修 士 論 文

移動エージェントシステムにおける
例外処理の体系化に関する研究

指導教官　篠田 陽一 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

小林 史陽

平成 13 年 2 月 18 日

要 旨

移動エージェントシステムは、システム全体が大きくなり易く、その例外処理は、一般に複雑になる。そこで、本研究では、システム全体を統合した例外処理設計の方法論を確立することを目的とする。

本研究では、まず、例外を分析し、例外処理設計が難しくなる原因を検討する。そのために、例外の分類を行い、どの類の例外処理設計が難しく、また、それはなぜなのかを議論する。そして、それらを元に、エージェントシステムの例外について検討する。

次に、例外処理設計に関するシステム設計者の思考過程を抽出し、難しいと考えられる外部検出例外やユーザ発行例外に対する対処法設計の方法論を確立する。この際、本研究では、従来のソフトウェア設計論を参照し、その過程に沿って、例外分析が行えることを観点とした。

その方法論は、エージェントオブジェクト別の例外処理設計法と複数のエージェントオブジェクトに渡る例外処理設計法の２つに分類される（ただし、前者は後者のサブセットとなっている）。

エージェントオブジェクト別の例外処理設計法においては、対処法がそのエージェントオブジェクトに閉じているため、容易に機能別に例外処理を設計できる。この方法が適用できるのは、一般に、内部検出例外である。

次に、システム全体の再検討を要し、エージェントオブジェクトを追加する必要のある外部検出例外、およびユーザ発行例外について議論する。その方法の基本的なアプローチは、従来の設計論に従い、システムの機能分析から必要なエージェントオブジェクトを決定し、エージェントオブジェクト別の例外処理設計を適用することである。

最後に、いくつかの具体的な例を用いて、方法論の検討を行う。エージェントオブジェクト別の例外に関しては、送信元プレースの機能であるエージェントの移動の機能に関する例外分析を例にあげる。ユーザ発行例外に関しては、エージェント位置管理サーバを導入する必要のある帰還命令の例外分析を例に上げて議論する。

最後に、本研究の過程で考案された例外処理設計支援アプリケーションである必要機能出力システムに関する議論を加えた。

目 次

図 目 次

表 目 次

第 1 章

背景と目的

1.1 はじめに

新しいネットワークソフトウェアモデルとして、移動エージェント技術が注目されている。移動エージェント技術は、従来のクライアント- サーバモデルとは、実行主体であるプログラム（エージェント）が移動するという点において本質的に異なっている。

例えば、通信量が多く、かつ処理量が多い計算などに対して、エージェントが移動することにより、通信量を減らすことができる。また、エージェントの移動に伴い、コネクションを張り続ける必要もなくなる。

1.2 背景と目的

移動エージェント技術には、これらのメリットもあるが、いくつかの問題も内包している。本質的には、処理の進行が見えないことやエージェントに対するセキュリティ（プレースからエージェントを保護することが難しい）などがあげられる。その他に、エージェントが移動することによって、例外処理が複雑になることもあげられる。

一方、例外処理は、実用的なシステムを設計する上で必ず検討しなければならない。そこで、本研究では、例外処理が複雑になるという点に着目して、移動エージェントシステムの例外処理設計の方法論を確立することを目的とする。

本研究では、機能分割例外分析を提案し、機能分割例外分析のシステム全体の例外分析への応用について議論する。

なお、本研究のアプローチは、例外処理を設計する際に行うことの分析を通して、方法論の確立を目指すものである。

1.3　本論文の構成

1. ２章では、エージェントシステムに関して紹介し、本論文への導入とする。
2. ３章より本研究の内容に入る。まず、例外処理設計の定義を行う。また、例外の分類も行い、どの類の例外が難しいのかを検討する。
3. ４章で、本研究で提案する方法論を提案し、その概要を述べる。
4. ５章では、内部検出例外に関する例外処理設計について検討する。
5. ６章では、外部検出例外、およびユーザ発行例外に関する例外処理設計について検討する。
6. ７章では、本研究を行う過程で考案した例外処理設計支援アプリケーション「必要機能出力システム」について議論する。
7. ８章では、まとめと今後の課題について述べる。
8. なお、付録には、エージェントの定義とその特質について述べておいた。

第 2 章

移動エージェントシステム

本章では、本研究の対象となるエージェント技術の簡単な説明を行う。

2.1 エージェント技術

エージェント技術は、本来人工知能の研究者によって議論されてきた。一方、移動エージェントシステムは、その名の通り、「移動」に焦点がおかれることが多く、人工知能の範囲ではなくなる。本研究においても、その傾向を踏襲している。そこで、エージェント技術の理解には重要であるが、人工知能的で移動という観点からはずれるエージェントの定義やエージェントシステムの特質などは、付録に配置した。

2.2 エージェントシステムの分類

エージェントシステムにはさまざまなシステムが存在する。本節では、その分類を行い、対象システムを明確にする。

エージェントシステムは、その役割とモデルから、以下の３つに分類される。

インターフェースエージェントシステム ユーザインターフェースのエージェントであり、ユーザの補助などを行うことを目的とし、一般的に、ネットワークは利用しない（図 2.1）。

ネットワークエージェントシステム ネットワークを利用したエージェントであり、以下の２つに分類される。

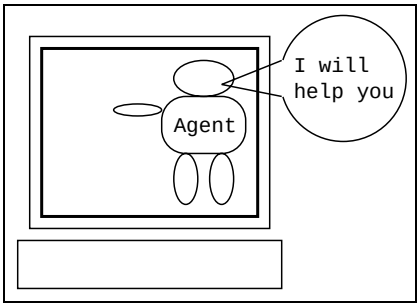

図 2.1: インターフェースエージェント

コンバーセーショナルエージェントシステム ネットワークエージェントで、エージェント間の通信により計算を行う。但し、エージェントは移動しない。ネットワークアプリケーションモデルは、クライアント-サーバモデルである [4]（図 2.2)。

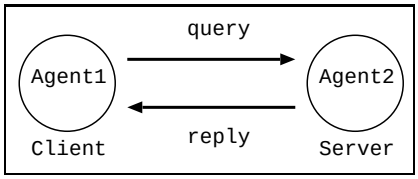

図 2.2: コンバーセーショナルエージェントシステム

移動エージェントシステム 実行主体であるエージェントが移動し、プレースとの通信により計算を行う。エージェントが移動するため、ネットワークアプリケーションモデルは、クライアント-サーバモデルではない。

2.3 移動エージェントシステム

本節では、本研究の対象である移動エージェントシステムの説明を行う。

2.3.1 移動エージェントシステムの目的、特徴

移動エージェントは、分散環境における１つのパラダイムである。従来のクライアント-サーバモデルとの違いは、実行主体が移動することである。移動エージェントシステムの主な目的は、通信量を減らすことにある。

