平成15年度 修士論文

既存JAVAプログラムを透過的に活用できるモバイルエージェントシステムの設計と実装

電気通信大学 大学院情報システム学研究科

情報システム設計学専攻

0250008 小野 優介

指導教官 多田 好克 助教授
星 守 教授
大森 匡 助教授

提出日 平成16年1月30日

目 次

第 1 章　背景と目的 **8**

1.1 モバイルエージェントの利点 9

1.2 モバイルエージェントの課題 10

1.3 目的 12

第 2 章　既存システムの問題点 **13**

2.1 既存プログラムの再利用性の問題 13

2.1.1 移動処理の記述の問題 14

2.1.2 実行状態保存の問題 14

2.2 JavaVM を利用した既存システム 16

2.2.1 Voyager 16

2.2.2 Aglets 17

2.2.3 JavaGO 17

2.2.4 MOBA 18

2.2.5 既存システムの比較と考察 19

2.3 本研究の提案 19

第 3 章　システム概要 **20**

3.1 設計方針 20

3.2 本システムの概要 21

3.3 passport の概要 22

3.4 AIR–port の概要 23

3.4.1 コード変換 24

第 4 章　移動ルール定義言語 passport **25**

4.1 移動ルール 25

4.2 言語の特徴 26

4.3 データ構造 27

4.3.1 リスト 27

4.4 集合 28

4.5 言語構造 29

4.5.1 FIELD 29

4.5.2 標準関数 30

4.5.3 flag 型変数 33

4.5.4 特殊なシンボル 33

4.6 移動経路の作成 33

4.6.1 AREA の作成 34

4.6.2 PLACE の順序付け 34

4.7 移動スケジュール 35

4.7.1 移動タイミング 36

4.7.2 移動スケジュール作成方法 37

4.8 環境依存処理の定義 38

4.8.1 route–table 38

4.8.2 移動時の処理 38

4.8.3 依存関係処理 38

4.9 例外処理 39

第 5 章　AIR–port **41**

5.1 AIR–port の構成 41

5.2 トランスレータによるエージェントの生成 43

5.2.1 エージェントの構造 43

5.2.2 クラス名の衝突回避のための名前変換 44
5.3 エージェント実行・制御機構 44
5.4 実行状態保存と復旧 45
5.4.1 状態の保存と復旧のメカニズム 46
5.4.2 コンテキストクラス 48
5.4.3 スタックフレームクラス 48
5.4.4 実行状態保存・復旧のための変換 50
5.5 移動処理の実行・制御 54
5.5.1 移動ルールクラス 54
5.5.2 移動処理実行の流れ 55

第 6 章 評価 57

6.1 利用例 57
6.1.1 考察 57
6.2 評価実験 58
6.2.1 考察 59
6.3 システムの評価 61

第 7 章 今後の課題 63

7.1 処理速度の改善 63
7.2 マルチスレッドへの対応 63
7.3 マルチエージェントへの対応 64

第 8 章 まとめ 65

付録 A 標準関数と特殊シンボル一覧 69

付録 B 利用例のサンプルコード 72

付録 C　評価実験に用いた Java コードの変換 75

図 目 次

2.1 マイグレーションの分離 (論文 [8] より引用) 16

3.1 エージェント化 22
3.2 AIR–port システムの概念図 23
3.3 コード変換によるエージェント生成 24

4.1 リストと集合 28
4.2 AREA の作成方法 34
4.3 移動経路作成方法 35
4.4 flag の関連付け 37
4.5 スケジュールの作成方法 37
4.6 route–table の使用方法 39
4.7 with 関数と without 関数 39
4.8 excepiton-table 制御関数 40

5.1 AIR–port の構成 41
5.2 全体の構成 42
5.3 トランスレータによる変換 44
5.4 エージェント実行・制御機構 46
5.5 状態保存のメカニズム 47
5.6 コンテキストクラス 48
5.7 保存クラスの生成原理 49
5.8 ローカル変数の置き換え 49
5.9 save メソッドの挿入箇所 52
5.10 基本的な変換 53

5.11 引数を持つメソッド 54
5.12 if 文の変換 55
5.13 移動処理の流れ 56

B.1 サンプルプログラム (passport コード) 72
B.2 サンプルプログラム (Java コード) 73
B.3 サンプルプログラム 実行結果 74

C.1 プログラム 1:変換前 75
C.2 プログラム 1:変換後 76
C.3 プログラム 2:変換前 77
C.4 プログラム 2:変換後 78

表 目 次

2.1 既存システムの比較 18

4.1 FIELD 29
4.2 集合の表現 31
4.3 集合の演算 32

6.1 実験環境 59
6.2 プログラム 1 59
6.3 プログラム 2 60
6.4 ファイル書き込みを制限した場合の測定結果 61

A.1 標準関数一覧 69
A.2 特殊シンボル一覧 71

第 1 章

背景と目的

モバイルエージェントは近年の大規模分散環境において次世代分散技術として注目されている。近年、情報化社会はネットワーク技術の発達やインターネットの普及に伴い著しい発達を遂げた。また、動的に変化する複雑なネットワークが構成され、そのようなネットワーク上に分散する情報も爆発的に増加してきた。このような複雑なネットワーク構成や、分散化した情報を持つ大規模分散環境において、ネットワークアプリケーションの開発や運用の負担が増加している。モバイルエージェントはこのような問題を解決するために考案された新しいネットワークアプリケーションの概念である。

モバイルエージェントとは実行状態を保存しながらコンピュータ間を自律的に移動するプログラムを意味する。この特徴は、ネットワークの断線などによる予期しない停止が想定される分散環境でのネットワークアプリケーションに有効である。たとえば、大規模な分散環境を利用したシミュレーションや、分散環境上のコンピュータリソース管理などの分野で注目されている。そのため近年、モバイルエージェント技術の研究は盛んに行われており、Aglets [1] や Voyager [2] など実用化されたモバイルエージェントシステムも数多く存在する。

しかし、モバイルエージェントは移動という新しいプログラミング概念を扱っているため多くの課題が残されている。以下、本章ではモバイルエージェントの利点と課題を議論し、本研究の目的を述べる。

1.1 モバイルエージェントの利点

モバイルエージェントの利点は、複雑化と大規模化が進む分散環境においてネットワークアプリケーション開発や運用の負担を軽減できることにある。動的に変化する近年の分散環境ではネットワークの断線を考慮しなくてはいけない。このような環境では、ネットワークアプリケーションの通信回数は可能な限り少なく実装したい。モバイルエージェントの最大の利点は、従来の通信処理をエージェントの移動処理に置き換えることにより通信回数を減らせることにある。モバイルエージェントは対象のコンピュータに移動すると、従来の分散アプリケーションでは通信処理として行っていた処理をコンピュータ内でのローカル処理として実行する。そして、その実行結果を元のコンピュータに報告するために移動する。つまり最も単純なクライアントサーバタイプの通信処理であれば、この 2 度の通信のみで要求を満たすことができる。

また、従来の通信処理をローカル処理として扱うことができるので、システムの分析・設計がより単純にできる。従来のネットワークアプリケーションでは、ネットワークの断線など、考慮しなくてはいけないことが多くあった。モバイルエージェントシステムでは、対象のコンピュータへの移動後は一般的なローカル処理プログラムとして扱うことができる。

実行状態を保存しながらコンピュータ間を移動するという特徴には以下のような利点がある。第一の利点は、耐故障性に優れたプログラムを容易に組むことができることである。シミュレーションなど実行が長時間におよぶプログラムではコンピュータの故障によって実行が途中でとまってしまうことは避けなくてはいけない。モバイルエージェントは実行環境に負荷やトラフィックが集中した場合、別コンピュータに移動し実行を途中から再開することができる。また、ネットワークが断線してしまっても、実行状態をデータ化してファイルに保存することにより、プログラムを退避することができる。第二の利点は、負荷分散性に優れたプログラ

ムを開発者が意識することなく組むことができることである。分散環境において 1 台のコンピュータに処理が集中することは非効率である。この場合、モバイルエージェントを処理するためのソフトウェア (以降、ランタイムシステムと呼ぶ) にホストのリソース監視・管理機構を備えることにより、エージェントを別のホストへ自律的に移動することができる。

1.2 モバイルエージェントの課題

モバイルエージェントは 1.1 節で述べたような利点を持つ一方で、解決されていないいくつかの課題を残している。その要因の 1 つは、実行状態を保存・復旧するための機能の実現が難しいためである。これはモバイルエージェントが新しい概念であるためと考えられる。既存モバイルエージェントシステムはプラットフォームに非依存な実行環境を持たせるために、JavaVM を利用する場合と、独自のランタイムシステムを利用する場合が多い。そのため、開発言語も Java 言語が用いられる場合と専用の記述言語が用いられる場合とに分かれる。しかし、どちらの場合も移動という新しい概念を表現するための課題がある。

まず JavaVM を用いた場合の課題について述べる。JavaVM を用いた場合以下の 3 つの課題がある。第一に、その記述手法があげられる。開発言語として Java 言語を利用しているシステムが多いが、Java 言語では、エージェントの移動処理と移動先のコンピュータ上で実行される処理 (以降、ローカル処理と呼ぶ) を分離することがとても難しい [7]。そのため、その記法がエージェントに特化してしまう傾向が強い。たとえば独自の文法拡張や文法上に制約が設けられていたり、独自のエージェント用ライブラリを用いたりするなど、Java 言語の記述力を低下させてしまう。結果的に既存プログラムをモバイルエージェントへ拡張 (以降、エージェント化と呼ぶ) させたい場合の修正が膨大になる。つまり、既存プログラム資産を活用することができない。また移動に関する処理をプログラム中に埋め込む

ため、移動経路や実行環境に依存した処理など、移動に関する処理をローカル処理から切り離して記述することが難しい。そのため、既存プログラムの再利用性の観点で問題がある [3]。第二に、Java 言語にはエージェントの移動という通信概念が無いことがあげられる。つまり、ホスト名や IP アドレスなどによる通信が前提とされている Java 言語で、プログラム自身が状態の変化に応じて行き先を選択する動作の表現は難しい。第三に、JavaVM は実行の安全性を重視した実装になっていることがあげられる。プログラムカウンタや内部スタックなどの実行状態にプログラムから変更を加えることができないよう実装されている。そのため、エージェントの実行中に実行状態を保存して、他ホストへ移動させるための機構の実現が困難である。

次に独自のモバイルエージェントランタイムシステムを使用した場合の課題を述べる。独自のランタイムシステムを用いた場合、その開発言語も独自のエージェント専用言語が用いられることが多い。専用言語を用いた場合、移動処理とローカル処理の分離が容易になり移動に伴う処理の記述力も一般的に高い。また独自の実行状態保存・復旧機構を備えることによって、プログラムカウンタや内部スタックなどの実行状態を保存してプログラムを移動させることが可能になる。しかしながら一般的なプログラミング言語とは異なる独自の文法を用いるため、開発者にとっての言語の習得負担が大きくなる。また、既存プログラムをエージェント化するにはその言語によってプログラムを新しく書き直さなくてはいけない。さらにランタイムシステムの汎用性についても問題がある。JavaVM を利用した場合よりも、OS に非依存な環境を用意することが難しい。そのためランタイムシステムが OS 依存になってしまうことが多い。これでは、多様な分散環境に対応したエージェントシステムは構築できない。

1.3 目的

本研究の目的は、既存 Java プログラムの資産を開発者に負担かけることなく有効に活用できるモバイルエージェントシステムを開発することである。1.1 節で述べたようにモバイルエージェント技術は、近年の複雑なネットワーク構成を有する分散環境でのネットワークアプリケーション開発の負担を軽減するための有望な技術の 1 つである。それにも関わらず 1.2 節で述べた理由から、現在モバイルエージェント技術はそれほど広く普及していない。その理由は、既存プログラム資産の利用が困難であることが大きい。また、既存モバイルエージェントシステムでは、この問題を重要視したシステムは少ないのである。本研究では、この点に焦点を当てたモバイルエージェントシステムを設計・実装を行う。

第 2 章

既存システムの問題点

ここでは JavaVM を利用した既存モバイルエージェントシステムの具体的な問題点について述べる。既存モバイルエージェントシステムでは、JavaVM 上で移動という振る舞いを実現するため様々な工夫がされている。しかしそれと引き換えに、Java の記述力を低下させたり、また既存プログラムをエージェントへ拡張する場合の修正が大きくなるなどの短所がある。

以下では、まずこの原因となっている問題について述べる。次に既存モバイルエージェントシステムをいくつか例にあげ、その性質を比較する。そして、これを踏まえて本研究で提案するこれらの問題の解決方法を述べる。

2.1 既存プログラムの再利用性の問題

既存プログラムのモバイルエージェントへの再利用性を低下させている根本的な問題は、ローカル処理コードと移動処理コードを切り離せないことにある。なぜなら、実行状態保存処理や移動処理のようなモバイルエージェント拡張に必要な処理は、プログラム全体に関わる処理であるため、プログラム内に直接埋め込まなくてはいけないからである。また JavaVM ではプログラムカウンタや内部スタック情報を保存・復旧するための機能が備わっていない。そのため状態を保存・復旧するためのコードも開発者が明示的にプログラム内にうめこまなくてはいけない。このことはローカル処理を熟知していない者による既存プログラムのエー

ジェント化を困難にする。

以下では、移動処理の記述の問題と実行状態保存の問題について論じる。

2.1.1 移動処理の記述の問題

Java 言語による移動処理の表現は記述力が低い。たとえば、移動の際に依存するファイルを全て一緒に移動するといった、移動処理を表現するための記述方法は用意されていない。そのため、JavaVM を利用した既存モバイルエージェントシステムでは、独自のライブラリを用意したり、エージェント記述用フレームワークを作成するといった方法を取っている。しかし、このような Java 言語の拡張を行っているもののうち、既存プログラムのエージェント化を考慮した設計をしているシステムは少なく、エージェント化を行う場合の修正が大きい。また、このような Java 言語の拡張は、Java 言語の自由度や記述力自体を低下させてしまうこともある。

