成蹊大学理工学研究報告
J. Fac. Sci. Tech., Seikei Univ.
Vol.45 No. 1 (2008) pp.21-29

強マイグレーションモバイルエージェントのための ソースコード変換器の実装

櫻井　康樹*1，加藤　史彬*1，津田　達也*2，甲斐　宗徳*3

An Implementation of a Source Code Translator for Strong Migration Mobile Agent

Yasuki SAKURAI*1, Fumiaki KATOH*1, Tatsuya TSUDA*2, Munenori KAI*3

ABSTRACT：We have been developing a strong migration mobile agent system, called AgentSphere, which enable users to write strong migration agent codes in Java by putting “migrate” method at any place in the code. The strong migration agent codes have to be transformed into the codes which can be executed on existing Java virtual machines without any modification. In this paper, we first show some basic structural patterns in which “migration” methods are used in practical agent codes, then implement an automatic agent code transformer from strong migration mobile agent codes to the code for ordinary Java virtual machines by using JavaCC(Java Compiler-Compiler).

Keywords：mobile agent system, strong migration, source code transformation

（Received March 25, 2008）

１．はじめに

並列分散処理とは，ネットワークに接続された複数のマシンに負荷を分散させて並列に処理することにより，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多くのユーザがそれぞれのアプリケーションの分散処理を要求するため，それらが投入されたことによる負荷の変動に合わせて，その後のアプリケーションの分散処理の仕方を適切に変えていかなければ，高性能でかつ信頼性の高い分散処理は実現できない。その対応を個々のユーザがアプリケーションに組み込むのは決して容易ではない。

そこでシステム自身に自律性を持たせることにより，それを利用するユーザのアプリケーションにおいても自然と自律性を記述でき，実際に自律的な振る舞いをすることが可能となる。

筆者らは，これまでに自律性を実現するためのモバイルエージェントシステムを設計してきた。しかし，これまで設計し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。

高い自律性を実現するには，採用するモビリティは強マイグレーションであることが望ましい。そこで，強マイグレーションモバイルエージェントシステムを実現する方法として，JPDA(Java Platform Debugger Architecture) によるローカル変数の取得・再現手法と，ソースコード変換手法について提案した[1]。

これまでに，ソースコード変換手法の概念は完成していたが，その変換は手動で行っていた。そこで，本研究では，JavaCCを用いたソースコード変換自動化の実装を行った。

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，システムの状況の変化にシステム自身が自律的に対応しながら分散処理を行うシステムである。一般的に分散システムにおいて，タスクの負荷分散や停止したタスクの再生処理等の作業にはシステム全体の状況把握や管理が必要であり，それらのコントロールをすべて考慮してプログラム記述者がアプリ

*1：工学研究科情報処理専攻修士学生
*2：工学部経営・情報工学科学生
*3：情報処理専攻教授 (kai@st.seikei.ac.jp)

ケーションプログラムを記述するのは非常に困難であると考えられる。そこで，あらかじめシステムに自律性を持たせることで,これらの問題を解決することができる。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。そこで本研究では，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た。

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としては，AgentSpace（佐藤一郎氏）[2]やAglets（日本IBM社）[3]，JavaGO（東京大学米澤研究室）[4]，MOBA（首藤一幸氏）[5]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

筆者らは，モバイルエージェントシステムのソースが公開され改造が可能となっていて，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システムのプラットフォームの開発を行っていた。

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。

前者の目的を充たすために，ユーザが要求する処理を効率よく分散するような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。そして後者の目的を充たすために，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[6]。

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，その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中断した時点の実行時データを保持し，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，これまで利用してきた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った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った。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おける問題点とその解決法

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（プログラムの実行コードを格納する領域），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 1.6.0_04 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。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としては，JavaGoX[7]，MOBA，Nomads[8]，Brakes[9] などが挙げられるが，これらの研究では JavaVM 自体の変更を行ったり，バイトコード変換を行ったりしている。JavaVM を変更する場合，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。

