B-033

モバイルエージェントシステムAgentSphereの開発
—強マイグレーションエージェントコードのためのソースコード変換器の改良—
Development of Strong Migration Mobile Agent System AgentSphere
-Improvement of Source Code Translator for Strong Migration Agent Code-

疋田 直也† 坂巻 渉‡ 甲斐 宗徳†
Naoya Hikida Wataru Sakamaki Munenori Kai

1.はじめに

本研究ではモバイルエージェントの自律性を利用し、専門的な知識を持たない人でも並列分散処理を利用できるシステムを構築するため、自律型並列分散システムのためのプラットフォームとなる AgentSphere の開発を行っている[1] [2]。AgentSphere は OS やマシンに左右されず利用できるよう、仮想マシン上でプログラムが実行される Java を用いて実装を行う。また、AgentSphere では移動後も移動前の実行状態を継続して行うエージェントも利用できることを目指している。そこでエージェントの移動方式は、弱マイグレーションに加え、強マイグレーションにも対応させる。

Java では、シリアライズ機能を利用することによりヒープ領域内の情報を保存・移動する弱マイグレーションを実現することはできるが、スタック領域内の情報とプログラムカウンタを保存・移動する機能がサポートされておらず、何らかの手段でこの二つの情報を保存する機能を実装する必要がある。そこで本研究では、強マイグレーション記述されたコードを同等の動作を行う弱マイグレーションコードへとソースコード変換することにより対応する。

FIT2012 では、このソースコード変換の手法について、基本的な部分についての提案を行った。しかし、エージェントが移動を行う状況やタイミングによってはコード変換を正しく行うことが出来ないというケースが見つかった。そこで今回は、エージェントがどのようなタイミングで移動を行っても正しくコード変換が出来るようソースコード変換器を改良したので報告する。

2.ソースコード変換の流れ

2.1 ソースコード変換器の概要

AgentSphere におけるソースコード変換はスタック領域内の情報の保存・復元とプログラムカウンタに相当する情報の取得を目的としており、ソースコード変換器を実装することにより実現する。具体的なソースコードの変換方式は次章で述べ、ここではソースコード変換器の概要について述べる。図 1 のように、ソースコード変換器は変換対象となるエージェントのソースコードの構文を解析する構文解析器と、構文解析器に記述された各生成規則に対応したノードを生成するためのクラスによって構成され、解析結果をデータ構造として保存する。

構文解析器は Java 向けのパーサジェネレータである JavaCC を用いて実装を行う。そして実際のソースコード変換は図 2 で示すように、まず変換対象となるエージェントのソースコードを構文解析器にかけ、各生成規則と一致した際に、その生成規則と対応する構文木用クラスのインスタンスを生成する。ここで生成されたインスタンスをノードと呼び、各ノードをツリー状に保存する。こうして保存したツリー状のデータ構造を構文木と呼び、実際のソースコード変換は構文木に様々な操作を加えながら行っていく。