2.1.2 実行状態保存の問題

JavaVM でモバイルエージェントシステムを実装する場合、移動処理の記述処理コードをエージェントプログラムから切り離すことをできにくくしている要因の 1 つは、実行状態保存である。これは JavaVM には実行状態保存機能が部分的にしか備わっていないためである。Java VM ではインスタンス変数のデータをシリアライズし、プログラムの再実行時にデシリアライズしてインスタンス変数を復旧する機能が備わっている。しかしローカル変数やスタック内の情報を保存することはできず、より完全に実行状態を保存するには、既存プログラムに修正を加えなくてはいけない。そのため、既存モバイルエージェントシステムではローカル処理の記述に様々な制約を設け、移動処理の実行位置を限定して実行状態を保存させることが多い。

このように、JavaVM を利用したモバイルエージェントシステムでは、実行状態

をどこまで完全に保存できるかということが重要になる。このような実行状態保存の度合いによって、エージェントの移動は一般的に、コードマイグレーション、ウィークマイグレーション、ストロングマイグレーションの 3 つに分類される。以下にその分類方法を述べる。

コードマイグレーション: プログラムコードのみが移動する。移動先コンピュータでは実行位置もデータも全て決まった初期値から実行される。Java Applet などがこれに当たる。

ウィークマイグレーション: プログラムコードに加え、インスタンス変数などのヒープ領域内の情報を実行状態として保存し移動する。Java 言語で実装されるモバイルエージェントシステムでは、ウィークマイグレーションの実現にはシリアライズ機構を利用することが多い。シリアライズ機構とはインスタンス変数データを直列化し、保存し復旧するための機構である。この機構は JDK1.1 より Java の仕様に追加された。シリアライズ機構によるウィークマイグレーションを行う既存システムの例として、Voyager や Aglets などがあげられる。ウィークマイグレーションによるエージェント開発では、開発者が保存したいデータをインスタンス変数として定義しなくてはいけない。

ストロングマイグレーション: 実行状態を完全な形で保存し移動する。ストロングマイグレーションでは図 2.1 で示されるようなプログラム中の全ての状態を保存し復旧することが可能である。しかし JavaVM 上での実現は難しくマルチ・スレッドや、ユーザインターフェースなどの保存は、ストロングマイグレーションに含まれないことが多い。また実行が再開される位置は固定されている場合が多く、ネットワーク断線時や負荷の集中が起きた場合の動的な移動ができない。ストロングマイグレーションを行う既存システムの例として、Plangent [10] や MOBA (Mobile Agent Facilities on Java Language Environment) などがあげられる。

ウィークマイグレーションとストロングマイグレーションの最大の違いは、ウィークマイグレーションはインスタンス変数のみの保存になるので、開発者は保存と復旧を意識して、ローカル処理を記述しなくてはいけないことにある。そのため既存プログラムを、そのまま活用することはできない。

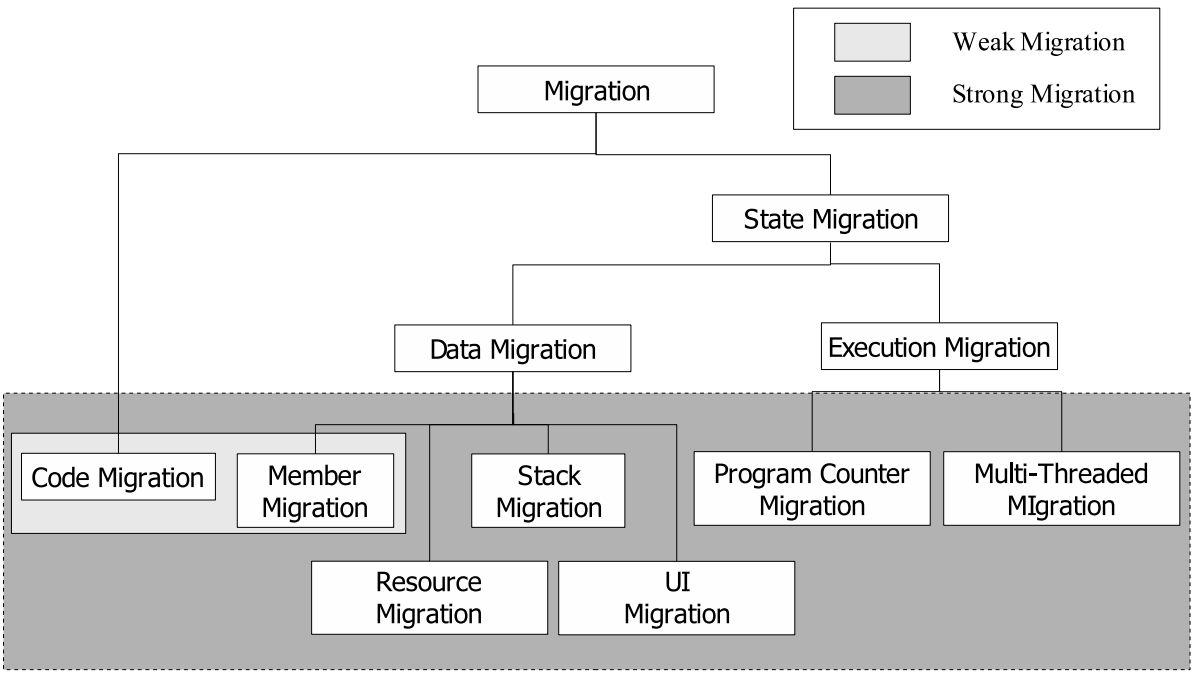


図 2.1: マイグレーションの分離 (論文 [8] より引用)

2.2 JavaVM を利用した既存システム

ここでは具体的な既存モバイルエージェントシステムを紹介し、その特徴と個々のシステムにおける各問題へのアプローチ方法について述べる。

2.2.1 Voyager

Voyager とは ObjectSpace により開発された JavaVM 上のエージェントシステムである。Voyager では、実行状態保存の対象となるのは Java のシリアライズ機

構を利用したインスタンス変数である。

また移動の後に、移動先コンピュータ上で最初に呼び出されるメソッドを指定することができる。言語の拡張などは少ないため、単純な移動処理のモバイルエージェントには適している。しかし指定されたメソッドを呼び出すという機能しかもっていないため、移動経路のための処理や移動後のデータの復旧などは全て開発者が記述しなくてはいけない。そのため移動処理のための記述力が十分とは言えない。

2.2.2 Aglets

Aglets は日本 IBM によって開発されたモバイルエージェントシステムである。モバイルエージェントプログラムは複数のコールバックメソッド群から構成されている。開発者は、モバイルエージェントの移動処理を、ランタイムシステムから呼び出されるタイミングごとにコールバックメソッドとして定義できる。そのため移動に伴う処理を明示的に記述することが可能で、エージェントの振る舞いが明確になる。しかし Aglets は Voyager 同様、Java のシリアライズ機構を利用しているためインスタンス変数のみしか保存の対象にしていない。また、コールバックメソッドは同じ Java コード内に記述するので、ローカル処理を独立して記述することが難しい。そのため既存プログラムコードを熟知していないとエージェント化は難しい。

2.2.3 JavaGO

JavaGO [6] は東京大学米澤研究室によって開発されたモバイルエージェントシステムである。JavaGO ではプログラムソースコードを変換することで実行状態の保存と復旧の機能を付加する。そのため開発者は移動について意識せずにローカル処理を記述することができる。しかし、エージェントの移動タイミングはプログラム内に記述する必要があるため、ローカル処理の記述内に独自の Java メソッド

を埋め込まなくてはいけない。そのため、ローカル処理部分を熟知していなくてはエージェント化ができない。また、決められた位置でしか実行状態の保存と復旧ができないため、耐故障性のあるモバイルエージェントシステムとしては不十分である。

2.2.4 MOBA

MOBA (Mobile Agent Facilities on Java Language Environment) [9] は、首藤一幸氏により開発されたモバイルエージェントシステムである。MOBA は、ネイティブメソッドを利用して JavaVM 自体に実行状態保存機能を追加している。そのため開発者はプログラムに特別な記述をすることなく、プログラムカウンタやスタック内部情報までも移動することができる。このことから既存プログラムのエージェント化はとても容易である。しかし、MOBA の手法は JIT コンパイラとの整合性が悪く JIT 無しでないと動作しない。また、JavaVM のバージョンアップに伴う修正のための負担が大きい。

表 2.1: 既存システムの比較

System	Voyager	Aglets	JavaGO	MOBA
実行状態保存方法	直列化機構	直列化機構	コード変換	JavaVM の変更
状態の保存の度合い	weak	weak	strong	strong
移動処理の記述力	×	○	×	△
既存プログラムの利用	△	×	△	○
パフォーマンス	○	○	△	×

2.2.5 既存システムの比較と考察

表 2.1 に示されるように、Moba を除いた純粋な JavaVM を利用したシステムでは、実行状態保存や移動処理の記述力は既存プログラムの利用のし易さとトレードオフになる傾向が強い。これは既存の記述手法では、同じ Java コード内に移動処理のためのコードを埋め込まなくてはいけないため、様々な記述の制約を設けざるを得ないためである。また実行状態保存や移動処理は、実際にいつ呼び出されるか分からないため、どのようにプログラムコード中に埋め込むかが大きな問題となる。さらに以上のような埋め込みや JavaGO のようなソース変換が必要な場合、プログラムコードを熟知していることが必要になる。この問題は既存プログラムの容易な利用を妨げている。

2.3 本研究の提案

ここで既存システムの問題を解決するための本研究の提案を述べる。本研究では、ローカル処理部分と移動処理部分を完全に分離するため、移動処理のみを独自の専用言語によって表現することを提案する。専用言語で表現することから、エージェントの移動処理をローカル処理に依存することなく記述することができる。また実行状態保存が必要な場合には、ローカル処理部分のソースコードの自動変換により実現する。

第 3 章

システム概要

3.1 設計方針

JavaVM を利用してモバイルエージェントシステムを実現しようとしたとき、Java 言語の移動処理の記述力や JavaVM の実行状態保存機構の問題から、移動処理部分をエージェントプログラムから分離して記述できない。ここでは、この問題を解決するため本研究で開発するモバイルエージェントシステムの設計方針を述べる。

- 最適の実行状態保存の実現

  本システムではプログラムに合った最適な実行状態の保存を、ソースコードの自動変換により実現する。第 2 章で述べたように実行状態を保存するために Java 言語に制約を設けると、既存プログラムのエージェント化が難しくなる。そのため、JavaGO のようにコードの自動変換手法を用いて、実行状態保存機能の付加を実現する。さらに本研究では、JavaGO のように決められた位置で実行状態を保存するのではなく、チェックポイントを自動的に挿入することにより、実行中の定期的な保存を実現させる。このことにより、予期しないネットワークの断線などにも容易に対応できる。

- 移動処理部をローカル処理部分から分離

  本システムでは既存プログラムを容易にエージェント化できるように、移動

処理部をローカル処理から分離する。そして移動処理部をエージェントに特化した独自の専用言語によって記述する。第 1 章で述べたように Java 言語には移動という概念が欠けており、移動経路の作成や依存関係のような移動処理のための記述力が低い。そのためプログラムコードが複雑化したり、肥大化したりして開発者以外の者による修正が困難になる。本研究が提案する専用言語を使用することにより、明確な移動処理部分の記述を実現する。また、ローカル処理部分は純粋な Java 言語で記述できる。このため、開発者の習得負担は少なく済む効果がある。

- ネットワーク構成把握の負担を軽減

  本システムは、開発者のネットワーク構成を把握する負担を軽減するための記述法を考案する。第 1 章で示したようにモバイルエージェントは、移動という概念を用いることにより従来の通信処理を単純化できる。しかし、移動処理自体の記述が複雑であっては開発者の負担は変わらない。本研究で提案する専用言語は、ホスト名や IP アドレスを熟知していない開発者が、容易に移動経路を作成することができる。

3.2 本システムの概要

この節では、我々が開発するモバイルエージェントシステムの概要を示す。まず本システムではエージェントの記述から移動処理部分とをローカル処理と分離し、互いに独立して記述できるようにする。本システムでは、純粋な Java 言で記述したローカル処理、移動処理を定義した移動ルールを付加することによって、既存プログラムを容易にエージェント化させることができる (図 3.1 参照)。移動ルールを独立に記述できるため、移動経路の変更や、ローカル処理部分の変更による依存関係の修正が明確で最小限で済む。

我々は本システムを実現するため、まず移動ルールを定義する移動ルール定義

言語 passport を考案した。また passport により記述された移動ルールを利用するモバイルエージェントシステムを実現するため、ランタイムシステム AIR–port を実装した。以下に passport と AIR–port それぞれの概要を示す。

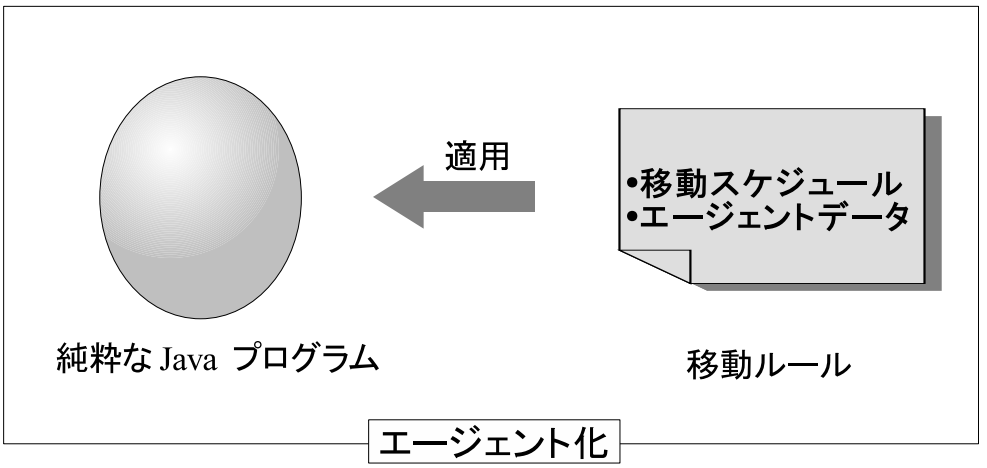


図 3.1: エージェント化

3.3 passport の概要

passport はモバイルエージェントの移動概念を移動ルールとして記述するための定義言語である。passport の主な特徴は以下のとおりである。

- エージェントのローカル処理から移動処理部分を分離することができる
- ネットワーク構成を意識することなく移動経路の作成ができる
- エージェントの移動に伴う依存関係を明示的に記述することができる

