博士論文 平成15年度(2003)

社会・経済シミュレーションの基盤構築

— 複雑系と進化の理論に向けて —

慶應義塾大学大学院 政策・メディア研究科

井庭 崇

博士論文要旨 平成15年度 (2003)

社会・経済シミュレーションの基盤構築
— 複雑系と進化の理論に向けて —

本論文では、複雑系のシステム観に基づく社会・経済シミュレーションを作成するためのモデル・フレームワークとシミュレーション・プラットフォームを提案する。複雑系とは、広義には「内部状態をもつ多数の構成要素が相互作用し、それぞれの内部状態を変化させていくシステム」であり、狭義には、これに加えて「構成要素の振舞いのルールが変化し得る」という定義が加わる。近年、社会科学においてこのような捉え方が重要視されているが、現状では、複雑系のモデルを記述し操作するための有効な手段は存在しない。

本論文では、複雑系の社会・経済モデルを記述・操作するために、オブジェクト指向計算モデルを導入する。そして、基本となるモデル要素を定義し、モデル化からシミュレーションまでの一貫した支援を行うためのモデル・フレームワーク「Boxed Economy Foundation Model」を提案する。さらに、提案モデル・フレームワークに基づくモデルのシミュレーションの作成・実行・分析を支援するために、コンポーネントベースのソフトウェア「Boxed Economy Simulation Platform」を提案する。また、動的な振舞いの構成方法を、モデル・パターンとして記述することを提案し、具体的なモデル・パターンを提示する。

本論文では、提案モデル・フレームワークと提案ソフトウェアの有効性を明らかにするため、これらを既存モデルおよび独自モデルに適用する。既存モデルでは、成長するネットワークモデル、繰り返し囚人のジレンマモデル、貨幣の自生と自壊モデル、Sugarscape モデル、人工株式市場モデルという 5 つの代表的なモデルを取り上げる。独自モデルとしては、家庭用 VCR の規格競争シミュレーションを取り上げる。これらの事例への適用により、本論文の提案の有効性が実証された。

慶應義塾大学大学院 政策・メディア研究科
井庭崇

Abstract of Ph.D Thesis Academic Year 2003

A Study on Simulating Economies and Societies as Evolutionary Complex Systems

This dissertation presents a model framework and a simulation platform for simulating economies and societies as complex systems. In a broad sense, the complex system means that the system has the components where each component changes the internal states by mutually interacting with the other components. In addition, in a strict sense, the complex system means that the rules of each component behavior are changed dynamically during the simulation. There is no satisfactory scheme for modeling and simulating the complex systems, although the complex system model has been highly demanded in social sciences.

We introduce an object-oriented computational modeling for social sciences in order to model and simulate the complex system where the model framework, "Boxed Economy Foundation Model," is proposed. Moreover, the component-based software system "Boxed Economy Simulation Platform" is proposed for building/simulating a model and analyzing the system. In addition, the model pattern is proposed to describe examples on how to build the dynamic behavior.

In this dissertation, we apply the proposed framework and software system to existing models and an original model. The existing models include the followings: the model of evolving networks, the model of iterated prisoner's dilemma, the model of emergence and collapse of money, Sugarscape model, and the model of artificial stock markets. Then, the original model is examined and used for video cassette format competition. The performance of the proposed framework and the software system are justified in the application.

Takashi Iba
Graduate School of Media and Governance
Keio University

目 次

図 目 次

表 目 次

第1章　本論文の目的と概要

1.1　新しい思考の道具をつくる

本論文の目的は、社会・経済を分析するための新しい思考の道具を提案することにある。この思考の道具は、「組織化された複雑性」の領域にある社会・経済現象を、複雑系のシステム論的アプローチで取り組むことを支援するものである。社会のような組織化された複雑性をもつ対象は、従来の解析的方法や統計的方法では理解が困難であるといわれているが、本論文では、システム論的な捉え方とコンピュータ・シミュレーションによってアプローチすることを目指す。

社会・経済システムの研究は、同時代のシステム論からの影響を受けつつ発展してきており、近年「複雑系」(complex system) と呼ばれるシステム観が重要視されはじめている。しかし現在のところ、「複雑系」という用語について確立された定義や明確な合意があるわけではない。そこで本論文では、複雑系研究の現状を踏まえ、「広義の複雑系」と「狭義の複雑系」という二つの定義に分けて整理することにする。第一の定義である「広義の複雑系」とは、「内部状態をもつ構成要素が多数相互作用するシステム」のことである (図 1.1)。そして、第二の定義である「狭義の複雑系」とは、上記の定義の中でも特に「構成要素の振舞いのルールが動的に変化するシステム」のことである。いずれの場合でも、複雑系の構成要素は原子論的な意味でのアトムではなく、内部状態をもつという点に特徴がある (図 1.2)。これは、「分解を推し進めることによって、最終的には不変の最小単位に到達する」と考える物理学とは異なる立場をとることになる。このシステム観は、特に社会・経済や生命を理解する上で不可欠であると近年考えられている。

ところが、現在「広義の複雑系」および「狭義の複雑系」として社会・経済を分析するための有効な方法と道具立ては存在しない。社会システム論では、システム論的な捉え方の重要性を早くから訴えてきたが[(1)]、記述したモデルを操作するための具体的な道具立ては提案されていないのが現状である。また、シミュレーションの分野では、「広義の複雑系」の支援システムがいくつか提案されているものの、「狭義の複雑系」については未だ有効な支援がなされていない。その結果、有効な方法と道具立ての不在が、研究の進展を困難なものにしているということ、そして今後その問題がさらに深刻化するということが、本論文の基本的な問題意識である。

このような現状に対し、本論文では、そのモデル記述の方法としてオブジェクト指

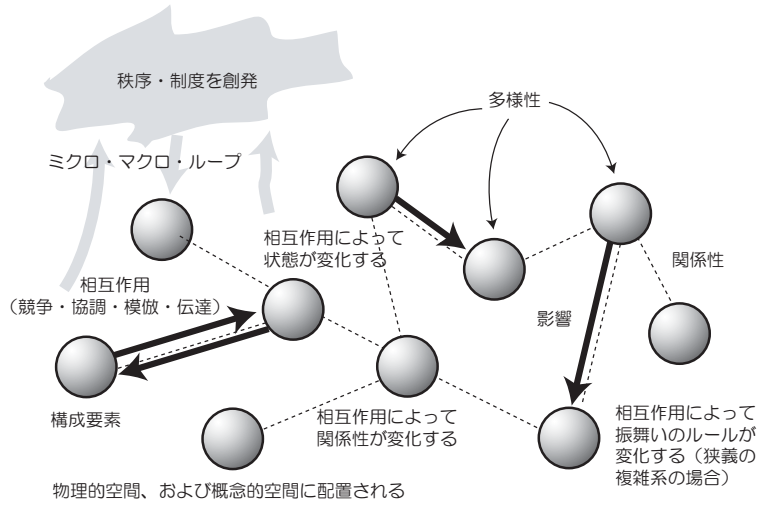


図 1.1: 複雑系のシステム観 (第 2 章より)

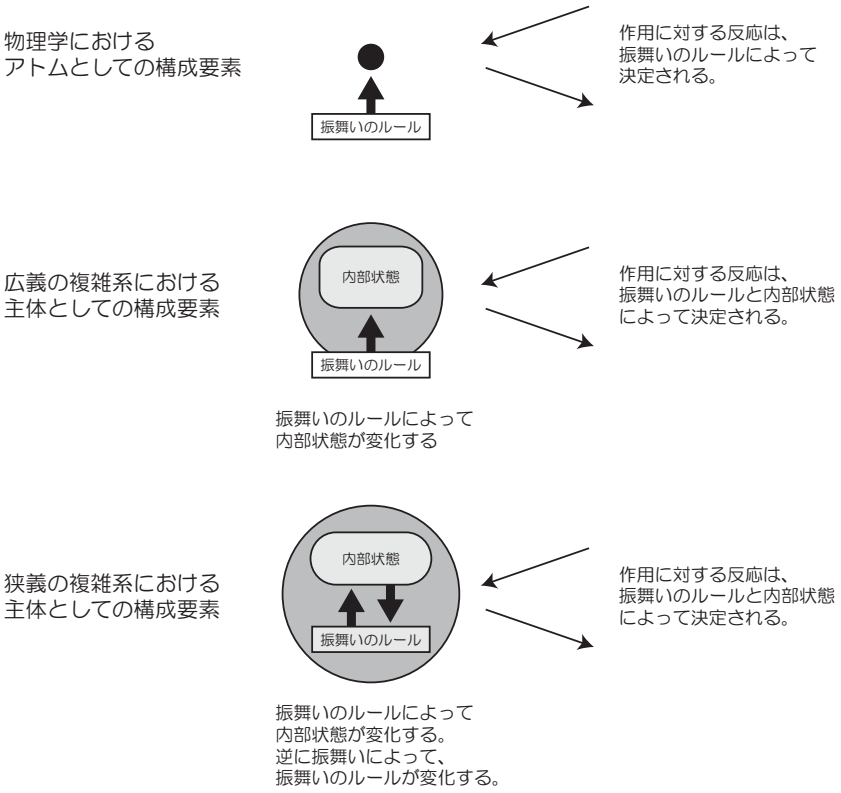


図 1.2: 物理学、広義の複雑系、および狭義の複雑系における構成要素の特徴 (第 2 章より)

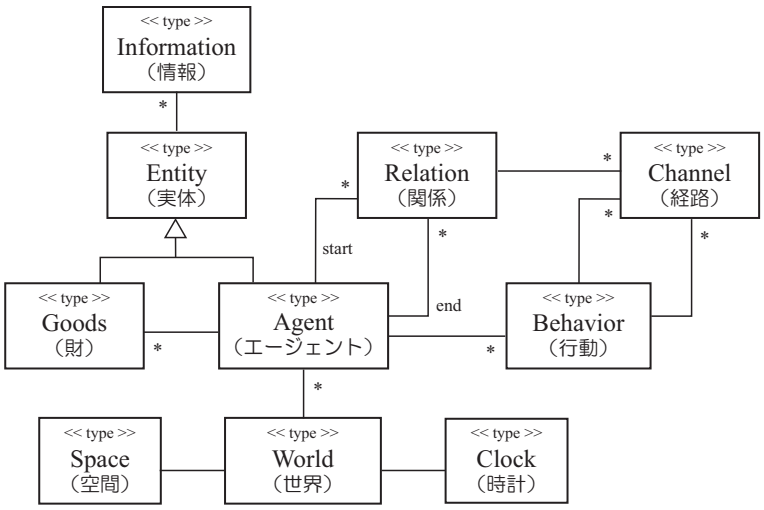


図 1.3: BEFM 概念モデル・フレームワークのクラス図 (第 4 章より)

向計算モデルによる記述を採用し、「広義の複雑系」および「狭義の複雑系」としての社会・経済を記述するためのモデル・フレームワークを提案する。また、そのフレームワークに基づくコンピュータ・シミュレーションを作成・実行するためのソフトウェア、および動的な振舞いの記述を支援するモデル・パターンを提案する。本章では、本論文における重要な論点を先取りして、その概要提示することにしたい。

1.2 モデル・フレームワークの提案

本論文では、複雑系の社会・経済モデルを記述・操作するために、基本枠組みとしてオブジェクト指向計算モデルを導入する (第 3 章)。そして、基本となるモデル要素を定義し、モデル化からシミュレーションまでを一貫して支援するモデル・フレームワークを提案する (第 4 章)。

モデル・フレームワークとは、社会・経済のモデルを記述する際に、頻繁に登場する要素と構造を定義したメタモデルのことである。モデル・フレームワークには、モデルの概念レベルの「概念モデル・フレームワーク」と、プログラムの実装レベルの「シミュレーションモデル・フレームワーク」の 2 種類がある。モデル作成者は、モデル化しようとしている対象が「どのようなものであるか」(What) を洗い出し、記述する際に、概念モデル・フレームワークを用いることができる。また、概念モデル・フレームワークがシミュレーションモデル・フレームワークと一貫性を有することから、シミュレーションモデルへの移行をシームレスに行うことができる。

本論文で提案するモデル・フレームワーク「Boxed Economy Foundation Model」は、オブジェクト指向計算モデルによって、複雑系としての社会・経済を記述するためのフレームワークである (図 1.3)。エージェント間の相互作用を、財 (情報が付随す

ることがある) のやりとりとして明示化するという点と、エージェントの行動を、エージェントとは別のモデル要素として定義するという点に特徴がある。このようなエージェントの設計は、新しい行動の追加や削除、そして行動の組み換えなどを簡単に行えるという柔軟性がある。したがって、「振舞いのルールに従って状態が変化する」だけでなく、「状態の変化によって振舞いのルールが変化する」という「狭義の複雑系」のモデルも表現できるようになる。

1.3 シミュレーション・プラットフォームの提案

本論文では、複雑系としての社会・経済のシミュレーションを作成・実行・分析するためのソフトウェア「Boxed Economy Simulation Platform」(図 1.4) を提案する (第 5 章)。Boxed Economy Simulation Platform (以下、BESP) を用いることで、提案モデル・フレームワークに基づくモデルのシミュレーションを作成・実行・分析することができるようになる。

BESP が目指しているのは、研究プロセスを一貫して支援するための統合環境を提供することである。このような統合環境の支援によって、「モデルの作成」から「実装」、「実行」、「評価」、「現実との比較」というプロセスをシームレスに、かつ効率的に行うことが可能となる。時々刻々と変化していく社会・経済を up-to-date に捉えていくためには、モデルを迅速に作成し、実行・分析することが重要となるが、BESP ではそのための工夫がなされている。

BESP では、モデルのコンポーネント (部品) を組み合わせて、シミュレーションを設定できる仕組みになっている (図 1.5)。シミュレーションは、複数のコンポーネントの組み合わせによって動作するが、それぞれのコンポーネントは、独立して理解したり作成したりすることができる。今後作成したいモデルが複雑かつ大規模になるにつれて、一つの研究グループでモデルのすべてを作りきれなくなると予想されるため、コンポーネントの再利用性はますます重要になるだろう。

また、BESP では、シミュレーションを作成する際のプログラミングを大幅に軽減させる支援ツールも提供している。BESP とともに提供している「コンポーネントビルダー」は、 Behavior の状態遷移図を作成すると、モデルコンポーネントのプログラムコードを自動生成してくれるツールである。このツールを用いると、シミュレーションにおいて最も重要であるがプログラミングが難しい「動的な振舞い」についてのプログラミングをしなくて済むので、複雑なモデルも比較的容易に作成できるようになる。また、BESP 本体が提供するさまざまな機能 (例えば実行や制御に関する部分) は、すでにプログラミングされているため、シミュレーション作成者がプログラミングする必要はないということも、プログラミング作業の軽減や品質の向上に寄与している。

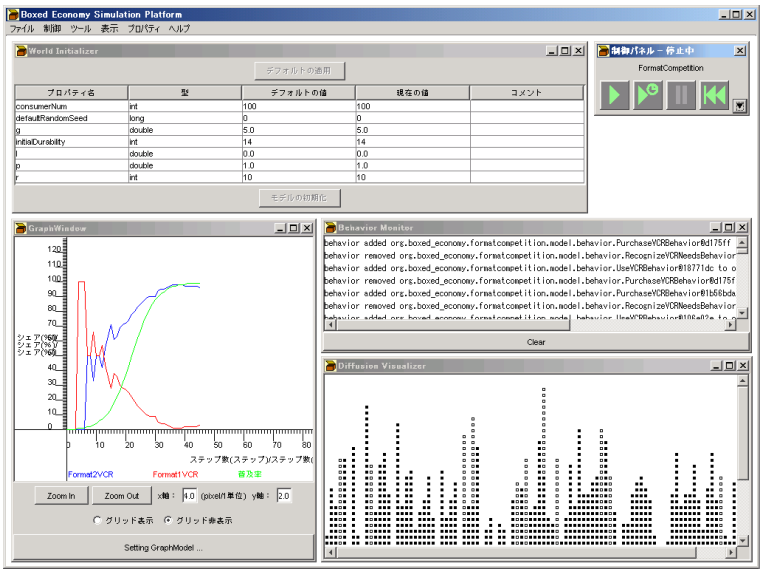


図 1.4: Boxed Economy Simulation Platform (BESP) の画面 (第 5 章より)

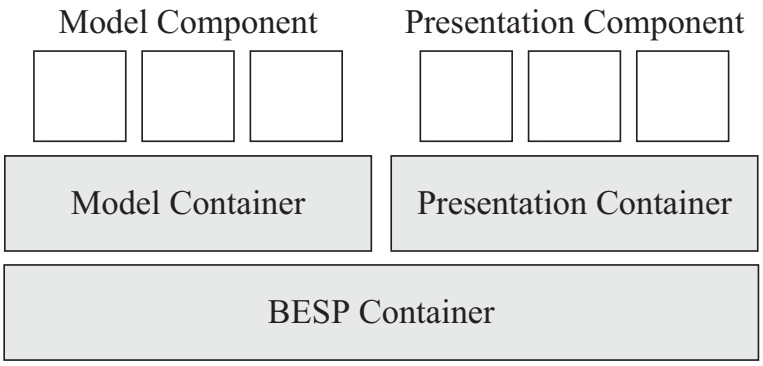


図 1.5: BESP の内部構造 (第 5 章より)

1.4 モデル・パターンの提案

本論文では、動的な振舞いの構成方法のノウハウを、モデル・パターンとして記述することを提唱し、実際に重要だと思われるモデル・パターンを提案する (第 6 章および付録 C)。モデル・パターンとは、モデルを構成する部分部分の「組み立て方」に関する知識である。パターンの考え方は、もともと建築デザインのために考案され、その後ソフトウェア・デザインに取り入れられたものであるが、本論文では、そのパターンの考え方をモデル・デザインに応用することを提唱する (表 1.1, 図 1.6)。

モデル・パターンは、モデル・フレームワークと異なり、それを必ずしも採用する必要はない。モデル・パターンのなかには、お互いに対立するものも含まれており、モデル作成者は、代替案の中から選びながらモデルをつくることになる。

表 1.1: 本論文で提案するモデル・パターン (第 6 章より)

モデル・パターンの分類	モデル・パターン名
エレメンタリーなモデル・パターン	Agent Creation Relation Creation Related Agent Creation Agent Destruction Goods Creation Information Creation
コミュニケーションのモデル・パターン	Information Sending Blank Information Sending Internal Information Sending Immediate Reply Collect Immediate Replies Appointed Destination Reply Super BehaviorType Calling
行動変化のモデル・パターン	Behavior Creation Behavior Destruction Behavior Switching Temporary Behavior Attachment Requested Behavior Attachment Forced Behavior Attachment
アクティベーションのモデル・パターン	TimeEvent Distributer Agent TimeEvent Filtering TimeEvent Distributer Behavior Time-Consuming Behavior

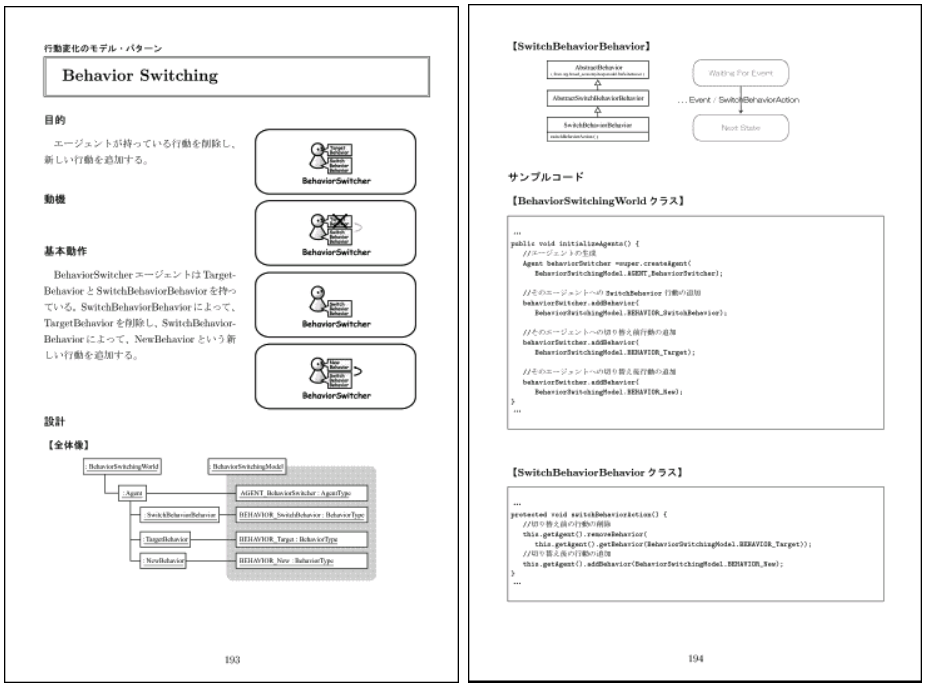


図 1.6: カタログ形式で記述されたモデル・パターンの例 (第 6 章より)

1.5 提案システムの適用事例

1.5.1 既存モデルの再現

本論文の提案 (モデル・フレームワーク、シミュレーション・プラットフォーム、モデル・パターン) の適用可能性を明らかにするために、それらを用いて既存モデルを複数作成する (第 7 章)。取り上げるモデルは、(1) 成長するネットワークモデル、(2) 繰り返し囚人のジレンマモデル、(3) 貨幣の自生と自壊モデル、(4)Sugarscape モデル、(5) 人工株式市場モデルの 5 つである。これらは、社会・経済シミュレーションの典型的な特徴を備えた代表的なモデルである。

成長するネットワークモデル: 関係とエージェントの動的生成

成長するネットワークモデルは、社会ネットワークの分野におけるモデルである。この分野は近年、スケールフリー・ネットワークという新しいネットワークの表現方法の研究が進み、注目を集めているが、そのスケールフリー・ネットワークの生成モデルを取り上げる。提案システムでは、ノード (Agent) の追加やリンク (Relation) の追加が容易であることから、このようなモデルに対して有効であることを示す。

繰り返し囚人のジレンマゲーム: 行動変更による振舞いの変化

繰り返し囚人のジレンマゲームは、政治学等で非常によく用いられる分析枠組みである囚人のジレンマを、繰り返し行うゲームとして展開したものである。ここでは、戦略を Behavior の状態遷移で直接的に記述する。また本論文では、試合やコンテストの結果から、自分より強いプレイヤーの戦略を模倣するという拡張を行い、マクロ情報がある場合とない場合の振舞いの違いについて分析する。この戦略模倣シミュレーションでは、プレイヤー (Agent) が戦略 (Behavior) を状況に応じて切り替えるという「狭義の複雑系」のモデルを実現している。

貨幣の自生と自壊モデル: 段階的なモデル拡張

貨幣の自生と自壊モデルは、商品の物々交換社会において、他者の欲するものを記憶として持たせることで、貨幣の機能を果たす商品が登場するということをシミュレートしたものである。ここでは、エージェントに対して、Behavior を追加・組み替えを行うことによって、モデルの拡張が可能であることを示す。

SugarScape モデル: 環境のエージェント化

SugarScape モデルは、砂糖が生成される 2 次元セル平面上で、蟻のようなエージェントが移動しながら砂糖を採取・消費し、取引を行うというモデルである。このモデルは、2 次元セル空間による社会シミュレーションの枠組みを広く普及させた代表的なモデルである。ここでは、砂糖が生成される「環境」も、エージェントによって表現することができることを示す。

人工株式市場モデル: 情報変化による振舞いの変化

人工株式市場モデルでは、状況によって異なる行動をとるトレーダーを実現するため、クラシファイアシステムを導入したモデルを取り上げる。株式市場では、自分の戦略だけでなく、取引する他のトレーダーの戦略によって結果が異なるため、絶えず優位な戦略というものは存在しないことが観察される。ここで取り上げる SantaFe モデルの枠組みは、その後多くの研究で、代表的な人工市場のモデルとして取り入れられている。ここでは、Behavior の切替ではなく、戦略情報の内容によってエージェントの振舞いを変化させるモデルを実現する。

1.5.2 独自モデルによる分析

独自モデルとして、「家庭用 VCR の規格競争」のモデルを取り上げる (第 8 章)。このモデルは、家庭用 VCR の VHS 方式と Beta 方式の規格競争における、消費者間の相互依存関係を組み込んだ需要側の人工市場モデルである。提案した人工市場モデルでは、ミクロレベルのモデル化を行うため、従来のマクロ集計的なネットワーク外部性モデルでは分析できない局所性や逆転現象などが分析可能であることを示した。なお、このモデルの記述面における特徴は、エージェントが必要に応じて、行動を追加したり削除したりしている点である。

第2章　社会・経済のモデル化とその分析方法

2.1　メタファーとしてのモデル

社会・経済を理解したいとき、社会科学では、対象となる現象の説明根拠を、その社会そのもの、あるいはそれを構成する人間に求め、その要因を把握することを試みる。もちろん実際には、「そのような要素は、具体的現象においては相互に入り組んでおり、ほとんどのばあい説明もできなければ把握できないほどに縺れ合っている」(Schumpeter, 1915) ため、本当の意味ですべての因果の連鎖を把握することは不可能である。それゆえ、本質的に重要だと思われる連関についての「モデル」を作成し、現象を理解したり予測したりすることになる (図 2.1)。

モデルとは何かという定義にはいろいろなものがあるが、ここでは Wilson (1990) による次のような定義を想定しておくことにしよう。「“モデル” とは、ある人間にとっての、ある状況、あるいは状況についての概念 (idea) の明示的な解釈 (explicit interpretation) である。モデルは、数式、記号、あるいは言葉で表すことができるが、本質的には、実体、プロセス、属性、およびそれらの関係についての記述 (description) である」(Wilson, 1990)。

このように捉えると、モデルという知的構築物は、メタファー (隠喩)[(2)]の役割を果たしていると考えることができる (Black, 1962; Hesse, 1966; Hesse, 1980)。ここでいうメタファーとは、単なる言葉の綾や修辞的な文飾のことではなく、人間の認知や思考に組み込まれた「見立て」の方法のことである (Lakoff and Johnson, 1980)。つまり、メタファーとは、「より抽象的で分かりにくいカテゴリーに属する対象を、より具体的で分かりやすいカテゴリーに属する対象に見立てることによって、世界をよりよく理解する方法」(瀬戸, 1995) である。メタファーの基本要素は、「たとえられるもの」と「たとえるもの」、「そのたとえの根拠」であるが、この場合、現実世界における対象が「たとえられるもの」、モデルが「たとえるもの」である。科学的研究とは、「たとえるもの」(モデル) を作成し、「そのたとえの根拠」を、実験などを通じて検証・確証・反証していくという営みということになる。

近年の科学哲学では、Hanson (1970) や Kuhn (1962) 等で主張されているように、科学的知識も客観的なものではなく、それぞれの科学者の認識の枠組みで解釈され構成されたものであると考えられている。観察行為というプリミティヴな行為でさえ、

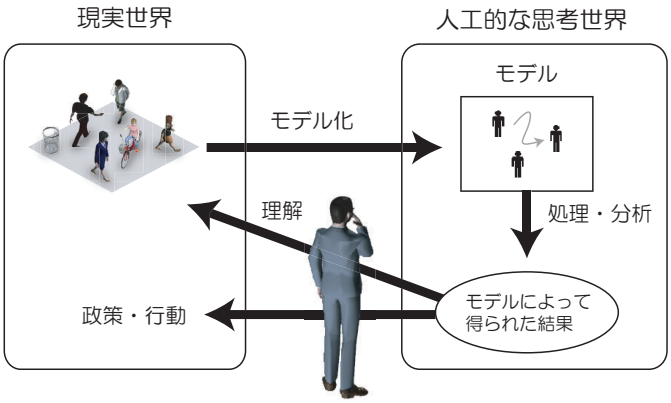


図 2.1: モデルによる思考

現実からありのままの「事実」を受動的に受けとっているのではなく、「〜として見る」(seeing as) や「〜ことを見る」(seeing that) というメカニズムが不可避的に組み込まれているのである。そのため、私たちの思考の中の「たとえるもの」の表現力が貧弱であれば、分析によってわかることも貧弱なものにならざるを得ないということになる。「認識装置が新たに開発されてはじめて、既存のそれにはなかったリアリティが取りだせる」(今田, 1986, p.28) ことから、「たとえるもの」の表現力を上げていくことも、科学における重要な活動となる。

本章では、これまで社会科学において、社会・経済がどのようなメタファーで捉えられてきたかかを明らかにし、本論文で扱おうとしてている複雑系の捉え方までを概観する。本論文の目的は、複雑系のシステム観にもとづく思考の道具を構築することにあるが、「問題状況を記述する方法 (モデル化言語) は、扱っている問題の本質にあったものでなければならない」(Wilson, 1990) ため、このような準備が不可欠となる。

2.2 いまどのようなメタファーが求められているのか

社会・経済のメカニズムの解明を目指し、「システム」の考え方を自覚的に適用するのが、社会・経済システムの考え方である。システムという語は多義的であり、分野や時代によって様々な意味で用いられてきているが、大方の共通する定義としては、「複数の諸部分が互いに関係をもって相互作用しており、より大きな全体として統合されている」という点である。

社会・経済システムの研究は、並行して発展してきたシステム論[3]の流れのなかでどの段階のシステム観を採用するかによって、さまざまな形態が存在する (図 2.2)。まず、社会有機体論や社会機械論から始まり[4]、その後、パーソンズによって社会システム論が打ち立てられた[5]。同時期に、サイバネティクス[6]や一般システム論[7]が発展し、それらの影響を受けて社会科学への導入も試みられてきた[8]。その後、システ

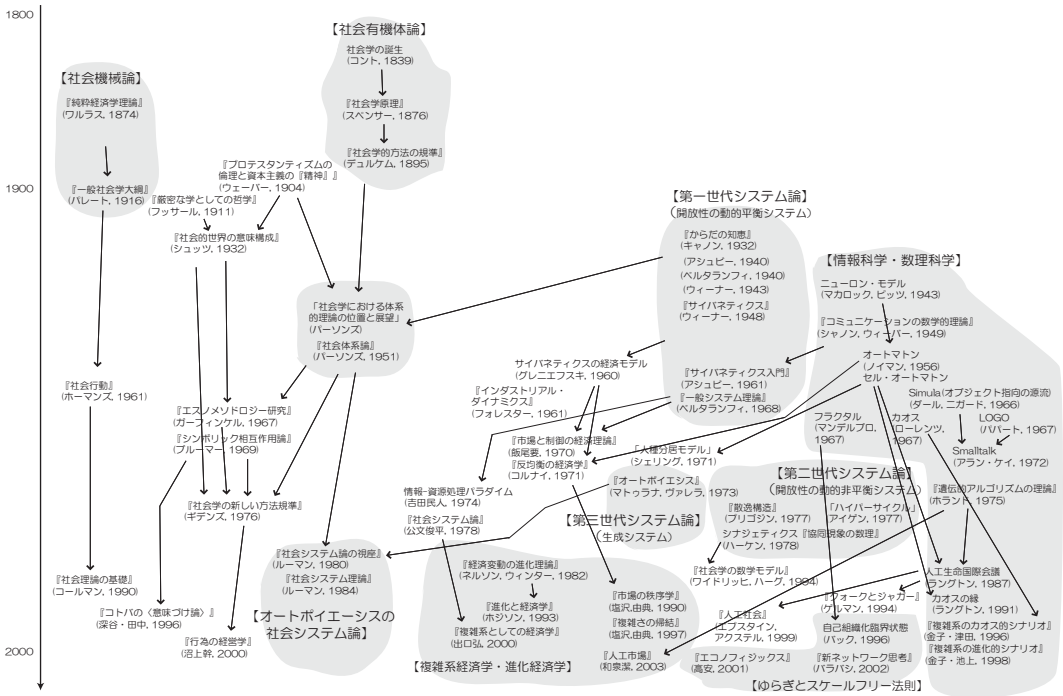


図 2.2: 社会・経済システム論の変遷の大まかな流れ

ム論では、第二世代といわれる「開放性の動的非平衡システム」[9]、すなわち散逸構造理論[10]、シナジェティクス[11]、ハイパーサイクルなどが登場する。これらの考え方は、社会シミュレーションのモデル化に導入されたほか (Weidlich and Haag, 1983)、社会科学や科学哲学における思想面で影響を及ぼしている。また、神経生理学において提唱されたオートポイエーシス (自己創出) 理論は、N. ルーマンによって社会システム論として展開されている。そして近年、これらのシステム論の流れと、分散人工知能などの情報科学の流れを受けて、「複雑系」のシステム観が登場する。

しかし、現在のところ「複雑系」という用語について確立された定義や明確な合意があるわけではない。複雑系という概念の定義は、研究者によって、あるいは時代によって、まったく異なる意味で用いられており、また、未定義語のまま使用されていることも多い。そこで本論文では、混乱した状況を整理して議論しやすくするために、ひとまずの定義を行うことにしたい。その定義とは、「内部状態をもつ構成要素が相互作用するシステム」(広義の複雑系) と、「構成要素の振舞いのルールが動的に変化するシステム」(狭義の複雑系) という二つの定義である (図 2.3)[12]。これらは、明確に分けることはできないが、現段階における理解の助けとしては、有効な区分であると思われる。以下では、この二つの定義について述べた後、これらのメタファーに期待される分析対象についての考察を行う。

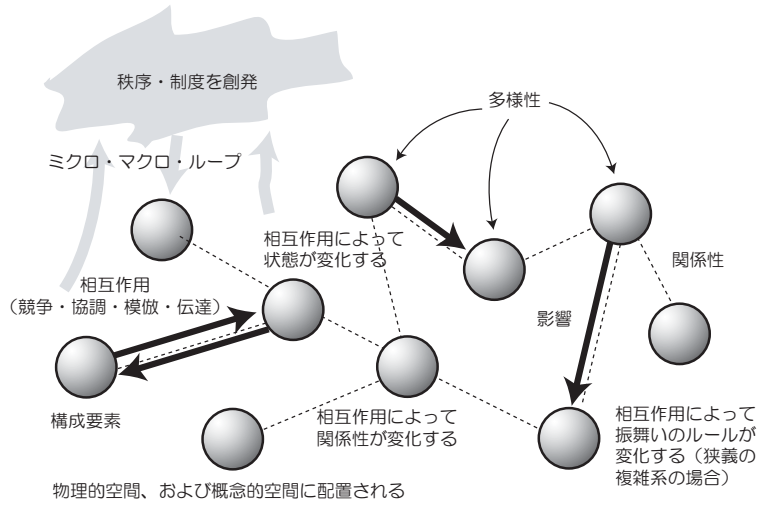


図 2.3: 複雑系のシステム観

2.2.1 広義の複雑系: 内部状態をもつ構成要素からなるシステム

本論文で定義する「広義の複雑系」とは、「内部状態をもつ構成要素からなるシステム」のことである。物理学をはじめとして自然科学では、対象をより小さな部分へと分解していくことにより、最終的には不変の最小単位 (アトム) に到達すると考えられてきた。原子論的なアトムである構成要素は、全体から切り取られても、切り取る前の性質を保ったままである。それゆえ、対象を要素に還元して理解し、その後、要素を足し合わせて全体を理解するという理解の仕方が可能となる。

これ対し、広義の複雑系の構成要素は、原子論的な意味でのアトムではなく、内部状態をもつという点に特徴がある。内部状態をもつということは、外から決めることのできない内部自由度をもっていることを表している。それゆえ、その振舞いを知るためには、いまどの状態にあるのかということを考慮する必要がでてくる。また、そのために、どのようにその状態に行き着いたのか、という文脈 (コンテクスト) についても、注意を払う必要がある。

社会科学では、社会を構成する人間は「物理学的なアトムではない」ということが繰り返し強調されてきた[13]。人間は複数の内部状態をもっており、それらはその人が置かれている状況や役割、体調などの要因によって刻々と変化していく。そして、その内部状態に依存して、価値基準や判断が変化したり、状況の認知や他者との関係が影響を受けるのである。

このように構成要素を捉えることは、内部に自由度をもった主体が、相互作用を行ってその状態を変化させていく点に注目するということである。このため、構成要素は、「自律的」(autonomous) であるといわれる。自律とは、外部からの作用が行なわれたとしても、自分自身の原理で処理することである[14]。

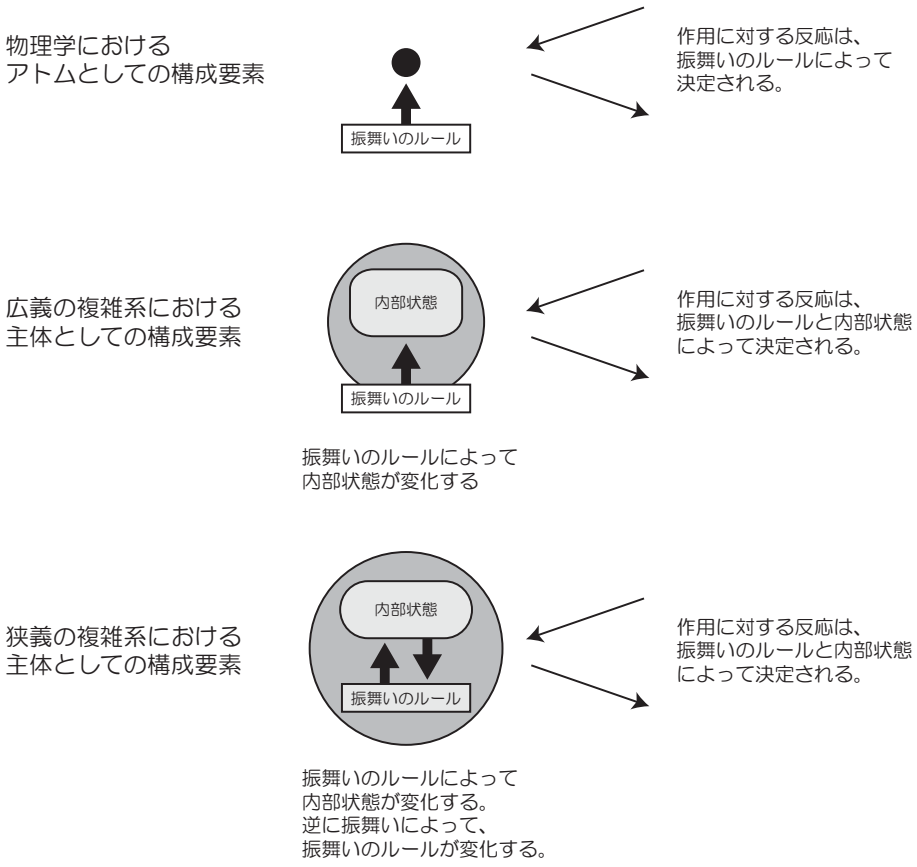


図 2.4: 物理学、広義の複雑系、および狭義の複雑系における構成要素の特徴

また、それぞれに状態が異なることから、それぞれの構成要素は、個性をもつことになり、それゆえマクロ的にみると多様性 (diversity) があるということになる。多様性というのは、同じ種類の主体同士であっても、それぞれがもっている特徴や置かれている社会的状況が様々であるということを指している。この意味で、広義の複雑系は「不均質なシステム」であるということができる。

2.2.2 狭義の複雑系: 構成要素の振舞いのルールが動的に変化するシステム

本論文で定義する「狭義の複雑系」とは、システムが内部状態をもつ構成要素からなっているという広義の複雑系のなかでも、「構成要素の振舞いのルールが動的に変化するシステム」のことである (井庭および福原, 1998)。

このような複雑系の定義を採用しているものには、Casti (1996)、塩沢 (2000)、安冨 (2000)、出口 (2000) などがある。Casti (1996) は、「エージェントは一定のルールに従って決断を下し、そこで得られた新しい情報に基づいて自発的に行動ルールを修

正できる」(邦訳 p.4) とし、特に「複雑系におけるエージェントは、グローバル (全体的) な情報ではなくローカル (局地的) な情報に基づいて決断を下し、みずから行動ルールを更新していく」としている。塩沢 (2000) は、「人間はその内部 (すなわち脳内) に外部世界 (すなわち環境) に関する仮説を構築し、外部世界を観察するとともに、その状況に応じて仮説を修正し、行動パタンを変える存在である」(p.61) と捉えることが「人間という行為主体をふくむ相互作用システムを複雑系とみる」ということであると指摘している。また、安冨 (2000) は、「要素の性質が変化するとシステムも変化せざるをえないが、要素はその変化に対応して再度変化することができる」システムとし、出口 (2000) は「主体のような自律的エージェントは、固有の活動ルールを持ち、それがさらに学習や創発、進化などにより変化する」(p.38-39) としている(15)。

2.2.3 進化: 変異を伴う複製

狭義の複雑系は、行動の「進化」という捉え方と関連が深い。経済学では近年、最適化原理における合理性の考え方に変わる枠組みとして、進化経済学という領域ができつつある (進化経済学会, 1998; 進化経済学会および塩沢, 2000; 江頭, 2002)。複雑系の場合と同様、進化経済学とは何かという確立された定義や明確な合意があるわけではないが、ひとまず「進化経済学は、制度・組織・技術・システムなどの多様性に注目し、内生的に進化するものとしてそれらを分析・研究しようとする」(有賀ほか, 2000) アプローチだと考えてよいだろう。産業化の経済学における進化論的アプローチの必要性を指摘した村上 (1994) は、河田 (1989) を引用して「進化とは、世代を通じて受け継がれていく生物の性質が変化していくこと」であるとし、次のような説明を加えている。「変わらないものが変わっていくというパラドックスが進化なのである。変わるものが変わるのは、単なる変化であって進化ではない。不変性を貫こうとする力とその不変なるものを変える力が絡み合う二重の機制が、進化に他ならない」(村上, 1994, p.119)。社会・経済においてこのような進化の対象だと思われるものには、制度、組織、技術、作業ルーティン (Nelson and Winter, 1982)、定型行動 (塩沢, 1998)、習慣、知識、商品 (Witt, 1997; Witt, 1998)、戦略などがあるだろう。進化的な視点で捉えるということは、生成されるものがまったくの無から生まれると考えるのではなく、既存のものを組み合わせたり、土台としてその上に新たなものを構築したりすることで生まれると捉えることになる。

2.3 複雑系と進化のメタファーに期待されていること

複雑系と進化というメタファーによって、私たちは社会・経済のどのような側面をより深く理解することができるのだろうか。ここではその期待される分析対象について整理することにしたい。

2.3.1　戦略とルーティンの進化

まず第一に戦略やルーティンが進化することの分析があげられる。ヴェブレン[16]などの古典的な進化経済学を現代的に復活させたネルソンとウィンターの『経済変化の進化理論』(Nelson and Winter, 1982) では、企業の決定ルールが基本的な操作概念として扱われている。この決定ルールは、「ルーティン」となって日々繰り返される[17]。ルーティンとは、「ものを作るための高度に特殊化された技術的手順や、雇用と解雇の手続きを通じた新規の在庫の指示、需要の多い項目の生産の増大といったことから、投資政策、研究開発、宣伝、製品の多様化と海外投資に関するビジネス戦略まで」(Nelson and Winter, 1982)[18]を含めた概念である[19]。このようなルーティンは、成功しているうちは維持されるが、業績悪化などを契機として再考され、組み替えられたり改良されたりすることになる。これを、進化的な視点でみると、「行動の代替的諸様式が相互に競争し、適切さの劣るものをふるい捨てる傾向をもつ淘汰過程が体系的に、また理解可能なかたちで作用する」(Nelson, 1998, p.8) と捉えることができるのである。このような視点は、人間や組織の意思決定を、その場その場の選択ではなく、それらの選択パターンの選択という、一段抽象度の高いレベルで捉えなおすということを意味している (塩沢, 1998)[20]。

2.3.2　意味と解釈を扱う社会モデルの構築

コミュニケーションにおける意味と解釈という問題、そして主体における世界像の問題を扱う道が開けてくる。知識は、「分散された諸断片としてだけ存在する」(Hayek, 1945, p.53) のであるが、人びとはそれらをコミュニケーションによって伝達する。コミュニケーションでは、伝達された情報をもとに、受け手は自らの解釈体系 (知識、メンタルモデル) を用いて意味を理解する。そのため、必ずしも意図した意味が伝達されるとは限らず、むしろあらゆる場合において、厳密に正確な伝達は不可能であるともいえる。従来の社会・経済システム論では、情報が多くの場合シグナルとして用いられてはいるものの、あらかじめその情報の意味が外生的に与えられてきた。これに対し、広義の複雑系では、構成要素に内部状態を想定することにより、同じ情報を受け取っても、受け手 (の内部状態) によって、異なる意味を得るということを含むモデルの探求を行うことが期待される。

また、情報は、単に主体のもっている解釈体系によって解釈されるだけではなく、解釈体系の変化を促す可能性ももっている。つまり、知識は社会を認識し思考するための枠組みを構成し、人々の行動に影響を与えるのである。そのため、「コミュニケーションは単に情報を伝達するのではなく、それは知識体系の発展と伝達に資することを理解することが重要である」(Boulding, 1985, p.150) のである。

このような、社会科学の文脈関係で考えてみると、現象学の影響を受けた社会学理

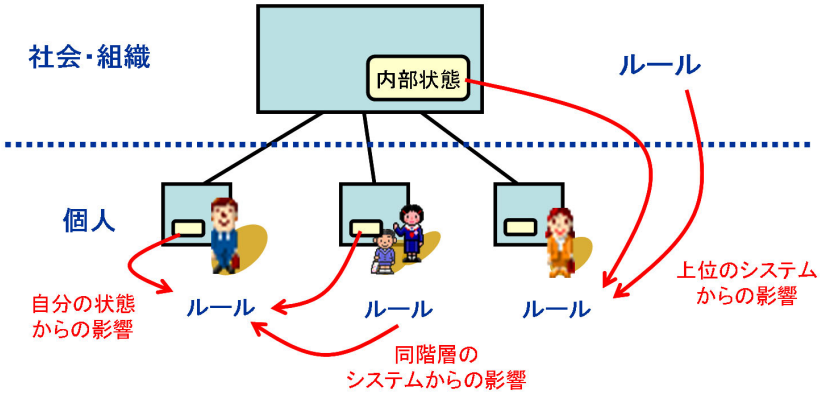


図 2.5: マクロからミクロへの影響

論などに関連が深いことがわかる。シュッツの現象学的社会学、バーガー=ルックマンの知識社会学 (Berger and Luckmann, 1966)、生活世界論などでは、主体による世界像の構築が重視されている。しかし、「現象学的社会学やエスノメソドロジーのように主体のリアリティの構成を重視する領域と社会システム論では、互いの概念に相互翻訳可能性がない状況が続いてきた」(出口, 2000, p.63) のであり、複雑系はこの二つの流れを接続するということが期待されている[21]。

2.3.3 制度と行動の関係の探究

社会科学において、このような狭義の複雑系が問題となるのは、主体の行動の結果生まれた秩序や制度が、また主体の行動に影響を及ぼすという循環的な関係を捉える必要があるからである (図 2.5)。このような要素 (部分) とその上位の階層 (全体) の間に循環的な関係があるからこそ、複雑系という対象は、要素還元的には理解し得ないことになる。なぜなら、部分に分けた時点で、その振舞いのルール (機能) を規定するメカニズムを、取り除いてしまうことになるからである。現在の行動は、これまでに起こったことの結果として採用されているのであり、状況からばらばらに切り離して考えることはできないのである。

秩序や制度の創発という問題は、これまでも社会科学では自己組織化の文脈の中で取り上げられてきた。自己組織化とは、システムが自身の構造をつくり替え、新たな秩序を形成することをいうが、自然科学における自己組織化と、要素が思考する社会科学における自己組織化は、同じ用語を用いていても異なる概念であるという点に注意が必要である。今田 (1986) は、「社会システムにおける創発特性は、その構成要素である人間個人が創発特性事態を主題化することにある。これは要素がシステムの全体を主題化することでもあり、人間行為者の自省作用によってはじめて可能である。」(p.186) と指摘している。つまり、秩序 (上位の階層) が形成された場合、物理・化学

であればそこまででであるが、社会システムを考えたときには、構成要素である人間が、その秩序を認識し、それをもとに行動を変えるといったことが起こる。それゆえ社会科学では、散逸構造やシナジェティクスなどで展開された形態形成としての自己組織化の理論で満足することはできず、さらに狭義の複雑系の自己組織化論へと展開されなければならないのである(22)。その点、社会学においては他の社会科学と異なり、この今田の自己組織性の社会理論やルーマンの社会システム理論などのような独自に発展してきた理論をもっており、これが狭義の複雑系と社会科学との接点となるだろう。

2.4 複雑系の記述と分析に関する課題

これまで、多くの研究者によって、複雑系や進化するシステムの捉え方の重要性が指摘されてきた。ところが、それは多くの場合問題意識として、あるいは認識の枠組みとして導入されているに過ぎない。このようなシステム観に則って社会科学研究を行おうとするならば、私たちは実際にシステムのモデルを記述し、操作できる必要がある。まさに現在、このような複雑系としての社会・経済を記述し操作する手段が求められているのである。

しかし、このような複雑系のモデルを厳密に記述するためには、従来の力学系を超えて、相空間の次元やルールなどの点で「開いた力学系」(金子および池上, 1998) の開発が必要となるが、現在のところ、そのような記述体系は考案されていない。また、これらの記述力に加えて、そのモデルの「操作性」が容易であり効率的である点も重要となる。モデルを操作する人間の能力は限られているため、モデルの操作性は社会科学にとって本質的に重要な点である。

さらに、モデル化の容易さや可読性の観点から、人間の「経験的感覚との対応」が取りやすい方法が望ましいといえる。それは、把握した社会のイメージを素直に、かつ直接的にモデルに表現できるならば、そのイメージとモデルの間に歪みが生じる可能性が少なくなるためである。厚東 (1991) は、「想像力なしには「社会」を認識することはできない」と指摘した上で、「モデルは、想像力を生き生きと活動させるための触媒」であるとし、「重要なのはモデルの導入によって、想像力が解き放たれるかどうかである」という。その点、わかりやすく記述されたシミュレーションモデルであれば、動的に変化する現象の表現力や説得力の面で優れているといえるだろう。

これらを踏まえ、本論文では、社会・経済を複雑系として「擬似的に記述する」ための体系について提案することにしたい。

2.5　シミュレーションによる計算と分析

2.5.1　シミュレーションと計算科学

本論文の対象である広義の複雑系や狭義の複雑系のモデルは、解析的に解くことが非常に困難であるか、あるいは不可能であるため、コンピュータ・シミュレーションによってモデルの特徴を理解するという方法がとられることが多い。シミュレーションとは、用意したモデルと初期条件からそのモデルを時間的に展開させるということであり、それを通じて、モデルの特徴についての経験的な知見を得ることができる[(23)]。また、シミュレーションでは、モデルの設定や条件などを変更して試すことが容易であり、頭の中ではもはや自由に操作することのできないような大規模で複雑なモデルを扱うことも可能となる。そのため、現在の科学的研究においては、シミュレーションは理論を発展させるための非常に重要な方法となっているのである。

コンピュータ・シミュレーションを用いた科学的研究は、1986 年に K. G. Wilson によってその必要性が提唱されて以来、「計算科学」(computational science) として発展しつつある[(24)]。計算科学は、「科学や工学の問題を解決するため、シミュレーションや実験データ解析にコンピューターを積極的に利用して、理論や実験と補完し合う手段（実験と理論的アプローチの間にあるギャップを埋める）」(田子, 1998) というものであり、科学研究の両輪と言われる「理論」と「実験」に加えて「計算」を重視する。なお、シミュレーションをコンピュータ上における「実験」と捉えることもあるが、シミュレーションは現実内における実験とは性格が異なることに注意が必要である。実験は、現実世界の中で対象によって現象を生成し、それを仮説と比較することによって経験的な知見を得る手段である。これに対し、シミュレーションは、人工的な思考世界の中で仮説モデルから現象を生成し、それを現実世界の現象と比較することによって、経験的な知見を得る手段である。この違いは、得られた結果の妥当性と関係するため、意識する必要がある。

2.5.2　科学的研究におけるシミュレーション利用

科学的研究におけるシミュレーションの利用法には、大きく分けて次の 3 つのアプローチがある。第一のアプローチは、対象の将来に関する「予測」である。予め妥当と思われるモデルがあり、それを時間経過させることによってどのような結果になるのかを観察・分析するというものである。つまり、過去のデータを用いてシミュレーションを行うことによって、将来の動向を予測するのである。第二のアプローチは、対象の「特徴についての理解」である。部分モデルの振る舞いがわかっている場合に、それらを組み合わせると全体としてどのような振舞いをするのかを観察するために用いられるのである[(25)]。第三のアプローチは、対象の「内部メカニズムについての理解」である。これは、全体的な振る舞いがわかっているが、内部のメカニズムがわかっ

ていないという対象を理解したい場合に行われる。内部メカニズムの仮説的なモデル(構成モデル)を作成し、その振る舞いと対象を比較して改良し、徐々にモデルを対象に近づけていくのである。

これらのアプローチの中で、特に「予測」については、ビジネスや政策分析からの現実的要請として求められることが多いものの、シミュレーション研究者の中では、このような利用の効果を疑問視する声も多い。社会シミュレーションの先駆者である J. W. Forrester も、予測については、当初から懐疑的な意見を述べている。「とくに注意したいのは、将来の特別な時点における 特定の事象 の定量的な予測が、モデルの目的には含まれていないということである。従来、有用なダイナミック・モデルならば、ある将来の時点におけるシステムの特定の状態を予測できなければならないということは自明である、と間違って考えられてきた。これは望ましいことかもしれないが、モデルの有用性は、未来における特定の進路を予測する能力にかかっている必要はない。」(Forrester, 1961, 下線は原文より)。また、Gilbert and Troitzsch (1999) も、「従来の社会科学の科学哲学では、説明と予測を過剰に関係づけてきたと言えるだろう。つまり、理論をテストするのに、その理論がうまく将来を予測できるかどうかで判断される傾向があるのである。これは非線形理論、特にミクロレベルにおいては、適切な判断基準であるとはいえない。」と注意を促している。

これらの指摘からもわかるように、現在では、理論を発展させるための方法として、「特徴についての理解」や「内部メカニズムについての理解」に重きが置かれることが多い。「特徴についての理解」というのは、例えば、現象の発生頻度やネットワーク構造などについてのマクロ的な特性を知るということである。多数の要素が相互作用するシステムでは、べき乗法則という特性がよくみられる。例えば、砂山における雪崩の規模と頻度、地震の規模と頻度、そして価格変動の規模と頻度の関係は、べき乗法則に従っていることが知られている。また、最近のネットワーク理論では、友人関係や経済ネットワーク、ワールド・ワイド・ウェブ (WWW) など、成長するネットワークにおいても、べき乗法則が見られることが明らかになっている。しかし、私たちが知ることができるのは、このような全体的な特性とそのメカニズムのみである。どのタイミングでどの規模の現象が起きるのかということや、どの点とどの点がリンクされるのかというミクロレベルの予測は非常に困難、もしくは不可能である。

「内部メカニズムについての理解」は、複雑系研究でよく行われており、「構成的手法」や「構成による分析」(analysis by synthesis) と呼ばれている。従来のような還元的な方法では分析できない複雑系のモデルを、コンピュータ上にヴァーチャルに構成し、そのシミュレーションの振る舞いを観察しながらモデルを修正していくのである。

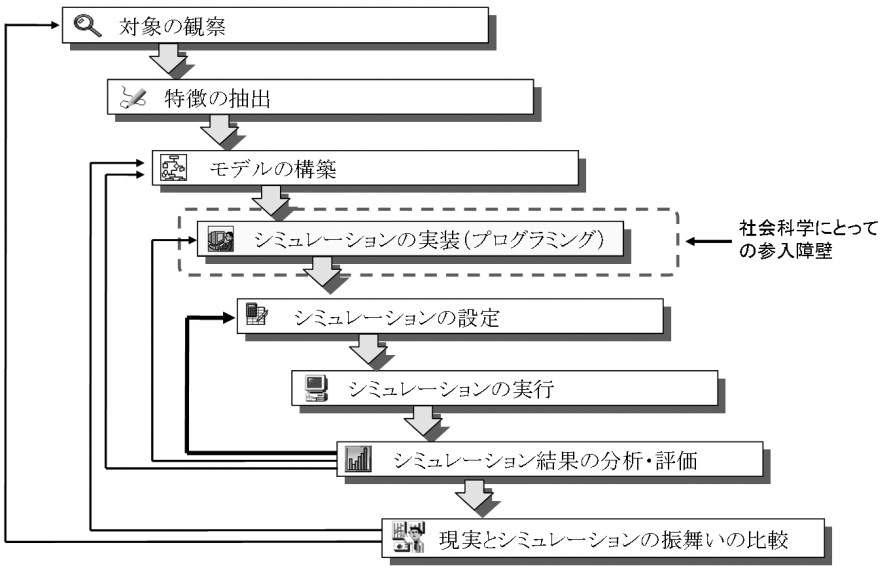


図 2.6: シミュレーション研究の典型的なフロー

2.6 シミュレーションの作成に関する課題

シミュレーションによる研究方法には特有の問題が存在する。それは、コンピュータ・シミュレーションが一種のコンピュータ・プログラムであるため、モデルだけでなくプログラムについても注意を払う必要があるという点である。シミュレーションによる研究プロセスは、一般に、「対象の観察」、「特徴の抽出」、「モデルの構築」、「シミュレーションの実装 (プログラミング)」、「シミュレーションの設定」、「シミュレーションの実行」、「シミュレーション結果の分析・評価」、「現実とシミュレーションの振舞いの比較」というフェーズで構成されている (図 2.6)。このように、社会科学理論以外の作業が多く存在する。それゆえ、シミュレーションを作成するためのプログラミングや、研究分野全体におけるプログラムの共有や蓄積までを含めて考えなければならないのである。

プログラミングの必要性は、シミュレーションのソフトウェア品質に必要な技術を含めると、社会科学者などが参加する際の障壁となっている。シミュレーションが正しくプログラムにコード化されているかどうかの判定を「正当性の検証」(verification) というが、シミュレーションの場合、モデルが複雑になるほど結果の事前予測が難しくなり、プログラムの正当性の検証が困難になる。そのため、事後的に正当性をテストするという方法に頼るのではなく、シミュレーションの開発プロセスの中にソフトウェア品質を確保するための仕組みを導入する必要がある。一般にソフトウェアには潜在欠陥はつきものであるため、開発過程や保守過程における徹底的な品質管理が重

要となる。言うまでもなく科学的研究や政策分析で用いられる場合、「正しさ」に関する品質はゼロ欠陥であることが求められる。ところが、このゼロ欠陥というのは、専門的なプログラマでさえ実現が難しい要求である[26]。潜在欠陥はソフトウェアの規模が大きくなるほど増大するが、欠陥除去率は規模が大きくなるにつれて低下することが知られているため、今後シミュレーション・プログラムにおいても問題が深刻化することは必至である。そのため、研究者への負担は最小限に留めたままで、ソフトウェア品質の高いシミュレーションを作成することを支援する仕組みが求められる。

また、従来のシミュレーション支援システムでは、作成したモデルの一部を他の研究者と交換したり再利用したりすることを支援する仕組みは提供されていない[27]。そのため、シミュレーションにおけるモデルの再利用はほとんど行われず[28]、その都度ゼロから作られることが多い。実際、これまで作成されてきたシミュレーションモデルを調べてみると、モデルの構成単位レベルでは共通部分が多いため、これらを重複して開発するというのは非効率であることがわかる (表 2.1, 2.2, 2.3)。このようにその都度ゼロから作るという開発方法は、モデルが大規模になるようになるにつれて開発時間とコストの面で限界が生じることが予想される。これはまさに、シミュレーション分野における「ソフトウェア危機」と言えるだろう。このような状況に対し、開発時間の短縮とコストの低減のために、モデルの再利用や複数の開発者による独立した並行開発などの対処が課題となっている。

表 2.1: 先行研究のモデルにおけるエージェント (1)

モデル化の対象	実装言語	エージェント	エージェントの行動
株式市場 (Palmer et al., 1994)	Objective-C, Swarm	トレーダー	株価予測, 注文, 株売買
株式市場 (山本ほか, 2001)	C++(X-Mart)	トレーダー	売買のタイミング決定, 注文, 株売買
株式市場 (横田および小林, 2001)	VC++	投資家	株価予測, 注文, 株売買
株式市場 (Iba, 1999)	Java	トレーダー	株価予測, 注文, 株売買
先物取引市場 (佐藤ほか, 2001)	Java(U-Mart)	取引所会員	株価予測, 注文方法・注文量・注文価格決定, 注文, 売買
外国為替市場 (和泉および植田, 1999)	Pascal	ディーラー	レート予想, 戦略決定, レート・注文量決定, 注文, 売買
商品市場 (水田ほか, 2000)	Java, Java (ASIA)	生産者 投機家 オークション仲介者	生産, 消費, Bid, 売買 価格推定, Bid, 売買 Bid 要求, Bid 集計, 価格決定
オンラインオークション (水田, 2001)	Java, Java (ASIA)	買い手 オークショナー	状況確認, Bid, 購入 Bid 要求, Bid 集計, 落札者・落札価格決定
排出権取引市場 (Mizuta and Yamagata, 2001)	Java (ASIA)	Nation COP	国内削減量決定, Bid, 取引 Bid 要求, Bid 集計, 価格決定, 取引許可
サプライチェーン (谷口ほか, 2001)	IF/Prolog	工場 事業部 販社	生産, 在庫, 出荷 調達, 在庫, 出荷 調達, 在庫, 販売
規格競争 (VCR)(井庭ほか, 2001)	C, Java(BESP)	消費者	欲求認識, シェア認知, 商品購入
規格競争 (フリーソフトウェア)(Dalle and Jullien, 2000)	MATLAB	潜在的採用者	シェア認知, 商品購入
環境マーケティング (石川および寺野, 2000)	Delphi	消費者 生産者	商品購入 商品企画, 商品販売, 倒産, 行動の模倣, 行動規範の模倣

表 2.2: 先行研究のモデルにおけるエージェント (2)

モデル化の対象	実装言語	エージェント	エージェントの行動
貨幣の自生と自壊 (安冨, 2000)	C	経済主体	財の生産, 物々交換, 記憶
社会 (Sugarscape) (Epstein and Axtell, 1996)	C, Java (Ascape)	架空の主体	移動, 収集, 消費, 生殖, 文化伝播, 略奪, 価格交渉, 物々交換, 融資, 返済, 疾病感染, 免疫応答
市場経済 (吉地および西部, 2000)	C, C++	企業 消費者	稼動率調整, 価格調整, 生産, 販売 購入
ケインジアン経済 (Bruun, 1997)	Pascal	消費者 消費財生産者 資本財生産者	消費, 労働, 商品購入, 許容証券価格決定, 証券売買 生産, 雇用, 投資決定, 商品配送, 証券発行, 許容証券価格決定, 証券売買 生産, 雇用, 投資・注文決定, 資本財引渡, 証券発行, 許容証券価格決定, 証券売買
アメリカ経済 (ASPEN) (Basu et al., 1998)	C, C++	家計 食品製造業 非耐久財製造業 自動車製造業 住宅建設業 銀行 政府 連邦準備 不動産業 資本財製造業 「金融市場」	労働, 商品購入, 貯蓄・引出, 国債売買, 納税, 失業保険受取, 社会保障受取, ローン借入・返却 生産, 価格決定, 設備投資, 雇用, 販売, 納税, ローン借入・返却 生産, 価格決定, 設備投資, 雇用, 販売, 納税, ローン借入・返却 生産, 価格決定, 設備投資, 雇用, 販売, 納税, ローン借入・返却 生産, 価格決定, 設備投資, 雇用, 販売, 納税, ローン借入・返却 預金・引出受入, 貸出・返却受入, 国債売買, 雇用, ローン貸出・返却受入, 中央銀行への準備預金預入・引出 徴税, 失業保険支給, 社会保障支給, 国債発行, 雇用 銀行への貸出・返却受入, 国債売買 賃貸料徴収, 雇用, 納税 生産, 雇用, 販売, 納税 国債取引の調整

表 2.3: 先行研究のモデルにおけるエージェント (3)

モデル化の対象	実装言語	エージェント	エージェントの行動
バーチャル経済 (出口, 2000)		政府	公共投資, 雇用, 有利子国債発行・償還受入, 国債発行・償還受入, 徴税, 税率決定, 国債金利決定, 補助金支給
		銀行	預金・引出受入, 貸出・返却受入, 中央銀行からの借入・返却, 中央銀行への準備預金預入・引出, 預金金利決定, 貸出金利決定
		中央銀行	国債引受・償還, 銀行への貸出・返却受入, 銀行からの準備預金預入・引出受入, 公定歩合決定
		パン製造業	価格決定, 原料購入, 生産, 販売, 設備投資, 雇用, 貯蓄・引出, 借入・返却, 納税, 有利子国債購入・償還, 補助金受取
		製粉業	価格決定, 原料購入, 生産, 販売, 設備投資, 雇用, 貯蓄・引出, 借入・返却, 納税, 有利子国債購入・償還, 補助金受取
		農家	価格決定, 生産, 販売, 設備投資, 雇用, 貯蓄・引出, 借入・返却, 納税, 有利子国債購入・償還, 補助金受取
		機械製造業	価格決定, 原料購入, 生産, 販売, 設備投資, 雇用, 貯蓄・引出, 借入・返却, 納税, 有利子国債購入・償還, 補助金受取
		製鉄業	価格決定, 生産, 販売, 設備投資, 雇用, 貯蓄・引出, 借入・返却, 納税, 有利子国債購入・償還, 補助金受取
		家計	労働力供給, 購入, 住宅投資, 貯蓄・引出, 借入・返却, 納税, 有利子国債購入・償還, 補助金受取

第3章　オブジェクト指向計算モデルの導入

3.1　どのように写し取るのか: 計算モデルの導入

モデル化方法について考えるためには、対象を「どのように写し取るのか」(how) ということを考える必要がある。本論文の対象となる複雑系のモデルは、従来の言語モデルや図式モデル、数学モデルで扱うことが困難であることから[29]、ここでは、「計算モデル」(computational model) という表現形式を導入することにしたい。計算モデルは、コンピュータ・サイエンスとソフトウェア工学の分野で考えられ発展してきたものであり、処理の種類と対象を記述してあるため、「計算」(computing)[30]を行うことができる。

計算モデルは、数学モデルに比べて、社会科学のモデル化に適していると言われている。例えば、Gilbert and Troitzsch (1999) では、計算モデルは (1) 現実との対応関係の把握が容易であり、(2) 並列的なプロセスや順序の決まっていないプロセスの扱いが容易であり、(3) モジュール性[31]をもたせることが容易であり、(4) 多様な主体を組み込んだモデルの構築が容易であるとしている。また、Knuth (1985) は、数学とコンピュータ・サイエンスを比較して、次のような暫定的な結論を導いている。数学には、「複雑度」すなわち「操作の節約」 という概念がほとんどなく、また過程の状態に関する動的な概念 (代入の概念) がないのに対し、コンピュータ・サイエンスでは、同時に実行される諸過程間の相互作用を研究するときにも状態の概念が重要となり、また、本質的に均質でない諸概念に柔軟に対処できる、としている。

計算モデルの代表的なものには、手続き型計算モデルや関数型計算モデル、論理型計算モデル、オブジェクト指向計算モデルなどがある (図 3.1)。近年の計算モデルの発展は、「命令から宣言へ、手続きからオブジェクトへ、逐次集中から並列分散へ」(青木 1993) という方向にある。第一の点は、「これをやってから、次にこれをやる」というように詳細な計算手順を命令的に記述するスタイルから、「これはそれとこのような関係にある」というように、計算の意味を宣言的に記述するスタイルになってきているということである。第二の点は、主に計算手順を記述するスタイルから、計算手順とデータをひとまとまりとして扱うようなスタイルになってきているということである。そして第三の点は、計算の実行がひとつのところで集中的に行われるというスタイルから、分散して存在する複数の実行部が、協調して計算を行うというスタイルが多くなってきたということである。このような方向性によって、計算モデルの特

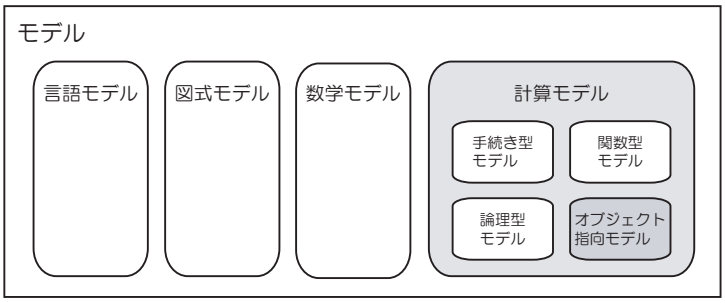


図 3.1: モデルの種類

徴がますます強化されているが、そのなかでも特に有力なパラダイムのひとつが、本稿で導入する「オブジェクト指向」である。以下では、オブジェクト指向によるモデル化について、簡単に見ていくことにしよう。

3.2 オブジェクト指向計算モデルの考え方

計算モデルのなかでも、オブジェクト指向によるモデル化は、人間の認知の仕組みに近く、より自然なモデル化が可能であるといわれている。「オブジェクト指向」では、物理的あるいは概念的なモノのひとつひとつを「オブジェクト」として捉え、その状態や関係の変化で対象となる現象をモデル化する。ここでいう「オブジェクト」とは、現実に存在する有形無形の「モノ」(thing) を表す一つのまとまりのことである。日常的な例でいうと、机や椅子、テレビ、携帯電話、新聞、人間、犬、観葉植物などは、どれもオブジェクトとして表現することができる。モデルの観点からいうと、オブジェクトとは、現実世界における個々のモノを、その「状態」と「振る舞いのルール」をひとまとめにして (カプセル化して) 表現したものであり、もっとも基本的な思考の単位となる。