そこで本研究では，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移動命令を migrate(移動先ホスト名); のように記述するだけでマイグレーションが可能で，この命令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. 2 スタック領域の取得

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である[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 3.1 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。

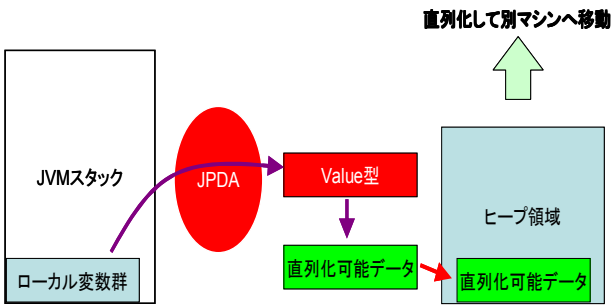

図 3.1 ローカル変数持ち出しまでのプロセス

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)であるため，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。そうすることにより，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，別マシンに送信することができるようになる。

3. 3 スタック領域の復元

スタック領域を取得し，別マシンに移動した後，移動先のマシンで実行を再開するには，マイグレーション命令の直後までの状態を復元し，そこから実行を再開する必要がある。その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される。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である。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。

この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。

3. 4 ソースコード変換

3. 4. 1 ソースコード変換の基本形

ソースコード変換の手法は，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。

この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 を用意する。このフラグが true のときには移動直後であることを示す。初期値は false であり，エージェントの移動が行われると，true になる。

この MigFlag と if～else 構文を組み合わせ，移動直後には実行再開位置までのスタックフレームを再構築し，移動前にすでに得られていた値についての計算処理は全てスキップするようにソースコードを変換する。この

変換により，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。

具体的な変換作業は，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，その後の変換の前準備として，注目しているスコープ内で定義している宣言文をスコープの先頭にまとめて配置する。その結果として得られる**「宣言部・処理群・migrate 関数・処理群」**という構造が変換対象の基本形となる。プログラム中で利用されるどのような構文でも一般的にこの基本形に直すことができる。

図 3.2 は，この基本形から JavaVM 上での強マイグレーションコードへの変換方式を示している。ここで，ローカル変数の復元は，移動前の個々のローカル変数の値を移動後の対応するローカル変数に再代入するための一連の代入文をソースコードに挿入することにより実現する。

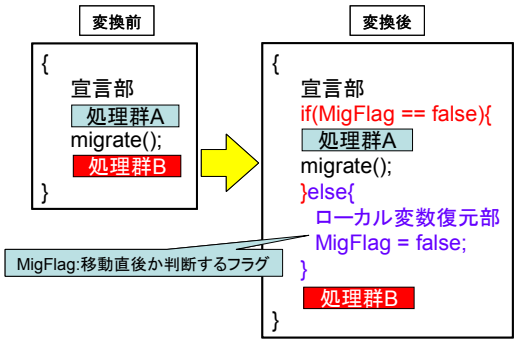

図 3.2　変換の基本方式

4．JavaCC を用いたソースコード変換の自動化

ソースコード変換の自動化にあたり，JavaCC(Java Compiler Compiler)を使用した。JavaCC とは，lex/yacc を Java 言語に置き換えたものであり，サン･マイクロシステムズ社が提供しているオープンソフトである。[11]本研究では, JavaCC を用いることによりソースコード変換の自動化を実現する。

4．1　コード変換までの流れ

JavaCC を用いてソースコードを解析し変換を行うとき，migrate 関数が使用されている個数およびその位置によって変換後のソースコードの形が異なるので，ソースコードを 1 パスで解析し変換するのは困難である。そこで，字句解析・構文解析を一度行って migrate の使用状況をはじめ必要な情報を取得し，その後もう一度ソースコードを解析して，最初に取得した情報を元にコード変換を行う。このように，2 パスの解析・変換方式を用いた。まず字句解析・構文解析について説明する。