このような特徴から passport を用いることで次のような 3 つの利点を得ることができる。第一に既存プログラムをエージェント化することが容易になる。ローカル処理部分と移動処理部分が切り離されたことから、既存のプログラムを修正す

ることなく活用することができる。第二にネットワーク構成を熟知していない開発者によるエージェント開発ができる。移動経路をホスト名や IP アドレスではなく、OS やマシンスペックなどの実行環境によって作成するための記述法が備わっている。そのため開発者はエージェントを動作させたい実行環境を定義するだけで良い。第三に開発者以外の管理者による運用が容易にできる。passport を使用することにより、移動先に応じたエージェントの依存関係を明確に記述することができる。そのためネットワーク構成を変化させたい場合、開発者以外のネットワーク管理者は、容易にその修正を行うことが可能である。

3.4 AIR–port の概要

AIR–port は移動ルールの概念を利用したモバイルエージェントのランタイムシステムである (図 3.2 参照)。AIR–port のエージェントは passport により記述された移動ルールと、Java 言語によって記述されたローカル処理から生成される。AIR–port では、エージェントが実行途中で停止したとしても実行状態保存・復旧機構を備えているため、実行状態を保存し別コンピュータ上で実行を再開することができる。実行状態の保存は開発者の意図する点だけで起こるのでなく、プログラム中に自動的に挿入された複数の箇所で行われる。そのため、負荷の集中などによって起こる動的な移動にも対応することが可能である。

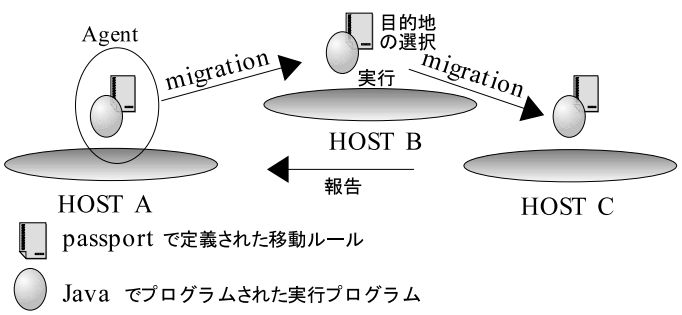


図 3.2: AIR–port システムの概念図

3.4.1 コード変換

AIR–port 上で動作するエージェントは、ローカル処理を記述した Java コード、移動処理を記述した passport コードから、独自の変換により生成される。具体的には、既存プログラムに実行状態の保存・復旧機能や、移動処理のための機能が付加される。コードの変換後は、全てのコードが純粋な Java コードに変換され、Java コンパイラを用いることによって AIR–port 上で実行可能なクラスファイルにコンパイルされる (図 3.3 参照)。

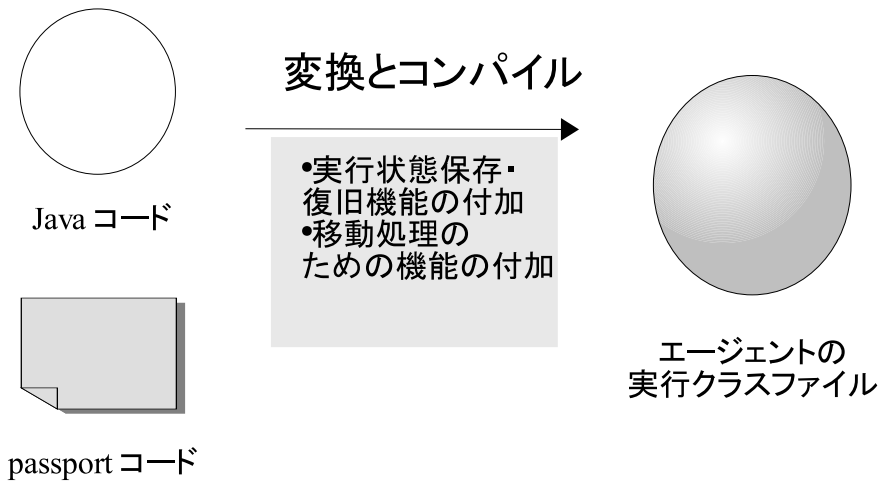


図 3.3: コード変換によるエージェント生成

第 4 章

移動ルール定義言語 passport

ここでは、我々が提案する移動ルール定義言語 passport の詳細を述べる。passport の使用によって実現できることを以下にあげる。

- 移動処理が機能ごとにモジュール化されるため、視覚的に移動処理のための定義 (移動ルール) が把握しやすい
- エージェントの依存関係や移動経路などエージェント独特な処理のための表現が容易になる
- 具体的なホスト名や IP アドレスを利用せず、エージェントの実行環境の指定により移動先コンピュータを指定できる
- 移動先や移動元の実行環境に応じた処理の定義ができる

このことからネットワーク構成を熟知していない開発者による移動経路の作成や、ローカル処理を熟知していない管理者によるエージェントの管理が容易になる。その結果、既存 Java プログラムのエージェント化、そしてその管理が円滑になる。

4.1 移動ルール

移動ルールとは、エージェントプログラムから移動の概念を分離したものである。本研究では以下に示すデータと処理をエージェントの移動ルールとして passport

によって定義する。

- 各ホストに滞在できる時間や、最大移動回数などのエージェントデータ
- 移動経路や移動条件を含めた移動スケジュール
- 依存関係を含めた環境適応処理
- ユーザの要求時やコンピュータ故障時における例外処理

4.2 言語の特徴

passport は以下のような特徴を持っている。

- 文法構造

  passport は Lisp をもとにした文法構造をもつ。これは移動経路の柔軟な操作を実現するために、全ての要素をリスト構造として扱うためである。また、OS やマシンスペックなどの実行環境を条件にした移動先候補を集合として表現できる。集合とは、リスト構造とは違い、順序付けされていない要素を扱うためのデータ構造である。集合の概念を用いることにより、移動先候補の自由な作成が可能になる。passport では、このようにリスト構造と集合により移動経路の作成ができる。

- 明確な記述法

  passport はエージェント特有の機能を明確に分離するため、FIELD という概念を持っている。エージェントは状態 (生成、到着、移動、破棄など) の変化によって移動処理を起こす。FIELD は処理の記述をこのような状態ごとにモジュール化するための機能である。

- ネットワーク構成を意識させない記述方法

  passport の記述は複雑なネットワーク構成を熟知していない開発者を考慮した記述方法である。具体的なホスト名や IP アドレスを知らなくても移動先の実行環境 (OS やマシンスペックなど) を表現する機能が備わっている。

- 移動条件の定義

  passport には実行環境の変化に伴う処理を定義するための環境定義変数が用意されている。モバイルエージェントは実行環境の変化に応じてプログラムを中断し移動処理に移らねばならない。またそれに伴い実行状態の保存や復旧、実行の再開などの処理を行う。このような環境変化の通知は全てランタイムシステムから発行される。ランタイムシステムから通知を受け取るためのインターフェース変数として passport では環境定義変数 (4.5.3 小節 flag 型変数参照) を用意している。

4.3 データ構造

passport は複数のデータ要素を表現する方法として、リストと集合という 2 つのデータ構造をもっている。集合とは要素が順序付けされていないデータ構造を意味する。リスト構造とは、集合の要素を順序付けしたものである。

4.3.1 リスト

passport で扱うリストは Lisp と同様に S 式を用いて表現する。しかし関数の役割や変数のデータ型はエージェントに特化している。

以下では passport におけるリストの定義を述べる。

- リストはアトムとリストから構成される
- 集合はアトムとして扱う

4.4 集合

passport では集合を ‘[’ と ‘]’ とで囲むことによって表現する記述法を用いる (図 4.1 参照)。以下に集合の定義を述べる。

- リストはアトムとして扱う
- 集合の要素はアトムと集合から成る
- 各集合にはそのデータ型によって決まる全体集合が定まっている
- 全体集合はデータ型に依存する
- 空集合を [emp] と表現する

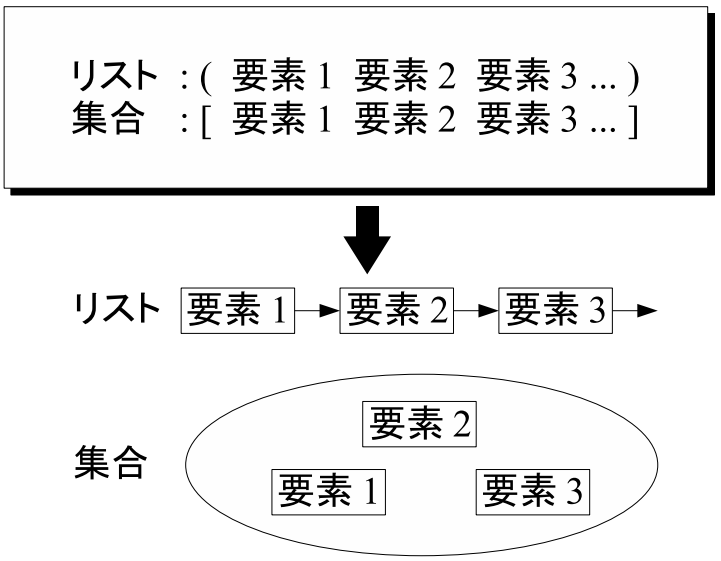


図 4.1: リストと集合

4.5 言語構造

4.5.1 FIELD

FIELD とはエージェントの移動処理を、呼び出される状態ごとに明確に分離して記述するための機構である。この考え方は Aglets のコールバックメソッドに似ている。しかし Aglets では Java 言語を用いているため、移動スケジュールを作成するためのモジュールを分離したり、依存関係処理のためのモジュールを分離したりすることは難しい。passport では、そのようなエージェント特有の処理をモジュールとして分離することができる (表 4.1 参照)。また FIELD ごとに使用可能な処理が定義されており、記述の安全性が考慮されている。

表 4.1: FIELD

FIELD 名	対応するタイミング
Creation	エージェント生成時
Destraction	エージェントの消滅時
Schedule	スケジュール作成時
Departure	エージェントの移動時
Arrival	エージェントの到着時
Dependence	依存関係の変更時
Exception	故障・例外の発生時

4.5.2 標準関数

passport ではリストを扱う関数と集合を扱う関数の 2 種類の標準関数をいくつか用意している (付録 A、表 A 参照)。どちらも第一要素が評価の対象となる。

リストを扱う関数

リストを扱う関数は文法上、Lisp の記法と同じである。passport に用意されている関数は全てエージェントに特化した関数である。たとえば集合を順序付けしてリストにする関数や行き先の環境依存処理を書き分けるための制御関数などが用意されている。

集合を扱う関数

集合を扱う関数は集合の演算をしたり、集合の生成を行ったりするものがある。集合の演算には、和集合や積集合の演算などの基本的な集合演算、ならびに、条件に従って部分集合を形成するもの (表 4.3、4.2 参照) が用意されている。部分集合を形成する場合の条件とは、たとえば移動先の候補地の集合を作成する場合に、ある OS の種類などが動作しているマシンを集合としたい場合などに用いる。

表 4.2: 集合の表現

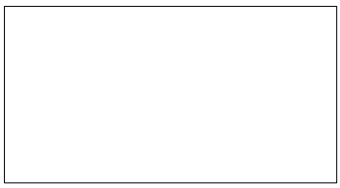
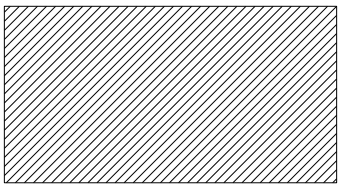
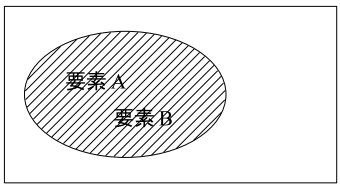
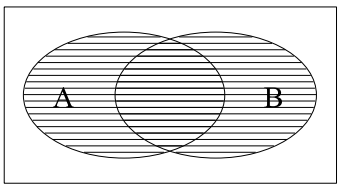
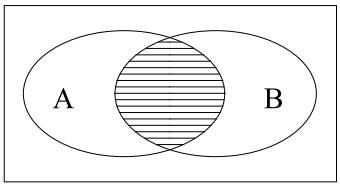
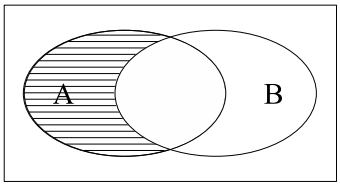
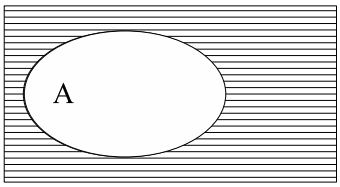
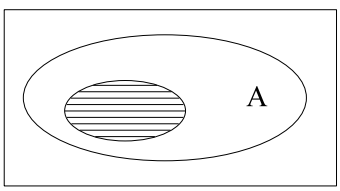
記述方法	図	意味
[emp]		空集合
[uni(データ型)]		全体集合
[quote 要素 A 要素 B ...] (or '[要素 A 要素 B ...])		集合

表 4.3: 集合の演算

記述方法	図	意味
[+ A B]		和集合
[* A B]		積集合
[- A B]		差集合
[! A]		補集合
[subset(条件) A]		部分集合

4.5.3 flag 型変数

エージェントは状態の変化に応じて移動処理を実行する。そのため、passport ではランタイムシステムから状態変化の通知を受け取るための flag 型変数が用意されている。flag 型変数が通知を受け取る状態変化とは以下の 3 つがある。

