成蹊大学理工学研究報告
J. Fac. Sci.Tech., Seikei Univ.
Vol.51 No.1 (2014)pp.43-48

AgentSphereにおけるマルチスレッド型エージェントの強マイグレーション機構

疋田　直也[*1]，甲斐　宗徳[*2]

The Strong Migration Mechanism for Multi-thread Agents in AgentSphere

Naoya HIKIDA[*1], Munenori KAI[*2]

ABSTRACT：In AgentSphere, in addition to weak migration, strong migration is also adopted as the move manner of the agent. Although the weak migration which saves the information in a heap area and moves to another JVM by using a serialization function is realizable in Java, the function which saves the information in a stack area and program counter is not supported. Therefore, it is necessary to implement the function to save these two important information by a certain means. In the conventional AgentSphere, although strong migration of a single thread agent was implemented using the method of source code conversion, it was not implemented that the multithreaded agents move all together by a strong migration manner. Moreover, reducing the source code conversion overhead as much as possible was required. So, in this paper, in order to reduce the conversion overhead and to implement the strong migration of multithreaded agents, a new strong migration manner which used Javaflow is proposed.

Keywords：mobile agent system, parallel distributed processing, strong migration, program conversion

(Received March 31, 2014)

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向けのプラットフォームとなるAgentSphereの開発を行っている。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するエージェントを利用している。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ている。

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いない。そのため，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。

従来，AgentSphereでは，ソースコード変換の手法を用いて強マイグレーションを実現しているが，シングルスレッドエージェントのみに対応しており，マルチスレッドを利用したエージェントは強マイグレーションすることができていない。また，ソースコード変換はAgentSphereを利用するすべてのエージェントコードに対して実行されるため，変換処理で発生するオーバーヘッドを少しでも抑えることが求められる。

そこで本発表では，オーバーヘッドの発生を軽減し，かつ，マルチスレッドエージェントに対応した新たな強マイグレーション方式について提案を行う。

[*1]：理工学研究科理工学専攻修士学生
[*2]：理工学研究科理工学専攻教授（kai@st.seikei.ac.jp）

2．AgentSphereにおける強マイグレーション

2．1　強マイグレーションの実現方法

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には，JavaGO[1]などが存在する。これらのシステムでは，Java仮想マシン(JVM)に対する変更や，ランタイムシステムの拡張やネイティブメソッドの追加を行うことによって強マイグレーションを実現している。

しかしAgentSphereでは，一般的なJavaコードの実行環境上で稼働する事を目指しており，ほかの関連研究のように特別な実行環境を用意する方法を考えていない。

そこでAgentSphereでは特別な実行環境を用いない方法として，ソースコード変換器を用いた強マイグレーションの実現を進めてきた。

2．2　ソースコード変換器

ソースコード変換器では，変換対象である強マイグレーション記述されたエージェントのソースコードを，強マイグレーションコードと同様の動作をする弱マイグレーションコードへと変換することによってJavaでの強マイグレーションを実現する。

変換内容は図 2.2.1 の様になる。図 2.2.1 左図は変換前のエージェントコードを表している。コードの記述者は，エージェントが実際に行う振る舞いに加え，移動を実行させたい任意のタイミングに強マイグレーション命令を記述しておく。

ソースコード変換器は，変換対象となるソースコード中の強マイグレーション命令の位置を基にしてソースコード変換を行う。図 2.2.1 左図のコードを変換した結果は図 2.2.1 右図の様になる。

変換後のコードを実行すると，はじめに宣言部の処理を行い，if文の処理に移る。if文の条件判別式として使用されているMigFlagの初期値はfalseであるため，そのまま処理群Aを実行する。ローカル変数の保存は，ヒープ領域内にスタック領域のデータを保存する為の変数を設け，退避させることによって行う。また，移動命令の直前でMigFlagの値をtrueへと変更する。次に，移動命令である強マイグレーション命令が実行され，エージェントは他のマシンへと移動を行う。

移動後 のエージェントはソースコード先頭から処理を再開するが，移動前にMigFlagの値がtrueへと変更されている為，if文内の処理は実行されずelse文以降の処理が行われ，すべての命令を実行した後，終了する。

