個のバブルソート(昇順)を２台のマシン間で行うことにした。あらゆる制御構造にも対応していることを確認するため，ソースコード中の色々なスコープ内に migrate 関数を記述し，それぞれの場合においても正しく動作が行われているかを確認した。図 10 には，動作確認に用いた migrate 関数記述後のソースコードの一例を，そして図 11 にはソースコード変換後のコードを示す。

```
import java.io.Serializable;
import agentsphere.*;
public class BubbleSort extends StrongMigAgent implements Serializable {
    public void create() {
    int data[] = ソートを行うデータ;
    sort(data);
}
void sort(int[] data){
    int i, j, temp;
    for ( i = 0; i < data.length - 1; i++ ) {
        for ( j = data.length - 1; j > i; j-- ) {
            if ( data[j - 1] > data[j] ) {
                temp = data[j - 1];
                data[j - 1] = data[j];
                data[j] = temp;
            }
        }
        print( data );    /*配列表示用の関数*/
        if (i%2==0) migrate("IPAddress2");
        else migrate("IPAdress1")
    }
  }
}
```

図 10　動作確認に用いたソースコードの一例

表１　動作確認に使用したマシン一覧

マシン名	CPU名	クロック数	メモリ
マシンA	Core 2 Duo	2.2GHz	2.0GB
マシンB	PentiumD	2.8GHz	1.0GB

このコードは，ソートを行う配列のデータの中で一番小さい値が確定したら次のマシンに移動し，次に小さい値を確定し，最初のマシンに戻って処理を行うというような処理を行っている。そして，移動前のマシンでの配列の状態が保存されているかの確認を行っている。

表１は，今回評価に使用したマシンである。マシン A の１台で処理を行った結果を図 12 に示す。次に，２台のマシンで処理を行った時のマシン A での結果を図 13－１に，マシン B での結果を図 13－２に示す。

図を見て分かるとおり，移動前の配列の状態を保持しながら移動し，移動後ではその続きから処理を行っているのがわかる。

```
public class BubbleSort extends StrongMigAgent
                                implements Serializable{
    private String IPaddress;
    private AgentInfo Ainfo;
    public void create() {
    /** 確認用のソート対象データ*/
    int[] array = { 117, 75, 24, 32, 98,72, 88, 43, 60,
35};

    if(Ainfo.MigFlag == 0){
        print(array);
        System.out.println("Sort Start!!");
    }else{
      //ローカル変数復元部
      array = this.getArrayIntegerValue("array",0);
    }
    sort( array );

    //ソートの結果の確認
    System.out.println("SortFinished!!");
    print(array);
    }

public void sort( int[] array ) {
    int tmp=0; //作業領域
    for( int i=0;Ainfo.MigFlag!=0 || i<array.length-1;
i++ ) {
        if(Ainfo.MigFlag == 0){
            for( int j=0;j<array.length-i-1; j++ ) {
                if(array[j] > array[j+1] ) {
                    tmp = array[j];
                    array[j] = array[j+1];
                    array[j+1] = tmp;
                }
            }
            if(i%2 == 0) IPaddress = "133.220.114.108;
            else IPaddress = "133.220.114.240";
            System.out.print(i + " : ");
            print(array);
            migrate(IPaddress,1);
        }else{
          //ローカル変数復元部
          array =
this.getArrayIntegerValue("array",0);
          tmp = this.getIntegerValue("tmp",0);
          i = this.getIntegerValue("i",0);
          Ainfo.MigFlag = 0;
        }
    }
}
```