4．2　字句解析

字句解析では，ソースコードからトークンを切り出して構文解析へと渡さなければならない。図 4.1 は，字句解析のコードの一部を示したものである。

```
【Javaのプリミティブ型】
  TOKEN:
  {
        < BOOLEAN: "boolean" >
  |     < BYTE: "byte" >
  |     < CHAR: "char" >
  |     < DOUBLE: "double" >
  |     < FLOAT: "float" >
  |     < INT: "int" >
  |     < LONG: "long" >
  |     < SHORT: "short" >
  }
```

図 4.1　字句解析の例

4．3　構文解析

構文解析フェーズでは，トークンの並びに対し，BNF 記法に基づいて構文を定義する。また認識された生成規則に対し，変数情報のリスト，変数宣言のリスト，migrate メソッドの位置情報をアクション部にて取得する。

図 4.2 は，構文解析のコードの一部を示したものである。

```
【Javaの文】
 void Statement(): {}
{
        Block( )
  |     <ASSERT> Expression( ) ( <CL> Expression( ) )? <SM>
  |     <IF> ParExpression() Statement( ) ( <ELSE> Statement( ) )?
  |     <FOR> <LP> ForControl() <RP> Statement( )
  |     <WHILE> ParExpression( ) Statement( )
  |     <DO> Statement( ) <WHILE> ParExpression( ) Statement( )
  |     <TRY> Block( ) ( Catches( ) )? ( <FINALLY> Block( ) )?
  |     <SWITCH> ParExpression( ) SwitchBlockStatementGroups( )
  |     <SYNCHRONIZED> ParExpression( ) Block( )
  |     <RETURN> ( Expression( ) )? <SM>
  |     <THROW> Expression( ) <SM>
  |     <BREAK> ( Identifier( ) )?
  |     <CONTINUE> ( Identifier( ) )?
  |     <SM>
  |     LOOKAHEAD(2) StatementExpression( )
  |     LOOKAHEAD(2) Identifier() <CL> Statement( )
  }
```

図 4.2 構文解析の例

4．4　変数リスト，変数宣言行リストの作成

次にソースコード変換に必要な情報の取得について説明する。変数宣言をスコープの先頭にまとめるためには，一度目の解析の際に，宣言されている変数のリストを作成する必要がある。そのリストに含まれる情報は以下の通りである。

・変数の型，名前
・宣言されている関数の名前
・配列か否か(復元部での記述が変化)
・for,while 文などで宣言されているか
・宣言されているスコープの深さ

変数の型，名前の他に，初期値が宣言されている場合には，その情報も保持しなければならないが，配列などの場合，その取得が困難であるため，変数宣言の行そのものをリスト化することにした。この操作により，変数宣言をスコープの先頭にまとめる際の作業が簡略化できる。

変数リスト，変数宣言行リストの例を以下に示す。

```
public class ClassTest {
    println();
    public void create(){
        int[] d = new int[3];
        int[] array = { 117, 75, 24, 32};
        int a[],c=3;

        c = 5;
        d = new int [3];
        int aaaa=10;
        a[2] = 10;
    }

    public static void main(int a, int b){
    }
}
```

図 4.3 ソースコードの例

変数リスト

```
変数型:int 名前:d 配列:1
変数型:int 名前:array 配列:1
変数型:int 名前:a 配列:1
変数型:int 名前:c 配列:0
変数型:int 名前:aaaa 配列:0
```

図 4.4a 変数リスト

変数宣言行リスト

```
int[] d = new int[3];
int[] array = { 117, 75, 24, 32};
int a[],c=3;
int aaaa=10;
```

図 4.4b 変数宣言行リスト

図 4.3 はリスト作成のために使用するソースコードの例であり，それを解析にかけ，取得した各リストが図 4.4a，図 4.4b である。取得したこれらのリストは，変換を行うための解析で用いることになる。

4. 5 migrate 関数に関する情報の取得

ソースコード変換のためには，宣言されている変数のリストと共に，使用されている migrate 関数についての情報も必要となる。その必要な情報とは，以下の通りである。