このように強マイグレーション命令の前後をif文で分割する事で移動後のエージェントが移動前の状態から処理を再開する事が可能となり，JVMに変更を加えずに強マイグレーションな振る舞いを実現する事が可能となる。

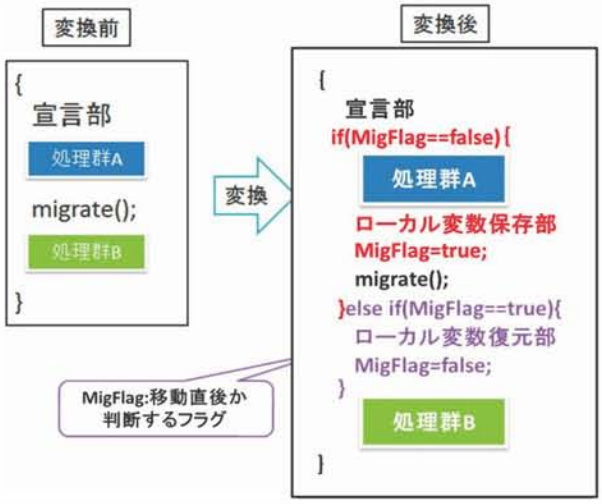

図 2.2.1　ソースコード変換前後の様子

2．3　従来方式の問題点

ソースコード変換器を用いることによりJVM上で強マイグレーションを実現する事は可能であるが，大きなオーバーヘッドの発生や，他のソースコードに依存関係がある場合のソースコードの変換，マルチスレッドで実行されるエージェントの移動への対応に関して問題が残る。

大きなオーバーヘッドが発生する原因は，ソースコードの変換方法にある。ソースコードの変換操作では，変換対象となるソースコードに対して構文解析を行った後，解析結果をデータ構造として保存し，保存したデータ構造に対してソースコード変換に操作を加える。そして，変換が加えられたデータ構造からソースコードの出力を行う事によって，変換後のソースコードを取得することが出来る。この変換操作を実行する時間がそのままオーバーヘッドとなる。

また，エージェントは複数のソースコードのファイルから構成されることが多いが，ソースコード変換を行う際は，エージェントを構成するファイル中で，migrateメソッドを含むすべてのファイルに対してソースコード変換を行う必要がある。現状，変換器にかけられるソースコードのファイルが，変換処理を行う必要があるものか否かをすぐ判別する事は出来ない。その為，変換対象となるエージェントコードを構成するすべてのソースファイルに対して構文解析を行い，変換が必要か判断しなくてはならない。加えて，エージェントを構成するソースファイルの数は膨大な数になることが考えられ，エージ

ェントを構成する全てのソースファイルに対して変換操作を行うことは効率的とは言えず，より良い方法を考える必要がある。

また，AgentSphereのようにJVMに変更を加えない方式では，スレッドの外部から実行状態（強制的にスタック領域の情報及びプログラムカウンタの値）を取得する事が出来ない。そのため，あるスレッドによってmigrate()メソッドのような移動命令が呼び出された時，呼び出し元以外のスレッドも強制的に移動させることが難しく，マルチスレッド記述されたエージェントをほかのマシンへと移動させることが困難である。

3．新たな強マイグレーション方式

3．1　既存の問題に対する解決方法の提案

ソースコード変換器を用いた強マイグレーションの実現が抱える課題の解決方法として，Java言語のバイトコードを変換することによる強マイグレーションの実現を提案する。今回，バイトコード変換による強マイグレーションを実現するにあたり，Javaflow[2]を利用する方法を考える。

3．2　Javaflowの概要

Javaflowは，オープンソースのソフトウェアプロジェクトを支援する団体であるApacheソフトウェア財団傘下のトッププロジェクトであるApache Commonsで開発，提供が行われているJavaコンポーネントである。

Javaflowは，Java言語で「実行状態の継続」を実現する事を目的としており，「実行状態の継続」を実現するために必要となるスタック領域内の情報やプログラムカウンタを含めた実行状態の保存・復元を実現するために，プログラムのバイトコードの変換を行う。