オブジェクト指向では、さまざまなオブジェクトが、それぞれ役割を分担しながら相互作用することで世界が動いていると捉える。オブジェクト間の相互作用は、互いにメッセージを送り合うことで表される。オブジェクトは、自分の知っている (リンクをもつ) オブジェクトに対してメッセージを送ることができ、そのメッセージを受けたオブジェクトは自分の状態を変化させたり、必要があれば何らかのメッセージを送り返したり、他のオブジェクトにメッセージを送ったりする。これが、オブジェクト指向の基本的な世界観である (図 3.2)。状態と振る舞いのルールを保持しているオブジェクトが多数存在し、それらが相互作用しているという点が、オブジェクト指向の本質である。

オブジェクト指向が現実世界のモデル化に適しているのは、実はオブジェクト指向が考案された背景が関係している。オブジェクト指向の起源は、1960 年代にノルウェーで

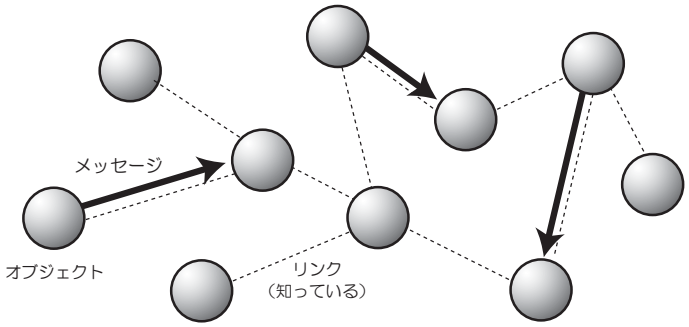


図 3.2: オブジェクト指向のイメージ

開発された Simula というコンピュータ言語にさかのぼる (Dahl and Nygaard, 1966)。Simula は ”simulation language”の略で、その名が示す通り、現実世界のさまざまな現象をシミュレートするためのコンピュータ言語であった。Simula は、何千もの構成要素からなる複雑なシステムのモデルをコンピュータ上で動かすことを目的に設計されたため、動的で複雑な現実世界をそのままコンピュータ上に取り込むための工夫がなされている。このような問題意識から生まれた考え方だからこそ、オブジェクト指向は、複雑な現実をモデル化する際の有力な方法となり得るのである[32]。

3.3 クラスによるモデル設計

3.3.1 概念とクラス

オブジェクト指向がもつ記述力を発揮するためには、「クラス」という考え方を導入する必要がある。「クラス」とは、複数のオブジェクトを共通の性質ごとに分類したもののことである。クラスを用いることによって、オブジェクトの体系的な整理が可能となる上、効率的な記述が可能となる。モデル化の対象となる現実世界はさまざまなものから構成されているため、これらを個別に把握したりモデル化したりしようとすると、時間と手間が膨大にかかってしまう。クラスの考え方を用いれば、よく似たオブジェクトを個別記述していかなくても、共通項を一括して表現できるようになる。

オブジェクトをクラスに分類するという仕組みは、実は、人間の認知プロセスにおける「概念化」と同じメカニズムである。私たち人間は世界のさまざまな物事を認知するとき、複数のものを共通する性質に着目して、概念を用いてまとめて把握したり、体系化したりしている (図 3.3)[33]。具体的なものと概念の関係、そしてオブジェクトとクラスの関係の対応を図示すると図 3.4 のようになる。認知における「具体的なもの」と、オブジェクト指向におけるオブジェクトは、それぞれ概念とクラスの「インスタンス」(実例) と呼ばれる (図 3.5)。クラスは状態をもたないが、そのインスタンスであるオブジェクトは必ず状態をもつ。個別のものをクラス (概念) で分類するとい

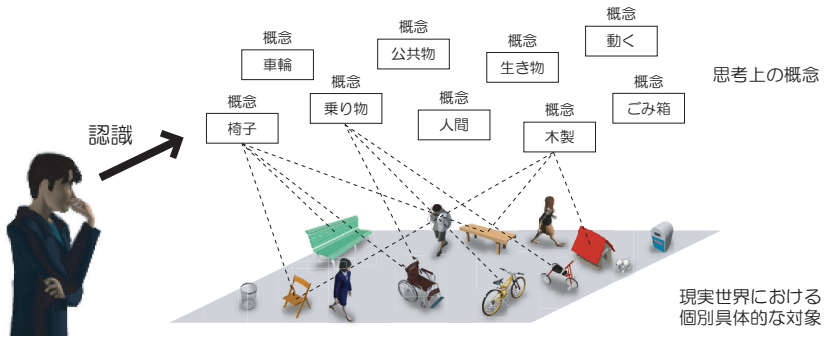


図 3.3: 現実認識における概念

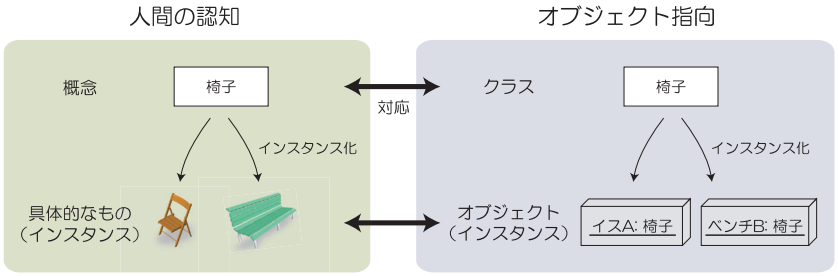


図 3.4: 人間認知における「概念」とオブジェクト指向における「クラス」

うことは、世界の複雑さに対処するためのひとつの方法なのである。

オブジェクト指向では世界の動きを複数のオブジェクトの相互作用で表現することはすでに述べた通りであるが、このときもそれぞれのオブジェクトを生成するクラス同士の関係として一括して定義することができる。オブジェクト同士の関係は、それぞれの型であるクラスどうしの関係に準ずるからである。クラスの関係は、関連 (association) と呼ばれる。関連のインスタンスはリンク (link) と呼ばれ、関連するそれぞれのクラスのインスタンスであるオブジェクト間の意味的な結び付きに用いられる (図 3.6)。あるクラスが他のクラスとまったく関連を持たなければ、そのクラスのインスタンスであるオブジェクトは、協調作業することのない孤立したオブジェクトということになる。

3.3.2 クラス間の関連

人間は現実世界を認知する際に、その複雑さに対処するために複数の概念を関係づけるメカニズムを利用しているのだが、オブジェクト指向でも同様に複数のクラスを関係づけるメカニズムを用いる。ここでは、特によく用いられる「汎化／特化」と「集

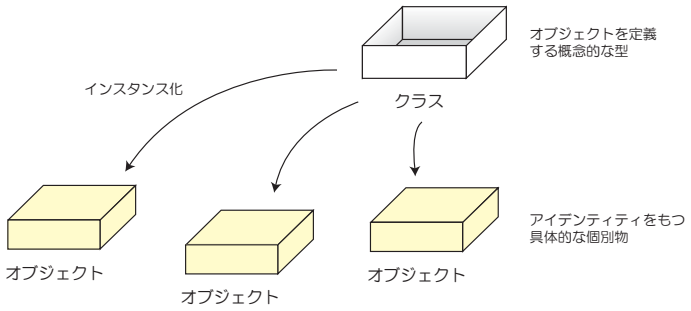


図 3.5: クラスとオブジェクト

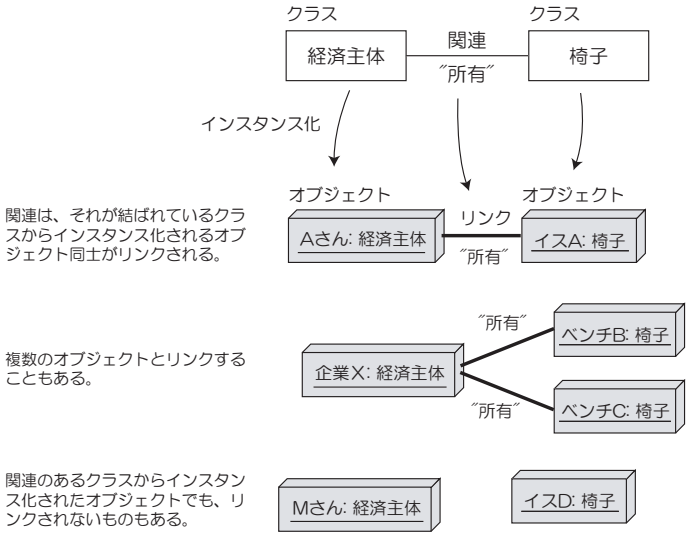


図 3.6: クラス間関係とオブジェクト間関係

約」について説明する[34]。

「汎化 / 特化」(generalization / specialization) の関係は、共通する属性や振舞いを一括して定義した上位クラスと、特化した下位クラスによる階層関係である (図 3.7)。この階層において、上位クラスは下位クラスを「汎化」したものであり、逆に下位クラスは上位クラスを「特化」したものとなる。この関係は、「a-kind-of」(〜の一種である) ということができる。汎化／特化の関係を用いる利点は、あるクラスが他のクラスよりも包括的であることを識別することで、クラスを体系化することができる点にある。また、この関係を用いて、少しだけ違うようなもの同士を体系化して整理することもできる。また、モデル記述の観点から言えば、共通の振る舞いや属性は上位のクラスで一度だけ記述すれば良く、下位クラスで記述する必要がないという利点もある。おそらく、クラス階層の下位クラスでは上位クラスにはない特化した属性や振舞いをもっていると思われるが、上位クラスの属性や振舞いはすべて下位クラスが

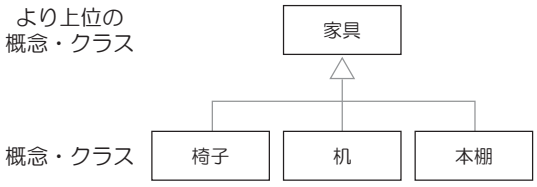


図 3.7: 概念の特化／汎化

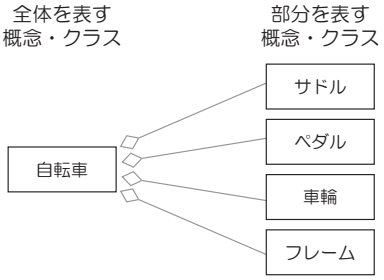


図 3.8: 概念の集約

引き継ぐ。これを「継承」というのだが、このメカニズムによって、下位クラスには、上位クラスと下位クラスとの差の部分のみを記述すればよいということになる。

複雑さに対処するためのもうひとつのメカニズムは「集約」(aggregation) である。集約は、あるオブジェクトが、別のオブジェクトから構成されていることを表す関係である。コンポジション (composition) の関係は、全体をあらわす全体クラスと、その構成要素にな部分クラスの関係であり、「composed of」(〜から構成される) ということができる (図 3.8)。コンポジションの利点は、多くの部分から組み立てられているものを 1 つの全体として扱うことができる点にある。全体に対する属性が、部分についても適用できるのである。例えば、「自転車」オブジェクトは、「サドル」、「ペダル」、「車輪」、「フレーム」などのオブジェクトから構成されている。この自転車が他の場所に移動すれば、その部品も移動するだろう。また、オブジェクトの構成要素が、時間とともに変化することを許すことでもある。先ほどの「自転車」オブジェクトにある時点で「ライト」が装着されるかもしれないが、その後は「ライト」オブジェクトも含めて移動することになるのである[(35)]。

3.4 UML(統一モデル化言語)

オブジェクト指向のモデルを記述するための記法は、近年、UML(Unified Modeling Language: 統一モデル化言語) として、標準化されている。UML は、「ソフトウェアシステムの成果物を規定、構築、可視化、文書化する言語」(Object Management Group,

2000) であり、「大規模で複雑なシステムをモデル化する上でその有効性が実証された工学上の最良の実践を収集し、代表するもの」(Object Management Group, 2000) である。UML の開発の背後にあった目標の中で、「第 1 の最も重要な目標は、UML がすべてのモデル作成者が利用することのできる汎用のモデリング言語となること」(Rumbaugh et al., 1999) であり、そのため「UML は所有権の設定されたものではないと同時に、コンピュータ業界の大多数による共通の合意の基づいたもの」(Rumbaugh et al., 1999) であるという。また、「できるだけシンプルでありながら、それでいて構築しなければならない広範な実用システムをモデリングできるようにすること」(Rumbaugh et al., 1999) が目指された(36)。

本論文で用いる UML 図の記法については、付録 A にまとめておく。

第4章　モデル・フレームワークの提案

4.1　モデル・フレームワークとは

本章では、複雑系として社会・経済をモデル化する際によく現れる要素や構造を抽出し、「モデル・フレームワーク」として定義する。ここでは、抽象度の違いにより、「概念モデル・フレームワーク」と「シミュレーションモデル・フレームワーク」という2種類のフレームワークを考える。これらのフレームワークは一貫性を有しており、概念モデルの作成からシミュレーションモデルの実装までをシームレスに行うための支援をする (図 4.1)。

4.1.1　概念モデル・フレームワーク

概念モデル・フレームワークは、対象についての概念モデルを作成する際に、共通して登場する要素と構造を定義したものである。モデル作成者は、モデル化しようとしている対象が「どのようなものであるか」(What) を洗い出し、記述する際に、この概念モデル・フレームワークを用いることができる。加えて、概念モデル・フレームワークがシミュレーションモデル・フレームワークと一貫性を有することから、シミュレーションモデルへの移行をシームレスに行うことができる。概念モデル・フレームワークは、(1) 現実世界の認識のための準拠枠の明示化、(2) 概念モデルを記述するための語彙の提供、(3) 概念モデルの共有化とコミュニケーションの支援、という3つの役割を果たす (図 4.2)。

(1) 現実世界の認識のための準拠枠の明示化

概念モデル・フレームワークは、対象となる現実世界を認識する際の準拠枠となる。概念モデル・フレームワークを、「世界の中に含まれるものを拾い集めようとして用いる、一種のふるい」(William, 1925) として用いることにより、動的で捉えどころのない世界を把握することが容易になる。

私たち人間は、現実世界からありのままの「事実」というものを受動的に受けとっているのではなく、認知枠のフィルターを通じて能動的に選びとっている。生体的・認知的制約から現実世界のすべてを認識することはできないため、重要と思われる部分に焦点を当てて把握しているのである。このように、「知覚する」という行為は「枠

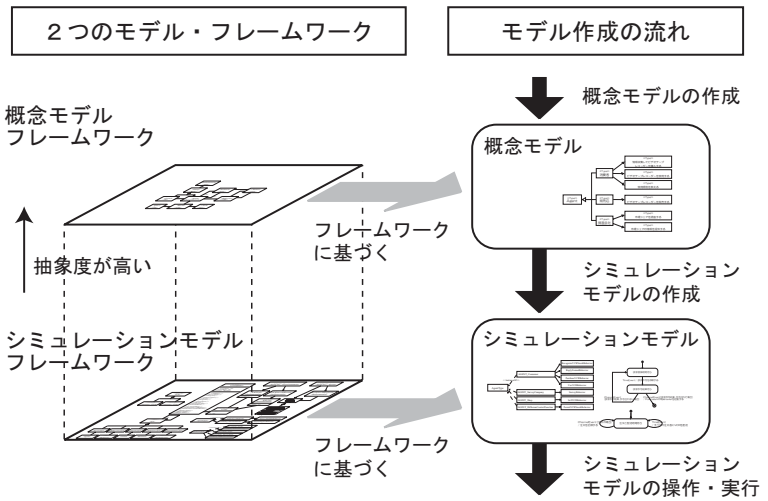


図 4.1: ２つのモデル・フレームワークとモデル作成の流れ

組みのなかへの定位」(村上, 1971) を意味している。前提をおくことによってそれが認識の枠組みとなり、動的で捉えどころのない世界をある側面から把握できるようになるのである(37)。

(2) 概念モデルを記述するための語彙の提供

概念モデル・フレームワークにおけるモデル要素は、モデルを記述するための語彙として用いることができる。認識や分析で得られた概念は、それを記述するためのなんらかの表現手段が必要となるが、概念モデル・フレームワークは、そのドメインに特化した語句と、それらの可能な組み合わせの規則を提供する。

認識や分析で得られた概念は、そのままでは形がないため、それを記述するためにはなんらかの具体的な表現手段が必要である。適度の長さと複雑さをもつ表現を用いて思考し表現するのであるが、それを効率よく運用するために、表現の意味を記号に圧縮する (藤村, 1999)。このような言語——ここでの対象としてはモデル化のための言語——は、表現単位となる語句(38)と、その組み合わせの規則(39)とによって成り立っている。このような対象世界を表現するための体系によって、私たちは安定的に、あらかじめ定められた有限個の記号を用いて、並べ方の形式に従って組み合わせて表現することができるようになる。

(3) 概念モデルの共有化とコミュニケーションの支援

概念モデル・フレームワークは、複数のモデル作成者に共通の視点を与えるため、モデル要素の粒度や捉え方がモデルごとに異なってしまうという問題を回避できる。

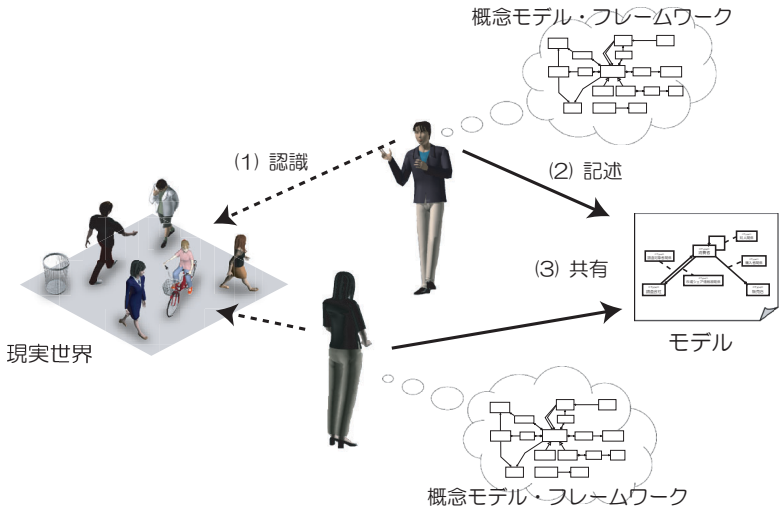


図 4.2: モデル作成における概念モデル・フレームワークの役割

また、共有された語彙を用いることで、詳細な情報を伝達することなく重要な点を強調することができるため、円滑で効率的なコミュニケーションが可能となる。

一般にコミュニケーションは、発信者が決まりに従って表現し、受信者はその決まりによって読みとって理解する。この決まりを「コード」と呼ぶのであるが、「理想的なコミュニケーションでは、話し手と聞き手が「メッセージ」の作成と解読に際して利用する「コード」は同一のもの」(池上ほか, 1994) である場合で、特に科学的コミュニケーションにおいては重要となる[40]。共通の枠組みを用いることで、認識・分析して得られたものをそのまま記述し、他者へ伝達し、知的蓄積へ積み上げていくことができるのである。

4.1.2 シミュレーションモデル・フレームワーク

シミュレーションモデル・フレームワークは、得られた概念モデルを、シミュレーションモデルとして「どのように実現するか」(How) を規定するものである。シミュレーションモデル・フレームワークは、ソフトウェア・フレームワークとして実行環境の一部となることで、シミュレーションモデルを実行することができる。シミュレーションモデル・フレームワークは、(1) シミュレーションモデルへの変換方法の明示化、(2) シミュレーションの実装の支援、(3) シミュレーションモデルの共有化と再利用の支援、という 3 つの役割を果たす。主に次の 3 つの役割を果たす (図 4.3)。

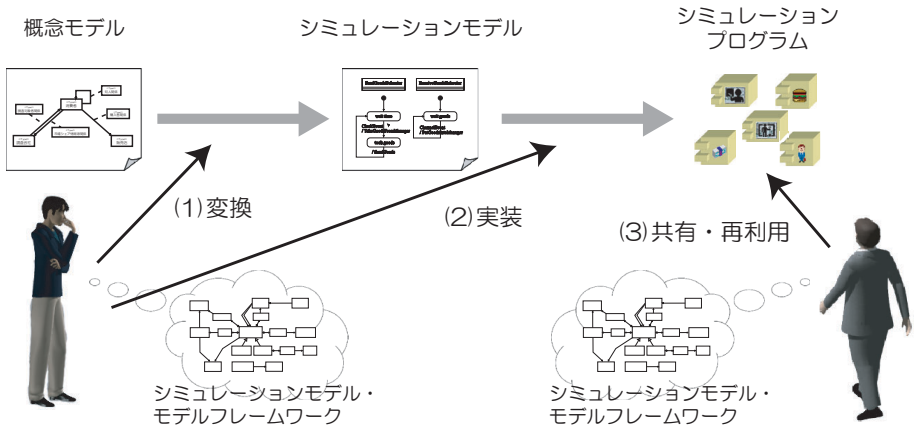


図 4.3: シミュレーション作成におけるシミュレーションモデル・フレームワークの役割

(1) シミュレーションモデルへの変換方法の明示化

シミュレーションモデル・フレームワークは、シミュレーションモデルとして何をどのように記述すべきかを明らかにする。概念モデル・フレームワークと一貫性があるため、概念モデルをシミュレーションモデルに変換する手続きを事前に規定することができる。

(2) シミュレーションの実装の支援

シミュレーションモデル・フレームワークは、シミュレーションモデルの実装の方法を明らかにする。そして、シミュレーションモデル・フレームワークは、シミュレーションの基本的な仕様の定義を行うため、シミュレーションの実行に関するプログラムや環境をあらかじめ用意しておくことが可能となる。また、一部のプログラムを自動生成する作成支援ツールを開発することができるようになる。

(3) シミュレーションモデルの共有化と再利用の支援

シミュレーションモデル・フレームワークは、モデルコンポーネント間の仕様やそれらの接続方法を規定するため、モデルコンポーネントを共有したり、再利用したりするための仕組みを提供する。

4.2 提案モデル・フレームワーク: Boxed Economy Foundation Model (BEFM)

複雑系として社会・経済のモデルを作成するためのモデル・フレームワークとして、「Boxed Economy Foundation Model」(以下、BEFM) を提案したい[41]。以下では、概念モデルの作成を支援する「BEFM 概念モデル・フレームワーク」と、シミュレーションモデルの作成を支援する「BEFM シミュレーションモデル・フレームワーク」について、順に説明する。

4.2.1 BEFM 概念モデル・フレームワーク

BEFM の主な特徴としては、エージェント間の相互作用を、財 (情報が付随することがある) のやりとりとして明示化するという点と、エージェントの行動を、エージェントとは別のモデル要素として定義するという点である。BEFM 概念モデル・フレームワークは、World、Space、Clock、Entity、Agent、Goods、Information、Behavior、Relation、Channel などのクラス (型) から構成されている (図 4.4)。各クラスについてまとめると以下のようになる。

World, Space, Clock

対象世界を表現する土台が「World」である。World は、その世界に固有の空間と時間によって規定されている。空間は「Space」によって、その世界の地理的な構造が表される。また、不可逆的な時間の流れを表すために「Clock」があり、この時間が経過することで現象が進行する。World には、後述の Entity、すなわち Agent と Goods が配置される。

Entity, Agent, Goods

世界に存在する実体が「Entity」である。Entity には、Agent と Goods の 2 種類があり、どちらにも後述する Information を付随させることができる。

社会・経済において、さまざまな活動を行う個人や社会集団 (企業・政府・家族・学校・地域社会・国) が「Agent」である。また、動的に変化する環境や「モデルの外部」なども Agent として表現することがある。Agent は Goods を所有し、その行動 (振る舞い) は、後述するように Behavior として定義する。

Agent に所有し交換される有形/無形のものが「Goods」である。BEFM における「Goods」とは、人間の欲求を充足する性質をもつという経済学における狭義の意味ではなく、世界におけるさまざまなものを示す広義の概念として用いている。例えば、

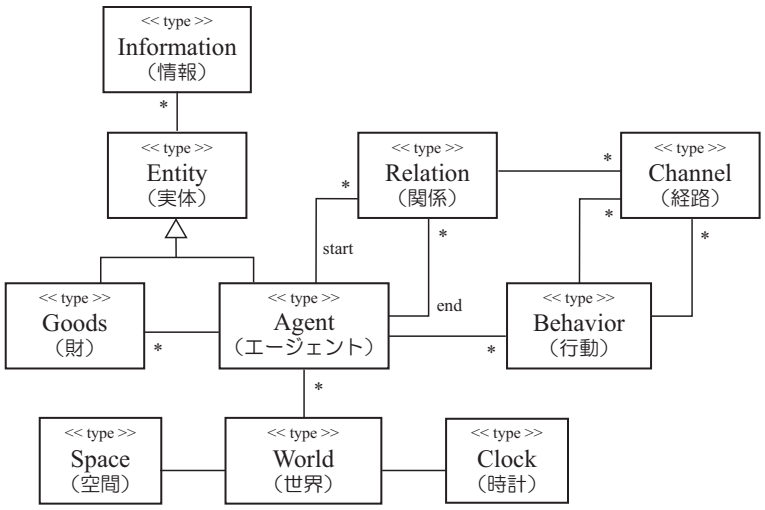


図 4.4: BEFM 概念モデル・フレームワークのクラス図

自動車、石油、トウモロコシ、株、土地の権利、書籍、広告、日記、回覧板、水、声、騒音、ごみ、貨幣などは、どれも Goods である。

Information

Entity が保持する情報は「Information」として表される。例えば、Agent が記憶した情報や、Goods に付随して取引される情報などが、Information である[(42)]。Information は単独では存在せず、必ず Entity、つまり Agent や Goods によって保持されている。Agent が保持する Information は、主体の内部に貯蔵されている「記憶」や「遺伝子」をはじめとして、Agent のさまざまな属性を表現する。Goods に Information が付随するというのは、例えば、新聞は「紙」(Goods) に「記事内容」(Information) が付随したものであり、会話は無形で瞬間的な「声」(Goods) に「会話内容」(Information) が付随したものと捉えるということである。Goods に付随している Information は、その Goods が他の Agent に渡されると、Goods とともに送られる。また、Agent によって保持されている Information は、すでに持っている Goods か、そのために作成した Goods に付随させて、他の Agent に送ることができる。

Behavior

エージェントの行動は、「Behavior」として表される。例えば、企業における生産行動や販売行動、個人における購買行動や労働行動などは、どれも Behavior である。後述するように、オブジェクト指向によってエージェントをモデル化する場合には、エージェントの行動を Agent クラスの操作として記述するのが一般的であるが、BEFM で

は行動をモデル要素の 1 つとして分離する。これは、状況によって振る舞いが動的に変化することを表現できるようにするためである。

BEFM では、Agent は複数の Behavior を並列的に実行することができる。内部状態を各 Behavior にもたせることで、複数の行動の多様な組み合わせを実現することができる。

Relation, Channel

ある Agent から他の Agent への関連性は、「Relation」によって表される。これにより、友人関係や家族関係、雇用関係などの関係を表現することができる。実際のコミュニケーションの際には、この Relation に基づいて開設されるコミュニケーション・パスである「Channel」を通じて、商品や会話、貨幣などの Goods をやりとりする。

Relation と Channel は、密接に関わっている概念であるが、これらは別のものである。Relation は「参加」という「構造的関係」(西部, 1997) を表しており、Channel は「活動」という「過程の関係」(西部, 1997) である。Relation は Agent 間の関連性を表すが、それに基づいて開設される Channel は、Behavior 同士を接続する。同一の Agent 内における Behavior 間のやりとり[43]であっても、Channel 上の Goods のやりとりで表現されるので、Behavior どうしのやりとりはすべて、Channel を通じた Goods の取引として抽象化されることになる。

4.2.2 BEFM シミュレーションモデル・フレームワーク

BEFM シミュレーションモデル・フレームワークは、BEFM 概念モデル・フレームワークと一貫性をもつように設計されている。また、このフレームワークに基づいて作成されたモデルは、Boxed Economy Simulation Platform 上でシミュレートすることができる。

BEFM シミュレーションモデル・フレームワークでは、概念モデルのモデル要素のそれぞれに対し、種類の違いをどのように表現するのかの変換方法を規定している[44]。種類の表現方法としては、「継承」による方法と「パワータイプ」による方法の 2 種類がある。

継承による方法は、クラスを特化したサブクラスを定義することで、種類の違いをクラスレベルで実現するものである。この方法を用いるものには、Behavior と Information がある。この 2 つのモデル要素は、種類の違いが、単なる属性値の差異ではなく、振る舞いや内部構造の差異であるためである。

パワータイプによる種類の表現は、種類クラスを作って、その種類クラスのインスタンスレベルで多様性を実現するというものである (Martin and J.Odell, 1995; Fowler, 1996)。この方法によって種類を表現するものは、Agent と Goods、そして Relation

である (表 4.1)。この場合、新たにクラスを作成することなく、インスタンスレベルで差異を表現することができる(45)。BEFM シミュレーションモデル・フレームワークでは、後述する「Type」クラスをパワータイプとして用いることができる。

以下では、BEFM シミュレーション・プラットフォームの概要を説明する (詳細については、付録 B を参照してほしい)。

Agent

BEFM シミュレーションモデル・フレームワークでは、エージェントの行動を、Behavior オブジェクトとして外部化し、「オブジェクトコンポジション」によって Agent オブジェクトに付加するという方法を採用している。オブジェクトコンポジションとは、役割を外部化するためのオブジェクトを用意して振る舞いを委譲し、そのオブジェクトを実行時に関連づけるという設計のことである (Gamma et al., 1995; Coad and Mayfield, 1999)。同様に、エージェントの記憶 (情報) や、他のエージェントへの関係も、Information や Relation のオブジェクトコンポジションとして保持する設計になっている。

このような設計により、シミュレーションモデルにおけるコーディング作業は、Behavior や Information の記述が中心となり、それらの組み合わせ方によって、エージェントが設定できるようになる。この場合、Agent 自体はそれらの行動を束ねる役割を果たしているにすぎない。例えば、Agent をインスタンス化すると、単なる Agent オブジェクトが得られるが、そこに PurchaseBehavior を加えると、購買行動を行う「消費者エージェント」となる。このようなエージェントの設計は、新しい行動の追加や削除、そして行動の組み換えなどを簡単に行えるという柔軟性がある。また、既に作成されている Behavior を利用することも可能になるため、再利用性を考慮した設計といえる。

Agent のもつ主な機能は、TimeEvent の受信、OpenChannelEvent の受信、Behavior の追加・削除・取得、Goods の追加・削除・調査、Relation の追加・削除・取得である。

表 4.1: 各モデル要素の作成方法

基礎モデル要素	作成方法
Agent	AgentType を追加
Goods	GoodsType を追加
Relation	RelationType を追加
Information	Information インターフェースを実装
Behavior	Behavior クラスを継承（コンポーネントビルダーによって自動化）
World	World クラスを継承
Clock	AbstractClock を継承（StepClock、RealClock を利用可能）
Space	Space インターフェースを実装（CellSpace を利用可能）

Agent は、時刻の経過を示す TimeEvent を受信し、もっている Behavior に配信する。また、経路の開設を求める OpenChannelEvent を受信した場合には、経路開設を試みる。成功すれば、適切な Behavior に対して開設された Channel を返す。Agent は、BehaviorType を指定することで Behavior の追加や取得、削除をすることができる。追加された Behavior は自動的に開始状態となる。また、Goods の追加や、GoodsType をもとにした Goods の取り出しや Goods の量を調査が可能である。そして、Relation の追加や Type による検索・削除を行うことができる。

Behavior

BEFM シミュレーションモデル・フレームワークでは、エージェントの持つそれぞれの Behavior を「状態機械」として定義する。状態機械とは、何らかのトリガーとなるイベント (影響を及ぼすさまざまな出来事) を受け取ると、現在の状態に応じた「アクション」(動作) を行い、新しい状態へ遷移するシステムである。ある時点をみてみると、Behavior オブジェクトは、必ずどれか 1 つの状態にとどまっている。Behavior の状態遷移を引き起こすイベントには、時間が経過したことを表す「TimeEvent」と、Goods が送られてきたことを表す「ChannelEvent」がある。

このような設計により、外界のイベントの種類と、現在の自分の状態によって、振る舞いが異なるというモデルを実現できる。本来、システムの内部状態とは、システムのすべてのパラメータの値の集合のことであるが、状態機械では、それらの状態のうちの一部を意識的に切り出して注目することになる。このことは、本論文で目指している複雑系 (状態の変化によって振る舞いのルールが変化するシステム) のモデルを記述する際に、モデルの状態の複雑さを隠蔽し、注目すべき状態を強調することができるのである。

エージェントが複数の行動を並列的に動作させる場合には、その内部状態は複雑にならざるを得ないが、BEFM に基づくモデル化では、Behavior という分かりやすい単位ごとに状態を把握することができる。また、Behavior の多くの詳細な部分はユーザーには見えない形で隠蔽されているので、状態機械としての Behavior の動作部分の実装については、ユーザーはほとんど意識する必要はない[46]。

Event

Behavior が受け取る Event には、代表的なものとして「TimeEvent」と「ChannelEvent」がある。TimeEvent は、時間の経過を知らせる Event である。モデル外部から時刻が経過したことを知らせるために Agent に送られて、Agent が Behavior に転送する。ChannelEvent は、ほかの Behavior から Goods が送られたことを知らせる Event である。

このほか、通常用いないが用意されている Event もある。「OpenChannelEvent」は、Relation から Agent を通して受け取る Event である。明示的に書かなくても受け取って Channel を開設するが、あえて明示的にこの Event を受け取るように記述することもできる。「(AutoTransition)」は、状態機械において遷移は外部からの刺激のみによるという定義があるため推奨はしないが、Event を受け取らなくても自動的に状態が遷移するように記述することも可能である。これは条件によって分岐させる場合などに用いることができる。

Type

BEFM シミュレーションモデル・フレームワークでは、モデルに存在するオブジェクトの要素を分類するための識別子として、「Type」クラスが用意されている。Type には、AgentType、GoodsType、InformationType、RelationType、BehaviorType の 5 種類がある (図 4.5)。Type の導入により、柔軟な検索・取得・識別が可能となる。例えば、同じ振る舞いをする Agent であっても、異なる Type が付加してあれば、別々に識別可能となる。

Type は、複数の Type 間に親子関係を定義することができるので、上位概念・下位概念を表現することができる。これによって、異なる Type をもつものであっても、同じ親 Type をもつものどうしであれば、一括して扱うことができるのである。例えば、図 4.6 左は、「ビデオ」には「VHS 規格のビデオ」と「Beta 規格のビデオ」があるということを表している。このような概念間の関係性を定義することによって、エージェントにその Goods の種類についての複雑な知識をもたせることができるようになる。例えば、市場におけるビデオの普及率を知りたい場合には、GoodsType「ビデオ」を親タイプとしてもつ Goods の数を調べればよいし、各規格の市場シェアを知りたい場合には、GoodsType「VHS 規格のビデオ」と GoodsType「Beta 規格のビデオ」の Goods の数をそれぞれ調べればよい。また、図 4.6 右は、「ビデオ一体型テレビ」が「テレビ」でもあり「ビデオ」でもある、ということを表している。あるエージェントが GoodsType「ビデオ一体型テレビ」の Goods をもっている場合、そのエージェントは、「ビデオ一体型テレビを所有しているか」という質問にはもちろんのこと、「テレビを所有しているか」や「ビデオを所有しているか」という質問にも、正しく答えることができるようになるのである。

World

World は、まず、モデルにおける時間・空間を定義するためにそれぞれただ 1 つの Clock、Space を持つ。World は Agent を配置してそれらを管理し、それらの追加、削除、生成、取得を行うことができる。また、RandomNumberGenerator を同様に

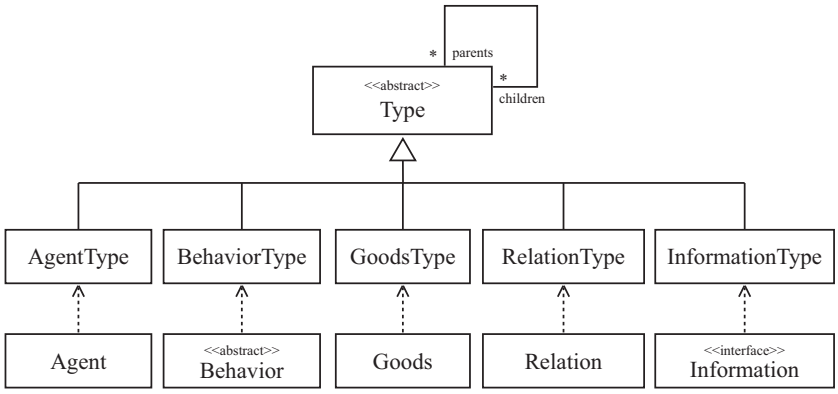


図 4.5: Type とその関連クラス

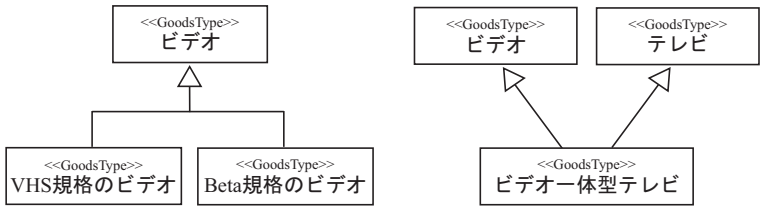


図 4.6: Type の継承の例

複数もつことができる。RandomNumberGenerator は、モデルで乱数を利用するときに用いるクラスである。コンピュータは、その構造上完全にランダムな数字を生成することはできないが、複雑なアルゴリズムによって事実上乱数と捉えて差し支えない数字 (擬似乱数) を生成することができる。乱数生成にはいろいろな方法があるため、モデルやモデル作成者によって利用したい方法も異なると考えられる。このことを考慮し、RandomNumberGenerator は、乱数として数字を生成するアルゴリズムを内包して、さらに値を取得するためのメソッドを持つクラスと定義されている。RandomNumberGenerator は、World 上に同時に複数存在できる。その場合、RandomNumberGenerator は名前によって識別され、Behavior などから名前で取得されて、利用されることになる。

モデル作成の際には、World クラスを継承したクラスで、initializeWorld() および initializeAgents() をオーバーライドし、Agent の配置や Relation の構築、Behavior の追加を行う。

Goods

Goods は、種類 (GoodsType)、量 (GoodsQuantity)、付随情報 (Information) からなるオブジェクトとして定義する。Goods の Quantity を直接変更することはできず、必ず Agent のメソッドを用いて分割や結合を行う。

Entity

Entity は Agent、Goods の親クラスである。これらの 2 つのクラスの共通の性質である Information を管理するインターフェースを持っている。

Information の管理はハッシュテーブルによって行われる。ハッシュテーブルにおけるキーも Information である。これにより、Information をキーとして Information を格納し、取り出すことができる。また、取り扱いを簡単にするために、自動的に取り扱いたい Information の InformationType をキーとして操作するメソッドが追加されている。

Information

Information の実装は Information インターフェースを実装する必要がある。Information インターフェースは空インターフェースである。メソッドは何も定義されていないので、それ以外は完全に作成者に依存している。

Relation と Channel

Channel の作成（経路の開設）は Relation の openChannel() メソッドを呼び出すことによって行われる。openChannel() は BehaviorType を引数にとなる。経路開設では、Relation の先の Agent の Behavior の中から BehaviorType に該当する Behavior を探し、その Behavior と経路を開設する。開設した経路 (Channel クラス) は openChannel() の返り値として返される (ただし、この部分の多くは Behavior の持つメソッドで隠蔽されているので Relation の openChannel（）メソッドを呼び出す必要はない)。

また、Channel の持つパラメータとして keep パラメータがある。これは財送信後その Channel が継続して存在するかを設定するものである。false の場合には、財送信・受信処理が完了すると Channel は自動的に close() が呼ばれる。true の場合には、財送信・受信処理後も Channel オブジェクトは存続し、Behavior から取得することができる。

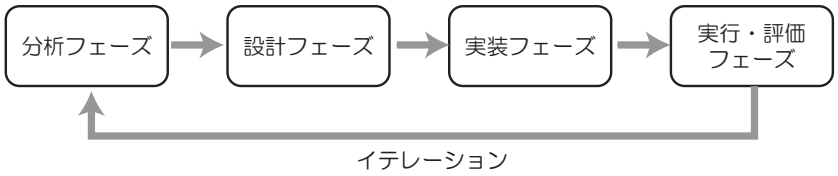


図 4.7: BEFM を用いたモデル作成プロセス

4.3 提案モデル・フレームワークを用いたモデル作成のプロセス

BEFM を用いたモデル作成プロセスは、オブジェクト指向開発プロセスに基づいて、「分析」、「設計」、「実装」「実行・評価」のスパイラルモデルとなる (図 4.7)。このモデル作成プロセスは、次のような流れで進められる (Boxed Economy Project, 2003; 松澤ほか, 2003)。

4.3.1 分析フェーズ

モデル作成プロセスの第一段階である「分析フェーズ」は、どのような対象領域のシミュレーションを行うのかを明らかにし、それを概念モデルとして記述するフェーズである。モデル化しようとしている対象が、「どのようなものであるか」(What) を明確化するために、BEFM 概念モデル・フレームワークを利用して、対象領域に登場する Agent、Information、Goods、Behavior、Relation をすべて洗い出して定義することから始める。まず最初にエージェントとその行動を明らかにし、それらのエージェントの関係について、「概念モデルクラス図」を記述する。そして、エージェントの行動のフローチャートを「行動アクティビティ図」として記述する。この過程で、登場する財や情報も洗い出して概念モデルクラス図に追加・修正していく。また、これらの分析をもとに、各行動の間でどのような相互作用 (財や情報のやりとり) があるかを確認し、その一つのシナリオを時系列に表現する「取引シーケンス図」を記述する。以上のような記述によってモデル要素の洗い出しと定義が終わるまで、このプロセスを繰り返していく。

4.3.2 設計フェーズ

モデル作成プロセスの第二段階である「設計フェーズ」は、 分析フェーズで作成された概念モデルをもとに、シミュレーションモデルの設計を行うフェーズである。概念モデルの各モデル要素について、シミュレーションとして動作させるための詳細を決めていくのである。まず、パワータイプを用いて種類を表現する Agent、Goods、Relation について、それぞれ AgentType、GoodsType、RelationType を定義する。

Behaviorは、Behaviorクラスを継承して詳細を定義するので、行動アクティビティ図と取引シーケンス図をもとに、Behaviorの状態遷移図を記述する。Informationには、Informationクラスを継承して詳細を定義するものと、InformationTypeだけでよいものという2種類がある。まず、Informationクラスを継承して詳細を定義するのは、どのような情報がどのような形式で格納されるかを定義しなければならないInformationの場合である。他の行動に対して依頼や質問をする際に使用するものなど、内容を含まない情報は、InformationTypeを定義するだけでよい(47)。次章でみるように、この設計フェーズは、記述を支援するツールを用いることができる。

4.3.3 実装フェーズ

モデル作成プロセスの第三段階である「実装フェーズ」は、設計フェーズで作成されたシミュレーションモデルを、Java言語を用いて実装するフェーズである。次章でみるように、Typeの実装は、私たちの提供しているタイプエディタを用いて行うことができる。また、Behaviorの実装は、私たちの提供しているコンポーネントビルダーからソースコードの雛型を生成し、その雛型の一部を埋めるようにして実装する。WorldおよびInformationの実装は、ソースコードを記述することで行う。

4.3.4 実行・評価フェーズ

モデル作成プロセスの第四段階である「実行評価フェーズ」は、実装フェーズで作成されたシミュレーションを、実行して評価するフェーズである。シミュレーションの実行は、次章で提案するシミュレーション・プラットフォーム「Boxed Economy Simulation Platform」を用いることができる。この段階で、シミュレーションが意図した通りに動作するかという正当性の検証も行う。そして、シミュレーションの設定をさまざまに変化させながら、シミュレーションの結果と現実の現象を比較し、妥当性の検証を行う。このとき必要があれば、分析フェーズに戻るなどして、次のイテレーションに入ることになる。

4.4 先行研究との比較

オブジェクト指向によってエージェントを表現する場合、現状ではエージェントをひとつのオブジェクトとして設計することが多い。例えば、Bruun (2002) はエージェントベース経済シミュレーションのためのフレームワークとして、図4.8のようなエージェントの設計を提案している。またAxtell (2002) などでも、エージェントを表すオブジェクトの操作としてエージェントの行動が定義されている。しかし、実はこのような設計は、エージェントをひとつのオブジェクトにカプセル化してしまうので、行

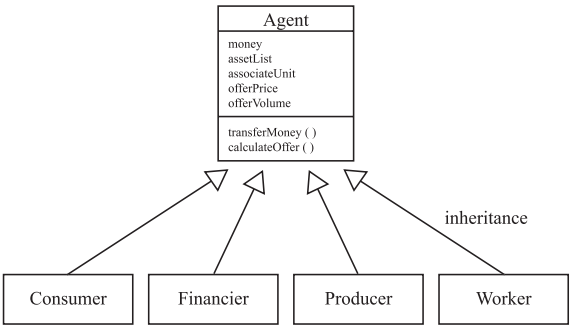


図 4.8: 一般的なエージェントの設計 (Bruun, 2002)

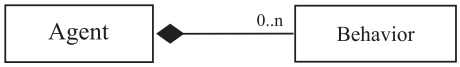


図 4.9: 提案モデル・フレームワークにおけるエージェントの設計

動の種類そのものが動的に変化するエージェントを作成する場合や、エージェントの一部を再利用する場合に限界が生じる。

これに対し、提案モデル・フレームワークでは、「継承」を用いる代わりに「コンポジション」を用いることで、柔軟性と再利用性を考慮した設計になっている。この設計では、エージェントクラスの中に操作として行動を記述するのではなく、エージェントの役割別に行動クラスを作り、それらのオブジェクトを「主体」の核となる部分が保持するという設計となる (図 4.9)。

このような行動のコンポジションによるエージェント表現は、モデルの意味的な側面における利点もある。それは第一に、ひとつのエージェントが複数の社会的役割を担っているということを自然な形で表現できるということである。例えば、「消費者エージェント」や「労働者エージェント」という実在しない主体をモデル化するのではなく、「個人」が時として消費者の役割を担ったり、労働者の役割を担ったりするという表現になるのである。このようなエージェントが複数の役割を担うというモデル化は、特に経済全体をシミュレートするような場合に不可欠となる。経済社会におけるエージェントの異質性というのは、結局のところ行動や役割が異なるということから生じるからである。存在するのはあくまで個人や社会集団であり、それらの行う行動が異なるがゆえに異なる種類のエージェントとして識別されるのである。

行動のコンポジションの第二の利点は、新しい行動の追加や削除、行動の手続きの変更などが可能になるということである。例えば、小売業が銀行機能の一部を担うようになるということを表すためには、小売業のエージェントが、銀行のもつ機能の一部を取り入れる必要がある。従来のようなエージェント単位の設計では、小売業エージェントや銀行エージェントというようにエージェントにその振る舞いをカプセル化

しているため、多重継承で拡張するか、銀行兼務小売業エージェントのようなものを新たに作成しなければならくなる。しかしこのような方法では、継承の階層が深く複雑化するため、長期的にみると限界がある。経済全体が分析対象の場合には、時間的にも長期となり、状況によってエージェントが成長して行動の種類を変更したりすることが考えられる。それゆえ、エージェント単位で定義するのではなく、それらの個別の行動ごとに分割して定義し、コンポジションによってエージェントを特徴づけていくという設計が不可欠となるのである。

第5章　シミュレーション・プラットフォームの提案

5.1　シミュレーション・プラットフォームとは

本章では、シミュレーションの作成・実行・分析を支援するシステムを提案する。ここでは、シミュレーションプログラムを、個々の機能単位ごとに分割された「コンポーネント」と、それをまとめるプラットフォームからなるシステムとして設計する(図 5.1)。コンポーネントとは、明確に定義された用途と境界を持つソフトウェアモジュールのことであり、他のコンポーネントと協調することでシステムの動作の一部を担うものである。

このようなコンポーネントとプラットフォームという構造を採用するのは、モデルや機能をコンポーネントに分割して独立させることで、容易に一部を再利用・拡張したりできるようにするためである。それは、シミュレーション研究の支援システムは様々なモデルや分析に対応できることが要求されるため、システムのモジュール性や拡張性、そして再利用性を高める必要があるからである。

コンポーネントベースのシミュレーション・プラットフォームには、研究プロセスを一貫して支援する統合環境の提供、モデル部品の再利用と並行開発を支援する仕組みの提供、シミュレーション環境の再利用と拡張を支援する仕組みの提供、という利点がある。

5.1.1　研究プロセスを一貫して支援する統合環境の提供

シミュレーション・プラットフォームは、シミュレーション研究のプロセスを一貫して支援するための統合環境を提供する。これにより、研究プロセスの各フェーズをシームレスに、また効率的に行うことが可能になる。また、一つの統合環境上で動かしているため、初期値を自動的に変化させて結果の振る舞いをチェックするような自動化機能などを実現することもできるようになる (Iba et al., 2000)。さらに、シミュレーションに関係する様々な要素の体系的管理が可能となる。

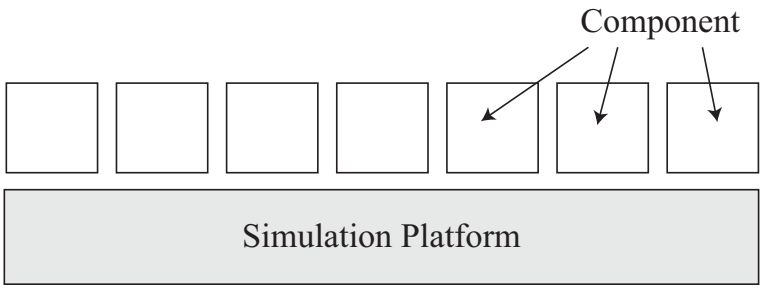


図 5.1: シミュレーション・プラットフォームの基本構造

5.1.2 モデル部品の再利用と並行開発を支援する仕組みの提供

シミュレーション・プラットフォームは、モデル部品の再利用を支援する仕組みを提供する。これによって、モデル作成時に既存のモデル部品を再利用することや、構成的理解のために部分モデルを交換するという模索的な作業を行うことが可能になる (図 5.2)。また、再利用する側だけでなく、モデル部品を作成する側にとっても、再利用性を意識せずに自然と再利用可能なプログラム部品を作成することができる[48]。

このことは、「シミュレーションの実装 (プログラミング)」と「シミュレーションの設定」のフェーズを明確に分離することが可能になるということでもある。コンポーネントの実装を行う「実装」フェーズでは、コンポーネントはソースコードを見ることができるホワイトボックスとして扱われる。これに対し、複数のコンポーネントの組み合わせと設定を行う「設定」フェーズでは、それぞれの機能が実装されているブラックボックスとして扱うことができる。このような実装と設定の分離は、コンポーネントを開発する人と、それを組み合わせてシミュレーションを設定・実行する人が別の人でもよい、ということ意味している (図 5.3)[49]。これにより、プログラミング技術をもたない社会科学者であっても、設定と実行を行うことができるようになる。

また、シミュレーション・プラットフォームは、複数のコンポーネント開発者がコンポーネントを並行して開発することを可能とする。例えば、あるコンポーネント開発者が企業間取引の行動を作成するときに、他のコンポーネントデベロッパーが消費者の独立に商品購買行動を作成しても、これらは同じフレームワークのもとで整合的に動くことが保証されるのである。

5.1.3 シミュレーション環境の再利用と拡張を支援する仕組みの提供

シミュレーション・プラットフォームは、シミュレーション環境の再利用と拡張を支援する仕組みも提供する。シミュレーション研究を中長期的に捉えた場合、時間が経つにつれて発展するのはモデルだけではなく、そのモデルを分析する手法も同様に発展すると考えられる。そのため、シミュレーション環境は、様々なモデルや分析に

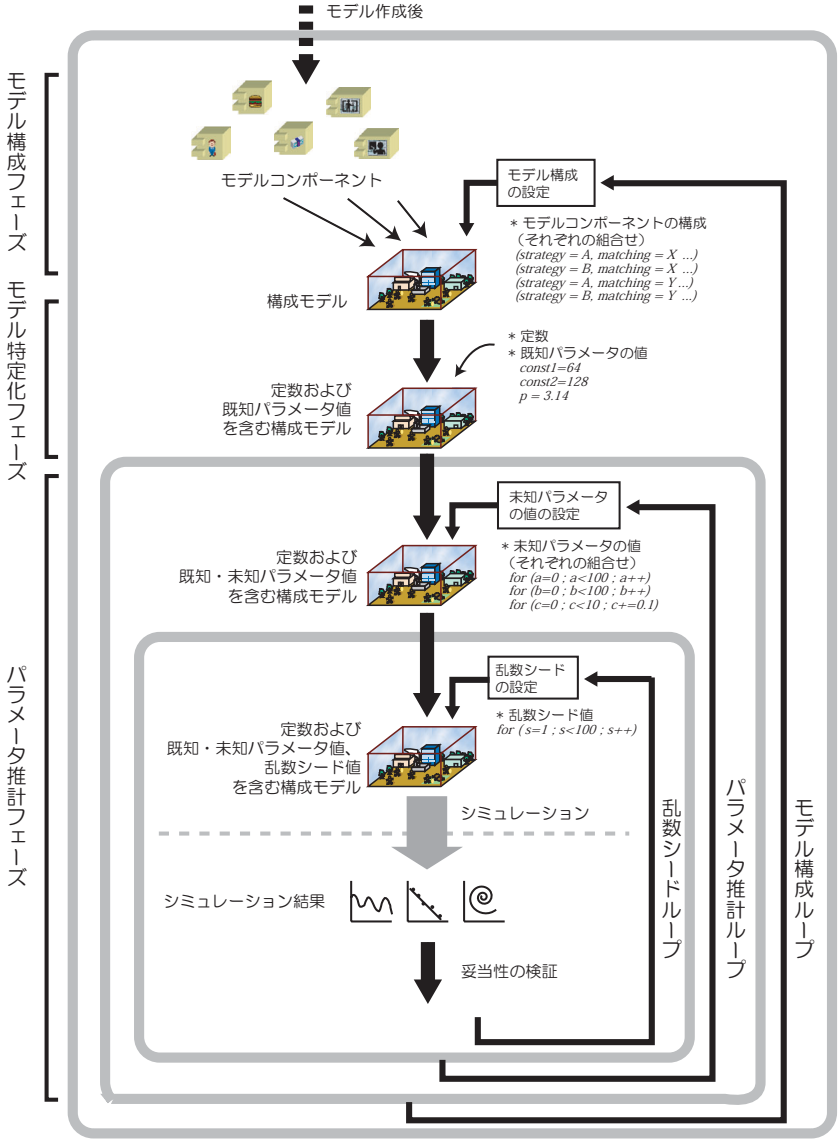


図 5.2: 探索的モデルビルディング (Iba et al., 2000)

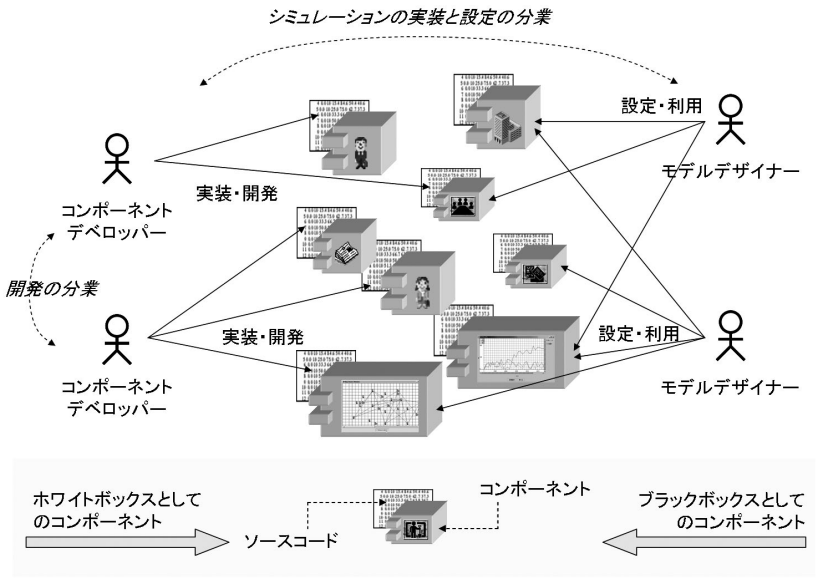


図 5.3: コンポーネントによる実装と設定の分離

対応できることが要求され、システムの拡張や表現に対する柔軟性がもとめられることになる。

コンポーネントベースのシミュレーション・プラットフォームであれば、利用者ごとに多種多様なシミュレーション環境を実現できるほか、独自に新たな機能を追加することができるようになる。

5.2 提案シミュレーション・プラットフォーム: Boxed Economy Simulation Platform (BESP)

ここでは、広義の複雑系と狭義の複雑系のシミュレーションを作成・実行するための Boxed Economy Simulation Platform を提案する。Boxed Economy Simulation Platform (以下、BESP) は、エージェントベースの社会・経済モデルのシミュレーションを作成・実行・分析するためのプラットフォームである (図 5.4)。BESP は、オブジェクト指向に基づいて Java 言語で実装されたマルチプラットフォームのソフトウェアである[(50)]。

5.2.1 基本アーキテクチャ

BESP の基本的なアーキテクチャは、ベースとなるコンテナと、それらにプラグインするコンポーネント群で構成される (図 5.5)。「モデルコンテナ」、「プレゼンテー

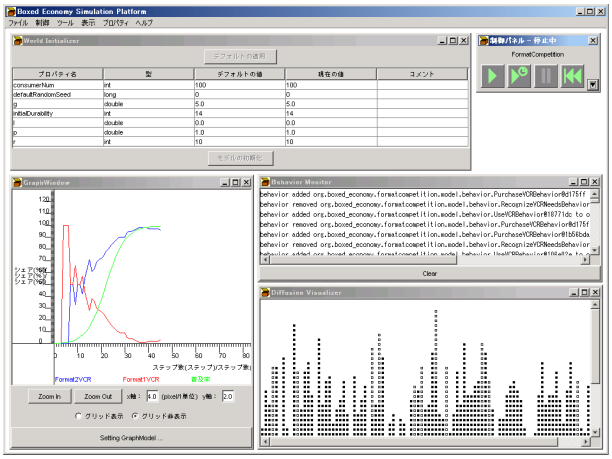


図 5.4: Boxed Economy Simulation Platform (BESP)

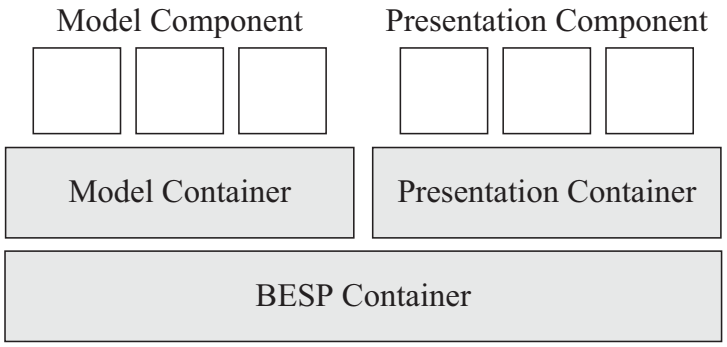


図 5.5: BESP の内部構造

ションコンテナ」、および「BESP コンテナ」を含んだ部分がシミュレーション実行の基盤となるプラットフォーム部分である。BESP のユーザは、このプラットフォームに、必要なコンポーネントをセットすることで自分のニーズに適合したシミュレーション環境を実現することができる。BESP にセットされるコンポーネントは大きくわけて「モデルコンポーネント」と「プレゼンテーションコンポーネント」の二種類がある。

以下では、モデルコンポーネント、プレゼンテーションコンポーネント、モデルコンテナ、プレゼンテーションコンテナ、BESP コンテナについて説明する。

モデルコンポーネント

モデルコンポーネントは、ユーザが分析したいシミュレーションモデルの各要素をコンポーネント化したものである。このモデルコンポーネントを BESP のモデルコ

ンテナに配置することで、シミュレーションを行うことができる。BEFM シミュレーションモデル・フレームワークが定義されているため、このようにモデルコンポーネントとして切り分けることが可能なのである。

プレゼンテーションコンポーネント

プレゼンテーションコンポーネントは、シミュレーションの操作や表示、および記録を行うためのソフトウェア部品である。このプレゼンテーションコンポーネントを BESP 上のプレゼンテーションコンテナに配置することで、その機能が利用できるようになる。プレゼンテーションコンポーネントには、コンピュータ画面にシミュレーションの状況を文字やグラフィックスで表示するためや、マウスやキーボードからシミュレーションの設定を変更するために用いられる「GUI コンポーネント」と、シミュレーション状況を記録としてファイルに書き出すために用いられる「レポートコンポーネント」の二種類がある。

モデルコンテナ

「モデルコンテナ」は、シミュレーションを実行するモデル (World クラス) を管理する。その第一の役割は、「モデルの管理」である。モデルコンテナは World オブジェクトを持ち、これをシミュレーションを行うモデルとして管理する。この World オブジェクトは、World クラスのインポートによって更新される。第二の役割は、「モデルスレッドによるシミュレーションの実行」である。シミュレーションの実行をするということはモデルの時間を進めることだが、この役割はモデルスレッドによって行なわれる。モデルスレッド (ModelThread) は独立に稼動するスレッドであり、一定時間ごとにモデルの Agent に TimeEvent を配信して時刻を進める。これにより、モデルの時間が経過してモデルが動くことになる。TimeEvent の Agent への配信順は、AgentType ごとに Priority を設定することで変えることができる(51)。

第三の役割は、「Type の管理」である。モデルで利用する Type の追加・削除・取得を行うことが出来る。モデルコンテナにこの機能を持たせることによってモデルとモデルで扱われる意味空間を分離させることができ、複数のモデルで意味空間を共有することができるようになる。

プレゼンテーションコンテナ

「プレゼンテーションコンテナ」は、プレゼンテーションコンポーネントを配置するためのコンテナである。プレゼンテーションコンポーネントからシミュレーション実行の制御などを行うための制御コマンド群、プレゼンテーションコンポーネントフ

図 5.6: Control Panel プレゼンテーションコンポーネント

レームワークを含んでいる。BESP におけるユーザーの入出力はプレゼンテーションコンテナに集約され、プレゼンテーションコンテナが管理することになる。このことによって、プレゼンテーションコンポーネント作成者はモデルの操作について実装をする必要がない。また、プレゼンテーションコンテナは、メインウィンドウを持っているが、このメインウィンドウに追加するプラグイン (プレゼンテーションコンポーネントとしてのウィンドウも含む) を管理する機能ももっている。

BESP コンテナ

「BESP コンテナ」は、モデルコンテナとプレゼンテーションコンテナを持ち、BESP で最初に起動されるコンテナ部分である。BESP の初期化・終了シーケンスを行う。この中にはプラグインの読み込みが含まれており、BESP のパッケージには含まれていないコンポーネントの読み込みを行ったりする。この機能により、ユーザは追加したいコンポーネントのクラスファイルや JAR ファイルをクラスパスに置くだけで、簡単にプラグインが追加することができる。

5.2.2 提供されるプレゼンテーションコンポーネント

プレゼンテーションコンポーネントの例として、現在提供している汎用のプレゼンテーションコンポーネントの概要を説明する。

Control Panel

Control Panel は、シミュレーションの実行や停止の操作を行うためのプレゼンテーションコンポーネントである (図 5.6)。制御パネルには、「実行」、「一定時間実行」、「停止」、「リセット」のボタンがあり、BESP に現在読み込まれているモデルの名前が表示されるようになっている。「実行設定」ボタンを押すことによって、指定した時間だけ実行するという「一定時間実行」の期間指定は、できる (図 5.7)。1 ステップごとにどのくらいの時間を進めるのかといった設定も可能である。

図 5.7: Control Panel プレゼンテーションコンポーネント (一定時間実行設定)

図 5.8: WorldInitializer プレゼンテーションコンポーネント

World Initializer

World Initializaer は、モデルの初期設定を行うためのプレゼンテーションコンポーネントである (図 5.8)。World クラスで指定の準備を行っている変数について、その初期値を変更することができる。

Data Collector

Data Collector は、シミュレーションの実行結果を選択的に記録するためのプレゼンテーションコンポーネントである (図 5.9)。分析のためにはすべてのデータを採取しておくことが望ましいが、シミュレーションの進行に伴い膨大な量のデータが生成されるため、実際にはコンピュータのメモリや実行速度の制約を考慮して、必要な情報を選択的に記録するという方法が現実的である。Data Collector では、指定したデータセットについて、シミュレーション実行時にデータを格納する場所を作成する。

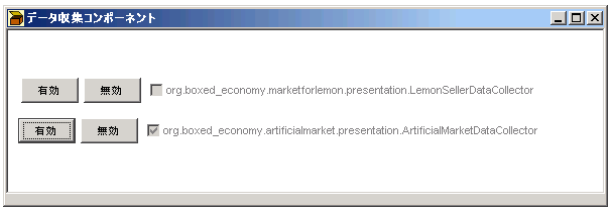


図 5.9: DataCollector プレゼンテーションコンポーネント

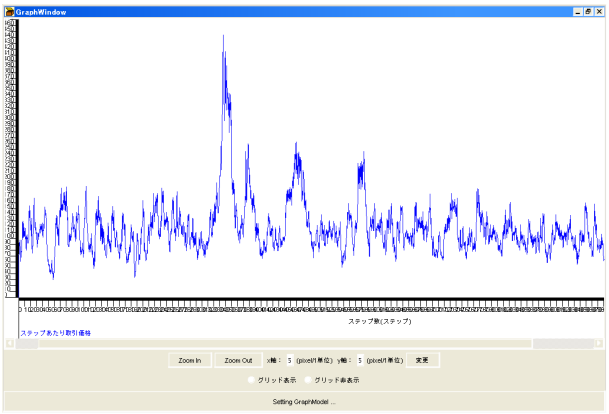


図 5.10: Graph プレゼンテーションコンポーネント

Data Collector

Graph は、Data Collector で登録したデータをグラフとして表示し、シミュレーションの状況をリアルタイムに反映するプレゼンテーションコンポーネントである (図 5.10)。

Relation Viewer

Relation Viewer は、エージェント間の関係を表示するためのプレゼンテーションコンポーネントである (図 5.11)。どの AgentType のエージェントを表示するのか、そしてどの RelationType の関係を表示するのかを設定することができる。ネットワーク構造の表示は、エージェントが円形に配置される「Circle」ビュー、AgentType ごとに階層的に配置される「Layer」ビュー、ランダムに配置される「Random」ビューがある。

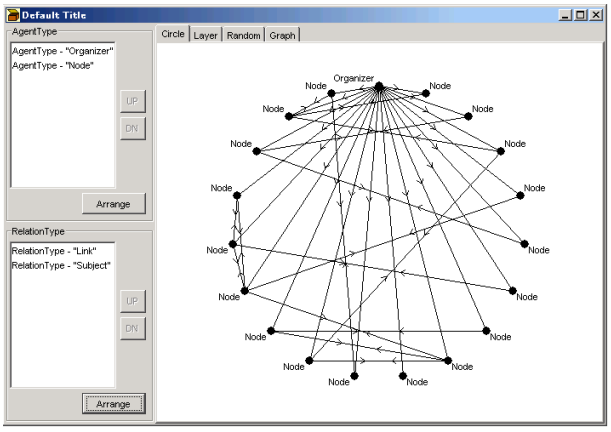


図 5.11: RelationViewer プレゼンテーションコンポーネント

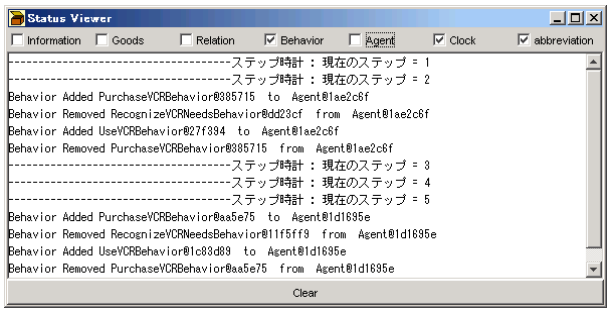


図 5.12: StatusViewer プレゼンテーションコンポーネント

Status Viewer

Status Viewer は、モデル要素の生成や消滅の状況を表示するためのプレゼンテーションコンポーネントである (図 5.12)。どのモデル要素についての状況を表示するのかを設定することができる。

Board

Board は、板寄約定方式の人工市場における「板」を表示するためのプレゼンテーションコンポーネントである (図 5.13)。それぞれの価格に対する売り注文量と買い注文量が表示される。