・migrate が配置されている関数の名前
・migrate が配置されているスコープの深さ
・migrate が使用されている数

4. 6 関数呼出に関する情報の取得

migrate が，ユーザが作成した関数内にある事も考えられるため，どの関数を呼び出しているかの情報も取得する必要がある。その理由は，migrate 関数を呼び出すまでに終了している関数は，スタックフレームの復元を行う変換の必要が無いので，migrate に関与している関数の情報だけを取得し，その他の関数に対して不必要な変換を行わないようにするためである。これに必要な情報は以下の通りである。

・自身の関数名
・その関数内で呼ばれている関数のリスト
・呼び出した関数が migrate に関係しているか否か

4. 7 制御文に関する情報の取得

さらに，migrate 関数が if 文や for 文といった制御文の中に記述されていることも考えられるので，そのスコープ内で migrate 関数が記述されている制御文に関する情報も取得する必要がある。この場合も migrate 関数が記述されている制御文のみを変換し，その他の制御文に対して不必要な変換を行わないようにする。必要な情報は以下の通りである。

・if や for など，制御文のタイプ
・記述が開始されている行番号
・その制御文内で migrate 関数が呼ばれた回数

5．様々なソースコードに対する変換のパターン

5. 1 簡単なコードの変換

4.4 で述べた各リストと，migrate 関数の位置情報を元に，二度目の解析で，ソースコード変換の基本形に則って変換を行う。どのように変換を行うを以下に示す。

・プログラムファイル名は“Translated_”＋ オリジナルのファイル名とする。それに伴い，コード内のクラス宣言部分も変更する。
・migrate 関数呼出が含まれるユーザ関数に到達したら，取得した変数リスト，変数宣言行リストを用いて，スコープの先頭に変数宣言をまとめる。
・変数宣言をまとめた後，その直後の行に AgentCoordinator を取得する命令文を挿入する。
・migrate 前後かどうかを判断する if 文を挿入する。
・migrate 関数呼出までの処理郡を出力する。
・migrate 関数呼出後，自身のスレッドを終了させる関数の呼出文を挿入する。この関数自体も自動で挿入する。
・migrate 後に変数復元を行うための else 文，変数復元部，MigFlag の初期化を行う関数呼出を挿入する。

以上の作業を行うことにより，ソースコード変換が完了する。実際に変換を行ったコードの一部を図 5.1，図 5.2 に示す。図 5.1 は変換前のソースコードの一部で，図 5.2 がソースコード変換後のコードの一部である。変換後のコードを，実際に本研究で開発している強マイグレーションモバイルエージェントシステム AgentSphere で起動させたところ，正しく動作することが確認できた。

```
public class TransTest extends StrongMigAgent implements Serializable {
private String IPaddress = "192.168.11.2";
private static AgentCoordinator userAC;

    public void create(){
        int tmp = 0;
        String line = "hello";
        tmp = 3;

        int a = 0;
        a = 10;
        System.out.println(tmp);
        System.out.println(a);

        userAC.migrate(IPaddress,1);

        System.out.println(a);
        System.out.println(tmp);
    }
}
```

図5.1　変換に用いたソースコードの一部

```
public class Translated_TransTest extends StrongMigAgent implements Serializable {
    private String IPaddress = "192.168.11.2";
    private static AgentCoordinator userAC;

    public void create(){
        int tmp = 0;
        String line = "hello";
        int a = 0;
        userAC = getAgentCoordinator();
        if(userAC.checkflag()== 0){
        tmp = 3;
        a = 10;
        System.out.println(tmp);
        System.out.println(a);
        userAC.migrate(IPaddress,1);
        agent_cessation(userAC);
        }else{
            tmp = userAC.getIntegerValue("tmp",0);
            line = userAC.getStringValue("line",0);
            a = userAC.getIntegerValue("a",0);
            userAC.setFlag();
        }
    System.out.println(a);
    System.out.println(tmp);
    }...
```

