成蹊大学理工学研究報告
J. Fac. Sci. Tech., Seikei Univ.
Vol.46 No.1 (2009) pp.25-34

強マイグレーション化モバイルエージェントの自己バックアップ機構の実装

加藤　史彬*[1]，甲斐　宗徳*[2]

Implementation of Self-backup Mechanism for Strong Migration Mobile Agents

Fumiaki KATOH*[1] , Munenori KAI*[2]

ABSTRACT: The authors have developed the platform of the autonomic distributed processing system, AgentSphere, which is based on strong migration mobile agent. In this paper, a new mechanism for making backup of an agent itself with all the data obtained by the process at the backup point is introduced into AgentSphere. Users can use this agent's self-backup mechanism by describing "*backup*" commands in an agent's code. The backed-up agent is sent into other AgentSphere machines via a network. When the original agent will be unfortunately stopped or lost by a contingency, the backed-up agent will wake up and be able to continue its process instead of the original agent. Thereby, the fault tolerance of the system can be improved. In addition, the function which inserts automatically the backup commands at the suitable locations in an agent code is proposed and implemented. Using this function make it easy for users to transform their application agents into the agents with the self-backup mechanism.

Keywords: mobile agent system, strong migration,self-backup

(Received March 23, 2009)

１．はじめに

一般的に分散処理システムでは，ユーザタスクを分散させる際に，動的に変化する局所的および全体的な負荷の状況把握や管理が必要となる。それは分散処理開始時のマシン性能に合わせたタスクの初期分散が処理終了まで適切であるとは限らないためであり，負荷によるマシンの処理能力の変化や参加マシン数の変化といった様々な状況を考慮したプログラムを記述する必要がある。そのためには分散処理およびシステムの構造に対する詳細な知識や経験が必要となり，プログラム記述者にとって大きな負担となる。

そこでシステムに参加しているマシンの状況の判断・制御を利用者が行わなくても，システム側で自律的にそれを行うようにすることで，プログラム記述者はメインの計算処理を行うコード作りに専念でき，誰でも手軽に分散処理の恩恵を受けられるようにすることが望ましい。本研究室では，このような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同様に弱マイグレーションのモビリティを利用していたため，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が困難であった。移動先での処理の継続が可能な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ためには，移動の際持ち運ぶ情報に必要なものとして，ヒープ領域内のデータ，実行コード，スタック領域内のローカル変数，プログラムカウンタがある。ヒープ領域内のデータ及び実行コードの移動は Java 言語に備わっているシリアライズ機能で容易に実現出来る。しかしスタック領域内のローカル変数とプログラムカウンタについては，Java 従来の機能では実現が出来なかった。そこでスタック領域内のローカル変数の取得に関しては，Java Platform Debugger Architecture(以降 JPDA)を利用して行うことにした[1]。

*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）

そしてソースコード変換を行うことにより，スタック領域内のローカル変数を取得・復元する処理を自動挿入し，プログラムの中断位置から処理を再開する構造にプログラムを自動変換する方式を実現した[2]。この従来の研究により，移動前の処理を無駄にすることなく移動先での処理の継続が可能な強マイグレーション化モバイルエージェントシステムのベースが完成した。

しかし移動前の処理を継続する命令において，取得・復元できるローカル変数はプリミティブ型とString型およびそれらの配列に限定されたもので，参照型については記述出来るものの，取得・復元は出来ない状態であった。このままでは，外部クラスを利用したプログラムが記述できず，様々な種類のプログラムを記述できるシステムとは言い難い。そのため本論文では，まず，参照型について取得・復元が出来るように移動命令の強化を行う。

また分散処理システムでは高信頼性や耐障害性が期待される。そこで，システムの全体な負荷を見ながらタスクの再負荷分散を行うようにすることで，応答の高速化のみならず，更なる高信頼性や耐障害性の向上を見込むことが出来る。これを実現するために，本研究では，エージェントが処理をある程度行ったときに，自分自身とそれまでの処理の途中結果をバックアップし，ネットワーク上でAgentSphereが動作する他のマシンに送り込む機能を加えることにした。そして例えばオリジナルのエージェントが不測の事態により停止したり失われたりしたら，バックアップされていたエージェントが活動を開始し，最新の途中結果を用いて処理を再開できるようにする。さらにエージェントのバックアップをどの時点で取らせるのが良いかを決めることはユーザにとって容易ではないので，エージェントのコードを解析して自動的に適切な位置に自己バックアップ命令を自動挿入する機能も提案する。

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，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。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。

モバイルエージェントシステムは大きく分けて弱マイグレーションシステムと強マイグレーションシステムに分けられる。両者は持ち運び出来る情報に違いがあり，弱マイグレーションシステムでは，移動の際持ち運ぶ情報が不十分なため，移動後には移動前の続きからの処理を行うことが出来ず，処理は最初からになってしまい，それまでの処理が無駄になってしまう。一方，強マイグレーションシステムでは移動後に移動前の続きからの処理を行うことが出来るが，実装が困難になる。