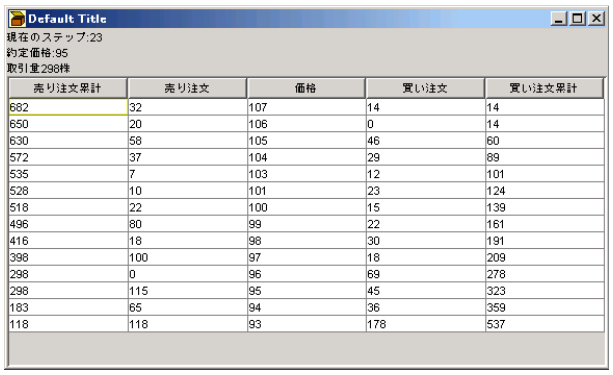


図 5.13: Board プレゼンテーションコンポーネント

5.3 提案シミュレーション・プラットフォームにおける設計と実装の支援

5.3.1 プログラミングの軽減の仕組み

BESP では、シミュレーションを作成するためのプログラミングを大幅に軽減させる仕組み・ツールを提供している。これを使用することにより、ユーザはシミュレーションを迅速に作成・変更できるようになるため、シミュレーションの分析などに研究の重点を置くことが可能となる。また、このプログラミングの軽減によって、社会科学者から見た参入障壁の多くを取り除くことができる。プログラミングにおいて難関となりやすい構造に関する設計や実装をしなくて済むので、基礎的なプログラミングの知識さえあればシミュレーションを作成することができるようになるからである。

プログラミングの軽減は具体的には三つの方法によってなされる。第一に、エージェントベース経済モデルをシミュレーションとして実行するために必要なプログラムの多くが、すでに BESP 本体に実装されているということがあげられる。第二に、シミュレーション・プログラムの作成を支援するツール (コード・ジェネレータ等) が、プレゼンテーションコンポーネントとして提供されるということがあげられる。第三に、モデルコンポーネントやプレゼンテーションコンポーネントを再利用できる点があげられる。ユーザは、自分の作成したいモデルの一部がすでにモデルコンポーネントとして作成されているならば、それを再利用することで、プログラミングの量を減らすことができる。また、将来的にモデルコンポーネントの蓄積が充実すれば、コンポーネントを組み合わせて設定するだけでシミュレーションを作成するというコンポーネントベースの開発が可能になる。

また、いま述べてきたプログラミングの軽減と密接な関係があるのだが、BESP は、ユーザへの負担を大きくせずにシミュレーションのソフトウェア品質を向上させるための仕組みを提供していることにもなる。それは、第一に、シミュレーションを実行

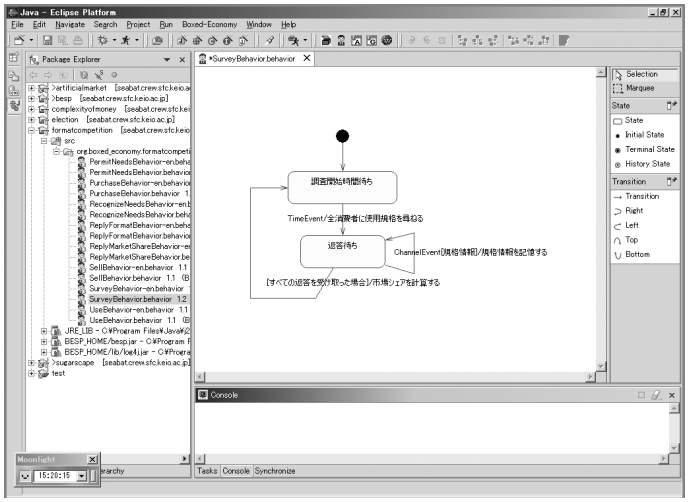


図 5.14: Component Builder

するためのプログラムの多くの部分が、すでに BESP において実現されているため、各ユーザは自分で新たに作成した部分のみをチェックすればよいということになる。第二に、作成支援ツールを用いることで、プログラミング時における人間のミスを減らすことができるため、念入りなチェックを行わなくて済むようになる。第三に、すでに作成されテスト済みのモデルコンポーネントを再利用するならば、その部分に関しても正当性の検証が軽減されるのである。このように、BESP の利用は、モデルが正しくプログラムに実装されているかという正当性の検証を行うべき範囲を小規模に抑える。

5.3.2 支援ツール: Component Builder

BESP では、BEFM に基づいたモデルコンポーネントを作るためのツールとして、Component Builder を提供している (図 5.14)[52]。Component Builder を用いることで、Agent が持つ Behavior の状態遷移図を記述すると、それに対応するソースコードを生成することができるほか、World クラスの雛型や、Type を定義する Model クラスを生成することができる。Component Builder を用いてモデルを作成する流れを示したものが図 5.15 である。

Behavior の作成

Behavior は、Component Builder を用いて状態遷移図を記述すると、それに対応するソースコードを生成することができる (図 5.16)。例えば、図 5.17 のような Behavior を、Component Builder を用いて作成するとしよう。この行動は、第 8 章の規格競争モデル

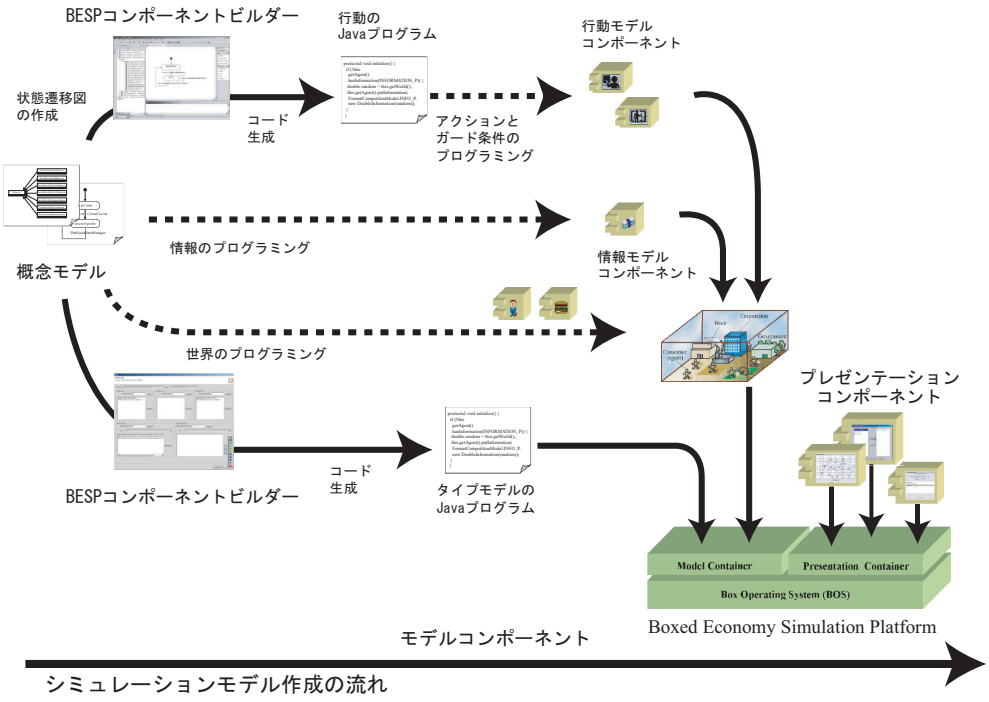


図 5.15: BESP の支援ツールを用いたシミュレーションの作成の流れ

の SellVCRBehavior である。Component Builder でこの状態遷移図を記述し、コード生成を行うと、「AbstractSellVCRBehavior.java」と「SellVCRBehavior.java」というソースコードが生成される。この AbstractSellVCRBehavior.javaは、状態やその遷移の枠組みを定義している部分であり、モデル作成者が触れる必要のない隠蔽されている部分である (図 5.18 および図 5.19)。モデル作成者は、もう一方の SellVCRBehavior.java の一部 (図 5.20 の網掛け部分) に、具体的なアクションやガード条件の内容を書くだけで、エージェントの Behavior を作成することができる。

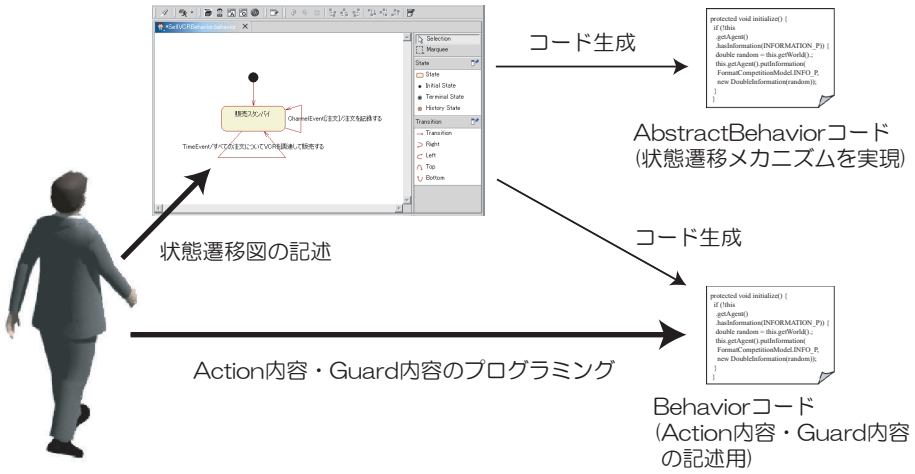


図 5.16: Component Builder を用いた行動の作成

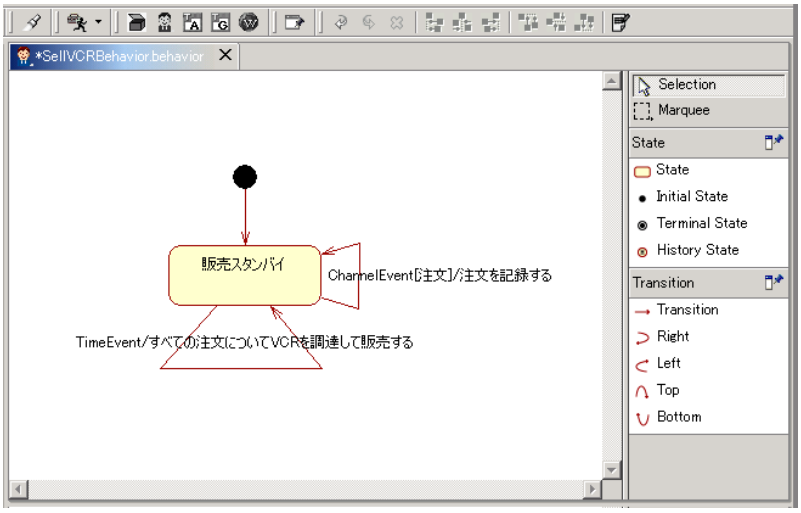


図 5.17: Component Builder 上で作成した状態遷移図

```
/*
 * AbstractSellVCRBehavior.java
 */
package org.boxed_economy.formatcompetition.model.behavior;

import org.boxed_economy.besp.model.fmfw.behavior.AbstractBehavior;

import org.boxed_economy.besp.model.fmfw.ChannelEvent;
import org.boxed_economy.besp.model.fmfw.TimeEvent;
import org.boxed_economy.besp.model.fmfw.behavior.Action;
import org.boxed_economy.besp.model.fmfw.behavior.CompositeState;
import org.boxed_economy.besp.model.fmfw.behavior.Event;
import org.boxed_economy.besp.model.fmfw.behavior.GuardCondition;
import org.boxed_economy.besp.model.fmfw.behavior.State;
import org.boxed_economy.besp.model.fmfw.behavior.StateMachineFactory;
import org.boxed_economy.besp.model.fmfw.behavior.Transition;

/**
* AbstractSellVCRBehavior
*/
public abstract class AbstractSellVCRBehavior extends AbstractBehavior {

 /**
 * This method automatically generated from AbstractBehavior Builder
 * Don't touch by hand
 */
 protected void initializeStateMachine() {
     //factory
     StateMachineFactory factory = this.getStateMachine();

      //states
      State initialState = factory.createInitialState();
      CompositeState 販売スタンバイ = factory.createCompositeState("販売スタンバイ");

      //actions
      Action すべての注文についてVCRを調達して販売する = new Action() {
           public void doAction() {
               すべての注文についてVCRを調達して販売する();
           }
           public String toString() {
                return "すべての注文についてVCRを調達して販売する";
           }
      };

      Action 注文を記録する = new Action() {
           public void doAction() {
               注文を記録する();
           }
           public String toString() {
                return "注文を記録する";
           }
      };

       //guard-conditions
       GuardCondition 注文 = new GuardCondition() {
            public boolean isMatched(Event e) {
                return 注文(e);
            }
       };

       //transitions
       Transition transition販売スタンバイTo販売スタンバイ = factory.createTransition();
       Transition transition販売スタンバイTo販売スタンバイ1 = factory.createTransition();
       Transition transitionInitialStateTo販売スタンバイ = factory.createTransition();
```

図 5.18: Component Builder によって自動生成された AbstractBehavior のコード (1)

```
    //states setting

    //structure of states
    this.setInitialState(initialState);
    this.addState(販売スタンバイ);

    //transitions setting
    transition販売スタンバイTo販売スタンバイ.setEvent(TimeEvent.class);
    transition販売スタンバイTo販売スタンバイ.addAction(すべての注文についてVCRを調達して販売する);
    transition販売スタンバイTo販売スタンバイ1.setEvent(ChannelEvent.class);
    transition販売スタンバイTo販売スタンバイ1.setGuardCondition(注文);
    transition販売スタンバイTo販売スタンバイ1.addAction(注文を記録する);

    //connection of transitions
    transition販売スタンバイTo販売スタンバイ.setSourceState(販売スタンバイ);
    transition販売スタンバイTo販売スタンバイ.setTargetState(販売スタンバイ);
    transition販売スタンバイTo販売スタンバイ1.setSourceState(販売スタンバイ);
    transition販売スタンバイTo販売スタンバイ1.setTargetState(販売スタンバイ);
    transitionInitialStateTo販売スタンバイ.setSourceState(initialState);
    transitionInitialStateTo販売スタンバイ.setTargetState(販売スタンバイ);
}

/**
 *すべての注文についてVCRを調達して販売する
 */
protected abstract void すべての注文についてVCRを調達して販売する();

/**
 *注文を記録する
 */
 protected abstract void 注文を記録する();

 /**
  *注文
  */
 protected abstract boolean 注文(Event e);
}
```

図 5.19: Component Builder によって自動生成された AbstractBehavior のコード (2)

```
/*
 * SellVCRBehavior.java
 */
package org.boxed_economy.formatcompetition.model.behavior;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

import org.boxed_economy.besp.model.fmfw.Channel;
import org.boxed_economy.besp.model.fmfw.Goods;
import org.boxed_economy.besp.model.fmfw.behavior.Event;
import org.boxed_economy.besp.model.fmfw.informations.IntegerInformation;
import org.boxed_economy.formatcompetition.model.FormatCompetitionModel;
import org.boxed_economy.formatcompetition.model.information.OrderInformation;


public class SellVCRBehavior extends AbstractSellVCRBehavior {

    private Map orders = new HashMap();

    /**
     * @see org.boxed_economy.besp.model.fmfw.behavior.AbstractBehavior#initialize()
     */
    protected void initialize() {
    }

    /**
     * @see org.boxed_economy.besp.model.fmfw.behavior.AbstractBehavior#terminate()
     */
    protected void terminate() {
    }

    /**
     * @see org.boxed_economy.formatcompetition.model.behavior.AbstractSellVCRBehavior#注文を記録する()
     */
    protected void 注文を記録する() {
        this.orders.put(this.getActiveChannel(), this.getReceivedInformation());
    }

    /**
     * @see org.boxed_economy.formatcompetition.model.behavior.AbstractSellVCRBehavior
     * #すべての注文についてVCRを調達して販売する()
     */
    protected void すべての注文についてVCRを調達して販売する() {
        Iterator i = this.orders.keySet().iterator();
        while (i.hasNext()) {
            Channel channel = (Channel) i.next();
            OrderInformation order = (OrderInformation) this.orders.get(channel);
            Goods vcr = this.getWorld().createGoods(order.getFormat(), 1.0);
            channel.sendGoods(vcr, this);
        }
        this.orders = new HashMap();
    }

    /**
     * @see org.boxed_economy.formatcompetition.model.behavior.AbstractSellVCRBehavior#注文(Event)
     */
    protected boolean 注文(Event e) {
        return this.getWorld().getInformationType(this.getReceivedInformation())
          == FormatCompetitionModel.INFORMATION_Order;
    }

}
```

図 5.20: Component Builder によって自動生成された Behavior のコード、およびそこに追加したコード (網掛け部分)

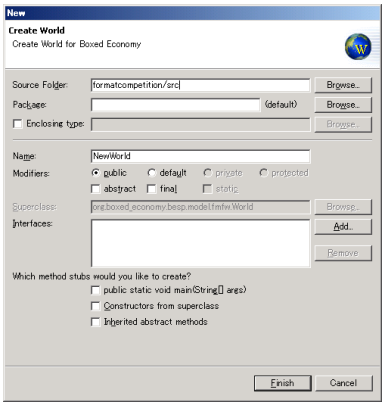


図 5.21: BESP Component Builder の World 生成ウィンドウ

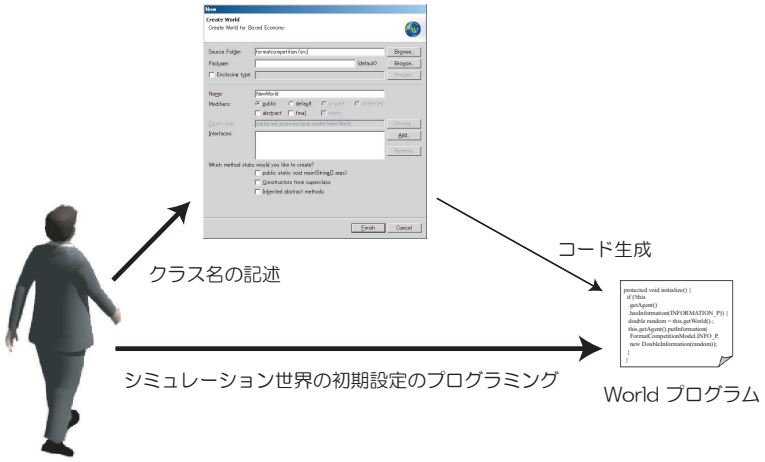


図 5.22: Component Builder を用いた世界の作成

World の作成

World は、Component Builder を用いて、その雛型をつくることができる (図5.21)。この雛型に、Agent の生成や、Agent への Behavior や Relation の追加などのプログラムを書き込むことで、シミュレーションの初期設定を作成する (図 5.22)。

Model の作成

モデル要素の種類の概念を表す Type は、Component Builder を用いることで、そのを定義を行う Model クラスのソースコードを生成することができる。AgentType、GoodsType、RelationType、BehaviorType、InformationType の入力部分に定義したい名前を入力することで、それぞれの設定が登録される (図 5.23)。Behavior と In-

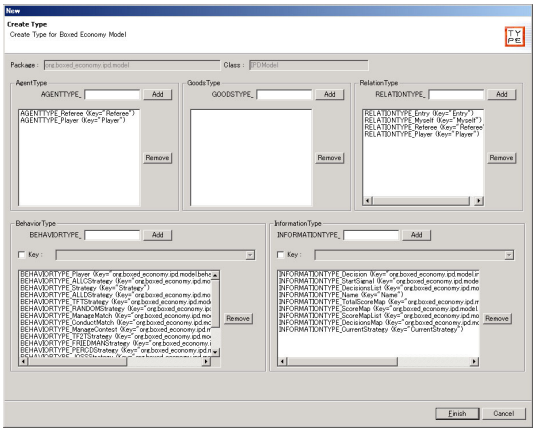


図 5.23: BESP Component Builder の Model 設定生成ウィンドウ

図 5.24: Component Builder を用いたモデル設定の作成

formation については、対応する Behavior クラスと Information クラスを対応づけることもできる。最後にこのエディタを終了すれば、Model クラスのソースコードを自動生成することができる (図 5.24)。

5.4 先行研究との比較

これまでにも、エージェントベースシミュレーションを支援するための言語やツールがいくつか開発されている (Dugdale, 2000)。その中で最も有名であり利用されていると思われる「Swarm Simulation System」は、複雑適応系のシミュレーションのためのクラスライブラリ (Objective-C 言語と Java 言語) を提供している (Langton et al., 1998)。また、Swarm と同様のコンセプトの「RePast」(REcursive Porous Agent

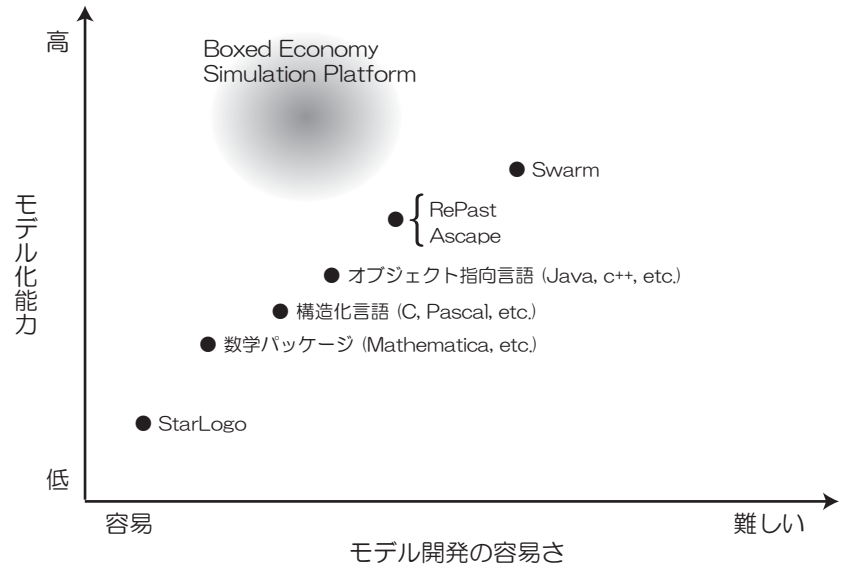


図 5.25: 既存シミュレーションシステムとの比較 (North(2002) を元に改変)

Simulation Toolkit) は、マルチエージェントモデルを作成するためのクラスライブラリ (Java 言語) を提供している (RePast,)。「Ascape」は、フレームワーク (Java 言語) を提供しており、Swarm や RePast に比べてコード記述量が少なくて済むといわれている (Parker, 2000; Parker, 2001)。

このほか、汎用のプログラミング言語を用いるのではなく、独自の簡易言語を設定している支援システムもある。シミュレーション原理の教育に有効であるといわれる「StarLogo」では、LOGO 言語を拡張した簡易言語を定義している。また、Swarm と StarLogo の中間レベルを実現しようとしている「MAS」(Multi-Agent Simulator) では、Visual Basic 言語に似た言語文法を独自に定義している (服部ほか, 2000; 玉田, 2001)。

このように、これまで提案されてきた支援システムは、プログラミング経験が少ない (もしくは無い) 作成者への支援が必要であるという問題意識を共有しており、その解決策として汎用的なライブラリなどの提供による補助を行っている。しかし、このようなソースコードレベルでの補助は、モデル作成プロセスにおける実装フェーズを支援するが、分析フェーズや設計フェーズを支援することはない。本論文の提案シミュレーション・プラットフォームは、提案モデル・フレームワーク (BEFM) とともに提案されているため、分析や設計の段階から、一貫した支援が可能である。

North (2002) のシミュレーター比較図を参考に、提案シミュレーション・プラットフォームを位置づけると、図 5.25 のようになると思われる。この図は、あくまで概略的なものであり、数値的な評価によってプロットされているわけではないが、先行研究との比較の参考にはなるだろう。Boxed Economy Simulation Plarform が、広がり

をもって示されているのは、どのようなモデルを作成するのかによって、その難易度が異なるためである。まず、狭義の複雑系をモデル化するのであれば、BESP は直接的な支援をしているため、他のツールに比べて圧倒的に優位にあると言える。そのため、その点を考慮するならば、モデル化能力は、プログラミング言語をそのまま使う場合や、Ascape、RePast、Swarm を使う場合よりも高いと言える。ただし、セル空間を用いるモデルであれば、それを直接的に支援している Ascape の方がモデル化が容易である。また、プログラミング言語としての Java 言語と比べた場合には、複雑なモデル (相互作用や多い、状態遷移が複雑、狭義の複雑系など) であれば、Java 言語でゼロから作成するよりは、BESP を用いたモデル化の方が容易である。これがフレームワークの効用であり、フレームワークに関する学習の負荷を考慮に入れたとしても、これらの複雑なモデルの作成においては利点があるように思われる。これに対し、状態遷移を持ち出すまでもない簡単な相互作用モデルや、セル状の簡単なモデルであれば、ゼロから作成した方が簡単なこともあり得る。

第6章　モデル・パターンの提案

6.1　モデル・パターンとは

これまでの章では、フレームワーク (Boxed Economy Foundation Model) における要素と、ソフトウェア (Boxed Economy Simulation Platform) におけるコンポーネントという部品を組み合わせることで、モデル全体を作成する仕組みについて提案してきた。このような仕組みは、大規模で複雑なモデルを作成する場合に、強力な支援の道具となるだろう。

しかし、モデル作成者に対する支援は、はたしてこれだけで十分だろうか。私は、時間の流れに伴う変化・生成を扱うモデルの作成支援としては、不十分だと考える。なぜなら、Boxed Economy Foundation Modelでは、モデル要素とその関係性について整理しているが、動的な振舞いの構成については構文的な定義しか行っていないからである。これに加えて、種々の動的な振舞いをどのように実現できるのかという整理が必要であると思われる。

そこで本章では、モデル要素をどのように組み合わせて、動的な振舞いを構成すればよいのかということを、「パターン」の考え方を取り入れてまとめることにしたい。パターンの考え方は、もともと建築デザインのために考案され、その後ソフトウェア・デザインに取り入れられたものである。本章ではさらに、モデル・デザインへ応用することを提案し、それを「モデル・パターン」と呼ぶことにしたい。本章でのモデル・パターンは、次章とその次の章におけるモデル事例を作成する際に用いることにする (各パターンの詳細は、付録 C にモデル・パターン カタログとして示してある)。

6.2　パターンによる記述

6.2.1　パターンとは何か

ここで取り上げる「パターン」とは、分析や設計の際に繰り返し現れる問題を明らかにし、その問題の解法をまとめたもののことである。パターンを利用することによる利点は、大きく分けて二つある。一つは、熟練者が自らの経験から得た経験則を明文化しているため、その問題の初心者であっても、効率的かつ洗練された方法でその問題を解決することができるという点である。もう一つは、その設計原理についての共通の語彙を提供するので、これまで直接指し示すことができなかった関係性などに

ついて、簡単に言及することができるようになるという点である。それゆえ、パターンは「複雑な設計を行うための建築素材である」(Gamma et al., 1995) といわれる。

6.2.2 パターンの基本構造

パターンは、「状況」、「問題点」、「解決策」という 3 つの観点で構成されるルールである (Alexander, 1979, 邦訳 p.202)。 状況とは、どのような時にそれを用いるのかということであり、パターンの適用条件である。問題点とは、何の問題を解決したいのかということであり、パターンを適用する目的である。解決策は、設計の要素や、それらの関連、責任、協調関係などである。これらに対し、問題とその解法を簡潔に記述した「名前」をつけることで、パターンが作成される(53)。

これらのパターンは、単独で使うというよりは、使う人が状況に合わせて組み合わせて使うことが前提となっている。それゆえ、それらを体系づけたパターン言語やパターンシステムが志向される。パターンの体系を得ることができれば、一つのパターンを適用した結果、新しい文脈が生まれ、そこに次のパターンを適用する ··· というように連鎖的な適用を行うことができるようになる。

なお、パターンを集めて体系化して記述したものを、パターンの「カタログ」と呼ぶ。カタログ内の各パターンは、統一したフォーマット (テンプレート) に従って書かれている。

6.2.3 パターンの役割

パターンを明示化し共有する第一の意義は、巧みな設計の再利用とその生成力にある。パターンは一種のルールであり、その目的を実現するためにどうすべきかを述べている。このような経験に基づいたパターンを知っていれば、そのような巧みな設計を再発見する必要がなくなるため、効率的によい設計を実現できる。それゆえ、その設計問題に対する初心者であってもパターンを身につけていれば、生成力のあるパターンの力を借りて、これまでの定石やよい設計を行うことも可能となる。「私たちの頭にあるパターンは動的であり、力をもち、生成力を備えている。それは私たちになすべきことを教えてくれる。それをいかに生成すべきか、または生成できるかを教えてくれる。さらに、一定の環境ではそれを作り出すべきだと教えてくれるのである。」(Alexander, 1979, 邦訳 p.151)。パターンは、繰り返し現れる問題と解法における関係性を抽象レベルで表しているため、それでどのような具体的な問題を扱うのかということは限定しておらず、多様な具体物を生成するための生成機構となり得るのである(54)。

パターンを明示化し共有する第二の意義は、設計に関する共通の語彙 (ボキャブラリ) を増やし、コミュニケーションを支援するということである。「パターンに名前を

付けることで、設計における用語の語彙を増やすことになる。それによって高い抽象レベルで設計することが可能となる。パターンに関する語彙が増えれば、同僚と議論したり、文書に記録したり、自分自身で考えを整理するのにも役立つ。設計に関して検討したり、設計上のトレードオフを人に伝えることも容易になる。」(Gamma et al., 1995, 邦訳 p.15)[55]

6.2.4 これまで提案されてきたパターン

建築におけるパターン

クリストファー・アレグザンダーは、建物や街の形態に繰り返し現れるものを観察し、それが要素の関係性、すなわちパターンであることを突き止めた。『時を超えた建設の道』で、建物や街を組み立てる本質を探っている際に、「『要素』は単純な積み木に見えても、実は変化しつづけ、現れるたびに異なる」(Alexander, 1979, 邦訳 p.73) ため、「この要素と称するものは空間を構成する究極の『原子』とは言えない」(同上) としている。そして、「要素と要素との関係も、要素そのもの以上にくり返し発生する」(同上, 邦訳 p.75) ということに注目する。ここで、各要素間の関係のパターンという考え方が登場するのである[56]。

アレグザンダーは、建物や街を構成するためのパターンの数はそれほど多くないと指摘し、建物は 2、3 ダースのパターンで定義でき、都市も 2〜300 のパタンで定義できるという (Alexander, 1979)。このように、比較的少ないパターンで世界が構成されるのは、「私たちは自分の頭にある類似のパタンからこの世界の現実のパタンをイメージし、心に描き、作り出し、建設し、そこに住む」ためであるという[57]。

Alexander (1977) では、8 年間の建設作業や計画作業をまとめた 253 のパターンが紹介されている。アレグザンダーは、このようなパターン言語を普及させることにより、デザイン・プロセスへのユーザー参加を実現しようとした。「これを活用すれば、隣人と共に自分の町や近隣を改良したり、家族と共に自宅を設計することができる。また関係者と力を合わせて、オフィス、作業場、学校のような公共建物も設計できる。さらにこのランゲージは、実際の工事手順の手引きとしても使える」(Alexander, 1977, 邦訳 p.ix) のである。

ソフトウェア開発におけるパターン

1980 年代後半に、建築の分野で提案されたパターンの考え方を、ソフトウェアの分野に適用するというヴィジョンが提案された (Beck and Cunningham, 1987; Rochat and Cunningham, 1988)。また、同じ頃、E. Gamma は GUI フレームワークの設計上のパターンを抽出・記述した (Gamma, 1991)。

ソフトウェア開発におけるパターンには、大別すると、「アナリシスパターン」、

「アーキテクチャパターン」、「デザインパターン」、「プログラミングパターン」などの種類がある。アナリシスパターンは、分析の際に繰り返し現れるパターンであり、ソフトウェアの設計や実装ではなく、ビジネスドメインの概念構造を反映したものである (Fowler, 1996)。アナリシスパターンに分類されるもののなかで代表的なものには、Fowler (1996) のアナリシスパターン、Coad et al. (1995) のパターン、Hay (1996) のデータモデルパターンがある (中谷および青山, 1999)。典型的な分析モデルの例を示しており、パターン言語の形式を採用していない。

アーキテクチャパターンは、典型的なソフトウェア全体の構造 (ソフトウェア・アーキテクチャ) に関するものである。構成要素とその役割、それらの関係について記述するのである。アーキテクチャパターンのカタログとして有名な Buschmann et al. (1996) では、「混沌から構造へ」「分散システム」「インタラクティブシステム」「適応型システム」などが提案されている[58]。このほか、Shaw et al. (1996) によってまとめられたアーキテクチャパターンもある。

デザインパターンは、設計段階において、優れた設計に繰り返し現れるオブジェクトとその構成を記述したものである (Gamma et al., 1995)。「種々状況における設計上の一般的な問題の解決に適用できるよう、オブジェクトやクラス間の通信を記述したもの」(Gamma et al., 1995, 邦訳 p.15) というように、アーキテクチャパターンに比べると、局所的な構造や振舞いをパターン化したものだといえる[59]。

プログラミングパターンは、イディオムとも呼ばれ、プログラミング言語に特化したプログラミングスタイルのことである。Coplien (1992) では、良いプログラムを書くのに重要なのは、文法だけでなく、「イディオム」と「スタイル」であると指摘している。例えば、Beck (1997) などのように、プログラミング言語に依存しているのが普通である。記述は、直接的にソースコードなどを提示することで、パターン言語よりも簡潔に書かれることが多い。

プロジェクトマネジメントのパターン

プロジェクトマネジメントのパターンは、開発組織やプロセスに関するパターンである。組織をパターンの観点で分析するのは新しいことではないが、Coplien (1995) はそれに生成的パターンとして持たせることを最初に提案した。Coplien (1995) は、「パターンは、とりわけ組織の構築と発展に適している。パターンは、関係のパターンによって文化を定義する、より現代的な文化人類学の基礎を形成する。」(Coplien, 1995, 邦訳 p.347) と述べている。その後、Ambler (1998); Ambler (1999) としてまとめられている。

この他にも、これまでの優れた手本としてのパターンではなく、悪い見本のパターンというのもまとめられている。このような反面教師のパターンは、Koenig (1998) で「アンチパターン」と名づけられ、その後、Brown et al. (1998); Brown et al.

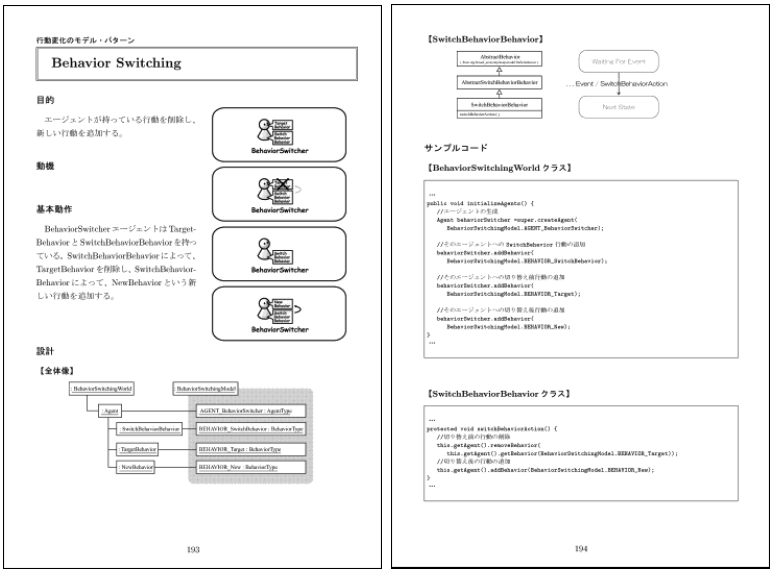


図 6.1: カタログ形式で記述されたモデル・パターンの例

(1999); Brown et al. (2000) で広く知られるようになった。Brown et al. (1998) は、ソフトウェアプロジェクトの失敗の原因となることを解説しており、Brown et al. (1999) は、「ソフトウェア構成管理（SCM）」のアンチパターンについて解説している(60)。

6.3 提案モデル・パターン

ここでは、モデル要素をどのように組み合わせて、動的な振舞いを構成すればよいのかということを、「モデル・パターン」(model patterns) としてまとめることにしたい。提案するモデル・パターンは、Boxed Economy Foundation Model に基づいてモデルを作成する際に頻繁に遭遇する問題に対して、どのようにモデル化すればよいのかを提示するものである。建築分野やソフトウェア開発におけるパターンと同様、モデル・パターンも、巧みな設計の再利用を可能とし、モデルの変化についての共通の語彙を増やしてコミュニケーションを支援する。

モデル・パターンは大きく分けて、「エレメンタリーなモデル・パターン」、「行動変化のモデル・パターン」、「コミュニケーションのモデル・パターン」、「アクティベーションのモデル・パターン」の 4 つに分けられる。各パターンの基本動作やサンプルコードなどの詳細は、本論文の付録 C のモデル・パターン カタログにまとめてある (図 6.1)。これらのパターンを分類すると、表 6.1 および図 6.2 のようになる。

6.3.1 エレメンタリーなモデル・パターン

第一のエレメンタリーなモデル・パターンは、モデル・パターンのうち、「エレメンタリー・パターン」(Wallingford, 1998) として捉えることができるものである。エレメンタリー・パターンとは、初心者に良いプログラミングを教えるための手段として用いるパターンである。エレメンタリーなモデル・パターンは、初心者が Boxed Economy Foundation Model に基づくモデルをどのように作成すればよいのかを示すものである。そのほとんどが、Behavior のアクション内で、1、2 行のプログラムで実行できる程度のものである。

6.3.2 コミュニケーションのモデル・パターン

第二のコミュニケーションのモデル・パターンは、他のエージェントとの Goods や Information のやりとりに関してのモデル化をまとめたものである。複数のエージェントに質問を投げて回収するなど、相互作用を含むモデルで頻繁に登場するパターンを集めてある。

6.3.3 行動変化のモデル・パターン

第三の行動変化のモデル・パターンは、行動の生成・削除・組み換え等のモデル化をまとめたものである。これらの行動変化は、すでに本論文の最初の方で述べたように、複雑系の記述において不可欠である。

6.3.4 アクティベーションのモデル・パターン

第四のアクティベーションのモデル・パターンは、エージェントの活性化に関するモデル化をまとめたものである。ある時間ステップにおいて、一部のエージェントにだけ行動を起こさせたい場合などのパターンがある。

6.4 発展のための覚書

本章では、Boxed Economy Foundation Model に基づくモデル化のためのモデル・パターンについて述べてきた。今後、これらのパターンは経験的に検証されて修正されるべきである。なぜなら、「パタンは単なる仮説にしかすぎない」(Alexander, 1979, 邦訳 p.218) からである。また、新しいパターンを追加していくことも重要である。これらのモデル・パターンを組み込んだモデル作成支援ツールを開発することも考えられるが、これは今後の課題である。

表 6.1: 本論文で提案するモデル・パターンの一覧

モデル・パターンの分類	モデル・パターン名
エレメンタリーなモデル・パターン	Agent Creation Relation Creation Related Agent Creation Agent Destruction Goods Creation Information Creation
コミュニケーションのモデル・パターン	Information Sending Blank Information Sending Internal Information Sending Immediate Reply Collect Immediate Replies Appointed Destination Reply Super BehaviorType Calling
行動変化のモデル・パターン	Behavior Creation Behavior Destruction Behavior Switching Temporary Behavior Attachment Requested Behavior Attachment Forced Behavior Attachment
アクティベーションのモデル・パターン	TimeEvent Distributer Agent TimeEvent Filtering TimeEvent Distributer Behavior Time-Consuming Behavior

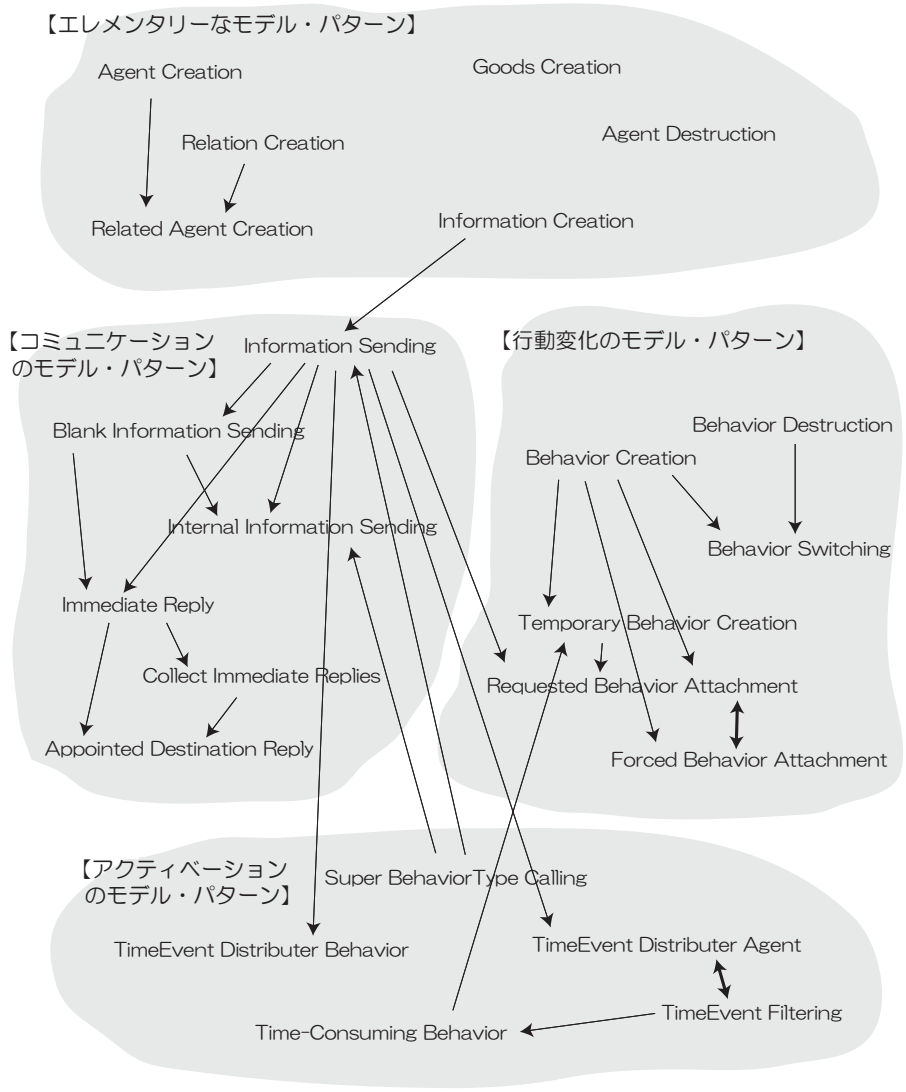


図 6.2: パターン間の関連

第7章　提案システムによる既存モデルの再現

本論文の提案 (モデル・フレームワーク、シミュレーション・プラットフォーム、モデル・パターン) の適用可能性を明らかにするために、ここでは、代表的な既存モデルを再現することにしたい。取り上げるモデルは、成長するネットワークモデル、繰り返しの囚人のジレンマモデル、貨幣の自生と自壊モデル、Sugarscape モデル、人工株式市場モデルの 5 つである。

7.1　成長するネットワークのモデル

近年、ノードの生成やリンクの生成・組み替え・除去など、ネットワークのトポロジーに影響するプロセスを扱うことのできる「成長するネットワーク」の理論研究が進んでいる。Barabási (2002) の指摘するように、現実世界に存在するネットワークは、そのほとんどが「成長する」という特徴をもっている。このような成長するネットワークについての理解を深めるためには、シミュレーションによるアプローチが不可欠であるが、Agent(ノード) や Relation(リンク) が動的に生成・削除できる BEFM / BESP は、このようなシミュレーションに適しているといえる。

ここでは、成長するネットワークの研究における代表的なモデルをいくつか再現することにしたい。まず最初に、ノード数が一定のままリンクが生成される「ランダムリンクモデル」を作成し、次に、新しいノードを生成してランダムにリンクを張っていく「ランダム選択成長モデル」、そして最後に、優先的選択によってリンクを張っていく「優先的選択成長モデル」を作成する。

7.1.1　ランダムリンクモデル

まず最初に作成するランダムリンクモデルでは、最初多くの孤立したノードが存在する。時間の経過とともに、このノード同士をランダムにつないでいく (図 7.1)。

このモデルの全体像は、図 7.2 のようになる。ここでは、2 人の Node エージェントを選んでリンクを張る処理を行う Organizer エージェントを用意する。シミュレーションの流れは、次のようになる (図 7.3)。Organizer エージェントは、TimeEvent を RandomNetworkingBehavior (図 7.4) で受け取り、Node エージェントの中からラン

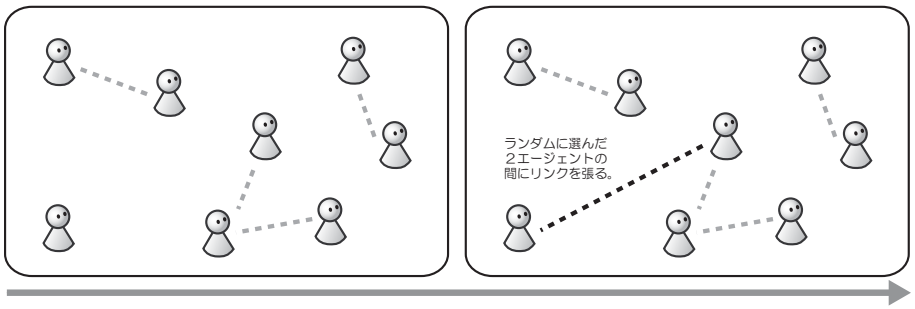


図 7.1: ランダムリンクモデルのイメージ

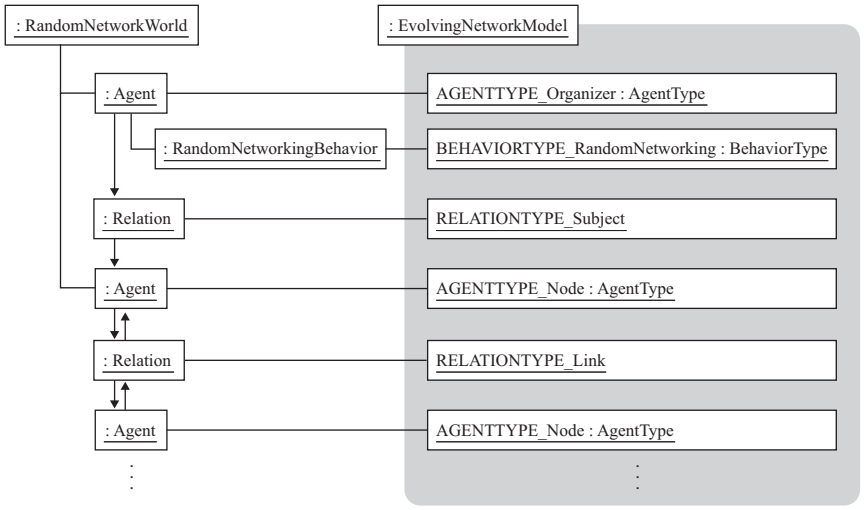


図 7.2: ランダムネットワークモデルの全体像

ダムに 2 人を選び、その 2 人の間に関係を結ぶ。

シミュレーション結果は、図 7.5, 7.6 のようになる。最初は、ノードとノードがつながったペアが生まれるだけだが、しばらくすると、ペアとペアがつながってクラスターが形成される。初期のクラスターは小規模なものだが、ある時、クラスター同士が結びついて巨大クラスターが出現する。その時々の最大クラスターに属するノードの数を時系列で描くと図 7.7 のようになる。

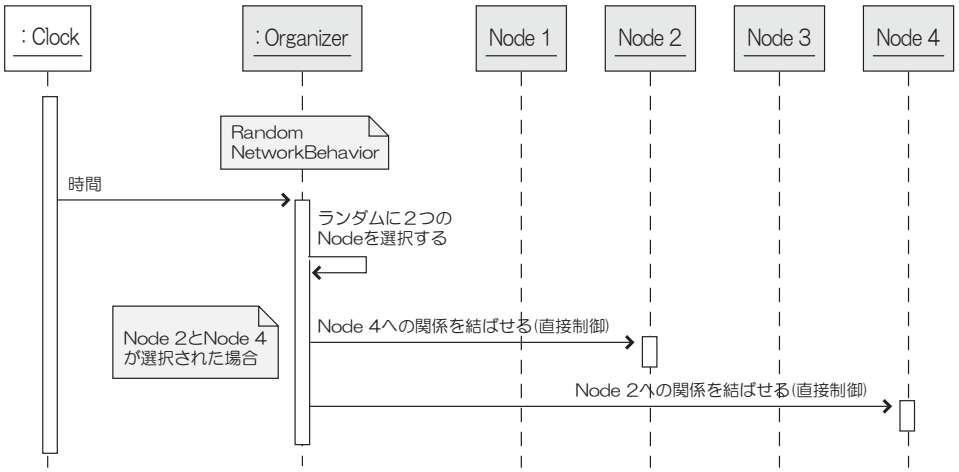
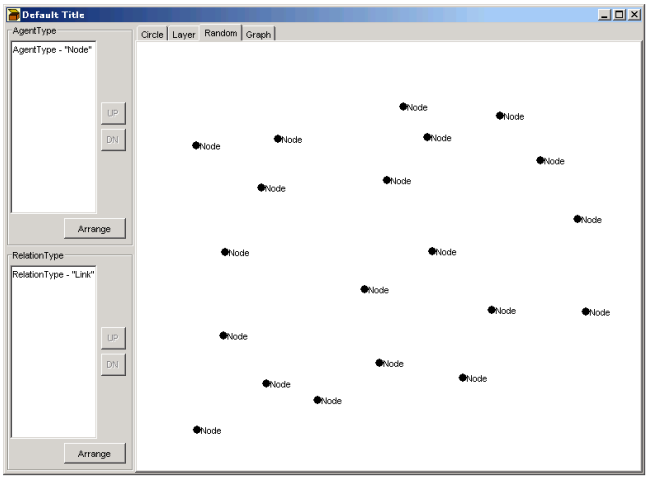


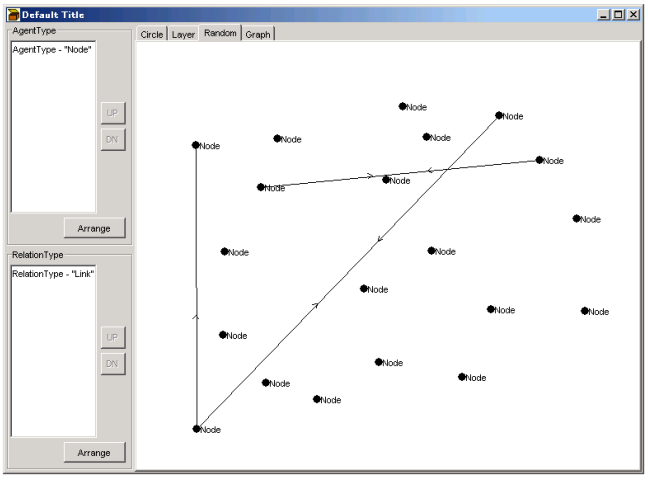
図 7.3: ランダムリンクモデルのシーケンス図

図 7.4: ランダムリンクモデル: RandomNetworkBehavior

step 0



step 3



step 12

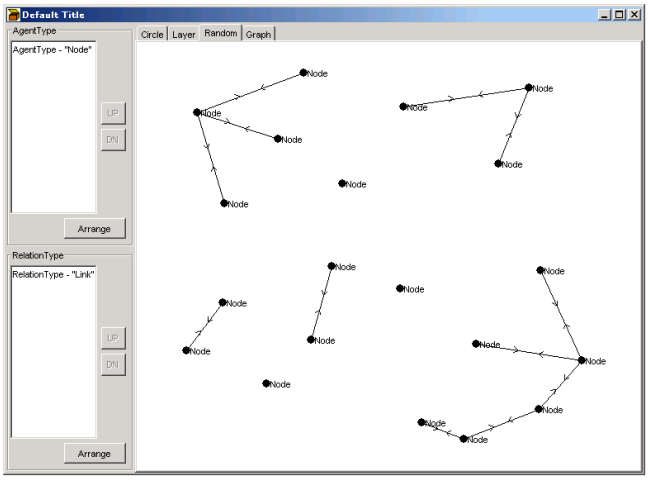
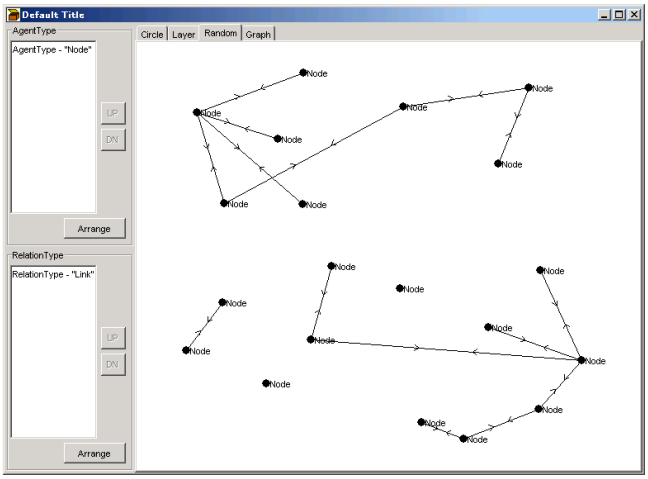
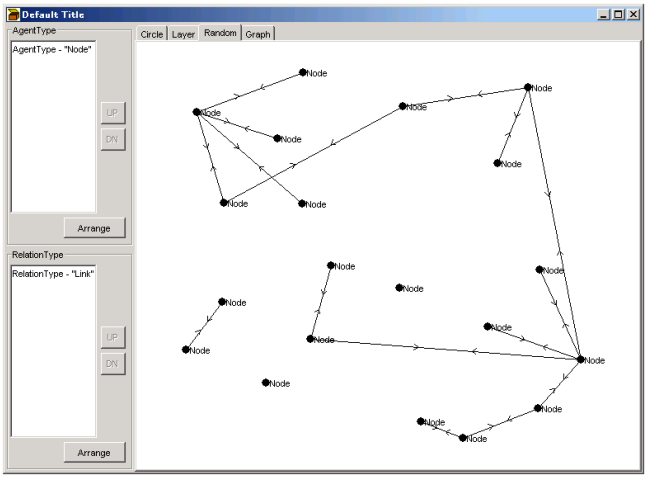


図 7.5: ランダムリンクモデルのシミュレーション結果 (1)



step 15



step 16

図 7.6: ランダムリンクモデルのシミュレーション結果 (2)

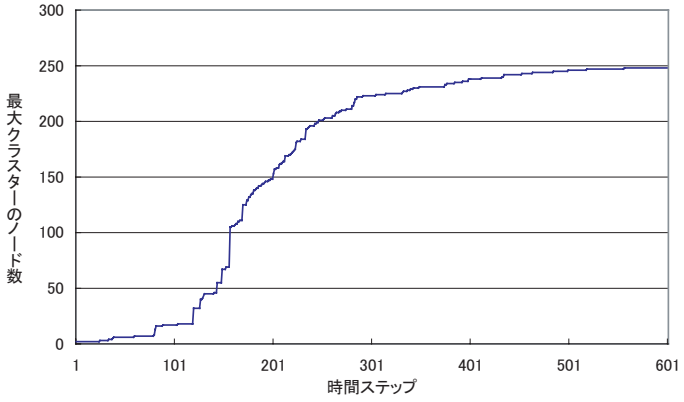


図 7.7: ランダムリンクモデルにおける最大クラスターのノード数の推移

7.1.2　ランダム選択成長モデル

成長するネットワークは、小さなネットワークから始めて、徐々にノードを追加していくというモデルで表現できる。ここではまず最初に、ランダム選択成長モデルを考えることにしよう。ランダム選択成長モデルでは、新しく追加されたノードは、ランダムに選ばれた 2 つのノードとリンクが張られる (図 7.8)。

図 7.9 は、このモデルの全体像である。新しい Node エージェントを追加し、既存の Node エージェントとリンクを張る処理を行う Organizer エージェントを用意するする。このシミュレーションの流れは、次のようになる (図 7.10)。TimeEvent を RandomAttachmentBehavior で受け取った Organizer エージェントは、Node エージェントを 1 人生成し、世界に追加する (図 7.11)。そして、既に存在する Node エージェントの中からランダムに 2 人を選び、さきほど追加した Node エージェントとの間に関係を結ぶ。

シミュレーションの結果は、図 7.12 のようになる[61]。このネットワークの各ノードのリンク数とその順位を両対数グラフにプロットすると、図 7.13 のようになる。このグラフから、度数分布が指数関数的な減少を示していることがわかる。

近年、友人関係や経済ネットワーク、ワールド・ワイド・ウェブ (WWW) などのような「成長するネットワーク」では、従来考えられてきたような均質的なネットワーク構造ではなく、多数のリンクをもつ「ハブ」が少数存在するという構造になっていることがわかっている。そして、そのリンクのつながれ方にはべき乗分布がみられる[62]。べき乗分布に従うネットワークでは、普通は少ない数のリンクをもっているが、ごく一部のノードは非常に多くのリンクをもっている。通常、自然界で起こる現象では、大きな変動の頻度は指数関数的に急激に減少するが、べき乗法則に従う場合には、このような「稀有な事象」も共存することになる。べき乗分布は両対数グラフで表すと、図 7.14 のように直線になる。このシミュレーションからわかることは、ランダム選択成長モデルでは、現実世界におけるネットワークは説明できないということである。

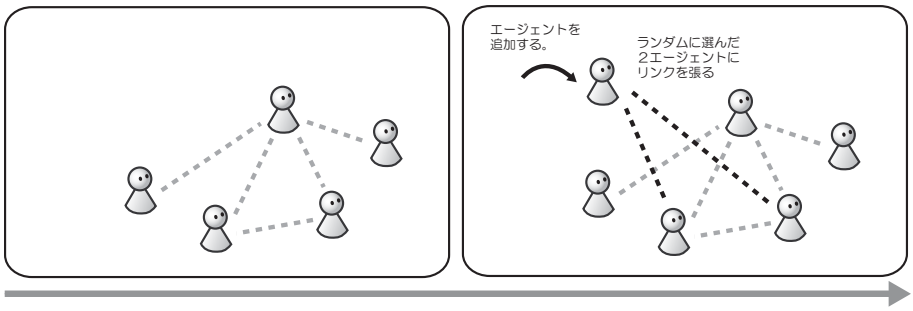


図 7.8: ランダム選択成長モデルのイメージ

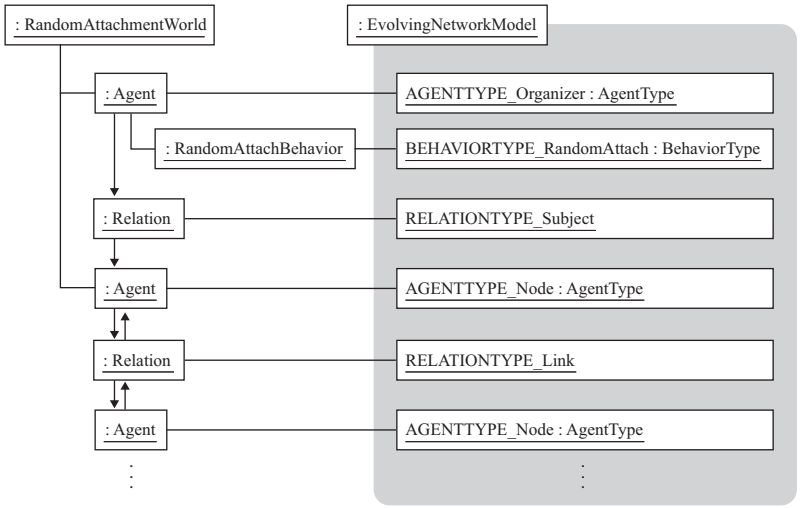


図 7.9: ランダム選択成長モデルの全体像

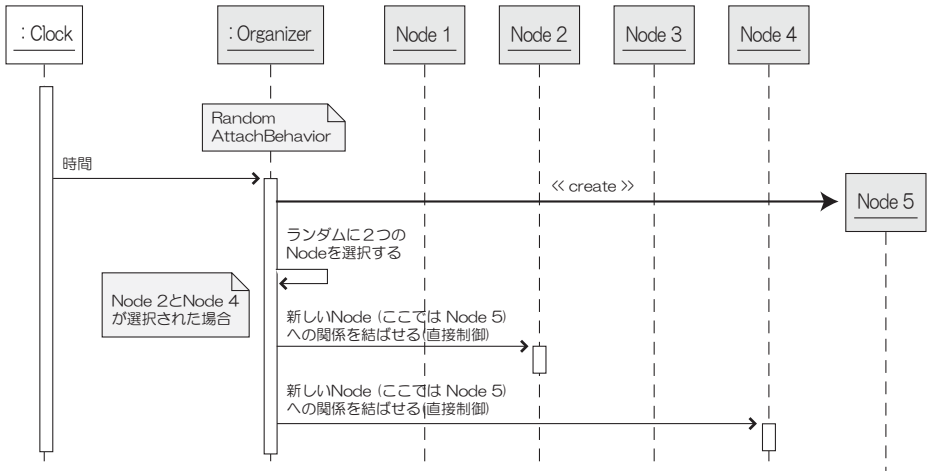


図 7.10: ランダム選択成長モデルのシーケンス図

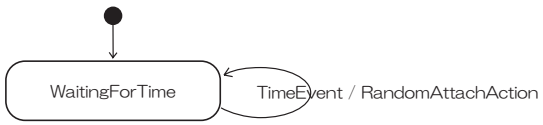


図 7.11: ランダム選択成長モデル: RandomAttachBehavior

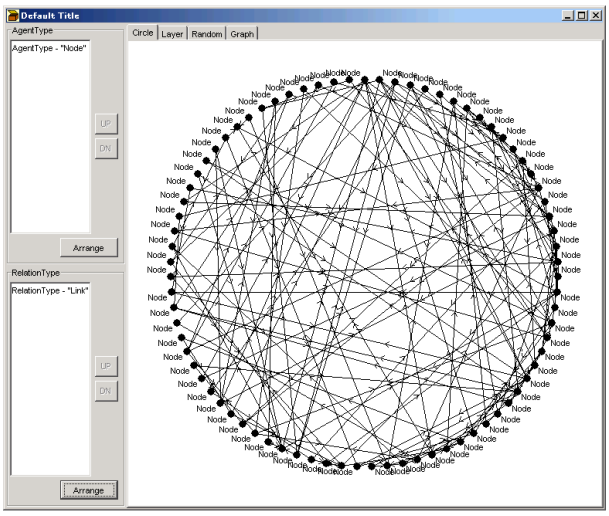


図 7.12: ランダム選択成長モデル: 形成されたネットワーク

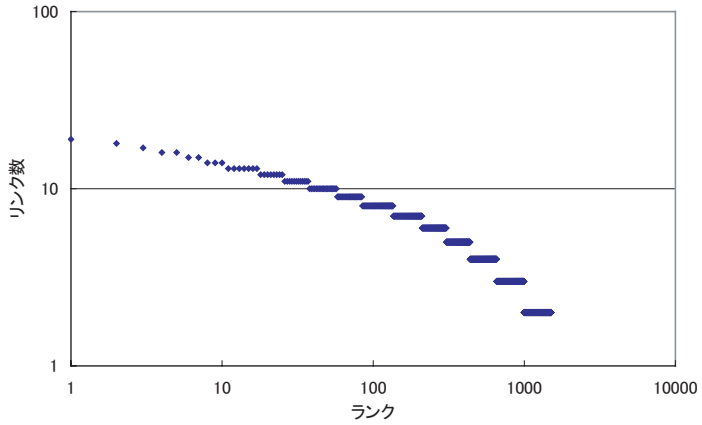


図 7.13: ランダム選択成長モデル: リンク数と順位の関係 (両対数グラフ)

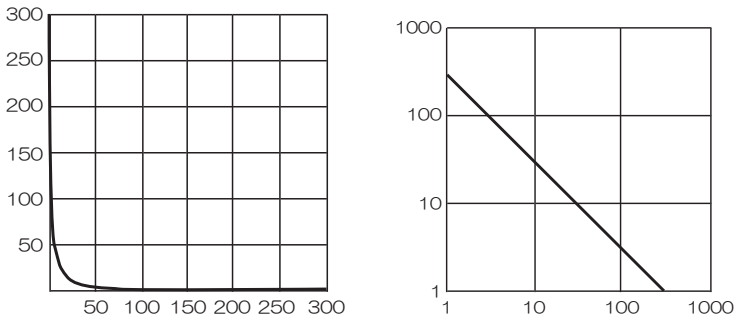


図 7.14: 線形グラフと両対数グラフにおけるべき乗分布

7.1.3　優先的選択成長モデル (スケールフリー・モデル)

現実世界でみられるようなべき乗の結合分布をもつネットワークは、どのようにすれば形成することができるのだろうか。この問題に取り組むには、「現実のネットワークでは、リンクがランダムに張られたりはしない」(Barabási, 2002) という点に注目することが重要となる。現実には、ウェブページでも友人関係でも、リンク数が多いノードほどますます多くのリンクを獲得するということが起こっている。実はこの原理を用いることで、現実に近い成長するネットワークを作成できることがわかっている。

Barabási et al. (1999) は、「成長」と「優先的選択」という 2 つの基本原理によって、結合分布がべき乗になるネットワークを生成できることを解析的に示した。これらの原理は、どちらか片方だけではべき乗の結合分布を生み出さないため、両者とも不可欠であることがわかっている。「成長」とは、新しいノードを加えるということであり、「優先的選択」とは、新しいノードを追加する際に、多くのリンクを持っているノードを優先的に選択することである (図 7.15)。新しいノードが、k 個のリンクをもつノードにリンクを張る確率は、次の式で与えられる。

$$prob = \frac{k}{\sum_i k_i}$$

図 7.16 は、このモデルの全体像である。新しい Node エージェントを生成・追加して既存の Node エージェントとリンクを張る処理を行う Organizer エージェントを用意する。このシミュレーションの流れは、次のようになる (図 7.17)。Organizer エージェントは、TimeEvent を PreferentialAttachBehavior (図 7.18) で受け取り、Node エージェントを 1 人生成し、世界に追加する。そして、既に存在する Node エージェントのもつリンク数を調べ、もっているリンク数に比例した確率で Node エージェントを 2 人選ぶ。そしてこの 2 人の Node エージェントと、新しく追加した Node エージェントの間にリンクを結ぶ。

このシミュレーションの結果は、図 7.19 のようになる。非常に多くのリンクをもつ

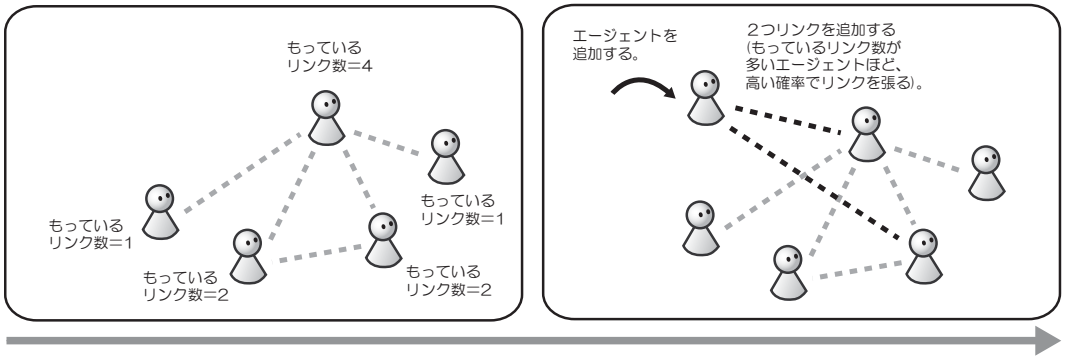


図 7.15: 優先的選択成長モデルのイメージ

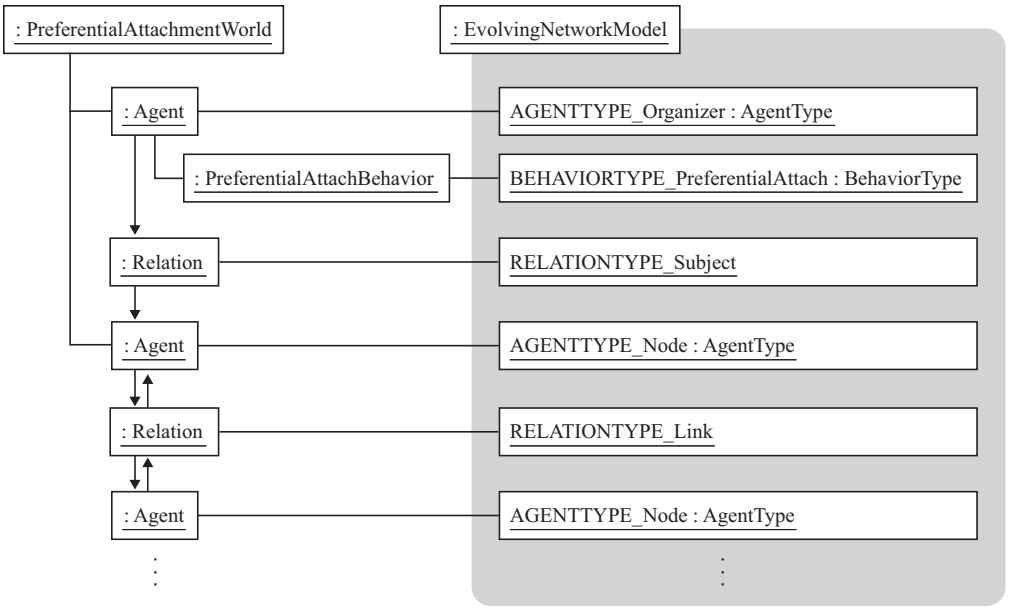


図 7.16: 優先的選択成長モデルの全体像

ハブが出現していることがわかる。この結果を両対数グラフで表すと、図 7.20 のようにベキ乗になっていることが確認できる(63)。このモデルは、最初にスケールフリーのべき法則を説明したため、「スケールフリー・モデル」と呼ばれるようになった。このモデルによって、「まず第一に、ベキ法則によってハブの存在に正当性が与えられた。次に、スケールフリー・モデルによって、現実のネットワークに見られるベキ法則が、数学的基礎をもつ概念上の進歩に格上げされた。さらには、進化するネットワークという洗練された理論に支えられて、スケーリング指数やネットワークのダイナミクスが精密に予測できるようになった。」(Barabási, 2002, 邦訳 p.135) といわれている。