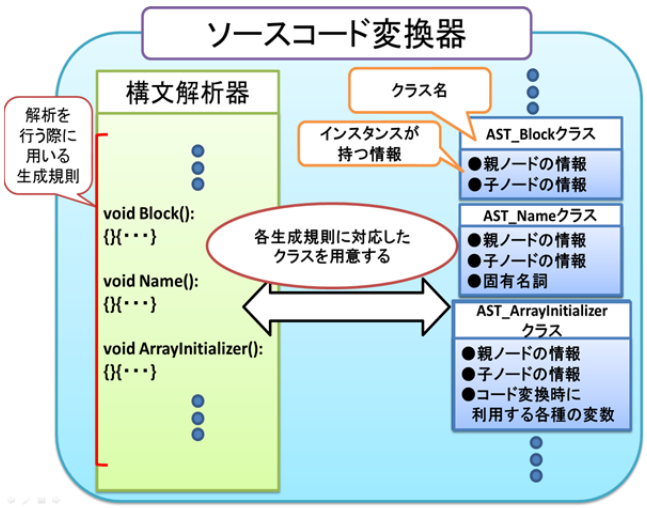


図 1:構文解析器とノード生成用クラスの関係

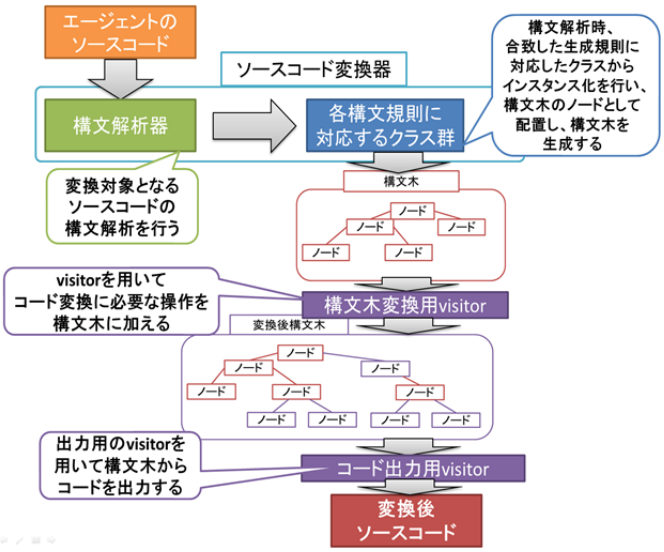


図 2:ソースコード変換器によるコード変換の流れ

† 成蹊大学理工学研究科理工学専攻 Graduate School of Science and Technology, Seikei University
‡ 成蹊大学理工学部情報科学科 Department of Computer and Information Science, Seikei University

2.2 構文木の概要

図 3 で示すように、構文木を構成する各ノードは共通して、自分の親ノードの情報と子ノードの情報を持つ。加えて、メソッド名や変数名などの固有名詞を表す終端記号を含む生成規則から生成されるノードには、解析された固有名詞の情報等が保存され、ソースコード変換時に利用される。

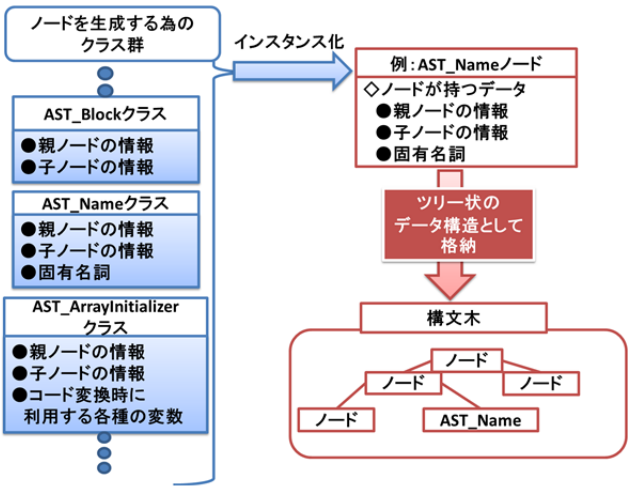


図 3:ノードを生成するためのクラスと構文木の関係例

2.3Visitor パターンの概要

Visitor パターンとは「オブジェクト」と「オブジェクトに対する操作」を分離して記述するソフトウェアの設計技法の一種である。オブジェクトと操作を分離することにより、オブジェクトのクラスに変更を加えずに、新たな操作を定義することが可能となる。この操作をまとめ、オブジェクト化したものは visitor と呼ばれる。

ソースコード変換器における visitor の振る舞いは、操作の対象である構文木全体を辿りながらソースコード変換に必要な操作を行う。構文木上の移動は、構文木の各ノードが持つ親ノードと子ノードの情報を利用して行う。また、特定ノードに移動した際、ノード内部の情報の変更や構文木に新たなノードを追加するなど、適切な操作を行うことでソースコード変換を行う。最終的に visitor が全てのノードを辿り、操作を加えた後の構文木からソースコードを生成することで、変換後のソースコードを取得することが出来る。

3.ソースコードの変換方式

3.1 従来の構文木変換方法

AgentSphere ではエージェントの移動を migrate()というメソッドを記述することで行い、migrate()の位置を基準としてソースコード変換を行うことによって、スタック領域の情報の保存・復元と移動後の実行再開位置の指定に対応する。そのコード変換に構文木を用いている。

FIT2012 で報告したソースコード変換器[1]では、基本の変換操作となる単一のスコープ中での変換を実装するまでにとどまった。今回は基本の変換操作を応用し、migrate 関数の含まれたどのような制御構造でも変換可能であるようにソースコード変換器の改良を行った[2]。