Javaflowを用いる事により，プログラムのある時点における実行状態をContinuationと呼ばれるデータ型に保存し，後で保存した状態から実行を再開する事が出来ることが可能となる。図 3.2.1 はJavaflowを用いたクラスのサンプルコードである。このクラスの実行は，クラス実行用のメソッドを呼び出して行う（startWithメソッド）。呼び出されたクラスはrunメソッド内の処理を実行する。runメソッド内に中断命令が存在している場合，命令が呼び出された時点での実行状態をContinuationというデータとして保存し，クラス実行用のメソッドの戻り値として返す。保存されたContinuationはシリアライズ可能なデータであり，別のマシンへの転送を容易に行うことができる。保存された実行状態の再開は，実行状態再開用のメソッドを呼び出すことで可能となる。また，Javaflow自身はすべてJava言語を用いて実装されており，標準的なJVM上での動作が可能となっている。

```
Class MyClass() implements Runnable  {
  public void run(){
    処理1;
    Continuation.suspend();    //中断
    処理2;
  }
};

//MyClass#run()を実行開始
//処理1を実行した時点で中断し、実行状態を保存
Continuation c = Continuation.startWith(new MyClass());
C = Continuation.continueWith(c);  //処理2から実行を再開
```

図 3.2.1　Continuationの利用例

以上のように，Javaflowを用いる事によって，AgentSphereが求める「JVMに変更を加えることなく，かつ，簡単に利用できる強マイグレーション」が実現可能となる。

3．3　新方式の利点

ソースコード変換器を用いた方式では，強マイグレーション利用したいコードに対して構文解析や解析結果のデータに対する変換操作など，様々なコード変換処理を行わなければならない。その結果，オーバーヘッドが増加するという問題が発生している。対してJavaflowを利用した方式では，構文解析などの処理を行う必要がないので，コード変換処理を行うのにかかっていたオーバーヘッドと変換方式の効率面での大幅な改善が見込める。

また今までの方式では，マルチスレッドで実行されるエージェントの移動に対応できていないという問題が存在した。この点に関しても，実行状態の中断・再開が容易であるというJavaflowの特性を用いることで，対応が可能である（6 章参照）。

4．新たな強マイグレーション方式の実装

4．1　新たな強マイグレーション方式の概要

Javaflowを利用してエージェントを生成するには，強マイグレーションコードの記述とバイトコード変換を行う必要がある。

4．2　強マイグレーションコードの記述

前述した通りJavaflowを用いることで，プログラムの実行状態の中断や再開が容易に実現できる。しかし，強マイグレーションを実現するため，中断した実行状態を移動させ，移動先で再開させる処理を記述しなくてはならず，エージェントコード記述者にとって負担となる。

そこでAgentSphereではJavaflowを利用し，処理の中断・移動・再開をすべて行う強マイグレーション命令を用意した。図 4.2.1 は強マイグレーションコードの利用例であり，ソースコード変換器利用時と同様に，エージェントコード中の強マイグレーションを実行したいタイミングに命令を記述することで利用が可能となっている。

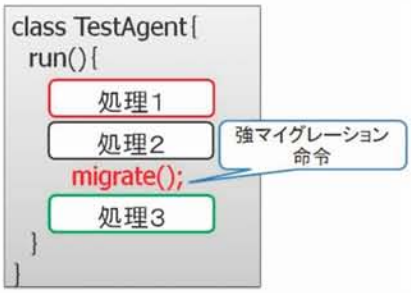

図 4.2.1　強マイグレーションコードの利用例

4. 3　バイトコード変換

続いて，ユーザによって記述された強マイグレーションコードをコンパイルし，Classファイルを生成する。そして，生成されたClassファイルに対してバイトコード変換を行う。バイトコード変換には，OSなどの環境に依存しにくいビルドツールであるApache Antを用いる。Apache Antでは，XML文書でビルド時のルールをタスクという形で記述することが可能であり，Javaflowにおけるバイトコード変換は，バイトコード変換用のタスクを用意することで行っている。

4. 4　新たな強マイグレーション方式の問題点

以上のように，強マイグレーションコードの記述とバイトコード変換を行うことにより，処理の中断・再開が可能となるが，このまま単純にJavaflowを利用しただけでは保存した実行状態を別マシン上で再開することができない。この問題は，Javaが標準に搭載しているクラスローダでは，別マシンから転送されてきた未知のクラスを実行することができないということに起因する。しかしAgentSphereでは，転送されてきた未知のクラスを実行するため，階層型クラスローダ[3]を開発し，採用している。階層型クラスローダを用いることで，保存した実行状態を別のマシン上で再開させることが可能となる。