また、社会・経済シミュレーションの研究における意義も大きい。というのは、社会・経済のシミュレーションを行う場合には、何らかの社会ネットワークを想定する必要があるが、その際に現実と同型のスケールフリー・ネットワークを用いることで、より現実に近いモデルになるからである。また、動的に成長する社会ネットワークにおける社会・経済現象を分析することも可能になる。

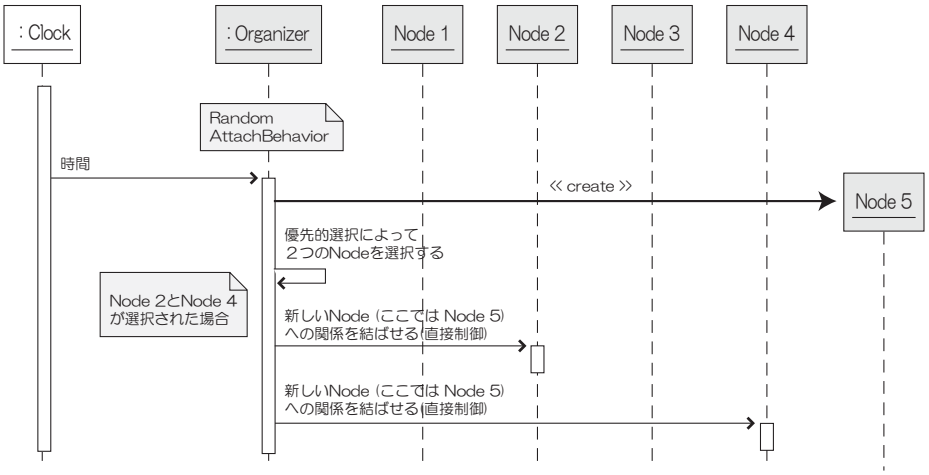


図 7.17: 優先的選択成長モデルのシーケン ス図

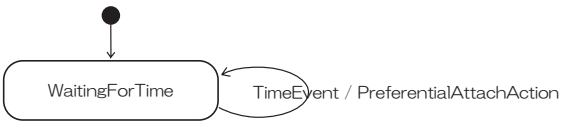


図 7.18: 優先的選択成長モデル: PreferentialAttachBehavior

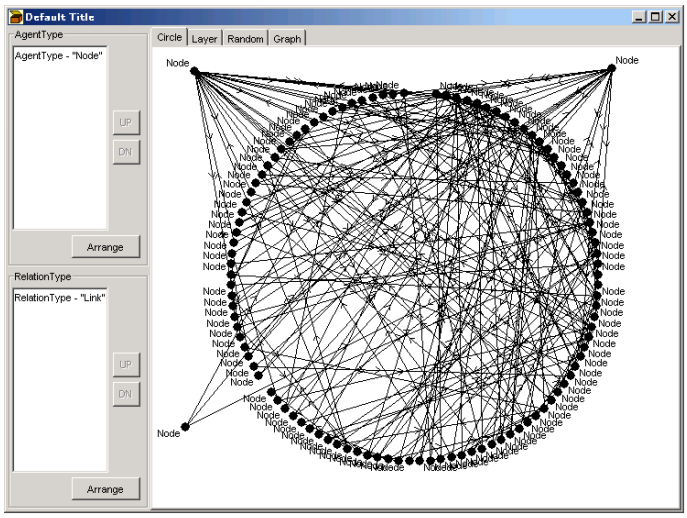


図 7.19: 優先的選択成長モデル: 形成されたネットワーク

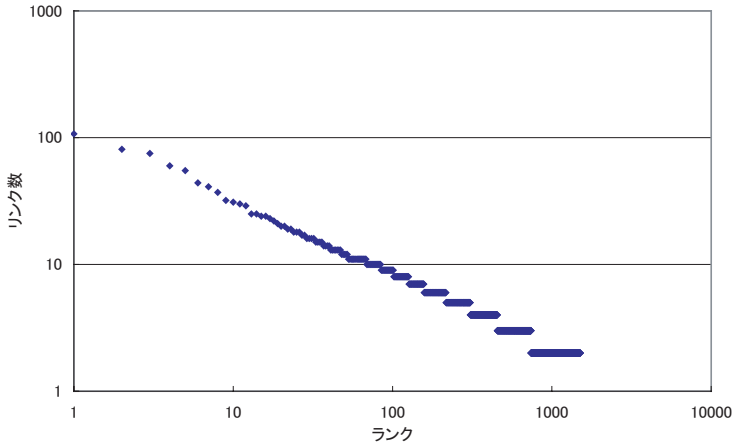


図 7.20: 優先的選択成長モデル: リンク数と順位の関係 (両対数グラフ)

7.2　繰り返し囚人のジレンマモデル

「囚人のジレンマ」は、利己的な主体間で利害が対立する状況の中で、どのように協調が形成されるのかを調べる枠組みとしてしばしば用いられている。このジレンマは、1950 年頃、心理学研究のなかで M.Flood と M.Dresher によって提唱されたものであり、A.W.Tucker が「囚人のジレンマ」というストーリー仕立てで広めてからは、政治学や経済学、社会学において、ジレンマの社会的モデルとして頻繁に用いられてきた。二人のプレイヤーが、それぞれ独立に、協調 (Cooperation) か裏切り (Defection) かのどちらかの行動をとり、自分の選択と相手の選択の組合せによって、得られる利得が異なるようになっている。

囚人のジレンマは一回限りのゲームであるが、これを反復的に行うというゲームの考察も行われており、これを「繰り返し囚人のジレンマ」という(64)。この繰り返しの囚人のジレンマは、あらかじめわかっている有限回の対戦であれば、裏切る方がより高い利得を得られることがわかっている。しかし、いつまで続くかわからない場合には、必ずしもそのような結果にはならず、万能の戦略がないといわれている。

ここでいう「戦略」とは、各回の選択のことではなく、過去の手を踏まえて自分の手を決めるための行動決定規則である。同じ戦略でもこれまでの経緯によって (選択の履歴によって)、協調することもあれば裏切ることもある。自分の状態によって反応が異なるという意味で、このモデルは、本論文でいうところの広義の複雑系のモデルになっている。また、戦略を変更するということが起こるならば、それは狭義の複雑系と捉えることができる。

7.2.1　コンテスト・シミュレーション

ここでは、まず最初に、広義の複雑系のモデルとして、コンテスト・シミュレーションを行う。戦略の状態の変化は、BEFM の Behavior の状態変化でモデル化する。

モデルの基本的な枠組みを説明すると、シミュレーションには、1 人の Referee エージェントと、複数の Player エージェントが登場する (図 7.21)。この Referee エージェントは、コンテストを仕切る役割を担っている。Referee エージェントは、Player エージェント同士が総当りになるように、順番に 2 人ずつ Player エージェントを呼び出し、試合を行わせる。このとき、試合では 200 回の対戦が行われる。各対戦における得点は、両者が協調すれば 3 点ずつ、裏切りあえば 1 点ずつ、片方だけ協調し他方が裏切れば、それぞれ 0 点と 5 点となる。試合が終了すると、最終的な得点が記録される。こうして試合が次つぎに行われ、総当りが実現したら、そのコンテストが終了し、Referee エージェントは結果を公表する。コンテストと試合と対戦の関係は、図 7.22 のようになる。

Referee エージェントは、コンテスト全体を管理する ManageContestBehavior と、

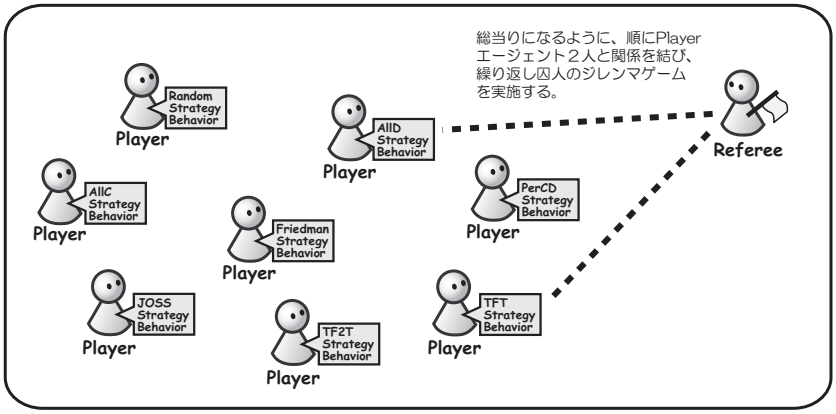


図 7.21: コンテスト・シミュレーションのイメージ

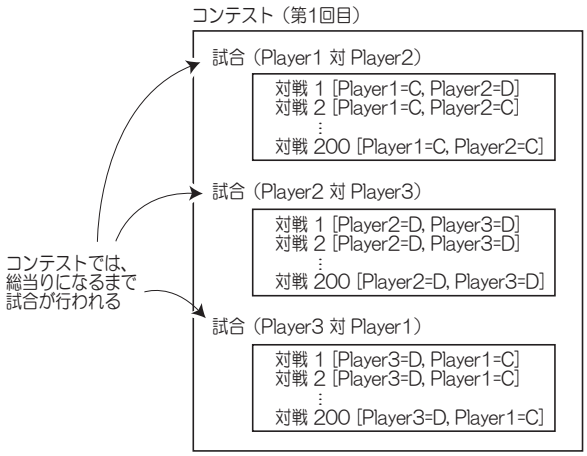


図 7.22: コンテストと試合と対戦の関係

各試合を取り仕切る ConductMatchBehavior をもっている。Player エージェントは、Referee エージェントとコミュニケーションをとるための PlayBehavior と、戦略行動をもっている (図 7.23)。対戦では、各 Player エージェントは、前回の相手の手のみが知らされ(65)、自分の次の手を決めていくことになる。戦略行動は、戦略の状態変化を Behavior の状態変化で表現したものであり、次のような種類がある。

ALL-C 相手の手に関係なく、必ず協調する (ALLCStrategyBehavior; 図 7.25)。

ALL-D 相手の手に関係なく、必ず裏切る (ALLDStrategyBehavior; 図 7.26)。

RANDOM 相手の手に関係なく、協調と裏切りをランダムに選択する (RandomStrategyBehavior; 図 7.27)。

TFT 最初は協調し、次からは相手が前回とった行動を真似する (TFTStrategyBehavior; 図 7.28)。

図 7.23: 戦略を行動として表現する

TF2T 最初は協調し、2 回連続して相手が裏切ったときに、裏切る (TF2TStrategyBehavior; 図 7.29)。

FRIEDMAN 最初は協調し、相手が裏切らないかぎり協調を続ける。相手が一度でも裏切ると、それ以降はずっと裏切り続ける (FRIEDMANStrategyBehavior; 図 7.30)。

JOSS TFT (しっぺ返し) と同様に、最初は協調し、相手に裏切られると裏切り返す。相手が協調した場合には、9 割協調して、1 割裏切る (JOSSStrategyBehavior; 図 7.31)。

PER-CD 協調、裏切り、協調、裏切り …· を繰り返す (PERCDStrategyBehavior; 図 7.32)。

PER-CCD 協調、協調、裏切り、協調、協調、裏切り …· を繰り返す (PERCCDStrategyBehavior; 図 7.33)。

モデルの全体像は、図 7.24 のようになり、シミュレーションの流れは、次のようになる (図 7.34)。最初に TimeEvent を受け取るのは、Priority が高く設定されている Referee エージェントである。Referee エージェントは、ManageContestBehavior(図 7.35) で TimeEvent を受け取り、総当たりの対戦表を作成する。その後、ConductMatchBehavior(図 7.36) に対戦組合せの情報を 1 組分ずつ渡し、その試合を開始する。

Referee エージェントは、今回対戦する 2 人の Player エージェントに、まず初回の手を尋ねる。Player エージェントは PlayBehavior(図 7.37) で連絡を受け取り、それぞれのもつ StrategyBehavior に初回の手を出すよう連絡する。StrategyBehavior は初回の手を返答し、PlayBehavior を通じて、Referee エージェントに伝える。Referee エージェントは、ConductMatchBehavior で返答を受け取り、2 人の Player エージェントの手を照合して、今回の得点を計算する。

2 回目以降の対戦では、Referee エージェントは、各 Player エージェントに対戦相手の前回の手を知らせて、次の手を求める。Player エージェントは対戦相手の前回の手を PlayBehavior で受け取り、その手を StrategyBehavior に伝える。そして、StrategyBehavior から次の手を受け取り、それを Referee エージェントに返答する。この返答を Referee エージェントは ConductMatchBehavior で受け取り、各 Player エージェントの手を照合して、今回の得点を計算する。そして、それぞれの Player エージェントがこれまでに獲得した得点に今回の得点を加算する。200 回の対戦が終了するまで、以上の処理を繰り返す。

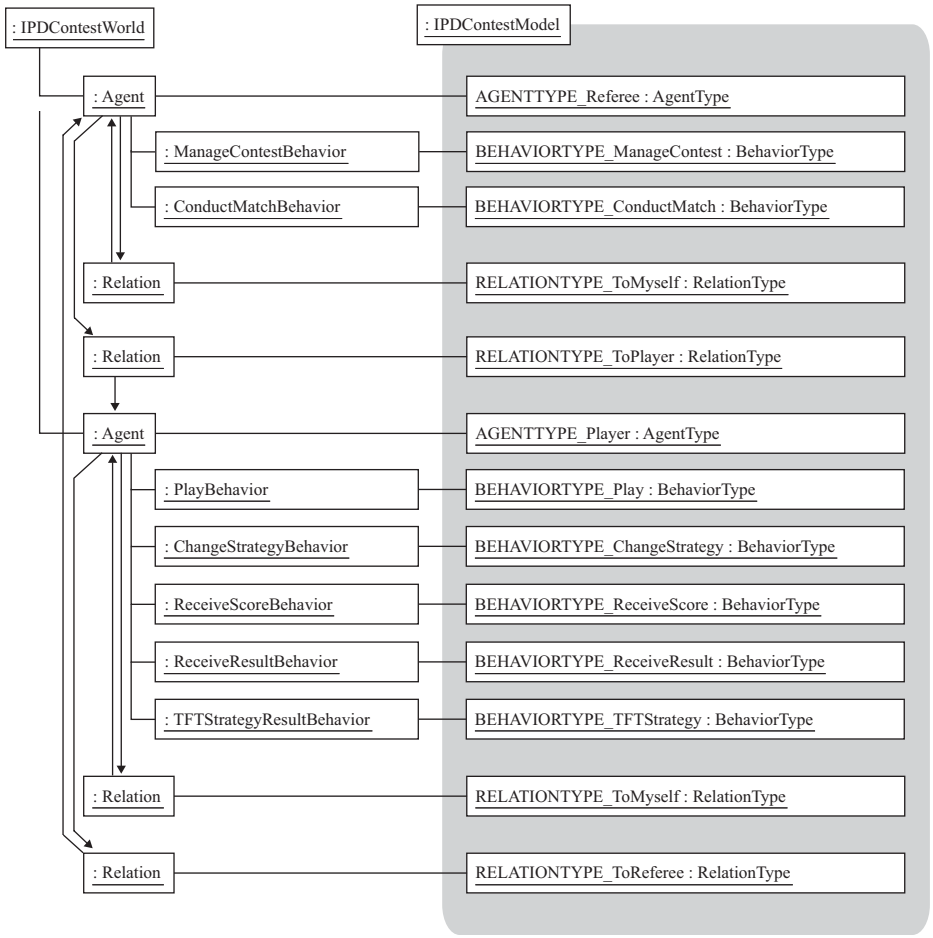


図 7.24: コンテスト・シミュレーションの全体像

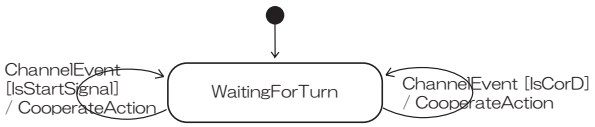


図 7.25: 戦略行動: ALLCStrategyBehavior

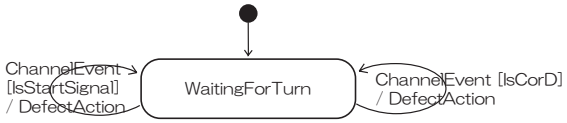


図 7.26: 戦略行動: ALLDStrategyBehavior

200 回の対戦終了後、Referee エージェントは、対戦した Player エージェントに、今回の試合におけるお互いの得点を知らせる。Player エージェントは、ReceiveScoreBehavior でその情報を受け取り、記憶する。次に Referee エージェントは、今回の試合の得点を、自らの ManageContestBehavior に知らせる。知らせを受けた ManageContestBehavior は、今回の試合の得点を、これまでの試合で各 Player エージェントが獲得した総得点のリストに加算する。こうして 1 組の試合が終了する。以上の処理を、すべての Player エージェントが総当りで試合を終えるまで繰り返す。

総当たりを終了した後、Referee エージェントは、今回のコンテストの総得点のリストを受け取り、すべての Player エージェントに知らせる。Player エージェントは、ReceiveResultBehavior で総得点のリストを受け取り、記憶する。

このシミュレーションの結果は、表 7.1 のようになった。同じ戦略でも総得点が異なるのは、乱数を用いる戦略 (RANDOM と JOSS) があるからである。

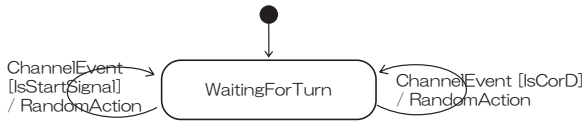


図 7.27: 戦略行動: RandomStrategyBehavior

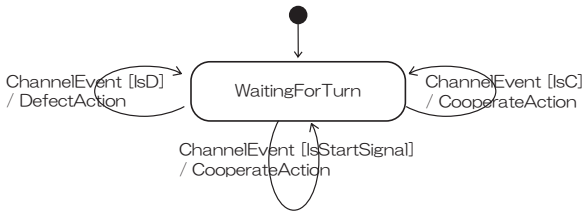


図 7.28: 戦略行動: TFTStrategyBehavior

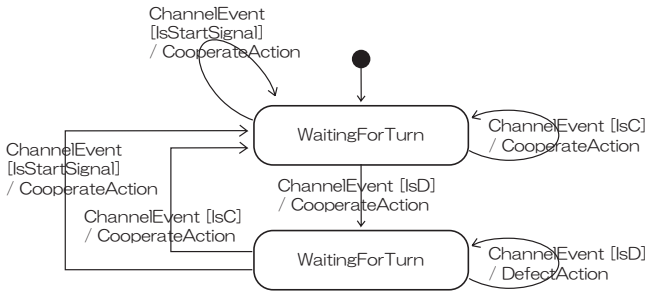


図 7.29: 戦略行動: TF2TStrategyBehavior

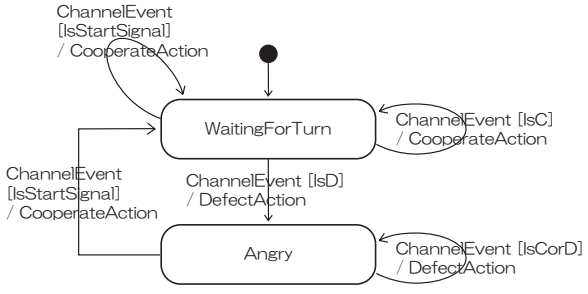


図 7.30: 戦略行動: FRIEDMANStrategyBehavior

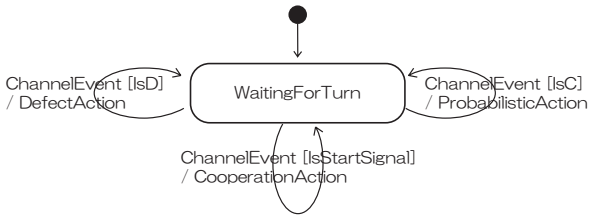


図 7.31: 戦略行動: JOSSStrategyBehavior

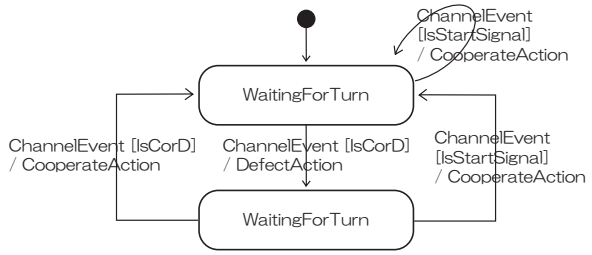


図 7.32: 戦略行動: PER-CDStrategyBehavior

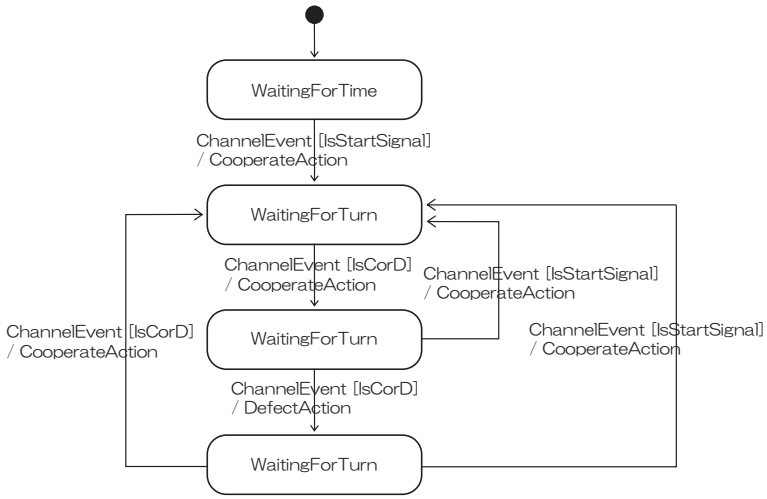


図 7.33: 戦略行動: PER-CCDStrategyBehavior

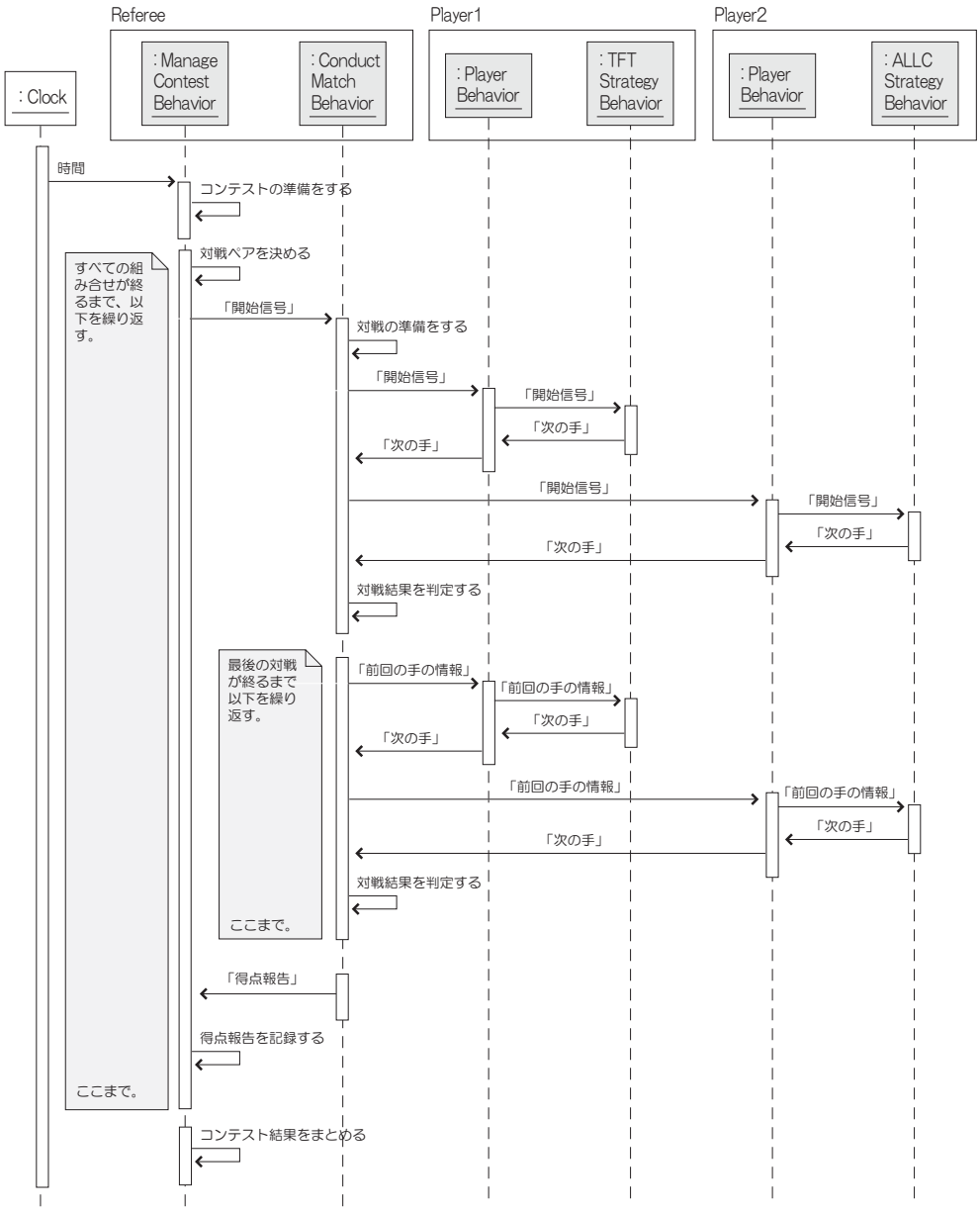


図 7.34: コンテスト・シミュレーションのシーケンス図

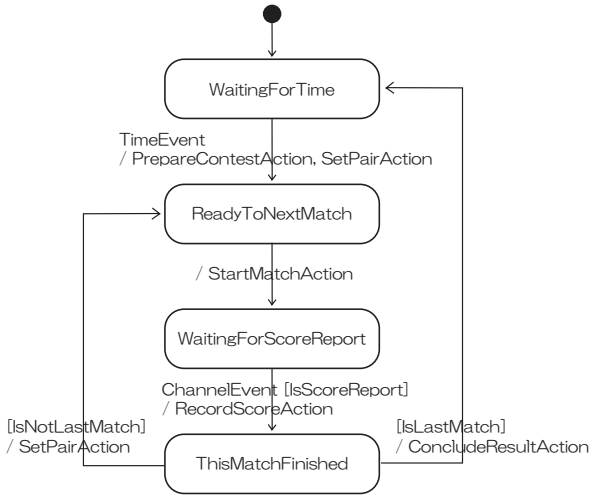


図 7.35: コンテスト・シミュレーション: ManageContestBehavior

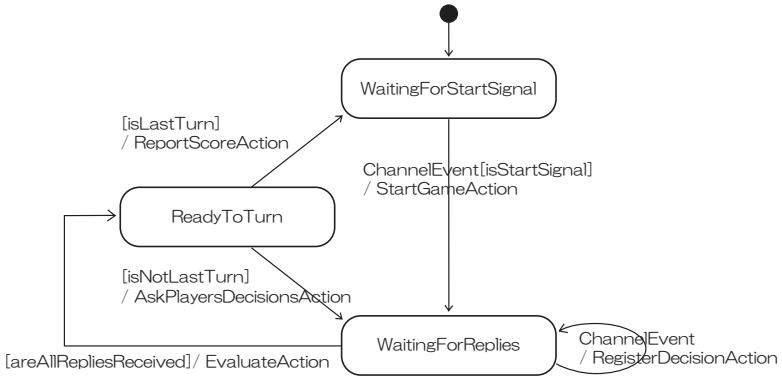


図 7.36: コンテスト・シミュレーション: ConductMatchBehavior

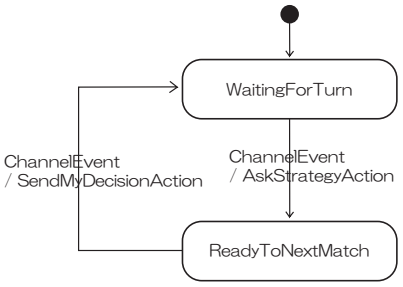


図 7.37: コンテスト・シミュレーション: PlayBehavior

表 7.1: コンテスト・シミュレーションの結果

戦略	総得点
FRIEDMAN	8872
FRIEDMAN	8824
TFT	8128
TFT	8041
PERCD	8011
PERCD	8001
TF2T	7846
TF2T	7799
ALLD	7716
ALLD	7684
RANDOM	7481
RANDOM	7468
PERCCD	7384
PERCCD	7348
ALLC	7296
ALLC	7275
JOSS	7197
JOSS	7281

7.2.2　戦略模倣シミュレーション

ここでは、繰り返し囚人のジレンマのコンテスト・シミュレーションに、独自の拡張を行ってみることにしたい。その拡張とは、各コンテスト終了後に、各 Player エージェントが自分より強い相手の戦略を模倣するというものである(66)。模倣相手の候補の選択は、次の 2 つの方法のいずれかで行い、候補が複数の場合には、その中からランダムに選択することにする(67)。

1. 個別対戦において、自分に勝ったプレイヤーの戦略を採用する (試合結果による選択)。
2. コンテストにおける総得点が、自分よりも高いプレイヤーの戦略を採用する (コンテスト結果による選択)。

この拡張モデルでは、それぞれの Player エージェントが戦略の変更を行うが、これは行動のルールが変化するという意味で、本論文でいうところの狭義の複雑系のモデルになっている。

戦略模倣シミュレーションでは、コンテスト・シミュレーションの最後の部分に、次のような流れを追加する (図 7.38)。TimeEvent によって発火する Referee エージェントの一連の行動が終わった後、Player エージェントが TimeEvent を受け取る。Player エージェントは、TimeEvent を ChangeStrategyBehavior で受け取り、他者を模倣して戦略変更する (図 7.39)。

戦略ごとに 2 人ずつ Player エージェントを用意したシミュレーションを行ったところ、結果は次のようになった。まず、候補選択方式 1(試合結果による選択) の場合には、数ステップで「ALL-D」戦略のみになり (図 7.40)、平均得点は初期状態よりも低い水準になる (図 7.41)。最終的に「ALL-D」戦略のみになった状況では全員が裏切りあうため、社会的にみて得点が低い水準になるのである。この結果を Player エージェントの個別状況で見てみると、次のことがわかる (図 7.42)。協調を基本とする戦略は、裏切りを交える戦略よりも得点が低いことが多く、また、対戦相手が協調を基本とする場合には同点になるため、戦略模倣の候補にはなりにくい。これに対し、裏切りを交える戦略は、協調を基本とする戦略よりも高い得点を獲得するため、戦略模倣の候補となる。裏切りを交える戦略のなかでも、裏切りが多いほど得点が高くなるため、「ALL-D」戦略が広まることになる。この結果は、乱数シードを変えても同様の結果になる。

これに対し、候補選択方式 2(コンテスト結果による選択) の場合には、「FRIEDMAN」戦略が広まり (図 7.43)、平均得点は初期状態よりも高い水準になる (図 7.44)。平均得点が候補選択方式 1(試合結果による選択) の場合に比べて高いのは、「FRIEDMAN」戦略における協調の効果である。最終的に「FRIEDMAN」戦略のみになったときには、すべての対戦で協調するため、社会的にみて得点が高い水準になるのである。こ

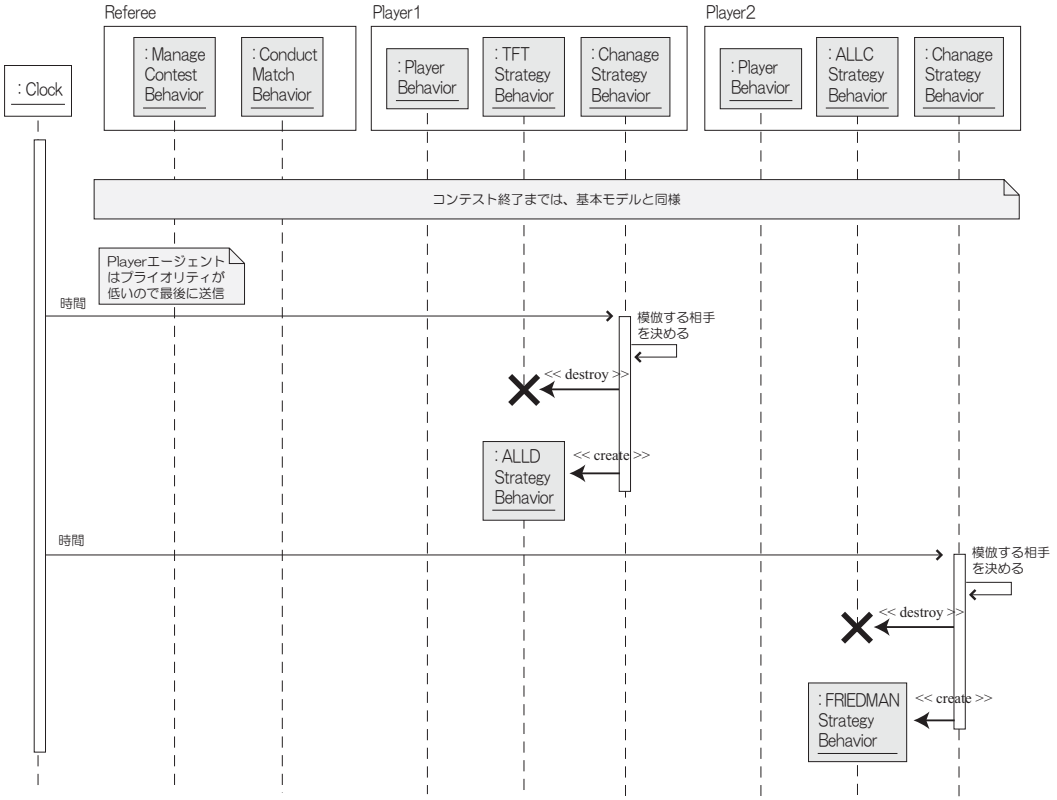


図 7.38: 戦略模倣シミュレーションのシーケンス図

の結果を、Player エージェントの個別状況で見てみると、協調を基本とするいくつかの戦略が残った後に、「FRIEDMAN」戦略が広まっていることがわかる (図 7.45)。協調を基本とする戦略の中でも特に「FRIEDMAN」戦略が残るのは、「ALL-D」戦略などの裏切りの多い戦略に対して効果的に反撃することができるためだと考えられる。この候補選択方式 2(コンテスト結果による選択) では、乱数シードを変更して実行すると、「FRIEDMAN」戦略に混じって「TFT」戦略が残ることがある (図 7.46)。この場合にも、両戦略とも協調するので得点は高い水準になる (図 7.47)。

以上の結果をまとめると、次のようになる。まず、試合結果というミクロ的で個別的な勝敗を判断材料にして戦略を変化させると、裏切りが強調されて加速度的に広まっていく。これに対し、コンテスト結果というマクロ的で総合的な勝敗を判断材料にして戦略を変化させると、協調による得点の上昇が効果を発揮し、協調が加速度的に広まっていく。裏切りがもっている得点のインパクトに比べ、協調の効果は静かなものであるが、総合的に見た場合には、この効果が大きくなる。以上のシミュレーション結果だけで何かを主張することはできないが、社会全体への公表や表彰の存在が、社会的効用を高めるのに効果がある可能性が示唆されたといえるだろう。

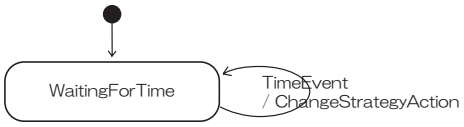


図 7.39: 戦略模倣シミュレーション: ChangeStrategyBehavior

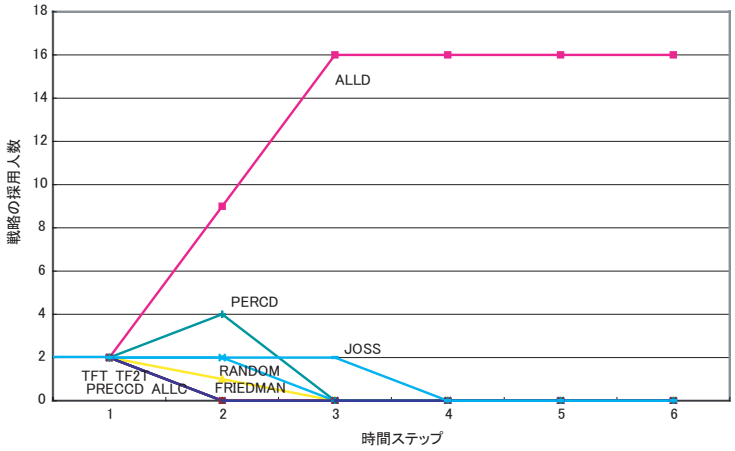


図 7.40: 戦略模倣シミュレーション: 各戦略を採用しているプレイヤー数の推移 (試合結果による戦略変更)

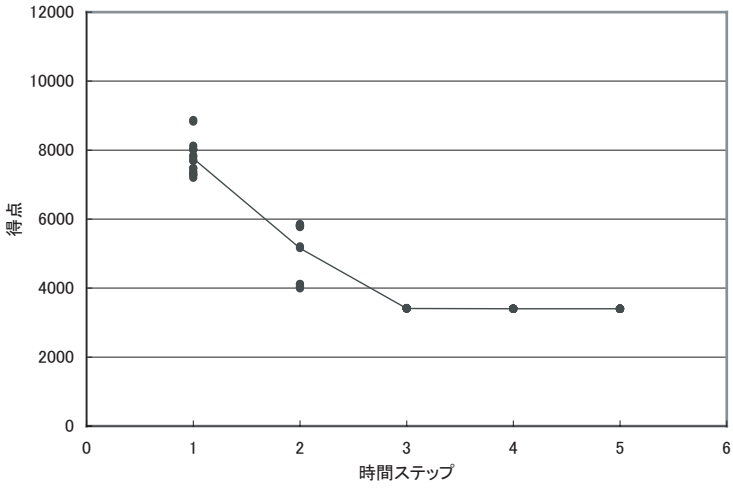


図 7.41: 戦略模倣シミュレーション: 各プレイヤーの得点と平均得点の推移 (試合結果による戦略変更)

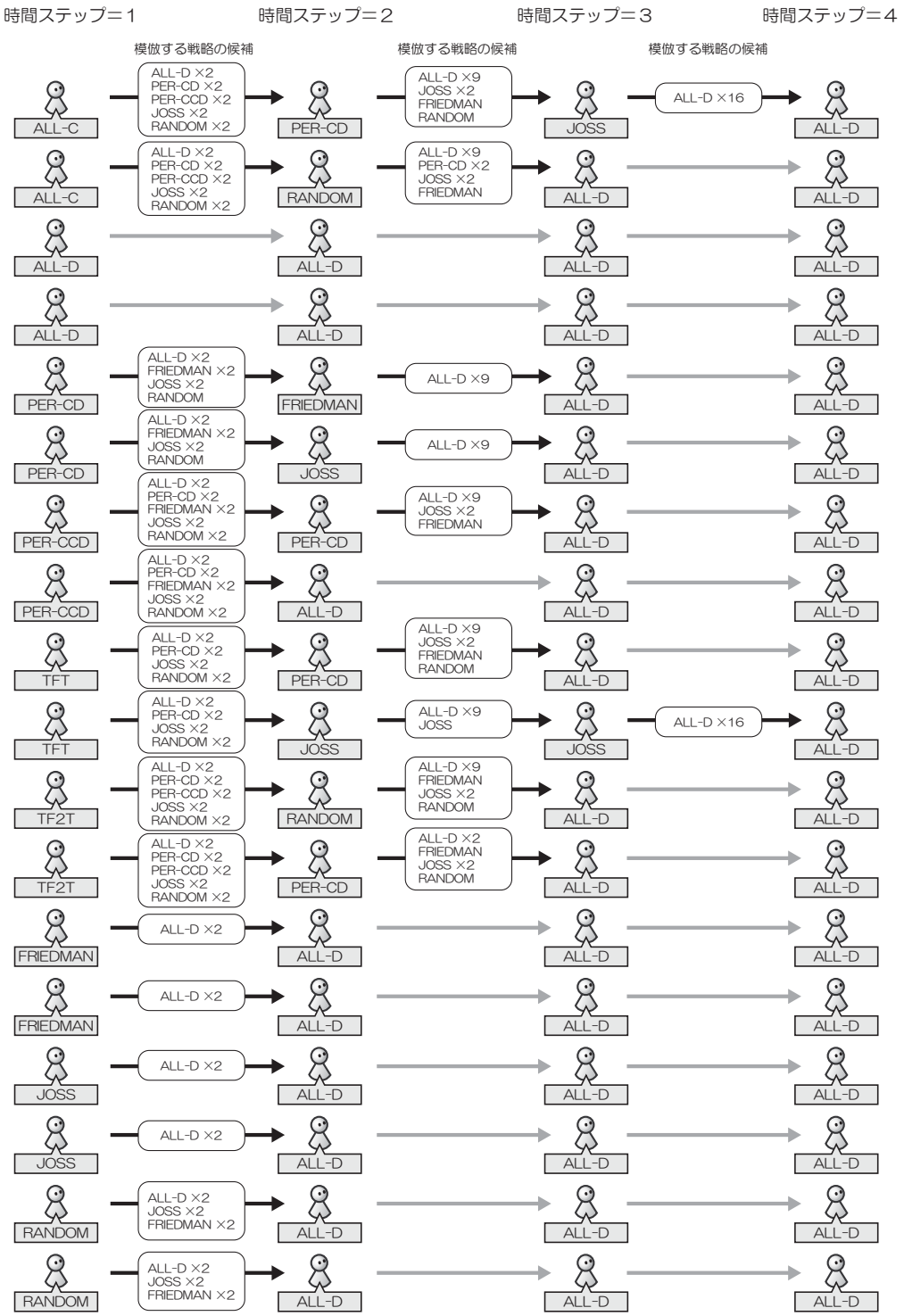


図 7.42: 戦略模倣シミュレーション: プレイヤーの戦略の変化 (試合結果による戦略変更)

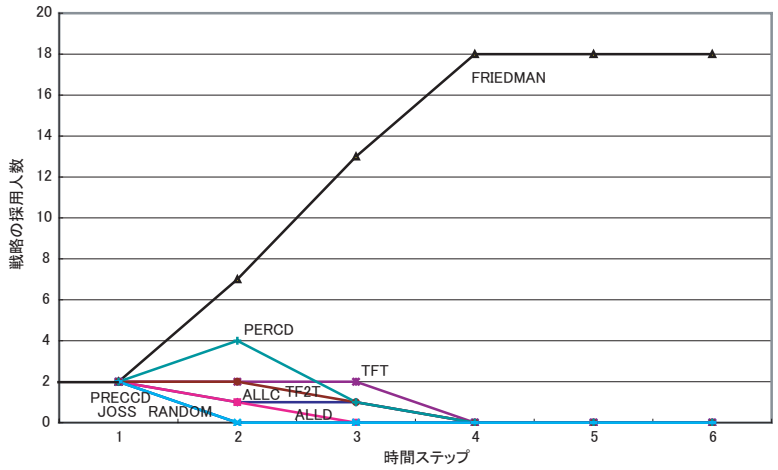


図 7.43: 戦略模倣シミュレーション: 各戦略を採用しているプレイヤー数の推移 (コンテスト結果による戦略変更　)

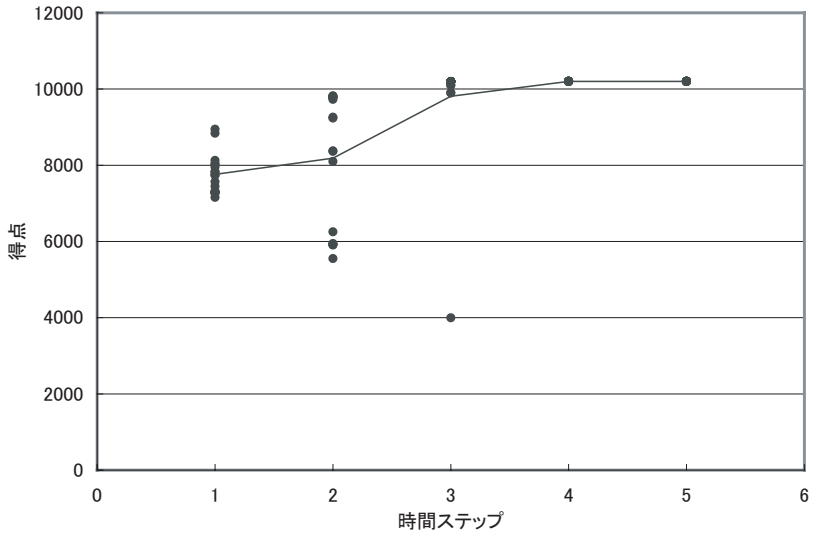


図 7.44: 戦略模倣シミュレーション: 各プレイヤーの得点と平均得点の推移 (コンテスト結果による戦略変更　)

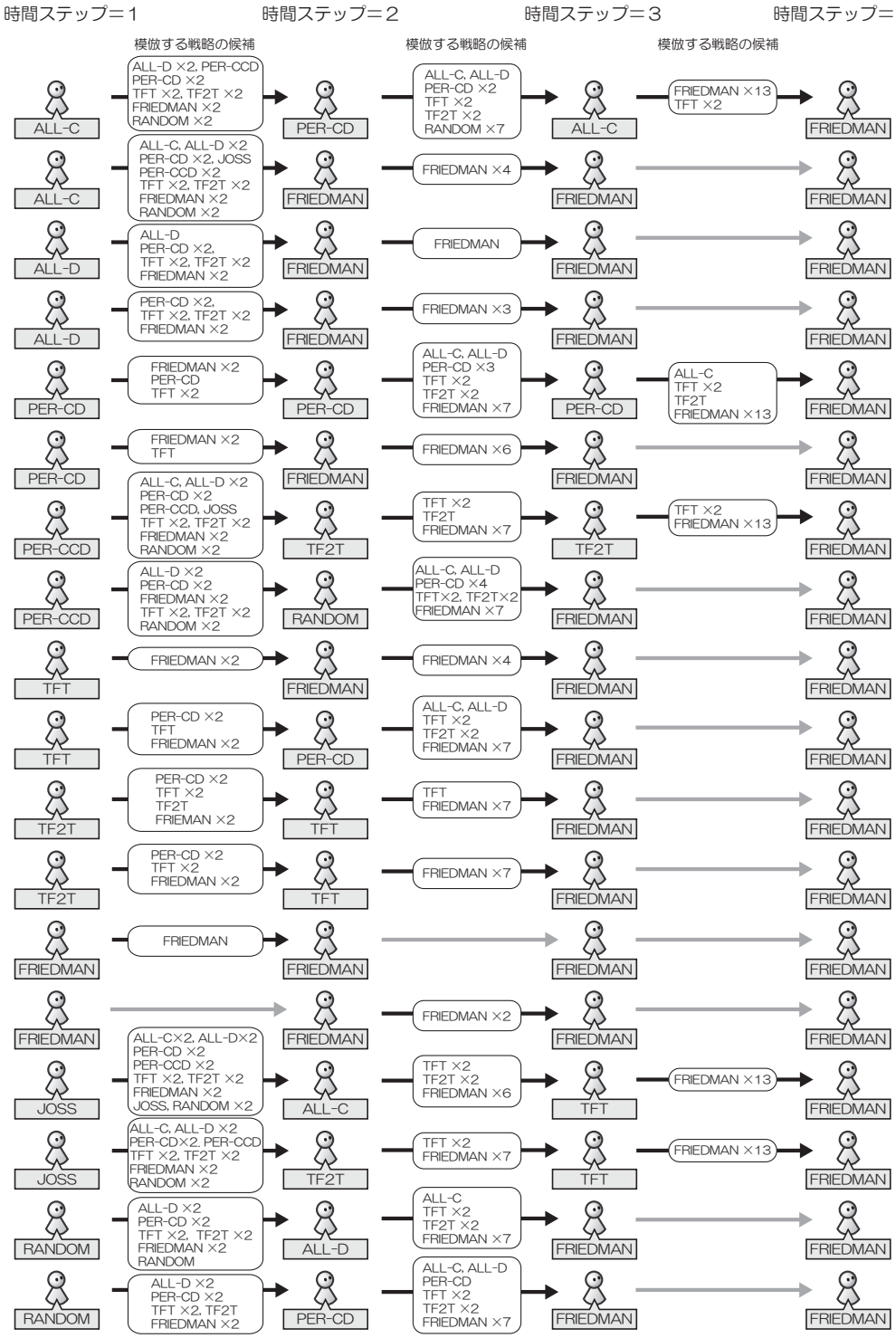


図 7.45: 戦略模倣シミュレーション: プレイヤーの戦略の変化 (コンテスト結果による戦略変更　)

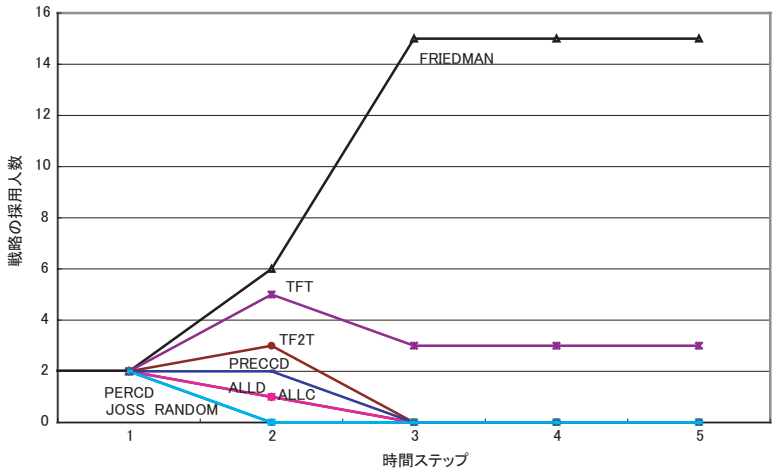


図 7.46: 戦略模倣シミュレーション: 各戦略を採用しているプレイヤー数の推移 (コンテスト結果による戦略変更　)

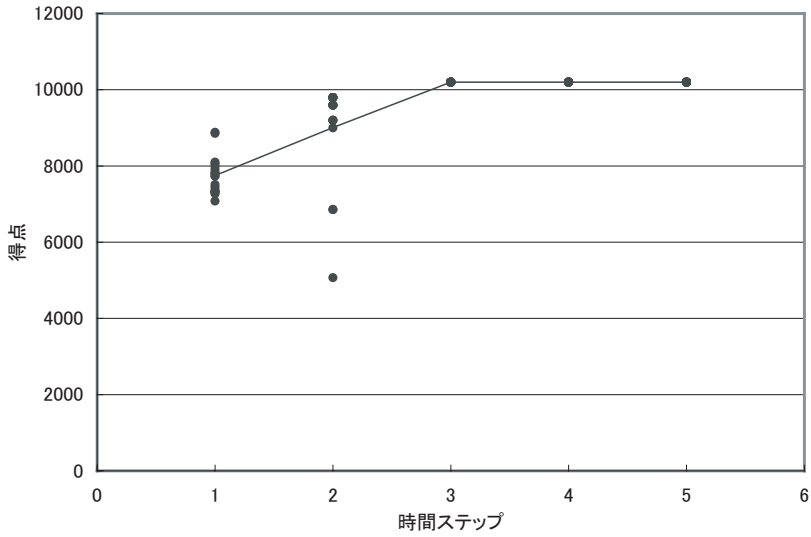


図 7.47: 戦略模倣シミュレーション: 各プレイヤーの得点と平均得点の推移 (コンテスト結果による戦略変更　)

7.3 貨幣の自生と自壊モデル

次に取り上げるモデルは、安冨 (2000) の貨幣の自生と自壊モデルである。このモデルは、主体全員が生産者かつ消費者である社会において、物々交換している商品のひとつが、ある時から貨幣としての役割を担うようになるという興味深いモデルである。また、同じモデルにおいて、貨幣の自生という事実への各主体の適応行動が、今度は逆にその貨幣が崩壊させてしまうということも観察されている。

安冨 (2000) によると、貨幣経済は、人間が単独で自給自足することができず、生存・欲求充足のために他人とのコミュニケーションが必要となる分業社会において発生しうる。原始的な分業社会における人びとは、自分の生産する財を、他人が生産しかつ自分が欲する財と物々交換することで、自分の生存・欲求を充足させている。このような物々交換取引の際に、人びとが「自分の欲求する財しか受け取らない」のであれば、欲望の二重の一致の困難が生じるため、取引がきわめて成立しにくい。しかし、実際は人間はこのように近視眼的ではなく、将来予測を行うことができる。それでは、人びとが将来の交換の有便性を考慮し、「自分は欲求しないが、多くの人が需要する財も受け取る」という行動をとる場合には、どのようなことが起こるのだろうか。安冨 (2000) は、近視眼的な社会と将来予測を行う社会の 2 つのケースについて、シミュレーションによる分析を行っている。その結果、将来予測を行う社会において、ある商品の交換可能性が極端に高くなり、「貨幣的商品」が発生することが観察されている。

ここでは、安冨 (2000) における「物々交換モデル」、「貨幣的交換モデル」、「進化的モデル」の 3 つのシミュレーションを再現することにしたい。

7.3.1 物々交換モデル

物々交換モデルに登場するエージェントは、生産者であり消費者である主体 1 種類だけであり、これを Agent エージェントとする (図 7.48)。このシミュレーションの流れは次のようになる (図 7.49, 7.50)。

まず、Agent エージェントが TimeEvent を受信する。どの順番で Agent エージェントが TimeEvent を受信するのかは、ランダムになっている。TimeEvent が奇数回目のときには、Agent エージェントは、SearchBehavior(図 7.51; 厳密には SearchBehavior を特化した MaximumSearchBehavior) で、他のすべての Agent エージェントに、どのような財を所有しているのかを質問する。他の Agent エージェントは、それぞれ RespondToSearchBehavior (図 7.52) で質問を受けると、現在所有している財のリストを返答する。Agent エージェントは、MaximumSearchBehavior で返答を集めていき、返答をすべて受け取った後に、自分の欲求する財を最もたくさんもっている Agent エージェントを選び、その Agent エージェントと物々交換を行うために自分の

DecideTradeBehavior (図 7.53) に連絡する。

Agent エージェントは、DecideTradeBehavior で取引相手が誰なのかという情報を受け取ると、お互いが実際に交換できる財を特定するために、取引相手に自分のもっている財のリストを知らせる。取引相手となった Agent エージェントは、RespondToDecideTradeBehavior (図 7.54) で財のリストを受け取ると、自分の欲求する財から、相手の所有財リストから自分の需要する財のリストを作成して、自分の所有財リストと一緒に返答する。相手の需要する財のリストと、所有財リストを受け取った Agent エージェントは、同様に自分の需要する財のリストを作成して、相手の需要する財のリストと照合する。この時、相手が何も需要していなければ、実際には何も交換しない。

相手が需要している場合には、お互いに渡す財の量が同じになるように、実際に交換する財のリストを作成し、お互いの ExchangeBehavior (図 7.55) に知らせる。Agent エージェントは、ExchangeBehavior で実際に交換する財のリストを受け取ると、お互いの RespondToExchangeBehavior (図 7.56) に財を渡す。この時、渡した個数に応じた運送コストを効用から減少させる。RespondToExchangeBehavior で財を受け取った Agent エージェントは、その財を自分の所有財とする。

一連の物々交換の過程が終了した後に、Agent エージェントは DecideTradeBehavior からお互いの ConsumeAndProduceBehavior (図 7.57) に対して、財を消費して効用を高め、欲求を変更するよう連絡する。ConsumeAndProduceBehavior で連絡を受け取ったお互いの Agent エージェントは、もし欲求する財をもっていればそれをすべて消費して、個数分だけ効用を高め、必ず欲求する財を変更する。もし欲求する財をもっていなければ、何も消費せず、効用も高めないが、一定の低い確率で欲求する財を変更する。

Agent エージェントは、偶数回目の TimeEvent を受け取ったときに、ResetUtilityBehavior (図 7.58) によって自分の効用得点を 0 にリセットする。

シミュレーションの結果は、図 7.59 のようになった。この図からわかるように、自分の欲求する財しか受け取らない主体によって構成される社会では、欲望の二重の一致の困難によって、自分の欲求する財を入手・消費するのは容易ではない (図 7.60)。

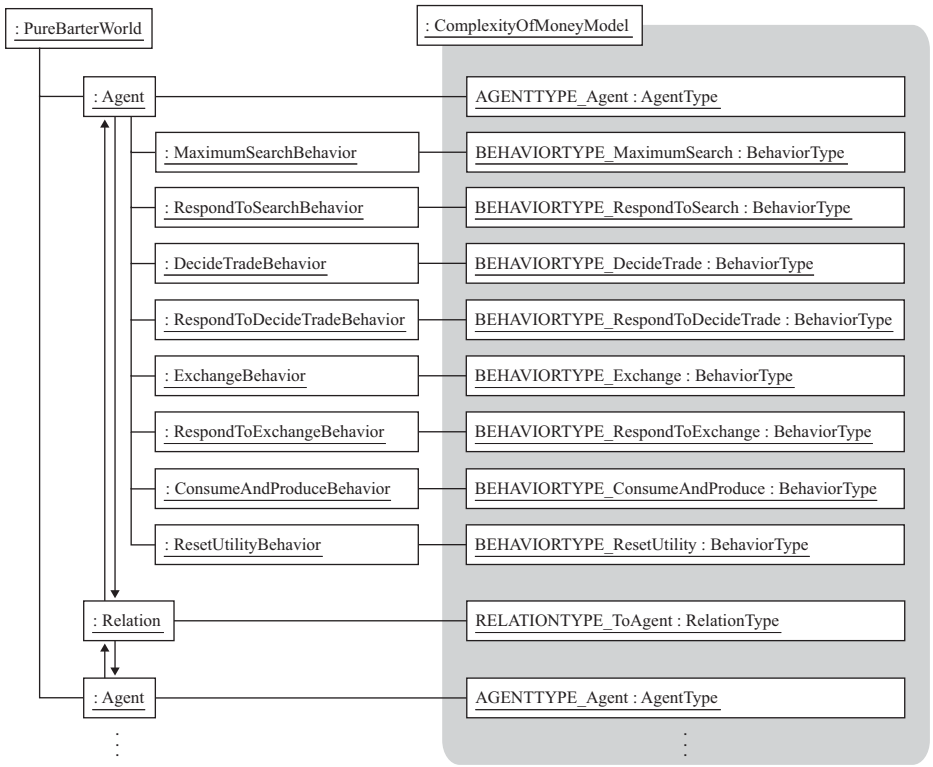


図 7.48: 物々交換モデルの全体像

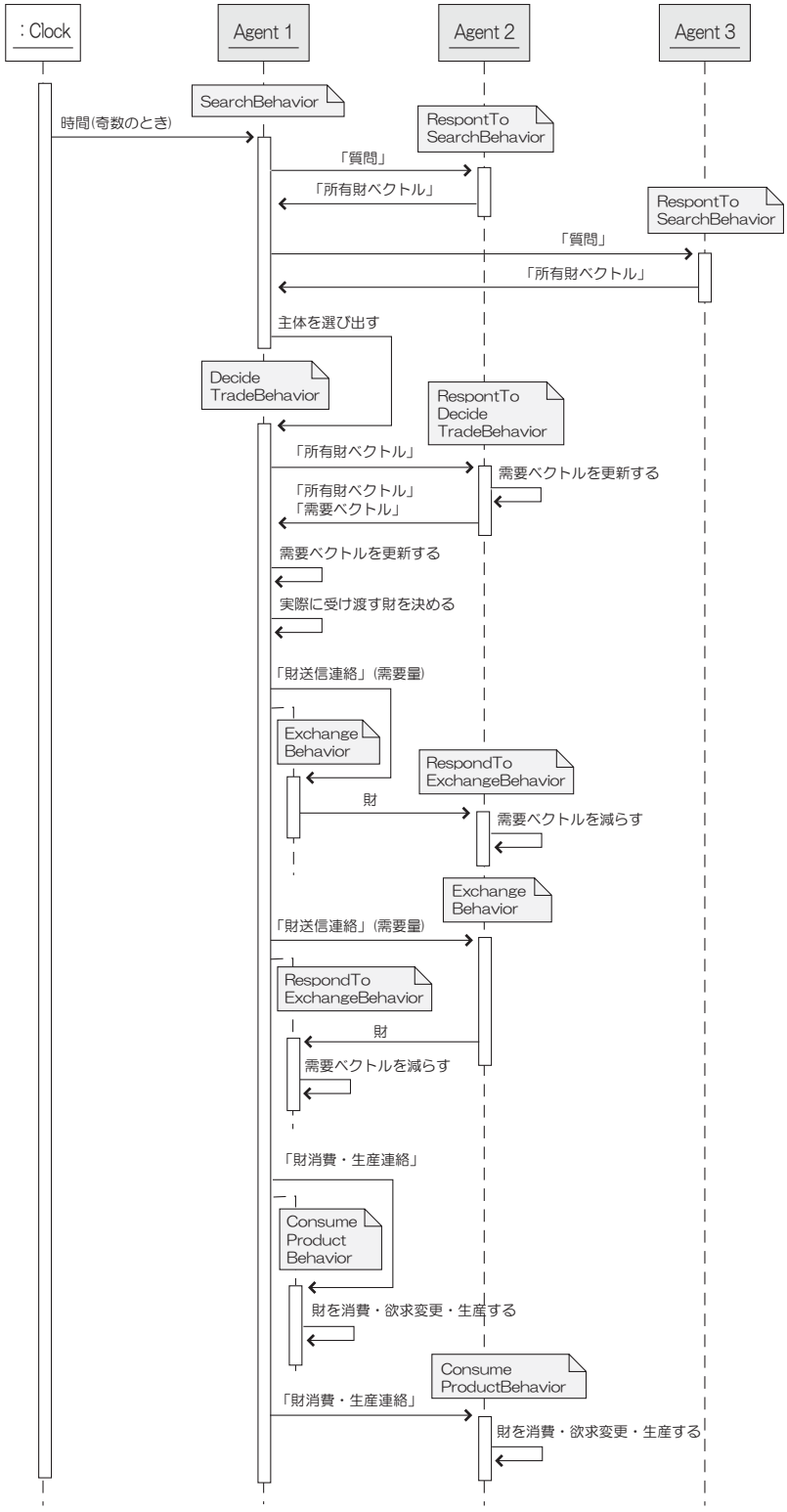


図 7.49: 物々交換モデル: TimeEvent(奇数) のときのシーケンス図

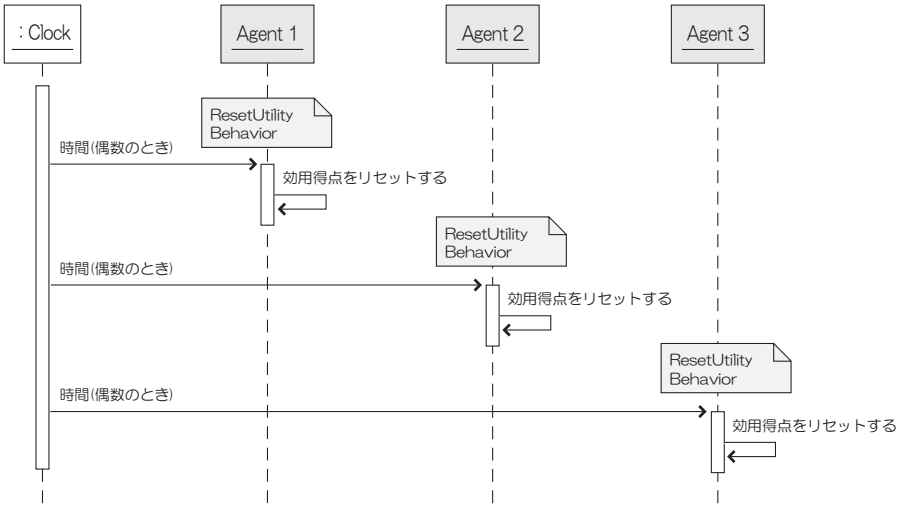


図 7.50: 物々交換モデル: TimeEvent(偶数) のときのシーケンス図

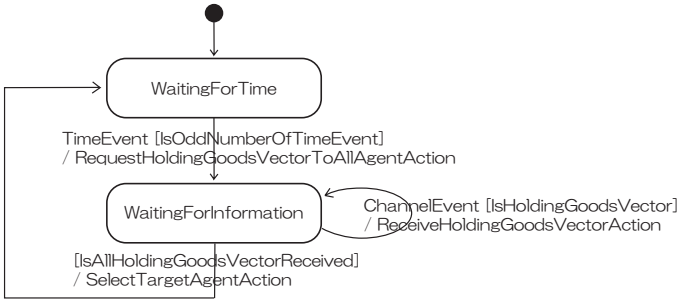


図 7.51: 物々交換モデル: SearchBehavior

図 7.52: 物々交換モデル: RespondToSearchBehavior

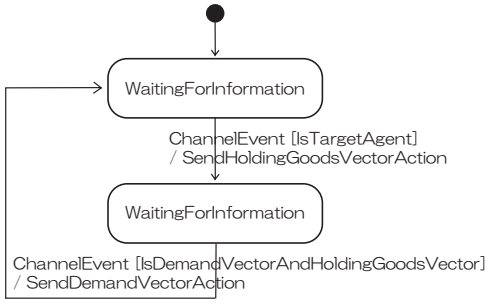


図 7.53: 物々交換モデル: DecideTradeBehavior

図 7.54: 物々交換モデル: RespondToDecideBehavior

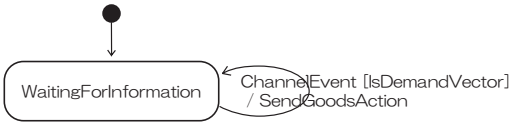


図 7.55: 物々交換モデル: ExchangeBehavior

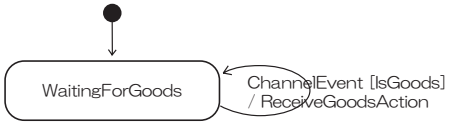


図 7.56: 物々交換モデル: RespondToExchangeBehavior

図 7.57: 物々交換モデル: ConsumeProductBehavior

図 7.58: 物々交換モデル: ResetUtilityBehavior

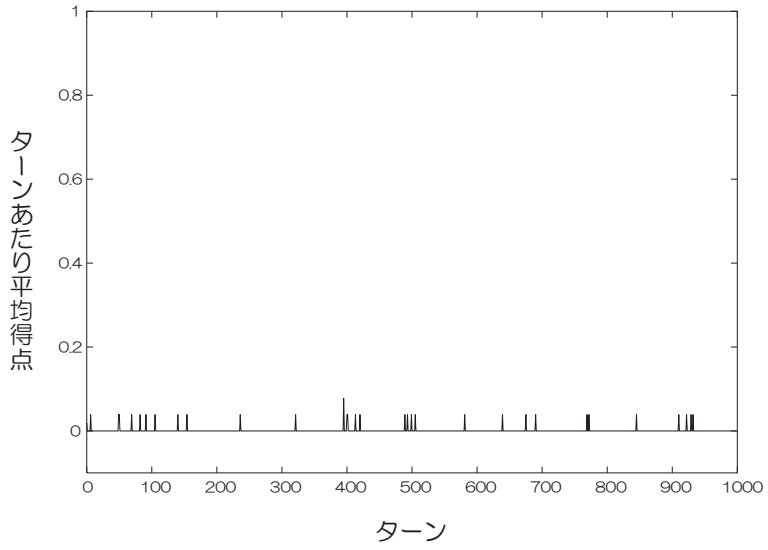


図 7.59: 物々交換モデル: 各ターンごとの得点の推移 (N=50, Threshold=0.078)

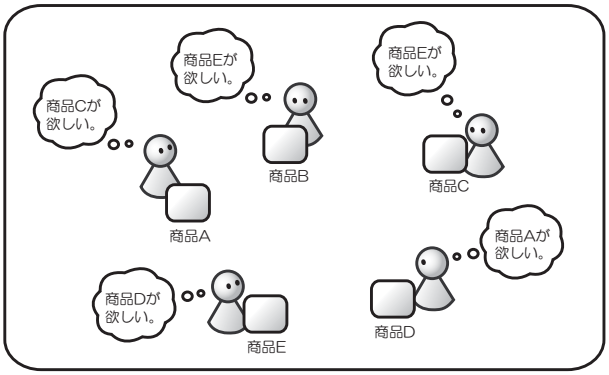


図 7.60: 物々交換モデルで起こっていることのイメージ (欲望の二重の一致の困難)