図5.2　ソースコード変換後のコード

5. 2　より複雑なコードの変換

5.1 では単一関数での変換の例を述べたが，ここではユーザ関数からマイグレートする場合や，migrate 関数が複数存在する場合，またその両方の条件を満たす場合の変換について述べる。

どの変換においても，4.4，4.5，4.6，4.7 で述べたような情報が必要となる。以下の図にそれぞれの場合での情報取得の例を示す。

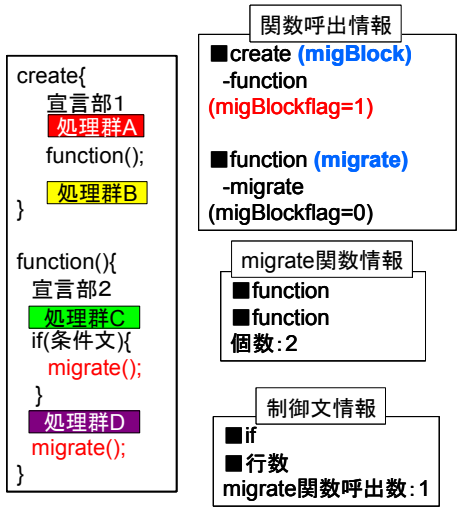

図5.3　変換に必要な各情報の取得

図 5.3 は，ユーザ関数内に複数の migrate 関数が記述されており，その一つが if 文の中に記述されている場合の例である。

5. 2. 1　ユーザ関数からマイグレートする場合

migrate 関数がユーザの作成した関数の中に存在した場合には，図 5.3 で示した情報を元に変換を行う。その変換の例が図 5.4 である。

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同様の形になる。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る。

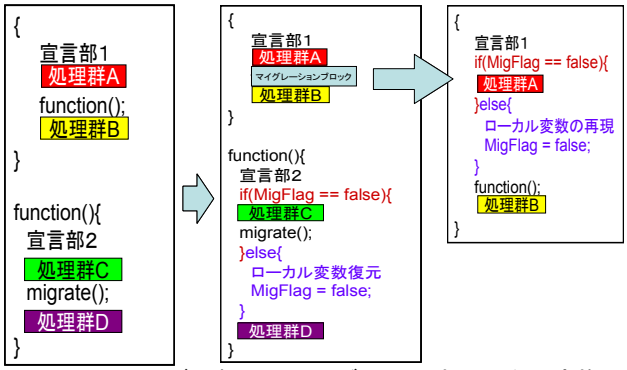

図 5.4　ユーザ関数からマイグレートする場合の変換

5. 2. 2　migrate 関数が複数存在する場合の変換

コード中に関数が複数存在する場合の変換には，図 5.3 で示した情報を元に変換を行う。その変換の例が図 5.5 である。

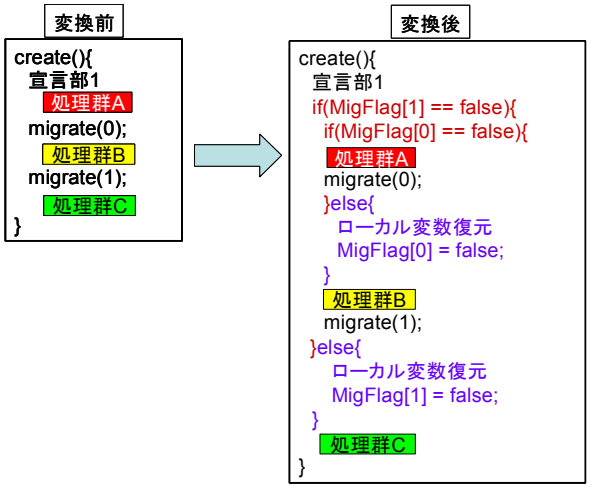

図 5.5　migrate 関数が複数ある場合の変換

図 5.3 で示しているように，ソースコード中に存在する migrate 関数の個数は取得できているため，その数だけ MigFlag を用意する。このとき MigFlag は boolean 型の配列となる。