既存の研究では，AgentSpace（佐藤一郎氏）[3]やAglets（日本IBM社）[4]が挙げられるが，これらは弱マイグレーションシステムのため，先に述べた弱点がある。また既存の研究でも強マイグレーションシステムの研究は行われている。しかしJavaGO（東京大学米澤研究室）[5]では決められた位置でしか状態の保存・復旧が出来ず，また移動を実現するためにランタイムシステムを拡張したことで，移動プログラムは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作しないようになっているので，移植性が失われているという弱点がある。またMOBA（首藤一幸氏）[6]ではJava仮想マシン（以降JVM）の変更や，ネイティブメソッドの追加をすることで強マイグレーションを実現しているため独自のJVMを利用する必要があり，JVMのバージョンアップに伴い強マイグレーションのため変更したJVM を修正する必要があるためシステム開発者の負担が大きい，といった弱点がある。

2．2　自律分散処理システムAgentSphereの概要

当研究室で独自に開発を行っている自律分散処理システムAgentSphereではJVMの変更をすることなく，どのようなプログラムに対しても，ソースコード中の任意の位置に移動命令を記述するだけで，移動を行い，移動前からの処理を継続させることが出来る，強マイグレーション化モバイルエージェントシステムを実現している。

このことにより，当研究室の自律分散処理システムで目的としている「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が受けられるシステム」で，「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」の完成を目指している。

３．既存の移動命令に対する強化

3．1　移動命令 migrate の不足部分について

本研究で提案するのは，ユーザがエージェントに移動命令である migrate メソッドを記述するだけで，処理の中断，移動，再開を可能にすることである。処理の中断時には，移動した後の途中からの再開に必要な実行時データを取得している。従来の migrate メソッドでは，プリミティブ型，String 型そしてそれらの配列型の取得・復元が出来ていた。しかし，参照型の取得・復元が出来ておらず，クラスを利用したプログラムを記述できないという制限があったため，参照型の取得・復元について実装を行った。

3．2　ローカル変数の取得について

ローカル変数の取得の説明のために，初めに JVM のメモリ構造について説明する。JVM のメモリ構造は，図３．１で示すように，スレッド毎に割り当てられる PC レジスタと Java 仮想マシン・スタック，JVM 毎にひとつ割り当てられるヒープ領域に大別される。

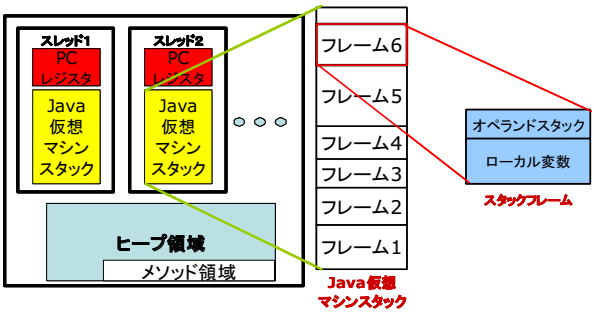

図 3．1　JVM のメモリ構造

JVM はスレッドという単位でアプリケーションにリソースを割り当てる。このスレッドが起動すると，JVM は自分のメモリ構造内に Java 仮想マシン・スタックを割り当てる。Java 仮想マシン・スタックは，C 言語のような従来からある言語におけるスタックに相当するものであり，後入れ先出しの待ち行列(Stack)で実現されている。スタック上のデータの単位をスタックフレームと呼び，そのフレームに対応するメソッドのローカル変数の配列や，オペランドスタックなどを保持する。

Java 言語を用いてモバイルエージェントを実装する際，Java 言語に備わっているシリアライズ機能を用いることにより，プログラムコードに加え，ヒープ領域内の情報を実行状態として保存して移動することが可能となっている。しかしシリアライズ機能は，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を提供していない。強マイグレーションシステムの実現には，プログラムコード，ヒープ領域内の情報に加え，これらの情報についても必要なため，スタック領域内の情報やプログラムカウンタ相当の情報の取得が必要となる。本研究では既に，スタック領域内の情報については，JVM に標準で実装されている JPDA を利用して，スタック領域内のローカル変数を取得し，またプログラムカウンタ相当の情報については，プログラムカウンタを使わなくてもソースコード変換を行うことによって，移動後に処理を中断した場所でまず移動前に取得したローカル変数の復元をし，実行を再開することが出来るようにした。

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，シリアライズ出来ず，マシン外部へ持ち出し出来ない。そこで，この値をシリアライズ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，別マシンに送信することができるようになる。