- ネットワーク断線やコンピュータリソースの低下などの故障時
- ユーザからの移動要求が出た場合
- 開発者や運用者の意図する実行位置、または移動条件に合致した場合

passport では flag 型変数を用いて、この 3 つの状態変化を定義することにより、エージェントが移動するタイミングを指定する。

4.5.4 特殊なシンボル

passport では、環境の状態や、flag 変数で利用する状態変化の定義のために特殊なシンボルが用意されている (付録 A、表 A.2 参照)。これらはリストのアトムとして使用される。

4.6 移動経路の作成

移動経路とは集合を使って作成した移動先候補を順序付けしたものである。passport では開発者にネットワーク構成を意識しない移動経路の作成を実現するために、ホスト名や IP アドレスを指定する必要なく移動経路を作成することができる。本研究ではエージェントが移動可能なホストを PLACE と呼び、ある条件によって定義された PLACE の集合を AREA と呼ぶ。

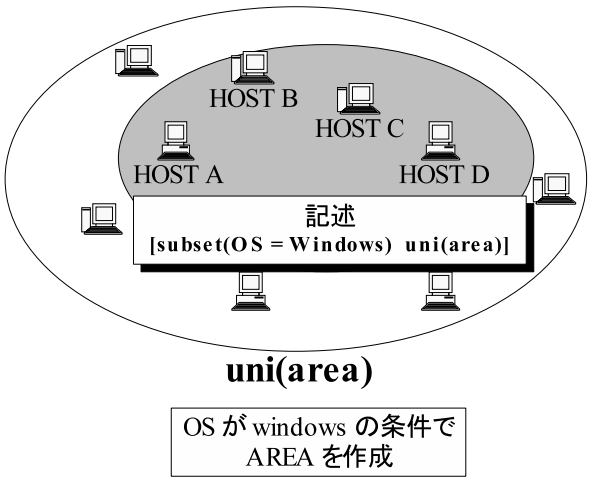


図 4.2: AREA の作成方法

4.6.1 AREA の作成

開発者が定義する AREA を作成するには集合の関数を利用して行う。また area 型の全体集合はエージェントが移動できる本システムの仲介サーバ (後述) により管轄される PLACE の集合と定めている。開発者は全体集合に対する部分集合として AREA を定義する。部分集合を定義するには subset 関数を使用する。subset 関数は元となる集合と部分集合の条件を与えることで新しい集合を作成するための関数である。subset 関数を利用すると図 4.2 に示すように、たとえば全体集合から OS が Windows である PLACE の部分集合を作成することができる。

4.6.2 PLACE の順序付け

作成された AREA を sequence 関数を利用して、順序付けすることにより移動経路を作成することができる。passport では移動経路の種類を以下の 3 つにわけている (図 4.3 参照)。

- エージェント生成時に AREA を固定的に順序付けした移動経路

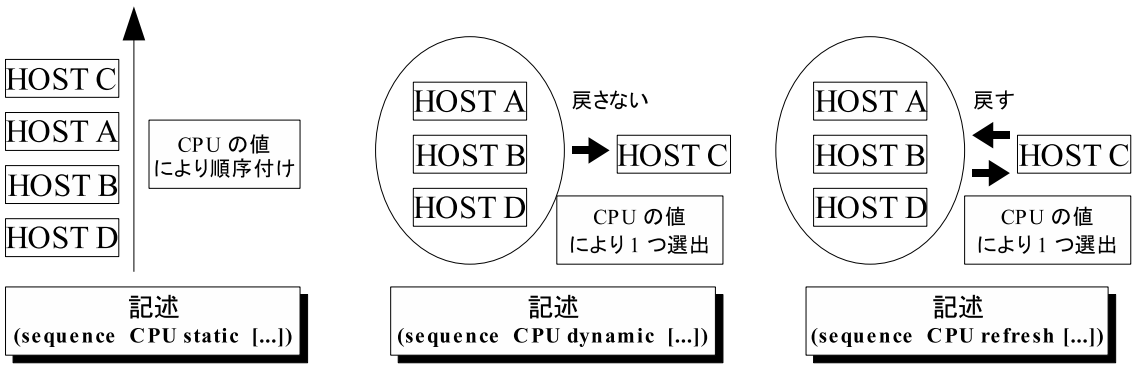


図 4.3: 移動経路作成方法

- AREA から PLACE を動的に 1 つずつ選出していき、選出された PLACE を AREA から抜いていく移動経路
- AREA から PLACE を動的に 1 つずつ選出していき、選出された PLACE を AREA から抜かない移動経路

passport ではこのような移動経路をある条件を与えることにより作成することができる。ある条件とは各 PLACE の CPU クロック周波数やメモリ容量、または単位時間当たりのトラフィック平均値などである。このような条件を設定することで、最もリソースの空きがある PLACE を選択しながら移動できる経路を作成することができる。

4.7 移動スケジュール

passport における移動スケジュールとは移動経路に移動条件を関連付けたものを言う。作成した移動経路はそのままでは利用することはできない。各 PLACE へ移動するタイミングや条件を定義付ける必要がある。passport では、移動タイミングや条件を flag 型変数を用いることによって定義することができる。

4.7.1 移動タイミング

移動のタイミングとはエージェントが移動するための条件である。passport では移動タイミングの定義を以下の 2 つに分類する。移動タイミングを移動スケジュールで定義した点、つまり開発者の意図する条件でのタイミングと、開発者の意図しないタイミングである。

まず、移動スケジュールとして定義される開発者の意図する点でとは、ローカル処理内に定義された任意の実行位置である。passport ではこの開発者が移動するタイミングをリスト構造の特殊シンボル “POINT” で表すことができる。(POINT のシンボル名はエージェント開発者により変更することもできる。) 開発者は、ローカル処理の記述中に単純な Java 言語メソッドを埋め込むことによって、任意の実行位置を定義して、POINT として扱うことができる。ただし、POINT の終端には、ローカル処理の終端という実行位置だけは自動的に POINT リストの終端に付加される。

移動スケジュールは移動経路に、POINT を関連付けることによって作成することができる。POINT との関連付けは移動経路に似た以下の 3 つの種類がある。POINT の関連付けは、は set–point 関数を用いて行う (図 4.4 参照)。

- POINT が順序付けされており、POINT を順序通りに評価していく。評価が終端までいくと先頭に戻る。
- POINT は順序付けされず、全ての POINT を毎回評価する。1 度合致した POINT は破棄される。
- POINT は順序付けされず、全ての POINT を毎回評価する。合致した POINT はそのまま残される。

次に、ユーザによる要求や負荷集中を起因に起きる移動の場合、移動スケジュール外のタイミングで移動が発生する。そのため passport では、このような場合の移動は移動スケジュールの例外の移動処理として定義する。

この移動タイミングの記述手法により、開発者の意図する点での移動と、例外での移動とが明確に分離し、視覚的に把握しやすくなる。

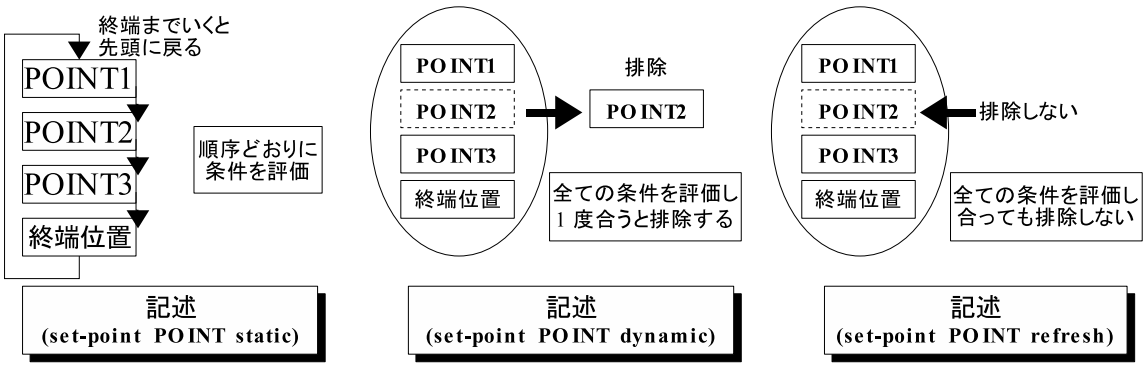


図 4.4: flag の関連付け

4.7.2 移動スケジュール作成方法

移動スケジュールは、移動経路と移動タイミングから作成される。具体的な作成方法は、リスト操作関数 set–schedule 関数を用いて行う (図 4.5 参照)。set–schedule 関数は引数として、スケジュールを格納するための変数 (シンボル)、そして移動経路と移動タイミングを持つ。移動経路と移動タイミングから生成されたスケジュールは第一引数の変数に格納される。

```
(set-schedule
  schedule1
  (sequence ...)
  (set-point ...))
```

図 4.5: スケジュールの作成方法

4.8 環境依存処理の定義

環境依存処理とはエージェントの移動先の環境に依存した処理のことである。passport ではこのような環境依存処理を 2 種類に分ける。1 つはクラスファイルの置き換えやメソッド呼び出しなどのようなローカル処理に関わる処理である。もう 1 つは移動の際に生じるエージェントの依存関係処理である。この 2 つの処理はどちらも route–table という passport 独自の制御関数を使って記述する。

4.8.1 route–table

route–table とは、移動の際の状況を、移動先、移動元、移動時の状態によって場合分けをするための制御関数である。図 4.6 に示すように、route–table は移動先や移動元によって処理を場合分けし、それぞれに応じた処理を { } 内に記述することができる。たとえば図 4.6 の例において、Windows のコンピュータから Linux のコンピュータに移動する場合、2 行目と 8 行目の条件に合い、処理 1 と処理 3 が実行される。

4.8.2 移動時の処理

移動時におけるエージェントの処理とは、環境に依存する同名のクラスファイルを置き換える処理や、経過報告をエージェントが生成されたホストに報告する処理のことである。さらに、Voyager と同様に移動後に実行したい Java のメソッドを定義することもできる。

4.8.3 依存関係処理

依存関係処理とはエージェントの移動時に、一緒に移動するファイルやディレクトリ (以降、移動メンバと呼ぶ) を明示的に指定するための処理である。ここで使用される標準関数は with 関数と without 関数である。その記述例を図 4.7 に示す。

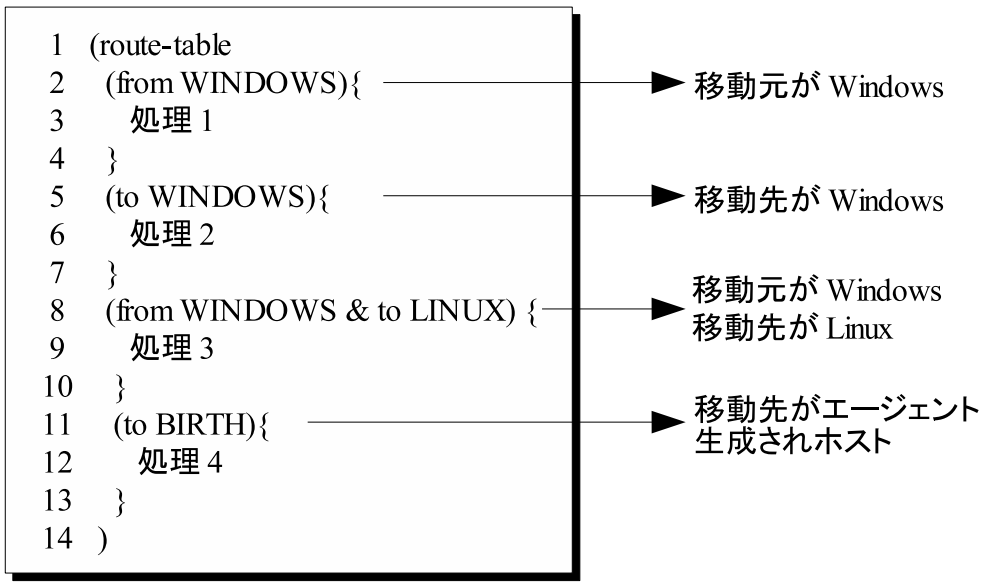


図 4.6: route–table の使用方法

with 関数 指定したファイルやディレクトリを移動メンバに加える

without 関数 指定したファイルやディレクトリを移動メンバから外す

```
(with "dirA")
(without "dirA")
```

図 4.7: with 関数と without 関数

4.9 例外処理

passport では、ユーザの移動要求やコンピュータの故障時の処理を例外処理として定義できる。例外処理の定義は catch 制御関数を用いる。図 4.8 では、2 行目でネットワーク断線、5 行目でユーザの要求が定義されている。

```
1   (exception-table
2     (DISCONNECTION {
3       処理1
4     })
5     (USER {
6       処理2
7     })
8   )
```

図 4.8: excepiton-table 制御関数

第 5 章

AIR–port

この章では移動ルールを用いたモバイルエージェントのためのランタイムシステム AIR-port の実装方法について述べる。

5.1 AIR–port の構成

AIR–port は大きく分けて 3 つの機構から構成される (図 5.1 参照)。以下に各機構の詳細を述べる。

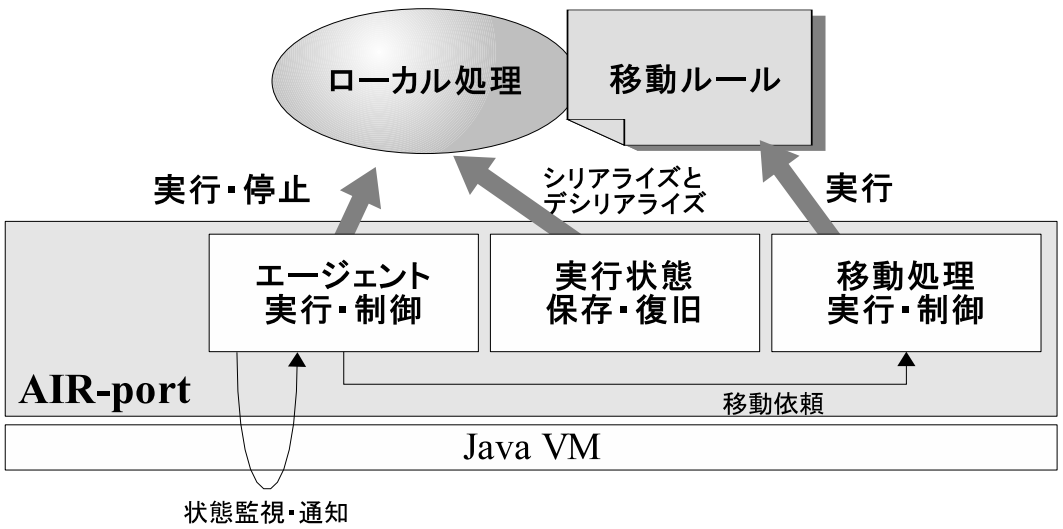


図 5.1: AIR–port の構成

エージェント実行・制御機構: 移動の前後でローカル処理の実行と停止を制御するための機構。また実行環境の状態を監視し、その状態変化から移動命令などをエージェントに通知する。