3.2.1 ソースコード変換手法改良の詳細

ソースコード変換器による対象コードの変換を、どのような制御構造をとるソースコードにおいても利用可能にするため、以下で説明する変換仕様を、構文木に対する操作を用いることによって実現した。

① 複数のスコープが関係する場合

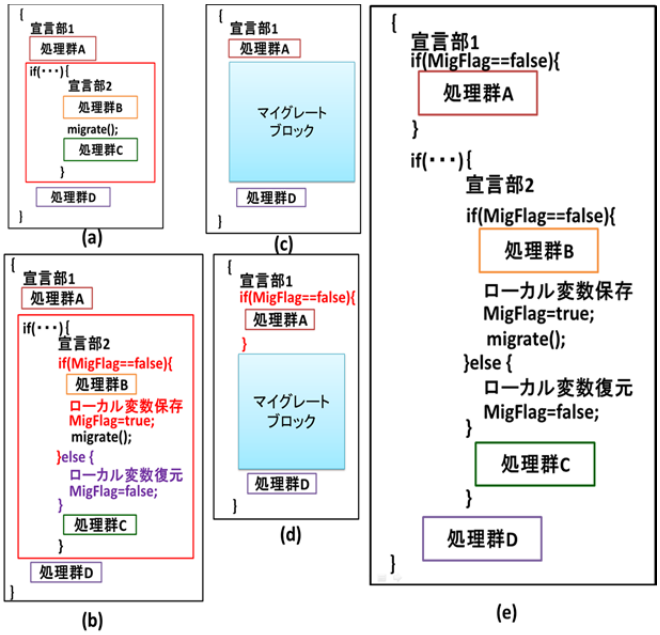


図 4:複数スコープが関係するコードの変換

多重スコープ中での変換も基本的な変換方法は変わらない。図 4 は複数の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。(a)は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。初めに、migrate 関数が存在するいちばん内側のスコープに着目する。このスコープ内部は基本の形となっている。これを基本の方式で変換すると(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と呼ぶことにする。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。これも基本の方式に則って、変換することができる。ただし、ローカル変数の復元処理に関してはマイグレートブロックの中で行ってしまったので、ここでの変換では復元処理を必要としない。ここでの変換の結果、(a)は(e)のように変換される。以上の手順によって、何重にスコープがある構造であっても、変換を行うことが可能となる。図 5 のように、migrate 関数が関係していないスコープが存在する場合、そのスコープをまるごと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。