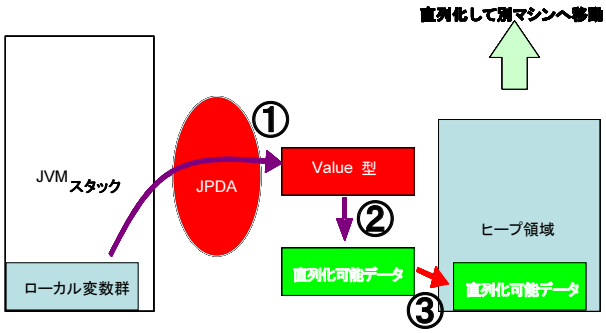

図 3．2　ローカル変数の取得のプロセス

図３．２にローカル変数の取得のプロセスについて示す。

スタックフレームは中身として複数のローカル変数を持っている。そのスタックフレームから各ローカル変数を取得したものが Value 型のデータである。これをシリアライズするまでの流れについて，int 型のローカル変

数の取得・復元を例にして，図３．２とソースコードを用いて説明する。

スタックフレームから情報を取得する際必要な情報として，スタックフレームそのもののデータ StackFrame とスタックフレームからアクセス出来るローカル変数についてのリスト localValues がある。

アクセス出来るローカル変数リスト localValues を引数とすることによって，ローカル変数群をスタックフレームから取得し，Map 型の変数 values に格納する（図３．３①-A）。各ローカル変数はローカル変数リストから Iterator で各ローカル変数の存在情報について取り出し，LocalVariable 型の変数 var として取得する（図３．３①-B）。そして各ローカル変数の存在情報である var を使いローカル変数群 values から，以下のコードのようにローカル変数として，Value 型の変数 value を取得する（図３．３①-C）。これが図３．２と図３．３の①である。

```
Map values = StackFrame.getValues(localValues);  A
for( Iterator j = localValues.iterator() ; j.hasNext() ;){
  LocalVariable var = (LocalVariable) j.next();   B
  Value value = (Value)values.get(var);           C
  /*各型ごとの動作*/
}                                                  ①
```

図 3．3　スタックフレームから Value 型を取得するまでのコード

```
if(value instanceof IntegerValue){
   int x = ((IntegerValue)value).value();   ←②
   /*ヒープ領域に格納*/
   localvalues.add(x);
   name.add(var.name());                    ←③
   type.add(var.typeName());
 }
```

図 3．4　int 型を取得しヒープ領域に格納するコード

そして value の型を instanceof で条件分岐し，各変数毎の動作を行う。まず value から int 型の値を取得する。これが図３．２と図３．４の②である。

そして Vector 型の localvalues，name，type にそれぞれ取得した値，名前，型を追加することでヒープ領域に格納している。これが図３．２と図３．４の③である。この Vector 型の localvalues，name，type は，ローカル変数取得後は，スタックフレームから取得した全ローカル変数について格納している。

参照型についても同様に取得するが，参照型はプリミティブ型などのように図３．２と図３．４の②で示した参照型を直列化できる形でリターンするメソッドがない。そのため，クラス変数の内部を走査していき，内部の変数一つ一つについて調べて行く必要がある。

図３．５はスタックフレームから取得したローカル変数が参照型の場合の動作を示している。まず図３．５の④で int 型同様に参照型を取得する。そして現在の参照型の型を取得し，参照型の変数 obj と obj の型，そして obj の名前を引数として orscan メソッドを呼び出す。

```
if(value  instanceof  ObjectReference){
  ObjectReference obj = (ObjectReference)values.get(var);   ←④
  RerefenceType refType = obj.referenceType();
  orscan(obj,refType,var.name());
   localvalues.add(objectList);
   name.add(var.name());                                    ←⑦
   type.add(var.typeName());
}
```

図 3．5　参照型を取得しヒープ領域に格納するコード

```
void orscan(ObjectReference obj, ReferenceType  refType, String s){
    for(Iterator i=refType.fields().iterator();i.hasNext();){
        if(PrimitiveValue){}
        else if(StringReference){}                          ←⑤
        else if(ObjectReference){                           ←⑥
        orscan((ObjectReference)obj.getValue(field),(ReferenceType)fi
ield.type(),s+"."+field.name());
        }
    }
}
```

図 3．6　orscan メソッドのコード

図３．６は参照型を確認したら呼び出される orscan メソッドのコードを示している。orscan メソッドは引数として，スタックフレームから取得したローカル変数 obj と型 refType，そしてクラスを特定するための名前 s を持っている。スタックフレームから取得したローカル変数 obj について Iterator で中身がなくなるまで参照する。そのとき型を判別し，ローカル変数の取得を行う。これが図３．６の⑤である。そして図３．６の⑥のように，参照型の変数を確認したら再び orscan メソッドを呼び出し，再帰的に取得して行くことで，クラスの内部にクラスがあっても取得を可能とした。