7.3.2　貨幣的交換モデル

貨幣的交換モデルでは、「将来の交換の簡便を考えて、たとえ自分が欲求してなくても、他人が需要するものを需要するようにする」という戦略を各主体にもたせる (図 7.61)。モデルでは、実際に交換する財を特定する前に知識交換を行うようにするために、DecideTradeBehavior を ExtendedDecideTradeBehavior に拡張し、ChangeKnowledgeBehavior と RespondToChangeKnowledgeBehavior を、Agent エージェントにもたせる (図 7.62)。また、各 Agent エージェントに「商品に対する見解」と「閾値」(0 から 1 までの実数) の情報をもたせ、商品に対する評価が「閾値」以上であれば、自分が欲求していなくてもその商品を受け取るようにさせる。この貨幣的交換モデルでは、すべての Agent エージェントは同じ閾値をもっており、閾値の値も初期設定のまま変化しないとする[68]。

シミュレーションは、次のような流れになる (図 7.63)。Agent エージェントは、ExtendedDecideTradeBehavior で取引相手が誰なのかを知り、知識交換を行うように自分の ChangeKnowledgeBehavior (図 7.64) に連絡する。連絡をうけた ChangeKnowledgeBehavior は、商品に対する見解を取引相手の Agent エージェントに伝える。取引相手の Agent エージェントは、RespondToChangeKnowledgeBehavior (図 7.65) で見解を受け取り、自分の見解に加算・規格化し、新しい見解を返答する。返答を受け取った Agent エージェントは、新しい見解の数値を記憶する。

物々交換モデルと異なり、貨幣的交換モデルでは、相手の所有する財のリストから、自分の欲求する財のみならず、自分が欲しくなくても他人が需要しているものがあれば需要する。ある商品に対して需要を表明するか否かは、見解の数値が Agent エージェントのもつ閾値を超えているか否かで判断する。こうして得られたお互いの需要する財のリストを照合し、実際に交換できる財のリストを作成する処理以降の流れは、物々交換モデルと同じである。

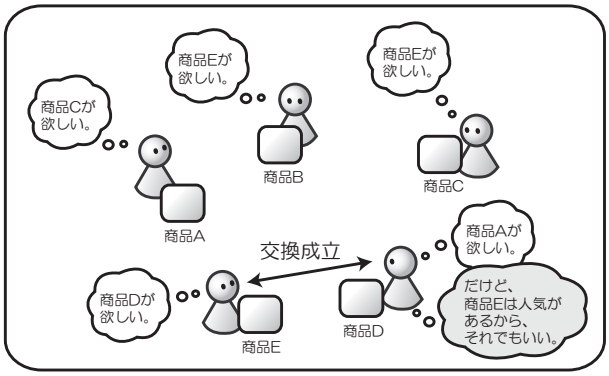


図 7.61: 貨幣的交換モデルのイメージ (人気のある商品の需要)

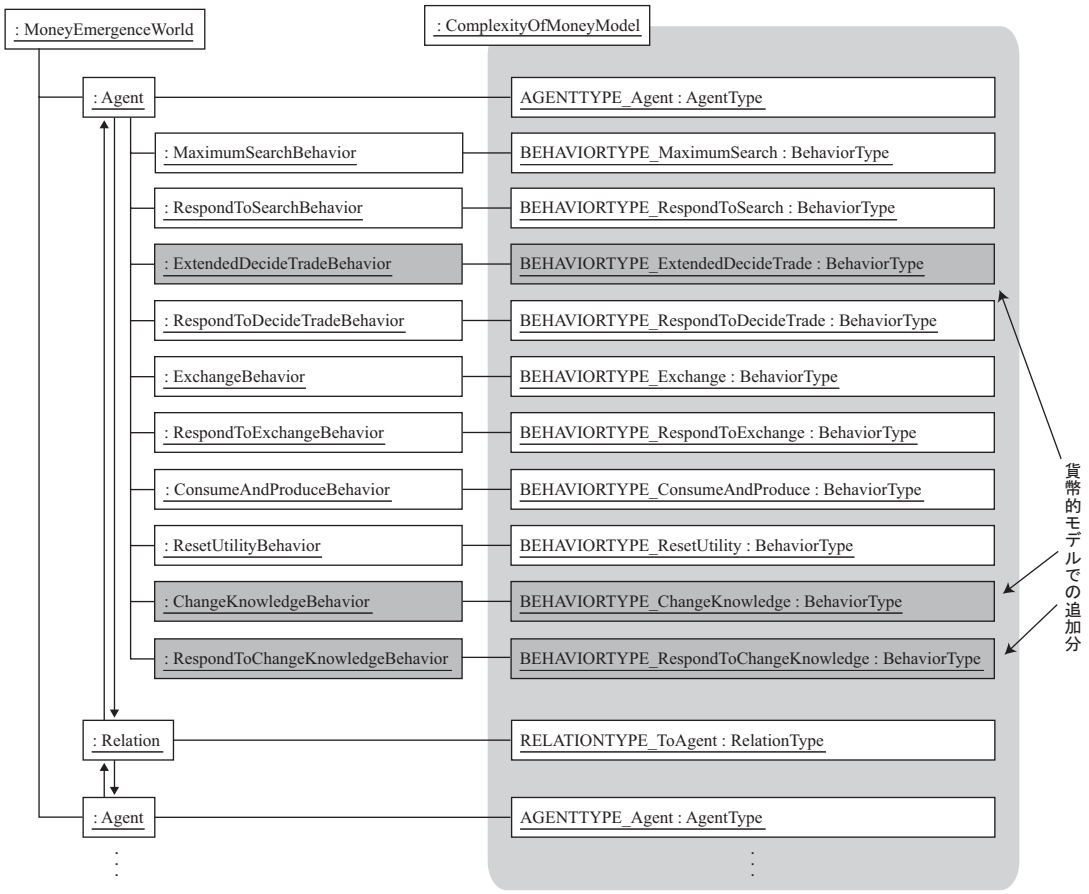


図 7.62: 貨幣的交換モデルの全体像

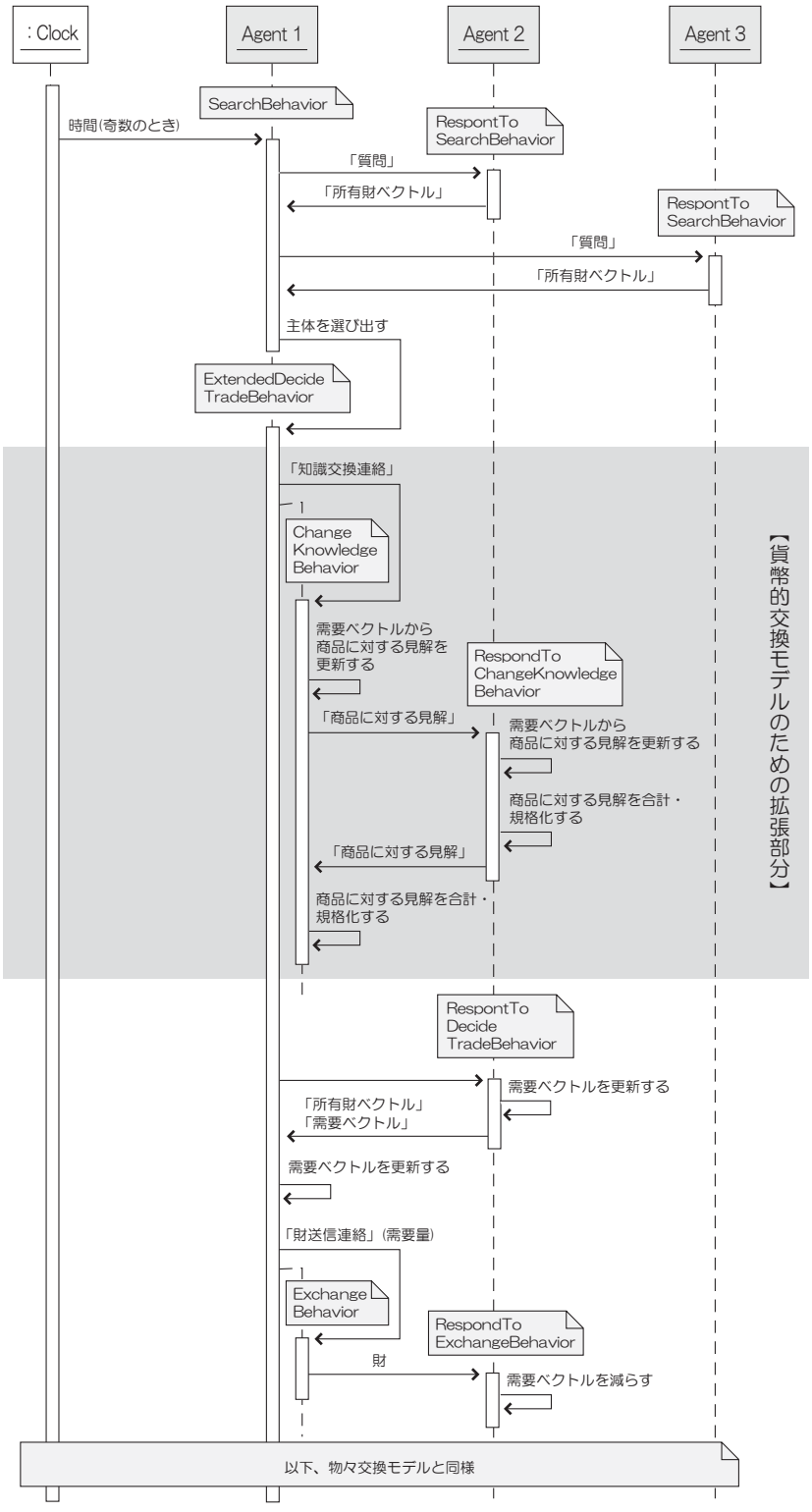


図 7.63: 貨幣的交換モデル: TimeEvent(奇数) のときのシーケンス図の一部

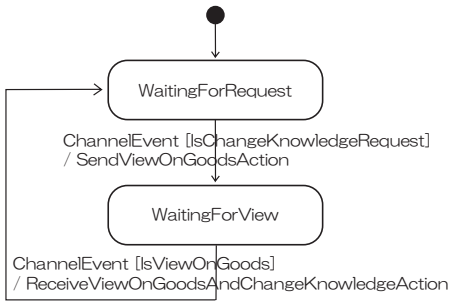


図 7.64: 貨幣的交換モデル: ChangeKnowledgeBehavior

図 7.65: 貨幣的交換モデル: RespondToChangeKnowledgeBehavior

シミュレーション結果は、図 7.66, 7.67, 7.68 のようになる。図 7.67 と図 7.66 からわかるように、最も市場性の高い商品の市場性が急激に高くなったと同時に、交換のために保有されている最も市場性の高い商品の単位数も増加し、それ以外の商品は交換の媒介として所有されなくなっている。このことから、「自分が欲してなくても他人の受け取るものであれば自分も受け取る」という戦略をもつ主体によって構成される社会では、ある時点において、最も市場性の高い商品が交換の媒介として認識されるようになり、それ以外のものを交換の媒介と見なすことがなくなるということがわかる。また、図 7.68 のように、貨幣的商品が生まれたことによって、主体の効用得点の平均値は明らかに高くなっている。貨幣的商品を媒介とすることによって、主体は本来自分の欲する財を、より効率的に入手・消費できるようになっているからである。

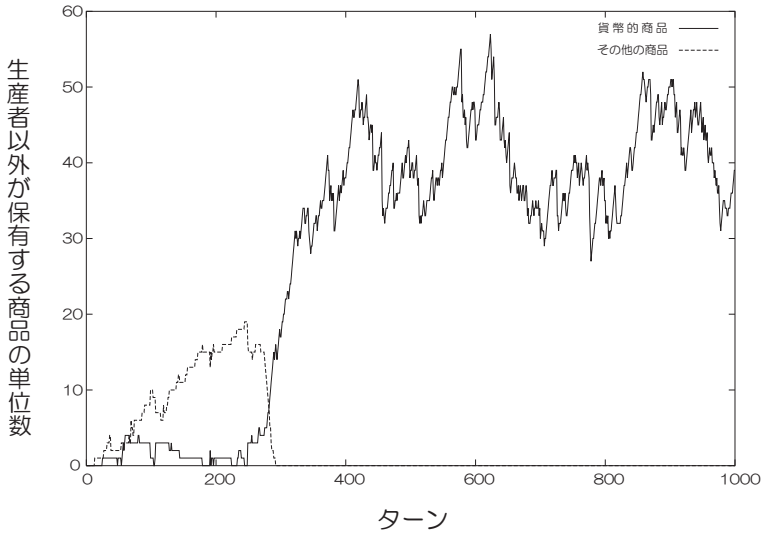


図 7.66: 貨幣的交換モデル: 交換のために保有されている商品の単位数の推移 (N=50, Threshold=0.078)

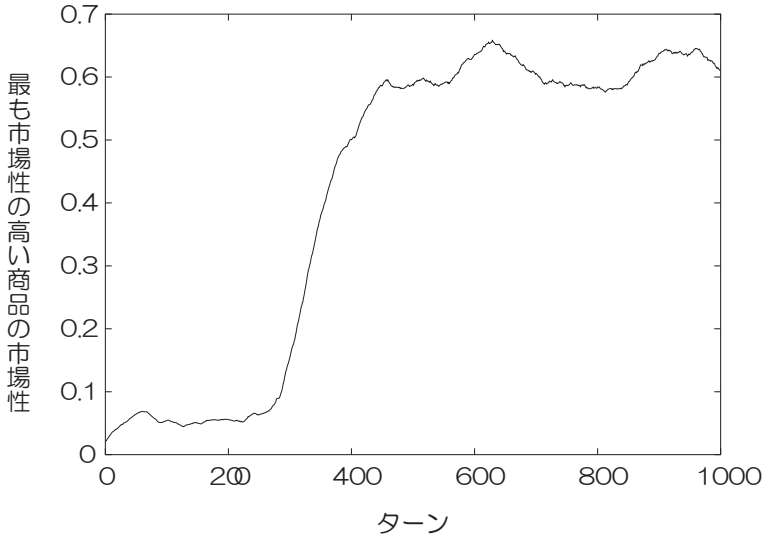


図 7.67: 貨幣的交換モデル: 最も市場性の高い商品の市場性の推移 (N=50, Threshold=0.078)

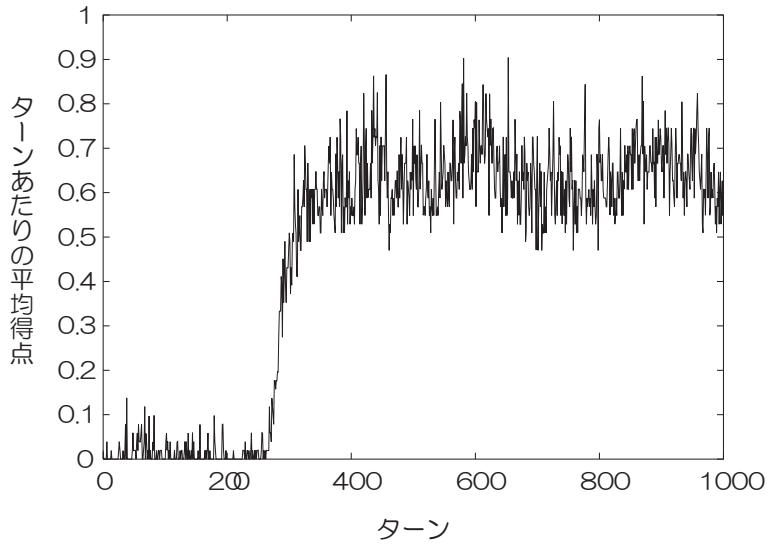


図 7.68: 貨幣的交換モデル: 各ターンごとの得点の推移 (N=50, Threshold=0.078)

7.3.3　進化的モデル

進化的モデルでは、「欲求する財を入手しやすい閾値をもつ Agent エージェントの閾値を、他の Agent エージェントが模倣する」という行動をモデルに導入する。具体的には、一日の最後に、得点の低い三人が得点の高い三人の閾値を模倣し、全員の閾値にノイズをかけるということを行う。

このことを実現するために、閾値の模倣を仲介する EvolutionFunction エージェントをモデルに追加する (図 7.69)。この EvolutionFunction エージェントが、2024 ステップ(69)に 1 回、効用得点が高い Agent エージェント 3 人の閾値を、低い Agent エージェント 3 人に模倣させる。

このシミュレーションの流れは、次のようになる (図 7.70)。EvolutionFunction エージェントは、ChangeThresholdBehavior (図 7.71) で TimeEvent を受け取ると、2048 回に 1 回、すべての Agent エージェントに対して、効用得点と閾値を尋ねる。RespondToChangeThresholdBehavior で質問を受け取った Agent エージェントは、過去 2048 ステップの効用得点の合計値と現在の閾値を答える (図 7.72)。すべての Agent エージェントから返答をうけとった EvolutionFunction エージェントは、得点の高い Agent エージェント 3 人と得点の低い Agent エージェント 3 人を調べ、得点の高い方の Agent エージェントの閾値を低い方の Agent エージェントに教える。得点の低い Agent エージェント 3 人は、RespondToChangeThresholdBehavior で閾値を受け取ると、それぞれの閾値を更新する。

以上の処理の後、EvolutionFunction エージェントは、すべての Agent エージェントに、閾値にノイズをかけるよう連絡する。Agent エージェントは、RespondToThresholdBehavior で連絡を受け取ると、自分の閾値に平均 0、分散 0.00005 のノイズをかける。この時、ノイズによって閾値が 1 を超える、あるいは 0 未満になる場合は、差分だけ跳ね返す。例えば、1. 02 になってしまう場合は、$1 - (1.02 - 1) = 0.98$ とする。

なお、この進化的モデルでは、過去 2048 ステップの効用得点の合計値を手に入れるため、以前のモデルでは偶数回目の TimeEvent で呼び出されていた ResetUtilityBehavior が、2048 回に 1 回呼び出されるように変更する。

このシミュレーション結果は、図 7.73, 7.74 のようになる。およそ 58 日目に商品 0 (GOODSTYPE_Goods0) が貨幣として選ばれ、310 日目まで用いられるが、その後突然崩壊してしまう。次に商品 39 (GOODSTYPE_Goods39) が貨幣となるが、この商品は 20 日間貨幣であり続けた後に崩壊し、その座を商品 15 (GOODSTYPE_Goods15) に受け渡している。図 7.74 を見ると、最初急激に閾値が落ちている。これは、貨幣的商品がまだ生まれていない社会では、閾値の低い主体のほうがより効率よく自分の欲する商品を入手・消費できることを表している。

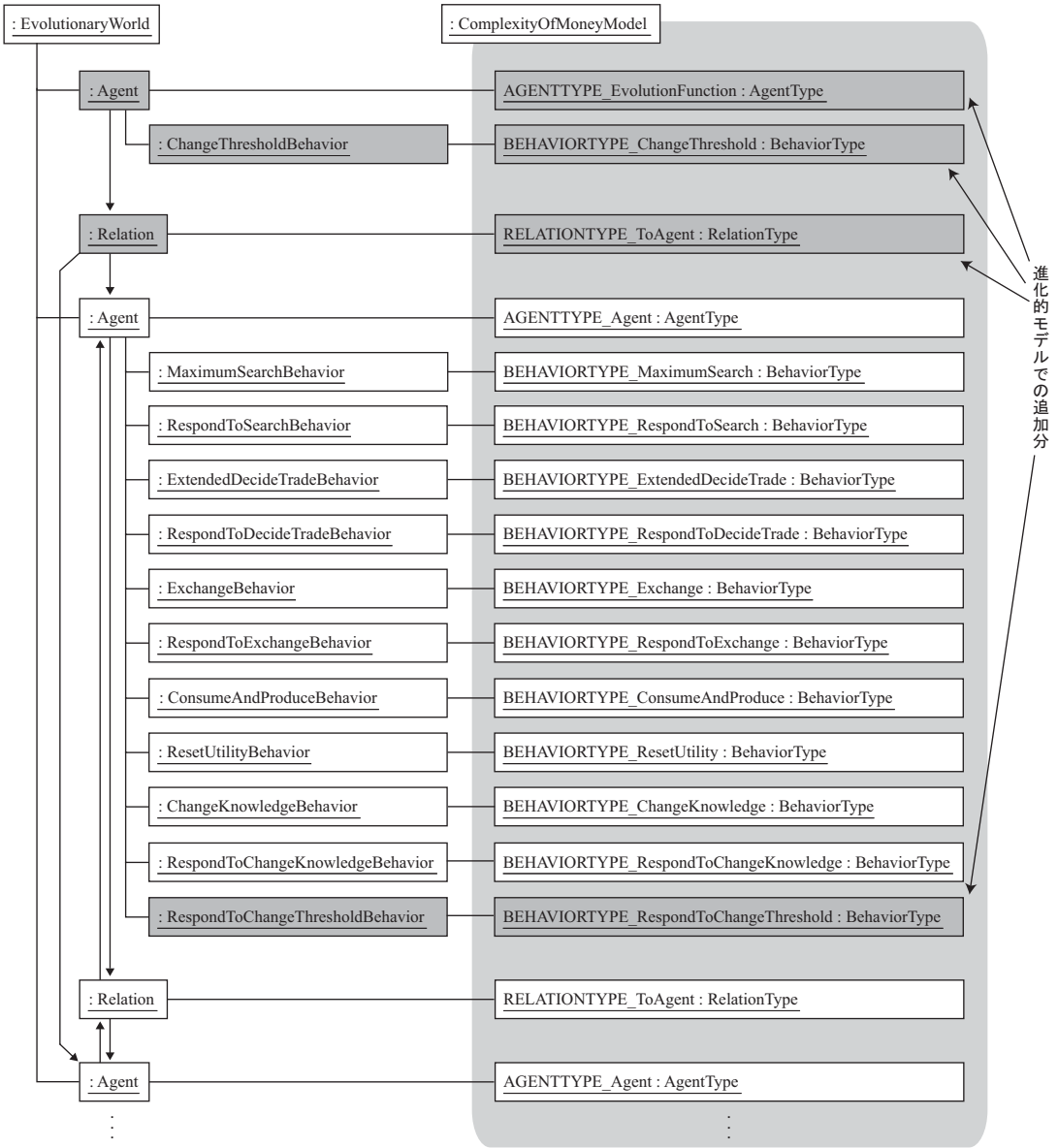


図 7.69: 進化的モデルの全体像

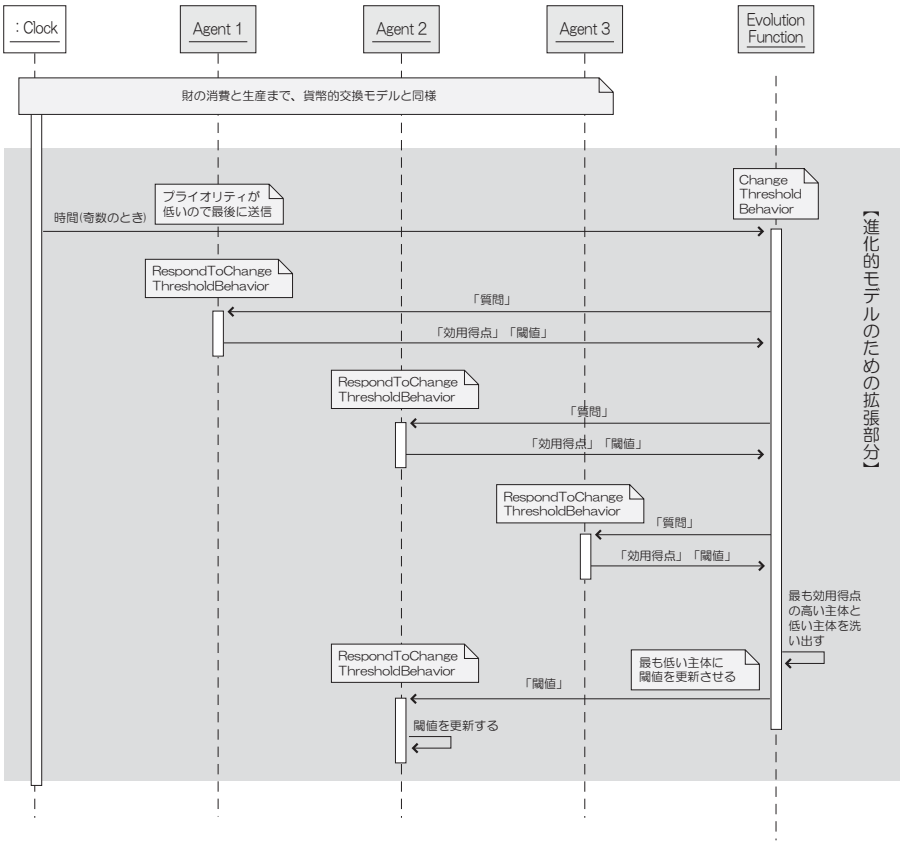


図 7.70: 進化的モデル: TimeEvent(奇数) のときのシーケンス図の一部

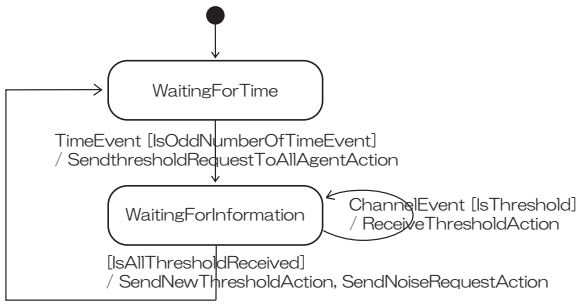


図 7.71: 進化的モデル: ChangeThresholdBehavior

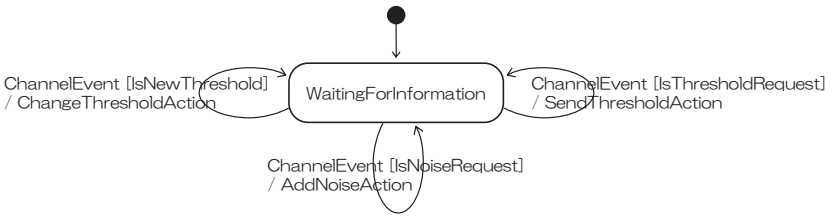


図 7.72: 進化的モデル: RespondToChangeThresholdBehavior

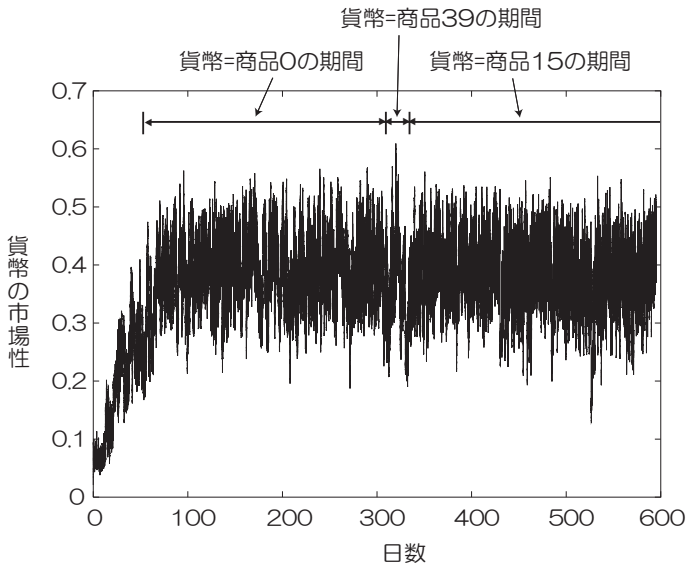


図 7.73: 進化的モデル: 貨幣の市場性の推移

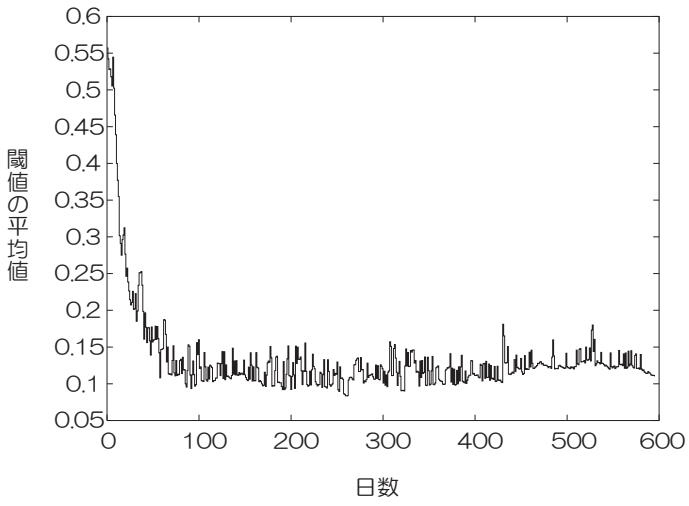


図 7.74: 進化的モデル:閾値の平均値の推移

7.4 SugarScape モデル

Sugarscape は、Epstein and Axtell (1996) によって提案・分析が行われた人工社会モデルである。2 次元セル空間の一部に、時間とともに再生する砂糖が配置されており、その砂糖をエージェントが取得する。この Sugarscape を直接的に表現するならば、エージェントは、環境 (空間に配置された砂糖) と相互作用することになるが、BEFM では、このようなモデルをそのまま記述することはできない。なぜなら、空間に (どのエージェントにも所有されていない) Goods を配置することや、その Goods が自動的に変化することを表現できないからである。この制約が生じるのは、BEFM が「時間経過とともに変化するものは、エージェントとしてモデル化する」という方針を採っているためである。

空間に配置された砂糖が時間とともに再生することを表現するためには、次の二つの実現方法が考えられる。第一の方法は、2 次元セル空間の全セルに、砂糖を生成・保持するエージェントを配置するというものである。第二の方法は、2 次元セル空間上の砂糖を管理する環境エージェントを作成するという方法である。ここでは、実装の容易さと実行負荷を考慮して、後者の方法によってモデル化することにする。

ここでは、Epstein and Axtell (1996) の中でも最も基本となる無限再生モデルを再現することにしたい。

7.4.1 Sugarscape モデル

Sugarscape モデルの全体像は図 7.75 のようになる。Epstein and Axtell (1996) における「エージェント」を Agent エージェントとし、砂糖の山などの環境を制御するエージェントを Environment エージェントとする。Agent エージェントは、2 次元セル空間上の 1 セルに存在し、時間とともに移動する。Environment エージェントは、セル上のどこにも存在しないが、Agent エージェントと関係を持ち、砂糖の取引を行うことができる。Agent エージェントが配置される 2 次元空間は、Space クラスを継承した CellSpace クラスを作成して表現する。モデルの初期設定で、エージェントが配置される格子状で端がトーラスとなっている Cell の空間を定義し、各 Cell に Sugar という財を配置する。CellSpace クラスを用いて、Cell クラスと Agent クラスを関連付けることによりエージェントのセルへの配置を表現する。また、この関連付けを変えることで、エージェントの移動を表現する。各セルの砂糖の量は、Environment エージェントのもつ FieldInformation によって保持される (図 7.76)。

このシミュレーションの流れは、次のようになる (図 7.77)。まず最初に TimeEvent を受け取るのは、Priority が高く設定されている Environment エージェントである。AddSugarBehavior (図 7.78) で TimeEvent を受け取った Environment エージェントは、あらかじめ設定されているターンあたりの Sugar の回復量に応じて、セル上に配

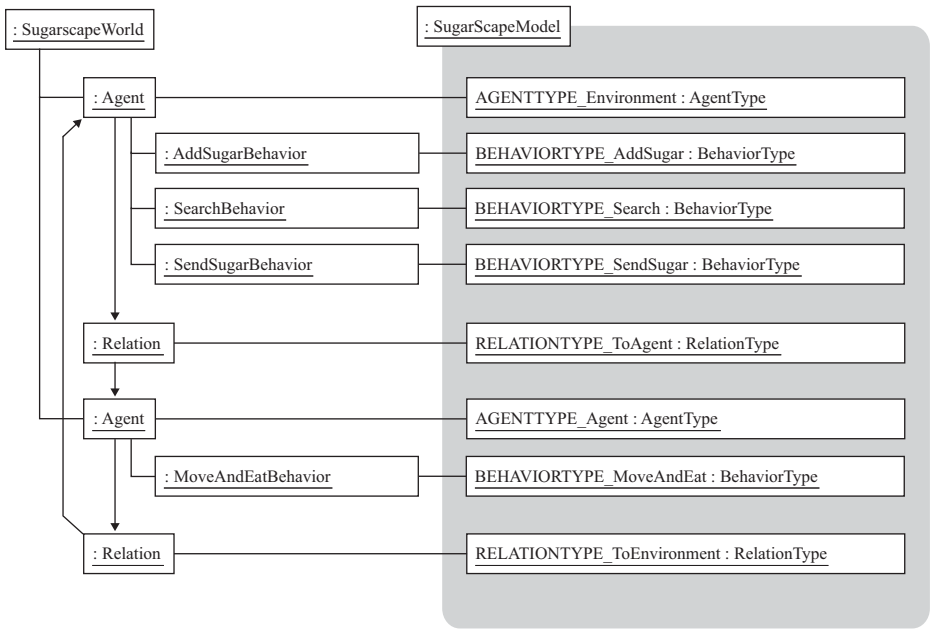


図 7.75: Sugarscape モデルの全体像

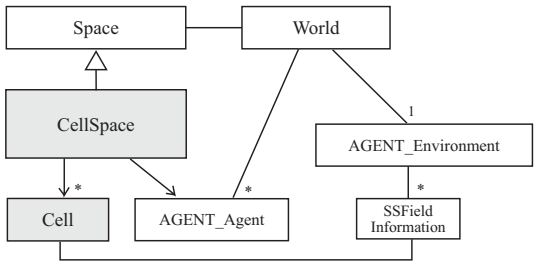


図 7.76: Sugarscape モデルのための CellSpace クラス

置された Sugar の量を回復させる。

次に TimeEvent を受け取るのは、Agent エージェントである。MoveAndEatBehavior (図 7.79) で TimeEvent を受け取った Agent エージェントは、自分の周囲の Cell に Sugar がどれだけあるのかを Environment エージェントに質問する。

Environment エージェントは、SearchBehavior (図 7.80) で質問を受け取り、Agent エージェントの視界に応じて、周囲の Cell の Sugar の配置状況を答える。Agent エージェントは、MoveAndEatBehavior で返答を受け取り、Sugar が最も多い Cell の方向へ移動し、移動先の Cell に配置された Sugar を Environment エージェントに要求する。Environment エージェントは要求を SendSugarBehavior (図 7.81) で受け取ると、Agent エージェントのいる Cell に配置されている Sugar をすべて渡す。Agent エージェントは、MoveAndEatBehavior で Sugar を受け取り、所有財としてもっておく。

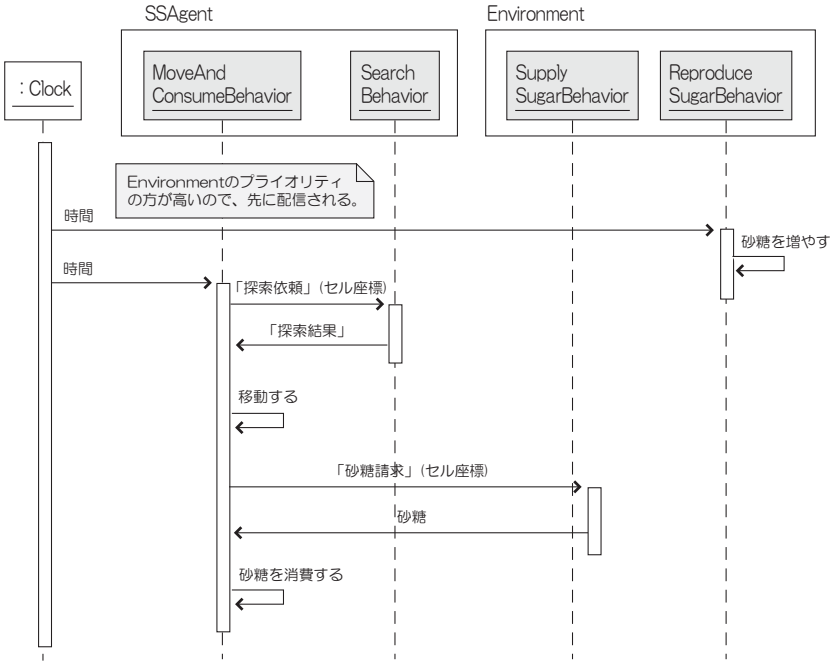


図 7.77: Sugarscape モデルのシーケンス図

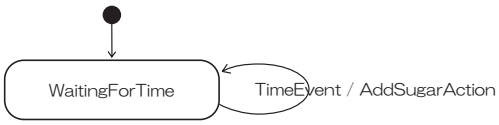


図 7.78: Sugarscape モデル: AddSugarBehavior

砂糖獲得後、Agent エージェントは、自分の代謝量に応じた Sugar を消費する。もし Sugar が足りなければ、Agent エージェントは死亡し、モデル上から削除される。

このシミュレーションの結果は、図 7.82 のようになる。最初、全体に散在していた Agent エージェントは、時間が経つにつれて砂糖の山に集まってくる。また、山から遠くにいた Agent エージェントは死亡してしまう。

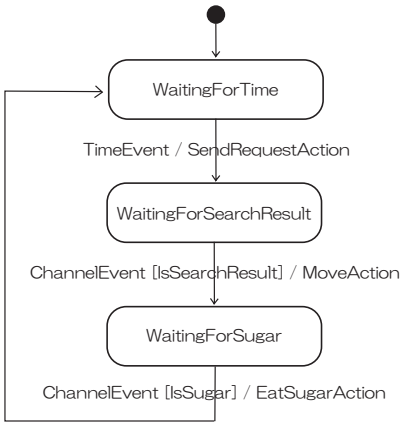


図 7.79: Sugarscape モデル: MoveAndEatBehavior

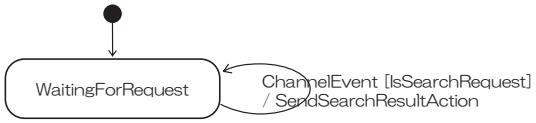


図 7.80: Sugarscape モデル: SearchBehavior

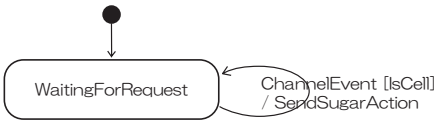


図 7.81: Sugarscape モデル: SendSugarBehavior

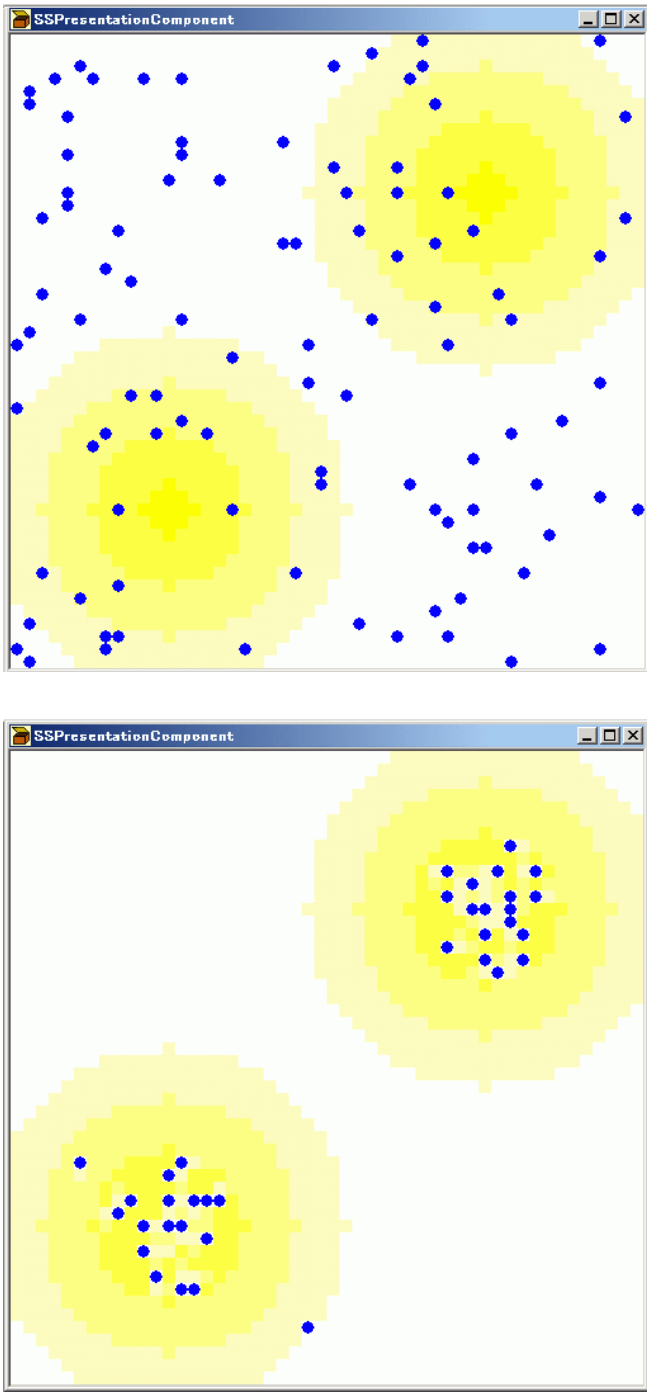


図 7.82: Sugarscape モデル: シミュレーション結果

7.5 人工株式市場モデル

最後に取り上げるモデルは、Arthur et al. (1996); Palmer et al. (1994) の人工株式市場モデルである。この人工株式市場では複数のトレーダーエージェントが、自分の戦略に従って株を売買し、その結果として株の価格などの市場全体の動きが決まる。そして結果に応じてエージェントは自分の戦略を適応的に変更していく。この研究では、「合理的期待論」に代替する新しい「進化経済学」のアプローチを展開しようとしている。このアプローチでは、得られる情報が不完全であり、かつ問題の文脈がわからない状況において、エージェントが学習しながら行動するというモデルを構成する。

ここでは、Information で表した戦略によって振舞いが決まるモデルの例として、Arthur et al. (1996) の人工株式市場を再現することにしたい。

7.5.1 人工株式市場モデル

このモデルの全体像は、図 7.83 のようになる。市場には 1 種類の株式があり、この取引を StockExchange エージェントが管理している。Trader エージェントは、各期において自分の資産を、安全資産と株式に資産配分を行うが、株式保有者には各期ごとに配当が与えられ、安全資産には各期ごとに利子がつく。これらを実現するために、Company エージェントと RiskFreeSecuritySupplier エージェントが存在する。また、市場の状況を調べて伝えるための Press エージェントがいる。

Trader エージェントは、それぞれがもつクラシファイアシステムによって予測・学習を行う。クラシファイアシステム (Holland, 1986; Goldberg, 1989) は、強化学習 (Sutton and Barto, 1998) の一種であるため、行動の有効度を伝える報酬 (強化信号) が必要となるだけで、最適な行動を直接的に指示するような教師信号を必要としない。そのため、エージェントが複雑な環境に自律的に適応するためのメカニズムとして注目されている。クラシファイアーシステムの条件部は市況を識別する。この市況は 12 桁の 2 進数で表される。

- 1-6：$Currentprice \times interestrate/dividend > 0.25, 0.5, 0.75, 0.875, 1.0, 1.125$
- 7-10：$Currentprice > MA(t,5), MA(t,10), MA(t,100), MA(t,500)$
- 11：always on 1
- 12：always on 0

但し、

$$MA(t,m) = \sum_{\tau=0}^{m-1} p(t-\tau)/m$$

である。

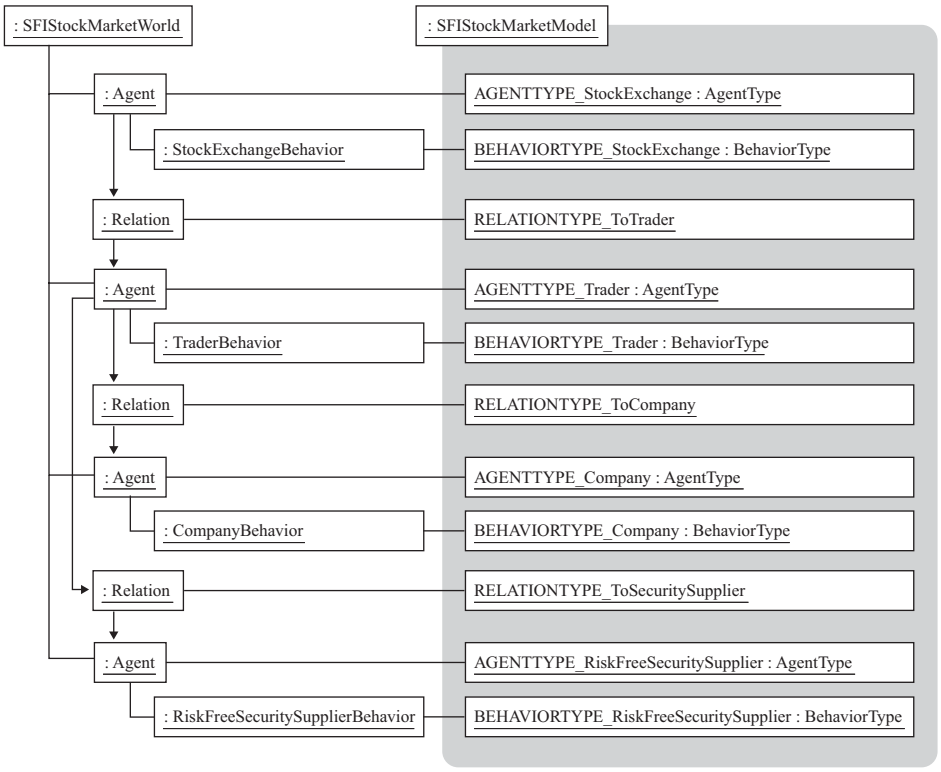


図 7.83: 人工株式市場モデルの全体像

クラシファイアーシステムの行動部は、その市場環境のもとでとるべき行動を示している。行動部には predictors（予測部）と呼ばれる整数の組 (a, b) が出力として出される。出された (a, b) を以下の線形予測式に代入することで、来期の予測を行うことになる。

$$E[p_{t+1} + d_{t+1}] = a(p_t + d_t) + b$$

多数の If Then ルールがあるとき、複数の If Then ルールが同時に起動する可能性が出てくる。その際は、If Then ルール毎に与えられている強度に比例した確率でルールを選択することになる。ルールの条件部と市場状況が合致するたびに、合致したすべてのルールの強度を更新することで、学習が行われる。

このシミュレーションの流れは、次のようになる (図 7.84)。まず最初に TimeEvent を受け取るのは、Priority が一番高く設定されている StockExchange エージェントである。StockExchange エージェントは、StockExchangeBehavior (図 7.85) によって、すべての Trader エージェントに売買注文を提示するように連絡する。Trader エージェントは、連絡を TraderBehavior (図 7.86) で受け取ると、クラシファイアにもとづいて、売買注文を StockExchange エージェントに返答する。StockExhange エージェントは、すべての Trader エージェントから注文を受け取ると、売り注文と買い注文の量から、

今期の株価と各 Trader エージェントが実際に取引できる量を計算し、その結果を各 Trader エージェントに連絡する。Trader エージェントは、その結果を TraderBehavior で受け取り、売り注文の場合は株を、買い注文の場合はお金を、StockExchange エージェントに支払う。StockExhange エージェントは、Trader エージェントから送られてきたすべてのお金と株を集計し、先ほどの結果に準じた対価を各 Trader エージェントに支払う。

取引が終わった後に、各 Trader エージェントは、RiskFreeSecuritySupplier エージェントに利子を要求する。RiskFreeSecuritySupplier エージェントは、RiskFreeSecuritySupplierBehavior(図 7.87) で要求を受け取ると、Trader エージェントの保有する安全資産の量に応じた利子を支払う。Trader エージェントは、TraderBehavior で利子を受け取る。

次に、各 Trader エージェントは、Company エージェントに配当金を要求する。Company エージェントは、CompanyBehavior(図 7.88) で要求を受け取ると、Trader エージェントの持ち株数に応じた配当金を支払う。Trader エージェントは TraderBehavior で配当金を受け取る。配当金を受け取った後、各 Trader エージェントは、複数の条件文が起動した場合の選ばれやすさを決める条件文の強度を更新する。

StockExchange エージェントの次に TimeEvent を受け取るのは、Priority が 2 番目に高く設定されている Company エージェントである。Company エージェントは、各 Trader エージェントに支払う配当金の額を切り替える。

最後に TimeEvent を受け取って行動するのは、Priority の最も低い Press エージェントである。TimeEvent を PressBehavior で受け取った Press エージェントは、StockExchange エージェントに過去の株価の推移について質問する。StockExchange エージェントは、質問を StockExhangeBehavior で受け取ると、過去の株価の推移について答え、Press エージェントはその返答を記憶する。次に、Company エージェントに過去の配当金の推移について質問する。Company エージェントは、質問を CompanyBehavior で受け取ると、過去の配当金の推移について答え、Press エージェントはその返答を記憶する。最後に、RiskFreeSecuritySupplier エージェントに安全資産の利息について質問する。RiskFreeSecuritySupplier エージェントは、質問を RiskFreeSecuritySupplierBehavior で受け取り、安全資産の利息について答え、Press エージェントはその返答を記憶する。

その後、Press エージェントは、現在の世界の状況をの 2 進数の文字列に変換し、それをすべての Trader エージェントに伝える。Trader エージェントは、ReceiveClassifierBehavior で連絡を受け取ると、それを現在の世界の状況として記憶しておく。

シミュレーションの結果は、図 7.89 のようになる。

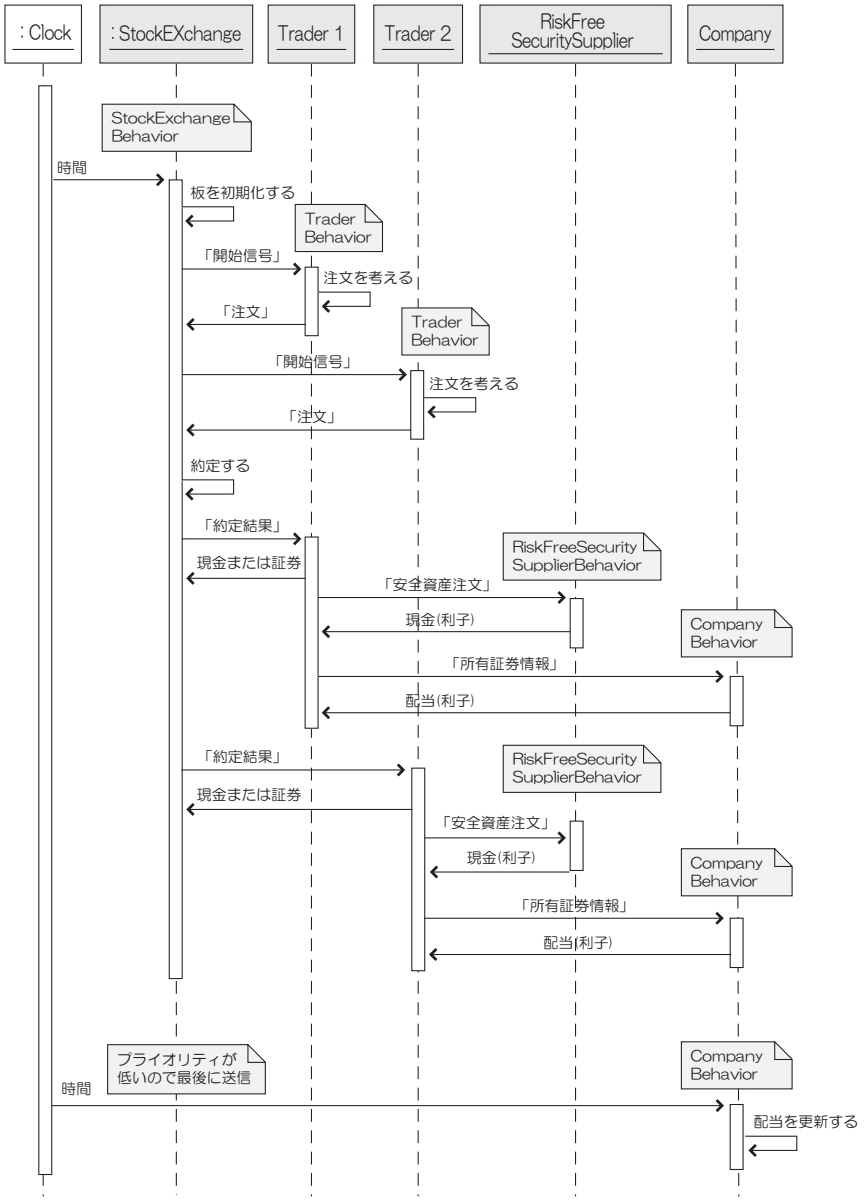


図 7.84: 人工株式市場モデルのシーケンス図

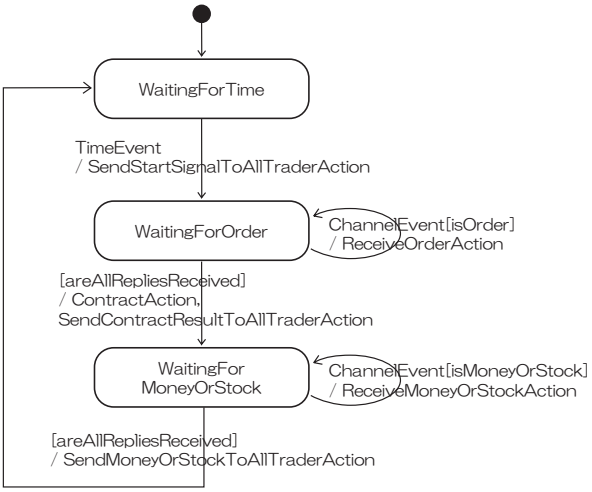


図 7.85: 人工株式市場モデル: StockExchangeBehavior

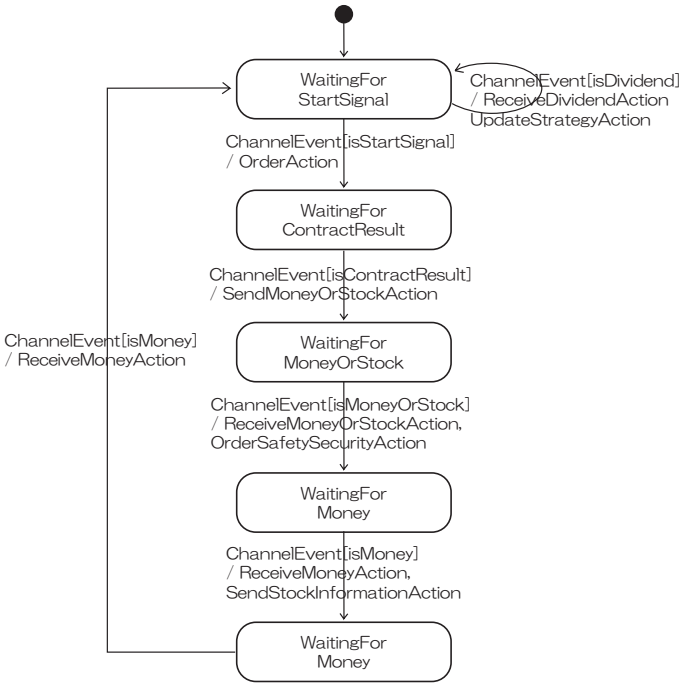


図 7.86: 人工株式市場モデル: TraderBehavior

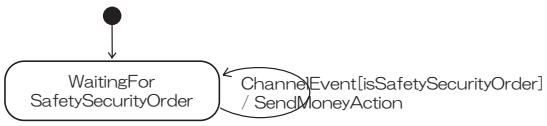


図 7.87: 人工株式市場モデル: RiskFreeSecuritySupplierBehavior

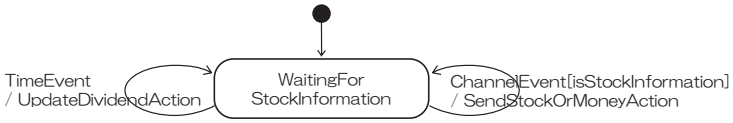


図 7.88: 人工株式市場モデル: CompanyBehavior

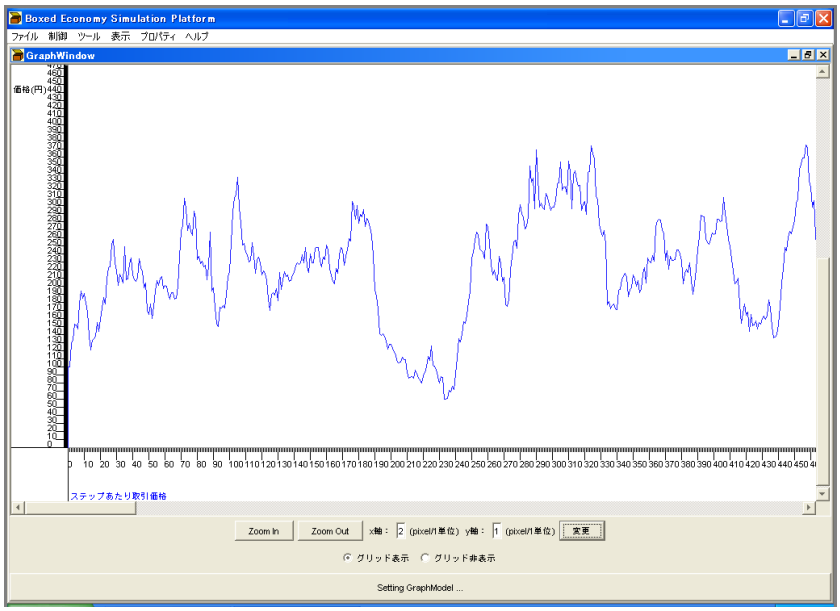


図 7.89: 人工株式市場モデル: シミュレーションの実行画面

第8章　提案システムによる事例研究

本章では、独自の適用事例として、規格競争のモデルを取り上げたい。具体的には、規格競争の典型といわれる家庭用 VCR の市場を、消費者の相互作用としてモデル化し、その振舞いを分析する。このモデルでは、マーケティング・サイエンスや消費者行動論などのモデルを用いてミクロレベルのモデル化を行うため、従来のマクロ集計的なネットワーク外部性モデルでは行うことができなかった分析が可能となる。

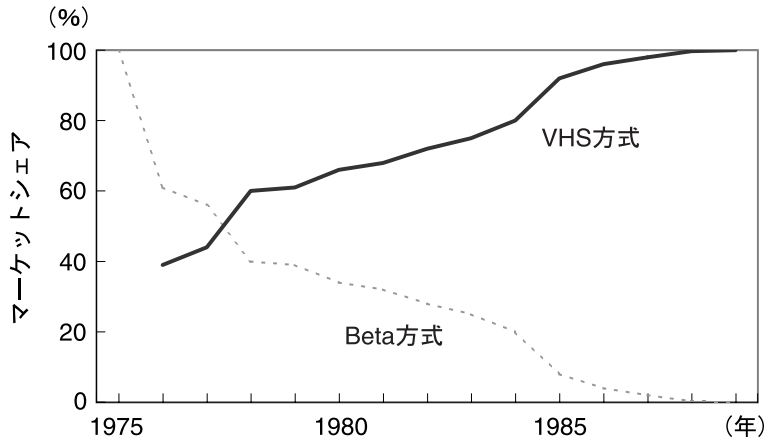
8.1　家庭用 VCR における規格競争

8.1.1　規格競争におけるネットワーク外部性の特徴

ある類似した機能を提供する製品において、複数の異なる規格が存在する場合に繰り広げられる企業間 (または企業グループ間) の競争のことを規格競争という。一般の製品とは異なり、規格競争では一つの規格が圧倒的なマーケットシェアを獲得するという「ウィナー・テイク・オール現象 (一人勝ち現象)」(Frank and Cook, 1998) が起こりやすい。そこには互換性という要因が引き起こす「ネットワーク外部性」(Katz and Shapiro, 1985) が存在するからである。

外部性とは、市場での取引の結果が第三者に影響を及ぼすことをいう経済学の概念であり、ネットワーク外部性とは、ネットワークの価値が参加人数によって決まるという外部性のことである (Katz and Shapiro, 1985)。ネットワーク外部性は必ずしも物理的なネットワークで結ばれている必要はなく、互換性のない規格やソフトウェアを媒介とした「見えないネットワーク」についても当てはまる。このような間接的なネットワーク外部性の典型的な例としては、家庭用 VCR における VHS 方式と Beta 方式、オーディオ機器における DCC 方式と MD 方式、パソコンのオペレーティングシステムにおける Mac OS と Windows などがある。

ネットワーク外部性が存在する場合、消費者がその製品から得る効用は、その消費者と製品との間で決まるのではなく、他の消費者の選択に依存して決まるという点に特徴がある。その結果、シェアが優勢になった規格がますますシェアを高めるというポジティブ・フィードバックがはたらくことになり、一人勝ち現象を生み出すのである (Arthur, 1994)。



(VHS 方式と Beta 方式のシェアのデータは累積の生産に関するものであり、文献 (Cusumano et al., 1992) より引用。そのソースは、1976–83 年に関しては日経ビジネス (1983 年 6 月 27 日号)、1981–83 年に関しては日本経済新聞 (1984 年 11 月 21 日)、1975 年と 1985–88 年に関しては JVC, Public Relations Dept.(Cusumano et al., 1992))

図 8.1: 日本における VHS 方式と Beta 方式のマーケットシェアの推移

8.1.2 取り上げる事例の概要と特徴

本研究では規格競争の具体的な事例として、日本における家庭用ビデオカセットレコーダー (VCR: Video Cassette Recorder[70]) の規格競争を取り上げる。家庭用 VCR は、1975 年に Beta 方式、1976 年に VHS 方式という異なる二つの方式が発売されて以降、多くの企業を巻き込んだ激しい規格競争が繰り広げられた製品である。Beta 方式は VHS 方式に比べて画像品質などの面で優れていると言われていたにもかかわらず[71]、結果的には VHS 方式が圧倒的なマーケットシェアを占めるに至っている (図 8.1)。

家庭用 VCR は一般に「タイムシフト利用」と「ビデオソフト利用」という二つの目的で用いられる (吉井, 2000)。タイムシフト利用とは、テレビの放送番組を録画し、後で再生することである。具体的には、留守中に録画しておく「留守録」や、テレビ番組を見ている最中に他局で放送されている別の番組を録画する「裏録」などがこれにあたり、時間に固定されているテレビ番組を時間的にシフトするために使用することである。基本的に自分で録画したものを後に再生することになるので、「製品利用の自己完結性」(山田, 1993) が高く、VHS 方式と Beta 方式のどちらでも構わないということになる。また、録画したものを家族や友人と貸し借りをすることも考えられるが、市場での取引や流通は起きないため、最低限身近な人との互換性が保たれてい

ればよいことになる。

一方、ビデオソフト再生とは、セルやレンタルのビデオソフトを再生するという使用方法である。市場で流通しているビデオソフトを購入または借用することになるので、自分の持っている家庭用 VCR と同じ方式のビデオソフトを入手する必要がある。そのため、消費者は市場に出回っているビデオソフトの方式を意識するようになり、結果としてそのハードウェアである家庭用 VCR のシェアに関心を持たざるをえなくなる (Katz and Shapiro, 1985)。

日本における家庭用 VCR の利用においては、普及の序盤ではタイムシフト利用がほとんどであったが、1980 年半ばに主に二つの要因がきっかけとなり、ビデオソフト再生が消費者の使用目的の中で重要な位置を占めるようになった。第一の要因は、セルビデオの普及である。セルビデオは 1970 年代から存在していたものの、高価であったため利用者は限られていた。しかし、1984 年にハリウッドの七大メジャー映画会社が直接販売のために日本法人を設立し、セルビデオの普及のための低価格路線を歩んだことなどから、ビデオソフトを購入するという消費行動が消費者のなかに定着した。第二の要因は、1983 年にレンタルビデオが正式に許可されたことである。これを受けて、後にみるように 1980 年代半ばから 1990 年頃にかけてレンタルビデオ店が急激に増加している。このような状況になると、家庭用 VCR の購入時の方式選択において、市場における各方式のソフト流通量が重視されるようになり、市場の動向が購入の意思決定に重要な影響を及ぼすことになったと考えられる(72)。

このような特徴をもつ規格競争は、「標準化」という観点から企業提携や経営戦略という供給側の観点から多くの議論がなされてきており (伊丹および伊丹研究室, 1989; 山田, 1993; 淺羽, 1995; 山田, 1997)、複雑系経済学においても頻繁に取り上げられている (Arthur, 1994)。しかし技術の選択は最終的には消費者に委ねられているため、その現象を理解するためには消費者の選択という需要側にも着目する必要がある。特に家庭用 VCR の事例においては、その使用形態が普及の途中で変化したことに伴って、方式選択に影響を及ぼす範囲が局所から大域へと変化していることから、視野が変化する需要側モデルを扱うことが求められる。本論文では、市場全体の大域的なマーケットシェアが効用を高めるという従来のネットワーク外部性の概念を拡張し、各消費者を取り巻く局所的なシェアも方式選択に影響を及ぼすというモデルを提案する。

8.2 概念モデル

8.2.1 全体像

本論文で提案する人工市場は N 人の消費者エージェントから構成されている(73)。消費者エージェントは、大域的状況を知るための情報源をもち、必要であれば各方式の大域的なマーケットシェアを知ることができるとする(74)。また各消費者エージェン

トは他の消費者エージェントと局所的な関係をもっている。消費者エージェント i と消費者エージェント j が関係をもつとき、その関係性の強さにより、エージェント間関係は $0 < R_{ij} \leqq 1$ の実数をとり、そうでないとき $R_{ij} = 0$ と表現する。

本論文では市場構造を、消費者エージェントが一列に並んで配置される「一次元格子市場構造」とし、終端がもう一方の終端とループ状に繋がっていると仮定する。すなわち、エージェント i とエージェント j の関係性は以下のように設定される。

$$R_{i,j} = \begin{cases} 1 & \\ & if \quad i - r \leqq j \leqq i + r, j \neq i \\ 0 & \\ & otherwise \end{cases}$$

ただし、j の範囲は定義通り $0 \leqq j < N$ である。また、近傍範囲 r は片側の並びにおける関係人数であり、各消費者エージェントは $(r \times 2)$ 人と関係をもっているということになる。

シミュレーションは離散的な時間ステップに従って行われ、各消費者エージェントは並行して動作する。一次元格子状の世界の場合、それを時系列に並べていくと、状態遷移の歴史が一目で把握できるという利点がある。このような表示の仕方をここでは、歴史的な遷移を含んだ地図という意味で「ヒストリカルマップ」表現と呼ぶことにする。

8.2.2 エージェント

内部モデルのプロセスに従って意思決定する消費者エージェントのモデル化にあたっては、消費者行動論における先行研究が参考になる。ここでは、代表的なモデルの一つである Engel-Blackwell-Miniard(EBM) モデル (Engel et al., 1995) を基本的な枠組みとしてとりあげたい。EBM モデルは消費者の購買意思決定を初期状態から目標状態に至る心的操作の系列とみなし、記憶や情報処理などの認知的なメカニズムによって購買の過程を記述したものである。EBM モデルには、各フェーズにおける意思決定のアルゴリズムなどは定義されていないため、包括的な枠組みを提供するための概念モデルといえる。

本論文では、EBM モデルの欲求認識、情報探索、購買前代替案評価、購買、消費、購買後代替評価、処分の七つの基本フェーズに基づいて各消費者エージェントの意思決定プロセスを定義していくことにする。

欲求認識フェーズ

欲求認識は、イノベーションの普及家庭に基づいて行われる。このモデル化では、市場に存在する消費者エージェントのうち、ある特定の割合のエージェントが家庭用

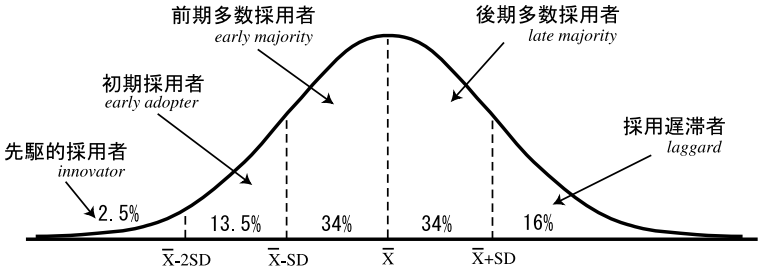
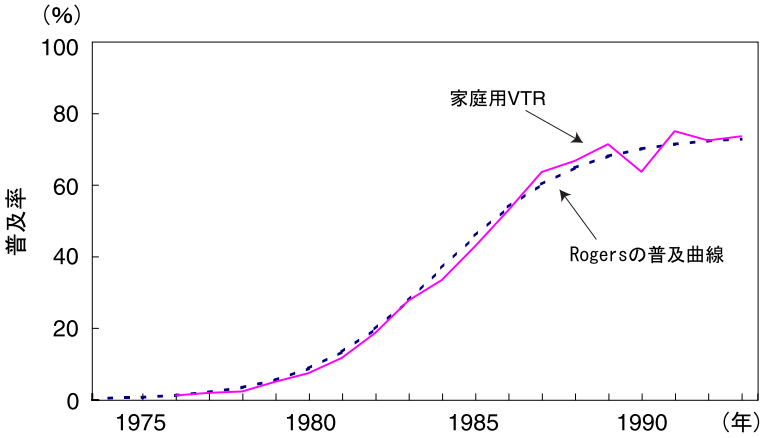


