B-024

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コードのための構文木を利用したソースコード変換器—
Development of Strong Migration Mobile Agent System AgentSphere
-Source Code Transformer Using Parse Tree for Strong Migration Agent Code-

疋田 直也† 鈴木 幸祐‡ 甲斐 宗徳‡
Naoya Hikida Kousuke Suzuki Munenori Kai

1 はじめに

複数のコンピュータが接続された状況において並列分散処理を行う場合、各コンピュータの処理能力や各コンピュータにおける負荷を考慮した分散を行う必要がある。しかし、これを実現するためには利用するコンピュータ郡の処理能力や負荷状態のリアルタイム情報が必要であり、それを把握して並列分散を制御するプログラムを実装することは困難である。その為、ソフトウェア自身が移動や処理などを自律的に行う並列分散処理システムが望まれる。そこで本研究では、自律的に移動や処理を行うモバイルエージェントを採用する。エージェントが移動を行う際に、移動前の実行状態を維持したまま移動することは強マイグレーション、保持せずに移動することは弱マイグレーションと呼ばれる。弱マイグレーションは比較的容易に実現が可能であり、継続実行の必要性が薄いプログラムに対しては有効である。たとえばテキストを編集するプログラムを例に考える。プログラムは編集の対象となるテキストファイルを読み込み、編集結果をテキストファイルとして出力する。出力されたテキストファイルを編集する際にも同様に、テキストファイルを読み込んだのちに編集を行い、出力を行う。このようなプログラムを実行する場合、テキストの読み込みから編集、出力という処理を毎回行うので実行状態を保持した移動によって処理の途中から再開をする必要は無く、弱マイグレーションが有効となる。しかし本研究では継続実行が望まれるエージェントの利用を考慮し、JVM で導入されている弱マイグレーションに加えて、強マイグレーションにも対応したモバイルエージェントシステム AgentSphere を開発している。強マイグレーションを実現する為には、実行コード、ヒープ領域内のデータ、スタック領域内のデータ、プログラムカウンタの保存と移動が必要になってくる。その中で、実行コードとヒープ領域内のデータに関しては、Java のシリアライズ機能を利用することで保存が可能である。しかし Java において、スタック領域のデータとプログラムカウンタに関しては実行状態として保存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データを保存する機能を実装する必要がある。既存の強マイグレーションモバイルエージェントシステムでは、JVM に対する変更を加えることや、バイトコード変換によって対応している。JVM に変更を加える理由は JVM が弱マイグレーションにしか対応していない点にある。しかし、JVM への変更を加える方式ではオリジナルの JVM との互換性が失われてしまう。そこで本研究では JVM への変更や独自の実行環境の導入なしでの強マイグレーション実現を目指す。そのために、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へと変換するソースコード変換を行う。