4. 5　課題

以上のように，Javaflowを用いることで強マイグレーションの実現が可能となる。しかし，Javaflowを用いた強マイグレーションはマルチスレッドプログラムの移動には対応していないため，何らかの対応を行う必要がある。具体的な対応方法は 6 章で説明する。

5．評価

5. 1　評価の概要

評価は，Javaflow 利用の新方式とソースコード変換器利用の旧方式の二つを比較して行う。比較は，各方式を採用したエージェントが「実行までの事前処理にかかる時間」と「実行後の処理にかかる時間」を計測した結果を用いて行う。

計測は二台のPC間で強マイグレーションを繰り返すことで行った。マシンの性能はそれぞれ，CPU:i5-2400CPU 3.1GHz・メモリ:8GBのものと，CPU:i5-2500CPU 3.3GHz・メモリ:4GBのものを使用した。計測対象として，四則演算など簡単な処理を行う 100 行程度・500 行程度・1000 行程度の 3 つのサイズのエージェントコードを用意した。また，各サイズのコードには，強マイグレーション命令の呼び出し回数が 2・4・6・8・10 回の種類を用意し，違いを確認する。

5. 2　エージェント実行の事前処理にかかる時間の比較

今回比較を行う各方式では，エージェントを実行する際，事前処理を行う必要がある。そこでJavaflow利用方式におけるバイトコード変換とソースコード変換器利用方式におけるソースコード変換の各事前処理にかかる時間を計測し，比較を行った。図 5.2.1 は計測結果である。

図中の上の表にあるソースコード変換器利用時の結果を見ると，より行数の長いコードを変換すると変換にかかる時間が増加する事が見て取れる。一方，図中の下の表にあるJavaflow利用時の結果を見ると，コードの行数によって変換時間は大きく変わらず，処理に要する時間もソースコード変換器を利用した場合よりも短いことが分かる。また，両方式に共通している事として，ソースコード中の強マイグレーション命令数の大小では，事前処理にかかる時間に大きな差が出ないことが分かる。
結果を比較すると，Javaflowを用いた方式のほうが短時間で事前処理を終了することが分かる。特に行数の多いコードに対して事前処理を行う際に，差が明確に表れた。

ソースコード変換器利用時		コード中の強マイグレーション命令数				
	行数	2	4	6	8	10
サンプルコードの行数	100行	2640.2	2640.2	2640.2	2640.2	2640.2
	500行	3666.7	3730.2	3513.6	3366.5	4080.2
	1000行	4798	4763.4	4364.5	4933.2	5103.3

Javaflow利用時		コード中の強マイグレーション命令数				
	行数	2	4	6	8	10
サンプルコードの行数	100行	577.3	589.6	581.4	579.2	598.2
	500行	581.5	589.9	577.3	578.1	596.6
	1000行	587.7	583.3	578.2	598.7	601.4

（単位：ms）

図 5.2.1　各方式の事前処理にかかる時間

5. 3　エージェント実行時の処理にかかる時間の比較

ここでは，実際に各方式で強マイグレーションを行うエージェントが処理を完了するまでの違いを調べる。図5.3.1は計測結果である。今回計測した結果では，両方式共に1000行程度のサンプルコードの実行時間が500行の実行時間よりも短いという結果が出た。この点については，原因が不明であるため，参考記録として扱う。

図中の上の表にあるソースコード変換器の結果を見ると，エージェントコードの行数に比例して実行時間が増加する事が分かる。また，同じ行数であっても，強マイグレーション命令の呼び出された回数が多いほど処理の終了までにかかる時間が多くなることが分かる。

図中の下の表にあるJavaflowを見ると，ソースコード変換器利用時と同様に，行数の増加と強マイグレーション命令の呼び出し数に比例して実行時間が増加する事が分かる。

結果を比較すると，行数の短いコードを実行する際，両方式における実行時間の差は見受けられなかった。しかし，エージェントコードの行数，及び強マイグレーション命令の呼び出し回数が増加するにつれ，両方式の実行時間の差が大きくなる。その為，行数が多いか別マシンへの移動機会が多いエージェントにはJavaflowを用いたほうが短時間で処理を終了できると考えられる。