図 8.2: Rogers によるイノベーションの採用時期の採用者分布 (Rogers, 1982)



(家庭用 VCR の普及率は、経済企画庁 (1982–1996) より作成)

図 8.3: 日本における家庭用 VCR の普及と Rogers の普及曲線の比較

VCR 製品に対する欲求を認識する[75]。イノベーションの普及過程は、Rogers (1982) によって 3000 件以上の事例研究をもとにモデル化されており、日本における家電製品の普及もこの普及モデルで記述できることが知られている。Rogers の普及モデルにおいては、消費者は採用する時期によって先駆的採用者、初期採用者、前期多数採用者、後期多数採用者、採用遅滞者に分類され (図 8.2)、これを累積で表すとシグモイド関数となる。

Rogers の普及曲線を家庭用 VCR の普及に照らし合わせてみると、普及率 $d(t)$ は以下のように近似できることがわかる (図 8.3)[76]。

$$d(t) = \frac{1}{1 + exp(-((t - 1975) - 10)/2)}$$

ここで t は年を表している。

ここでのモデルでは、各時間ステップにおいて普及率 $d(t)$ に見合う数の消費者エー

ジェントをランダムに選出し、欲求を認識させる。

情報探索フェーズ

欲求を認識したエージェントは、次に情報探索を行う。情報探索は大きく分けて外部情報探索と内部情報探索に分けられるが、前者はエージェントの外部の情報の探索を意味し、後者はエージェント内部にある好みや記憶などの情報の探索を意味している。外部情報探索の結果、大域的なマーケットシェアとその消費者エージェントを取り巻く局所的なシェアの情報を獲得する。時間 t における方式 j の大域的なマーケットシェア $G_j(t)$ は、

$$G_j(t) = \begin{cases} \dfrac{\sum\limits_{0 \leqq k<N} H_{kj}(t)}{\sum\limits_{0 \leqq h<F} \sum\limits_{0 \leqq k<N} H_{kh}(t)} \\ \quad if \sum\limits_{0 \leqq h<F} \sum\limits_{0 \leqq k<N} H_{kh}(t) \neq 0 \\ 0 \\ \quad otherwise \end{cases}$$

で与えられる。ここで、F は規格競争をする方式の総数である。また、$H_{ij}(t)$ は時間 t においてエージェント i が方式 j を所有しているかどうかを表し、所有していれば $H_{ij}(t) = 1$、所有していなければ $H_{ij}(t) = 0$ となる。エージェント i の方式 j の局所的なシェア $L_{ij}(t)$ は、

$$L_{ij}(t) = \begin{cases} \dfrac{\sum\limits_{0 \leqq k<N, k \neq i} (H_{kj}(t) \cdot R_{i,k})}{\sum\limits_{0 \leqq g<F} \sum\limits_{0 \leqq k<N, k \neq i} (H_{kg}(t) \cdot R_{i,k})} \\ \quad if \sum\limits_{0 \leqq g<F} \sum\limits_{0 \leqq k<N, k \neq i} (H_{kg}(t) \times R_{i,k}) \neq 0 \\ 0 \\ \quad otherwise \end{cases}$$

で与えられる。さらに内部情報探索の結果、各エージェントは自分自身の各方式に対する選好を得る。エージェント i の方式 j に対する選好 P_{ij} は、0 から 1 までの実数値とする。

購買前代替案評価フェーズ

欲求を認識した消費者エージェントは情報探索を行った後、その情報をもとに各方式の評価を行う。ここでは、時間 t におけるエージェント i の方式 j に対する効用を計算するために、以下のような線形の効用関数 $U_{ij}(t)$ を仮定する。

$$U_{ij}(t) = l \times L_{ij}(t) + g(t) \times G_j(t) + p \times P_{ij}$$

ここで、l と p はシミュレーション開始時に設定される定数であり、それぞれ、局所的なシェアと選好が効用におよぼす影響度を表している。$g(t)$ は大域的なマーケットシェアの影響度を表すものであるが、ここでは代替的な二つのモデルを用意する。

本論文で主に用いる「シグモイド型大域的影響度」のモデルでは、大域的影響度 $g(t)$ は以下のような時間 t の関数として定義される。

$$g(t) = \frac{g'}{1 + exp(-((t - 1975) - 7))}$$

このシグモイド関数は 1983 年付近から増加する曲線である。既に述べたように 1980 年代半ばのレンタルビデオの解禁や普及版セルビデオなどの影響によって、普及の途中から大域的な互換性が重要になってきたことを表現しており、現実のビデオレンタルショップの店舗数の推移に近似するように設定されている (図 8.4)。ここで、g' は $g(t)$ の最大値を表しており、l と p と同様、シミュレーション開始時に設定される定数である。

比較分析のための代替的なモデルである「定数型大域的影響度」モデルでは、大域的影響度 $g(t)$ は時間とは無関係な定数であると定義し、以下ように定義する。

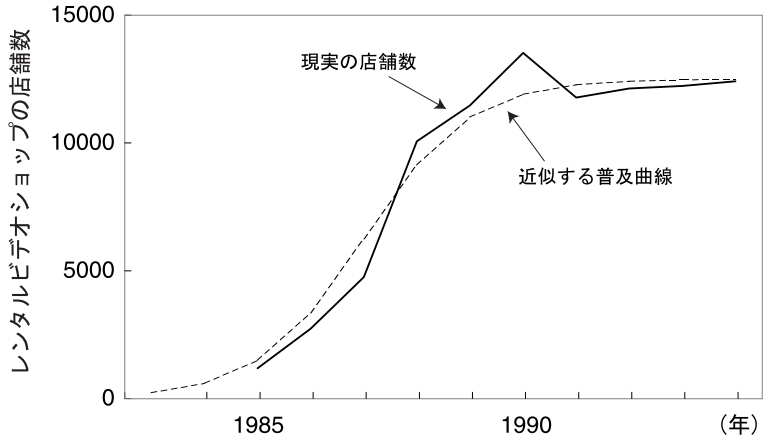
$$g(t) = g'$$

ここで、g' は大域的影響度を表す定数で、シミュレーション開始時に設定するものである。

購買フェーズ

欲求認識した消費者エージェントは、情報探索で得た情報を用いて購買前代替案評価を行い、その評価をもとに一方の方式を購入する。本論文では方式選択について代替的な二つのモデルを用意する。

本論文で主に用いる「多項ロジット選択」モデルでは、購入時の方式選択を多項ロジットモデルに従って確率的に行うとする。すなわち、エージェント i が方式 j を選ぶ確率は



(レンタルショップの店舗数は電通総研 (1996) より)

図 8.4: レンタルビデオ店舗数の推移とそれに近似する Rogers 普及曲線

$$Prob_{ij}(t) = \begin{cases} \dfrac{exp(U_{ij}(t))}{\sum\limits_{0 \leqq k<F} exp(U_{ik}(t))} \\ \quad if \quad \sum\limits_{0 \leqq k<F} exp(U_{ik}(t)) \neq 0 \\ 0 \\ \quad otherwise \end{cases}$$

で与えられる (Luce and Suppes, 1965; Thiel, 1969)。この関数はマーケティング・サイエンスにおいて現実の購買選択との適合度が高いことが知られており、よく用いられているものである (片平, 1994; 片平および杉田, 1994; 清水, 1999)。

比較分析のための代替的なモデルである「効用最大化選択」モデルでは、効用が最大となるように方式選択を行うとする。すなわち、エージェント i は、各方式に対する効用 $U_{ij}(t)$ を比較し、もっとも高い値の方式を選択する。

消費フェーズ

ここでは消費を明示的に扱わないため、消費に関するモデルは定義しない(77)。

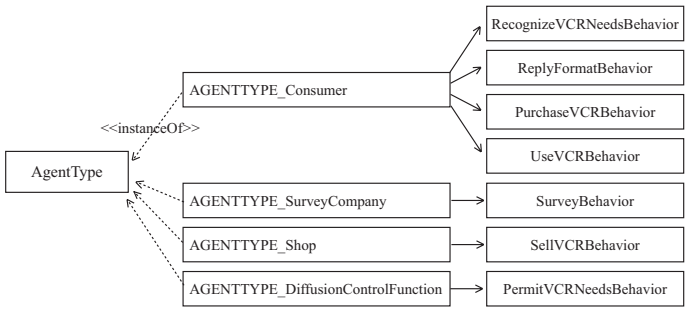


図 8.5: 規格競争モデルにおける AgentType と Behavior

図 8.6: 規格競争モデルにおける GoodsType

購買後代替案評価フェーズ

ここでは購買後代替案評価を明示的に扱わないため、購買後代替案評価に関するモデルは定義しない(78)。

処分フェーズ

本論文では処分に関連するモデルとして、製品の耐久性についての代替的な二つのモデルを定義する。「有限耐久性」モデルでは、家庭用 VCR の耐久性を D 年とし、購入から D 年たつと故障するように設定する。そのため購入から D 年経過した製品をもつ消費者エージェントは、所持製品を処分し、初回と同じプロセスによって新たな家庭用 VCR を購入する。もう一方の「無限耐久性」モデルでは、家庭用 VCR の耐久性は無限であり故障することがないとする。この場合消費者エージェントは故障による買い換えを行うことはない。

8.3 シミュレーションモデル

Behavior の状態遷移は、図 8.9 から図 8.15 のようになる。シミュレーションの流れは、以下のようになる。まず、Priority が高く設定されている SurveyCompany エージェントが、TimeEvent を受信する (図 8.9)。この TimeEvent を受けて、SurveyBehavior では、調査対象者に対して使用規格を尋ねる。これを受けて、Consumer エージェントは、ReplyFormatBehavior で自分の持っている規格を返答する (図 8.10)。SurveyCompany

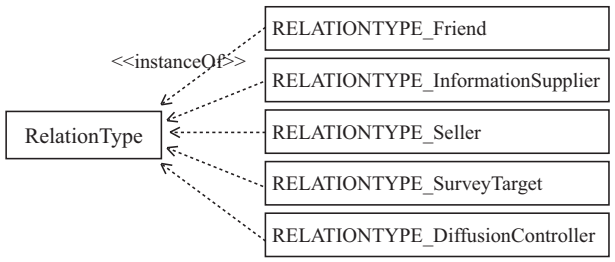


図 8.7: 規格競争モデルにおける RelationType

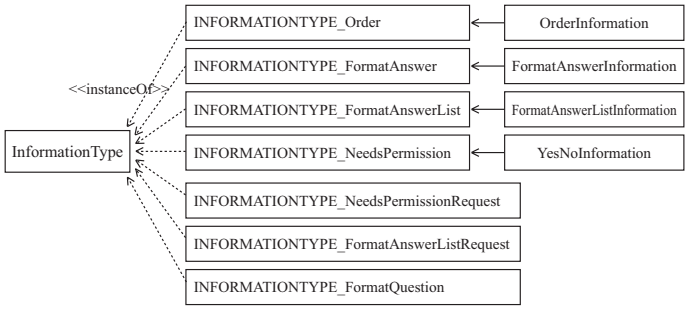


図 8.8: 規格競争モデルにおける InformationType

エージェントは、すべての返答を受け取った後、各規格の市場シェアを計算する (図 8.9)。

次に、Consumer エージェントが TimeEvent を受信する。どの順番で Consumer エージェントが TimeEvent を受信するのかは、ランダムになっている。最初の段階では、Consumer エージェントは、RecognizeVCRNeedsBehavior と ReplyFormatBehavior のみをもっている。RecognizeVCRNeedsBehavior で TimeEvent を受けると、DiffusionControlFunction エージェントに欲求許可を依頼する (図 8.11)。DiffusionControlFunction エージェントは、分析レベルでは登場しなかった設計レベルのエージェントで、VCR の普及率を制御するための機能エージェントである[79]。Consumer エージェント依頼を受けて、DiffusionControlFunction エージェントは、PermitVCRNeedsBehavior で、その可否を返答する (図 8.12)。Consumer エージェントの RecognizeVCRNeedsBehavior は、許可されなかった場合には最初の待ち状態に戻り、許可された場合には PurchaseVCRBehavior を生成し、RecognizeVCRNeedsBehavior 自らはその役割を終え、消滅する (図 8.11)。

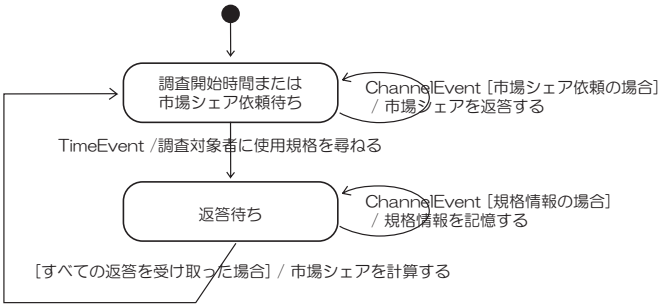


図 8.9: SurveyCompany エージェントの SurveyBehavior

図 8.10: Consumer エージェントの ReplyFormatBehavior

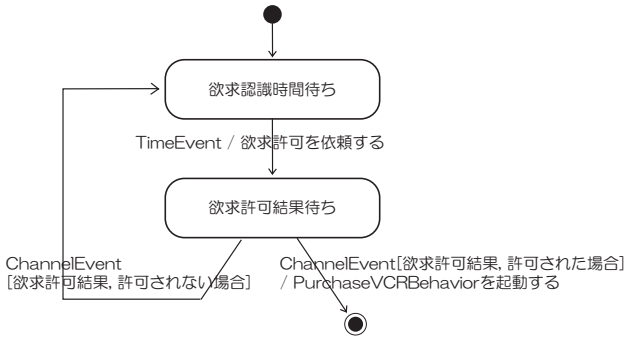


図 8.11: Consumer エージェントの RecognizeVCRNeedsBehavior

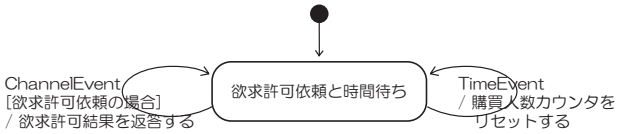


図 8.12: DiffusionControlFunction エージェントの PermitVCRNeedsBehavior

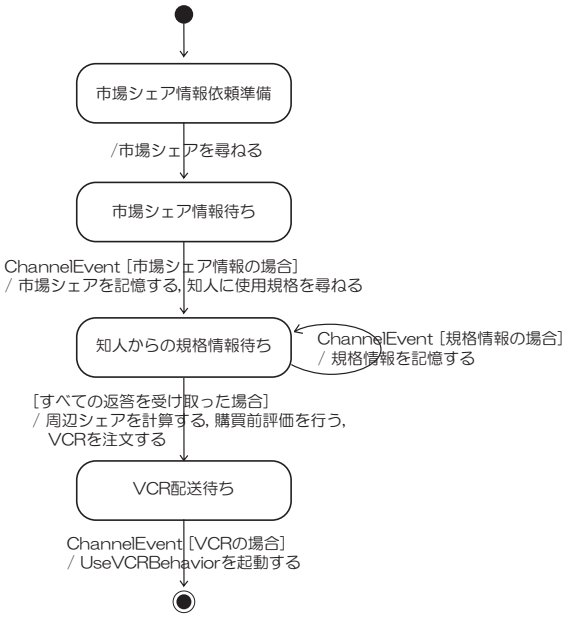


図 8.13: Consumer エージェントの PurchaseVCRBehavior

PurchaseVCRBehavior が起動すると、早速自動遷移が実行され、SurveyCompany エージェントに市場シェア情報を依頼する (図 8.13)。SurveyCompany は、SurveyBehavior で先ほど 調べた市場シェアを返答する (図 8.9)。それを受けた Consumer エージェントは、今度は知人全員に使用規格を尋ね (図 8.13)、知人 (これも Consumer エージェント) が ReplyFormatBehavior で返答する (図 8.10)。これらの情報をもとに、Consumer エージェントはどの規格を買うかを決定し、Shop エージェントに注文を出す (図 8.13)。Shop エージェントは、SellVCRBehavior で注文をストックしていく (図 8.14)。

そして、すべての Consumer エージェントが TimeEvent を受け取って行動した後、Priority が最も低いに Shop エージェントに TimeEvent が送信される。Shop エージェントは、ストックしていた注文の送信者全員に、VCR を配送していく (図 8.14)。VCR を受け取った Consumer エージェントは、UseVCRBehavior を生成し、PurchaseVCRBehavior 自らはその役割を終え、消滅する (図 8.13)。

Consumer エージェントは、次回から TimeEvent を受けるたびに、UseVCRBehavior

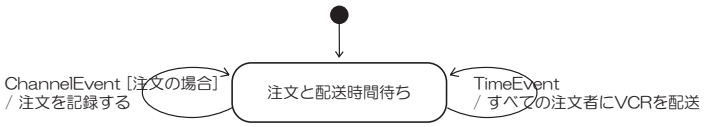


図 8.14: Shop エージェントの SellVCRBehavior

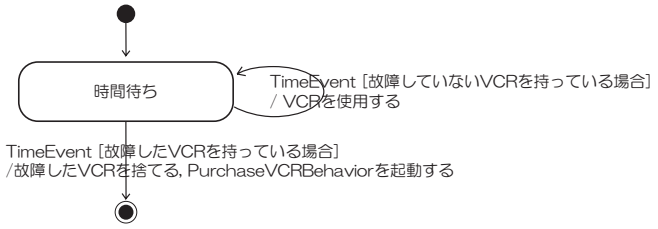


図 8.15: Consumer エージェントの UseVCRBehavior

によって VCR を使用 (耐久残存年数が減少) していく (図 8.15)。耐久残存年数が 0 になった場合には、その次の TimeEvent を受信したときに、PurchaseVCRBehavior を生成し、UseVCRBehavior が消滅する。Consumer エージェントは PurchaseVCRBehavior で、初回の購入と同じプロセスで VCR を再購入する (図 8.13)。

このモデル記述における特徴は、エージェントが必要に応じて、行動を追加したり削除したりしている点である (図 8.16)。このような行動の追加・削除の意義は、モデルの意味的側面と、シミュレーションの技術的側面がある。まず、モデルの意味的側面とは、エージェントは欲求を認識した時点で初めてその商品の購入を行うため、その時点で行動が生成される方が、より自然なモデルであるということである。また、技術的側面というのは、すべてのエージェントが可能性のあるすべての行動をあらかじめ持っているということは、メモリ使用が非効率となる可能性があるということである。ただし、このことは、その度ごとに行動オブジェクトを生成する負荷とのトレードオフになる。このモデルでは、市場にいる 1024 エージェントのうち、ある時間ステップにおいて商品購買行動を行うのはほんの一部のエージェントであること、そして、その商品購買行動は 1 回 (無限耐久性の場合)〜数回 (有限耐久性の場合) にすぎないということから、必要に応じて行動オブジェクトを生成するという方法を採用している。

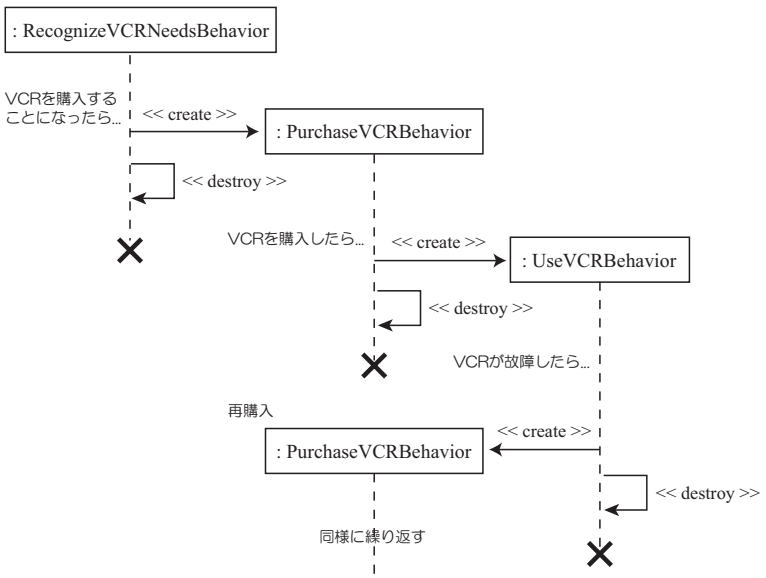


図 8.16: 規格競争モデルにおける Behavior の動的な生成と消滅

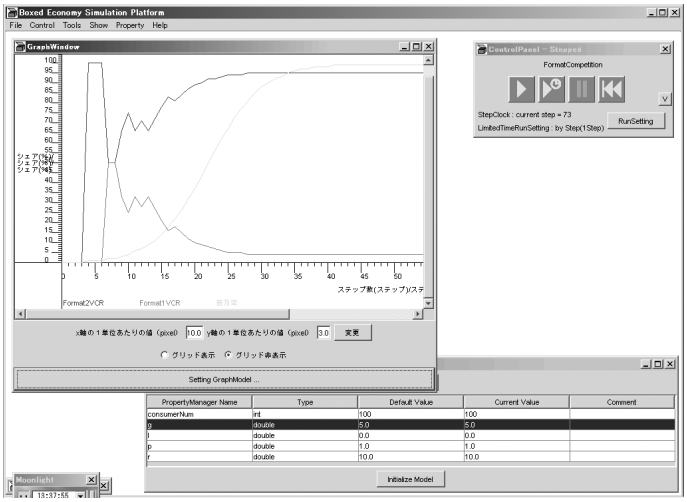


図 8.17: BESP 上での規格競争モデルのシミュレーション実行画面

8.4 シミュレーション結果

8.4.1 設定

ここでは、以下の設定で行ったシミュレーションの結果を紹介する(80)。まず、家庭用 VCR の事例では VHS 方式と Beta 方式の二方式の規格競争となるので、方式の総数は $F = 2$ となる。また、シミュレーション期間は $t = 1975$ 年から $t = 1995$ 年までとし、時間ステップは $\Delta t = 0.5$ 年とする。家庭用 VCR の耐久消費年数は約 7 年といわれているため (経済企画庁, 1982 – 1996)、有限耐久性の場合には耐久年数を $D = 7$ 年とする。

市場を構成する消費者エージェントの数は $N = 1024$ とする(81)。ここではエージェント間関係における強度を設定せず、関係がある場合には 0、ない場合には 1 をとるとし、双方向 $R_{ij} = R_{ji}$ とする(82)。これらはシミュレーションの開始時点で設定されてから不変とする。さらに、消費者エージェントの選好の影響度を $p = 1$ に設定し、エージェント i の方式 j に対する選好 P_{ij} は、各方式に差を設けず一様とし、シミュレーション開始時にランダムに決定する(83)。

シミュレーションは、分析したい設定について 40 回実行し、その結果のすべてもしくは一部を用いて分析を行う。本論文においては二方式の間に本来的な差異を設けていないため、以下の分析では各規格競争シミュレーションにおいて最終的に大きなマーケットシェアを獲得した方を「優位方式」として分析の対象とすることにしたい。

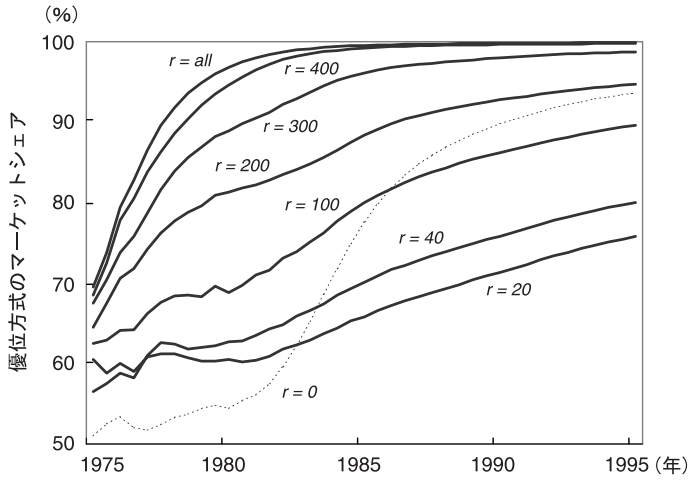


図 8.18: 近傍範囲 r を変化させた場合のマーケットシェア推移の比較［シグモイド型大域影響度, 多項ロジット選択, 無限耐久性, $l = 10$, $g' = 10$ の場合］

8.4.2 基本的な振舞いの確認

具体的なシミュレーション分析に入る前に、一部の設定とモデルの振舞いとの関係を把握しておくことにしよう。ここでは近傍範囲 r と耐久性についての基本的な特徴を理解した上で、次節では近傍範囲を固定することにし、また耐久性についても片方だけを取り上げることにしたい。

近傍範囲とマーケットシェアの関係

近傍範囲 r が優位方式のマーケットシェアの拡大にどのように影響しているかを調べるために、近傍範囲 r を変化させた場合のそれぞれのシェアの推移を表すと図 8.18 のようになる。この図から近傍範囲が大きくなるほど、シェアの拡大が大きくなることがわかる。これは、近傍範囲が大きくなると、初期の採用者の決定がより多くの追随者に影響を与えることになるため、初期のわずかな差異が大きく拡大することに起因している。

自明なことであるが、局所的シェアが大域的シェアに近づくことは最終的には全エージェントと関係をもつことになるため、すべての消費者エージェントの局所的シェアが大域的なマーケットシェアに等しくなり、局所性は消滅する。

耐久性の有無とマーケットシェアの関係

無限耐久性と有限耐久性ではマーケットシェアの推移にどのような差異が生じるかを調べると図 8.19 のようになる。有限耐久性モデルでは購入から $D = 7$ 年経ったと

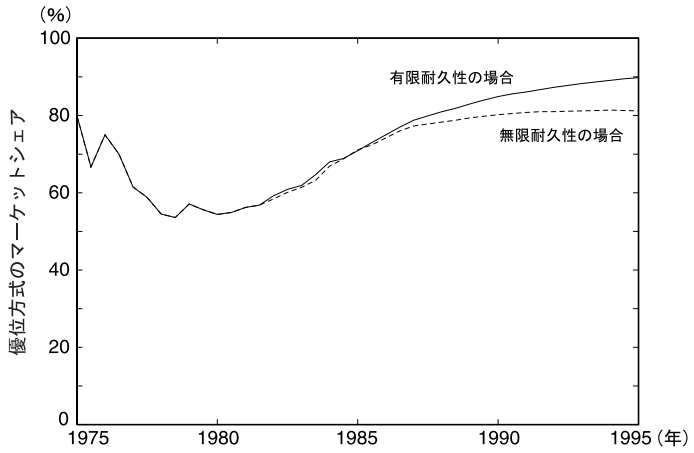


図 8.19: 耐久性の有無によるマーケットシェアの推移の変化［シグモイド型大域影響度, 多項ロジット選択, $r = 10$, $l = 0$, $g' = 5$ の場合］

きに買い換えが発生するため、普及の後半において優位な方式がシェアを伸ばす結果となる。この傾向は優位方式のシェアが大きいほど顕著になることが観察される。

8.4.3 マーケットシェアの推移と市場の状態遷移

マーケットシェアの推移を理解するために、市場状態の変化を観察し、マーケットシェアとの関係を分析してみよう。図 8.20 から図 8.23 は、マーケットシェアの推移と市場の状態遷移を表している。市場の状態遷移は、エージェント識別番号 $i = 0$〜139 の部分のヒストリカルマップによって、時間 t の変化に伴う各エージェントの所持方式の変化を示している。各セルは、そこに位置する消費者エージェントが方式 0 をもっている場合に黒色、方式 1 をもっている場合に灰色、そして何も持っていない場合に白色になる。ここでは、シグモイド型大域影響度, 多項ロジット選択, 無限耐久性、近傍範囲 $r = 10$ の場合の典型的な例を示す。

個人の選好のみに基づいて方式選択する場合には、マーケットシェアは初期の偏りの後、約 50%に落ち着く (図 8.20)。これは、各消費者エージェントの選好 P_{ij} が、乱数によって初期化されていることに起因しており、大数の法則によっても納得がいく結果である。ヒストリカルマップにおいても、各消費者エージェントがランダムに方式を選択している様子が観察される。

次に、個人の選好のほかに、局所的なシェアも考慮に入れて方式選択する場合を見てみよう (図 8.21)。マーケットシェアの推移を見る限りにおいては、個人の選好のみの場合と類似しているが、その具体的な状態遷移を調べると、大きく状況が異なっていることがわかる。図 8.21 のヒストリカルマップでは、地域ごとに採用されている

方式が分離するという「地域性」の発生が観察される。これは、各消費者エージェントは大域的なマーケットシェアを知ることなく、局所的なシェアに影響を受けるため、地域ごとの先駆的革新者によってたまたま選択された方式が、周囲の初期採用者や前期採用者の選択に影響を及ぼしていることによって生じている。このように、マクロ集計量では同様に見える現象であっても、ミクロ的にみると差異が観察されることがあり、ここにマルチエージェントモデルによってミクロ構造を明示的にモデル化する意義を見いだすことができる。

個人の選好と大域的なマーケットシェアに基づいて方式選択する場合は図 8.22 のようになる。マーケットシェアは、後半になってから大域的影響度 $g(t)$ のシグモイド関数の影響で優位方式がシェアを拡大する。ヒストリカルマップをみると、小規模の地域性が発生しているようにも見えるが、局所的影響の場合のような意味での地域性は発生していない。なぜなら設定上局所的影響が存在しないため、各消費者エージェントの選択方式と地理的要因との間に相関はないからである。採用人数が多いために、結果として同じ方式を選択した消費者エージェントが近接して位置しているにすぎないということである。

最後に、個人の選好、局所的なシェア、そして大域的なマーケットシェアに基づいて方式選択を行う場合を見てみることにしよう (図 8.23)。ここでも局所的影響によって引き起こされる地域性が観察される。しかも図 8.23 のヒストリカルマップからわかるように、地域性のクラスターが発生しており、このことが原因で優位方式の後半におけるシェア拡大が、大域的影響のみの場合に比べて抑制されていると考えられる。

8.4.4 局所的影響によるマーケットシェア抑制効果

局所的影響がある場合には、ない場合に比べて優位方式のマーケットシェアが若干大きくなる。それと同時に、図 8.21 および図 8.23 では、局所的影響がある場合には地域性が生じ、そのクラスターが防御壁となり優位方式のシェアの拡大傾向が抑えられるということも示唆された。ここでは、局所的影響によってマーケットシェアが抑制されるという仮説を詳しく調べていくことにしよう。

図 8.24 および図 8.25 は、局所的影響度 l が 0 から 20 まで、大域的影響度 g' が 0 から 50 までの整数値をとる場合の 1071($= 21 \times 51$) の各組み合せについてシミュレーションを行い、その優位方式の最終的なマーケットシェアを描いたものである。図 8.24 は無限耐久性の場合であり、図 8.25 は有限耐久性の場合である。このような「最終シェア・ランドスケープ」によって、局所的影響度と大域的影響度のパラメータ変化にともなうシミュレーションの振舞いを容易に把握することができる。

局所的影響がない場合 (図 8.24 と図 8.25 において局所的影響度 $l = 0$ の場合) には、大域的影響度 g' が大きいほど最終シェアも大きくなっているのがわかる。しかし局所的影響がある場合 ($l > 0$) には、ない場合に比べて最終シェアが大きくならないこと

がわかる。またその抑制の度合いは局所的影響度 l が大きいほど強くはたらいていることがわかる。

次に、大域的影響度を定義する「シグモイド型大域的影響度」モデルを「定数型大域的影響度」モデルに入れ換えた場合と比較してみよう (図 8.26)。定数型大域的影響度の場合にも大域的影響度 g' が大きいほど最終シェアは大きくなる。しかも定数型大域的影響度の場合は大域的影響が序盤から影響を及ぼすため、局所的影響による地域性が生じにくい。これに対しシグモイド型の場合には、大域的影響が効果をもちはじめたころにはすでに地域性が発生しているのでマーケットシェアの拡大が抑制されているということがわかる。この分析により、家庭用 VCR のように普及の途中まで局所的影響があるような規格競争においては、局所的な影響についても考慮する必要があるということが明らかになった。

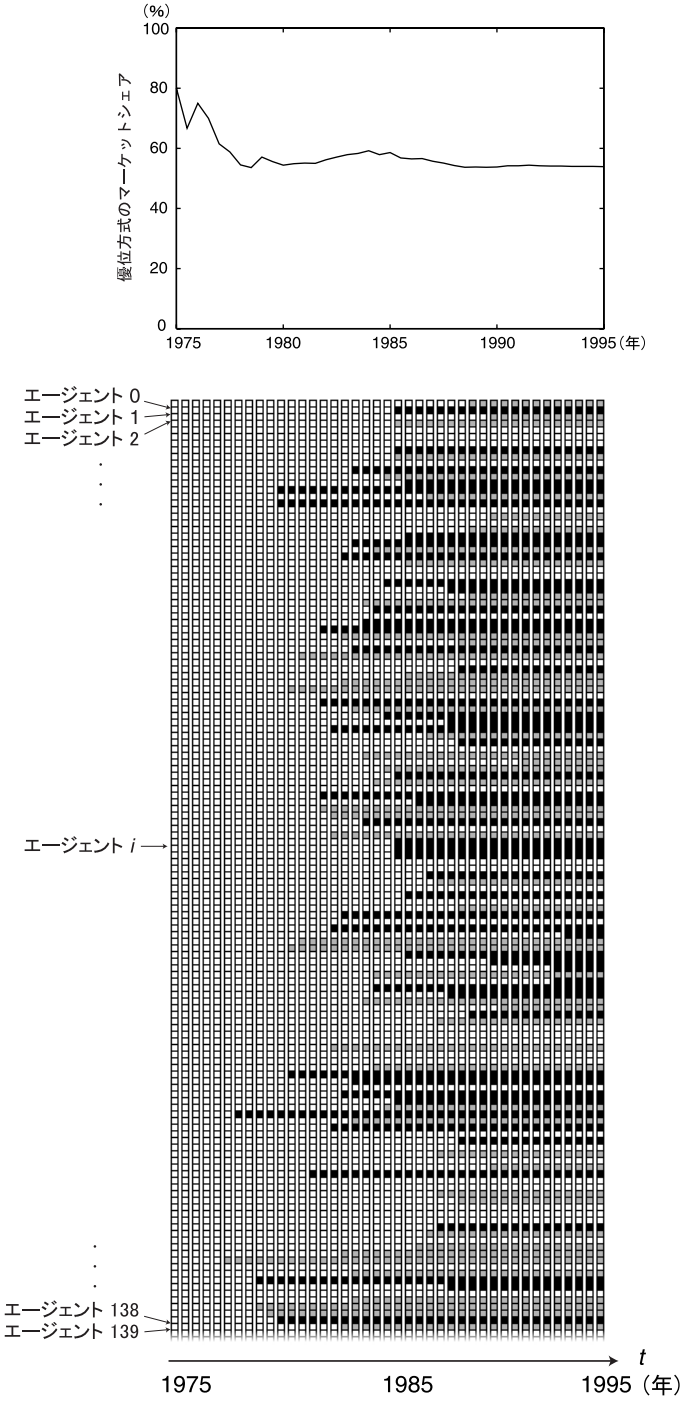


図 8.20: 個人の選好のみに基づいて方式選択する場合のマーケットシェアの推移と市場のヒストリカルマップ［シグモイド型大域影響度, 多項ロジット選択, 無限耐久性, $r = 10$, $l = 0$, $g' = 0$ の場合］

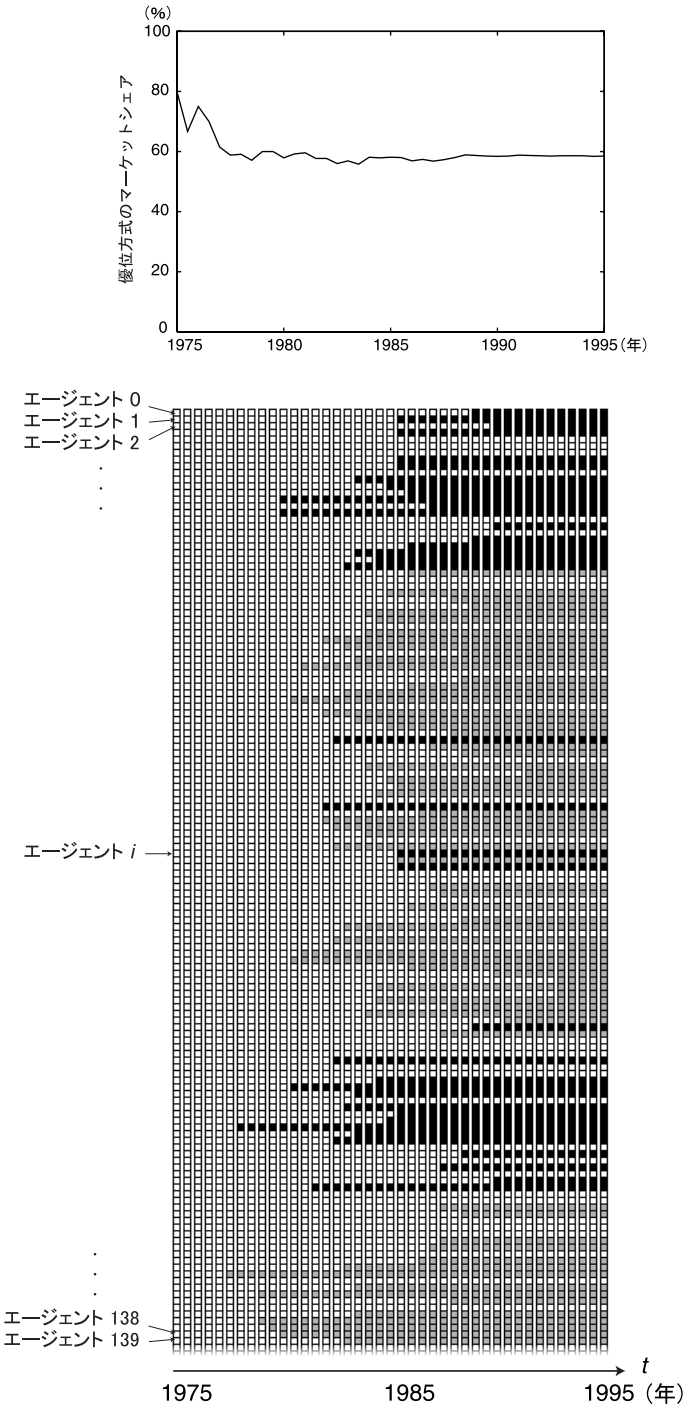


図 8.21: 個人の選好および局所的なシェアに基づいて方式選択する場合のマーケットシェアの推移と市場のヒストリカルマップ［シグモイド型大域影響度, 多項ロジット選択, 無限耐久性, $r = 10$, $l = 5$, $g' = 0$ の場合］

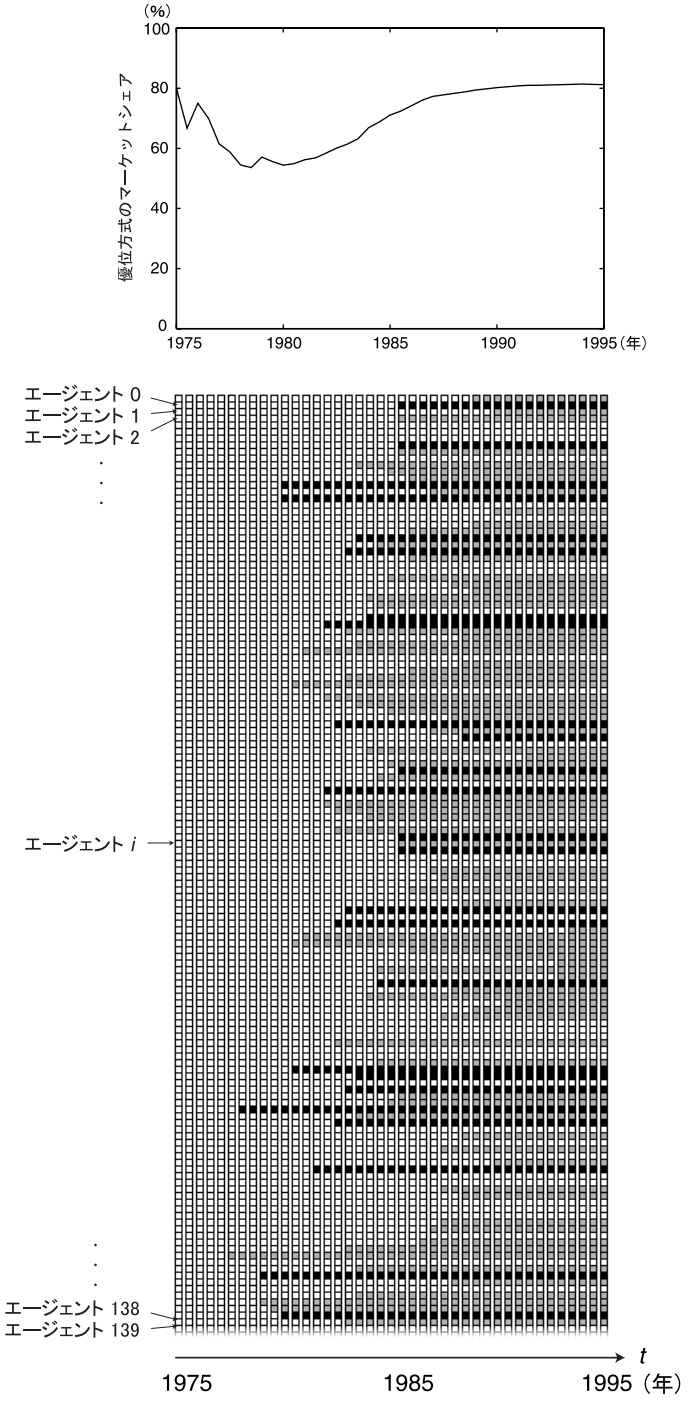


図 8.22: 個人の選好および大域的なマーケットシェアに基づいて方式選択する場合のマーケットシェアの推移と市場のヒストリカルマップ［シグモイド型大域影響度, 多項ロジット選択, 無限耐久性, $r = 10$, $l = 0$, $g' = 5$ の場合］

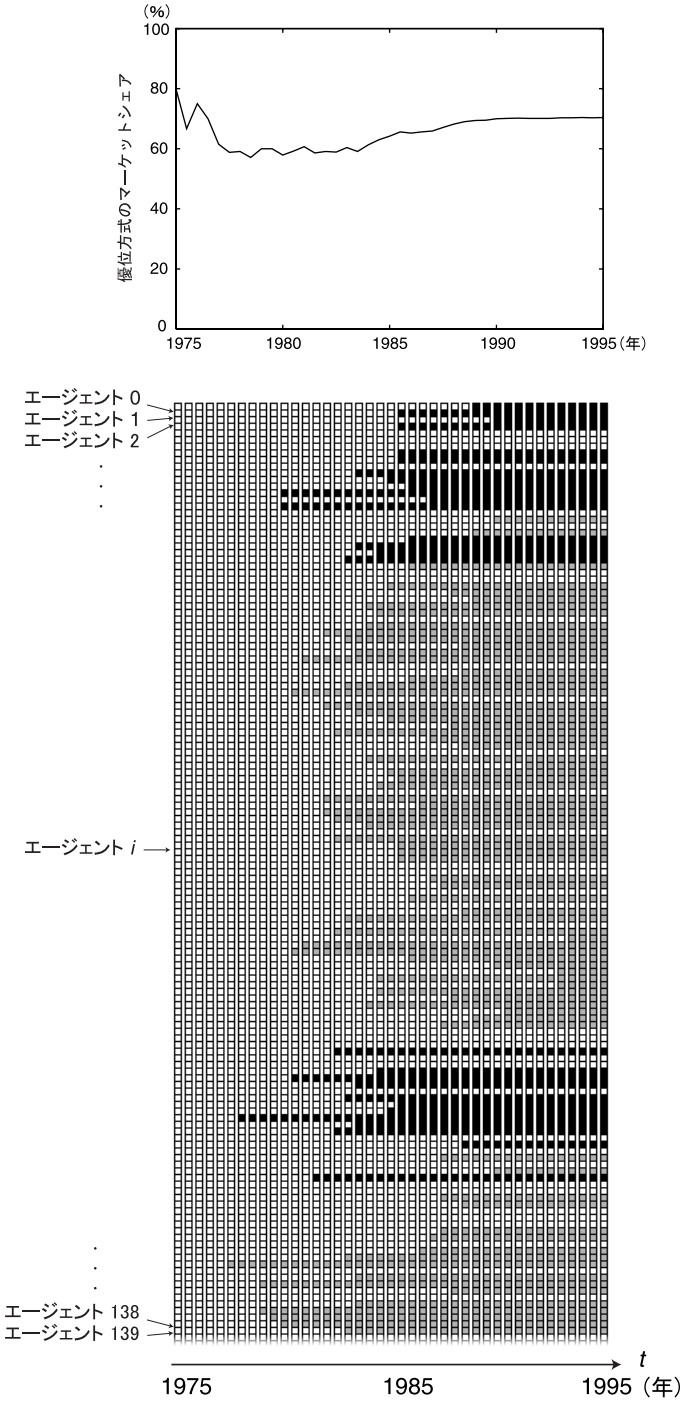


図 8.23: 個人の選好、局所的なシェア、および大域的なマーケットシェアに基づいて方式選択する場合のマーケットシェアの推移と市場のヒストリカルマップ［シグモイド型大域影響度, 多項ロジット選択, 無限耐久性, $r = 10$, $l = 5$, $g' = 5$ の場合］

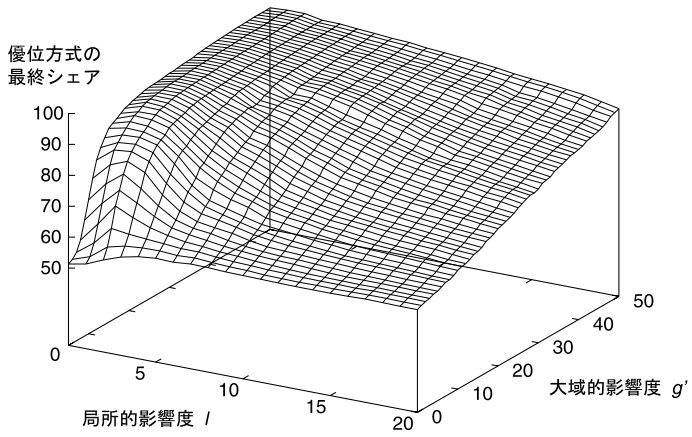


図 8.24: 最終シェア・ランドスケープ：局所的影響度 l と大域的影響度 g' のそれぞれの組み合せにおける優位方式の最終シェア［シグモイド型大域影響度, 多項ロジット選択, 無限耐久性, $r = 20$ の場合］

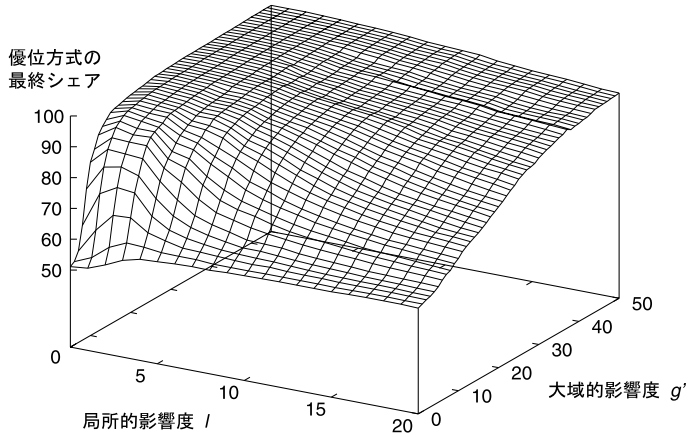


図 8.25: 最終シェア・ランドスケープ：局所的影響度 l と大域的影響度 g' のそれぞれの組み合せにおける優位方式の最終シェア［シグモイド型大域影響度, 多項ロジット選択, 有限耐久性, $r = 20$ の場合］

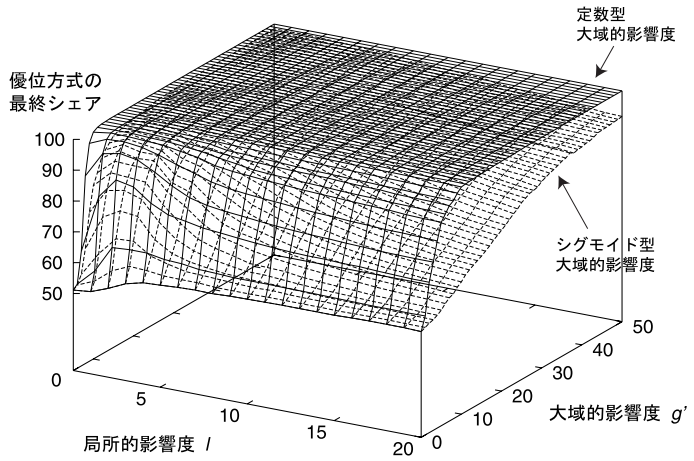


図 8.26: 大域的影響度に関するモデルの違いによる優位方式の最終シェア・ランドスケープの比較［多項ロジット選択, 有限耐久性, $r = 20$ の場合］

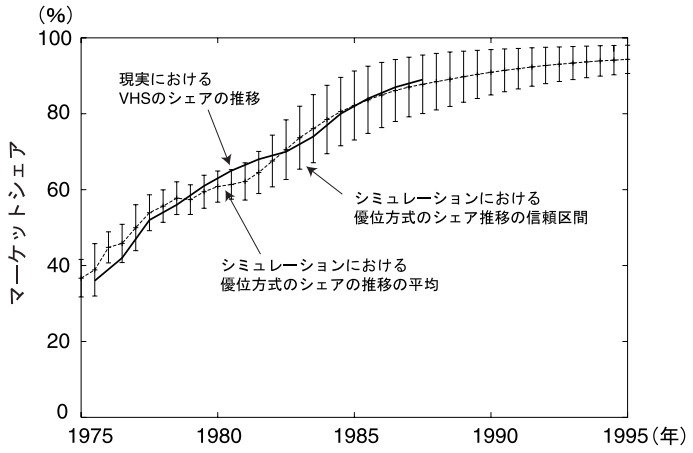


図 8.27: 現実のデータとの適合度が高い設定におけるマーケットシェア推移例［シグモイド型大域影響度, 多項ロジット選択, 有限耐久性, $r = 20$, $l = 10$, $g' = 49$ の場合］

8.4.5 現実のデータへの適合

モデルやパラメータがどのような組み合せの場合に、現実に近いマーケットシェアの推移が得られるのかを調べることにしよう。ここでは、現実に照らし合わせてより適していると思われる「シグモイド型大域的影響度」と「有限耐久性」、そしてマーケティング・サイエンスの研究成果を踏まえた「多項ロジット選択」のモデルを用いることにする。その設定のもとで、特定化されていない局所的影響度 l と大域的影響度 g' を変化させて適合度がどうなるかを観察する。

各設定について乱数の seed を変化させて 40 回シミュレーションを実行し、その平均と 95%信頼区間を算出する。評価に際しては、1976 年から 1988 年の現実の VHS のマーケットシェアに関する年次データ (表 8.1) の 13 項目のうち、信頼区間に入っている個数をここでの適合度と定義する。

図 8.27 は、適合度が 13 となる初期設定におけるマーケットシェアの平均推移とその信頼区間、および現実の VHS のマーケットシェア推移との関係を示している。また図 8.28 は、局所的影響度 l と大域的影響度 g' の組み合わせのそれぞれの場合の適合度を表している。このような「フィットネス・ランドスケープ」(Wright, 1931) で表現することにより、どのような値の組み合せのときに現実のデータに近くなるのかということが明確になる。

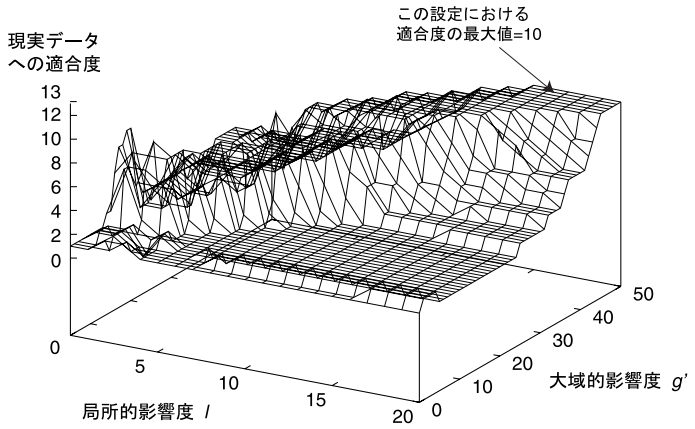


図 8.28: フィットネス・ランドスケープ：局所的影響度 l と大域的影響度 g' のそれぞれの組み合せにおけるシミュレーション結果の現実への適合度 (シミュレーション結果の 95%信頼区間内に存在する現実の推移点の数) ［シグモイド型大域影響度, 多項ロジット選択, 有限耐久性, $r = 20$ の場合］

表 8.1: 日本における VHS 方式と Beta 方式の累積マーケットシェアの推移 (Cusumano et al., 1992)

年	VHS 方式	Beta 方式
1975	-	100
1976	36	64
1977	42	58
1978	52	48
1979	56	44
1980	61	39
1981	65	35
1982	68	32
1983	70	30
1984	74	26
1985	80	20
1986	84	16
1987	87	13
1988	89	11

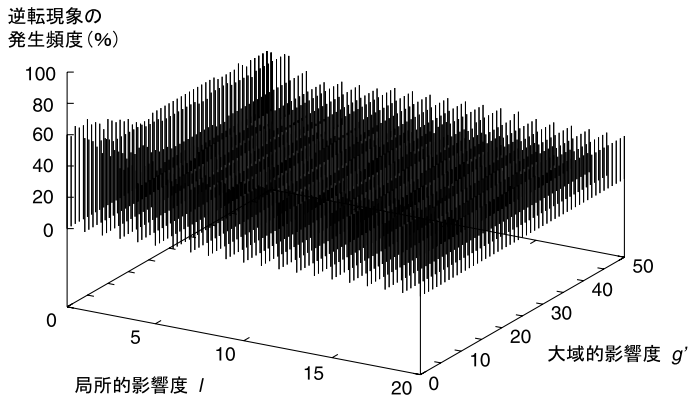


図 8.29: 「シグモイド型大域的影響度」と「多項ロジット選択」の組み合せにおける逆転現象の頻度［シグモイド型大域影響度, 多項ロジット選択, 有限耐久性, $r = 20$ の場合］

8.4.6 マーケットシェアの逆転現象

前節でのシミュレーション結果を、その平均ではなく個別に解析すると、約 3〜4 割程度の割合でマーケットシェアの逆転現象が生じていることがわかった。つまり、規格競争の序盤でマーケットシェアが他方より小さかった方式が中盤に逆転して優位方式となることが観察されるのである。局所的影響度 l と大域的影響度 g' のそれぞれの場合についてこの逆転現象がみられる頻度を表すと図 8.29 のようになる。

ここで、このような逆転現象を生み出す原因を明らかにするため、大域的影響度を定義する「シグモイド型大域的影響度」と「定数型大域的影響度」、そして購買における「多項ロジット選択」と「効用最大化選択」のそれぞれの組み合せの場合を比較してみることにしよう。図 8.30 から図 8.32 をみると、どの組み合せの場合もほとんど逆転現象が起こっていないことがわかる。つまり、マーケットシェアの逆転現象は、ある特定のモデル設定だけに起因するというのではなく、「シグモイド型大域的影響度」と「多項ロジット選択」の二つのモデルの組み合せによって生じやすくなるということである。

この組み合わせからわかることは、序盤において局所的な影響だけを受けながら確率的に方式選択する場合に逆転現象が生じ得るということである。序盤では製品を購入している消費者が非常に少ないため、そのわずかな差は欲求認識する消費者の位置や方式選択の偶然性によって簡単に覆される可能性があるのである。このような設定の場合には、Arthur (1994) が指摘するような初期値の鋭敏性が必ずしも言えるわけではないという結果となった(84)。また、普及の中盤以降では地域性の発生や大域的なマーケットシェアの影響などにより逆転現象は起こらないため、規格競争の結果を左右するのは、序盤の後半から中盤の始めにかけてであるということが示唆される。

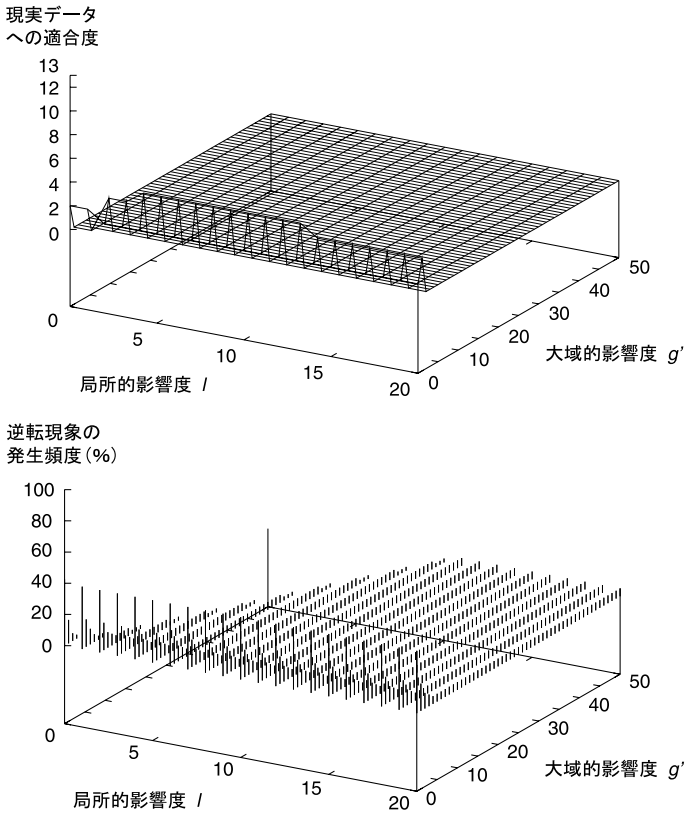


図 8.30: 「定数型大域的影響度」と「効用最大化選択」の組み合せにおけるフィットネス・ランドスケープと逆転現象の頻度［定数型大域的影響度, 効用最大化選択, 有限耐久性, $r = 20$ の場合］

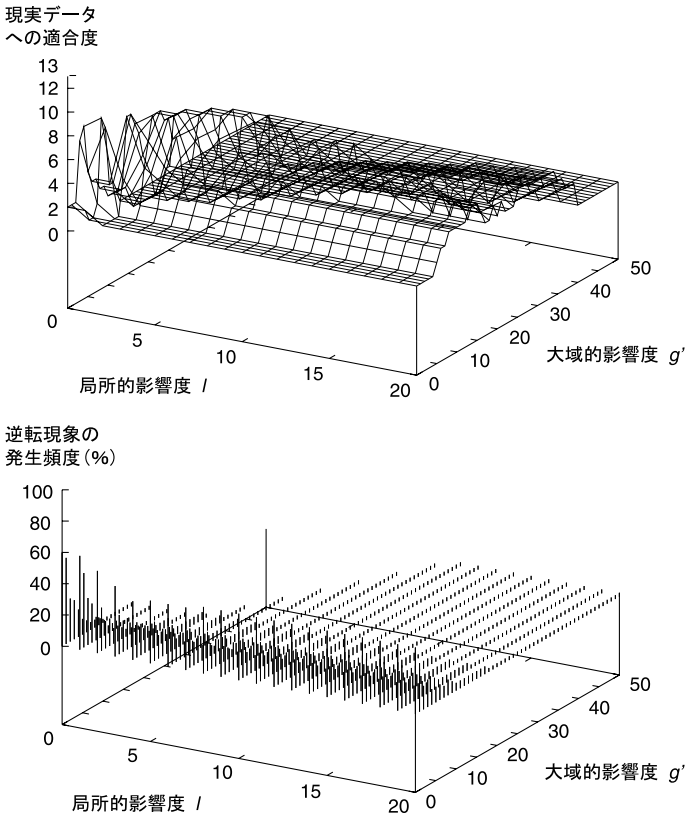


図 8.31: 「シグモイド型大域的影響度」と「効用最大化選択」の組み合せにおけるフィットネス・ランドスケープと逆転現象の頻度ランドスケープ［シグモイド型大域影響度, 効用最大化選択, 有限耐久性, $r = 20$ の場合］

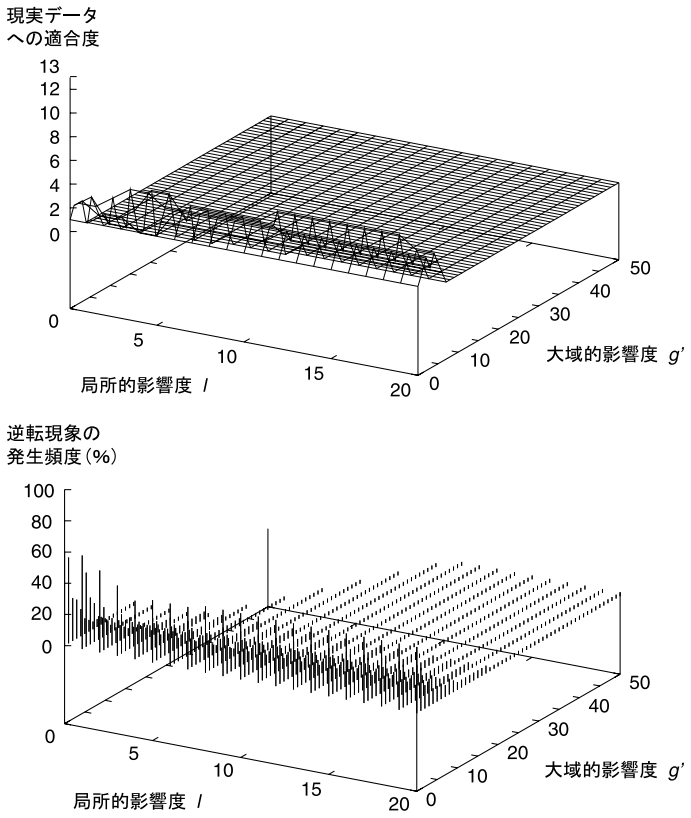


図 8.32: 「定数型大域的影響度」と「多項ロジット選択」の組み合せにおけるフィットネス・ランドスケープと逆転現象の頻度ランドスケープ［定数型大域的影響度, 多項ロジット選択, 有限耐久性, $r = 20$ の場合］

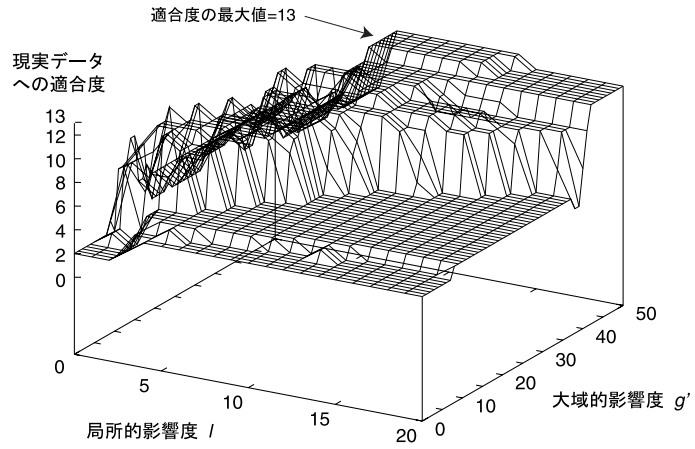


図 8.33: 逆転シミュレーションのみのフィットネス・ランドスケープ：局所的影響度 l と大域的影響度 g' のそれぞれの組み合せにおけるシミュレーション結果の現実への適合度 (シミュレーション結果の 95%信頼区間内に存在する現実の推移点の数)［シグモイド型大域影響度, 多項ロジット選択, 有限耐久性, $r = 20$ の場合］

マーケットシェアは数ではなく割合を表しているので、測定された時期によってその実質的な意味が異なってくるということが分析の際に注意すべき点であるといえるだろう。

逆転現象が生じたシミュレーション結果のみを用いて、信頼区間および適合度を再度計算し、フィットネス・ランドスケープを描くと図 8.33 のようになる。すべての結果を対象とした図 8.28 においては、結果の平均をとる際に序盤のばらつきが打ち消されてしまうため、序盤の推移が現実のデータと乖離してしまい、適合度が最高でも 10 であった。しかし、逆転現象の生じた結果のみを扱った図 8.33 では、序盤の推移も近くなるため、適合度が最高値の 13 となる組み合せが観察された。本シミュレーションによって、方式間に差異がない場合でも、マーケットシェアの逆転も含めて現実に近似したマーケットシェア推移を生み出すことが可能であるということが明らかになった。

8.5 考察

本論文では、商品市場の研究を進める手始めとして、特徴的な規格競争の事例を取り上げた。その中でも規格競争の事例として家庭用 VCR を取り上げた理由は、よく知られた事例である上、以下の点で扱いやすい事例だからである。

- 競争が二方式で行われているため、消費者の認知において混乱が生じにくい (伊丹および伊丹研究室, 1989)。

- どちらの方式も広告が盛んに行われており、消費者は両方式の製品ついて十分認知している (山田, 1993)。
- どちらの方式の製品も店頭に複数並んでおり、自由に選択できる状況にある (山田, 1993)。
- 高価な耐久消費財であるため、消費者の意思決定においてバラエティー・シーキング[(85)]のような例外的行動が起きにくい。
- 互換性が決定的に重要であったため方式間の価格差があまり問題にならない。
- 耐久消費財であるため、買い直しの回数が少ない。
- 普及の過程において次世代映像機器との世代間規格競争が起こらなかった。
- 代表的な耐久消費財であるため、データや文献が比較的多く存在している。

このような特徴により、一般の商品市場が本来もつ複雑性の多くを単純化することができた。その上で本論文では、モデルの作成において普及学やマーケティング・サイエンス、消費者行動論などの既存のモデルを用いて、モデル構築においても妥当性に注意を払った。また本論文ではモデルの評価として、逆転現象などの定性的な特徴と、現実のマーケットシェアとの定量的な比較を行った。

しかし、モデルの妥当性の検証に関しては、現実の年次データ 13 点との関係で評価したにすぎず、これだけでモデルが現実を説明しているということはできないだろう。データが比較的得やすい特徴的な事例であるにもかかわらず、妥当性を主張するのに必要なデータが圧倒的に不足しているという問題に直面した。以上のようなモデル構築の際の妥当性の組み込みと評価データの確保ということは今後のシミュレーション研究の課題として検討していく必要があるだろう。

同時に、社会シミュレーションの妥当性の検証を支えるものとして、ミクロレベルのデータやアンケートなどを多面的に用いて評価していくという方法論の構築も必要であると思われる。例えば、本論文で取り上げた家庭用 VCR を例にとると、家電製品を購入する際の情報経路に関する調査結果 (Robertson, 1971) によって、マスコミュニケーションは認知段階で強く影響し、パーソナルコミュニケーションが意思決定段階に強く影響を及ぼしているということがわかっている。また、家庭用 VCR の購入時の情報経路に関する調査結果 (廣島, 1985) では、家庭用 VCR の方式選択についてもパーソナルコミュニケーションが一つの重要な要因であったというデータが得られている。このようなアンケート・データをどのようにモデル構築や妥当性の検証に用いるのかということは、今後十分に議論されるべき問題であると思われる。

第9章　結言

本論文では、複雑系のシステム観に基づく社会・経済シミュレーションを作成すること支援するために、オブジェクト指向計算モデルを導入し、モデル・フレームワークとシミュレーション・プラットフォーム、そしてモデル・パターンを提案した。

まず、複雑系としての社会・経済モデルを記述するためのモデル・フレームワークとして、「Boxed Economy Foundation Model」を提案した。提案モデル・フレームワークでは、新しい行動の追加・削除・組み換えなどを柔軟に行うことができるため、狭義の複雑系のモデルの記述が可能となった。そして、シミュレーションの作成と実行を支援するソフトウェアとして「Boxed Economy Simulation Platform」を提案した。提案シミュレーション・プラットフォームでは、コンポーネントベースのアーキテクチャを実現しているため、複雑系研究で行われる構成的手法を支援することができ、コンポーネントの再利用も可能となった。さらに、動的なモデルの構成方法のノウハウを「モデル・パターン」としてまとめることを提案し、実際に 23 のモデル・パターンを提案した。これらの提案の有効性を明らかにするため、本論文では、代表的な既存モデルと独自モデルに提案しシステムを適用した。これらのモデルは、行動の組み換えやエージェントの生成等を伴う複雑系のモデルであり、これらを実際にモデル化し、シミュレートしたことで本提案の有効性を実証した。

本論文を貫くひとつの基本思想は、小さな部分の組み合わせで、より複雑なモデルを作るということである。「私たちの理解したい対象は部分に還元できない」ということと、「そのモデルを部分の組み合わせで作成する」ということは、一見矛盾するように思えるかもしれない。しかし、後者は、前者のようなモデルを作成するための工学的な知恵であって、必ずしも矛盾するわけではない。私たち人間は、同時にすべてを作成することはできないため、部分部分の作成の積み重ねで全体を作っていくほかない。しかしこのことは、全体が部分に還元できることを意味しているわけではない、ということは強調してもし過ぎることはない。部分と部分は、全体を構成するために不可分に結びついているのであり、それゆえ、モデル作成者に求められるのは、「全体性を考慮しながら部分を作りこむ」ということである。このとき、あまりに小さい粒度の部分に注目しなければならないとなると、全体との関係性を把握できなくなってしまうだろう。そこで、詳細を隠蔽して扱うことができるモデル・フレームワークのクラスや、さらにもう一つ大きなまとまりとしてのモデル・パターンが重要になるのである。