実行主体が移動することによってもたらされるメリットは、以下の通りである。

- 通信量の減量（例えば、大量のデータをフィルタなどで減らし、通信する）
- コネクションの切断（処理に時間がかかる場合など）

2.3.2 移動エージェントシステムのモデル

移動エージェントシステムには、一般に、エージェント、送信元プレース、送信先プレースの３つのエージェントオブジェクトが存在し、エージェントが移動先プレースに移動し、移動先プレースとのインタラクションによって計算が進められて行く。そして、最終的に、エージェントは、送信元プレースに戻り、計算結果をユーザ（ユーザクライアント）に報告し、計算を終了する（図 2.3 参照）。

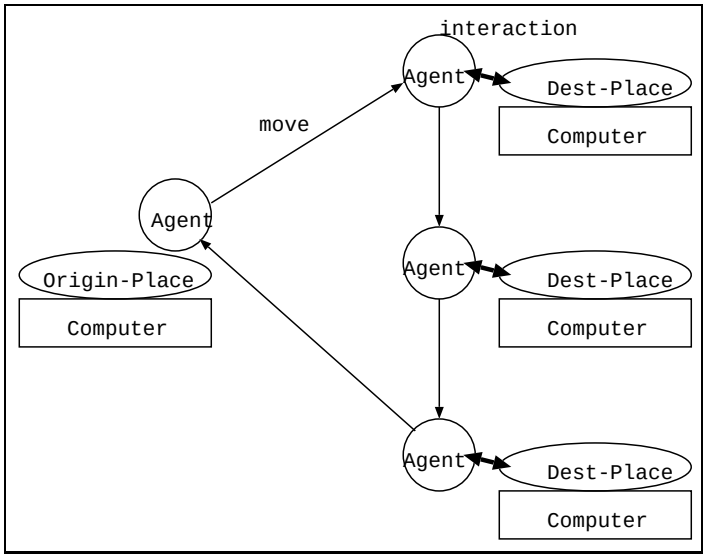

図 2.3: 移動エージェントシステム

エージェントの定義には、さまざまなものがある（付録 A 参照）が、本研究では、移動エージェントシステムを対象とするため、あまり抽象化せずに、

エージェント 移動したプレースとのインタラクションによって、与えられた処理を進める計算主体プログラム

と定義する。

2.3.3 さまざまな実存する移動エージェントシステム

現在既に、さまざまな移動エージェントシステムが実装されているが、それらの紹介は、参考文献 [7] を参照して欲しい。この文献には、Telescript, Aglets, kafka, Voyager, TACOMA, Ajenta, AgentSpace が紹介されている。参考文献 [7] 記載の移動エージェントシステム以外には、Beegent [14], Plangent [15], NOMADS [13] などがある。それぞれ参考文献をあげておく。

移動エージェントシステムと良く似た技術に、Process Migration という技術があるが、これは負荷分散を目的としている。Process Migration は、Strong Migration（実行状態を保存して移動する）についての技術である。

移動エージェントシステムにおいては、Weak Migration（実行状態を保存せずに、移動する）が一般的であるが、昨今、NOMADS [13] のように Strong Migration を実装したエージェントシステムも登場してきている[1]。

2.4 システムのレイヤ

移動エージェントシステムには、２つの意味があるので、本節で明確にしておく。１つは、フレームワークとしての意味であり、もう１つは、フレームワークを利用した実際のアプリケーションシステムである（図 2.4 参照）。本論文では、特に、フレームワークを利用した実際のアプリケーションシステムに対する例外処理の設計を対象とする。

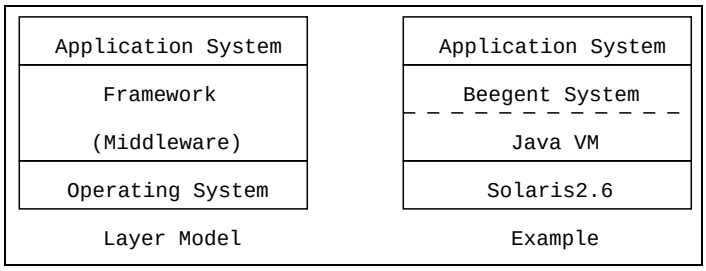

図 2.4: レイヤー

[1] このシステムは、他の多くのエージェントシステムと同様、Java の VM (Virtual Machine) 上で動作するが、Strong Migration を実現するために、VM を改変している。そのため、一般の VM との互換性はない。

2.5 エージェントオブジェクト

本論文では、エージェントオブジェクトを、フレームワーク上で実装されたソフトウェアと定義する。エージェントオブジェクトの例として、移動エージェントシステムの基本オブジェクトであるエージェント、送信元プレース、送信先プレースなどがあげられる。この他にも、例外処理やシステムに対する要求として導入されるバックアップサーバや位置管理サーバも含まれる。

第 3 章

例外の分析

本章では、本研究で扱う例外について議論する。まず、例外の分類を行い、これを元に、例外処理の難しさについて議論する。

3.1 例外処理設計

実用的なシステムを設計する上で必ず設計しなければならないことに例外処理があげられる。

本研究では、例外処理設計を、図 3.1 のように捉え、その定義を

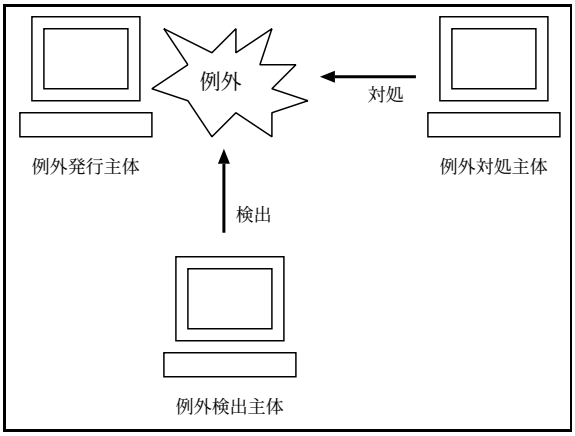

図 3.1: 例外処理設計

例外処理設計 どのオブジェクトがどのような例外を発行し、どのオブジェクトがどのよ

パラメータの名前	内容	例
発行主体	どのオブジェクトが、	エージェント
発行された例外の種類	どのような例外を発行し、	クラッシュ
検出主体	どのオブジェクトが	バックアップサーバ
検出の方法	どのように例外を検知し、	ポーリング
対処主体	どのオブジェクトが、	バックアップサーバ
対処の方法	どのように例外に対処するのか	バックアップの活性化

表 3.1: 例外処理設計のパラメータ

うにその例外を検出し、どのオブジェクトがどのようにその例外に対処するかを決定すること

とし、研究を進める。例外処理設計のパラメータを、表 3.1 にまとめた。

研究アプローチとして、まず、例外の検出主体を観点に例外を分類する。また、その検出主体に依存して例外に対する対処法設計の難しさが変化することを示し、研究対象を明確にする。