† ‡ 成蹊大学理工学研究科理工学専攻 Graduate School of Science and Technology, Seikei University

2 ソースコード変換方式

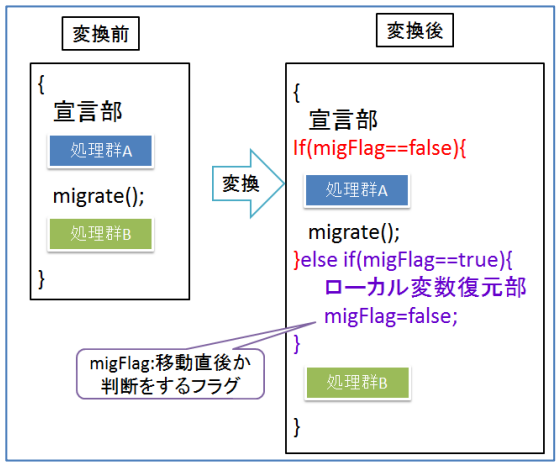

図 1 変換前後の if-else 文挿入の様子

ソースコード変換を行う目的はスタック領域内のデータの保存・復元とプログラムカウンタの取得である。

プログラムカウンタはプログラムの実行位置を知るために必要となる情報だが、移動前のエージェントのプログラムカウンタを移動先で利用することはできない。そのため、エージェントのソースコードが移動した直後から再開できるようにコードを変換する必要がある。そこで著者らは、エージェントの移動を migrate()という関数を記述することで行い、migrate()の位置を基準として if-else 文を挿入することで対応する。図 1 の様に if-else 文の挿入を行うと、まずプログラムは宣言部での処理を行う。次に migFlag は初期値として false で設定されているので if 文内部の処理群 A を行う。そして migrate 命令により移動を行うが、その際 migFlag は true がセットされる。移動後、プログラムはソースの先頭から行われるが、migFlag の値が true なので if 文における処理は飛ばされ、再開は migrate()の後の else 文から行われる。スタック領域のデータの保存は、スタック内の必要なデータをヒープ領域に保存するコードを生成することで対応を行う。これによりエージェント全体にシリアライズ機能を利用することができる。

```
Public class SampleCode implements Serializable{
Public void func(){
    変数宣言部
    Listを宣言して要素を追加
    要素をすべて出力
  migrate();
    要素をすべて足し合わせる
  System.out.println("要素の総和"+sum);
  }
}
```

図 2 変換対象となるコード

```
Public class Translated_SampleCode implements Serializable{
ArrayList<Integer>pre_testList=newArrayList<Integer>();
Boolean migFlag=false;
Public void func(){
    変数宣言部
  If(migFlag==false){
    Listを宣言して要素を追加
    要素をすべて出力
    Pre_testList=testList;
    migFlag=true;
    migrate();
  }else if(migFlag==true){
    testList=pre_testList;
    要素をすべて足し合わせる
    System.out.println("要素の総和"+sum);
    }
  }
}
```

図 3 変換後のコード

図 2 の様に記述されたコードを変換する場合、図 3 の様な形になる。図 3 から分かる通り、migrate による移動前に migrate の後の処理で足し合わせるための要素を保存した変数である list の中身を、メソッド外で宣言しておいた pre_testList へと移し替えることでヒープ領域へと移し、移動後に復元を行っていることがわかる。ソースコード変換機は以上のような変換処理を行う。

従来のソースコード変換器は二つの構文解析器を使用して処理を行っていた。一つはソースコードを構文解析してクラス名やメソッド名、ローカル変数、migrate の有無、スタック変数などを調べる解析器である。もう一つは最初の解析器で得られた情報をもとにソースコードを生成するための構文解析器である。

しかしこの方式では、ソースの内部情報を調べるためとソースを生成するために最低二回の構文解析を行わなくてはならない。加えて、情報取得やコード生成といった操作と構文解析が密にかかわっている為、操作の方式に変更がある際には操作部分だけの作り直しが困難であり、基本的に構文解析部分ごと作り直さなくてはならない。情報取得やコード生成以外の操作を追加する場合にも、その操作を行う為の構文解析器の作成をしなくてはならず、汎用性にも欠ける。その為、新たに構文木を利用したソースコード変換器の手法を提案する。

3　構文木を利用した変換器

構文木とは、構文解析を行った結果をデータ構造として保存したものであり、図 4 はコード中の　int a=5;　という記述を構文木化した例である。

構文木を構成するノードは、Block や Expression といった各構文規則に対応するノードクラスをあらかじめ作っておき、構文解析中に対応する構文規則があると、その規則に対してノードクラスからオブジェクトが生成されるようにする。このオブジェクトを集めたものが構文木である。各ノードは、自分の上下に存在するノードのアドレス情報を持っている。図 4 を例にすると、Block ノードは BlockStatement ノードと Block の上にあるノードのアドレス情報を持っており、LocalVariableDeclaration ノードは、Modifiers と Type と VariableDeclarator と BlockStatement のアドレス情報を持っている。コードを生成する際などは、この情報をたどることで構文木の捜査を行う。