本論文の成果は、複雑系と進化の社会科学に向けてのほんの一歩に過ぎないだろう。しかし、今後具体的なモデルによる実証研究を行うとともに、本論文で提案してきたような研究基盤についても議論を重ね、さらなる発展をはかっていく必要がある。そのような刺激的な研究を、今後も多くの方とともに続けて行きたいと思う。

謝辞

本研究を遂行し、まとめるにあたって、実に多くの方にお世話になりました。この場を借りて、感謝の意を述べさせていただきたいと思います。

まず、指導教官主査の武藤佳恭先生は、自由な雰囲気のもと、研究環境から学会発表の支援まで、さまざまな面で見守っていてくださいました。そして、同じく主査ともいえるほど指導・支援してくださったのが、竹中平蔵先生です。竹中先生は国務大臣を務めるにあたり、最終的には副査から抜けることになりましたが、研究の初期段階から実に多くの議論につきあっていただき、また研究環境の面でも支援していただきました。古川康一先生には、大学院プロジェクトを通じて助言をいただき、あたたかく見守っていただきました。本論文の執筆に入るころ、「歴史に残る博士論文を。」という励ましをいただいたことで、思い切って取り組めたように思います。大岩元先生には、直接的な助言だけでなく、後に述べる技術的な協力者の方々の指導教官としても間接的に多く助言をいただきました。小澤太郎先生には、修士のころから、折に触れてアドバイスをいただきました。熊坂賢次先生は、「新しい考え方で世の中をみるには、分析装置も新しくないとね。」と、本論文の意義を指摘してくださいました。

本研究の後半戦は、千葉商科大学に勤めながら進めました。千葉商科大学学長の加藤寛先生、政策情報学部長の井関利明先生には、研究環境の支援だけでなく、内容についても助言をいただきました。また、同僚であり、同じく博士 (政策・メディア) を目指す玉村雅敏さんと久保裕也さんには、幅広く議論していただいただけでなく、幾度となく励ましの言葉をいただきました。

本論文の内容は、私が立ち上げ代表をしている Boxed Economy Project のメンバーとの共同研究に多くを負っています。もともと社会・経済シミュレーションのフレームワークとシミュレーション・プラットフォームの構築ということを掲げてはいたものの、それを現在の BEFM や BESP のようなきわめて洗練されたかたちで実現することができたのは、中鉢欣秀さんの力が不可欠でした。そして、BESP の開発にあたり、海保研さん、松澤芳昭さん、浅加浩太郎さん、青山希さんには、かなり力を注いでいただき、設計思想や実装面での貢献ははかりしれないものがあります。また、上橋賢一さん、津屋隆之介さん、田中潤一郎さん、高部陽平さん、廣兼賢治さんとは、BEFM を作成するにあたり、かなり時間をかけて一緒に議論していただきました。特に、上橋さんとは、社会科学における BEFM の役割・意義について、徹底的な議論

を行いました。また、津屋さんには、本論文の執筆の段階において、モデル作成や作図などの点でも協力していただきました。そして、山田悠さんには、人工株式市場モデルの作成や科学哲学的な議論などにおいて協力していただきました。北野里美さんと森久保晴美さんには、ビジネスとの接点や国際化に向けての準備などでお世話になりました。岡部明子さんには、モデル・パターンの作成や作図等で協力していただきました。そして、Boxed Economy Project のメンバーではありませんが、島広樹さんと岩村拓哉さんとの議論や構想も、本研究の下敷きになっています。社会・経済シミュレーション、ひいては社会科学に、オブジェクト指向、フレームワーク、プラットフォーム、パターンなどを導入するという本研究の方向性が構想されたのも、島さん、岩村さんとの議論を行った初期の段階でした。

学会においてさまざまな先生方に助言と叱咤激励をいただいたことは、本研究に大きく影響しています。ここですべての方のお名前を挙げることはできませんが、その中でも特にお世話になった方は、塩沢由典先生 (大阪市立大学)、寺野隆雄先生 (筑波大学)、生天目章先生 (防衛大学)、出口弘先生 (東京工業大学)、松井啓之先生 (京都大学)、和泉潔さん (産業技術総合研究所) である。また、進化経済学会の若手研究者の方々、西部忠さん (北海道大学)、吉田雅明さん (専修大学)、橋本敬さん (北陸先端科学技術大学院大学)、江頭進さん (小樽商科大学)、中島義裕さん (大阪市立大学)、篠原修二さん (京都産業大学)、遠藤正寛さん (慶應義塾大学)、鈴木健さん (東京大学)、吉地望さん (北海道大学)、澤辺紀生さんとも、有益な議論をさせていただきました。

武藤佳恭研究室の仲間、そしてノーベルコンピューティングプロジェクトの仲間には、いろいろとお世話になりました。特に、『複雑系入門』の共同執筆者である福原義久さんや、吉池紀子さん、舘俊太さん、岡宗一さん、武田圭史さんには、博士論文とそれに関連する提出書類のことについて、相談にのっていただきました。

本研究を遂行するにあたり、フジタ未来経営研究所から研究環境と多額の予算支援をいただいています。藤田田名誉所長および藤田元所長をはじめとして、事務局と研究員の皆様にはさまざまな面でお世話になりました。また、本研究の一部は、日本学術振興会特別研究員 (DC1) 時代に行ったものであり、文部省科学研究費補助金 (特別研究員) の支援を受けています。また、何度かにわたる森泰吉郎記念研究振興基金の支援、森基金国外学会発表経費補助、および高度科推進研究費をいただいて遂行されています。

最後に、父 亨と母 伎世子、弟 麗、そして妻 美穂は、研究生活のベースを支えてくれました。

以上の皆様の助言・支援・協力・励ましに対し、深く感謝申し上げます。

井庭 崇

注

(1) 例えば、本論文の対象である複雑系に近い社会システム論を展開しているものに公文 (1978); 公文 (1995) などがある。

(2) メタファーは、物事の類似性を間接的に暗示する。直喩が「A は B のような」というのに対し、メタファー (隠喩) は「A は B である」という形式になる。経済学におけるメタファーの重要性を指摘しているものに、Hodgson (1993); 塩野谷 (1998); 西部 (2000) などがある。

(3) システム論は、おもに生物を対象として、次のように変遷してきた (河本, 1995)。まず、システム論の前身といえる有機体論と機械論がある。これらを受けて、システム論の第一世代といわれる「開放性の動的平衡システム」が登場する。このシステム観の代表的なものに、ホメオスタシス、サイバネティクス、一般均衡理論がある。そこでは、システムが環境との相互交換を通じて境界を維持するというシステムと環境の図式が取り入れられる。そして、次にシステム論の第二世代といわれる「開放性の動的非平衡システム」が登場する。散逸構造、シナジェティクス、ハイパーサイクルなどである。そして、システム論の第三世代といわれるのが、本論文のテーマとなる複雑系やオートポイエーシスである。

なお、この脚注を含む以降の理論変遷の整理において、各原典のほかに、富永 (1993)、富永 (1995)、新および中野 (1984)、Buckley (1967)、今田 (1986)、新田 (1990)、浜嶋ほか (1997)、森岡ほか (1993)、飯尾 (1995)、徳安 (2000) を参考にしている。

(4) 社会学が誕生した当初、社会をどのようなものとしてみるかという考え方には、二つの流れがあった。一つが、社会を生物すなわち「有機体」としてみるというものである。そして、もう一つが社会を「機械」としてみるというものである。どちらも、具象的な実在物をアナロジーとして用いている。この段階で、「システム」という考え方の萌芽は見られたが、その概念が自覚的に用いられていたわけではないため、システム論には成り得ていない。

有機体的社会観では、社会を有機体と捉え、社会的分業を有機体の各器官の相互依存とその機能分化に見立て、生物学・解剖学・生理学の概念で解釈する。このような有機体のアナロジーは、古代ギリシア時代から行われてきたが、社会理論としては、サン-シモンが唱え、社会学の創始者の一人であるコントが「社

会有機体」の語を創出し、個人や家族などが有機的に結びついて社会的全体が構成されているとした。そして、イギリス社会学の創始者といわれるスペンサーが独自の立場からこれを組織的に理論化した。社会は成長・分化・諸部分間の相互依存・進化の点で社会と有機体は共通するとした。

18 世紀末以降すでに展開されていた「生気論」では、生命現象をもたらすという「生命力」を想定し、機械と一線を画そうとしたが、神秘的な力を仮構するという点で、科学的な土壌にはなじまなかった。要素還元主義に陥らず、かつ神秘的な力を仮構しないためには、どうすればよいか。この難題に対して突破口を開いたのが、有機構成 (organization) という考え方である。物理・化学的な要素が複合体を構成すると、要素単体では見られなかった新しい特性が生じるとするのである。この考え方により、物理・化学と無理なく接続することができるが、生命をその要素の物理・化学的性質に還元しないという道が開かれた。これが、生物学が採用し、後にシステム論における階層性や創発の考え方につながる。

社会学が誕生した当時、よりどころとなるものがなかったため、その頃かなり発展していた生物学や生理学にたよったのである。社会についての一般原理を構築するのではなく、その原理を同様に具体化していると思われる有機体という具体的な実体を、つねにメタファーとして用いた。つまり、社会と有機体には原理上の平行関係があるとみなしたのが、そこにその二つの共通原理を抽象化したシステム一般といったものを定義していないという意味で、システム論にはなり得ていない。しかし、この考え方は、社会学第二世代のデュルケームらによって、社会の解剖学および生理学として受け継がれることになり、社会を諸機能の相互連関としてとらえる考え方は、パーソンズの社会システム論に受け継がれることになる。

これに対し、機械論的社会観は、人間の社会を機械として捉える社会観である。そして、機械であるがゆえに、社会は物理学的な方法で解明することができるということを意味する。社会を機械とみなす考え方は、古くはホッブズ、デカルト、スピノザなどに遡るが、18 世紀の啓蒙思想において展開された。それは、人間や社会も自然の原理によって説明しようとする企てである。19 世紀の初期の社会学においては、「社会物理学」「社会機械論」「社会エネルギー論」などの名の下に展開されたが、それらの多くは表面的なアナロジーに終始して、社会現象の理解にはあまり貢献しなかったといわれている (Sorokin, 1928)。

この社会機械説の中で成功したのは、パレートである。パレートは、力学で発展をとげた連立方程式体系を経済学で導入し、ワルラスとともに一般均衡理論を確立したが、さらにその考え方を社会学に持ち込み、社会的均衡理論を生みだした (Pareto, 1916)。ここでシステムというのは、複数の要素が相互依存している全体のことである。これの要素間の相互依存関係は方程式体系として表すことができる。その特徴は、作用と反作用の循環的波及の結果、最終的には均衡に収斂

するというものでる。この均衡の考え方は、経済学で現在までひとつの中心的な柱となっているだけでなく、社会学でもホーマンズの社会的交換理論や合理的選択理論に受け継がれている。

(5) パーソンズは、その初期の理論において、ホメオスタシス (恒常性維持) の考え方を導入した (Parsons, 1951; Parsons, 1954)。ホメオスタシスとは、有機体で発達している機能で、環境の変化に対して体内の状態を恒常的に保持するメカニズムのことである (Cannon, 1932)。血液中の水や栄養素などを一定水準に保ったり、外気温にかかわらず体温を一定範囲に保ったりするメカニズムがこれに該当する。このとき、機械論的な均衡とは異なり、外界との交換をする開いたシステムにおける恒常性を核心としている。

パーソンズは、ホメオスタシス原理を取り入れることで、その後展開されることになるサイバネティクスと一般システム理論と同様に、システム-環境図式の視点を得ている。つまり、それまでに展開されていた社会有機体論や社会機械論は、システムの内部 (部分と全体の関係性) に目を向けていたのに対し、パーソンズの社会システム論では、システムとその外部 (環境) との関係に目が向けられている。

パーソンズは、心理学や経済学、政治学、人類学など社会諸科学の成果を取り込むような「一般化された社会システム論」を構想した。パーソンズは、生物システムと社会システムの固有の分析方法として、物理的なシステムのものとは異なる「構造-機能的システム」の考え方を打ち出した。パーソンズは生物学で発達した「解剖学」と「生理学」に着目したのである。まず、解剖学から「構造」の考え方を取り入れ、社会システムの構成要素間の関係のなかであまり変化しない部分を構造として捉えた。また、生理学から「機能」の考え方を取り入れ、ある一定の構造のもとで社会システムの維持のために貢献するはたらきを機能として捉えた。このように、パーソンズは、生物学から取り入れた構造と機能の概念を接合して、機能によって構造を説明しようとした。これが、構造-機能分析と呼ばれるものである。

(6) サイバネティクスとは、「舵手」(舵とり) を意味するギリシア語からつくられた言葉である。1948 年に、ウィーナーの『サイバネティクス: 動物と機械における制御と通信』が出版されている。ウィーナーは、従来まったく異なる存在として考えられてきた機械と生物を、情報伝達とそれを通じての制御の観点から、統一的に捉えようとした。このことは、有機体特有と考えられていた機能作用が、機械論の立場から理解できることを示したことになる。ここで注意が必要なのは、ここでいう機械というのは、すでに述べたような古典的な機械論とは異なるという点である。古典的な機械論では、機械とは熱機関のことであり、それがエネルギーの観点で捉えられていた (例えばハーヴェイは、心臓をポンプによって説明

した。また、デカルトは、心臓を一種の熱機関に見立てた)。これに対し、サイバネティクスでは、サーモスタットやミサイルなどの自己制御機械が取り上げられており、情報と制御の観点で機械が捉えられているのである。自己制御では、アウトプットの一部を情報として再びインプットすることで、目標値との差異を減らすように制御される。このフィードバック原理は、キャノンがホメオスタシス原理として提唱したものと同様のものである。

ウィーナーは最初サイバネティクスを社会科学に適用することに対して、「これはあまりに楽観的にすぎ、また科学の成果の本質の誤解によるものと思う」として消極的な態度をとっていたが、後に、サイバネティクスの社会科学への適用を積極的に評価するようになった。

1956 年には、アシュビーの『サイバネティクス入門』が出版される。アシュビーは、シャノンとウィーバーの情報理論をサイバネティクスに接合し、システム理論に高めるという貢献をしている。

(7) 一般システム論は、理論生物学者であるベルタランフィによって、従来有機体論が主張してきた「全体は部分に還元できない」という考え方を統一的に把握する立場として提唱された。それまでの機械論になかった有機体の特性、すなわち目的の概念や、秩序、組織性などを取り込んだ。その際、物理学が閉鎖システムを対象としていたのに対し、開放システムとして生物や社会を捉えるという視点に力点が置かれた。

一般システム理論では、諸科学の専門分化によって各ディシプリンの知識が断片化し、非効率になっている状況を解消するため、統一的な方法論によって統合することが目指された。諸科学を横断するような一般的な概念枠組みが目指されたため、抽象化したシステムを一般的・形式的に定式化しようとした。階層性と創発の概念などを明示している点が重要である。

(8) 社会科学におけるサイバネティクスないし一般システム理論の導入は、社会学、経済学、政治学などで行われたが、その効果は、それぞれの分野によって異なっている。

社会学の場合には、パーソンズの社会システム論の展開が先にあったために、サイバネティクスや一般システム理論はそれを補強するために導入された。しかし後に、脱パーソンズの試みの中で、積極的に展開されることになる。社会システム論にサイバネティクスの考え方を導入しようとしたのは、バックレイである。また、吉田民人は、サイバネティクスの枠組みを利用して、情報-資源処理システムを打ち立てている。

経済学においても、経済サイバネティクスという名のもと、サイバネティクスや一般システム理論の導入が試みられたが、主流派理論を補完するにとどまった。この当時、経済学は、静学的均衡の安定性条件に焦点をあてたヒックスの

『価値と資本』と、静学的・動学的安定性条件を数学的に定式化したサミュエルソンの『経済分析の基礎』が刊行されている。後に、ランゲの経済サイバネティクス論やボールディングの経済システム論を経て、コルナイの『反均衡の経済学』などにつながる。そのほか、日本における青木昌彦、村上泰亮、公文俊平などの経済システム論もある。

それまで確固とした理論枠組みをもたなかった政治学では、サイバネティクスや一般システム理論の考え方が、政治学の概念用具としてストレートに導入された。その研究を先駆けて行ったドイチュやイーストンらは、その後の政治学の主流の位置を占めることになる。

(9) システムにおける階層生成のメカニズムは、自己組織化を考えるうえで避けて通ることのできない問題である。第二世代のシステム論では、この階層生成が取り上げられる。すでに存在する階層の関係が問われるのではなく、無秩序の状態から秩序が形成されるメカニズムがその探究の対象なのである。

第一世代のシステム論では、構造が維持されるメカニズムに着目しているため、構造変動にいたる過程の理論化を断念せざるを得なかった。本来、逸脱のなかには、社会変動を引き起こす原動力となるようなものも含まれているので、パーソンズのいうように、「社会システムの変動の過程についての一般理論は、現在の知識の状態においては不可能である」ということになったのである。階層性をもった機能システムでは、上位の機能の生成について考えることができないということがある。それは、機能という概念が、システムの作動を考慮しなくてすむために、システムの作動の結果を特徴づけるものとして導入されたものだからである。

第二世代のシステム論には、プリゴジンらによる持続的な秩序の形成を行う「散逸構造」や、ハーケンの多数の要素の協同現象による自己組織化などがある。これらはどちらも、熱平衡から遠く離れた開放系におけるパターン形成を扱う点で共通点をもっている。また、アイゲンは、生成プロセスを持続させる「自己触媒」メカニズムやハイパーサイクルについて研究している。

(10) 熱力学の立場から階層生成を論じたのが、I.プリゴジンである (Nicolis and Prigogine, 1977; Prigogine, 1981)。古典的な熱力学では熱平衡状態にある系の性質を調べるが、プリゴジンは、平衡から大きく離れた非平衡状態において安定に維持される構造を調べ、それを「散逸構造」と呼んだ。

それまでは、ゆらぎや攪乱はシステムを均衡から逸脱させるものであり、それゆえに制御の対象とされてきたが、散逸構造の理論では、それらを新たな秩序が生まれる契機だと捉える。プリゴジンの言葉でいうならば、「「構造」は常に不安定性の結果として出現する」(『構造・安定性・ゆらぎ』) のである。このような散逸構造の形成や維持のためには、その非平衡状態を保ち続けるために、エ

ネルギーや物質の不断の流れが必要となる。つまり、開放系でなければならず、外界とのエネルギーや物質の交換による動的な秩序であるといえる。

散逸構造の例として代表的なのは、ベロウソフ-ザボチンスキー反応の化学反応系や、ベナール不安定性を示す熱対流系などの流体力学系、ロトカ・ボルテラの捕食・被食モデルの生態系などである。

(11) H. ハーケンも、非平衡熱力学と物性物理の立場から、物理、化学、生物系における相転移と自己組織化のモデルを示し、その一般原理を探ろうとする「シナジェティクス」という分野を創始した (Haken, 1978)。相転移の際に、要素が協調的な特殊な振る舞いをみせることを「協同現象」と呼んだ。協働現象の典型例としては、レーザー光の発生、ベロウソフ-ザボチンスキー反応、流体パターンなどがあげられる。

ベロウソフ-ザボチンスキー反応では、反応の最初の物質が、いくつかの化学反応の結果、産物として再度登場する。そのため再び同じ反応が進行し、生成プロセスの連鎖が循環的に繰り返される。この円環的生成プロセスは、動的に安定しており、一つの階層をなしているととらえることができる。

このシナジェティクスの考え方を社会学に適用し、複雑な社会現象を扱ったのは、Weidlich and Haag (1983) である。彼らは、人間社会が多数の個人で構成されており、互いに相互作用するほか、外部環境とも相互作用しているとし、物理・化学的なシステムとの共通点を示した。「社会も多数の構成員からなっていて、それぞれ異なった”態度 (attitude)”なり行動の”状態 (state)”をとる。さらに社会の全体としての変動は個々の構成員のとる態度なり行動状態の変化に関連している。そして、社会の全体としての状況の変化は、やはり適切な巨視変数例えば社会の構成員のグループの平均的態度を表すような量を導入することによって記述される。」(Weidlich and Haag, 1983)。このシナジェティクスの社会モデルは、社会シミュレーションの分野では、マルチレベルシミュレーションとして取り組まれている。

(12) 「相互作用」という言葉から明らかなように、ここでの定義には、複数の構成要素の存在が不可欠である。「複雑系」を掲げる文献のなかには、単にカオス現象がみられるシステムのことを複雑系と呼ぶものがあるが、本論文ではそのようなシステムを複雑系と捉えることはしない。その理由は、金子および津田 (1996) における次の文章が端的に述べてくれている。「われわれは、一見複雑に見えるものを何でも複雑系だというきわめて寛容な態度はとらない。複雑系だとあえて呼ばなければならない必然性が存在すると思えるからである」(金子および津田, 1996, p.1)。単にカオスが見られるということであれば、カオスシステムもしくは非線形システムと呼べばよいのであり、わざわざ複雑系という名称を用いる必要はない。例えば、ロジスティック方程式 $X_{n+1} = aX_n(1-X_n)$ は、$a > 3.5699456$

のときにカオス的な振舞いをするが、この式 (システム) を複雑系と呼ぶことは、私には適切だとは思えない。このようなシステムを対象としないという意味でも、本論文の定義では、「複雑系というシステムでは、その構成要素 (サブシステム) が相互作用する」という限定を置くことにする。ただし、この限定は、複雑系におけるカオスの重要性を必ずしも否定するものではないということに注意が必要である。例えば、構成要素がカオス生成メカニズムを備えたシステムの研究は、今後もさらに重要になると思われる。このようなカオス結合系は、それぞれの構成要素が内部状態とその変化ルールをもっているという点で、広義の複雑系である。このようなシステムの特性については、金子および津田 (1996) を参照のこと。

(13) 例えば、村上 (1997, p.16) は、「社会の基本単位である個人は決してアトムではありえない」と指摘している。なお、社会システム以外にも、内部状態をもつ構成要素として捉える視点が重要だといわれているのが、生命システムである。金子および池上 (1998, p.4) は、生命システムを記述しようとすると、「ある種の内部自由度あるいは外部からの決定不能性が要素に要求される」(金子および池上, 1998, p.4) とし、「内部状態を持った系の相互作用系の構図は最低限必要」(金子および池上, 1998, p.8) であると主張している。ここで、そのような要素を、物理学的な「要素」と区別する意味で「主体」と呼ぶ方がよいのではないかと指摘している点も興味深い。

(14) これに対し、他律とは、外部から他の原理が持ち込まれ、それによって動かされるということである。

(15) 広義と狭義の複雑系の理解を深めるために、分散人工知能やゲーム論に関わっている研究者の中で、コンピュータシミュレーションを使用した分析方法として普及してきてきたのが、エージェントベースシミュレーションである。エージェントベースシミュレーションは、分散人工知能におけるマルチエージェントモデルがベースとなり、複数のエージェントが集まった集団による行動の振舞いを分析するものである。個々の構成要素に単位を落とした分析により、経済学などが主に進めてきたマクロ分析的アプローチでは解き明かせなかった現象を観察することに主眼を置いている。マルチエージェントはもともとネットワークシステムの制御や分散処理の最適化に適用が期待されて研究が行われてきたが、分散的に配置されたエージェントが大局的には協調するという観点 (自律分散協調) が、複雑系のテーマとも類似していることから、社会分析への応用としても研究が進められた。

エージェントシミュレーションは複雑系のシステム観を具体化し分析するための手段と捉えることができるので、本論文では、主題的に取り上げることはせず、複雑系の概念の枠内でエージェントベースシミュレーションについて参照す

ることにする。

(16) 近代経済学の批判と再構築を行ったヴェブレンは、歴史的な累積的変化に着目する必要があると指摘した。「人々の総体に共通のものとして定着した思考の習慣」(『現代文明における社会の地位』1919 年) としての「制度」に着目し、それが進化すると考えた。この 2 つの指摘が、ヴェブレンが進化経済学の記念碑的位置を占めるとともに、制度派のはじまりの位置を占めている理由である。

(17) カーネギー学派の J・G・マーチと H・A・サイモンなどは、企業の役割構造をプログラム化されたルーティンのゆるやかな連結であると捉えた。しかし、その後、「意思決定問題を、プログラム化されたルール集合をもつ主体による、ルール自体の変更をも含む活動として捉え、これを分析するアプローチは意思決定論の中では発展してこなかった」(出口, 2000, p.28)。しかし、プログラムやルール集合としてシステムを捉えるということは、人工知能や人工生命の分野で研究されてきた。

(18) 訳は江頭 (2002, p.69) による

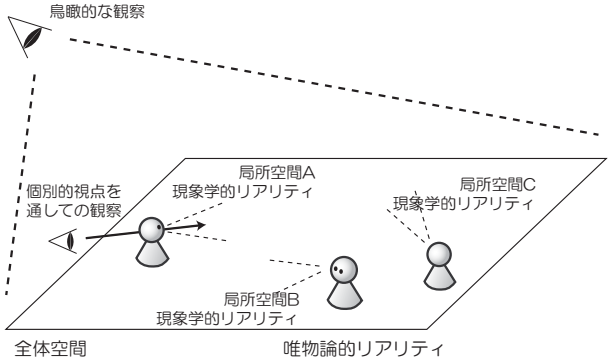
(19) ネルソンは、このような進化的な視点をもつに至った理由を次のように語っており、観察にもとづくものであることを強調している。「少なくとも私自身について述べれば、私は生物学に魅せられたために経済的変化の進化的理論に引きつけられたのではありません。むしろ、私は経済変化の過程を観察したことによって、そこで進行しているのが現存する複数の変異からの選択と新しい代替的変異の発生の組合せであるということを強く示唆されたのです」(Nelson, 1998, p.7)。

(20) 実は、そのルーティンの改良のメカニズムもまた、高次のルーティンによって行われることも多い (Nelson and Winter, 1982)。

(21) 社会のリアリティの問題については、これまで客観や主観に基づいた捉え方による様々な表現で試みられてきたが、橋爪 (2000) は「この試みを純化しようとすれば、唯物論と現象学のふたつの文体」に分けることができるという。唯物論 (materialism) は、世界について物質の根源性を主張し、それらが人間の意識の外に独立に存在すると考える。また、精神・意識などは物質にもとづいて成立するとされ、人間も社会も自然現象と同様の科学的な態度で解明できるとする。これに対し、現象学 (phenomenology) は、人間の経験にかかわらず世界そのものが客観的に存在すると考える自然的態度に対して、世界の現われを人間の意識の側に「還元」する。つまり、物理的・生理的過程を問題とするのではなく、私たちの経験そのものの内部に踏みとどまるのである。

　これらの二つの捉え方は互いに対立するものである。しかも、どちらも他方を排斥することもできない上、折衷したり包摂することもできない。どちらのリアリティも私たちの社会的現実の一面を捉えているように思われるのだが、純粋な

かたちでは唯物論も現象学も社会のリアリティを捉え切ることはできないのである。そこで橋爪 (1978) は、このような「唯物論的リアリティ」と「現象学的リアリティ」という社会的現実の相を「ダブル・リアリティ(二重の現実性)」としてみることを提案した。そしてこの二重のリアリティが社会の成立にとって根本的であるとし、「社会理論が解かねばならない問題の核心も、まさにそこにある」(*Ibid.*, p.2) とした。経済や政治、言語などの社会現象は、このふたつのリアリティが交錯するところで生起しているというわけである。

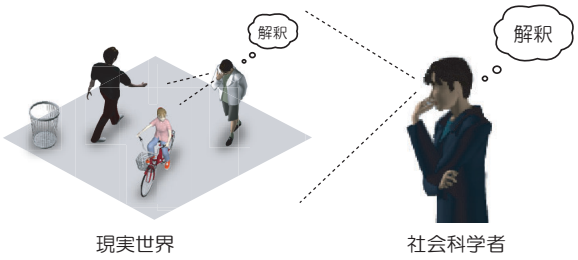


ダブル・リアリティの経済社会の模式図

個人は、それぞれの視野において周囲の空間についての了解を構成する。しかし、個人の感じる現象学的リアリティというのは、全体空間の単なる一部分なのではない。単に世界の局所を経験するというのではなく、個人的な「意味の世界」を構成しそれを経験するのである。また、この世界における個人は、決して全体空間を見ることはできない。せいぜいできることは、私たちが普段しているように、この全体空間を「社会」や「世界」などの言葉や概念によって「『全体』として見出し、個々の身体の近傍に開ける空間をその内部の『部分』として了解する」(若林, 1995, p.32) ことだけである。それぞれの主体が現象学的リアリティを個別的に経験するときには、その「まなざし」が主体に属しているということが重要である。それゆえ、各エージェントの感じる現象学的リアリティは必ずしも他者のものと整合的であるとは限らないということになる。複雑系としての社会・経済の記述は、このようなダブル・リアリティをもったモデルの構築を目指すことになる。

実はこの問題は、Giddens (1976) の指摘する社会科学における「二重の解釈学」の問題とも関係が深い。社会科学の対象は、「認識する主体」であり「自ら思考する主体」であるため、その認識や思考もモデル化しなければならないということになる。「オブジェクト・レベルにあった客体を、自ら思考する主体(サブジェクト)として描き出さなければならない」(Giddens, 2002, p.161) のである。それゆえ、対象自身の解釈とそれを研究する社会科学者の解釈の二重構造が存在し、「二重の

解釈学」になる。「社会科学では、自然科学とは異なり、この厄介な存在をなんとかきちんと理論化しなければならない」(今田, 1986, p.206) ということから、広義の複雑系のようなモデル化が、社会科学において不可欠となるのである。



社会科学における二重の解釈学

(22) 吉田民人は、自己組織性の概念を、物理科学の系譜、生物科学の系譜、社会科学の系譜といった３つのタイプに分類している (吉田, 1990)。物理科学の系譜とはプリゴジンらの散逸構造理論、生物科学の系譜とはマトゥラーナとヴァレラのオートポイエーシス理論、社会科学の系譜とは社会学的構造-機能理論や弁証法的発想である。このうち、後者の２つのタイプは、「プログラムによる制御」があるという点で、物理科学の自己組織性とは一線を画していると指摘している。また、Wallerstein (1996) は、社会科学の分析にとって複雑系の分析が重要であることを指摘しているなかで、このことを的確に述べている。「歴史上の社会システム [史的社会システム] は、多数の相互作用的な諸単位から構成され、重なりあった階層的な組織や構造ならびに複雑な時空行動の出現と進化によって特徴づけられる。さらにまた、固定的で顕微鏡的な相互作用メカニズムをもつ非線形動学システムによって表されるたぐいの複雑性に加えて、歴史上の社会システムは、その経験の結果として、内部的な適応と学習のできる個体的要素から構成されている。そこから、新しいレベルの複雑性 (進化生物学や生態学と共通する複雑性) がつけ加わるが、それは伝統的な物理系に関する非線形力学の複雑性を超えるものである。」(Wallerstein, 1996, 邦訳 p.121)。

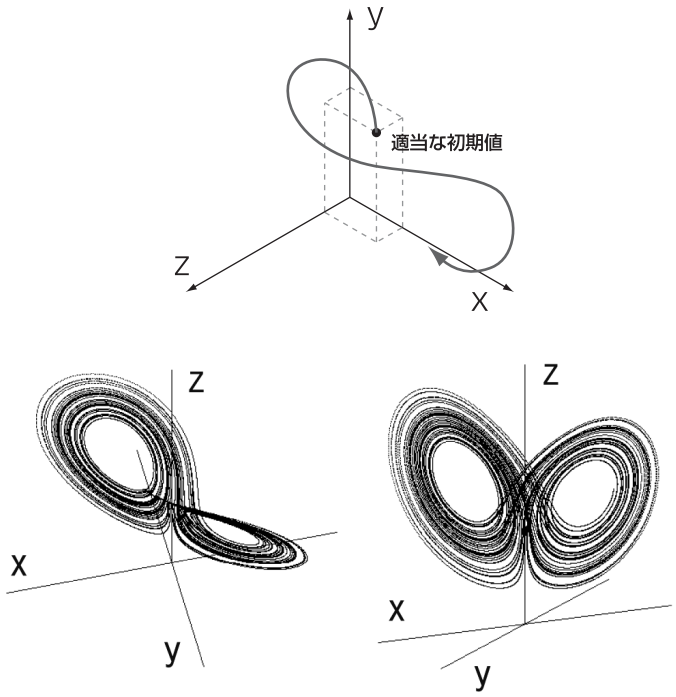
(23) 解析的には解けない問題に対するシミュレーションの適用の効果を表す典型的な例がカオスのシステムである。たとえば、次の非線形方程式は、すっきりとした形をしているにもかかわらず、解析的な一般解は知られていない。

$$\frac{dx}{dt} = -10x + 10y$$

$$\frac{dy}{dt} = 28x - y - xz$$

$$\frac{dz}{dt} = xy - \frac{8}{3}z$$

しかし、この式に対し、何か適当な初期値を与えて、相空間での x、y、z の変化の軌跡を追っていくことによって理解することができる。変数が x、y、z の三つなので、3 次元の相空間を考え、x、y、z の初期値を与え、得られて各変数の変化量から次のステップの x、y、z を求め、相空間にその軌跡を描いていく。このようにすることで、一般解を得るということとは別の仕方で、このシステムを理解することができる。



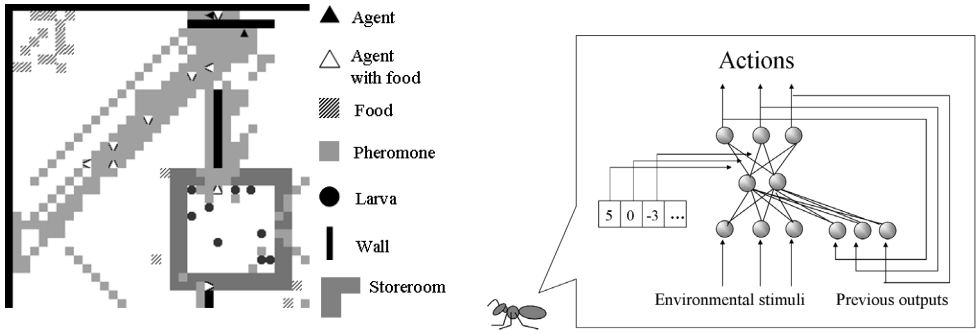
(24) 計算科学 (computational science) は、計算機科学 (computer science) とは異なることに注意が必要である。計算機科学では、コンピュータそのものに関する研究が行われるが、計算科学では、コンピュータは探求の対象ではなく手段となる。

(25) モデルの動きの振舞いには、変数化することが非常に困難なものがあり、そのような特徴を理解するためには、シミュレーションを行う以外に方法がない場合がある。例えば、岩村拓哉との共同研究 (岩村ほか, 1998; Iwamura et al., 1999) は、その点を理解するのにわかりやすい例となる。

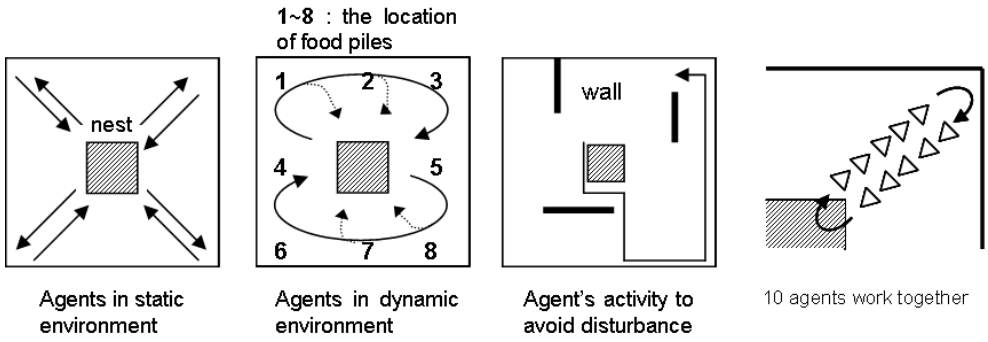
このモデルでは、2 次元セル上の蟻のコロニーにおける人工蟻の行動の進化をシミュレートしたものである。蟻の行動は、入力刺激を階層型のニューラルネットワークにインプットすることで決定される。入力には、自分の周囲のセルの状態や自分の内部状態、過去の出力の再入力などがある。出力には、「右を向く」「前に進む」「フェロモンを置く」などの動作が割り当てられており、これらの組み合せで行動が決定される。ニューラルネットワークの結合重みは、蟻の遺伝子

からマッピングされ、その世代内で変更されることはない (つまり、誤差逆伝播による学習などは行われない)。コロニー内のすべての蟻は、すべて同じ結合重みをもっている (同じ行動規則をもっている)。

蟻は、自らも砂糖を消費してエネルギーを回復させ、エネルギーがゼロになると死んでしまう。蟻は幼虫に砂糖を運ぶのだが、その結果と最終的に生き残っている蟻のエネルギー総量で適応度が決まる。進化は、蟻の個体レベルではなく、コロニーレベルで行われる。すなわち、複数のコロニーのなかで、適応度が高いコロニーの遺伝子を交叉して、新しいコロニーをつくることになる。この進化には、遺伝的アルゴリズムを用いて、突然変異も組み込む。

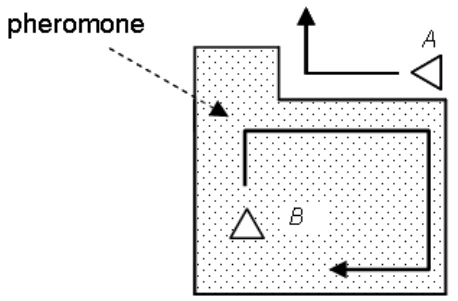


進化の過程を通じて、コロニーごとにさまざまな振舞いが観察される。例えば、セル空間の四隅に必ず砂糖の山が配置される静的環境においては、セル空間の四隅と中心との間を行き来する蟻のコロニーが生じる。また、どこに砂糖の山が配置されるかわからない動的環境においては、広い空間を探索する探索型の蟻のコロニーが出現している。壁をうまく回避しつつ広域の探索をする蟻のコロニーや、列を組んで砂糖の連続的な獲得を行う蟻のコロニーなども現れている。



また、興味深いことに、記号の意味の創発についても観察されている。この蟻は、環境にフェロモンを置くことと、そのフェロモンを感知することができるが、そのフェロモンが何を意味するのかは、モデルの設定では定めていない。進化の

過程において、その意味が規定され、それをコミュニケーションの手段に用いる蟻のコロニーが出現している。まず、第一のタイプは、すでに配置されたフェロモンを避けて移動するというものである。この場合は、「ここはすでに探索済みだから、ほかを探索してくれ」という意味で、フェロモンを用いていると考えることができる。これに対し、第二のタイプは、すでに配置されたフェロモンの中を移動するというものである。この場合には、「この周辺に砂糖がある」という意味で、フェロモンを用いていると考えることができる。というのは、フェロモンは時間が経つと消えてしまうため、フェロモンが残っているということは、最近まで蟻がその周辺にいたことを表している。また、蟻は砂糖がなければ死んでしまうので、蟻が生きているということは、その周辺に砂糖があることを表しているのである。このように、コロニーによって、まったく違う記号の意味を創発していることになる。コロニーによって、異なる進化を遂げることからも、歴史性をもっているともいえる。



以上のようなモデルの振舞いは、変数の変化によって把握できるような類のものではなく、実際に動きをみてみないとわからないものである。このような例からも、シミュレーションによる振舞いの観察の効果がわかるだろう。

(26) 一般に、欠陥除去率は「業界トップ」のソフトウェア開発会社で 95%、後進的な企業では 70%を越えることはめったにないというのが実情のようである (Jones, 1996)。

(27) これまでにも、エージェントベース経済モデルのシミュレーション研究のためのツールや言語が開発されてきた (Minar et al., 1996; Parker, 2001; RePast, ; Gulyas et al., 1999; 玉田, 2001)。しかし、これらはモデル部品の再利用を促進させるということにはつながっていない。モデル部品の再利用性を実現するためには、ドメインに特化したモデル化の指針と枠組みが必要であるが (岩村ほか, 1999; 井庭ほか, 2000)、これらのツールや言語では、分子相互作用や生態系などを含むマルチエージェントモデル全般を支援するという汎用性に重点が置かれている。また、分析・表示ツールの共有や、プログラム記述量の減少などを目的としているものもある。

(28) プログラム部品の再利用が行われていない大きな原因は、そもそも再利用可能なプログラム部品が存在しないということが挙げられる。あるプログラム部品が再利用可能であるためには、独立性をもったモジュールに切り分けられていなければならず、その上でそれが汎用性をもっている必要がある。そのため、再利用可能なコンポーネントを作成する側にも、それを利用する側にも負担が生じることになる。まず、プログラム部品を作成する側にとっては、通常の開発よりも初期開発にコストがかかるため、特別な理由がないかぎりは、再利用可能なプログラム部品を作成するというインセンティブは生じない (Jacobson et al. (1997) の推定によると、再利用可能なコンポーネントは通常より 1.5 倍から 3 倍のコストがかかり、そのコストを回収するために 3 回から 5 回使用される必要があるという)。そして再利用する側にとっても、モデルのプログラム部品の再利用には躊躇が伴う。なぜなら、再利用する場合には、自分の作成しているプログラムと既存のプログラム部品との間にプログラミング言語やプロトコルの互換性がなければならず、さらにはモデルの粒度や要素の分類基準なども同じである必要があるからである。

再利用可能なコンポーネントの必要性や重要性にもかかわらず、以上のような二重の負担が生じてしまうのである。それゆえ、単に再利用や蓄積の必要性を訴えるだけではなく、各研究者が再利用性をそれほど意識しなくても、自然と再利用性の高いプログラム部品を実現できるような仕組みが求められているのである。

(29) 多様性が本質的に重要となる対象は、「解析には複雑すぎ、統計にはあまりに組織的な領域」(Weinberg 1975) に属する。そのため、このような対象に対しては、構成要素の数を減らして複雑さに対応する力学的アプローチも、数多く存在する構成要素の平均値をとることで簡略化する統計学的アプローチも不適切となる。また、相互作用や学習を伴うシステムでは、相互作用の順序や状況が決定的に重要となるため、時間を不可逆な流れとして扱う必要がある。

(30) ”calculating”も同じ「計算」という訳語が当てられるため、混同しないように注意が必要である。”calculating”が数値計算や数学的な記号計算という狭い範囲の計算を指すのに対し、”computing”は、記号処理や論理演算、構造処理などを含む広い範囲の計算を指す。

(31) ここでいうモジュールとは、モデルの部品のことである。一般に、モジュールとは、建築・家具・機械・プログラムなどにおける機能単位のことであり、単独で一つのまとまった機能を持っているが、他のモジュールと組み合わせることで役割を果たすように設計されたものである。モジュール化されたシステムでは、その構成を変更する場合にも、関係のある一部のモジュールの変更だけで済ますことができる。また、モジュールごとに再利用することが可能となる。本稿の後半

に出てくる「コンポーネント」も、一種のモジュールである。なお、最近は、ビジネスの分野においてもモジュール化の重要性が注目されている (例えば、国領 1995, 青木・安藤 2002 など)。

(32) Penker and Eriksson (2000) や Marshall (1999) では、ビジネスモデルをオブジェクト指向によってモデル化するという試みが行われている。ただ残念なことに、これらの取り組みは、モデル記述の段階にとどまっており、本稿のようなシミュレーションによる分析につなげるということは行われていない。

(33) このような概念化の能力は、人間が世界を理解したり世界に働きかけたりする上で本質的なものであり、知覚、記憶、言語などの認知・思考活動の基礎であるといわれている (Roth and Frisby, 1986)。例えば、私たちは、ベンチや折りたたみの椅子、車椅子、背もたれのない横長の椅子などを見て、それぞれ形が異なっているにもかかわらず、これらをひとまとめに『椅子』として認識する。このような分類・概念能力がない場合には、「すべてのものは異なっているということが分かるだけ」(Martin and J.Odell, 1995) であり、見えたものをいちいち「初めてみるもの」として対処しなければならなくなってしまう。また、新しく出会った椅子を『椅子』であると理解できないため、その扱い方がわからず、うまく扱うことができなくなる。人間のもつ概念とオブジェクト指向の関係を明示的に論じたものに Martin and J.Odell (1995) がある。

(34) ここで示した図は、UML(統一モデル化言語) に従って書いたクラス図である。UML の記法とその意味については、付録 A にまとめたので、そちらを参照してほしい。

(35) 集約と関連は、実質的にはほとんど違いはない。集約には、集約リンクの連鎖が閉路を形成してはいけないという制約があるだけである。Rumbaugh et al. (1999) では、「集約と関連の区別は多くの場合、セマンティクスの違いというよりは、むしろ好みの問題」であるとしている。「集約は、集約側が本質的にその部分の総和であるという考えを伝えるもの」(Rumbaugh et al., 1999, p.265) である。これに対し、コンポジションは、「各部分が 1 つのオブジェクトに所有されていて、その各部分がそれを所有するオブジェクトから独立に存続しない」と (Rumbaugh et al., 1999, p.265) いう強い制約を表している。

(36) UML が誕生した経緯に軽く触れておくと、それは、オブジェクト指向の方法論は乱立状態が続いた 1990 年代前半に、方法論を統一しようとする動きから始まっている。ソフトウェア開発方法論 Booch 法を提唱していた G.Booch が、開発手法 OMT 法を提唱していた J.Rumbaugh とともに、これら 2 手法に登場する概念を組み合わせることに着手し、1995 年に Unified Method 0.8 を発表した。この流れに、ユースケースの概念を提唱した I.Jacobson が加わり、この 3 人 (“Three Amigos” と呼ばれる) を中心として、Unified Modeling Language 0.9 が

発表された。その後、他の企業なども加わって仕様が詰められ、1997 年に OMG (Object Management Group) で標準言語として採択された。当初、統一方法論を開発しようとしていたが、OPEN コンソーシアムによる OML(Open Modeling Language) の開発など、方法論の統一には、反発もあり、最終的には、方法論ではなく、モデリング言語を統一化するという決断をした。

(37) この点について認知科学では、「われわれは外界に構造 (structure) を付与する」(Roth and Frisby, 1986) と表現されているしている。また、科学哲学では、観察の「理論負荷性」(Hanson, 1958) として知られている。

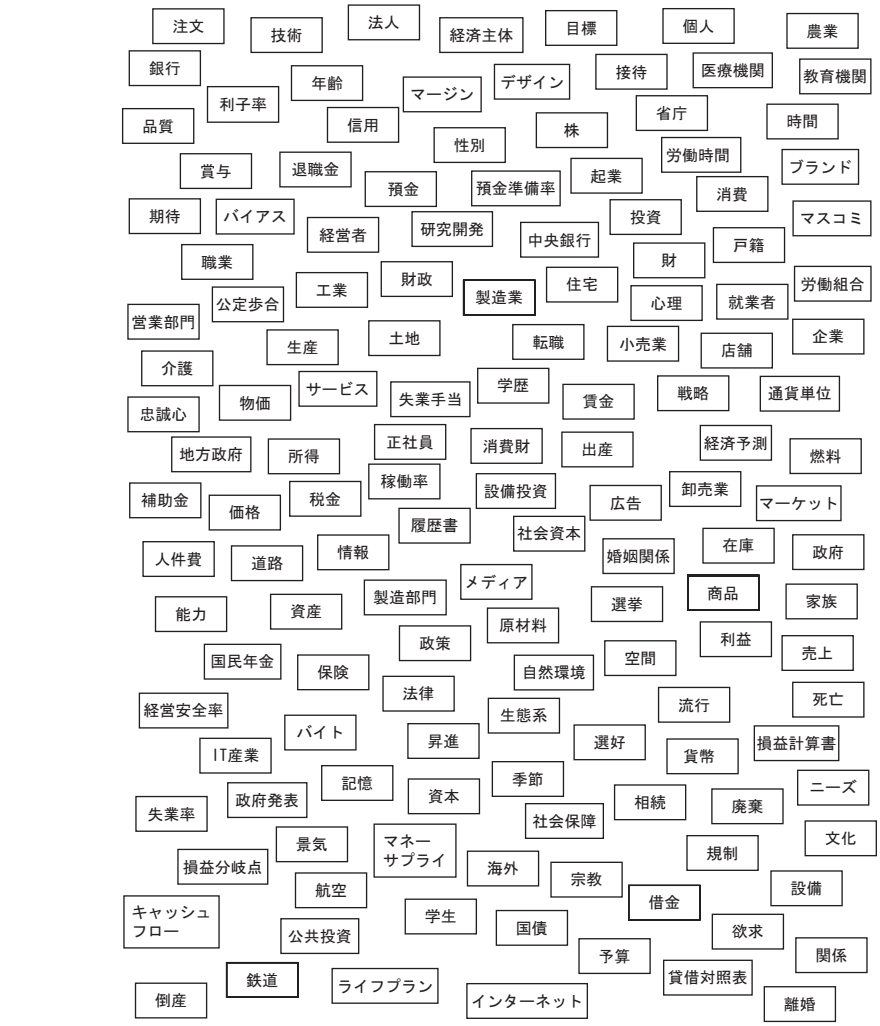
(38) それぞれの科学分野には、基本語句 (primitive terms) という「その体系のなかでは定義をうけず、逆に他の語句を定義する基礎となるもの」(村上, 1975) が存在し、それが分析的方法の土台として用いられることになる。基本語句の例としては、ユークリッド幾何学における点や線、古典力学における質点の位置、時間、質量、経済学における財などがそれにある (村上, 1975)。これらは理論的語句の基礎となり、他の理論的語句を説明するために用いられる。用いられる語句は、思考の単位として概念を固定するという役割も果すといわれている (Engel, 1993)。

(39) 組み合わせの規則は、語句と語句の関係性を規定するものであるが、複数の概念の関係性には、「全体と部分の関係などのいわゆる階層関係、または時間・空間・因果関係など非階層関係を形成」(仲本, 1999) している。

(40) 明確に規定されているコードがという共通の了解をいかにして実現するのか。「「コード」には、それを保証するような明確な規定が拘束力の強い形で含まれていなくてはならない」(池上, 1984) のであり、それには、次の 4 つのことが必要であるといわれている。第一にメッセージを作成する発信者が用いることのできる記号表現が明確に規定されていること、第二にそれぞれの記号表現に担わせうる情報が記号内容として明確に規定されていること、第三に規定されている記号表現と記号内容の対応が常に排他的に一対一であるということ、第四に記号表現の結合に関して、許容される結合がすべて規定されているということである (池上, 1984)。

(41) 本論文で取り上げる Boxed Economy Foundation Model は、バージョン 1.1 のものである (井庭ほか, 2001; Boxed Economy Project, 2003)。BEFM の定義は、次のようなプロセスで行った。2000 年 9 月に 3 日間に Boxed Economy Project のメンバーである井庭崇、中鉢欣秀、高部陽平、上橋賢一、松澤芳昭、廣兼賢治、津屋隆之介の 7 人でブレインストーミングを行い、重要な概念を洗い出した後、それらをグループ化し、抽象化していった。そこでの議論をもとに基礎モデルの基本的な構造が決定された。その後、田中潤一郎と北野里美が加わり、現実へ適用するための整理が行われ、最初のバージョンを発表した (井庭ほか, 2000; 井

庭ほか, 2001)。



ブレインストーミングであげられた経済社会に関係する言葉

2001 年 2 月から 6 月にかけて、田中潤一郎、上橋賢一、北野里美、津屋隆之介、廣兼賢治、山田悠、井庭崇によって、「個人」、「企業」、「金融機関」、「政府」の 4 部門を基礎モデルによってモデル化することを試みた。約 100 の行動がモデ

ル化された (Boxed Economy Project, 2001; 田中ほか, 2001; 上橋ほか, 2001)。また、基礎モデルフレームワークの設計にあたり、浅加浩太郎と海保研が加わり、技術的な面からの再検討が行われ、バージョン 1.0 としてまとめられた (Iba et al., 2001; Iba et al., 2001; 井庭ほか, 2002)。

要員計画行動	新人募集行動	新人採用行動	勤務記録処理行動	時間外労働命令行動	人材育成行動
人物評価行動	賃金決定行動	賃金支払行動	解雇予告通知行動	解雇行動	退職願受取行動
退職金計算行動	就職応募行動	就職雇用契約行動	勤め行動	勤務記録行動	労働行動
時間外労働命令受取行動	教育研修行動	給料明細受取行動	辞令受取行動	解雇予告通知受取行動	希望退職行動
退職行動	販売行動	商品販売行動	在庫確認行動	仕入行動	出荷行動
入荷行動	注文受取行動	仕入管理行動	在庫管理行動	受注行動	価格決定行動
価格戦略行動	市場地位分析行動	市場調査行動	長期戦略行動	資金調達行動	商品戦略行動
短期市場動向調査行動	生産管理行動	生産行動	公定歩合支払行動	利息支払行動	預金行動
融資行動	担保売却行動	税金支払行動	振込・引落行動	財務行動	口座照会処理行動
公定歩合受取行動	国債償還行動	国債引受行動	引出行動	準備預金預入・引出行動	口座移転行動
返済行動	買物行動	店舗選択行動	商品購買行動	受容行動	購買前代案評価行動
銀行振込行動	現金支払行動	現金受取行動	口座照会行動	経営戦略行動	行動計画実行行動
行動計画実行行動	移動行動	消費行動	生活保護解約行動	生活保護申込行動	納税行動
銀行に行く行動	預金引出行動	預金預入行動	家計記録行動	新聞発行行動	新聞紙面編集行動
新聞制作印刷行動	新聞発送配達行動	情報受信行動	情報提供行動	財受取行動	統計作成行動
公表統計作成行動	予算決定行動	企業税収行動	所得税収行動	国債交換行動	国債買取行動
国債発行行動	生活保護支給行動	生活保護受付行動	公共投資行動	手数料決定行動	地価査定行動
土地登記行動					

(42) 吉田 (1990) は情報の処理を、「情報貯蔵」、「情報伝達」、「(狭義の) 情報変換」の 3 つに分類しているが、これを BEFM における表現で表すと次のようになる。

情報貯蔵は、情報の時間変換ことを意味するが、それは「個体内貯蔵と個体外貯蔵に 2 分され、それぞれ記録・保存・再生の 3 段階から成り立っている」(吉田, 1990) という。基礎モデルでは、個体内貯蔵は、Information が Agent によって保持されることを意味しており、個体外貯蔵は、Agent のもつ Goods によって、Information が付随していることを表わしている。どちらの場合も「記録・保存・再生」は、Behavior によって行なわれることになる。

情報伝達は情報の空間変換のことを意味するが、「発信・送信・受信の 3 段階から成り立っている」(吉田, 1990) のであり、「個体間の情報伝達のみならず、個体内の情報伝達をも含意する」(吉田, 1990) という。「発信・送信・受信」は、Behavior によって行なわれることになり、その送受信には Channel を用いることになる。BEFM では、Agent 間の伝達であれ、Agent 内の伝達であれ、同じように Channel を介して行うことができる。

情報変換は情報の変容のことを意味するが、それはさらに「情報の担体変換」、「情報の記号変換」、「情報の意味変換」の 3 つに分けられるという。BEFM では、担体変換は、対象となる Information の担荷体である Goods が変化することで表現される。記号変換は、異なる形式の Information に内容を移し変えることで表現される。意味変換は、対象となる Information のもつ InformationContents の内容が変化することで表現される。

(43) BEFM に基づくモデルでは、一人のエージェントが複数の Behavior をもつことができ、それらの Behavior の連携も Channel を介した財や情報のやりとりで行う。

(44) このような方針が必要なのは、概念モデルでは、モデルに登場する要素はすべてBEFM 概念モデル・フレームワークのクラス (型) を「特化」するかたちで記述されるが、シミュレーションモデルの設計・実装では、その特化関係を必ずしも「継承」として実現するわけではないからである。

(45) 継承を用いた実装には、動的な変更ができないことや、多重継承ができないなどの限界があり、また、実装のしやすさや実行効率などの問題で、継承を用いるべきでないこともある。この問題を回避するための代表的な方法が、パワータイプを用いた設計である。

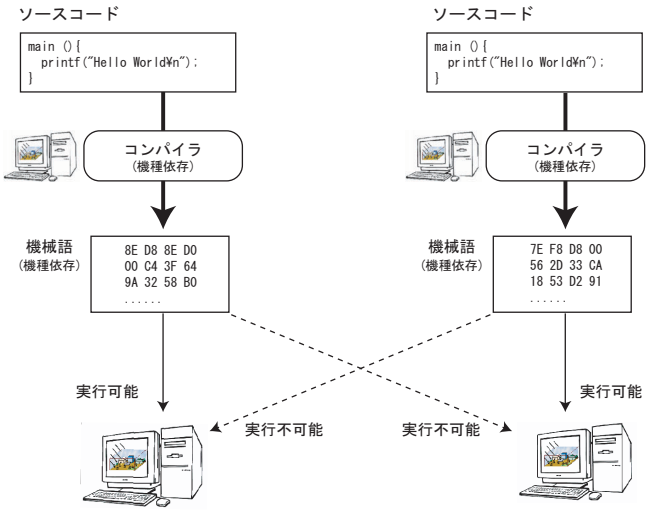
(46) BEFM フレームワークに基づいた Behavior の状態遷移を簡単に作成するために、「コンポーネントビルダー」を提供している。後述するように、モデル作成者は、GUI によって Behavior の状態遷移図を記述することで、Java プログラムのスケルトンを自動生成させることができる。

(47) InformationType も Information の一種であることから、エージェント間のやりとりでは InformationType をそのまま送受信することができるのである。

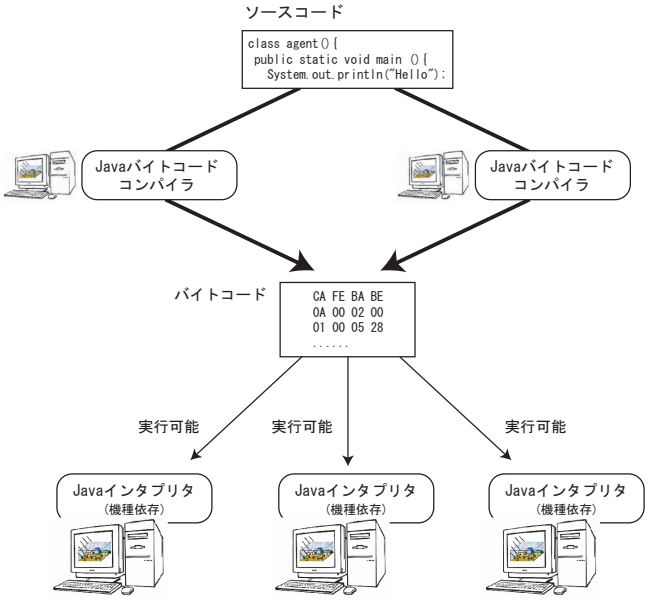
(48) 従来のようなソースコードレベルでの補助では、プログラミング技術や再コンパイルする必要があったが、コンポーネントによるプログラムの作成では、ソースコードに手を加えることなくモデルを作成・設定・変更できる。テスト済みの部分とそうでない部分を明確に区切ることができるので、品質管理上のリスクを回避できるという利点もある。

(49) これらコンポーネント開発者とシミュレーション実行者は、同一のチームメンバーであることもある。また、時間や空間を越えてお互いに知らない人同士であることもある。もちろん、コンポーネント開発者とシミュレーション実行者が同一人物であっても構わない。

(50) Java 仮想マシン (Java VM) 上であれば、オペレーティング・システムを問わず動作させることができる。つまり、BESP を利用するユーザがそれぞれ異なるコンピュータ環境を使っていたとしても、まったく同じように実行することができ、また、BESP のために作成されたコンポーネントも作成されたコンピュータ環境に依存しないため、他のコンピュータ環境でそのまま利用することができるのである。C 言語などでは、実行可能ファイルはマシン依存の機械語であるが、Java 言語は、その中間言語として機種に依存しないバイトコードに変換される。そのため、Java VM 上であれば、機種に関係なく、実行することが可能となる。



C のコンパイル



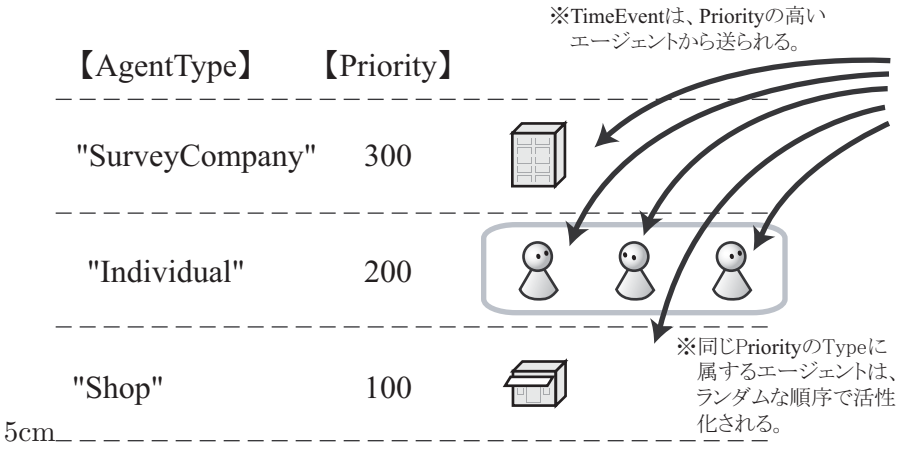
Java のコンパイル

(51) エージェントの優先順位は、Priority の数値の高い順に TimeEvent が配信される。

- 同じ優先順位のエージェントはステップ毎にシャッフルされ、ランダムな順番で時計信号を受け取る。
- 高い優先順位のエージェントは、低い優先順位のエージェントよりも若い順番で時計信号を受け取る。

異なる同優先順位の AgentType が存在する場合も、その全てのエージェントに対してシャッフルを行う。例えば、AgentType-A(優先順位 0), AgentType-B(優

先順位 0), AgentType-C(優先順位 10) のタイプを持つインスタンス Aa, Ab, Bc, Bd, Ce, Cf(小文字はインスタンス名) がいたとすると、まず、Ce, Cfがシャッフルされ、ランダムな順番で時計信号が送られる。次に、同優先順位の,Aa, Ab, Bc, Bd が一度にシャッフルされ、ランダムな順番で時計信号が送られる。



プライオリティによる優先順位づけ

(52) Component Builder は、統合開発環境である eclipse のプラグインとして作成されている。

(53) ここでのパターンの 3 つの部分というのは大枠での分類であり、具体的に何を記述するかという決まった形式はない。パターンの記述形式として有名なものには、Alexandrian 形式、Coplien 形式、GoF 形式、POSA 形式などがある。

Coplien 形式は、Alexander (1977) の Alexander 形式の本質的な部分を見出しとして採用したものであり、デザインパターンの記述に用いられている。**別名** (Alias)、**問題** (Problem)、**文脈** (Context)、**影響力** (Forces)、**解** (Solution)、**結果文脈** (Resulting Context)、**根拠** (Rationale)、**力学** (Dynamics)

GoF 形式は、Gang of Four(Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides の 4 人を指す) の Gamma et al. (1995) のなかで用いたデザインパターンのテンプレートである。基本的な項目は、**目的** (Intent)、**別名** (Also Known As)、**動機** (Motivation)、**適用の条件** (Applicability)、**構造** (Structure)、**構成要素** (Participants)、**協調関係** (Collaborations)、**結果** (Consequences)、**実装** (Implementation)、**サンプルコード** (Sample Code)、**事例** (Known Uses)、**関連するパターン** (Related Patterns) である。パターンの名前は、喚起的パターン名、名詞句名、意味のある喩え名で命名される。

POSA 形式は、Buschmann et al. (1996) で用いられている形式。基本的な項目は、**名前と別名** (Also Known As)、**例** (Example)、**前提** (Context)、**課題** (Problem)、**解決策** (Solution)、**挙動** (Dynamics)、**実装** (Implementation)、**補足**

(Example Resolved)、**バリエーション** (Variants)、**適用例** (Known Users)、**結果** (Consequence)、**参考** (See Also) である。

(54) アレグザンダーによると、このようなパターンの考え方は、建築家からは反発も多いという。これは、「パターン」という言葉が、「パターン化された」や「ワンパターン」というような使い方のように、ネガティブなイメージを伴うことによる。アレグザンダーは次のように言及している。「もの同士の関係の構造がデザイナーの優れた創造力からではなく、このような言語から生じるという考えは、建築家にとって不愉快きわまりないものです。彼らは、自分たちが建築や街やその一部を創造したのであり、それが彼らの豊富なイマジネーションのたまものであると思っているのです。」(Grabow, 1985, 邦訳 p.70)。同様の反発は、ソフトウェアパターンにおいてもみられるようである。「ソフトウェアーパターンを使うことで創造性が阻害される、あるいはソフトウェアパターンはあまりに即物的すぎて我慢がならない、という意見」(鈴木ほか, 2000) も根強いのだという。

このような現状に対し、「ルールが『制約』だと考えられている」(Grabow, 1985, 邦訳 p.71) ことが誤解のもとであるとし、パターンのもつ生成力の側面を強調する。「ルールを制約とみなすかぎり、創造の中心は独立に存在し、制約は単に創造を侵害するもののように感じられるでしょう。しかし一度ルールに生成力があると認めれば、創造の核心に迫ることができます。」(Grabow, 1985, 邦訳 p.71)。アレグザンダーのこのような議論では、しばしば自然言語との対比のため、チョムスキーによる文法の生成力の議論が取り上げられる。「英語のルールのおかげで創造的になれるのは、単語の無意味な組合せにいちいち思い悩まなくてすむからである。」(Alexander, 1979, 邦訳 p.170) と延べ、それと同じように、パターンは制限ではなく創造性の基盤であるという。「英語のルールのおかげで、膨大な数の無意味な文章に立ち入ることなく、意味をなすより少ない文章（といってもかなり多いが）に目を向けられる。その結果、人はそのより微妙な意味の違いに全精力を注げるのである。もし英語にルールがなければ、何時間も苦しんだあげくに一言もしゃべれないことになろう。」(Alexander, 1979, 邦訳 p.170)。

(55) パターンによる設計やコミュニケーションの支援の効果について、実務家からの定性的な評価が多いが (Coplien, 1996; Fraser et al., 1997)、変更容易性についての定量的な評価も若干行われている (Precht and B. Unger, 1997)。Precht and B. Unger (1997) によれば、パターンの知識があるグループの方が、知識のないグループに比べて、変更に必要な工数が 25%少なくて済み、品質の若干の向上が見られたという。

(56) モデル・パターンとは直接関係がないので、深入りはしないが、この先の文章で、アレグザンダーは、複雑系の考え方に通じる考えを展開していることは注目

に値する。「建物や町の『構造』の大部分が要素の関係のパタンで構成されている。(中略) 一見、この関係のパタンは要素から切り離されているかのように見える。(中略) なお詳しく見れば、これらの関係が付け足しどころか要素に不可欠のものであり、むしろ要素の一部でもあることがよく分かる。(中略) さらに詳しく見ていくと、このような見方ですらまだ十分に正確とは言えないことが分かる。つまり、関係が要素の属性であるばかりか、実は要素そのものが関係のパタンなのである。つまり、私たちが『要素』と考えるものの多くが、実はそれとその周辺のものとの関係のパタンの中に存在することを認めさえすればよい。いわゆる要素は神話にすぎず、事実、要素自体が関係のパタンに組み込まれているばかりか、それ自体が関係のパタン以外の何物でもないことが十分に認識できるのである。要するに、側廊を定義するには、身廊と内陣や東窓との関係のパタンが必要であり、しかもそれ自体が、その奥行、その間口、身廊との境にある柱、外部との境にある窓 … などとの関係のパタンでもある。」(Alexander, 1979, 邦訳 p.75-76)。この考え方は、要素が還元的に定義できない、すなわち、関係性の中で定義されるという複雑系の考え方に通じるものがある。

(57) アレグザンダーのパターンの考え方は、本論文で後に取り上げるメンタルモデルの考え方に通じるものがある。「すべての人びとがパタン・ランゲージを念頭に抱いている。あなたのパタン・ランゲージは、建設方法についてのあなたの知識の総計である。あなたの念頭にあるパタン・ランゲージは、隣人の頭にあるものとはわずかに異なっている。完全に同じものは二つとないが、多くのパタンやパタンの断片は共有される。設計行為に直面した人のとる行動は、その時点での自分の頭にあるパタン・ランゲージに支配される。もちろん、各人の記憶にあるパタン・ランゲージは、各人の体験の成長とともにつねに進化する。だが、彼に設計の必要が生じる特定の時点では、たまたまその時点で蓄積されたパタンに全面的に頼らなければならない。慎ましいものであれ、とてつもなく複雑なものであれ、彼の設計行為を完全に支配するのは、その時点で念頭にあるパタンとそれらの組合せで新たなデザインを形づくる能力である。」(Alexander, 1979, 邦訳 p.167)。

(58) 「インタラクティブシステム」の中には、本論文で提案する Boxed Economy Simulation Platform で採用している「Model-View-Controller」のパターンが含まれている。

(59) Boxed Economy Foundation Model や Boxed Economy Simulation Platform の設計においても、デザインパターンが数多く利用されている。例えば、Boxed Economy Foundation Model では、Behavior をコンポジションパターンによって、インスタンスレベルで合成している。また、一連の〜Type は、Type Object パターンであり、〜Manager は、Manager パターンの設計である。