実行状態保存・復旧機構: エージェントプログラムの実行状態を保存・復旧するための機構。この機構は実行中のエージェントを常時監視し、プログラム中に定義された保存箇所で実行状態の保存を実行する。

移動処理実行・更新機構: 移動ルールに従った移動処理の実行をするための機構。この機構は移動依頼を受けると、対象となるエージェントの移動ルールを分析して、それに従った移動処理を実行する。

また AIR–port を利用する場合、移動ルール概念を持ったエージェントを生成するためのコード変換機 (以降、トランスレータと呼ぶ) と、各ホストを監視し動的に移動経路を作成するためのサーバ (以降、仲介サーバ) が必要となる。仲介サーバを含むシステム全体図を図 5.2 に示す。

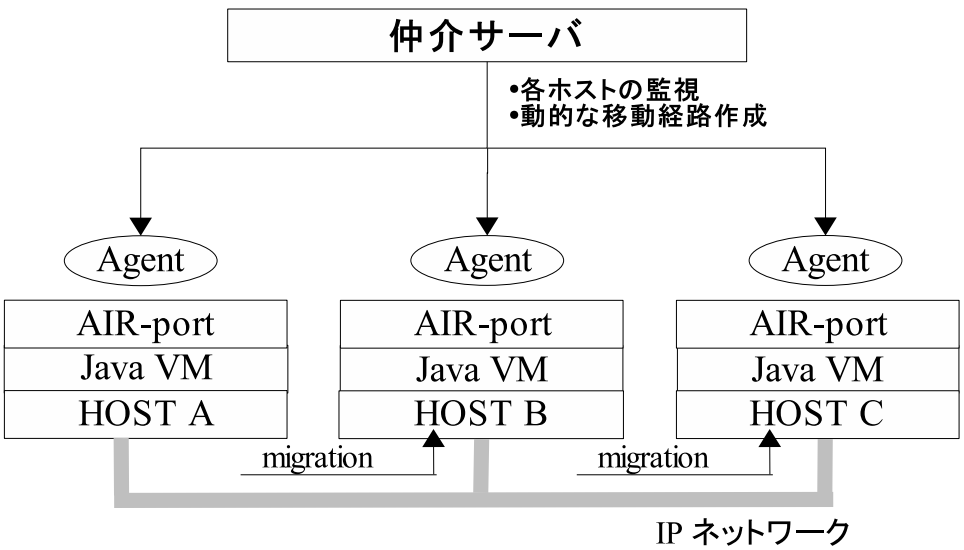


図 5.2: 全体の構成

5.2 トランスレータによるエージェントの生成

AIR–port 上で実行されるエージェントは専用のトランスレータによって生成する。トランスレータは passport コードと Java コードから AIR–port で実行可能な Java コードを生成する。これらの生成した Java コードは純粋な Java 言語で記述されたものである。Java コードを Java コンパイラに通すことにより、本システムで動作するクラスファイルが生成される。以下にモバイルエージェントを構成する、各クラスファイルを解説する。

5.2.1 エージェントの構造

AIR–port におけるモバイルエージェントは、大きく分けて以下の 4 つのクラスから構成される (図 5.3 参照)。

スレッド制御クラス: ランタイムシステムからエージェントのために与えられたスレッドを制御するためのクラス。スレッド制御クラスは、ランタイムシステムからの実行・停止命令を受け、ローカル処理を実行したり、このスレッドの停止を行ったりする。

ローカル処理クラス: ローカル処理部分の記述から生成されるローカル処理の実行クラス。開発者の提供する既存の Java コードにランタイムシステムから実行を開始するための変換 (main 関数の変換など) が行われたものである。また、実行状態保存と復旧をするための変換がされている。

移動ルールクラス: passport の記述から生成されるクラス。passport により定義された処理が、具体的に Java コードによって定義されている。

実行状態保存処理クラス: ローカル処理の実行状態を保存・復旧するためのクラス。このクラスはローカル処理のデータを保存するためのデータ格納クラスと、保存・復旧処理を行う処理クラスから成る。

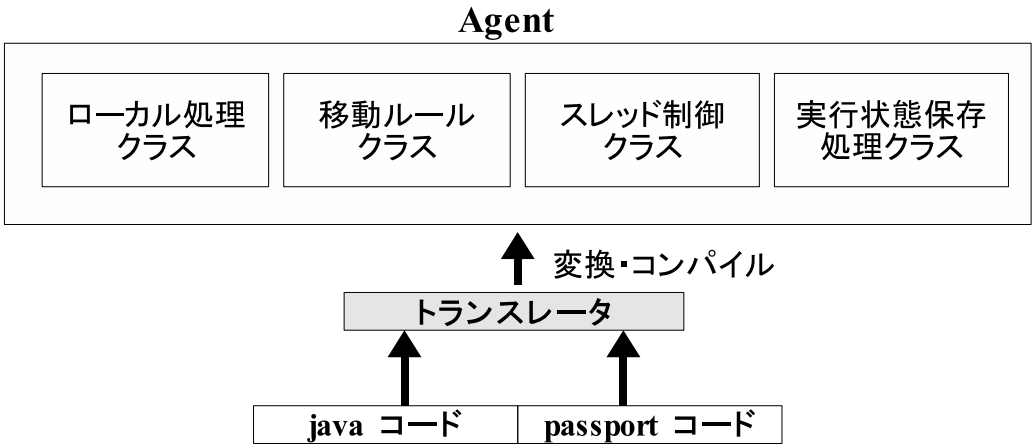


図 5.3: トランスレータによる変換

5.2.2 クラス名の衝突回避のための名前変換

移動するエージェントのクラスを動的にロードする場合に、同じ名前のクラスと衝突する場合がある。本システムではこのクラス名の衝突を避けるために、全てのクラス名の後ろにエージェント の ID を付加する。エージェントの ID とはエージェント生成時にランタイムシステムに登録される識別番号であり、システム全体で一意性が保証される。

5.3 エージェント実行・制御機構

ここでは、エージェント実行・制御機構の詳細を述べる。エージェント実行・制御機構はエージェントプログラムの実行と停止を制御する。また実行環境の状態を監視し状態変化に応じて移動命令などの通知を行う (図 5.4 参照)。

まず到着後のエージェントが実行されるまでの手順について述べる。エージェントの実行は、この機構が持つエージェントを動的にロードするためのエージェント専用ローダによって行われる。ローダはまずエージェントの到着通知を受けると、到着したエージェント内のクラスファイルを分析し、必要なクラスファイルを

全てロードする。次に、ロードしたクラスの中で “StartAgent”から始まる名前のクラス (以降、単に StartAgent クラスと呼ぶ) をインスタンス化する。StartAgent クラスとは、実行の開始と停止を行うためのスレッド制御クラスのひとつである。StartAgent クラスがインスタンス化されると、ランタイムシステムは StartAgent クラスを子スレッドとして実行する。次にランタイムシステムは、まずエージェントのデータ (これまでの移動経路など) を復帰させるため、AgentContext クラスをデシリアライズする。AgentContext クラスは移動時の移動経路更新などの時に使用される移動ルールクラスのひとつである。最後に StartAgent はローカル処理のエントリポイントとなる doMain 関数を実行する。

次に、エージェントの停止について述べる。エージェントの移動には、ランタイムシステムから移動依頼が出される場合と、プログラム中に移動ポイントを設定している場合とがある。ランタイムシステムから移動依頼が出される場合とは、故障やユーザの要求などの実行の終了要求が出た場合である。プログラム内に直接停止ポイントが定義されていた場合は、移動依頼は出されない。終了要求を受け取るか、または実行が設定された位置まで進んだ時 StartAgent クラスは、次に状態保存が行われるポイントまでローカル処理の実行が進むのを待機する。その後、実行状態保存機構により状態保存が確認されると、StartAgent クラスはランタイムシステムに終了通知を行う。ランタイムシステムは終了通知を受け取った時点で、このエージェントのスレッドを破棄してエージェントの移動依頼を出す。

5.4 実行状態保存と復旧

ここでエージェントの移動に伴う実行状態保存と復旧機構についての詳細を述べる。本研究の実装において保存・復旧の対象とする実行状態は、インスタンス変数、スタック内部情報、プログラムカウンタである。本研究では、トランスレータによってローカル処理部分のコードに変換を加えることによって、保存・復旧機能

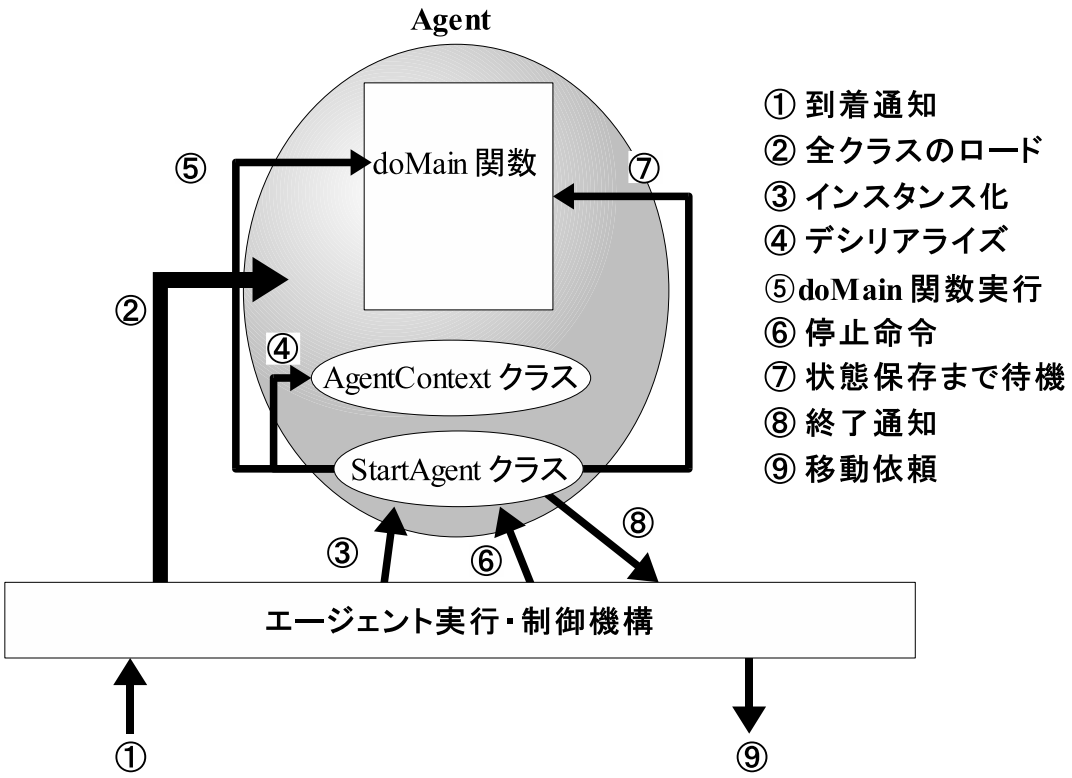


図 5.4: エージェント実行・制御機構

を付加する。

5.4.1 状態の保存と復旧のメカニズム

状態の保存のメカニズムは以下のとおりである。プログラムカウンタ情報、スタック内部情報などを全てインスタンス変数として生成する。また、ローカル変数などはインスタンス変数に置き換え、スタックに全ての実行状態情報を積み、定期的にファイルに書き込んでいく。復旧する場合には、状態が保存されたデータファイルからデシリアライズしてスタックを復旧し、プログラムカウンタ情報からエントリポイントを決定する。

実行状態情報をインスタンス変数として格納するため、トランスレータによってスタックの階層が進むごと (具体的にはローカル処理コード中のメソッドに入るた

び) に以下に示す 2 つのクラスを生成する (図 5.5 参照)。

コンテキストクラス 全てのメソッドに共通するデータや処理を定義したクラス

スタックフレームクラス ローカル変数など各メソッドごとに異なるデータや処理を扱うためのクラス

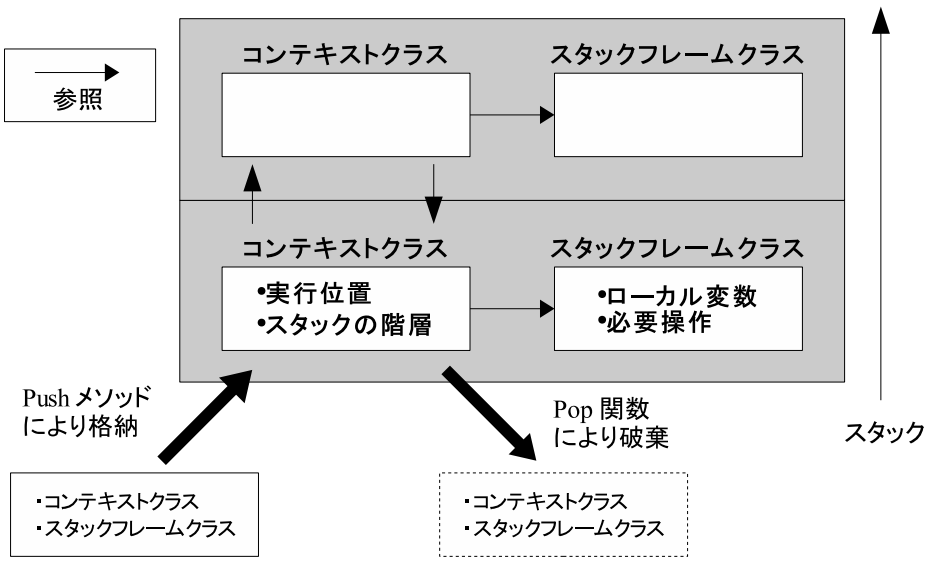


図 5.5: 状態保存のメカニズム

これら 2 つのクラスはエージェント実行中に push メソッドを呼ぶことによってスタックデータとして蓄積され、pop メソッドを呼ぶことによってスタックデータから取り除かれる。push および pop メソッドは、トランスレータによってエージェントコード内に自動的に埋め込まれる。