また、エージェントは移動後、実行状態を復元するために、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

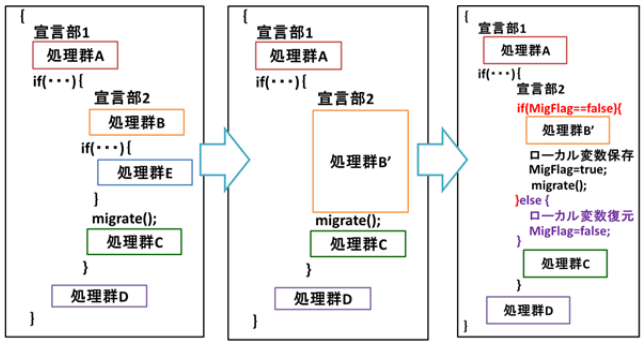


図 5:migrate 関数が関係していないスコープの変換

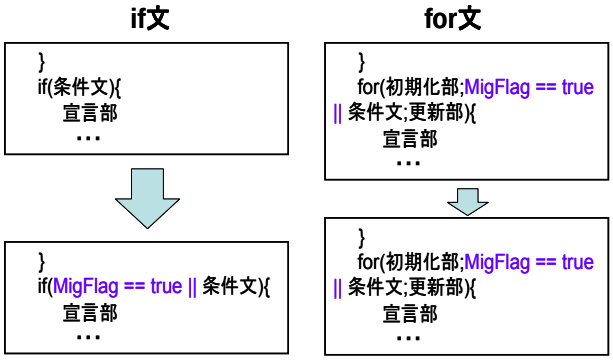


図 6:条件文に追加する条件式

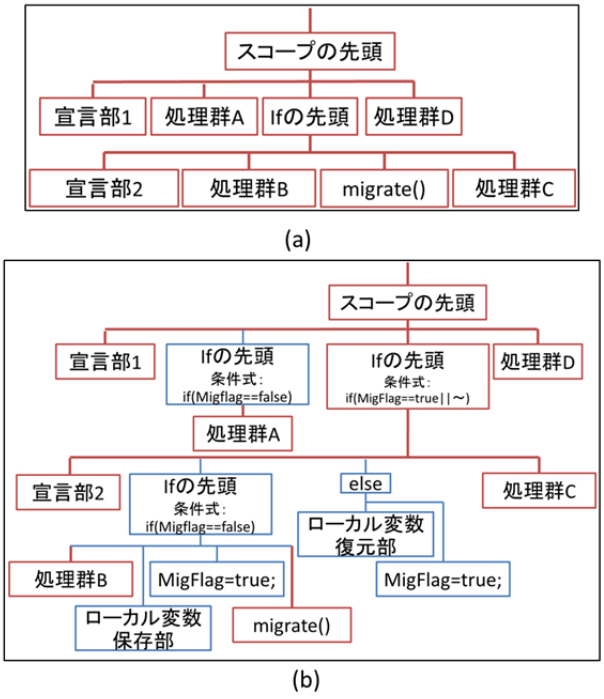


図 7:図 4 のソースコードの変換前後の構文木

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」という条件式を論理和により追加した。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 6 では if 文や for 文を例にとり、条件文に条件式が挿入される様子を示す。

ここで、構文木を用いてコードの変換を行う様子を見ていく。図 7 は図 8 の変換前コード(a)と変換後コード(e)を構文木化したものである。構文解析によって生成された図 7(a)に対して、構文木ために必要となる処理を加えるプログラム（visitor）は、すべてのノードを移動し、必要に応じてノードの移動や新たなノードの追加を行う。その結果、図 7(a)は図 7(b)のような構文木へと変換される。変換後の構文木から出力を行うことによって、図 8(e)のコードが生成される。

②　ユーザ関数からマイグレートする場合

次に、ユーザ関数内に migrate メソッドが含まれる場合の変換方法について説明する。図 8 はユーザ関数から migrate メソッドが呼び出され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ため、基本の変換及び複数スコープ内の変換方法を用いることで変換可能である。

そして、ユーザ関数の呼び出し元において、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。ここで、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。この変換によって、移動直前のスタック領域の構造まで復元することが可能となっている。

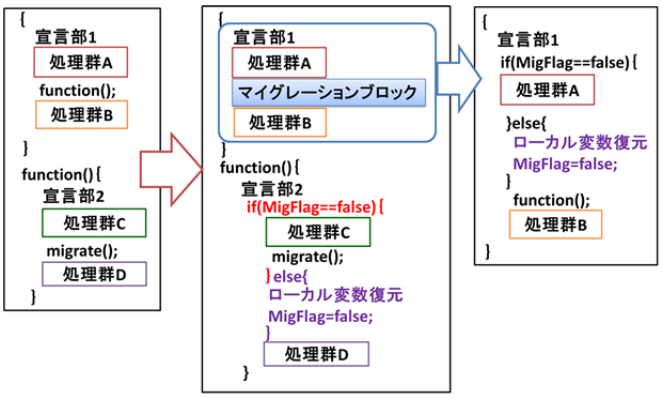


図 8:ユーザ関数からマイグレートする場合の変換

ここで、構文木を用いてコードの変換を行う様子を見ていく。図 9 は図 8 の変換前コードと変換後コードを構文木化したものである。ユーザ関数内に migrate メソッドが存在する場合も今までと同様に、構文木のために必要となる処理を加えるプログラム（visitor）は、すべてのノードを移動し、必要に応じてノードの移動や新たなノードの追加を行い、図 9(b)の形へと変換を行う。