この際引数として，残りのローカル変数，型，そして名前を必要とするが，名前は元の引数の名前 String s に現在の参照型の名前をつけることで一意なものにすることが出来る。例えばクラス s1 の変数 i であれば s1.i，ク

```
orscan((ObjectReference)obj.getValue(field),
        (ReferenceType)field.type(), s + "." + field.name());
```

図 3．7　orscan メソッドの引数のコード

ラス s1 のフィールドとして存在するクラス s2 の変数 d であれば s1.s2.d というようになっている。

取得したクラス変数は Vector 型の配列 objectList に格納される。図３．８のような構造となっている。objectList はそれぞれ第 0 要素が変数の値，第 1 要素が変数の一意に判断できる名前，第 2 要素が変数の型となっている。

objectList[0]	objectList[1]	objectList[2]
値	名前	型
3	s1.i	int
2.54	s1.s2.d	double

図 3．8　取得したクラス変数の構造

そして全ての参照型の値を取得し終わったら，図３．５の⑦に書いてある通り他の型と同様，Vector 型の localvalues，name，type にそれぞれ取得した値である ObjectList，名前，型を追加することでヒープ領域に格納し，シリアライズ可能にしている。

3．3　参照型の復元について

復元については，getClassValue 及び buildClassValue が実行されることでクラス変数の復元が可能とな。この getClassValue 及び buildClassValue は migrate 用のコード変換時に自動挿入される。

参照型の復元は変数リストを元に行われる。あるクラス A 内にクラス B のオブジェクトがある場合，クラス A にはクラス変数内部を復元するための buildClassValue が挿入される。そしてクラス B には buildClassValue のメソッド本体が挿入される。これはクラス内部に参照型の変数がある限り再帰的に行われる。そして参照型以外の変数があった場合は，getClassValue で値を取得する。図３．９は図３．８のクラス変数の構造に従って getClassValue と buildClassValue の記述位置について示したものである。

SampleClass s1 では，自身の復元のため s1.buildClassValue();が挿入され，その呼び出しの中でフィールドである i と s2 の復元が必要である。i は参照型ではないので getClassValue で値を復元する。s2 は参照型なので，SampleClass s2 の復元のため s2.buildClassValue();が挿入される。

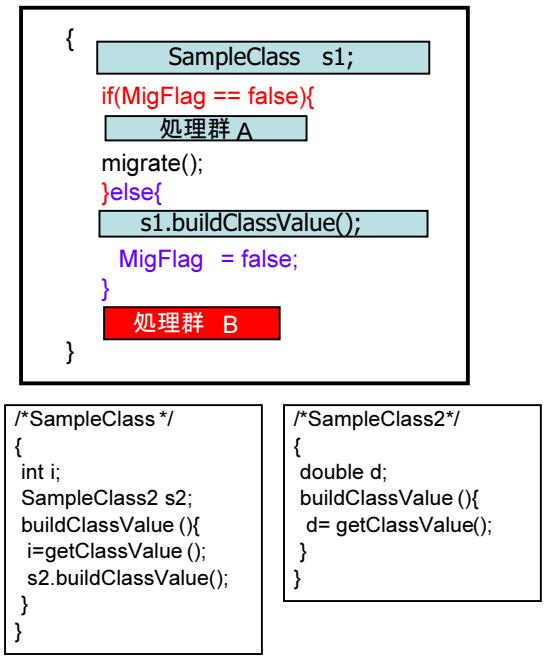

図 3．9　getClassValue と buildClassValue の記述位置

そして SampleClass2　s2 ではフィールドである d の復元が必要である。d は参照型ではないので getClassValue で値を復元する。

４．エージェントの自己バックアップ機構の実装

4．1　自己バックアップ機構についての概要

通常，分散処理では複数のマシンにタスクを分散させることで，高信頼性や耐障害性が期待される。しかし分散させて結果を統合するだけでは問題が発生する。それは各マシンがタスク分散時の性能で動作しているとは限らないため，初期分散で最適な負荷分散を行っていたとしても，その後のマシン性能の変化により最適ではなくなってしまう。この状態で処理を続けていけば処理時間が増加してしまうだけでなく，最悪負荷が掛かり続けたマシンがダウンしてしまい，それまでそのマシンで行っていた計算処理が無駄になってしまうことも考えられる。そこで処理をある程度行ったときにそれまでの実行データも含めてエージェント自身のバックアップを作成し，それを別マシンへ送り込んでおく。システム全体のマシン状況の監視をする Scheduler を用意することで，マシンの負荷が大きくなりそのマシンでエージェントが処理し続けるのが困難になった時や，万が一，元のエージェントが失われた時には，別マシンに送り込まれていたバックアップエージェントにバックアップ時点からの実行を再開させることが出来，高信頼性や耐障害性の向上を見込むことが出来る。