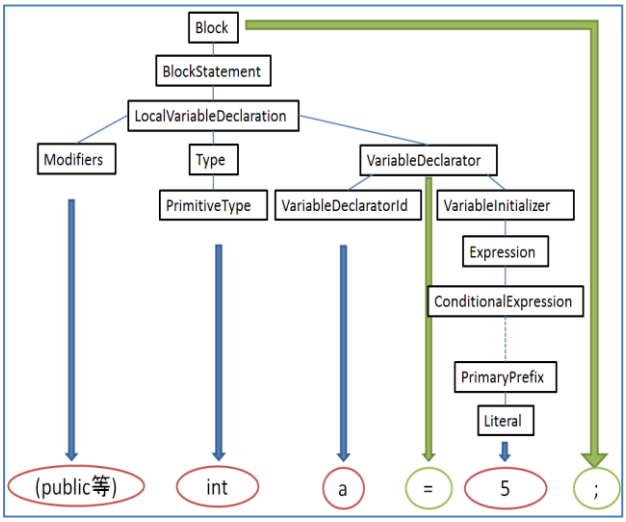

図 4 ソースコード中の int a=5;のコードを構文木として保存する例

加えて、PrimitiveType や Literal といった特定のノードには、情報が保存されている。図 4 を例にとると、PrimitiveType には文字列"int"が保存されており、VariableDeclaratorId には文字列"a"、Literal には文字列"5"が保存されている。今回 Modifiers ノードに情報は保存されていないが、public や new といった修飾子が使用されていた場合はこの中に使用された修飾子の文字列が保存される。

次に構文木を利用した操作の方法を見ていく。今回は、構文木からソースコードを生成する様子を例に進める。構文木に対する操作を行う場合、各ノードへ移動した際に行う処理について記述をしなくてはならない。このように、データ構造に対して操作を別に記述する方法は Visitor と呼ばれる。図 5 は"int a=5;"が構文解析されたノードクラスと、そのノードに対する処理を Visitor から抜粋した例である。図 5 の Visitor 処理を例に、LocalVariableDeclaration ノードからの挙動を見ていく。