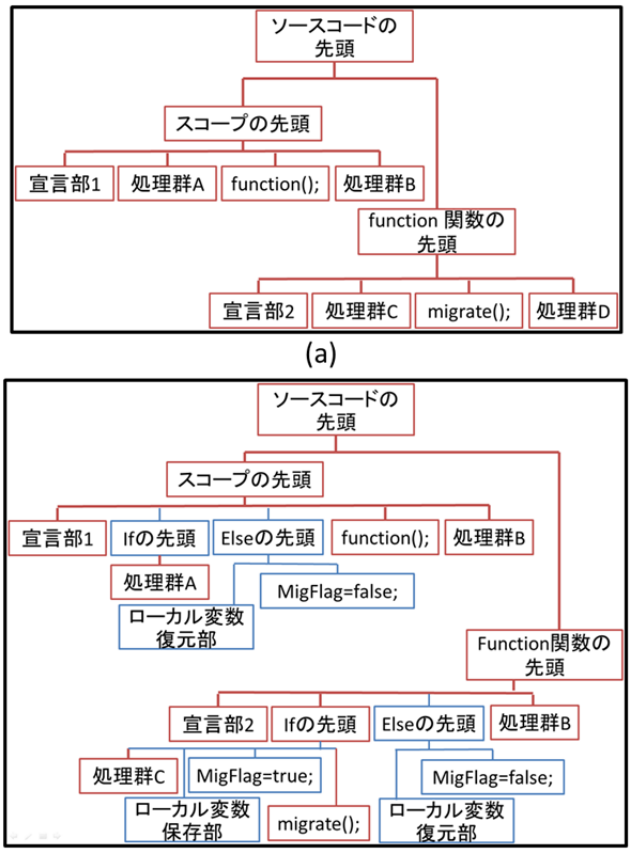


図 9:図 8 のソースコードの変換前後の構文木

③　プログラム中に複数の migrate 関数がある場合

図 10 は migrate メソッドが複数ある場合の変換について示している。構文解析時、変換対象のソースコード中に存在する migrate メソッドの数をカウントし、その数に応じた MigFlag を用意する。すなわち、MigFlag は boolean 型の配列にしている。

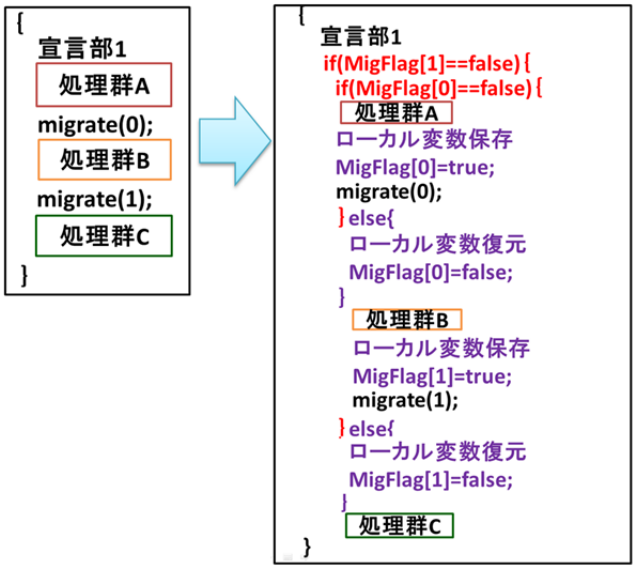


図 10:migrate メソッドが複数ある場合の変換

そして、ソースコード変換時に、各 migrate メソッドにシーケンス番号を付け、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 メソッドから見ると、他の migrate メソッド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 メソッド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 メソッドの変換は別の migrate メソッドの変換に対して干渉しないので、それぞれの migrate メソッドごとに独立して、変換を行うことができる。

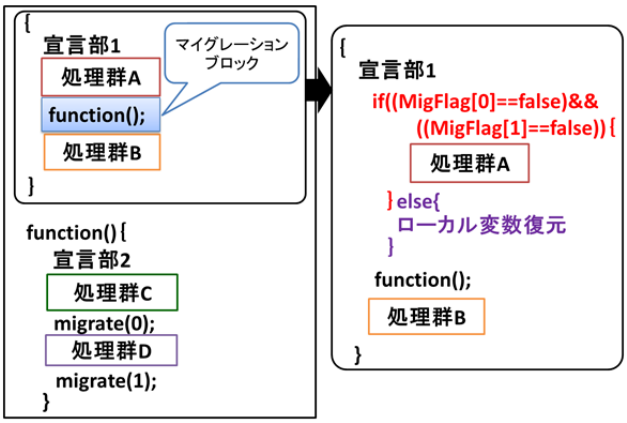


図 11:ユーザ関数内に migrate メソッドが複数ある場合の変換

また、ユーザ関数内に複数の migrate メソッドが存在している場合の変換において、ユーザ関数内は図 10 で示した変換と同様であるが、Migflag が複数存在するので、ユーザ関数を呼び出しているスコープの変換方法に違いが出る。この例を図 11 に示す。この例において migrate メソッドが呼び出され、移動先で実行状態を復元する際、いずれの migrate メソッドが呼び出された場合においても処理 A をスキップする必要がある。そのため、呼び出し元のスコープにおける変換の if 文の条件を、