そこで高信頼性や耐障害性の向上を目的としたエージェントの自己バックアップ機構について提案する。エージェントの自己バックアップ機構が存在することで，実行データも含めたエージェントのバックアップを作成し，別マシンへ送り込むことにより，それらを利用することでバックアップ時点から処理の再開が可能となる。

今回はメソッドが呼び出された時点でのバックアップを取得する backup メソッド，そしてその backup メソッドを自動挿入する手法を提案することにより，強マイグレーション化エージェントの自己バックアップ機構の実装を行った。

4．2　backup メソッドについて

backup メソッドは呼び出された時点での実行時データをバックアップし，別マシンまたは自マシンへ送り込む。そしてそのバックアップデータを使うことによって，その時点からの再開を可能とする。

backup メソッドも後での実行再開のために，実行時データの取得を行わなければならない。また実行再開をするために，migrate メソッドと同様に図４．１に示すソースコード変換を行う。migrate メソッドと異なるのは，migrate メソッドでは移動直後にそのエージェントは実行を再開するが，backup メソッドは Scheduler などから呼び起こされて，処理を再開することになるという点のみである。エージェントの中で backup メソッドが複数回実行されることが出来，その都度，エージェントのバックアップが生成される。生成されたエージェントのバックアップが直前と同じ AgentSphere に保管されるときには上書きとなり，最新のバックアップが残ることになる。

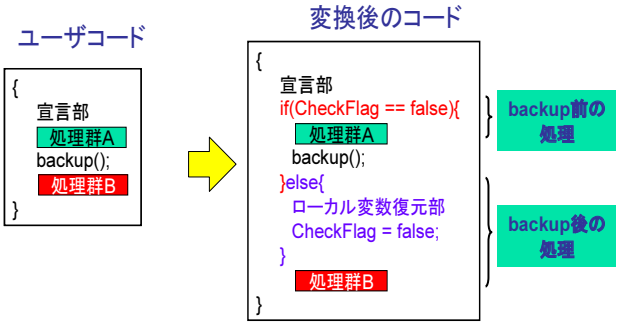

図 4．1　backup メソッドのソースコード変換

この backup メソッドは現段階では，backup メソッドの記述されたエージェントを管理する Scheduler が用意されていないため，5 章の評価では，手動で再開させるコマンドを利用してバックアップエージェントを再生させることにする。

今後，Scheduler を用意することで，負荷が大きくなり処理を続けるのが困難になったマシンにあるエージェントを停止させ，別マシンに送り込まれたバックアップから停止したエージェントの続きを実行したり，ダウンしてしまったマシンにあったエージェントを別マシンで再生させることなどが出来るようになる。

4．3　backup メソッドの自動挿入手法の提案

backup メソッドはユーザが任意に挿入することも出来る。しかしシステムで自動的により良い場所でのバックアップを取ってくれる backup メソッドの自動挿入機能があればユーザは自動挿入機能を利用するだけで，耐障害性を持つエージェントを利用できるようになる。そこでここでは backup メソッドの自動挿入手法の提案を行う。

今回提案する backup メソッドの自動挿入手法では，2 段階のフェーズに分けて backup メソッドの最適な位置を探すための分析を行っている。

まず第 1 フェーズでは静的分析でコードを走査し，backup メソッドの予約位置である reserved_backup メソッドを挿入する。ここでコードを走査するために，静的分析の実装にあたり JavaCC を利用した。JavaCC とは lex/yacc の Java 言語版で，サンマイクロシステムズが開発したオープンソフトである[7]。

そして第 2 フェーズでは動的分析で実際にコードを実行し，静的分析時に挿入された reserved_backup メソッドの中で，エージェントの開始からの時間を計り，その時点での各変数のアクセス頻度について調べる。そしてこれらの情報を利用して予約挿入されていた reserved_backup メソッドのうち，バックアップのために実際に使うものだけを適切に選び出して，backup メソッドに置換する。

以下では，静的分析と動的分析にそれぞれ詳述する。

4．3．1　静的分析

静的分析では，コードを走査し，backup メソッドの予約位置である reserved_backup メソッドを挿入する。reserved_backup メソッドは実際にバックアップを取るメソッドではなく，動的分析時に利用されるメソッドである。

エージェントをバックアップするオーバヘッドを出来るだけ少なくするにはバックアップの回数を減らし，また 1 回のバックアップに関わる実行時データを出来るだけ少なくする必要がある。スコープが深ければ深いほ