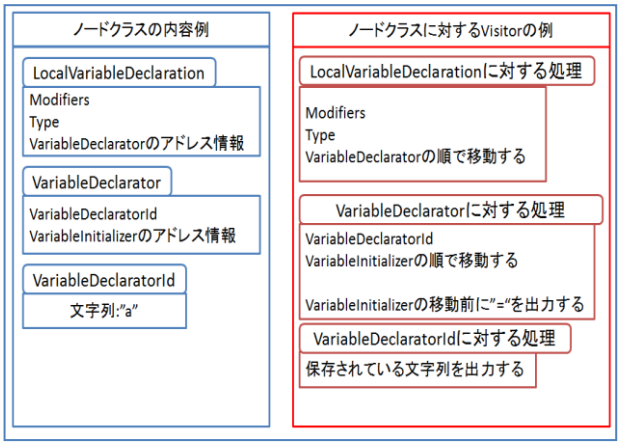

図 5 ノードクラスとそれに対応した処理の例

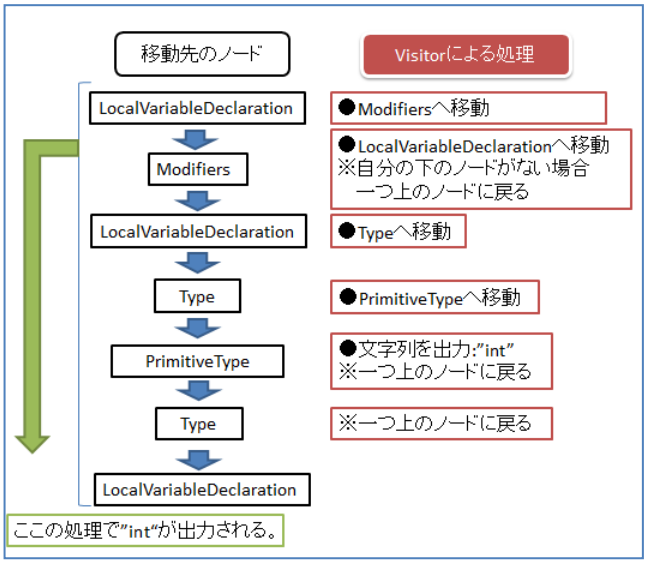

図 6 構文木から出力を行う流れの例の前半

まず、LocalVariableDeclaration に対する処理として、指定されたノードへの移動を行う。移動先ノードにおいて、本来であればノードに格納された修飾子文字列の出力命令があるが、今回修飾子は存在しないので、そのまま移動前のノードへと戻る。次に、Type ノードへ向かい、さらに PrimitiveType ノードへと進む。このノードでは格納されている情報の出力が行われるので"int"を出力する。自分の下のノードがない場合と処理がすべて終わった場合は自分の上のノードへと移動するので、Type、LocalVariableDeclaration の順に進む。次に、LocalVariableDeclaration ノードから VariableDeclarator に進む。自分の下のノードが二つあるが、処理による指定は VariableDeclaratorID への移動なので次のノードへ進む。移動先のノードに対する処理はこのノードに保存されている文字列の出力なので、保存された文字列"a"を出力する。このノードには自分の下のノードが存在しないので、移動前のノードへと戻る。

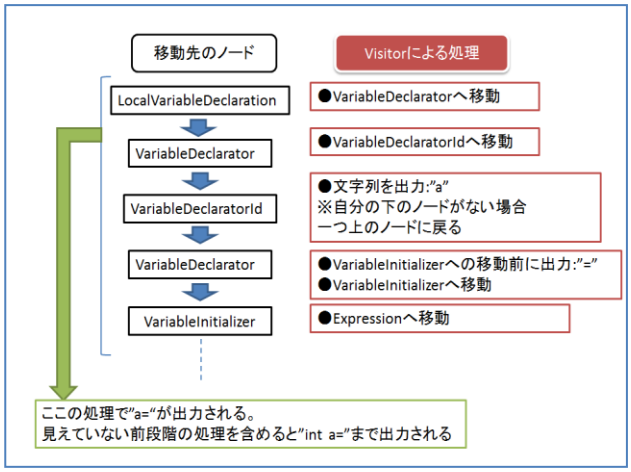

図 7 構文木から出力を行う流れの例の後半

再び VariableDeclarator ノード戻り、次のノードへと移動するが、処理では移動前に"="を出力することが指示されているので、"="を出力する。出力後は指示通りに VariableInitializer へと移動し、処理を続けていく。すべてのノードを捜査し終わった際に、各ノードで出力された結果をまとめることでソースコードが生成される。この出力用の Visitor を応用してソースコード変換を行う。

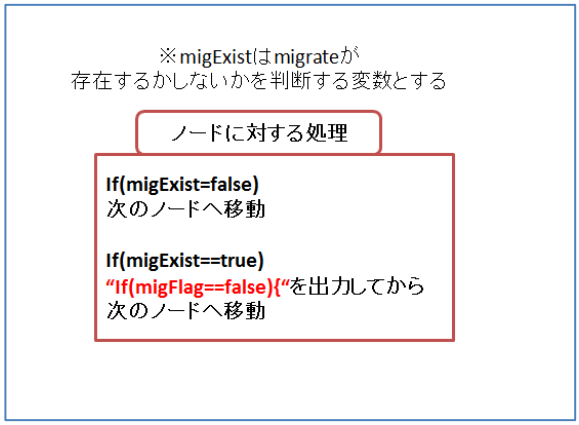

図 8 migrate の有無によって処理が変わる visitor の処理記述例

まず、構文木を生成するための構文解析を行う段階で、読み込んだソースコードに migrate 命令が含まれているかを調べておく。そして図 7 の様に、Visitor に migrate が存在する場合の処理と存在しない場合の処理を記述しておく。処理の流れとしては、図 8 で示すように、migrate が存在しない場合はそのまま出力を行う事で、読み込んだものと同じソースコードが生成できる。migrate が存在する場合は、図 9 のように Visitor に記述した migrate が存在する場合の処理を行う。