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

とすることによって処理 A のスキップを実現する。

また、さらにユーザ関数が増えた場合においても、MigFlag をマイグレーションブロックの個数と migrate メソッドの個数の積の数だけ用意すれば変換可能となる。

次に、構文木を用いて複数の migrate が含まれたコードの変換を行う様子を見ていく。図 12 は図 10 の変換前コードと変換後コードを構文木化したものである。コード内に複数の migrate メソッドが存在する場合も今までと同様に、構文木のために必要となる処理を加えるプログラム（visitor）は、すべてのノードを移動し、必要に応じてノードの移動や新たなノードの追加を行い、図 12(b)の形へと変換を行う。

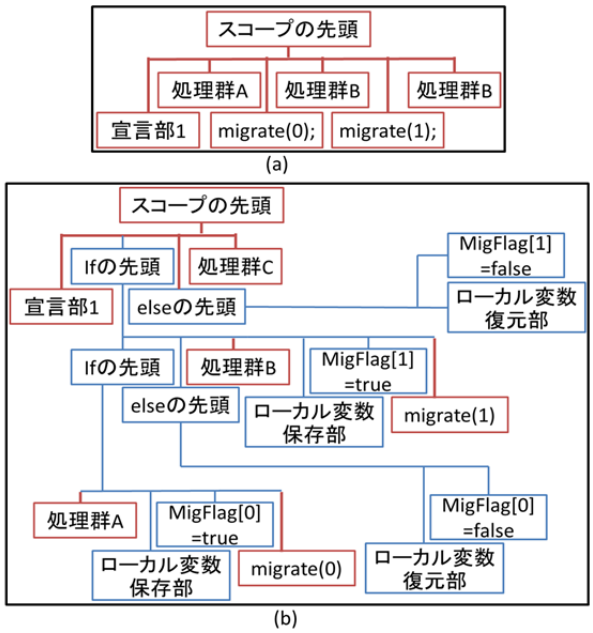


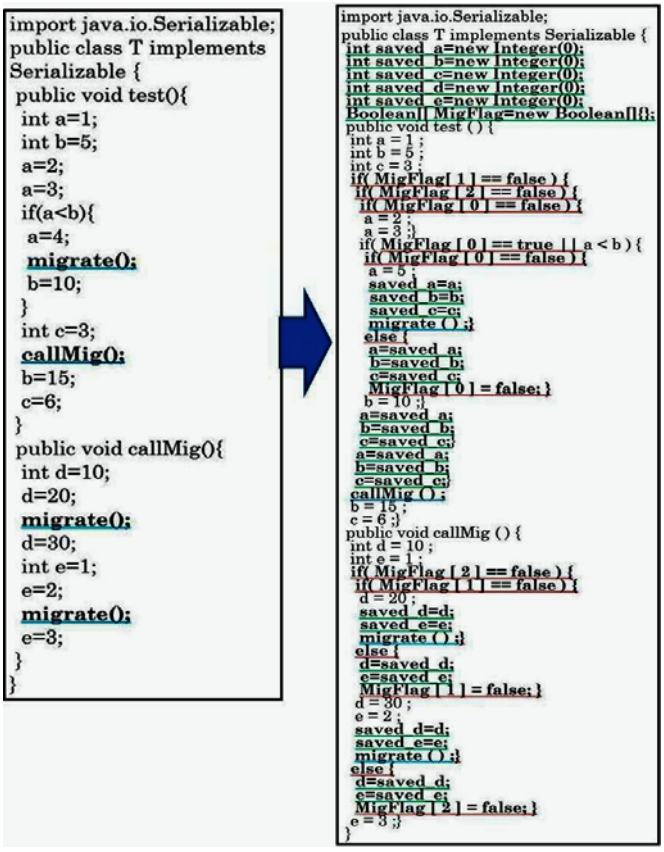
図 12:図 10 のソースコードの変換前後の構文木

④　再帰関数からマイグレートする場合

再帰関数からのマイグレートも実現したが、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.ソースコード変換器の実行結果