次に、例外処理処理設計の後者の部分、つまり「どのオブジェクトがどのように（例外に対して）対処するのかを決定する」ための手法について、次章以降で議論していく。

3.2 例外の分類

まず、例外処理設計の定義の検出部分について、検出主体という観点で例外を分類する。

内部検出例外 内部（自ホスト）で検出できる例外

外部検出例外 外部（ホスト）からのみ検出される例外

この他に、観点は異なる（観点は、例外の発行主体）が、

ユーザ発行例外 ユーザによって発行される例外

があり、分類に加える。

以下、それぞれの説明を行う。

名前	内容
SIGKILL	プロセスの終了（強制終了）
SIGBUS	バスエラー
SIGSEGV	メモリアクセスのセグメント例外
SIGTERM	プロセスの終了

表 3.2: Unix のシグナル

3.3 内部検出例外

内部検出例外とは、ホスト内部で検出される例外で、以下の２つに分類される。

関数検出例外 （実装言語の）関数[1] の返り値によって検出される例外である。C 言語の socket 関数 [6] を例に上げると、図 3.2 のようになる。図中の ERRORS は、大域変数 errno に代入され、perror 関数によって対応したメッセージを出力することが可能である。

当然であるが、関数の返り値の仕様に定義されていない例外は、仕様外となり、その関数からは検出できない。逆に、定義のある例外は、その関数の仕様であり、検出が容易である。対処法の設計も比較的容易である。

割り込み検出例外 もう１つの内部検出例外として、割り込みがあげられる。これは、コントロールプレーンが通常の関数とは別になっている。Unix では、シグナルが割り込みとして実装されている。シグナルとは、次のようなものである。別のコントロールプレーン（通常、OS）からシグナルを投げ、プロセスはそれを受けると、状態を保存し、指定された処理を行う。いくつかの Unix のシグナルを表 3.2 に紹介する。これら以外にも合わせて１５個のシグナルが存在するが、SIGALRM など例外検出とは目的が違うので割愛した。

3.3.1 Exception の実装

いくつかの言語では、例外（Exception）という概念が導入されており、関数の実行とは別のコントロールプレーンでこれを投げ（Throw）、関数実行系の方でそれを受け取る

[1] オブジェクト指向設計でいうメソッド、OS に対する API としてのシステムコール、およびライブラリ関数をまとめて、本論文では関数と呼ぶ。

```
int socket(int domain, int type, int protocol);
RETURN VALUES
        A -1 is returned if an error occurs.  Otherwise  the  return
        value is a descriptor referencing the socket.

ERRORS
        The socket() call fails if:

        EACCES  Permission to create a  socket  of  the  specified
                type and/or protocol is denied.

        EMFILE  The per-process descriptor table is full.

        ENOMEM  Insufficient user memory is available.

        ENOSR   There were insufficient STREAMS  resources  avail-
                able to complete the operation.

        EPROTONOSUPPORT
                The protocol type or the specified protocol is not
                supported within this domain.
```

図 3.2: C 言語の socket システムコールの例外

（catch）ことで例外の検出を行うものである。この方式の典型は、Java [5] である。Java の Socket クラスを例に上げ、図 3.3に示す。

本論文では、Java 言語におけるこの意味での例外（Exception）を単に Exception と示す。この類の例外も内部検出例外に分類される。

3.4 外部検出例外

これは、外部ホストから検出される例外である。この類の例外にカテゴライズされる例外に、送信先ホストのクラッシュなどによるエージェントの消滅が上げられる。この類の例外は、内部からは検出されない[2]。なお、エージェントのクラッシュに対する対処法として、バックアップエージェント [7] があげられる。

3.5 ユーザ発行例外

ユーザが発行する例外であり、システムから見れば例外であるが、設計時には、システムへの要求として捉えられる。第 6 章で議論する帰還命令（図 6.1）などが、ユーザ発行例外に分類される。この類の例外に対する対処は、システム全体に広がることが多く、複合的な例外と言える。

3.6 処理設計の難しい例外

以上のように例外を３つに分類して来たが、このうち外部検出例外とユーザ発行例外に対する対処法の設計が、内部検出例外と比較して難しい。本節では、その理由を議論する。

この類の例外に対する例外処理設計が難しくなる要因を、いくつか以下に列挙する。

- 内部検出例外は、実装言語の関数の仕様の一部であるため、その把握が容易であるが、外部検出例外やユーザ発行例外はそうではないため、その設計の必要性が設計段階で明確になりにくい。
- 内部検出例外と違い、検出方法から設計しなければならない。

[2]送信先プレースの暫時ダウン（Graceful Down）ならば、内部検出も可能な場合がある。

```
public final class Socket extends Object
 {
  // Public Constructors
  public Socket(String host, int port)
                throws UnknownHostException, IOException;
  public Socket(String host, int port, boolean stream)
                throws IOException;
  public Socket(InetAddress address, int port) throws IOException;
  public Socket(InetAddress address, int port, boolean stream)
                throws IOException;

  // Class Methods
  public static synchronized void
                setSocketImplFactory(SocketImplFactory fac)
                throws IOException, SocketException;

  // Public Instance Methods
  public synchronized void close() throws IOException;
  public InetAddress getInetAddress();
  public InputStream getInputStream() throws IOException;
  public int getLocalPort();
  public OutputStream getOutputStream() throws IOException;
  public int getPort();
  public String toString();
 }
```

図 3.3: Java の Socket クラスの例外（Exception）

- 外部検出例外やユーザ発行例外に対する対処法を導入した際に、エージェントオブジェクトが非常に増えやすい（バックアップサーバや位置管理サーバなど)。また、オブジェクトが増えると、増えたエージェントオブジェクトに対する例外を考慮する必要が出て来る。一般に、オブジェクトが少ないほど、例外設計は容易である。

これらのことは、移動エージェントシステムに限らず言える。

3.7 移動エージェントシステムの例外

移動エージェントシステムについて見てみると、実行主体が移動するという特性により、外部検出例外やユーザ発行例外が増えることが言える。

いくつかの例を上げて説明する。

エージェントのクラッシュ 外部検出例外である。エージェントのクラッシュに対するアプリオリな解決法として、バックアップ [7] が考えられるが、このためにはバックアップサーバが必要となる。これは、エージェントオブジェクトが増えることを意味し、対処は難しい。

帰還命令 ユーザ発行例外である。送り出したエージェントに対して、（帰って来いという）帰還命令を意味する。これには、エージェントの位置管理サーバが必要となり、対処も難しくなる。

本研究は、主に、移動エージェントシステムの外部検出例外とユーザ発行例外を扱うためになされた。

第 4 章

例外分析の手順

本章では、エージェントシステム設計者が、システムに例外処理を導入する際に考える必要のあることについて、従来からある設計論を参照して議論する。また、本研究で提案する例外分析法についての概要も述べる。

4.1 従来からの設計論

本研究で述べる例外分析法は、ソフトウェア工学において長く使用されている設計論 [10] を参照し、その流れからどのように効率的な例外分析が行えるかについて研究を行ったものである。その中でも、特に、機能分析を元にした方法論であるが、機能分析はソフトウェア設計において最も基本的な要素であり、どの設計論でも行われる [11] [12]。