構文木自身に変換を加えることでソースコード変換を行う例を図 10 を用い、スタック変数の復元を例にとって説明する。この処理に必要となることは退避させる変数の型や名前であるが、その情報は migExist などと同じく構文木を取得する際の構文解析時に取得しておく。次に、図 3 の二行目で変数宣言がされているようにヒープ領域に退避さ

せるときに使う変数を宣言する必要がある。同様に図 3 から、migrate 関数による移動命令前の退避と移動後の復元が必要となる。そこで、以上の三つの処理を行うよう構文木を変換するためにノードを木に追加する。結果は図の変換後のようになる。この変換後構文木に対して先述の出力を行うことで if-else 分の挿入も行われ、強マイグレーションな振る舞いをするコードとして変換が行われる。

以上の方式によって、構文解析は構文木を取得するための一回で済ませることができる。出力などの操作に関しても、今までは構文解析と合わせて行っていたものが、Visitor 記述に変更されることで、新たな構文木に対する操作を追加する際も、構文解析を意識することなく追加が可能となった。

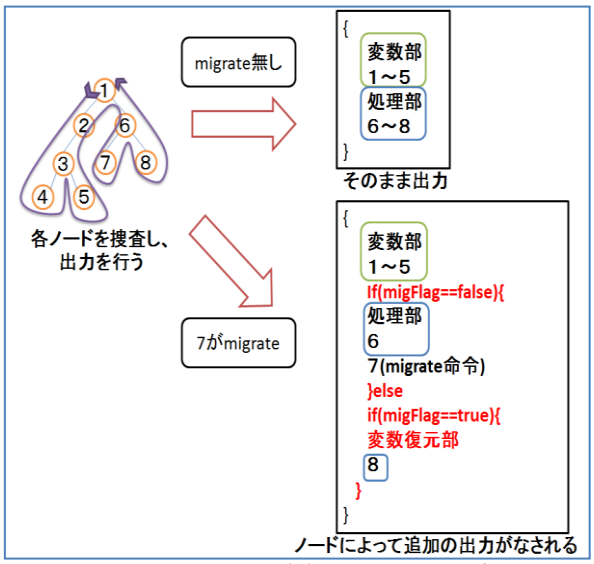

図 9 migrate の有無による挙動の違い

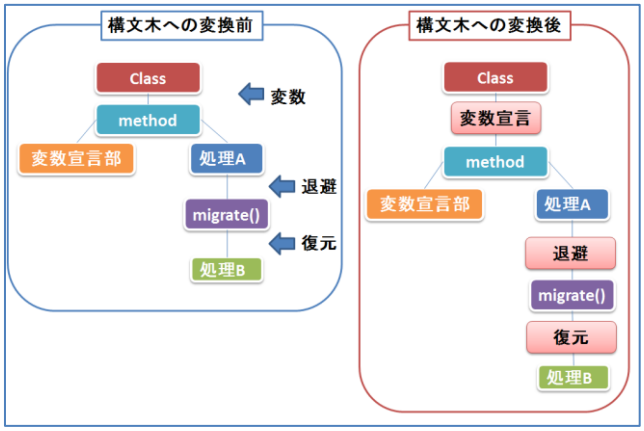

図 10 構文木への変換

4　終わりに

新たな変換方式への変更により、構文解析を行う回数が減り、ソースコードに対して行う操作の追加に関しても操作の記述のみをする方式へと変更をすることにより汎用性が上がった。今後、ソースコード変換における自動化を進めるために、現在コードの記述者が挿入している migrate 命令自体をノードとし、最適なタイミングで移動命令が呼び出されるような構文木内の場所へ自動挿入することを目標とする。

謝辞

本研究は科研費（基盤研究(C)21500041）の助成を受けたものであることをここに記し、謝意を表します。

[参考文献]

[1] 桜井康樹，“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”，成蹊大学大学院工学研究科情報処理専攻修士論文，2007

[2] 鈴木幸祐，“強マイグレーションの為のソースコード変換方式”，成蹊大学理工学部情報科学科卒業論文，2009

[3] 五月女健治，“JavaCC コンパイラ・コンパイラ for Java”，培風館，2003