5.4.2 コンテキストクラス

コンテキストクラスにはプログラムカウンタやスタック階層情報など、全メソッドに共通の実行情報と処理が定義されたクラスである (図 5.6 参照)。コンテキストクラスはローカル処理のメソッドに入ると呼び出される push 関数によって、インスタンス化される。インスタンス化されたコンテキストクラスは、対応するメソッドのスタックフレームクラスをインスタンス化する (図 5.7 参照)。

```
public class Context implements Serializable {

    /*  実行ポイント */
    public int point;
    /*  スタックの階層  */
    public int dep;
    /*  メソッド ID */
    public int id;
    /*  スタックフレームクラスへの参照  */
    public Object o;

    /*  上方へのスタック参照  */
    Context prevFrame;
    /*  下方へのスタック参照  */
    Context nextFrame;

     ...

}
```

図 5.6: コンテキストクラス

5.4.3 スタックフレームクラス

スタックフレームクラスは、各メソッドによって異なるローカル変数や処理を保存するためのクラスである。メソッドに定義されたローカル変数はシリアライズを可能とするために、全てスタックフレームクラスのインスタンス変数に置き換えら

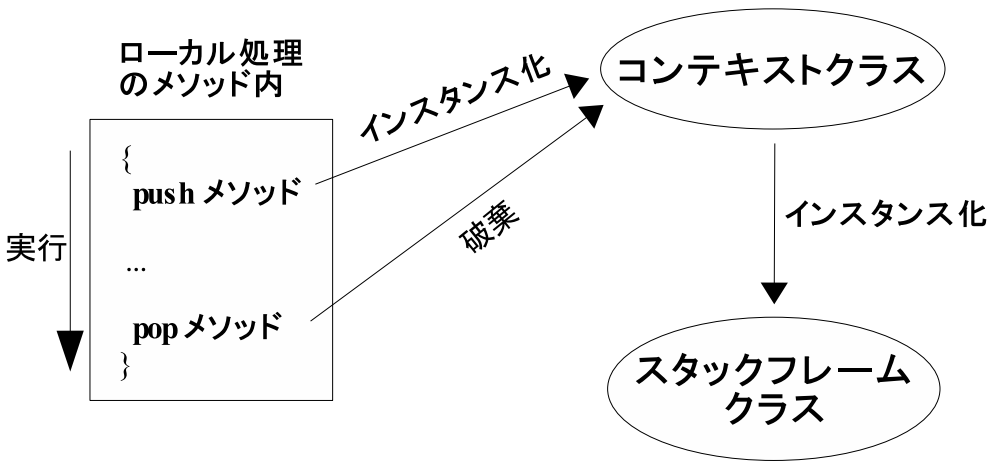


図 5.7: 保存クラスの生成原理

れる (図 5.8 参照)。しかし Java ライブラリが提供するクラスの中の Serialization インターフェースが実装されていないクラスや、static 宣言された変数はシリアライズすることができない。それらの変数は、スタックフレームクラスの restore メソッド内で明示的復旧させなくてはいけない。

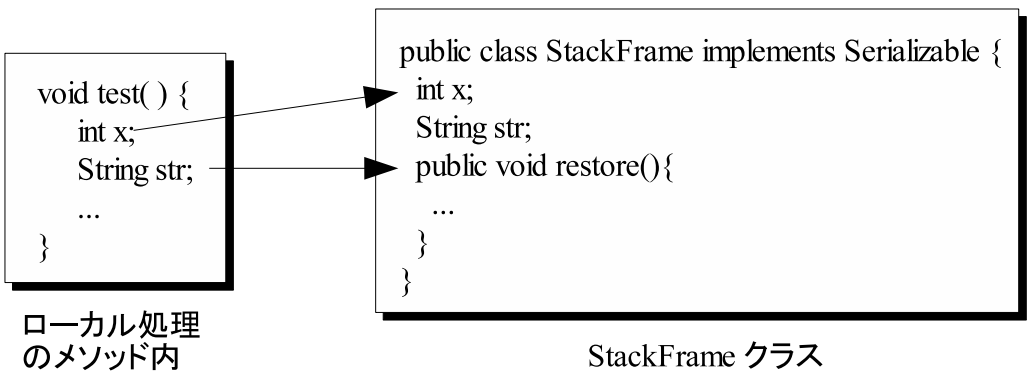


図 5.8: ローカル変数の置き換え

5.4.4 実行状態保存・復旧のための変換

次にローカル処理部分のコードに施されるコード変換の詳細について述べる。コード変換は限られた文法を用いて行われ、開発者が変換後のソースコードを把握できるように考慮されている。そのため、開発者によるより細かい実行状態保存・復旧の設定が可能になる。

基本的な変換

まず基本的な変換手法を説明するために、制御文の無い簡単な例 (図 5.10) を使用する。また以下に、変換例を使って、クラス、doMain 関数、各メソッドにおける詳しい変換方法について述べる。ならびに、状態を保存するための save メソッドと、復旧のための変換についても述べる。

- クラス

  まず全てのクラスに Serializable インターフェースが実装される (2 行目)。この変換により各クラスのインスタンス変数は、特別な操作をする必要なくシリアライズが可能となる。また状態保存処理のためのメソッドが定義された、SaveState クラスをインスタンス化するための処理を、コンストラクタに追加する (3 - 6 行目)。

- main 関数

  main 関数ではこのエージェントのためのコンテキストクラスがインスタンス化される (9 行目)。

- メソッド共通

  ローカル処理内の各メソッドには、全てに固有の ID が与えられる (8、24 行目)。ID は、スタックフレームを識別するために使用される。各メソッドでは、まず push メソッドを利用してスタックフレームクラスをインスタンス

化する (10 - 11、25 - 26 行目) 。この際、ID によってメソッドに対応するスタックフレームクラスを識別する。メソッド内のローカル変数は、この際与えられたスタックフレームクラスのインスタンス変数に変換される。メソッドの終端では、全てのコンテキストクラスとスタックフレームクラスが、メソッド最後に挿入される pop メソッドによって破棄される (21、28 行目)。

- save メソッド

  状態保存はコンテキストクラスの save メソッドによって行われる (15、18 行目)。save メソッドが呼び出される保存位置には、位置を識別するための固有の値が設定されている。そして、save メソッドが呼ばれるたび、その値がスタックフレーム内の実行位置を保存するための変数に代入される。本システムではオーバヘッドを抑えるため、実行状態を保存する位置として、save メソッドが自動的に挿入される位置は図 5.9 に示されるように、メソッドとループ文の前後だけに限定している。

- 復旧のための変換

  本研究では実行位置を復旧するために switch 文を利用している (12 - 20 行目)。保存された point 変数を switch 文のパラメータとして与えることにより実行位置まで処理を飛ばす (以降、早回しと呼ぶ) ことができる。ただし制御文が内在するメソッドについては、特殊な変換が必要になる。制御文の変換についての詳細は後述する。

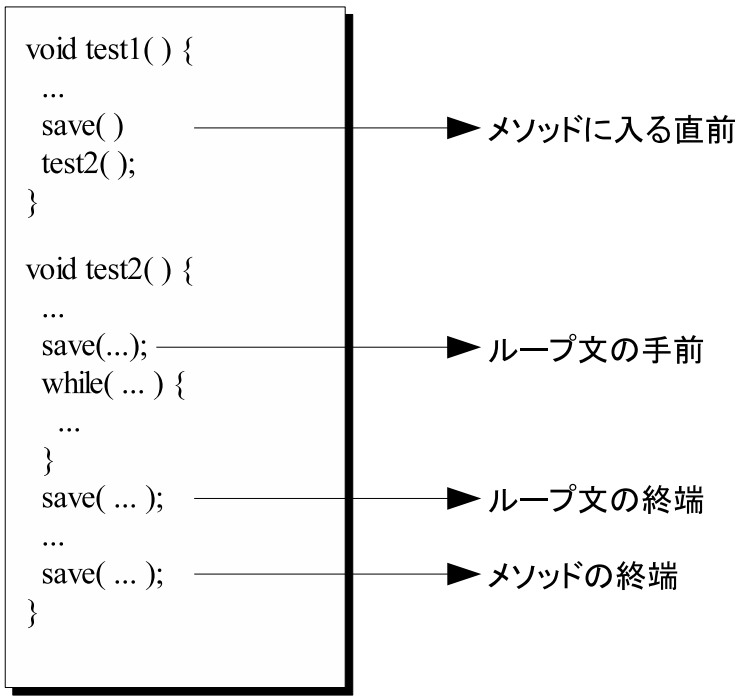
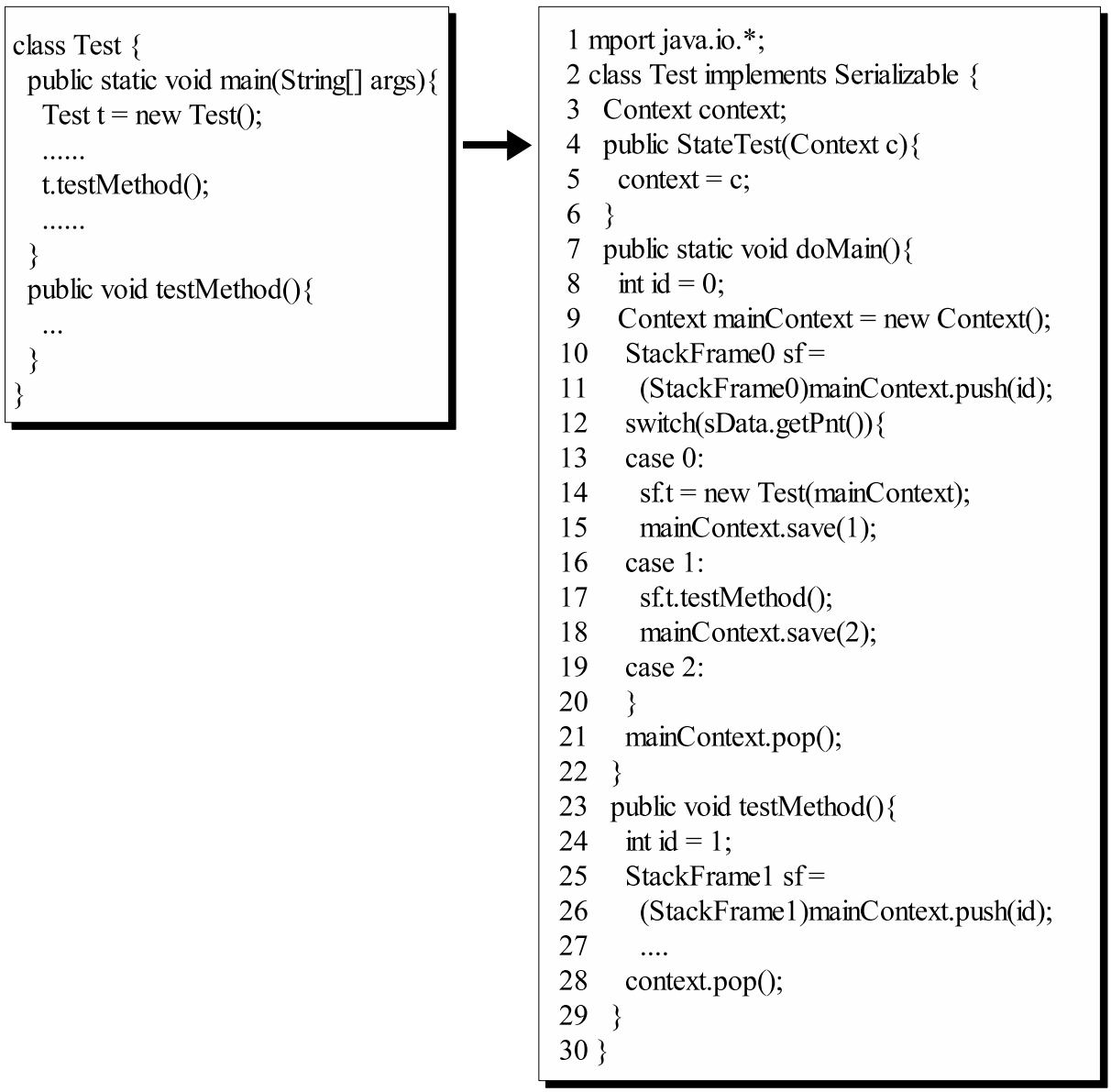


図 5.9: save メソッドの挿入箇所



```
class Test {
  public static void main(String[] args){
    Test t = new Test();
    ......
    t.testMethod();
    ......
  }
  public void testMethod(){
    ...
  }
}
```

```
1 mport java.io.*;
2 class Test implements Serializable {
3   Context context;
4   public StateTest(Context c){
5     context = c;
6   }
7   public static void doMain(){
8     int id = 0;
9     Context mainContext = new Context();
10     StackFrame0 sf=
11       (StackFrame0)mainContext.push(id);
12     switch(sData.getPnt()){
13     case 0:
14       sf.t = new Test(mainContext);
15       mainContext.save(1);
16     case 1:
17       sf.t.testMethod();
18       mainContext.save(2);
19     case 2:
20     }
21     mainContext.pop();
22   }
23   public void testMethod(){
24     int id = 1;
25     StackFrame1 sf=
26       (StackFrame1)mainContext.push(id);
27       ....
28     context.pop();
29   }
30 }
```