図 4.1 に従来からの設計論[1]を示す。

例外分析も、図 4.1 の従来からの設計論と同様、設計段階と実装段階に分けられる。設計段階では、実装言語に依存しない設計を行う。実装段階では、逆に実装言語に依存した具体的な設計を行う。

4.2 システム全体を統合した例外分析

設計時の例外分析と実装時の例外分析を合わせて、システム全体を統合した例外分析と名付ける。図 4.2 に全体像を示す。この方法論は、最初（1）に戻ることが場合によっては考えられるので、流れのモデルはスパイラルモデルとなる。

[1] この設計論は、ウォーターフォールモデルにも使用されており、最も基本的なモデルである。

4.3 設計段階の例外分析

設計段階におけるトップダウンの例外分析の手順について以下に述べる。

1. エージェントシステムへの要求定義
   エージェントシステムに対する要求を明確にするとともに、システムに関してどのような例外が可能で、どのような例外に対処するのかを明確にする。

2. エージェントシステムの必要機能の分析
   エージェントシステムに要求を導入する上で、エージェントシステムに必要とされる機能を分析する。

3. 必要エージェントオブジェクトの分析
   エージェントシステムに必要となるエージェントオブジェクトを分析する。

4. 機能分析
   それぞれのエージェントオブジェクトに必要な機能を分析する。

5. 例外分析
   その機能に可能な例外を分析する。

6. 処理法の考案
   その例外に対する対処法を考案する。

図 4.2 に示すように、ここまでが設計段階である。
この後に、実装言語の決定を行い、実装フェーズに入っていく（先に決定されている場合も多い）。

4.4 実装段階の例外分析

実装フェーズにおける例外分析について、以下に述べる（この時、実装言語は決定されていることを仮定する。）

1. 機能と関数のマッピング
   それぞれのエージェントオブジェクトのそれぞれの機能が、実装言語において、どの関数からなっているのかを調査する。

2. 関数の返り値と例外のマッピング
   それぞれの関数の返り値によって、どの例外が検出できるのかを調査する。この時点で、５で検討された対処しなければならない例外が、内部検出例外か、外部検出例外であるかが決定される。

3. 内部検出例外以外の例外があった場合は、１へ
   外部検出例外をどのように検出し、どのように対処するのかを設計するために１へ戻る。

本研究では、この手順に基づく例外分析を提案する。以降に例を示しながら、詳細を議論する。５章で、オブジェクト別の例外分析について議論し、６章で、全体を統合した例外分析について議論する。

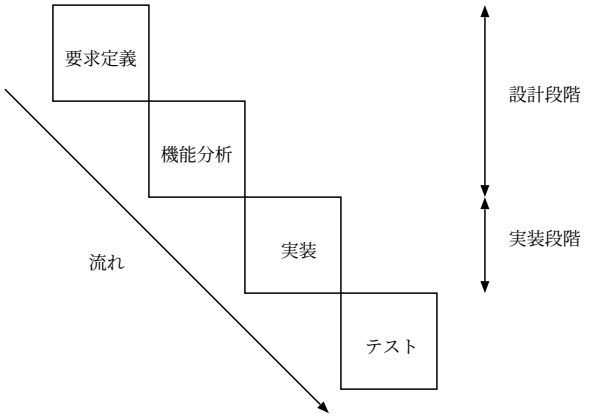

図 4.1: 従来からの設計論

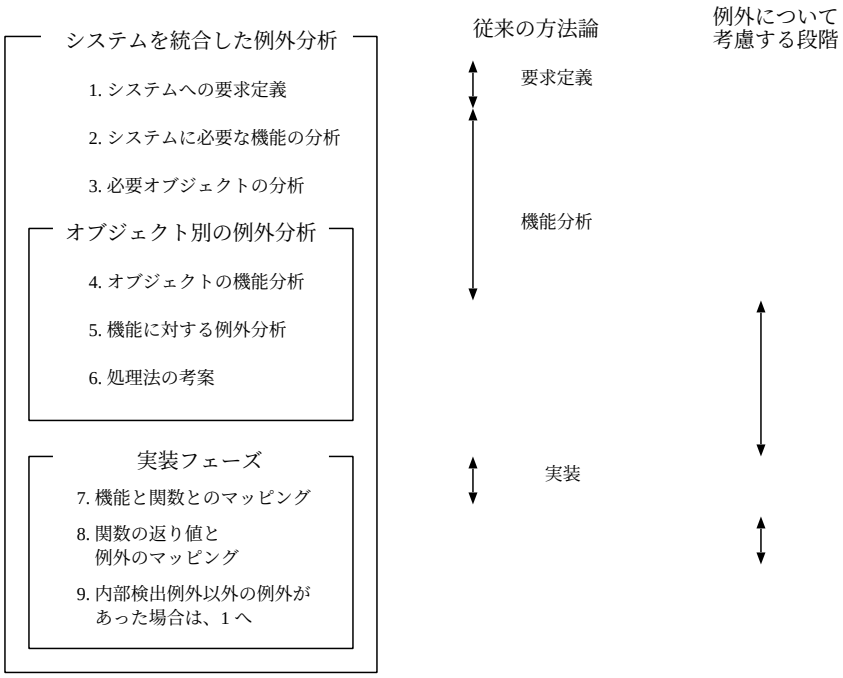

図 4.2: システム全体を統合した例外分析と従来の設計論との対応

第 5 章

エージェントオブジェクト別の例外分析

図 4.2 におけるオブジェクト別の例外分析が本章に該当する。

本章では、移動エージェントシステムの最も基本的なオブジェクトである、エージェント、送信元プレース、送信先プレースの例外分析について検討する。

5.1 単純なシステムの分析

対象システムは、図 5.1 のような最も単純なシステムを想定する。ただし、図 5.1 における 'ps' command とは、Unix システムにおいてプロセスの状態を表示するコマンドとする。

このような単純なシステムにおいては、図 4.2 における１～３は、基本エージェントオブジェクトのエージェント、送信元プレース、送信先プレースの基本機能からアプリオリに決定される。従って、図 4.2 の４番目である、それぞれのエージェントオブジェクトの機能分析から分析を始めることが可能である。本章では、エージェント別オブジェクトの例外分析に焦点を当てたいので、このような例を選択した。