ソースコード変換器利用時		コード中の強マイグレーション命令数				
	行数	2	4	6	8	10
サンプルコードの行数	100行	145.5	228.3	319.1	441.5	563.5
	500行	255.2	423.6	616.5	914.5	1339.2
	1000行	310.5	393.1	501.5	740	899.2

Javaflow利用時		コード中の強マイグレーション命令数				
	行数	2	4	6	8	10
サンプルコードの行数	100行	143.2	223.8	326.2	415.2	567.1
	500行	222.4	359.6	536	795	1096.4
	1000行	157.7	168.4	169.2	171.3	180.8

（ 単位：ms ）

図 5.3.1　各方式における実行時間

5. 4　結論

AgentSphereでは，長時間実行し続けるエージェントや何度も移動を繰り返すエージェントの利用も目指している。その為，今回の計測において，移動回数が多いエージェントコードを利用する際により優秀な結果を残し，また，事前処理の時間も短いJavaflowを利用する事が最適だと考えられる。

6．マルチスレッド型エージェントへの対応

6. 1　マルチスレッド型エージェントとは

マルチスレッド型エージェントとは，マルチスレッドプログラムをAgentSphere上で実行させるためのエージェントの形式である。本形式では，マルチスレッドとして動作させたい各処理をエージェントコードとして記述し，各エージェントを協調させることでマルチスレッドな動作を実現させる。

6. 2　マルチスレッド型エージェント実現の方法

AgentSphereではマルチスレッドな動作の実現を，エージェントに「実行・確認・停止」という３つの状態を用意することで行う。このうち，「実行」は自身が行う仕事を実行している状態であり，「停止」は仕事を中断し，移動を待機している状態である。そして「確認」状態のエージェントは，自分以外のエージェントの状態を確認する状態である。その際，「停止」状態のエージェントが一つでも存在していることを確認した場合，自身も「停止」の状態へと遷移する。また，「停止」状態にあるエージェントが存在しない場合，「実行」の状態へと遷移し，自身の仕事を再開する。エージェントはこれらの状態を遷移し，その状態に応じた処理を行う。

最終的に，すべてのエージェントが「停止」状態となったとき，すべてのエージェントの実行状態を別のマシンへと転送し，処理を再開させる。この手順を繰り返すことでマルチスレッド型エージェントの強マイグレーションの実現を行う。

6. 3　本研究における実装

前項で説明した三つの状態を用意するため，新たに状態確認命令を用意した。エージェントは実行直後，「実行」状態にあり，状態確認命令が呼び出されると「確認」，強マイグレーション命令が呼び出されると「停止」状態へと遷移を行う。

また，マルチスレッド利用したい各処理をエージェントとして生成し，別々のスレッド上での初期起動を行う，メインメソッドの代わりとなるエージェント（メインのエージェント）を用意した。メインのエージェントは，マルチスレッドな動作を行うすべてのエージェントが「停止」状態になったとき，各エージェントの実行状態を持って移動と，移動先で中断処理の再開を行う。

マルチスレッド型エージェントを利用する際，初めにメインのエージェントが生成される。次にメインのエージェントはマルチスレッドな振る舞いをさせたい各処理をエージェントとして生成し，別々のスレッド上で起動させ，自身は移動を待機する状態へと移る。別々のスレッド上で実行された各エージェントは，「実行」状態で処理をはじめ，最終的に停止状態へと遷移する。すべての

エージェントが停止状態になると，移動待機状態になっていたメインのエージェントは，停止状態の各エージェントの実行状態を持ち，別マシン上へと移動を行う。移動後，メインのエージェントは停止状態の各エージェントの実行状態を再開させ，自身は移動待機状態へと移る。