図 5.10: 基本的な変換

制御文と引数

制御文が内在するメソッドや引数を持つメソッドについては、より複雑な変換が必要となる。なぜなら復旧の際にも、制御文でパラメータを評価する処理やメソッドへ引数を渡す処理がある場合、通常の switch 文による変換では実行の早回しをすることができないためである。この間に変数の値が変化すると、復旧前の状態と変わってしまう。そのため、引数やパラメータを渡す処理を別途、状態復旧の際に実行しなければいけない (図 5.11、図 5.12 参照) 。

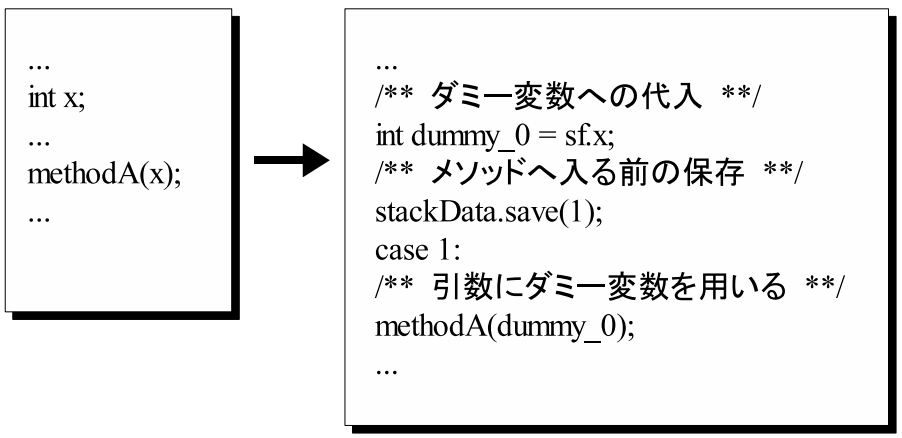


図 5.11: 引数を持つメソッド

5.5 移動処理の実行・制御

ここでは移動処理実行・制御機構についての詳細を述べる。

5.5.1 移動ルールクラス

移動ルールクラスとは passport の定義から生成された実行クラスである。移動処理実行・制御機構は移動ルールクラスを状態に応じて操作することにより、移動処理を実行する。移動ルールクラスは以下の 2 つのクラスから成り立つ。

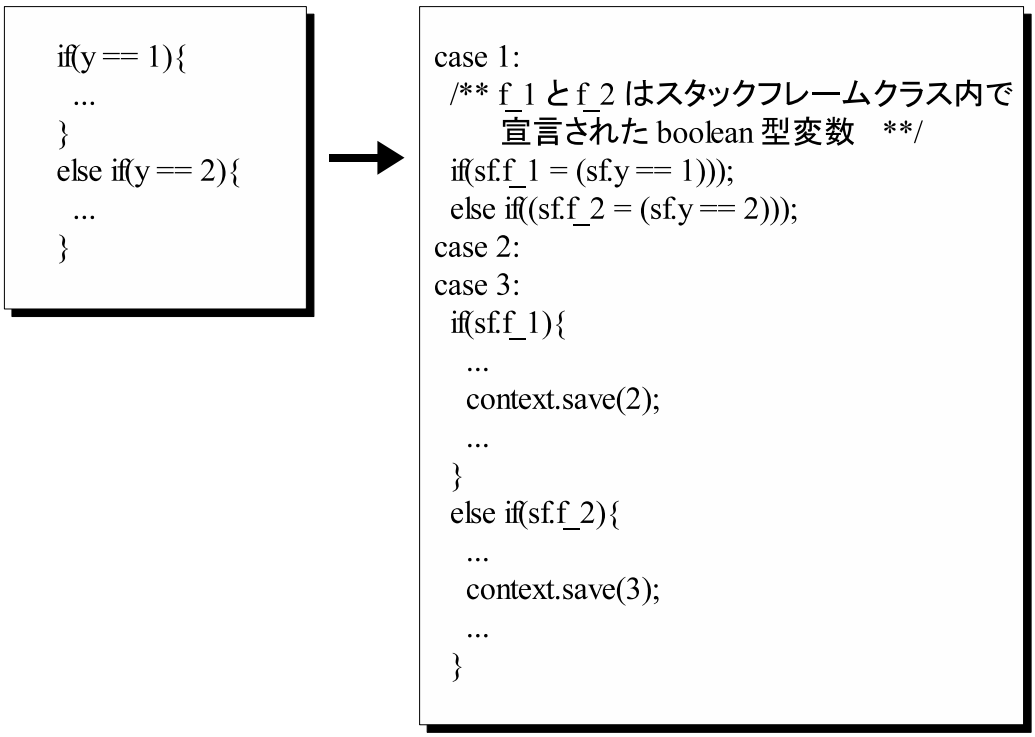


図 5.12: if 文の変換

Agent クラス: passport により定義された移動処理から生成されるクラス。このクラスは Aglets に似たコールバックメソッドを持っている。各メソッドは、ランタイムシステムからイベントを受け取ることで実行される。

AgentContext クラス: エージェントの移動に関わるデータを扱うクラス。移動に関わるデータとは、passport により定義された最大移動回数と、1 台のホストに滞在できる最大の時間や生成されたホスト情報などの固定データ、また、これまでの移動経路や移動回数などの移動により蓄積されたデータである。

5.5.2 移動処理実行の流れ

5.3 節で述べたように、移動ルールクラスは StartAgent クラスによりインスタンス化される。Agent クラスはイベントを受け付ける専用の Listener インターフェー

スを持ち、ランタイムシステムから通知されるイベントによりメソッドを実行する (図 5.13 参照)。

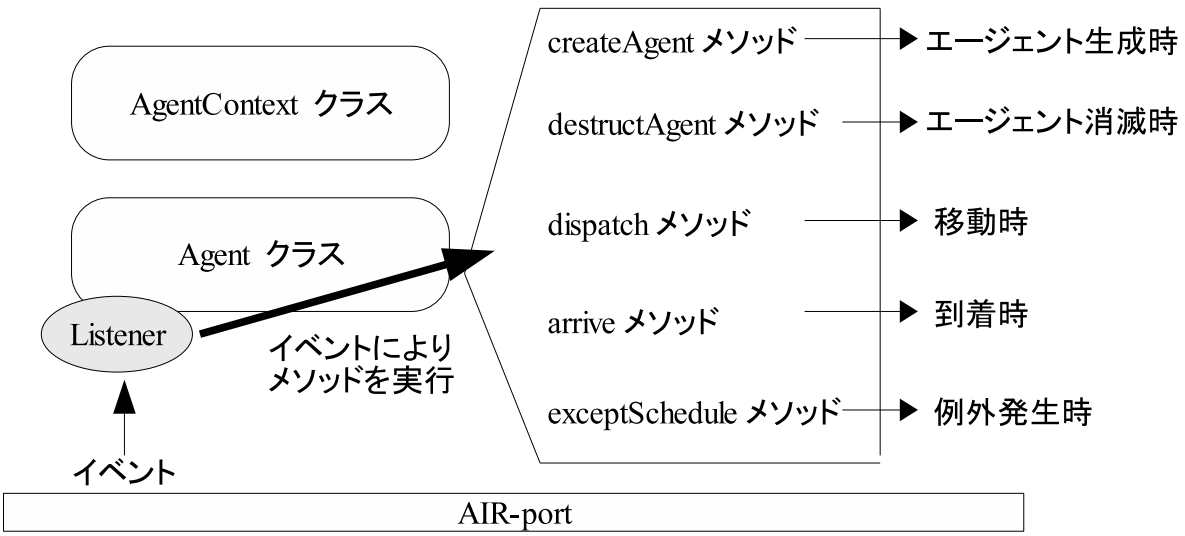


図 5.13: 移動処理の流れ

第 6 章

評価

6.1 利用例

ここで、本システムを利用して既存 Java プログラムをエージェント化した場合の例を示す。そして、その passport により追加された記述量や記述の理解の容易さについて考察する。利用するプログラムとして、モンテカルロ法シュミレーションプログラムを用いる (ソースコードは付録 B の図 B.2、図 B.1 参照)。

作成するエージェントを以下に示す。

- 移動をするたびにローカル処理を実行し、ローカル処理終了時に次のホストへの移動をする。それを 10 回繰り返し、結果を出発元ホストへ出力する。
- 移動経路はローカル処理終了時に最も CPU 使用率が低いホストとする。
- 例外として、ユーザの要求、ネットワークの断線の場合の処置を定義する。

6.1.1 考察

passport による追加コードは 44 行となった。この利用例の場合、エージェント開発者はこの passport の追加コードのみを記述すれば良い。ローカル処理に関する実行状態保存のための機能追加などは、トランスレータにより自動的に挿入される。既存モバイルエージェントシステムでは、ローカル処理に特定のコードを手動

で埋め込んだり、Agrets のようにコールバックメソッドに処理を対応付ける必要があったりする。passport では、そのようなローカル処理に対する修正は必要ない。

また、移動ルールを修正する場合も Java 言語ではフレームワークを定めるなどしない限り、修正箇所が明確にならない場合が多い。passport では、FIELD の言語構造を用いることによって、全ての処理の記述箇所が明確である。そのため、ローカル処理を全く熟知していない管理者でも、エージェントの移動処理を管理することができる。

ただし、エージェントに特化したプログラム (ローカル処理に依存したプログラム) は、passport による記述が難しい。たとえば、ローカル処理内のデータなどに依存した移動を定義するには、Java 言語で記述されたローカル処理の記述と、passport で記述された移動処理部分の記述をどちらも考慮してしなくてはいけない。そのため、開発者への負担が増え、また移動スケジュールも複雑になる。このことから、複雑な移動処理が行われるエージェントプログラムでは、必ずしも本システムが適切であるとは言えない。

6.2 評価実験

ここでは本システムの性能評価をするために行った実験とその結果について述べる。実験は、本システムのトランスレータによる変換前と変換後のそれぞれにおいて、処理速度の低下と実行ファイル容量の増加を計測する (実験環境は表 6.1 参照)。変換後の実行ファイル容量とは、エージェントが含む全てのクラスファイルと、シリアライズされた実行状態の保存データである。ただし実行状態の保存データの容量は実行中の最大値とする。対象となるプログラムは次の 2 種類とし、10 回の測定の平均を測定結果とする。(ソースコードは付録 C の図 C.1、図 C.2、図 C.3、図 C.4 参照)

プログラム 1 : while 文を含むプログラム。変換後は、while 文の内部で save メソッ

ドを繰り返し実行することになる。このプログラムの計測によって、実行状態保存における処理速度の低下を調べる。

プログラム 2 : プログラム内で再帰関数を呼び出すプログラム。変換後は各再帰ごとにコンテキストクラスとスタックフレームクラスが生成されることになる。このプログラムの計測によって、スタックを何層にも積んだ場合の処理速度の低下を調べる。また状態保存クラスが何度もインスタンス化された場合の保存データの増加を調べる。

表 6.1: 実験環境

CPU	Pentium4 1.7GHz
メインメモリ	256MB
OS	TurboLinux 10 Linux version 2.6.0–test5_2

表 6.2: プログラム 1

	変換前	変換後
処理速度	0.021 秒	2 分 11.275 秒
容量	865byte	3252byte

6.2.1 考察

表 6.2 と表 6.3 に実験結果を示す。表からもわかるように、変換前のプログラムに比較して変換後は極端に処理速度が低下する。これは今回の実装では実行が保

表 6.3: プログラム 2

	変換前	変換後
処理速度	0.001 秒以下	4 分 30.556 秒
容量	840byte	24036byte

存位置に達するたびに、実行状態データをファイルに保存するためである。また、この実験から本システムを用いるとスタックデータが通常の約 6 倍の容量を取ることが分かった。そのためスタック階層が深くなるような再帰プログラムを動作させた場合、ガベージコレクションによる処理速度の低下もある。また、スタックオーバフローを起こしやすいという問題も発生する。

実行ファイル容量については表 6.3 で見られるように、再帰プログラムを利用したプログラムで大きな増加が見られる。これは、スタックが深く積まれていった場合では、実行状態の保存データが爆発的に増加することを示している。

このような問題に対しては以下のような解決方法がある。

- 処理速度の低下の問題の解決策として、ファイルへの書き込みを、一定時間単位で行う方法が考えられる。実行が長時間に及ぶシミュレーションプログラムでは、ループ文で処理が繰り返されるたびにファイル書き込み処理がおこると、そのオーバヘッドが深刻になる。また、逐次実行状態を保存する必要性は低い。今回、プログラム 1 とプログラム 2 においてファイル書き込みを終了時のみ行われるようにして、同じように実験を行った。表 6.4 の測定結果が示すようにどちらのプログラムも処理速度が大幅に改善され、変換前の処理速度と比較しても大差がない。この結果から、時間や回数によってファイルへの書き込み制限機能を付加させることによって、オーバヘッドは改善可能であることがわかる。

- スタックの増加による処理速度の低下と保存データの増加の解決策として、再帰するメソッドでは、実行状態保存を実行しないことが考えられる。スタックが何層にも積まれた場合、保存データの増加は避けられない。また、メソッドを抜ける場合のガベージコレクション処理についても、避けることができない。再帰するメソッドを呼ぶ時に、実行状態保存を抑制すると、スタックのデータ量を減少できるが、再帰メソッドにおける実行状態を保存することはできない。

表 6.4: ファイル書き込みを制限した場合の測定結果

	処理速度
プログラム 1	0.029 秒
プログラム 2	2.119 秒

6.3 システムの評価

利用例と評価実験から、本システムでは以下のようなプログラムに適することがわかる。

- すでに元になる Java コードが存在するプログラム
- 開発者による複雑な移動処理の定義を必要としないプログラム
- 再帰的な処理が少ないプログラム

本システムは、既存 Java プログラムの資産の再利用を容易にすることを重視したシステムである。そのため、すべてのプログラムが本システムに適するとは言

えない。しかし、複雑な移動処理を行ったり、何度も再帰処理が行われたりするプログラムは、例外が発生しやすく、一般的にエージェントプログラムに適していないと考えられる。そのため、本システムの利点は十分に有効であると評価する。