5.2 機能分析

このシステムにおいて、それぞれのエージェントオブジェクトの機能の分析を行う。機能分析を行った結果を以下に示す。

- エージェントの機能：
  - メッセージの送受信

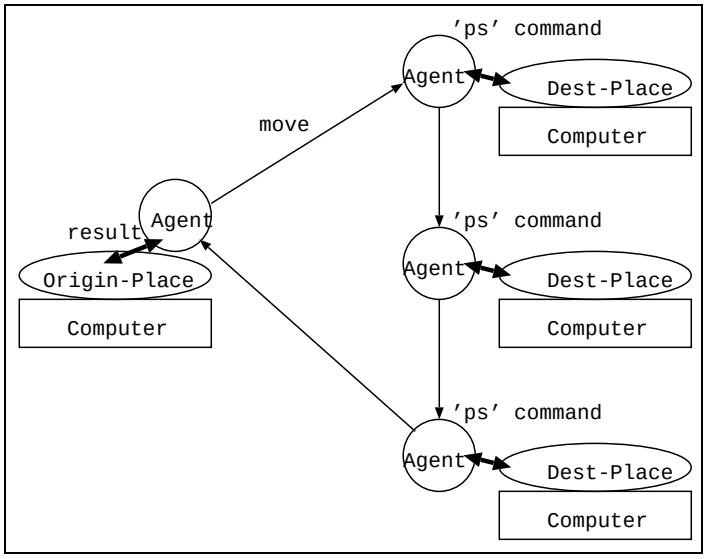

図 5.1: 対象システムの例

- 送信元プレース：
  - エージェントの移動
  - エージェントの生成
  - エージェントの受け入れ
  - メッセージの送受信
  - ユーザ、あるいはクライアントとのインターフェース
- 送信先プレース：
  - エージェントの受け入れ
  - エージェントの生成
  - メッセージの送受信
  - 'ps' コマンドの処理機能

この大別された機能をさらに詳しく分析する。ここでは、送信元プレースの機能である「エージェントの移動」について取り上げ、その結果を、表 5.1 に示す。

大別された機能	詳細な機能	関数
エージェントの移動	1. エージェントスレッドのサスペンド	Thread.suspend();
	2. エージェントのシリアライズ	ObjectOutputStream().write();
	3. シリアライズされたエージェントの送信	Socket(); OutputStream.write();

表 5.1: 「エージェントの移動」機能の分析

5.3 機能からの例外分析

次に、図 4.2 において５に移動し、分析された機能から、例外を分析する。ここでは、前節と同様にエージェントの移動という機能に対して、どのような例外が発生するかを検討してみる。前節において、機能が分割されているので、それぞれ、エージェントスレッドのサスペンドができない、シリアライズができない、送信ができないなどの例外が考えられる。

次に、図 4.2 において６に移行する。先に分析された３つのそれぞれに対して、その例外に対する対処を設計するが、対処法の設計については、本論文では議論しない。

5.4 機能と関数のマッピング

ここからは、実装段階に入る。

実装言語が決定されると、表 5.1 に示したように、分析された機能と実装言語における関数のマッピングが可能となる。この際、関数の仕様から検出される例外とそれ以外の例外が区別される。つまり、内部検出例外と外部検出例外が決定される。

一例として、Java 言語において、オブジェクトをシリアライズする関数 Serialize を図 5.2 に示す。この関数においては、NotSerializableException （および、それを継承している Exception）のみが検出される[1]。それ以外の例外は、検出されない（仕様外となる）。

[1]この例外は、オブジェクトが Interface Serializable を継承していない時に Throw される。

```
public static byte[] Serialize(Object obj)
throws NotSerializableException {
  try {
    byte[] data = null;
    ByteArrayOutputStream byteout = new ByteArrayOutputStream();
    ObjectOutputStream objout = new ObjectOutputStream(byteout);
    objout.writeObject(obj);
    objout.flush();
    data = byteout.toByteArray();
    objout.close();
    byteout.close();
    return data;
  }
  catch (IOException e) {
    throw new NotSerializableException(e);
  }
}
```

図 5.2: オブジェクトをシリアライズする関数

5.5 まとめ

本章では、エージェントオブジェクト別の例外分析法について、図 4.2 に従って、実際の例を上げながら、検討してきた。本章で上げた例は、内部検出例外に対する対処法の設計であり、比較的単純である。

次章では、より複雑な外部検出例外、およびユーザ発行例外に対する対処法の設計について論じる。

第 6 章

システム全体を対象とした例外分析

本章では、本研究の目的となる「システム全体を対象とした例外処理設計」について議論する。

6.1 システム全体を対象とした例外分析

本章では、例外処理に必要となる機能が複数のエージェントオブジェクトに渡るような例外を「システム全体を対象とした例外分析」と名付け、この類の例外について議論する。

この類の例外分析を行う必要がある例外は、外部検出例外やユーザ発行例外である。外部検出例外には、エージェントのクラッシュに対する対処法であるバックアップサーバ [7] などが上がられる。

6.2 帰還命令

ユーザ発行例外である「帰還命令」や「キャンセル命令」などもこの類の例外分析法が適用される。帰還命令とは、送り出したエージェントに対して、「途中経過を持って戻って来い」という命令を言う（図 6.1)。

本章では、この帰還命令を例に取り、システムを統合した例外分析について検討する。

6.3 システムに必要とされる機能

例えば、図 4.2 における 1 の段階で、帰還命令がシステムへの要求としてあったことを仮定する。

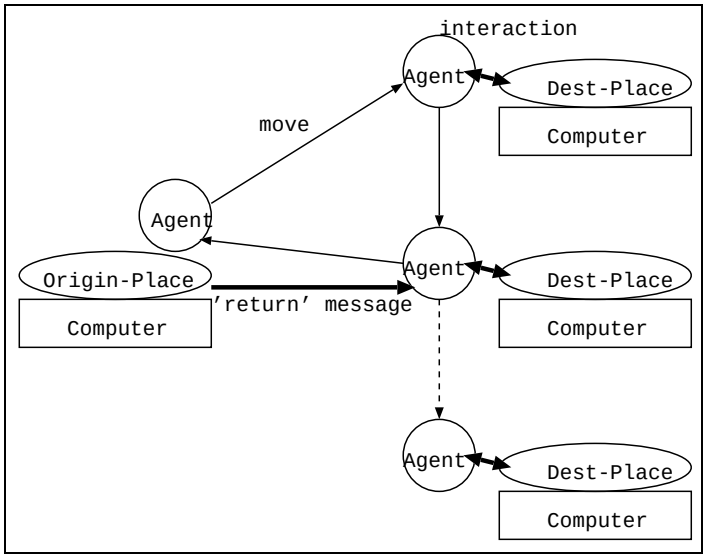

図 6.1: 帰還命令

次に（図 4.2 における２）、帰還命令をシステムに導入する上で、システムに導入しなければならない機能を分析してみると、以下のようになる。

- エージェントの割り込み機能
  帰還命令を受け取った際、計算を中断し、その処理を先行させるために必要となる。
- エージェントの位置管理機能
  帰還命令を送信するためには、エージェントの現在の位置を把握することが必要である。

本節では、「エージェント位置管理機能」に焦点を当てて議論する。

6.4 位置管理サーバ

図 4.2 における３の必要オブジェクトの分析で、位置管理サーバ（図 6.2）が導入されることが考えられる。位置管理サーバは、エージェント位置管理機能のための最もアプリオリな方法であるが、本節では、これをこのまま採用する。その他の方法については、後に議論する。