ど，スタックフレーム数が増え，取得する情報量が大きくなってしまい，それらをバックアップして持ち運ぶのは，オーバヘッドが大きくなる。そこでスコープから出た位置を reserved_backup メソッドの挿入位置とした。またメソッド呼び出しの直後にも reserved_backup メソッドは挿入される。それはメソッドが呼び出されると，新しいスコープに入り，そのメソッドが終了して戻ってくるためには，必ず１つのスコープを出るからである。

もしここで挿入された reserved_backup メソッドが多かったとしても，それらは次の動的分析時に絞り込まれる。

4. 3. 2　動的分析

動的分析では，静的分析が終わったコードを実際に実行し，reserved_backup がエージェントの動作開始からどのくらいの時間で実行されているのか，そして reserved_backup メソッドの時点で通算どのくらいの変数アクセスがあったのかについてのデータを取る。

変数のアクセスについては，各変数に対し，実際にエージェントを動作させ JDI を利用しどれだけアクセスがあったかを見ることが出来る。

reserved_backup メソッドはあくまで backup メソッドの予約位置であるので，動的分析後に reserved_backup メソッドの絞込みを行う。

今回は一例として，絞込みを行う条件を２つ考えた。一つ目の条件は，各 reserved_backup 間の時間で判断する。これは reserved_backup が短時間で複数回実行されていた場合は不要，と判断するものである。二つ目の条件として通算変数アクセス回数の差で判断するものであるが，実際に実装してみたところ，余りにもオーバヘッドが大きかったため，条件から外した。動的分析中に通った reserved_backup メソッドの位置で，どの reserved_backup メソッドを通ったか，その番号と通った時間をヒストリとして記録して行く。そしてそのヒストリから適切なタイミングで実行される reserved_backup メソッドを選び出すことが出来る。

従って，reserved_backup メソッドをどのくらいの間隔で残すべきかによって，生成されるバックアップエージェントの数は変わってくるが，それは今のところユーザがその間隔をパラメータとして指定すれば良いこととした。

５．評　価

ここでは今回行った実装について２つの動作確認を行う。表５．１は今回の評価に使用した Windows PC の一覧表である。

表 5．1　動作確認に使用したマシン一覧

マシン名	CPU	クロック数	メモリ
マシン A	Core 2 Duo	2.13GHz	2.0GB
マシン B	Pentium D	2.80GHz	1.0GB

5.1　クラス変数の取得・復元と backup メソッドについての動作確認

ここではクラス変数の取得・復元と backup メソッドについての動作確認を行う。そのため，backup メソッドを手動で入れたコードを用いた。下記コードはクラスを使ったエージェントで，利用クラスの内部にもクラスがある。そして別マシンへ backup メソッドで実行時データをバックアップし，別マシンでバックアップされたデータから再生することで，ローカル変数の取得・復元が出来ているかを確認した。

このコードはクラス s1 のフィールド i と j 及びクラス s1 のフィールドであるクラス s2 のフィールド d の値について，s1 のメソッド valincr()で増加させるプログラムで，増加させるたびに実行時データをバックアップし，別マシンへ送り込み，新しいバックアップが古いバックアップを上書きするものである。

図 5．1 は，backup メソッドが挿入された変換前のコードで，ユーザが記述するのはこれだけである。

この後図５．１のコードはソースコード変換器に掛け

```
public class BackupSampleTest{
 private String IPaddress = "133.220.114.108";
 private static AgentCoordinator ac;
 public void create (){
  SampleClass s1 = new SampleClass();
  ac = getAgentCoordinator();
  for (int i = 0;  i < 10; i++){
   System.out.println("backup前の値s1.i:" + s1.i + " s1.j:" +
s1.getj() + "  s1.s2.d:" + s1.s2.d);
   s1.valincr();
   ac.backup(IPaddress, 0);
   Thread.sleep(2000);
  }
  System.out.println("backup後の値s1.i:" + s1.i + " s1.j:" +
s1.getj() + " s1.s2.d:" + s1.s2.d);
 }
```

図 5．1　backup メソッドを挿入した変換前のコード

```
public class Translated_BackupSampleTest{
 private String IPaddress = "133.220.114.108";
 private static AgentCoordinator ac;
 public void create (){
  SampleClass s1 = new SampleClass();
  ac = getAgentCoordinator();
  if(ac.checkflag(0)==false){
   ac = getAgentCoordinator();
  }else{
   s1.buildClassValue(ac,0);
  }
  for (int i = 0;ac.checkflag(0)==true ||  i < 10; i++){
   if(ac.checkflag(0)== false){
    System.out.println("backup前の値s1.i:" + s1.i + " s1.j:" +
s1.getj() + "  s1.s2.d:" + s1.s2.d);
    s1.valincr();
    ac.backup(IPaddress, 0);
   }else{
    s1.buildClassValue(ac,0);
    i = ac.getIntegerValue("i",0);
    ac.setFlag(0);
   }
   Thread.sleep(2000);
  }
  System.out.println("backup後の値s1.i:" + s1.i + " s1.j:" +
s1.getj() + " s1.s2.d:" + s1.s2.d);
 }
}
```