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。

5. 2. 3　ユーザ関数内に migrate 関数が複数存在する場合の変換

5.2.1, 5.2.2 の両方の条件を満たしているコードの場合，migrate 関数が存在しているユーザ関数内の変換については，図 5.5 で示した変換と同様であるが，MigFlag が複数あるので，図 5.4 で示したユーザ関数を呼び出しているスコープの変換において if 文の条件式が異なる。その例を図 5.6 に示す。

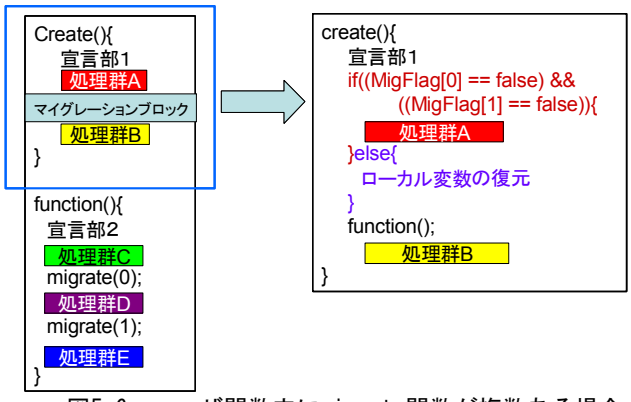

図5.6　ユーザ関数内にmigrate関数が複数ある場合

まず各 MigFlag が false の時は処理 A を実行し，2 つある migrate 関数を実行した場合は，どちらの場合でも処理 A をスキップする必要がある。そのため，ユーザ関数を呼び出しているスコープの変換の if 文の条件を，

((MigFlag[0] == false) && ((MigFlag[1] == false))

とすることによって上記の処理の流れを実現することが出来る。

さらに，ユーザ関数が増え，マイグレーションブロックが複数存在する場合でも，MigFlag を，マイグレーションブロックの個数と，migrate 関数の個数の積の数だけ用意すれば，変換可能である。

5. 2. 4　制御文の中に migrate 関数がある場合の変換

if 文や for 文などの制御文内に migrate 関数が記述されていた場合には，移動後に無条件にその制御文内に入る必要がある。そこで，その制御文の条件文に「MigFlag ==true」という条件式を追加し，これと元の条件文との論理和をとる。スコープ内で migrate 関数が複数記述されている場合は，その数だけの論理和を追加する。