エージェント位置管理オブジェクトを導入し、その機能を分析すると（図 4.2 における４）、以下のようになる。

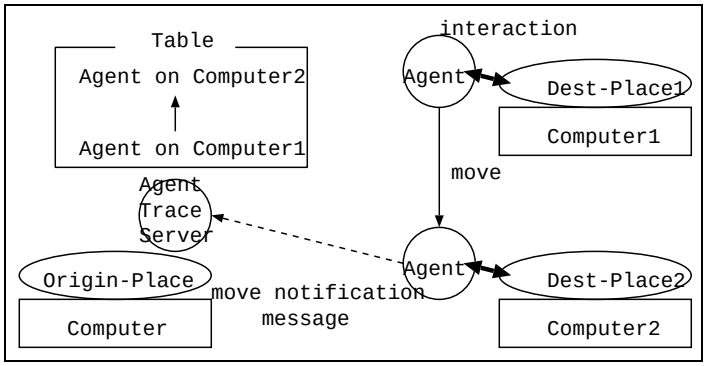

図 6.2: 位置管理サーバ

- メッセージ（問い合わせメッセージと更新メッセージ）の送受信
- テーブル管理機能

6.5 転送サーバ

テーブル管理機能に着目して例外を分析すると（図 4.2 における５）、システムにさらに機能が必要になることが分かる。図 6.3 に示すように、移動の最中に帰還命令が送信されることが考えられる。つまり、テーブルが常に正しいとは限らない。このような例外に対しては、転送エージェントを残す方法などが考えられる（図 6.3）。

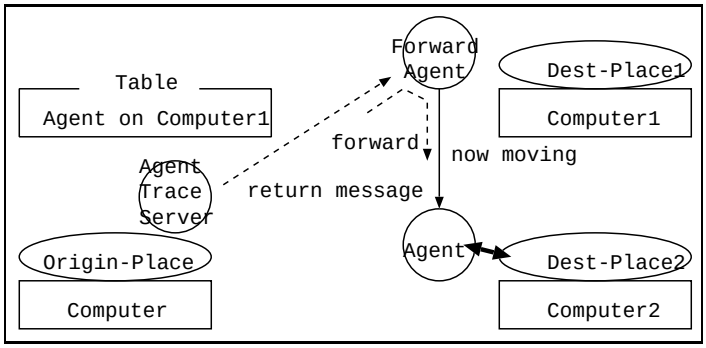

図 6.3: 転送エージェント

同様に転送エージェントについて機能分析を行うと、

- メッセージの送受信

となる。この機能に関する例外処理は、内部検出例外となるため、ずっと単純になる。
このように、図 4.2 の方法に従って、帰還命令の設計が行うことができる。

6.6 マルチキャストの方法

これまでは、帰還命令を実現するために、2 の段階で、エージェントの位置把握機能が必要としてきたが、この他に、移動先がある程度決まっている場合、マルチキャストが対処法として考えられる（図 6.4)。この機能を実現する上でも、同じように分析できる。

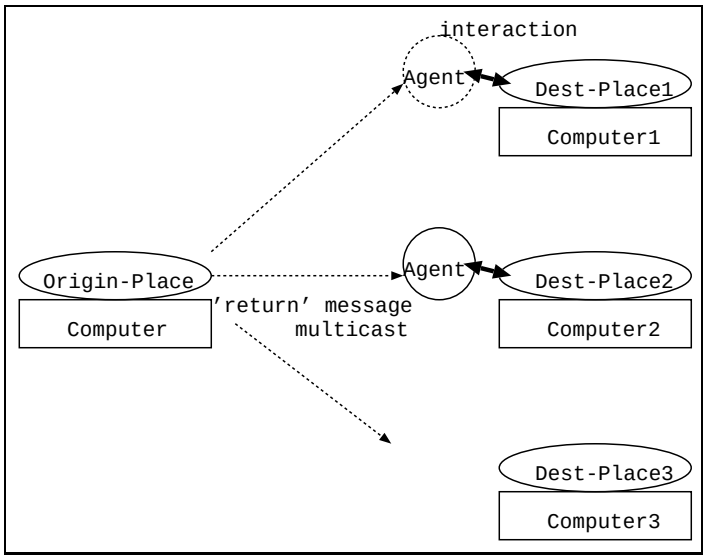

図 6.4: マルチキャストによるメッセージ送信

この方法も機能を分析すると、エージェント位置管理と同じように転送サーバが必要なことが分かる。
マルチキャストでも同じ方法論によって、例外処理設計が可能である。

第 7 章

必要機能出力システム

本研究の過程で考案した必要機能出力システムについて、本章で議論する。

7.1 背景と目的

例外処理設計が複雑になるという点については、これまでに述べたきた通りである。この背景から、例外処理設計支援アプリケーションが望まれていると考えられる。そこで、本研究で、例外処理設計支援アプリケーションの設計と実装についても視野に入れた。その結果、設計されたシステムが、本章で述べる必要機能出力システムである。その詳細を本章で述べる。

7.2 システム概要

まず、システム概要は、図 7.1 に示す通りである。

このシステムにおいて仮定されることは、システムは一旦完成していることである。そして、一旦完成したシステムに対して新たな要求を導入する際に、現システムの機能と新たな要求から、必要な機能を出力するというシステム図 7.1 である。

本研究では、図 7.1 のようなシステムを必要機能出力システムと名付け、設計を行った。ただし、図 7.1 は一般化されているが、特に例外処理設計に関して設計を行った。

システムの外部仕様を表 7.2 に示す。

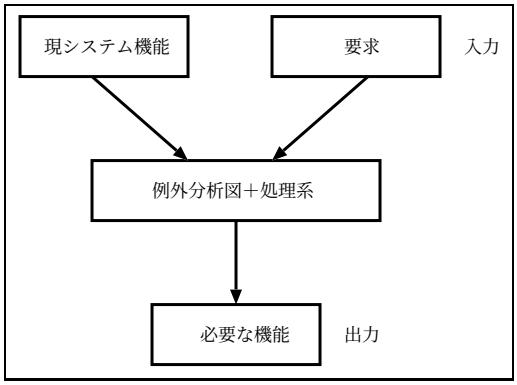

図 7.1: 必要機能出力システム

入力	現システムの機能 新たな要求
出力	必要な機能

表 7.1: 必要機能出力システムの仕様

7.3 システムの内部

必要機能出力システムを実現するために、システムの内部に例外分析図（データ構造）と処理系（アルゴリズム）という構造を考案した。以下に、それぞれ説明する。

7.3.1 例外分析図

例外処理設計を行う際、どのようなことを行っているかを分析してみると、以下の３つに分けられる。

1. 起こり得る例外を考え（例外把握)、
2. それらに対する対処法を考え、(対処法考案)
3. 対処法の実現方法を具体的に考えるとともに、その対処法に必要となる機能の分析を行う。(機能分析)

これについて、さらに分析した結果を、表 7.2 に示す。

	フェーズ（目的）	ステージ	分析対象
例外分析	例外把握	1. 2. 3.	オブジェクトとその関係 例外検出のベース 可能な例外
	対処法考案	4. 5.	例外に起因する問題 問題に対する対処法
	機能分析	6.	対処法に必要となる機能