上記で説明した各種動作を行う為に必要な機能を実装し，マルチスレッド型エージェントへの対応を行った。

6．4　実行結果

今回，マルチスレッド型エージェントのサンプルとして，A・B・Cの 3 処理をマルチスレッドで同時実行するエージェントを用意した。A・B・Cの処理内容はそれぞれ，ループを利用して 1 から順に数を出力するものとなっている。各処理にはあるタイミングで「確認」または「停止」状態へと遷移する為の命令が挿入されており，そのタイミングは各処理によって違う（例：処理Aは出力する数が 5 の倍数の時「確認」へと遷移し，処理Bは出力する数が 3 の倍数の時「確認」，10 の倍数の時「停止」に遷移する等）。サンプルのエージェントを実行するマシンには，5 章で評価を行う為に用いたものと同じ 2 台のマシンA・マシンBを使用した。

図 6.4.1 は，マシンA上でマルチスレッド型エージェントを起動した結果である。図中の拡大図における「Multi A」や「Multi B」といった記述は，それぞれ処理Aや処理Bの出力結果を表している。図からA・B・Cの処理は同時に動いており，マルチスレッドな動作を行っていることが分かる。図中の下にある「Migrate」という記述は，3 つの処理がすべて停止状態へと遷移したため別のマシンへと移動したことを知らせるものとなっている。

図 6.4.2 はマシンAから移動してきたエージェントがマシンB上で処理の再開を行った結果である。図中の「Continue」という記述は，別マシンからエージェントが移動してきて処理を再開したことを知らせるものとなっている。図中の拡大図における結果を見ると，エージェントの出力している値は 0 から始まるのではなく，移動前の出力の続きから行われている様子が見て取れる。また移動前と同様に，A・B・Cそれぞれの処理が同時に動いていることが分かる。マシンBにおいても各処理は出力の途中で「確認」や「停止」といった状態間を遷移し，すべての処理が停止の状態へと遷移すると図中下部の様に「Migration」の記述を残して別のマシンへと移動する。

以上の様に，今回の実装によってマルチスレッド型エージェントの実現が可能となったことが分かる。

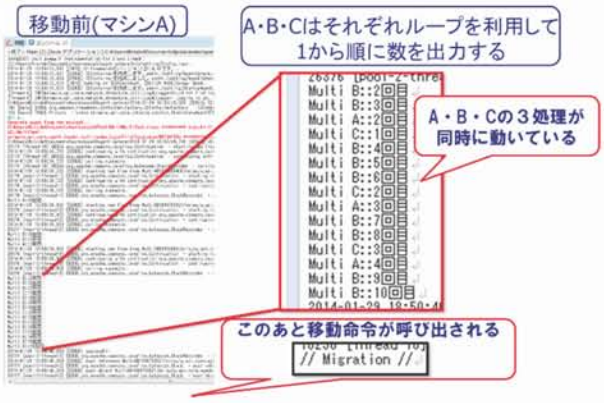

図 6.4.1　マシンAにおける動作の様子

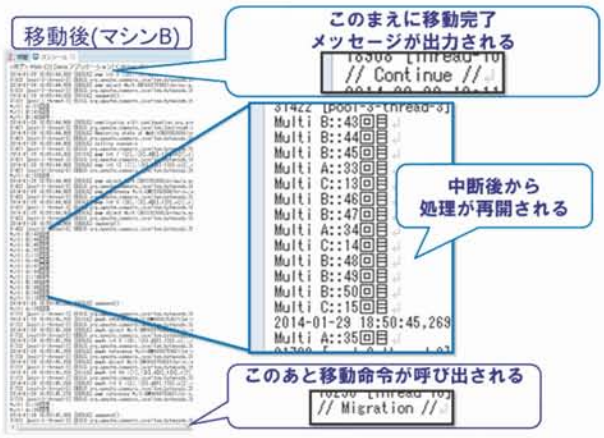

図 6.4.2　マシンBにおける動作の様子

7．終わりに

今回，強マイグレーションの方式を変更し，今まで未対応だったマルチスレッド型エージェントへの対応を進めた。今後は，今回行った研究を利用して様々なエージェントを作成・利用し，問題点の洗い出しと改善を行っていく必要がある。

参考文献

1. 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://web.yl.is.s.utokyo.ac.jp/amo/JavaGo/doc/, Apr.2006
2. ApacheProject.Javaflow. http://commons.apache.org/sandbox/javaflow/i.
3. 赤井雄樹・山口大祐・甲斐宗徳 「JavaVM 上での非手続オブジェクト転送を可能とする直列化方式の構築」 FIT2011, B-032, 2011