図 13 は変換操作の動作を確認するために作成したサンプルコードとソースコード変換器によって変換されたコードを示している。1 つ目の migrate メソッドは制御文内にあるため基本の変換に加え、if 文内に「MigFlag[0]==true ||」という字句が追加されている。また、スコープの外にも基本の変換の if 文が追加されている。2 つ目、3 つ目の migrate メソッドに対しては、他の関数内に存在しているため、入れ子構造になっている。また、migrate メソッドを含む関数(図 13 の callMig();)であるマイグレーションブロックの前後に追加された if-else 文は基本の変換操作の「MigFlag=false;」以外の字句が追加されている。さらに、マイグレーションブロックは必ず呼び出さなくてはいけないため、if 文内の処理にマイグレーションブロックは含まれていない。この図 13 によって複数の変換操作が実装され動作していることがわかる。また、参照変換後のコードに関しては動作確認を行い、エージェントコードが強マイグレーションを実現していることを確認した。



```
import java.io.Serializable;
public class T implements
Serializable {
 public void test(){
  int a=1;
  int b=5;
  a=2;
  a=3;
  if(a<b){
   a=4;
   migrate();
   b=10;
  }
  int c=3;
  callMig();
  b=15;
  c=6;
 }
 public void callMig(){
  int d=10;
  d=20;
  migrate();
  d=30;
  int e=1;
  e=2;
  migrate();
  e=3;
 }
}
```

```
import java.io.Serializable;
public class T implements Serializable {
 int saved_a=new Integer(0);
 int saved_b=new Integer(0);
 int saved_c=new Integer(0);
 int saved_d=new Integer(0);
 int saved_e=new Integer(0);
 Boolean[] MigFlag=new Boolean[6];
 public void test ( ) {
  int a = 1 ;
  int b = 5 ;
  int c = 3 ;
  if( MigFlag[ 1 ] == false ) {
   if( MigFlag [ 2 ] == false ) {
    if( MigFlag [ 0 ] == false ) {
     a = 2 ;
     a = 3 ;}
    if( MigFlag [ 0 ] == true || a < b ) {
     if( MigFlag [ 0 ] == false ) {
      a = 5 ;
      saved_a=a;
      saved_b=b;
      saved_c=c;
      migrate ( ) ;}
     else {
      a=saved_a;
      b=saved_b;
      c=saved_c;
      MigFlag [ 0 ] = false;}
     b = 10 ;}
    a=saved_a;
    b=saved_b;
    c=saved_c;
    a=saved_a;
    b=saved_b;
    c=saved_c;
    callMig ( ) ;
    b = 15 ;
    c = 6 ;}
   public void callMig ( ) {
    int d = 10 ;
    int e = 1 ;
    if( MigFlag [ 2 ] == false ) {
     if( MigFlag [ 1 ] == false ) {
      d = 20 ;
      saved_d=d;
      saved_e=e;
      migrate ( ) ;}
     else {
      d=saved_d;
      e=saved_e;
      MigFlag [ 1 ] = false;}
     d = 30 ;
     e = 2 ;
     saved_d=d;
     saved_e=e;
     migrate ( ) ;}
    else {
     d=saved_d;
     e=saved_e;
     MigFlag [ 2 ] = false;}
    e = 3 ;
}
```

図 13:サンプルコードの変換前（左）と変換後（右）

5.終わりに

本年度、ソースコード変換器を完成させたことにより、移動させたい位置に migrate 関数を書き込むだけで、強マイグレーションのモビリティを利用することが出来るようになった。今後は、migrate 関数をユーザーが記述するのではなく、変換器が自動で移動タイミングを判断し、migrate 命令を挿入する機能や、AgentSphere や他のエージェントからの移動要求に応じたエージェントの自律的な移動を実現し、モバイルエージェントシステムとしての性能向上を目指していく。

[参考文献]

[1] 疋田直也・鈴木 幸祐・甲斐 宗徳:「モバイルエージェントシステ AgentSphere の開発―強マイグレーションコードのための構文木を利用したソースコード変換器―」, FIT2012, B-011,Sep.2012

[2] 加藤史彬・田久保雅俊・櫻井康樹・甲斐宗徳:「コード変換による強マイグレーション化モバイルエージェントの実現」, FIT2007, B 分冊, pp.135-138, Sep.2007