表 7.2: 例外分析法

表 7.2 に示す通り、例外把握フェーズは３つ、対処法考案フェーズは２つ、機能分析フェーズは１つのステージに分けられる。それぞれのステージの説明を以下に示す。

1: オブジェクトとその関係 このステージでは、オブジェクトとオブジェクト間の関係について把握する。これは、例外が以下の２つに分類されると考えられるからである。

- 単一オブジェクトの例外として捉えられるもの。
- 複数オブジェクトの関係から例外を捉えた方が良いもの。

2: 例外検出のベース このステージでは、オブジェクトから派生する例外をリストアップするためのベースを考える。ベースは以下の３つに分類されると考えられる。

- ソフトウェア（実装言語）
- 外部（ハードウェア的な例外で、外部から検出する）
- ユーザ（ユーザが意図的に発生させる例外）

3: 可能な例外 このステージでは、ステージ２のベースから派生する例外を考察し、リストアップする。ここまでが、例外把握フェーズである。

4: 例外に起因する問題 ステージ４では、ステージ３でリストしたそれぞれの例外に起因する問題について考察し、分類する。

5: 問題に対する対処法 ステージ５では、ステージ４で考えられた問題に対する対処法を考案する。ここまでが対処法考案フェーズである。

6: 対処法に必要となる機能 ステージ６では、ステージ５で考えられた対処法に必要となる機能を分析する。ここが、機能分析フェーズである。

本研究では、この例外分析法で分析、図示された図を例外分析図と呼ぶ。例外分析図の１つの例を図 7.2 に示す。

この例外分析図が、必要機能出力システムのデータ構造となる。

7.3.2 処理系

処理系について説明する前に、システムの入出力と例外分析図との対応について示す。

入力: 現システム機能 { 対処法に必要となる機能 }（ステージ６）の部分集合

入力: 要求 { ステージ３の例外、ステージ４の問題、ステージ５の対処法の導入 } のいずれか

出力: 必要な機能 { 対処法に必要となる機能 } の部分集合－ { 現システムの機能 }

これらから、アルゴリズムは直ちに導かれる。それを以下に示す。

1. 例外分析図から要求を検索する。
2. 要求が見付かったら、リンクを辿り、必要な機能を得る。
3. 必要となる機能から現システムの機能を引き、出力を得る。

以上が、必要機能出力システムの詳細である[1]。

[1]実装は行わなかった。

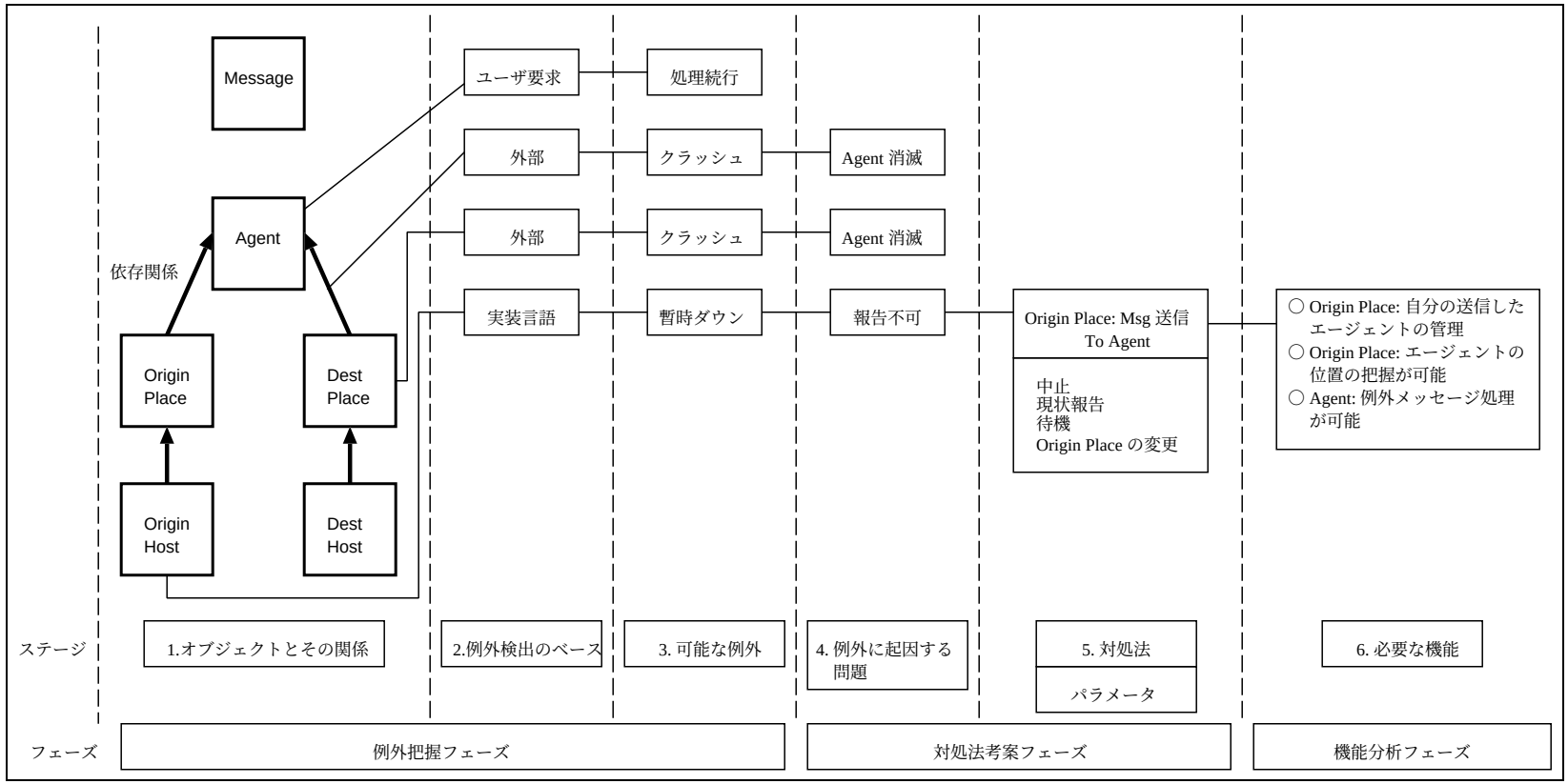
Message
Agent
依存関係
Origin Place
Dest Place
Origin Host
Dest Host
ユーザ要求
処理続行
外部
クラッシュ
Agent 消滅
外部
クラッシュ
Agent 消滅
実装言語
暫時ダウン
報告不可
Origin Place: Msg 送信 To Agent
中止
現状報告
待機
Origin Place の変更
○ Origin Place: 自分の送信したエージェントの管理
○ Origin Place: エージェントの位置の把握が可能
○ Agent: 例外メッセージ処理が可能
ステージ
1.オブジェクトとその関係
2.例外検出のベース
3. 可能な例外
4. 例外に起因する問題
5. 対処法
パラメータ
6. 必要な機能
フェーズ
例外把握フェーズ
対処法考案フェーズ
機能分析フェーズ