第 7 章

今後の課題

この章では、今回の設計と実装で考慮外とした主なエージェント機能について論じ、それらを実現するための今後の課題を述べる。

7.1 処理速度の改善

第 6 章で述べたように、本システムの変換方法を用いるとプログラムの処理速度は大幅に増加する。しかし、状態保存に要している時間はメモリの内容をファイルに書き出す時間が支配的である [11]。実験でも示したように、ファイルへの書き出し回数、あるいは容量を抑制できれば実行速度を改善できる。このようなファイル書き込みの制限を passport により可能にし、開発者または管理者が任意に操作できるようにしたい。

7.2 マルチスレッドへの対応

本研究の実行状態保存機構ではスレッド状態の保存を対象としていない。なぜなら今回用いたソースコードの変換手法では、同期や割り込みによって停止しているスレッドなどの保存や復旧するには、複雑なコード変換が必要になってしまうためである。そのためプログラム自体が肥大化し、またローカル処理の処理速度の増加が避けられない。そこで、このようなマルチスレッドプログラミングへ対応するための新しい手法の考案が必要となる。

7.3 マルチエージェントへの対応

今回は論点としなかったモバイルエージェントの利点のひとつに複数エージェント間の自律的な協調性があげられる。複数のエージェントが協調することによって、ひとつの処理を分散実行したり最適な移動経路を伝達し合ったりすることができる。このようなマルチエージェントシステムを実現するためには、エージェント間通信の開発が必要となる。そのため今後の課題として、passport にエージェント間通信のための記述法を付け加えたい。

第 8 章

まとめ

本研究では、既存 Java プログラム資産を有効に活用することを重視したモバイルエージェントシステムの設計と実装を行った。今までのモバイルエージェントシステムでは、ローカル処理部分を独立して記述することが難しいため、既存のプログラムを容易にエージェント化できないという問題があった。本研究では、これらの問題を解決するため、エージェントプログラムから移動処理を移動ルールとしてローカル処理と分離する手法を提案した。また、移動ルールの定義言語として、passport を設計した。第 6 章の利用例で示したように passport を利用することで、既存のプログラムを修正することなく利用することができる。また、Java 言語では表現が難しかった移動経路や依存関係の処理をネットワーク構成を意識することなく記述できる。

また本研究では、移動ルールを用いたモバイルエージェントシステムの評価を行うため、エージェントランタイムシステム AIR–port を実装した。AIR–port を利用することにより、開発者に意識させることのない、自動的な実行状態の保存が可能になる。

さらに、本システムの性能評価をするために変換後のオーバヘッドやファイルサイズの増加の測定する実験を行った。実験では、コード変換により性能劣化が著しく出る場合があり、すべてのプログラムに有効ではないことが分かった。しかし、ファイルへの書き込みを一定時間ごとに制限するなどといった改善策を示すことができた。

謝辞

本研究を遂行するにあたっては、いろいろな方々にお世話になりました。

まず、指導教官の多田好克先生には日頃から熱心なご指導、そしてご鞭撻を賜わりました。また、ご多忙中にもかかわらず論文の草稿を丁寧に読んで下さり、大変貴重なご助言をいただきました。また、本研究室の安田絹子先生には本研究を進めるにあたり広く詳細な御指導、ご協力を賜りました。ここに厚く御礼申し上げます。

そして、本研究が行なえたことは、研究方針や方法論について議論をしてくださった、福田伸彦さん、佐藤喬さんら博士後期過程のみなさんのおかげです。また、二年間の研究生活を共におくった佐々木直志さん、城勘友秀さん、そして多田研学生諸氏とは互いに励まし合い研究を進めることができました。最後に、これらの皆さんに感謝いたします。

参考文献

[1] Lange, D. B. Oshima, M.: “Programming and Deploying Java Mobile Agents with Aglets”, Addison- Wesley, 1998.

[2] Voyager core package technical overview, 1997. ObjectSpace Inc. Available at http://www.objectspace.com/Voyager/.

[3] 中澤仁，徳田英幸: “ホスト透過型オブジェクト移送システム Mogul の実現,” 情報処理学会論文誌, 40 巻, 6 号, pp2573－ 2584, 1999

[4] 浅野貴史, 脇田建, 佐々政孝: “スレッド移送のためのバイトコード上の変換技法,” 第 3 回プログラミングおよび応用のシステムに関するワークショップ (SPA ’2000) 論文集, 浜松, 日本ソフトウェア科学会.March 2000.

[5] Torsten Illmann,Frank Kargl,Michael Weber,Tilmann Kruger: “Migration of Mobile Agents in Java: Problems, Classification and Solutions,” Proceedings of MAMA’00, Wollongong, Australia, December 2000.

[6] Sekiguchi, T., H. Masuhara, and A. Yonezawa: “A simple extension of java language for controllable transparent migration and its portable implementation,” Coordination ’99, 1999.

[7] 佐藤一郎: “モバイルエージェントの経路記述と選択機構”, 情報処理学会論文誌，vol.44，no.6, pp.1473-1482, June 2003.

[8] Torsten Illmann,Frank Kargl,Michael Weber,Tilmann Kruger: “Migration of Mobile Agents in Java: Problems, Classification and Solutions,” Proceedings of MAMA’00, Wollongong, Australia, December 2000.

[9] Kazuyuki Shudo and Yoichi Muraoka: “Asynchronous Migration of Execution Context in Java Virtual Machines,” Future Generation Computer Systems, Elsevier Science, Volume 18, Issue 2, pp.225-233, October 2001.

[10] Ohsuga, A., Nagai, Y., Irie, Y., Hattori, M., and Honiden, S.: “Plangent: An Approach to Making Mobile Agents Intelligent,” IEEEInternet Computing, Vol. 1, No. 4 , pp. 50-57 1997.

[11] 西岡, 堀, 手塚, 石川, “クラスタにおけるコンシステントチェックポイントの実現,” 並列処理シンポジウム JSPP’99, 情報処理学会、pp. 229 – 236、1999.

付録 A

標準関数と特殊シンボル一覧

表 A.1: 標準関数一覧

リスト操作のための関数	
car	リストの第一要素を取り出す
cdr	リストの第一要素以外を取り出す
append	指定したクラスを置き換える
reverse	リストを逆順にする関数
length	リストの長さを取得する関数
last	リストの最後のコンスを取得する関数
list	リストを作成する関数
+	数値の加算を行う関数
–	数値の減算を行う関数
*	数値の積算を行う関数
/	数値の除算を行う関数

移動処理のための関数	
init–agent	エージェントの生成関数
sequence	与えられた条件により集合を整列する関数
set–schedule	スケジュール作成関数
set–stdout	標準出力の出力先ホストを指定する関数
replace–class	エージェントの実行前に同盟のクラスを置換する関数
print	指定したホストへ出力を送る
with	指定したファイルやディレクトリを移動メンバに加える
without	指定したファイルやディレクトリを移動メンバから外す
stay-away	スケジュール外の移動をするための関数
stay-in	実行状態をデータ化し現在のホストに留まるための関数

集合操作のための関数	
subset	与えられた条件から部分集合を作成する関数
quote	与えられた要素から集合を作成する関数
uni	与えられたデータ型の全体集合を取得する関数
!	補集合を計算する
+	和集合を計算する
–	差集合を計算する
*	積集合を計算する

表 A.2: 特殊シンボル一覧

環境状態定義のためのシンボル	
OS	OS の環境定義を行うためのシンボル
CPU	CPU のスペックによる環境定義を行うためのシンボル
CPUUSAGE	CPU 使用率による環境定義を行うためのシンボル
TRAFFIC	トラフィックによる環境定義を行うためのシンボル
AGENT	エージェントによる環境定義を行うためのシンボル
場所を表現するためのシンボル	
WINDOWS	OS が Windows であるホスト
LINUX	OS が Linux であるホスト
SOLARIS	OS が SOLARIS であるホスト
BIRTH	エージェントの生成された場所
ANYWHERE	移動可能なホスト中のある 1 つのホスト
EVERYWHERE	移動可能な全てのホスト
HERE	現在のホスト
NEXTPLACE	スケジュール上の次のホスト
条件定義のためのシンボル	
POINT	ローカル処理内の実行位置を示すシンボル
DISCONNECTION	ネットワークが断線した状況を示すシンボル
USER	ユーザが移動要求を出した状況を示すシンボル

付録B

利用例のサンプルコード

```
1 public class TestAgent{
2
3   double pai;
4   long NUM;
5
6   public TestAgent(){
7     NUM = 1000000;
8   }
9
10   boolean calMonteCarlo() {
11     double x,y;
12     x=Math.random();
13     y=Math.random();
14     if ((x*x+y*y)<1)
15       return true;
16     else
17       return false;
18   }
19
20   public static void main(String args[]) {
21     TestAgent p = new TestAgent();
22
23     int i,count;
24     count = 0;
25     for(i = 0;i < p.NUM;i++)
26       if(p.calMonteCarlo())
27         count++;
28     p.pai=(double)4 * count/p.NUM;
29     System.out.println("pai = " + p.pai);
30   }
31 }
```

図 B.1: サンプルプログラム (passport コード)

```
 1 Creation {
 2   (init-agent "TestAgent" 60 10)
 3 }
 4 Schedule {
 5   (set-schedule schedule1
 6     (sequence CPU 10 refresh
 7       [subset ((OS = WINDOWS) & (OS = LINUX))
 8         [- [uni area] BIRTH]])
 9     (set-point POINT static))
10   (travel PATROL schedule1)
11 }
12 Departure {
13   (route-table
14    (to LINUX {
15      (print BIRTH "move to Linux")
16    })
17    (to WINDOWS {
18      (print BIRTH "move to Windows")
19    }))
20 }
21 Arrival {
22   (route-table
23    (from EVERYWHERE {
24      (set-stdout BIRTH);
25    }))
26 }
27 Dependence {
28   (route-table
29     (from EVERYWHERE {
30       (without "*")
31       (with "*.class")
32     }))
33 Exception {
34   (exception-table
35     (USER {
36       (stay-away NEXTPLACE)
37     })
38     (DISCONNECTION {
39       (stay-in)
40     })
41 }
42 Destraction {
43   (print BIRTH "finished!")
44 }
```

図 B.2: サンプルプログラム (Java コード)

```
AgentTest > pai = 3.141624004
AgentTest > pai = 3.141569036
AgentTest > pai = 3.141578656
AgentTest > pai = 3.141516296
AgentTest > pai = 3.141544116
AgentTest > pai = 3.141533444
AgentTest > pai = 3.1415648
AgentTest > pai = 3.141550104
AgentTest > pai = 3.141676764
AgentTest > pai = 3.141685056
AgentTest > finished!
```

図 B.3: サンプルプログラム 実行結果

付録C

評価実験に用いたJavaコードの変換

```
 1 class StateTest {
 2   public static void main(String[] args){
 3     StateTest st = new Test();
 4     st.test();
 5   }
 6
 7   public void test(){
 8     int x = 0;
 9
10     while(x < 100000)
11       x++;
12   }
13 }
14
```

図 C.1: プログラム 1:変換前

```
 1 import java.io.*;
 2 class StateTest implements Serializable{
 3   AgentData agentData;
 4   public StateTest(AgentData aData){
 5     agentData = aData;
 6   }
 7
 8   public static void doMain(){
 9     int _id = 0;
10     AgentData aData = new AgentData();
11     LoVariable0 lv = (LoVariable0)aData.push(_id);
12     StateTest st = new StateTest(aData);
13
14     switch(sData.getPnt()){
15     case 0:
16       aData.save(1);
17     case 1:
18       st.test();
19       aData.save(2);
20     case 2:
21     }
22     aData.pop();
23   }
24
25   public void test(){
26     int _id = 1;
27     LoVariable1 lv = (LoVariable1)agentData.push(_id);
28
29     switch(agentData.getPnt()){
30     case 0:
31       lv.x = 0;
32       agentData.save(1);
33     case 1:
34     }
35     while(lv.x < 10000){
36       switch(agentData.getPnt()){
37       case 1:
38         lv.x++;
39         agentData.save(1);
40       }
41     }
42     agentData.pop();
43   }
44 }
45
```

図 C.2: プログラム 1:変換後

```
 1 class StateTest {
 2   int count;
 3
 4   public static void main(String[] args){
 5     StateTest st = new Toto();
 6     st.count = 0;
 7     int i;
 8     for(i = 0;i < 20;i++){
 9       st.test();
10       st.count = 0;
11     }
12     System.out.println("OUTPUT ; finish");
13   }
14
15   public void test(){
16     if(count > 500){
17       return ;
18     }else{
19       count++;
20       test();
21     }
22   }
23 }
24
```

図 C.3: プログラム 2:変換前

```
 1 import java.io.*;
 2
 3 class StateTest implements Serializable{
 4   AgentData agentData;
 5   public StateTest(AgentData agentData){
 6     agentData = s;
 7   }
 8   int count;
 9
10    public static void doMain(){
11
12      int _id = 0;
13      AgentData aData = new AgentData();
14      LoVariable0 lv = (LoVariable0)aData.push(_id);
15      lv.st = new StateTest(aData);
16      lv.st.count = 0;
17      int i = 0;
18
19      switch(aData.getPnt()){
20      case 0:
21        aData.save(1);
22      case 1:
23        for(i = 0;i < 20;i++){
24          lv.st.test();
25          lv.st.count = 0;
26        }
27      case 2:
28        System.out.println("OUTPUT ; finish");
29      }
30      aData.pop();
31    }
32
33    public void test(){
34      int _id = 1;
35      LoVariable1 lv = (LoVariable1)stackData.push(_id);
36      switch(agentData.getPnt()){
37      case 0:
38        if(count > 500){
39          return ;
40        }else{
41          count++;
42          test();
43        }
44        agentData.save(1);
45      case 1:
46      }
47      agentData.pop();
48    }
49 }
50
```

図 C.4: プログラム 2:変換後