図 5．2　backup メソッドを挿入した変換後のコード

```
public class SampleClass {
 int i;
 private double j;
 SampleClass2 s2;
 SampleClass(){
  i=1;  j=3.0;  s2=new SampleClass2();
 }

 /*クラス変数用の自動挿入メソッド*/
 public void buildClassValue(AgentCoordinator ac, int rank) {
  i=(Integer)ac.getClassValue("s1.i", 0);
  j=(Double)ac.getClassValue("s1.j", 0);
  s2.buildClassValue(ac,0);
 }

 double getj(){ return j; }

 public void valincr() {
 i++;  j++;  s2.d++;
 }
}
```

```
public class SampleClass2 {
 double d;

 SampleClass2(){
  d = 13;
 }
 public void buildClassValue(AgentCoordinator ac, int rank) {
  d = (Double)ac.getClassValue("s1.s2.d",0);
 }
}
```

図 5．3　テストで利用するクラスのコード

```
問題 Javadoc 宣言 コンソール JavaCC console
<終了> SphereManager [Java アプリケーション] C:¥Program Files¥Java¥jre6¥bin¥javaw.exe
[AgentSphere]> load Translated_BackupSampleTest.class
load Translated_BackupSampleTest.class
LoadAgent : Translated_BackupSampleTest.class
OK
[AgentSphere]> backup前の値s1.i:1 s1.j:3.0  s1.s2.d:13.0
dispatch: Address 133.220.114.108 port 60000
dispatch: Address 133.220.114.240 port 60000
backup前の値s1.i:2 s1.j:4.0  s1.s2.d:14.0
dispatch: Address 133.220.114.108 port 60000
dispatch: Address 133.220.114.240 port 60000
backup前の値s1.i:3 s1.j:5.0  s1.s2.d:15.0
dispatch: Address 133.220.114.108 port 60000
dispatch: Address 133.220.114.240 port 60000
backup前の値s1.i:4 s1.j:6.0  s1.s2.d:16.0
dispatch: Address 133.220.114.108 port 60000
dispatch: Address 133.220.114.240 port 60000
backup前の値s1.i:5 s1.j:7.0  s1.s2.d:17.0
dispatch: Address 133.220.114.108 port 60000
dispatch: Address 133.220.114.240 port 60000
backup前の値s1.i:6 s1.j:8.0  s1.s2.d:18.0
dispatch: Address 133.220.114.108 port 60000
dispatch: Address 133.220.114.240 port 60000
```

エージェントの停止

図 5．4　マシン A の実行結果

```
問題 Javadoc 宣言 コンソール JavaCC console
新規構成 [Java アプリケーション] C:¥Program Files¥Java¥jre1.6.0_05¥bin¥javaw.exe
[AgentSphere]> bc Translated_BackupSampleTest
> bc Translated_BackupSampleTest
backup calling
[AgentSphere]> backup前の値s1.i:7 s1.j:9.0  s1.s2.d:19.0
dispatch: Address 133.220.114.108 port 60000
dispatch: Address 133.220.114.108 port 60000
backup前の値s1.i:8 s1.j:10.0  s1.s2.d:20.0
dispatch: Address 133.220.114.108 port 60000
dispatch: Address 133.220.114.108 port 60000
backup前の値s1.i:9 s1.j:11.0  s1.s2.d:21.0
dispatch: Address 133.220.114.108 port 60000
dispatch: Address 133.220.114.108 port 60000
backup前の値s1.i:10 s1.j:12.0  s1.s2.d:22.0
dispatch: Address 133.220.114.108 port 60000
dispatch: Address 133.220.114.108 port 60000
backup後の値s1.i:11 s1.j:13.0 s1.s2.d:23.0
エージェント終了！
```

図 5．5　マシン B の実行結果

られる。変換後のコードを図５．２に示す。移動命令（ここでは backup）に関する変換を斜体で示し，さらに backup 用固有の変換を下線部で示してある。また図５．３に図５．１と図５．２で使用しているクラスの変換後のコードを示す。変換された部分は下線で示してある。

以上のコードを AgentSphere を動作させているマシン A で実行し，i が 6 になったところで AgentSphere を停止し，エージェントを終了させた。そしてここではバックアップされたエージェントが起動後に処理を再開出来るのかを確認するため，マシン B で AgentSphere のバックアップの再生コマンドである bc コマンドを利用し，マシン A での続きから実行されることを調べた。その結果を図５．４，図５．５に示す。マシン B で再開したところ，ループインデックスである i の値及びオブジェクト s1 の値が引き継がれており，マシン A での続きがマシン B で行われ，クラス変数について取得・復元できていることが分かる。