図 7.2: 例外分析図

第 8 章

まとめ

本章では、本研究のまとめ、およびこれからの課題について述べる。

8.1 まとめ

本論文では、システム全体を統合した例外処理設計の方法論を確立することを目的として議論してきた。

本研究では、まず、例外処理設計の定義を行った。次に、例外の検出主体という観点から例外を分類し、どのような例外処理設計が難しいのかを検討した。次に、我々の思考過程を抽出し、一般化を行った。最後に、いくつか具体的な例を用いて、方法論の検証を行った。

8.2 これから

以下に、これからの課題を示す。

- 対処法考案のための何らかの方法論
  必要とされる機能を元に、対処法を考案しなければならないが、この点については、何の指標も得ることができず、研究できずに終った。
- OMT の機能モデルを利用した例外設計論について
  本研究で得られたことを OMT の方法論への適用方法を考えたい。
- さまざまな例における方法論の検証
  バックアップサーバやビジネスの絡んだシステムに本手法を適用したい。

- オブジェクトの増えやすさの指標
  「オブジェ クトの増えやすさ」を定性的、定量的に計測する方法についても研究したい。

第 A 章

エージェントの定義とエージェントの特性

ソフトウェアアエージェントの定義は、まだ確立されておらず、さまざまな人によって行われている。一方、エージェントに必要とされる特性に関しては、ほぼ収束している。両者を、本章で紹介しておく。

A.1　エージェントの定義

エージェントの定義を３冊の本から引用し、簡単な説明を加えておく。

Webster Dictionary [2] 5. [1]one that acts for or in the place of another by authority from him. （図 A.1）

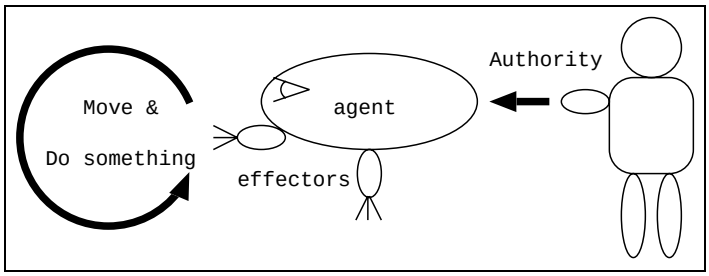

図 A.1: Webster Dictionary におけるエージェントの定義

この定義には、「権限の委譲」というエージェント技術においてもっとも重要な概念が含まれていることに注目する。

[1]この番号は、１つの単語に複数の意味があるときにつける番号で、この場合５番目の意味となる。（この定義が最もソフトウェアエージェントに近いと判断した。）

エージェントアプローチ人工知能 [1] エージェントは、ある環境をセンサで知覚し、ある環境にエフェクタを通して動作するものである（図 A.2)。

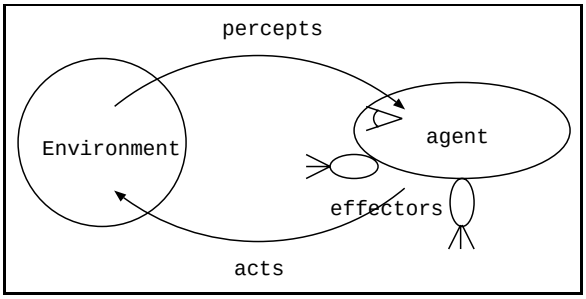

図 A.2: エージェントアプローチ人工知能におけるエージェントの定義

この定義は、エージェントを環境とのインタラクションを通して計算を行う主体としている。この定義は、機能的に観点からの定義であるが、現在のところ最も一般化されていると考えられる。

Multiagent Systems [4] In the case of cooperation several agents try to combine their efforts to accomplish as a group what the individuals cannnot （図 A.3）.

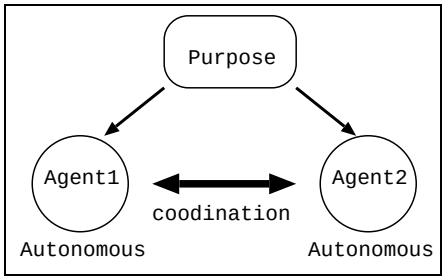

図 A.3: Multiagent Systems におけるエージェントの定義

この定義は、マルチエージェントに関してであり、協調動作を行うことによって、１つの目標を達成する。

A.2 エージェントの特性

エージェントの持つ４つの特質は、以下の通りである [8] [9]。

自律性 (Autonomy) エージェントは、自分自身の知識や種々の情報を活用して自ら判断し、決断することができる。この特性が、エージェントを最も特徴づける。

社会性 (Social Ability) エージェントは、相互に理解可能な言語（エージェントコミュニケーション言語）を用いて、他のエージェントと対話し、協調的な動作が可能である。

反応性 (Reactivity) エージェントは、自分が置かれた環境を認識し、その変化を検知し、適切に行動できる。

自発性 (Pro-activeness) エージェントは、ある目標を目指して、行動することができる。

以上、エージェントの定義と特性を述べたが、これらからエージェントの技術自体は、人工知能の分野として研究されてことが示唆される。

謝辞

本研究を行なうに当たり, 終始御指導を賜わった篠田 陽一助教授に深謝致します.

また, 本論文をまとめるに当たって御協力いただいた篠田研究室の諸兄に厚く御礼申し上げます.

参考文献

[1] 監訳: 古川康一, エージェントアプローチ : 人工知能, 東京 : 共立出版 , 1997.12.

[2] Philip Babcock Gove and the Merriam-Webster editorial staff., Webster's third new international dictionary of the English language Merriam-Webster, c1993.

[3] David Kotz, Friedemann Mattern, Agent Systems, Mobile Agents, and Applications ASA/MA 2000 Proceedings

[4] Gerhard Weiss, Multiagent Systems, The MIT Press, 1999

[5] David Flanagan, (永松 健司訳), Java Quick Reference, オライリージャパン, 1996.

[6] Sun OS 5.7, soket(3N) Manual Page, Sun Microsystems, 1997.

[7] 新堀 健治, 信頼性の高い移動エージェントシステムの構成方法, 北陸先端科学技術大学院大学, 修士論文, 1999.

[8] M.J.Wooldridge, & N. Jenninsg （eds）, Intelligent Agents, Lecture Note in Artif. Intell. 890, Springer.

[9] 木下 哲男, エージェント思考コンピューティング ～エージェントの基礎と応用～, ソフトリサーチセンター, 1995

[10] 山村 吉信, ソフトウェア開発の技術, 三元社, 1993.

[11] Tom DeMarco, 渡辺 純一 訳, ソフトウェア開発プロジェクト技法近代科学社 1987.

[12] J. ランボー, M. ブラハ, W. プレメラニ, F. エディ, W. ローレンセン, 羽生田 栄一 訳 トッパン 1992.

[13] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Strong Mobility and Fine-Grained Resource Control in NOMADS, Lecture Notes in Computer Science.

[14] 東芝（株）, コミュニケーションをエージェント化するマルチエージェントフレームワーク, http://www2.toshiba.co.jp/beegent/

[15] 東芝（株）, 考えて動きまわるエージェント Plangent, http://www2.toshiba.co.jp/plangent/