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どうかが決まる。

制御文の中は**「宣言部・処理群・migrate 関数・処理群」**という基本形の変換を行う。if 文の例を図 5.7 に，for 文の例を図 5.8 に示す。

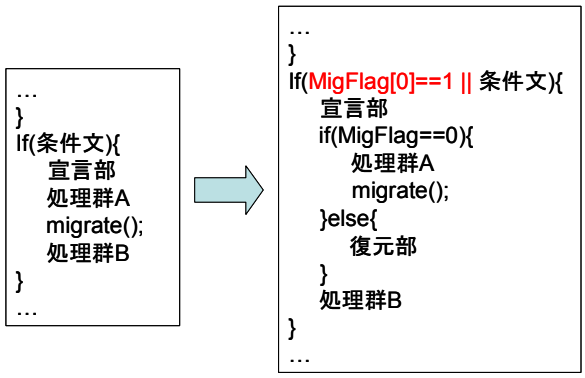

図 5.7　if 文に対する変換

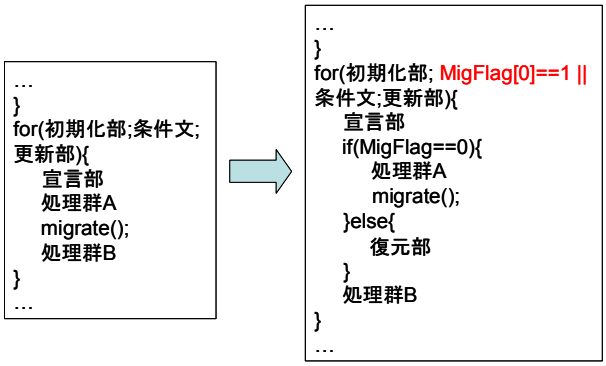

図 5.8　for 文に対する変換

6．　強マイグレーション方式にソースコード変換を行ったモバイルエージェントの動作確認

本研究室では，強マイグレーションモバイルエージェントを動作させるシステム AgentSphere を開発している。

これは，AgentSpace と同様に，AgentSphere を各コンピュータで起動することにより，エージェントをそれらのコンピュータに送り込むことができる。本研究で実装したソースコード変換器の動作確認をするために，図 6.1 のようなソースコードを用意した。そしてこのコードを自動変換し，AgentSphere 上で動かし，正しく動作が行われているかを確認した。図 6.1 のプログラムは，5 秒間 j の値をインクリメントし，その値を出力したら別のマシンへと移動を行う。また，i の値が指定した値になったら，別のマシンへ移動するというものである。これを 2 台のマシン間で行う。

```
public class TransCode extends StrongMigAgent
implements Serializable {
  private String IPaddress
  private static AgentCoordinator userAC;

  public void create(){
    int a = 20;
    func1();
  }

  public void func1(){
    int i=10,j=0;
    long start,end,sum;
    System.out.println("start!");
    for(int a=0; a<10; a++){
      start = System.currentTimeMillis();
      j=0;
      while(true){
        j=j+1;
        end = System.currentTimeMillis();
        if((end-start)/1000 > 5)  break;
      }
      if(a%2==0) IPaddress = "133.220.114.127";
      else  IPaddress = "133.220.114.240";
      i= i+1;
      System.out.println("a:" + a + " i:" + i + " j:" + j);
      userAC.migrate(IPaddress, 0);

      if(i==15){
        if(a%2==0)  IPaddress = "133.220.114.240";
        else  IPaddress = "133.220.114.127";
        userAC.migrate(IPaddress, 1);
      }
    }
    System.out.println("finish!!");
  }
}
```

図 6.1　動作確認に用いたソースコードの一例

そして migrate 関数は，どのような制御構造でも対応できることを確認するため，ユーザ関数の中かつ，for 文，if 文の制御文内に 2 つ記述してある。このソースコードを変換すると図 6.2 のようなコードが出力される。

そして，表 6.1 は動作に使用したマシンである。

表 6.1　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

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 6.3 に，マシン B での結果を図 6.4 に示す。

これらの図から，変数の状態を保持しながらエージェントが移動し，移動後ではその続きから処理を行っていることが確認された。

```
public class Translated_TransCode extends
StrongMigAgent implements Serializable {
  private String IPaddress;
  int count=0;
  private static AgentCoordinator userAC;

  public void create(){
    int a = 20;
    userAC = getAgentCoordinator();
    if(userAC.checkflag(0)==false &&
            userAC.checkflag(1)==false){
    }else{
       a = userAC.getIntegerValue("a",1);
    }
       func1();
  }
public void func1(){
  int i=10,j=0;
  long start,end,sum;
  if(userAC.checkflag(0)==false &&
            userAC.checkflag(1)==false){
    System.out.println("start!");
  }else{
       //変数復元部
       i = userAC.getIntegerValue("i",0);
       …..
  }
   for(int a=0;userAC.checkflag(0)==true ||
            userAC.checkflag(1)==true ||  a<10; a++){
     if(userAC.checkflag(1)== false){
          if(userAC.checkflag(0)== false){
               start = System.currentTimeMillis();
               j=0;
               while(true){
                  j=j+1;
                  end = System.currentTimeMillis();
                  if((end-start)/1000 > 5) break;
               }
               if(a%2==0)  IPaddress = "133.220.114.127";
               else   IPaddress = "133.220.114.240";
               i= i+1;
               System.out.println("a:" + a + " i:" + i + " j:" + j);
               userAC.migrate(IPaddress, 0);
          }else{
               //変数復元部
               userAC.setFlag(0);
          }
     }else{
          //変数復元部
     }
     if(userAC.checkflag(1)==true || i==15){
          if(userAC.checkflag(1)== false){
               if(a%2==0)  IPaddress = "133.220.114.240";
               else  IPaddress = "133.220.114.127";
               userAC.migrate(IPaddress, 1);
          }else{
               //変数復元部
              userAC.setFlag(1);
          }
     }
   }
  System.out.println("finish!!");
  }
}
```