図 11　ソースコード変換後のコード

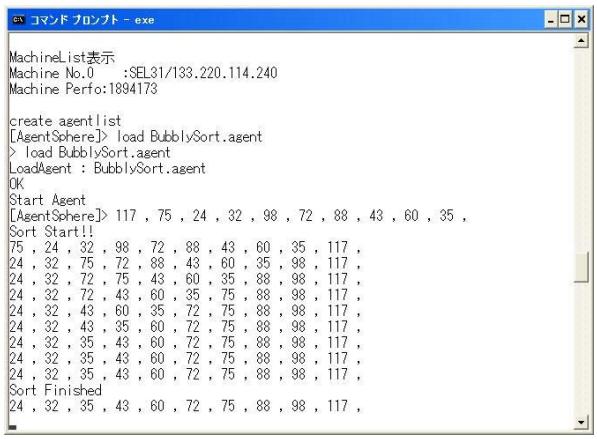


図 12　一台のマシン（マシン A）での実行画面

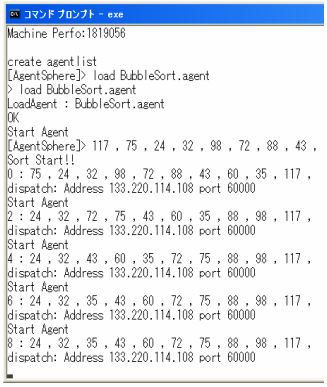


図 13－１マシン A

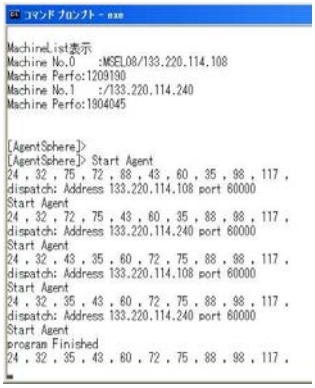


図 13－２マシン B

５．終わりに

スタック領域の取得，ソースコード変換を行うことによって，既存の JavaVM に変更を加えることなく，強マイグレーションエージェントの記述ができるようになった。

また，新しいソースコード変換方式にすることで，プログラムの制御構造に依存したソースコード変換しか行えなかった従来のシステムを，制御構造に依存せずに migrate 関数を記述することができるようになった。

本研究では，ユーザがエージェントに記述する強マイグレーションコードを弱マイグレーションコードにソースコード変換する仕様を定めた。しかし，ソースコード変換を自動で行えるソースコード変換機はまだ作成していない。今後の課題としては，本年度決定したソースコード変換仕様に基づいて，ソースコードを自動で変換する自動変換機の作成，移動オーバヘッドに関する評価を行い，ソースコード変換仕様・ローカル変数取得復元機能を改善し性能の向上を目指すこと，取得・復元できるローカル変数の種類を拡張することなどが挙げられる。

参考文献

[1] 佐藤一郎：「AgentSpace：モバイルエージェントシステム」,日本ソフトウェア科学会, Dec.1998

[2] 日本 IBM 東京基礎研究所：
http://www.trl.ibm.com/aglets/ , Jul.2006 参照可

[3] 米澤明憲,関口龍郎,橋本政朋：「移動コード技術に基づくモバイルソフトウェア」
http://homepage.mac.com/t.sekiguchi/javago/index-j.html, Jul.2006 参照可

[4] 首藤一幸： http://www.shudo.net/moba/ ,Jul.2006 参照可

[5] Ryusuke Sasaki, et al.: “Implementation And Evaluation Autonomic Distributed Processing System Using Mobile Agent”, Proc.of IEEE Pacific Rim conference, No.237, Aug.2005

[6] 坂井功，相河寛典：「モバイルエージェントを用いた自律分散処理システムの構築　－自律分散処理をサポートする管理エージェントの作成－」2003 年度電気学会電子・情報・システム部門大会講演論文集, pp.1142-1146,Aug.2003

[7] 佐々木竜介，遠藤　航，青山　迅，清水敏宏，小川大介，坂井　功：「モバイルエージェントを用いた自律分散処理システムの構築－自律分散処理システムにおける性能とユーザビリティの向上－」2004 年度電気学会電子・情報・システム部門大会講演論文集, pp.1036-1040, Sep.2004

[8] 小川大介，佐治大介，村本洋明：「モバイルエージェントを用いた自律分散処理システムの構築－タスクエージェントの記述法とタスク分散の効率化－」2003 年度電気学会電子・情報・システム部門大会講演論文集, pp.1116-1120,Aug.2003

[9] 田久保雅俊，佐々木竜介，内藤智史，小川大介:「モバイルエージェントを用いた自律分散処理システムの構築　－エージェント間通信命令の実装と分散処理エディタの試作－」2005 年度電気学会 電子・情報・システム部門大会講演論文集， GS15-4, pp.1034-1039, Sep.2005

[10] Torsten Illmann,et.al, “Transparent Migration of Mobile Agents Using the Java Platform Debugger Architecture”, Mobile Agents: Proceedings of the 5th International Conference, Ma 2001 Atlanta, Ga, December 2-4, pp.198- 212, 2001