5．2　backup メソッドの自動挿入手法の正当性の評価

backup メソッドの自動挿入がどのように行われているかを確認するため，巡回セールスマン問題（以後 TSP）の全探索での解法のコードに対し自動挿入を行った。その結果，静的分析によって図５．６に示されているように reserved_backup メソッドが挿入された。各 reserved_backup メソッドは自分と他を見分けるため引数としてシーケンシャルに数値を持っている。

```
public void create()
 start_re ();
 //Map生成
 for(i=0;i<NUM;i++){
  find(MAP[0][0],1,i);
  reserved_backup(1)
 }
 reserved_backup(2)
}
```

```
void find(int sum,int depth,int place){
 if(sum >
 reserved_backup(3)
 check[place] = true;
 for(int i = 1 ; i < NUM ; i++){
  if(check[i] == false){
 find(sum+MAP[place][i],depth+1,i);
 reserved_backup(4)
  }
  reserved_backup(5)
 }
 reserved_backup(6)
 if(depth == NUM- 1){
  sum += MAP[place][0];
  if(sum < point){
   point = sum;
  }
  reserved_backup(7)
 }
 reserved_backup(8)
 check[place] = false;
}
```

図５．６　静的分析後のコード

更に動的分析を行った結果，２～８の reserved_backup メソッドは絞り込まれ，１の reserved_backup メソッドのみが残った。３～８の位置の reserved_backup は再帰メソッドである find の再帰呼び出しが終了する際に短時間で実行される位置であり，バックアップとしては適さないものばかり，と考えられる。しかし１の reserved_backup は再帰が終了し，戻ってきてスタックフレームが減っている場所であり，大量の処理が終わったと考えられる。これは論理的に見ても TSP で１つの経路が得られた直後にバックアップ作業を行うことに相当するので，適切なバックアップであると考えられる。

またこの分析後の TSP について backup メソッドが機能していることの確認を行った。まず初めに 12 都市のマップを入力として TSP の処理を行うエージェントを停止させずに最初から最後まで行った。その結果約５分の時間が掛かった。次に再び TSP を動作させ，約４分の時点でエージェントを停止させた。そして bc コマンドで再生したところ，残りの約１分で最短経路の結果を出力した。これは backup メソッドでエージェントのバックアップが取り，その続きからの実行が出来ていて，バックアップが有効であることを示している。

６．結　論

6．1　まとめ

本研究では，まず最初に，強マイグレーションの移動命令 migrate の機能の改良を行った。以前の AgentSphere では，エージェントが migrate 命令によって移動する際は，プリミティブ型の変数と String 型のオブジェクトしか保存して移動することが出来なかった。本研究で行ったコード変換手法の改良により，これまで不可能であったクラスオブジェクトも取得して移動し，移動先で復元することが可能となった。これでユーザはどのようなコードでもエージェントとして記述出来るようになり，様々な目的の処理を行うプログラムを実行可能なシステムになったと言える。

さらに本研究では，エージェントに自己バックアップ機構を実装した。そして手動で自己バックアップ命令である backup 命令をエージェントに記述し，その動作を確認した。これにより，エージェントは実行中のマシンで不測の事態により停止した場合でも，他のマシンに送り込まれた自身のバックアップを動作させることにより，停止時に最も近い状態から，処理を再開・継続することができるようになり，エージェントシステムとしての高信頼性・耐障害性の向上を図ることが出来た。また，この backup 命令は，エージェントを記述する人が手動で適切な場所に入れることができるものであるが，既存のエージェントのコードに backup 命令を自動挿入する手法も提案した。これは，エージェントのコードを静的分析と動的分析の２フェーズに分けて解析することにより，人が適切と考えるコード内の位置，すなわち，それまでの処理の途中結果を極力無駄にしないで済み，かつバックアップの際のエージェントサイズを大きくしすぎて多大なオーバヘッドとならないような位置を自動的に見つけ出し，backup 命令を挿入するものである。この自動挿入機能を確認した TSP 解法のサンプルコードでは，適切と思える位置に挿入することができたが，他の色々なコードでも適切な位置に挿入できるようにするためには，まだ調整が必要であると考えている。

6．2　今後の展望

backup メソッドが及び backup メソッドの自動挿入手法を実現したことにより，AgentSphere システムの負荷管理を行う Scheduler を用意することで，初期分散だけ

でなく再負荷分散を行うことで，システムが自律的に動作するシステムを構築することが考えられる。

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] 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

[3] 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

[4] 日本 IBM 東京基礎研究所：http://www.trl.ibm.com/aglets/, Jan.2009 参照可

[5] 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.2009 参照可

[6] 首藤一幸: http://www.shudo.net/moba/, Jan.2009 参照可

[7] JavaCC：https://javacc.dev.java.net/, Jan.2009 参照可