図 6.2　ソースコード変換後のコード

```
コマンド プロンプト - exe
[AgentSphere]> load Translated_TransCode.class
> load Translated_TransCode.class
LoadAgent : Translated_TransCode.class
OK
[AgentSphere]> start!
a:0 i:11 j:107217203
dispatch: Address 133.220.114.127 port 60000
a:2 i:13 j:107063422
dispatch: Address 133.220.114.127 port 60000
a:4 i:15 j:106380543
dispatch: Address 133.220.114.127 port 60000
a:5 i:16 j:106395541
dispatch: Address 133.220.114.240 port 60000
a:6 i:17 j:106600873
dispatch: Address 133.220.114.127 port 60000
a:8 i:19 j:106574569
dispatch: Address 133.220.114.127 port 60000
finish!!
j:45028825
i:20
create:a->20
エージェント終了！
```

図 6.3　マシン A

```
コマンド プロンプト - exe
MachineList表示
Machine No.0    :SEL27/169.254.0.1
Machine Perfo:1350005
Machine No.1    :/5.161.20.146
Machine Perfo:2412426

[AgentSphere]>
[AgentSphere]> a:1 i:12 j:44126520
dispatch: Address 133.220.114.240 port 60000
a:3 i:14 j:44163856
dispatch: Address 133.220.114.240 port 60000
dispatch: Address 133.220.114.240 port 60000
a:7 i:18 j:45068821
dispatch: Address 133.220.114.240 port 60000
a:9 i:20 j:45028825
dispatch: Address 133.220.114.240 port 60000
```

図 6.4　マシン B

７．終わりに

本研究では，JavaCC を用いてソースコード変換器を実装したことにより，昨年度まで手動で行っていた変換が自動化できるようになった。そして，どのような制御構造でも変換が可能であることを確認できた。

また，変換されたコードが強マイグレーションモバイルエージェントとして正しく動作し，ソースコード変換が正しく行われていることが確認できた。

今後の課題としては，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，クラス変数等の取得・復元できるローカル変数の種類の拡張等が挙げられる。

参考文献

[1] 田久保雅俊, “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”, 成蹊大学大学院工学研究科情報処理専攻修士論文, 2006

[2] 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

[3] 日本 IBM 東京基礎研究所：http://www.trl.ibm.com/aglets/ , Jan.2008 参照可

[4] 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, Jan.2008 参照可

[5] 首藤一幸： http://www.shudo.net/moba/ ,Jan.2008 参照可

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

[7] Sakamoto, T., Sekiguchi, T., Yonezawa, A.: “Byte Code Transformation for Portable Thread Migration in Java”, In Proceedings of ASAMA’2000, Springer, Zuerich, Germany (2000) 16-28.

[8] Suri, N. et al.: “Strong Mobility and Fine-Grained Resource Control in NOMADS”, In Proceedings of ASAMA’2000, Springer, Zuerich, Germany (2000) 2-15.

[9] Truyen, E. et al.: “Portable Support for Transparent Thread Migration in Java”, In Proceedings of ASAMA’2000, Springer, Zuerich, Germany (2000) 29-43

[10] Illmann, T. 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

[11] https://javacc.dev.java.net/, Jan.2008 参照可