(60) アンチパターンには、プロジェクトマネジメント以外のパターンも含まれている。アンチパターンは、開発のアンチパターン、アーキテクチャのアンチパターン、マネジメントのアンチパターンの 3 種類に分類できる。

(61) このモデルでは、そのとき存在するノードのなかから平等にリンクされるため、存在時間が長いノードは多くのリンクを得る機会が多く、リンクを多く持つようになる。

(62) べき乗分布とは、確率密度関数がべき乗の関数に従っている分布である。原理的には平均値が 0 で標準偏差が無限大という奇妙な分布であるが、非常に強い安定性がある (高安および高安, 2001)。ここで取り上げたネットワークのほかにも、多数の要素が相互作用するシステムは、「べき乗法則」に従うということが知られている。例えば、砂山における雪崩の規模と頻度 (Grumbacher, 1993)、地震の規模と頻度 (Johnston and Nava, 1985)、そして価格変動の規模と頻度の関係 (Bak, 1996) などは、どれもべき乗法則に従っていることが知られている。また、都市人口とその順位 (Simon, 1995; 武者, 1980)、単語の出現頻度とその順位 (Zipf, 1949)、企業所得とその累積分布 (Okuyama et al., 1999; 高安および高安, 2001) などがある。このように、多数の要素が相互作用するシステムでは、絶えず臨界状態に近い状態を保つ性質があるとして、この状態を自己組織的臨界状態と呼ばれている (Bak and Chen, 1991; Bak, 1994; Bak, 1996)。

(63) この優先的選択成長モデルは、古参のノードほど多くのリンクを持つ可能性がある。これに対し、結合度だけでなく、適応度にも比例するように改良された適応度モデル (Bianconi and Barabási, 2001) もある。

(64) 1970 年代に R. Axelrod は、繰り返し囚人のジレンマゲーム大会を行っている。「選手権の出場者は、経済学者、心理学、社会学、政治学および数学の各分野で活躍するゲーム理論の研究者であり、これらのべ一四人の応募作に、『でたらめ』(RANDOM) というプログラムを加えて、総当りのリーグ戦方式で競わせてみた」(Axelrod, 1984, p.iv)。そこでの勝者は、『しっぺ返し』(TIT FOR TAT) という単純な戦略であった。『しっぺ返し』は、最初は協調し、次からは相手が前回とった行動を真似するという戦略である。これは、実際の人間のつきあいにおいて、協調関係を引き出すものとして知られている (Oskamp, 1971; Wilson, 1971)。さらに、第一回の結果を踏まえて、第二回の大会も行われた。「今度は六つの国から六二人の応募者があった。その大半はコンピュータ愛好家であったが、中には進化生物学者、物理学者、コンピュータ・サイエンスの各教授、それと前回の参加者も五人含まれていた」(Axelrod, 1984, p.iv) という。結果は、またしても『しっぺ返し』が勝利している。

(65) 過去の自分の手や前回以前の相手の手は、直接的もしくは状態遷移に埋めこまれたかたちで、記憶することができる。

(66) コンテスト・シミュレーションの拡張には、本論文での拡張のほかにも、強い戦略の個体数がより多く生き残るという「生態学的シミュレーション」(Axelrod, 1984) や、戦略の遺伝子を遺伝的アルゴリズムによって改善させる「進化的シミュレーション」(Axelrod, 1997) などがある。

(67) 自分に勝った「エージェント」の中からランダムに選択するので、それらのエージェントが同じ戦略を採用する場合には、その分その戦略は採用されやすくなる。

(68) この閾値に関する仮定は、後に述べる進化的モデルにおいてはずされる。

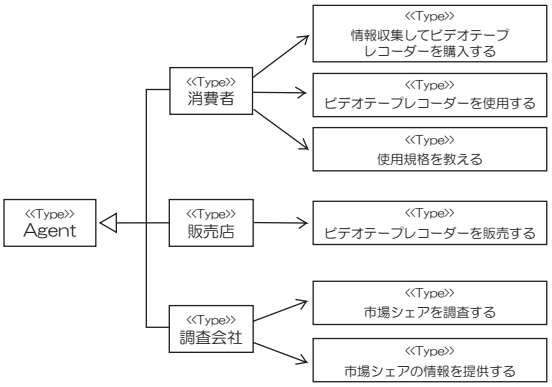
(69) ここでのモデル化では、安冨 (2000) のモデルにおける 1 ターンが、本モデルの 2 ステップになっている点に注意が必要である。

(70) 日本語では VTR(Video Tape Recorder) というのが一般的であるが、本論文では VCR に統一する。

(71) Beta 方式の方が技術的に優れていたというのが通説であるが (淺羽, 1995; 山田, 1997)、二方式が消費者にどう認知されていたのかについては疑問が残る。Klopfenstein (1989) は、アメリカにおけるコンシューマーレポートにおけるサーベイを取り上げ、各方式が画像品質で優れているといわれた回数は、Beta 方式が一度、VHS 方式が二度、大差なしが四度という結果であったと指摘している。VHS 方式の二時間録画が可能であるという点が勝敗を決めたという説 (吉井, 2000) や、VHS 方式で発売された特定分野のビデオソフトが原因で VHS 方式に傾いたという説もあり、必ずしも Beta の方が優れていたというわけではないようである。本論文では二方式に差異を設定せず、同じ条件での実験を行う。ここで提案するモデルの初期値の変更やモデルの拡張によって各方式に特徴を設定することが可能であるが、本論文では扱わず今後の課題とする。

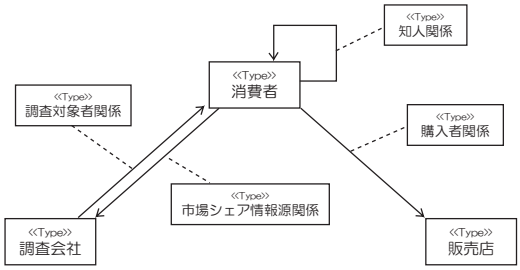
(72) 需要の相互依存性を提唱し分析した Rohlfs (1985) の言葉を借りるならば、家庭用 VCR の普及における前半期においては、市場全体におけるマーケットシェアが意思決定に影響を与える「一様な通話特性」よりも、密な関係をもった集団が内在し、それが影響を与える「非一様な通話特性」が強かったということになる。

(73) ここで、このモデルの作成プロセスにおける分析フェーズの図を掲載しておく。分析フェーズでは、モデル化しようとしている対象が、「どのようなものであるか」(What) を明確化するために、対象領域に登場する Agent、Information、Goods、Behavior、Relation をすべて洗い出して定義することから始める。まず最初にエージェントとその行動を明らかにする。



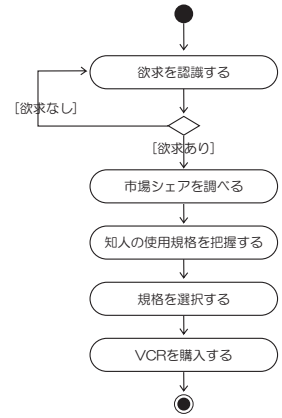
規格競争モデルにおけるエージェントとその行動

また、それらのエージェントの関係について記述する。



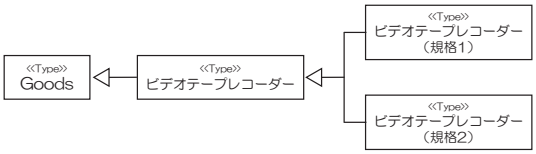
規格競争モデルにおけるエージェント間関係

そして、エージェントの行動のフローチャートを行動アクティビティ図として記述する。

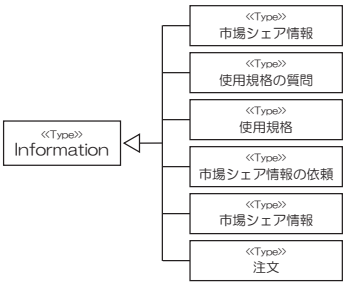


「情報収集してビデオテープレコーダーを購入する」行動のアクティビティ

この過程で、登場する財や情報も洗い出して行く。

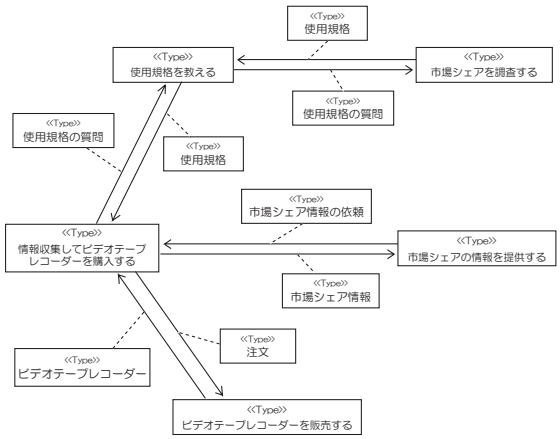


規格競争モデルにおける財



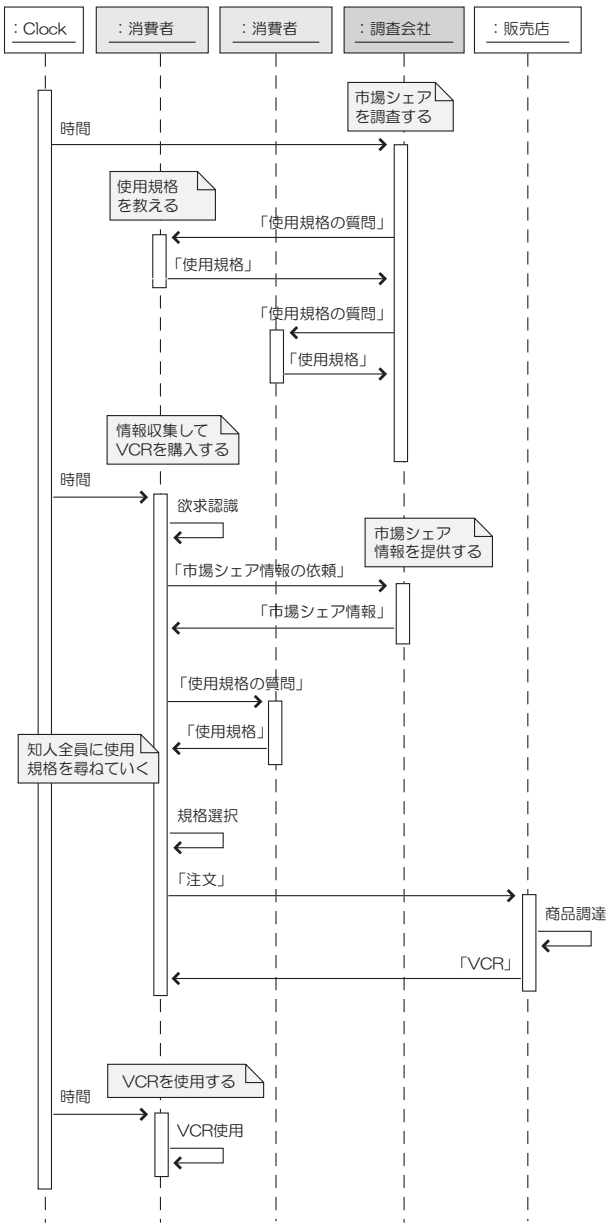
規格競争モデルにおける情報

これらの分析をもとに、各行動の間でどのような相互作用 (財や情報のやりとり) があるかを表現したものが、次の図である。



規格競争モデルにおける行動間のやりとり

また、エージェント間のやりとりの 1 つのシナリオを、時系列に示したものが、次の取引シーケンス図である。



規格競争モデルの 1 シナリオ

以上のような記述によってモデル要素の洗い出しと定義が終わるまで、このプロセスを繰り返していく。

(74) 実際には、大域的なマーケットシェアを完全に知ることはできないが、ここでは消費者は選択の際に流通・小売側の情報を通じて間接的に知ることができると仮定する。

(75) 欲求認識のモデルは本来であれば、広告やくちコミによって家庭用 VCR という製品の存在を知り、それに基づいてエージェント内部から欲求が発生するという

モデルが妥当であるが、本研究は家庭用 VCR 製品全体の普及ではなく、その普及の中で展開される規格競争を対象とするため、製品欲求の発生を外部から与えるという単純化をはかることにする。

(76) 現実の普及率のデータにおいて 1990 年が落ち込んでいるのは、その年からアンケートの中に「ビデオディスク」が加わったために引き起これた混乱が原因となった統計データ上の変調であることが知られている (電通総研, 1994)。

(77) 消費フェーズの拡張としては、セルやレンタルのビデオソフトの利用によって経験やソフトを蓄積するフェーズとすることが考えられる。

(78) 購買後代替案評価フェーズの拡張としては、消費経験により発生した所持方式に対する満足・不満足度を評価するモデルを用意し、買い換えや買い増し、くちコミなどの際に参照する知識を構築するフェーズとすることが考えられる。

(79) この規格競争モデルでは、各時間ステップにおける購入者数の集計が、Rogers の普及率 (Rogers, 1982) に合致するように、消費者エージェントの欲求認識を制御している。そのため、市場全体を眺め、普及率を制御する機能を実現するために、DiffusionControlFunction エージェントが導入された。

(80) ここで紹介するシミュレーション結果は、井庭ほか (2001) に基づいている。

(81) 参考までに本モデルのスケールを示しておく。本モデルでいう消費者エージェントは実際には個人というより世帯に近いと考えられるため、1985 年の世帯数との比で表すと 1024 : 37980 で約 1/37 のスケールのモデルということになる。

(82) エージェント間関係に対照性があるとする仮定は、ビデオソフトをやりとりする知人であれば相手も自分を知っているであろうという経験的理由によって妥当と思われる。

(83) 本論文では選好 P_{ij} をランダムに設定し、各方式に差を設けないが、各方式の特徴や性能の違いなどにより P_{ij} に偏りをつけ、その効果を調べる実験や、供給側のマーケティング活動などにより動的に変化する P_{ij} のモデルなどが可能である。

(84) なお、Arthur (1994) のモデルでは決定論的な選択モデルのため逆転現象は生じ得ないということも付記しておく。

(85) バラエティー・シーキングとは、低関与型の商品選択行動であり、購買時点においてさまざまな商品やサービスを探しまわる傾向のことである。

文 献

[清水, 1999] 清水 聰 (1999). **新しい消費者行動** (千倉書房).

[Alexander, 1977] C. Alexander (1977). *A Pattern Language* (Oxford University Press). クリストファー・アレグザンダー, **パタン・ランゲージ: 環境設計の手引**, 平田翰那 (訳), 鹿島出版会, 1984.

[Alexander, 1979] C. Alexander (1979). *The Timeless Way of Building* (Oxford University Press). クリストファー・アレグザンダー, **時を超えた建設の道**, 平田翰那 (訳), 鹿島出版会, 1993.

[Ambler, 1998] S. W. Ambler (1998). *Process Patterns: Building Large-Scale Systems Using Ibject Technology* (Cambridge University Press / SIGS Books).

[Ambler, 1999] S. W. Ambler (1999). *More Process Patterns: Delivering Large-Scale Systems Using Object Technology* (Cambridge University Press / SIGS Books).

[Arthur, 1994] B. W. Arthur (1994). Positive Feedbacks in the Economy in *Increasing Returns and Path Dependence in the Economy* (B. W. Arthur ed) pp. 1–12 (The University of Michigan Press).

[Arthur et al., 1996] W. B. Arthur, J. Holland, B. LeBaron, R. Palmer, and L. Tayler (1996). "Asset Pricing Under Endogenous Expectations in an Artificial Stock Market"; working paper 96-12-093, Santa Fe Institute.

[淺羽, 1995] 淺羽 茂 (1995). **競争と協力の戦略** (有斐閣).

[新および中野, 1984] 新 睦人, 中野 秀一郎 (1984). **社会学のあゆみ パート** *II:* **新しい社会学の展開** (有斐閣).

[Axelrod, 1984] R. Axelrod (1984). *The Evolution of Cooperation* (Basic Books). ロバート・アクセルロッド, **つきあい方の科学: バクテリアから国際関係まで**, 松田裕之 (訳), HBJ 出版局, 1987.

[Axelrod, 1997] R. M. Axelrod (1997). *The Complexity of Cooperation: Agent-Based Models of Competition and Collaboration* (Princeton University Press). ロバート・アクセルロッド, **対立と協調の科学: エージェント・ベース・モデルによる複雑系の解明**, 寺野隆雄 (監訳), ダイヤモンド社, 2003.

[Axtell, 2002] R. Axtell (2002). "Why Agents? On the Varied Motivations for Agent Computing in the Social Sciences"; Working PaperNo. 17, Center on Social and Economic Dynamics, The Brookings Institution.

[Bak, 1994] P. Bak (1994). Self-Organized Criticality: A Holistic View of Nature in *Complexity: Metaphors, Models, and Reality* (G. Cowan, D. Pines, and D. Melzer ed) vol. XIX (Addison-Wesley).

[Bak, 1996] P. Bak (1996). *how nature works* (Springer-Verlag).

[Bak and Chen, 1991] P. Bak and K. Chen (1991). Self-organized criticality. *Scientific American 264*, 46–53, Per Bak, Kan Chen, 「大地震や経済恐慌を説明する自己組織的臨界状態理論」, 山口昌哉, 木阪正史 (訳), **別冊日経サイエンス 複雑系がひらく世界**, 合原一幸 (編), 日経サイエンス社, 1997.

[Barabási, 2002] A.-L. Barabási (2002). *LINKED: The New Science of Networks, Perseus Book Group* (Perseus Book Group). アルバート＝ラズロ・バラバシ, **新ネットワーク思考: 世界のしくみを読み解く**, 青木薫 (訳), NHK 出版, 2002.

[Barabási et al., 1999] A.-L. Barabási, R. Albert, and H. Jeong (1999). Mean-field theory for scale-free random networks. *Physica A* (272), 173–187.

[Basu et al., 1998] N. Basu, R. J. Pryor, and T. Quint (1998). ASPEN: A microsimulation model of the economy. *Computational Economics 12*, 223–241.

[Beck, 1997] K. Beck (1997). *Smalltalk Best Practice Patterns* (Prentice Hall). ケント・ベック, **ケント・ベックの Smalltalk ベストプラクティス・パターン: シンプル・デザインへの宝石箱**, 梅澤真史, 小黒直樹, 皆川誠, 森島みどり (訳), ピアソン・エデュケーション, 2003.

[Beck and Cunningham, 1987] K. Beck and W. Cunningham (1987). Using Pattern Languages for Object-Oriented Programs in *OOPSLA-87 workshop on the Specification and Design for Object-Oriented Programming*.

[Berger and Luckmann, 1966] P. L. Berger and T. Luckmann (1966). *The Social Construction of Reality: A Treatise in the Sociology of Knowledge* (Doubleday & Company). ピーター・L・バーガー, トーマス・ルックマン, **現実の社会的構成: 知識社会学論考**, 新版, 山口節郎 (訳), 新曜社, 2003.

[Bianconi and Barabási, 2001] G. Bianconi and A.-L. Barabási (2001). Competition and multiscaling in evolving networks. *Europhysics Letters* (54), 436–442.

[Black, 1962] M. Black (1962). *Models and Metaphors: Studies in Language and Philosophy* (Cornell University Press).

[Boulding, 1985] K. E. Boulding (1985). *The World as a Total System* (Sage Publication).

[Boxed Economy Project, 2001] Boxed Economy Project (2001). 社会・経済シミュレーションのフロンティア — Boxed Economy Project — in *SFC Open Research Forum 2001*, 慶應義塾大学湘南藤沢キャンパス (SFC).

[Boxed Economy Project, 2003] Boxed Economy Project. *Boxed Economy Simulation Platform 1.1 Users Guide*. フジタ未来経営研究所 (2003).

[Brown et al., 1998] W. J. Brown, R. C. Malveau, H. W. M. III, and T. J. Mowbray (1998). *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis* (John Wiley & Sons). William J. Brown, Raphael C. Malveau, Hays W. McCormick III, Thomas J. Mowbray, **アンチパターン: ソフトウェア危篤患者の救出**, 岩谷宏 (訳), ソフトバンク, 1999.

[Brown et al., 1999] W. J. Brown, H. W. M. III, and S. W. Thomas (1999). *Antipatterns and Patterns in Software Configuration Management* (John Wiley & Sons). W. J. Brown, H.W. S. McCormick III, S. W. Thomas, **アンチパターン: ソフトウェア構成管理の悪夢**, 岩谷宏 (訳), ソフトバンク, 1999.

[Brown et al., 2000] W. J. Brown, H. W. M. III, S. W. Thomas, and H. W. McCormick (2000). *Antipatterns in Project Management* (John Wiley & Sons).

[Bruun, 1997] C. Bruun (1997). Agent-Based Keynesian Economics in *Simulating social phenomena* (R. Conte, R. Hegselmann, and P. Terna ed) pp. 279 – 285 (Springer-Verlag).

[Bruun, 2002] C. Bruun (2002). Prospect for an Economics Framework for Swarm in *Agent-Based Methods in Economics and Finance: Simulation in Swarm* (F. Luna and A. Perrone ed) (Kluwer Academic Publishers).

[Buckley, 1967] W. Buckley (1967). *Sociology and Modern Systems Theory* (Prentice-Hall). W. バックレイ, **一般社会システム論**, 新睦人, 中野秀一郎 (訳), 誠信書房, 1980.

[Buschmann et al., 1996] F. Buschmann, H. Rohnert, M. Stal, R. Meunier, and P. Sommerlad (1996). *Pattern-oriented Software Architecture: A System of Patterns* (Wiley). F. ブッシュマン, H. ローネルト, M. スタル, R. ムニエ, P. ゾンメルラード, **ソフトウェアアーキテクチャ: ソフトウェア開発のためのパターン体系**, 金沢典子, , 桜井麻里, 千葉寛之, 水野貴之, 関富登志 (訳), 近代科学社, 2000.

[Cannon, 1932] W. B. Cannon (1932). *The Wisdom of the Body* (W. W. Norton). W.B. キャノン, **からだの知恵: この不思議なはたらき**, 舘鄰, 舘澄江 (訳), 講談社, 1981.

[Casti, 1996] J. L. Casti (1996). *Would-Be Worlds: How Simulation Is Changing the Frontier of Science* (John Wiley & Sons). ジョン・キャスティ, **複雑系による科学革命**, 中村和幸 (訳), 講談社, 1997.

[Coad and Mayfield, 1999] P. Coad and M. Mayfield (1999). *Java Design: Building Better Apps & Applets* 2 edn. (Yourdon Press, Prentice Hall PTR).

[Coad et al., 1995] P. Coad, D. North, and M. Mayfield (1995). *Object Models: Strategies, Patterns, and Applications* (Prentice-Hall).

[Coplien, 1992] J. Coplien (1992). *Advanced C++ Programming Styles and Idioms* (Addison-Wesley). 安村通晃, 大谷浩司, 渦原茂 (訳), **C++ プログラミングの筋と定石**, トッパン, 1994.

[Coplien, 1995] J. Coplien (1995). A Generative Development-Process Pattern Language in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt ed) vol. 1 (Addison Wesley), PLoPD Editors(編), **プログラムデザインのためのパターン言語: Pattern Languages of Program Design 選集**, 細谷竜一, 中山裕子 (訳), ソフトバンクパブリッシング, 2001 所収.

[Coplien, 1996] J. Coplien (1996). Industrial Experience with Design Patterns in *Proc. ICSE '96* pp. 103–114, (IEEE CS Press).

[Cusumano et al., 1992] M. A. Cusumano, Y. Mylonadis, and R. S. Rosenbloom (1992). Strategic maneuvering and mass-market dynamics: The triumph of vhs over beta. *Business History Review 66*, 51 – 94.

[Dahl and Nygaard, 1966] O.-J. Dahl and K. Nygaard (1966). Simula — an algol-based simulation language. *Communication of the ACM 9*, 671 – 678.

[Dalle and Jullien, 2000] J. D. Dalle and N. Jullien (2000). Windows vs. Linux: Some Explorations into the Economics of Free Software in *Applications of Simulation to Social Sciences* pp. 399–416 (Hermes Science Publishing).

[出口, 2000] 出口 弘 (2000). **複雑系としての経済学：自律的エージェント集団の科学としての経済学を目指して** (日科技連).

[電通総研, 1994] 電通総研 . (1994). **情報メディア白書** *1994* (電通総研).

[Dugdale, 2000] J. Dugdale. An evaluation of seven software simulation tool for use in the social sciences. COSI Training on-line resources, http://www.irit.fr/COSI/training/evaluationoftools/Evaluation-Of-Simulation-Tools.htm (2000).

[江頭, 2002] 江頭 進 (2002). **進化経済学のすすめ: 「知識」から経済現象を読む** (講談社現代新書).

[Engel, 1993] G. Engel (1993). Unambiguity and ambiguity in *Unambiguity and ambiguity.*

[Engel et al., 1995] J. F. Engel, R. D. Blackwell, and P. W. Miniard (1995). *Consumer Behavior* 8 edn. (The Dryden Press).

[Epstein and Axtell, 1996] J. M. Epstein and R. Axtell (1996). *Growing Artificial Societies: Social Science from the Bottom Up* (Brookings Institution Press / The MIT Press). Joshua M. Epstein, Robert Axtell, **人工社会: 複雑系とマルチエージェント・シミュレーション**, 服部正太, 木村香代子 訳, 共立出版, 1999.

[Forrester, 1961] J. W. Forrester (1961). *Industrial Dynamics* (MIT Press). J.W. フォレスター, **インダストリアル・ダイナミックス**, 石田晴久, 小林秀雄 (訳), 紀伊国屋書店, 1971.

[Fowler, 1996] M. Fowler (1996). *Analysis Patterns: Reusable Object Models* (Addison-Wesley). マーチン・ファウラー, **アナリシスパターン: 再利用可能なオブジェクト モデル**, 堀内 一, 友野 晶夫, 児玉 公信, 大脇 文雄 (訳), 新装版, ピアソンエデュケーション, 2002.

[Frank and Cook, 1998] R. H. Frank and P. J. Cook (1998). *THE WINNER-TAKE-ALL SOCIETY* (Penguin Books USA Inc.).

[Fraser et al., 1997] S. Fraser, K. Beck, G. Booch, J. Coplien, R. Johnson, and B. Opdyke (1997). Beyond the Hype: Do Patterns and Frameworks Reduce Discovery Costs? in *Proc. ACM OOPSLA '97* pp. 342–344.

[藤村, 1999] 藤村 龍雄 (1999). 自然言語と普遍言語 [**科学／技術と言語** (岡田 節人, 佐藤 文隆, 竹内 啓, 長尾 眞, 中村 雄二郎, 村上 陽一郎, 吉川 弘之 編)] (岩波書店), 岩波講座 科学／技術と人間 10.

[Gamma, 1991] E. Gamma (1991). Ph.D. Thesis, Universität Zürich.

[Gamma et al., 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns : Elements of Reusable Object-Oriented Software* (Addison-Wesley). Erich Gamma, Richard Helm, Raplh Johnson, John Vlissides, **オブジェクト 指向における再利用のための デザインパターン**, 改訂版, 本位田真一, 吉田和樹 (監訳), ソフトバンクパブリッシング, 1999.

[Giddens, 1976] A. Giddens (1976). *New Rules of Sociological Method* (Centurym Hutchinson). アンソニー・ギデンズ, **社会学の新しい方法規準: 理解社会学の共感的批判**, 而立書房, 1987 (第 2 版 2000).

[Giddens, 2002] A. Giddens ed (2002). **情報技術と経済文化** (NTT 出版).

[Gilbert and Troitzsch, 1999] N. Gilbert and K. G. Troitzsch (1999). *Simulation for the Social Scientist* (Open University Press). N・ギルバート, K.G. トロイチュ, **社会シミュレーションの技法: 政治・経済・社会をめぐる思考技術のフロンティア**, 井庭崇, 岩村拓哉, 高部陽平 (訳), 日本評論社, 2003.

[Goldberg, 1989] D. E. Goldberg (1989). *Genetic Algorithms in Search, Optimization & Machine Learning* (Addison-Wesley).

[Grabow, 1985] S. Grabow (1985). *Christopher Alexander: The Search For A New Paradigm in Architecture* (Routledge Kegan & Paul). スティーブン・グラボー, **クリストファー・アレグザンダー: 建築の新しいパラダイムを求めて**, 吉田朗, 長塚正美, 辰野智子 (訳), 鹿島出版会, 1991.

[Grumbacher, 1993] S. K. e. a. Grumbacher (1993). Self-organized criticality: An experiment with sand piles. *Am. J. Phys. 61*.

[Gulyas et al., 1999] L. Gulyas, T. Kozsik, and J. B. Corliss (1999). The multi-agent modelling language and the model design interface. *Journal of Artificial Societies and Social Simulation 2*(3), http:/www.soc. surrey.ac. uk/ JASSS/2/3/8.html.

[Haken, 1978] H. Haken (1978). *Synergetics: An Introduction, Noequilirium Phase Transitions and Self-Organization in Physics, Chemistry and Biology* 2 edn. (Springer). H. ハーケン, **協同現象の数理: 物理、生物、化学的系における自律形成**, 牧島邦夫, 小森尚志 (訳), 東海大学出版会, 1980.

[浜嶋ほか, 1997] 浜嶋 朗, 竹内 郁郎, 石川 晃弘 編 (1997). **社会学小辞典** 新版 (有斐閣).

[Hanson, 1958] N. Hanson (1958). *Patterns of Discovery* (Cambridge University Press). N.R. ハンソン, **科学的発見のパターン**, 村上陽一郎 訳, 講談社学術文庫, 1986.

[Hanson, 1970] N. R. Hanson (1970). *Perception and Discovery: An Introduction to Scientific Inquiry* (Wadsworth Pub. Co.). ノーウッド・ラッセル ハンソン, **知覚と発見: 科学的探究の論理**, 上下巻, 復刊版, 紀伊国屋書店, 1982.

[橋爪, 1978] 橋爪 大三郎 (1978). "記号空間論" の基本視座. **ソシオロゴス** (2), 1–10, リーディングス日本の社会学 1:社会学理論 (塩原勉, 井上俊, 厚東洋輔 編, 東京大学出版会, 1997) に再録.

[橋爪, 2000] 橋爪 大三郎 (2000). **言語派社会学の原理** (洋泉社).

[服部ほか, 2000] 服部 正太, 玉田 正樹, 辺見 和晃, 桑原 敬幸 (2000). "ABS の概要と類似シミュレータとの比較"; Working Paper No.6, 新型シミュレータ開発プロジェクト, (http://hachibei.c.u-tokyo.ac.jp/users/yamakage/ntsp1.html).

[Hay, 1996] D. Hay (1996). *Data Model Patterns: Convention of Thought* (Dorset House).

[Hayek, 1945] F. A. Hayek (1945). The use of knowledge in society. *American Economic Review 35*(4), 519–530, F.A. ハイエク, 「社会における知識の利用」, **市場・知識・自由: 自由主義の経済思想**, 田中真晴, 田中秀夫 編訳, ミネルヴァ書房, 1986 所収, p.52-76.

[Hesse, 1966] M. B. Hesse (1966). *Models and Analogies in Science* (University of Notre Dame Press). M. ヘッセ, **科学・モデル・アナロジー**, 高田紀代志 (訳), 培風館, 1986.

[Hesse, 1980] M. B. Hesse (1980). *Revolutions and Reconstructions in the Philosophy of Science,* (Harvester Press). M. ヘッセ, **知の革命と再構成**, 村上陽一郎, 横山輝雄, 鬼頭秀一, 井山弘幸 (訳), サイエンス社, 1986.

[廣島, 1985] 廣島 康眞 (1985). Master Thesis, 慶應義塾大学 経営管理研究科.

[Hodgson, 1993] G. M. Hodgson (1993). *Economics and Evolution: Bringing Life Back into Economics* (University of Michigan Press). ジェフリー・M・ホジソン, **進化と経済学: 経済学に生命を取り戻す**, 西部忠 (監訳), 東洋経済新報社, 2003.

[Holland, 1986] J. H. Holland (1986). Escaping brittleness: The possibilities of a general purpose machine learning algorithm applied to parallel rule-based systems in *Machine Learning II* pp. 593–624.

[Iba, 1999] T. Iba (1999). Master Thesis, 慶應義塾大学 政策・メディア研究科.

[井庭および福原, 1998] 井庭 崇, 福原 義久 (1998). **複雑系入門*:* 知のフロンティアへの冒険** (NTT 出版).

[井庭ほか, 2002] 井庭 崇, 海保 研, 中鉢 欣秀, 上橋 賢一, 山田 悠 (2002). オブジェクト指向による社会のモデル化とフレームワーク [**第 *6* 回進化経済学会**].

[Iba et al., 2000] T. Iba, M. Hirokane, H. Kawakami, T. Y., and H. Takenaka (2000). Exploratory Model Building: Toward Agent-Based Economics [**第四回進化経済学会論集**] pp. 146–149.

[井庭ほか, 2000] 井庭 崇, 岩村 拓哉, 廣兼 賢治, 竹中 平蔵, 武藤 佳恭 (2000). “エージェントベース社会シミュレーションのためのフレームワークデザイン”; FIF Working PaperNo. 1, フジタ未来経営研究所.

[井庭ほか, 2000] 井庭 崇, 中鉢 欣秀, 高部 陽平, 廣兼 賢治, 津屋 隆之介, 田中 潤一郎, 上橋 賢一, 北野 里美, 高松 祐三, 石渡 元春, 竹中 平蔵 (2000). 箱庭経済シミュレーションの基礎モデル、および政策分析への可能性 [**政策分析ネットワーク 第 *2* 回年次研究大会「政策メッセ *2001*」研究発表要旨集**] p. 3.

[Iba et al., 2001] T. Iba, Y. Takabe, Y. Chubachi, J. Tanaka, K. Kamihashi, R. Tsuya, S. Kitano, M. Hirokane, and Y. Matsuzawa (2001). Boxed Economy Foundation Model: Toward Simulation Platform for Agent-Based Economic Simulations in *JSAI 2001 International Workshop on Agent-based Approaches in Economic and Social Complex Systems* pp. 186–193.

[Iba et al., 2001] T. Iba, Y. Takabe, Y. Chubachi, and Y. Takefuji (2001). Boxed Economy Simulation Platform and Foundation Model in *Workshop of Emergent Complexity of Artificial Markets, 4th International Conference on Computational Intelligence and Multimedia Applications* pp. 34–38.

[井庭ほか, 2001] 井庭 崇, 中鉢 欣秀, 高部 陽平, 田中 潤一郎, 上橋 賢一, 津屋 隆之介, 北野 里美, 廣兼 賢治 (2001). Boxed Economy の実現に向けて：エージェントベース経済シミュレーションのための基礎モデル [**情報処理学会研究報告** *ICS-123*] pp. 79–84.

[井庭ほか, 2001] 井庭 崇, 竹中 平蔵, 武藤 佳恭 (2001). 人工市場アプローチによる家庭用 vtr の規格競争シミュレーション. **情報処理学会論文誌*:* 数理モデル化と応用** *42*(SIG14 (TOM5)), 73–89.

[飯尾, 1995] 飯尾 要 (1995). 社会・経済システム論の歴史・現状・課題. **大阪経大論集** *45*(5), 17–46.

[石川および寺野, 2000] 石川 泰志, 寺野 隆雄 (2000). 分類子システムによるエージェントの共進化とマーケティングシミュレーション [**情報処理学会研究報告** *2000-ICS-119*] pp. 65–72.

[伊丹および伊丹研究室, 1989] 伊丹 敬之, 伊丹研究室 . (1989). **日本の *VTR* 産業*:* なぜ世界を制覇できたのか** (NTT 出版).

[岩村ほか, 1998] 岩村 拓哉, 井庭 崇, 武藤 佳恭 (1998). マルチエージェント社会における役割分担の生成: 蟻のコロニーにおける食糧運搬 [**情報処理学会第 *57* 回全国大会**].

[Iwamura et al., 1999] T. Iwamura, T. Iba, and Y. Takefuji (1999). Emergence of Cooperative Behavior by Simple Reactive Agents in *Joint Conference of Information Systems Analysis and Synthesis (ISAS) & The Third Conference of Systemics, Cybernetics and Informatics (SCI)*.

[岩村ほか, 1999] 岩村 拓哉, 廣兼 賢治, 井庭 崇, 竹中 平蔵, 武藤 佳恭 (1999). エージェントベース経済シミュレーションのためのフレームワークデザイン [**第 *8* 回マルチエージェントと 協調計算ワークショップ**].

[和泉および植田, 1999] 和泉 潔, 植田 一博 (1999). コンピュータの中の市場: 認知機構をもつエージェントからなる人工市場の構築とその評価. **認知科学** *6*(1), 31–43.

[Jacobson et al., 1997] I. Jacobson, M. Griss, and P. Jonsson (1997). *Software Reuse : Architecture, Process and Organization for Business Success* (ACM Press).

[Johnston and Nava, 1985] A. C. Johnston and S. J. Nava (1985). Recurrence rates and probability distribution estimates for the new madrid seismic zone. *J. Geophs. Res. B90*.

[Jones, 1996] C. Jones (1996). *Applied Software Measurement: Assuring Productivity and Quality* 2 edn. (The McGraw-Hill Companies). 鶴保征城, 富野壽 (監訳), **ソフトウェア開発の定量化手法**, 第 2 版, 共立出版, 1998.

[金子および池上, 1998] 金子 邦彦, 池上 高志 (1998). **複雑系の進化的シナリオ*:* 生命の発展様式** (朝倉書店).

[金子および津田, 1996] 金子 邦彦, 津田 一郎 (1996). **複雑系のカオス的シナリオ** (朝倉書店).

[片平, 1994] 片平 秀貴 (1994). **マーケティング・サイエンス** (東京大学出版会).

[片平および杉田, 1994] 片平 秀貴, 杉田 善弘 (1994). マーケティング・サイエンスの最近の動向：米国を中心として. **オペレーションズ・リサーチ** 178–188.

[Katz and Shapiro, 1985] M. L. Katz and C. Shapiro (1985). Network externalities, competition and compatibility. *American Economic Review 75*, 424–440.

[河田, 1989] 河田 雅圭 (1989). **進化論の見方** (紀伊国屋書店).

[河本, 1995] 河本 英夫 (1995). **オートポイエーシス*:* 第三世代システム** (青土社).

[経済企画庁, 1982 – 1996] 経済企画庁 . (1982 – 1996). **消費動向調査** (経済企画庁).

[富永, 1995] 富永 健一 (1995). **行為と社会システムの理論*:* 構造-機能-変動理論をめざして** (東京大学出版会).

[吉地および西部, 2000] 吉地 望, 西部 忠 (2000). 多層調整企業モデルによる複雑適応系シミュレーション [**第四回進化経済学会論集**] pp. 288–291.

[Klopfenstein, 1989] B. C. Klopfenstein (1989). The Diffusion of the VCR in the United States in *The VCR Age* (SAGE Publications).

[Knuth, 1985] D. E. Knuth (1985). Algorithmic thinking and mathematical thinking. *American Mathematical Monthly* 170–181, Donald E. Knuth, 「算法的思考と数学的思考」, **クヌース先生のプログラム論**, 有澤誠 (編), 共立出版, 1991 所収).

[Koenig, 1998] A. Koenig (1998). Patterns and Antipatterns in *The Patterns Handbook: Techniques, Strategies and Applications* (L. Rising ed) (Cambridge University Press / SIGS Books).

[厚東, 1991] 厚東 洋輔 (1991). **社会認識と創造力** (ハーベスト社).

[Kuhn, 1962] T. S. Kuhn (1962). *The Structure of Scientific Revolutions* (The University of Chicago Press). トーマス・クーン, **科学革命の構造**, みすず書房, 1971.

[公文, 1995] 公文 俊平 (1995). **情報文明論** (NTT 出版).

[Lakoff and Johnson, 1980] G. Lakoff and M. Johnson (1980). *Metaphors We Live By* (The University of Chicago Press). G. レイコフ, M. ジョンソン, **レトリックと人生**, 渡部昇一, 楠瀬淳三, 下谷和幸 (訳), 大修館書店, 1986.

[Langton et al., 1998] C. Langton, R. Burkhart, I. Lee, M. Daniels, and A. Lancaster. The swarm simulation system. http:// www.santafe.edu/ projects/ swarm (1998).

[Luce and Suppes, 1965] R. D. Luce and P. Suppes (1965). Utility and Subjective Probability in *Handbook of Mathematical Psychology* (R.D.Luce, R.R.Bush, and E.Galanter ed) (J.Wiley and Sons).

[Marshall, 1999] C. Marshall (1999). *Enterprise Modeling with UML: Designing Successful Software through Business Analysis* (Addison Wesley). クリス・マーシャル, **企業情報システムの一般モデル: UML によるビジネス分析と情報システムの設計**, 児玉公信 (訳), ピアソンエデュケーション, 2001.

[Martin and J.Odell, 1995] J. Martin and J. J.Odell (1995). *Object-Oriented Methods: A Foundation* (PTR Prentice Hall). ジェームズ・マーチン, ジェームズ・J・オデル, **オブジェクト指向方法序説： 基盤編**, 三菱ＣＣ研究会ＯＯタスクフォース (訳), トッパン, 1995.

[松澤ほか, 2003] 松澤 芳昭, 海保 研, 津屋 隆之介, 青山 希, 井庭 崇 (2003). エージェントベース経済シミュレーションの作成プロセス: Boxed Economy 基礎モデルに基づく分析と設計 [**第 7 回進化経済学会**].

[Minar et al., 1996] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The swarm simulation system:a toolkit for building multi-agent simulations. http://www.santafe.edu/ projects/ swarm/ overview/ overview.html (1996).

[水田, 2001] 水田 秀行 (2001). マルチエージェントシミュレーションとダイナミックオンラインオークション [**情報処理学会研究報告** *2001-ICS-123*] pp. 31–36.

[Mizuta and Yamagata, 2001] H. Mizuta and Y. Yamagata (2001). Agent-based Simulation for Economic and Environmental Studies in *Proceedings of the First International Workshop on Agent-based Approaches in Economic and Social Complex Systems* pp. 83–90.

[水田ほか, 2000] 水田 秀行, K. Steiglitz, E. Lirov (2000). マーケットの安定性と価格シグナル: エージェントによるシミュレーションと解析 [**情報処理学会研究報告** *2000-ICS-119*] pp. 51–56.

[森岡ほか, 1993] 森岡 清美, 塩原 勉, 本間 康平 編 (1993). **新社会学辞典** (有斐閣).

[村上, 1971] 村上 陽一郎 (1971). 物理学と数学の方法 [**現代科学の方法** (山内恭彦 編)] (日本放送出版協会).

[村上, 1975] 村上 泰亮 (1975). **産業社会の病理** (中央公論社).

[村上, 1994] 村上 泰亮 (1994). **反古典の政治経済学要綱**: **来世紀のための覚書** (中央公論社).

[村上, 1997] 村上 泰亮 (1997). 伝統的思考の宿酔から醒めるとき [**村上泰亮著作集** *1*] (中央公論社), 初出：週刊東洋経済 臨時増刊 <経済体制特集>, 1967.4, 25 頁.

[武者, 1980] 武者 利満 (1980). **ゆらぎの世界 自然界の** *1/f* **ゆらぎの不思議** (講談社).

[仲本, 1999] 仲本 秀四郎 (1999). 専門用語と人間 [**科学／技術と言語** (岡田 節人, 佐藤 文隆, 竹内啓, 長尾 眞, 中村 雄二郎, 村上 陽一郎, 吉川 弘之 編)] (岩波書店), 岩波講座科学／技術と人間 10.

[Nelson, 1998] R. Nelson (1998). 進化的経済理論の観点 [**進化経済学とは何か** (進化経済学会 編)] pp. 3–17 (有斐閣).

[Nelson and Winter, 1982] R. R. Nelson and S. G. Winter (1982). *An Evolutionary Theory of Economic Change* (Belknap Press of Harvard University Press).

[Nicolis and Prigogine, 1977] G. Nicolis and I. Prigogine (1977). *Self-Organization in Nonequilibrium Systems* (Wiley). G. ニコリス, I. プリゴジーヌ, **散逸構造: 自己秩序形成の物理学的基礎**, 小畠陽之助, 相沢洋二 (訳), 岩波書店, 1980.

[西部, 1997] 西部 邁 (1997). **ソシオ・エコノミックス** (中央公論社).

[西部, 2000] 西部 忠 (2000). “進化経済学の概念的・方法的基礎: メタファー・アナロジー・シミュレーション”; 経済学研究, 北海道大学, 第 50 巻, 第 1 号.

[新田, 1990] 新田 俊三 編 (1990). **社会システム論** (日本評論社).

[North, 2002] M. North (2002). ABMS Architectural Design in *Capturing Business Complexity with Agent-Based Modeling and Simulation: Useful, Usable and Used Techniques Workshop*, Argonne National Laboratory.

[Okuyama et al., 1999] K. Okuyama, M. Takayasu, and H. Takayasu (1999). Zipf’s law in income distribution of companies. *Physica A 269*, 125–131.

[Oskamp, 1971] S. Oskamp (1971). Effects of programmed strategies on cooperation in the prisoner’s dilemma and other mixed-motive games. *Journal of Conflict Resolution 15*, 225–229.

[Palmer et al., 1994] G. R. Palmer, B. W. Arthur, J. H. Holland, B. LeBaron, and P. Tayler (1994). Artificial economic life:a simple model of a stockmarket. *Physica D 75*.

[Pareto, 1916] V. Pareto (1916). *Trattato di sociologia generale* vol. 2 (Barbèra). V. パレート, **一般社会学提要**, 姫岡勤 (訳), 刀江書院.

[Parker, 2000] M. T. Parker (2000). Ascape: Abstracting Complexity in *Swarmfest 2000 Proceedings*.

[Parker, 2001] M. T. Parker (2001). What is ascape and why should you care? *Journal of Artificial Societies and Social Simulation 4*(1), http://www.soc.surrey.ac. uk/JASSS/4/1/5.html.

[Parsons, 1951] T. Parsons (1951). *The Social System* (Free Press). **社会体系論**, 佐藤勉 (訳), 青木書店, 1974.

[Parsons, 1954] T. Parsons (1954). *Essays in Sociological Theory* (Free Press).

[Penker and Eriksson, 2000] M. Penker and H.-E. Eriksson (2000). *Business Modeling with UML: Business Patterns at Wwork* (John Wiley & Sons). ハンス=エリク・エリクソン, マグヌス・ペンカー, **UML によるビジネスモデリング**, 鞍田友美, 本位田真一 (監訳), ソフトバンクパブリッシング, 2002.

[Precht and B. Unger, 1997] L. Precht and D. C. S. B. Unger (1997). “Applications of the First Controlled Experiment on the usefulness of Design Patterns: Detailed Description and Evaluation”; technical report WUCS-97-34, Washington University, St. Louise.

[Prigogine, 1981] I. Prigogine (1981). *From Being to Becoming* (Freeman). I. プリゴジン, **存在から発展へ: 物理科学における時間と多様性**, 小出昭一郎, 安孫子誠也 (訳), みすず書房, 1984.

[RePast,] RePast. Repast. *The University of Chicago's Social Science Research* , http://repast.sourceforage.net/.

[Robertson, 1971] T. S. Robertson (1971). *Innovative Behavior and Communication* (Holt, Rinehart and Winston, Inc.).

[Rochat and Cunningham, 1988] R. Rochat and W. Cunningham (1988). The Vision of the Pattern Language of Programs in *OOPSLA-88 workshop on the Specification and Design for Object-Oriented Programming*.

[Rogers, 1982] E. M. Rogers (1982). *Diffusion of Innovation* 3 edn. (The Free Press).

[Rohlfs, 1985] J. Rohlfs (1985). A theory of interdependent demand for a communications service. *Bell Journal of Economics and Management Science 5*, 16–37.

[Roth and Frisby, 1986] I. Roth and J. P. Frisby (1986). *Perception and Representation: A Cognitive Approach: Open Guide to Psychology* (Open University Press). I. ロス, J.P. フリスビー, **知覚と表象**, 第 2 版, 海文堂, 1991.

[Rumbaugh et al., 1999] J. Rumbaugh, I. Jacobson, and G. Booch (1999). *The Unified Modeling Language Reference Manual* (Addision Wesley Longman). ジェームズ・ランボー, イヴァー・ヤコブソン, グラディ・ブーチ, **UML リフ ァレンスマニュアル**, 石塚圭樹 (監訳), 日本ラショナルソフトウェア株式会社 (訳), ピアソン・エデュケーション.

[佐藤ほか, 2001] 佐藤 浩, 久保 正男, 生天目 章 (2001). マルチエージェントによる先物取引コンテスト: Pre U-Mart 2000 実施報告 [**情報処理学会研究報告** *2001-ICS-123*] pp. 67–72.

[Schumpeter, 1915] J. A. Schumpeter (1915). *Vergangenheit und Zukunft der Sozialwissenschaften* (Verlag von Dunker & Humbolt). シュムペーター, 「社会科学の過去と未来」, 谷嶋喬四郎 (訳), **社会科学の過去と未来**, 玉野井芳郎 (監修), ダイヤモンド社, 1972.

[瀬戸, 1995] 瀬戸 賢一 (1995). **空間とレトリック** (海鳴社).

[Shaw et al., 1996] M. Shaw, G. David, and D. Garlan (1996). *Software Architecture: Pespectives on an Emerging Discipline* (Prentice Hall).

[進化経済学会, 1998] 進化経済学会 . 編 (1998). **進化経済学とは何か** (有斐閣).

[有賀ほか, 2000] 有賀 裕二, 塩沢 由典, 八木 紀一郎 (2000). 「ゲネシス進化経済学」刊行にあたって [**方法としての進化*: ゲネシス進化経済学*** (進化経済学会, 塩沢由典 編)] (シュプリンガー・フェアラーク東京).

[塩野谷, 1998] 塩野谷 祐一 (1998). **シュンペーターの経済観*: レトリックの経済学*** (岩波書店).

[塩沢, 1998] 塩沢 由典 (1998). 複雑系と進化 [**進化経済学とは何か** (進化経済学会 編)] pp. 99–119 (有斐閣).

[塩沢, 2000] 塩沢 由典 (2000). システム・アプローチに欠けるもの: 経済学における反省. **社会・経済システム** *19*, 55–67.

[公文, 1978] 公文 俊平 (1978). **社会システム論***:* **社会科学総合化の試み** (日本経済新聞社).

[Simon, 1995] H. Simon (1995). On a class of skew distribution functions. *Biometrica* .

[進化経済学会および塩沢, 2000] 進化経済学会 ., 塩沢 由典 編 (2000). **方法としての進化***:* **ゲネシス進化経済学** (シュプリンガー・フェアラーク東京).

[Sorokin, 1928] P. Sorokin (1928). *Contemporary Sociological Theories* (Harper and Row).

[Sutton and Barto, 1998] R. S. Sutton and A. G. Barto (1998). *Reinforcement Learning* (The MIT Press).

[鈴木ほか, 2000] 鈴木 純一, 田中 祐, 長瀬 嘉秀, 松田 亮一 (2000). **ソフトウェアパターン再考***:* **パターン発祥から今後の展望まで** (日科技連).

[中谷および青山, 1999] 中谷 多哉子, 青山 幹雄 (1999). ソフトウェアパターン [*bit* **別冊 ソフトウェアパターン** (中谷多哉子, 青山 幹雄, 佐藤 啓太 編)] (共立出版).

[田子, 1998] 田子 精男 (1998). 計算の、計算による、計算のための科学 [**シミュレーション科学への招待***:* **コンピューターによる新しい科学**] (日経サイエンス社), 別冊日経サイエンス 130.

[今田, 1986] 今田 高俊 (1986). **自己組織性***:* **社会理論の復活** (創文社).

[高安および高安, 2001] 高安 秀樹, 高安 美佐子 (2001). **エコノフィジックス***:* **市場に潜む物理法則** (日本経済新聞社).

[玉田, 2001] 玉田 正樹 (2001). 日本発マルチエージェント・シミュレータのご紹介 [**計測自動制御学会システム工学部会・知能工学部会共催研究会**].

[田中ほか, 2001] 田中 潤一郎, 浅加 浩太郎, 中鉢 欣秀, 井庭 崇 (2001). Boxed Economy 基礎モデルによる消費者行動のモデル化 [**計測自動制御学会 システム工学部会・知能工学部会共催研究会**].

[谷口ほか, 2001] 谷口 憲, 倉橋 節也, 寺野 隆雄 (2001). エージェントに基づくサプライチェーンモデル [**情報処理学会研究報告** *2001-ICS-123*] pp. 109–114.

[Thiel, 1969] H. Thiel (1969). A multinomial extension of the linear logit model. *International Economic Review 10*, 251–259.

[徳安, 2000] 徳安 彰 (2000). 社会システム理論の現在. **社会・経済システム** *19*, 18–27.

[富永, 1993] 富永 健一 (1993). **現代の社会科学者***:* **現代社会科学における実証主義と理念主義** (講談社学術文庫).

[上橋ほか, 2001] 上橋 賢一, 松澤 芳昭, 井庭 崇 (2001). Boxed Economy 基礎モデルによる流通過程のモデル化と分析可能性 [**第** *5* **回進化経済学会**].

[若林, 1995] 若林 幹夫 (1995). **地図の想像力** (講談社選書メチエ).

[Wallerstein, 1996] I. Wallerstein (1996). *Open the Social Sciences: Report of the Gulbenkian Commission on the Restructuring of the Social Sciences* (Mestizo Spaces, Stanford University Press). イマニュエル・ウォーラーステイン + グルベンキアン委員会, **社会科学をひらく**, 山田鋭夫 (訳), 藤原書店, 1996.

[Wallingford, 1998] E. Wallingford. Elementary patterns and their role in instruction workshop. ChiliPLoP'98 (1998).

[Weidlich and Haag, 1983] W. Weidlich and G. Haag (1983). *Concepts and Models of a Quantitative Sociology: The Dynamics of Interactiting Populations* (Springer Verlag). W. ワイド リッヒ, G. ハーグ, **社会学の数学モデル**, 寺本英, 中島久男, 重定南奈子 (訳), 東海大学出版会, 1986.

[William, 1925] J. William (1925). *The Philosophy of William James: Drawn from His Own Works* (The Modern Library). Introduction by H.M.Kallen.

[Wilson, 1971] W. Wilson (1971). Reciprocation and other techniques for inducing cooperation in the prisoner's dilemma game. *Journal of Conflict Resolution 15*, 167–195.

[Wilson, 1990] B. Wilson (1990). *Systems: Concepts, Methodologies, and Applications* 2 edn. (John Wiley & Sons). Brian Wilson, **システム仕様の分析学：ソフトシステム方法論**, 根来龍之 (訳), 共立出版, 1996.

[Witt, 1997] U. Witt (1997). Self-organization and economics - what is new? *Structural Change and Economic Dynamics* 489–508.

[Witt, 1998] U. Witt (1998). 経済学とダーウィニズム [**進化経済学とは何か** (進化経済学会編)] pp. 19–44 (有斐閣).

[Wright, 1931] S. Wright (1931). Evolution in mendelian populations. *Genetics 16*, 97–159.

[山田, 1993] 山田 英夫 (1993). **競争優位の /規格/ 戦略** (ダイヤモンド社).

[山田, 1997] 山田 英夫 (1997). **デファクト・スタンダード** (日本経済新聞社).

[山本ほか, 2001] 山本 隆人, 川村 秀憲, 山本 雅人, 大内 東, 車谷 浩一 (2001). X-Economy を用いた人工市場における取引エージェントの設計 [**計測自動制御学会 システム工学部会・知能工学部会共催研究会**] pp. 45–48.

[安冨, 2000] 安冨 歩 (2000). **貨幣の複雑性*:* 生成と崩壊の理論** (創文社).

[横田および小林, 2001] 横田 毅, 小林 康弘 (2001). 人工市場を用いた株価シミュレーションシステムの開発 [**計測自動制御学会 システム工学部会・知能工学部会共催研究会**] pp. 55–60.

[吉井, 2000] 吉井 博明 (2000). **情報のエコロジー*:* 情報社会のダイナミズム** (北樹出版).

[吉田, 1990] 吉田 民人 (1990). **自己組織性の情報科学** (新曜社).

[Zipf, 1949] G. K. Zipf (1949). *Human Behavior and the Principle of Least Effort* (Addison-Wesley).

[池上, 1984] 池上 嘉彦 (1984). **記号論への招待** (岩波新書).

[池上ほか, 1994] 池上 嘉彦, 山中 桂一, 唐須 教光 (1994). **文化記号論** (講談社学術文庫).

[Object Management Group, 2000] Object Management Group (2000). *OMG Unified Modeling Language Specification* (Object Management Group). **UML 仕様書**, OMG Japan SIG 翻訳委員会 UML 作業部会 (訳), アスキー, 2001.

付 録A　UML(統一モデル化言語)の表記について

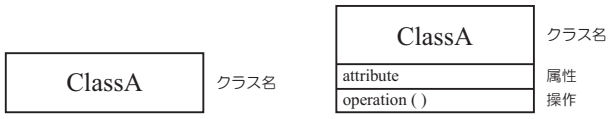
本論文では、モデル・フレームワークやシミュレーション・プラットフォームの構造、そしてシミュレーションモデルを記述するために、UML(Unified Modeling Language: 統一モデル化言語) を用いている。ここでは、この UML の表記法について、本論文に関係する部分のみを説明することにしたい (ここで説明する以外の項目や詳細については、Rumbaugh et al. (1999) などを参照してほしい)。以下では、クラス図、オブジェクト図、シーケンス図、ステートチャート図の順に説明する。

A.1　クラス図の記法

クラス図 (class diagram) は、おもにクラスの内容とそれらの間の関係を表すための静的ビューの図である。

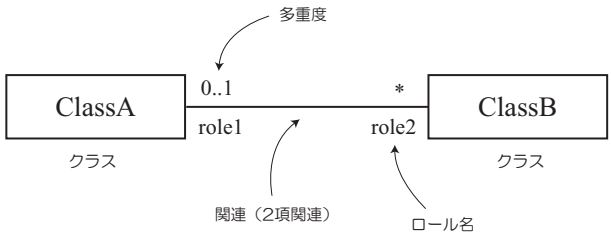
クラス

クラス図では、矩形で「クラス」が記述される。その矩形の中に「クラス名」が明示される。また、矩形の中に区画を区切って、そのクラスの「属性」と「操作」を示すこともできる。その場合には、上の区画にクラス名、中の区画に属性、下の区画に操作が記述される。これらの属性や操作は、どちらか一方を省略することもできる。

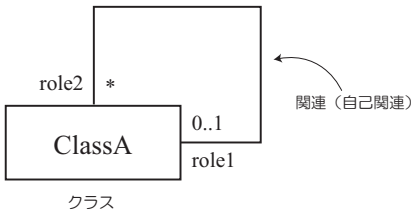


関連

クラス図において、クラスの結びつきを表す「関連」は、クラスの矩形の間の実線で記述する。関連がクラスに接続している部分には、「ロール名」や「多重度」を書くことができる。多重度とは、一方のクラスの１つのインスタンスに対して、もう一方のクラスのインスタンスが、いくつリンクを持つかを表す数である。

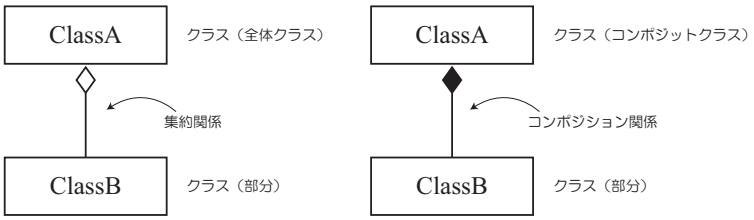


あるクラスから出ている関連がそのクラス自身に向けられている場合には、そのオブジェクトがそのオブジェクト自身にリンクされることもあれば、同じクラスのインスタンスである別々のオブジェクトがリンクされることもある。



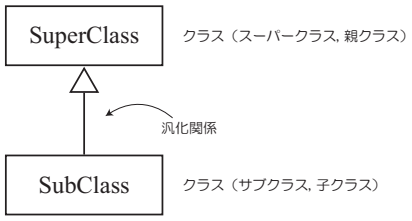
集約とコンポジション

クラス間関係において部分/全体関係を表す「集約」関係は、全体側のクラスに結び付けられる端に白抜きの菱形をつけた実線で表す。また、「コンポジション」(複合化) 関係は、塗りつぶした菱形をコンポジット側の端につけた実線で表す。

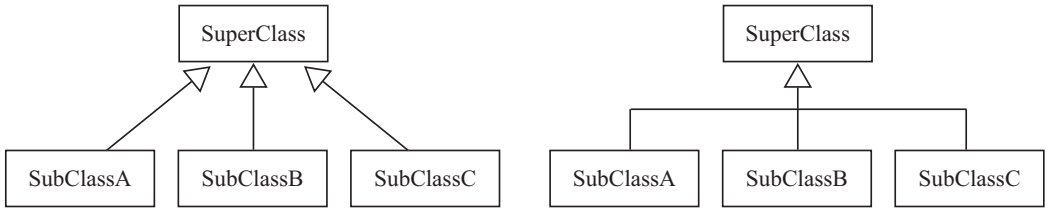


汎化

「汎化」関係は、スーパークラス側の端に白抜きの三角形をつけた実線で表す。



汎化関係や集約関係、コンポジション関係などは、いくつかの関係を 1 本にまとめて、複数に枝分かれするツリーで表すこともできる。一般に、図が体系的になって見やすくなるため、ツリーで表されることが多い。

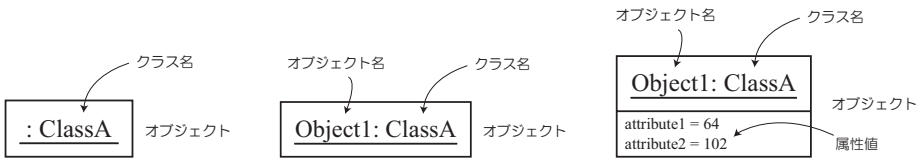


A.2　オブジェクト図の記法

オブジェクト図 (object diagram) は、ある時点でのシステムのスナップショット・イメージである。注意が必要なのは、オブジェクト図はシステムの 1 状態を表す例に過ぎず、システムの定義ではないという点である。

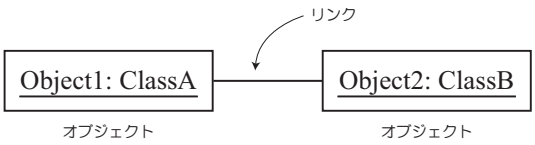
オブジェクト

オブジェクトは、クラスと同じ矩形を用い、名前に下線を引いて表す。どのクラスのインスタンスであるかを表すために「クラス名」が記述され、その名前の前にコロン (:) を書く。これで、そのクラス名のクラスから生成されたオブジェクトであるということを表す。オブジェクト名を明示することもでき、その場合には、コロンの前にオブジェクト名を書く。また、オブジェクトは具体的な属性の値をもっているので、それを明示する必要がある場合には、区画を区切って表示することができる。



リンク

オブジェクトとオブジェクトの個々の関係は、リンクによって表される。リンクは、関連のインスタンスである。リンクは、オブジェクトの矩形を結ぶ実線で表される。



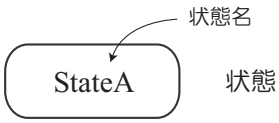
A.3　ステートチャート図の記法

ステートチャート図 (statechart diagram) は、オブジェクトの振る舞いを記述するために用いられる図である。イベントの反応の結果、どのように状態が変化するかの状態動作系列を表す。

状態機械を、状態 (state) を状態シンボルで、遷移 (transition) をそれらをつぐ矢印で表される。そこには、そのオブジェクトの振る舞いのすべての可能性が書かれている。これは１つのオブジェクトに着目して記述するので、局所化されたビューである。

状態 (state)

状態は丸み付き四角で表す。



初期状態 (initial state)

初期状態は、ふつうの状態への遷移をもつ擬似状態 (pseudostate) である。擬似状態 (pseudostate) とは、完全な状態としての振る舞いは行なわないという意味であり、オブジェクトは初期状態にとどまることはできず、直ちに遷移する。初期状態は、トリガーなし遷移 (triggerless transition) をもっており、必ず 1 つ出ていくことが保証されている必要がある。

初期状態は、塗りつぶした小さい黒丸として表示される。

最終状態 (final state)

最終状態 (final state) は、その実行が完了したことを示す状態である。そのため、最終状態からはイベントによりトリガーされて出ていく遷移をもつことはできない。最終状態は、初期状態のような擬似状態ではなく、一定期間アクティブになることが可能である。

最終状態は、標的アイコン、すなわち小さい白丸で囲まれた小さい黒丸で表記する。

遷移 (transition)

遷移は、ある状態からどのイベントに対して発火し、どのようなガード条件を満たし、どのようなアクションを実行し、どの状態に移るのか、ということを指定する。ステートチャート図では、遷移はソース状態からターゲット状態に至る実践の矢印で示す。

遷移の矢印には、次の形式の遷移文字列を記述する。

イベント名 [ガード条件] / アクション式

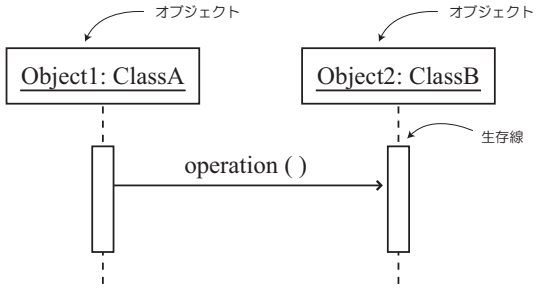
ガード条件はトリガーイベントのパラメータやその状態機械をもつオブジェクトの属性や関連についての論理式である。

遷移には、アクションを記述するアクション式 (action expression) が含まれる。これは、その状態機械をもつオブジェクトに影響を及ぼす手続き的計算を表す式であり、属性値を変更したり、オブジェクトの生成や計算、他のオブジェクトへのイベントの送信などが行なわれる。アクション式の構文は、UML では仕様化されていないため、プログラミング言語、擬似コード、自然言語などを用いてアクションを表現する。

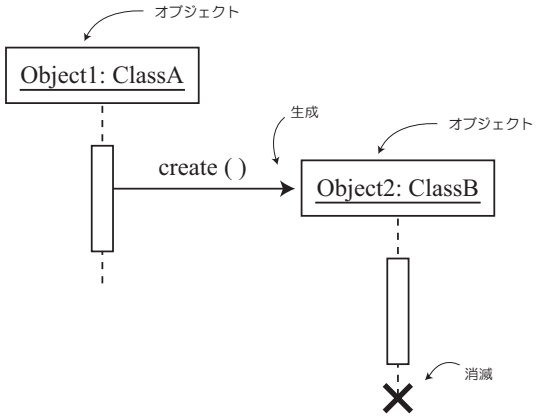
A.4 シーケンス図の記法

シーケンス図 (sequence diagram) は、オブジェクトの相互作用を時系列に表すための図である。注意が必要なのは、シーケンス図は、可能なシナリオのうちの 1 つのシナリオ例を記述するためのものであって、システムの定義ではないという点である。

シーケンス図では、通常、横に相互作用するオブジェクトを並べ、縦に上から下に進む時間軸を設定する。図の一番上の部分には、オブジェクトが書かれる。オブジェクトの配置の順番には意味はなく、わかりやすい配置がとられる。オブジェクトの矩形から、そのオブジェクトが消滅する時点まで、下方に破線が引かれる。これを生存線という。あるオブジェクトから他のオブジェクトへの作用は、作用するオブジェクトの生存線から、作用されるオブジェクトの生存線への矢印で示される。



生成と消滅を伴うシーケンス図を書くこともできる。新しいオブジェクトの生成は、塗りつぶした三角形を先端にもつ矢印で表す。また、オブジェクトの消滅は、×印で表す。



付 録B　BEFMシミュレーションモデル・フレームワークの詳細

B.1　Worldクラス

B.1.1　シミュレーション時計・空間の設定/取得

メソッド名	引数	戻り値	説明
setClock	Clock clock	なし	モデル内の時間経過をつかさどる時計を設定する。このメソッドを用いて時計を設定することで、シミュレーション実行時にモデル内へ TimeEvent が投げられるようになる。離散的な時間経過を扱うモデルであれば、引数として、new StepClock() を使用して差し支えない。
getClock	なし	Clock	この World が保持する Clock を取得する。存在しない場合、null を返す。
setSpace	Space space	なし	モデル内の地理的空間を設定する。地理的空間を扱わないモデルであれば、このメソッドは使用しなくても構わない。
getSpace	なし	Space	この World が保持する空間を取得する。存在しない場合、null を返す。

B.1.2　財の生成/明示的な消費

メソッド名	引数	戻り値	説明
createGoods	GoodsType type, double quantity	Goods	引数の type と quantity の Goods を生成して返す。
consumeGoods	Goods goods	なし	Goods を明示的に消費して Event を送りたい時に呼ぶメソッドである。通常は Agent の removeGoods メソッドで「財を世界から消す」ことはできるが、このメソッドを使用すると、使用時に UpdateWorldEvent が送られるため、プレゼンテーションコンポーネントなどで消費を追うことが容易になる。

B.1.3 エージェントの生成/参照/削除

メソッド名	引数	戻り値	説明
createAgent	AgentType type	Agent	引数の Type のエージェントを生成して、World に追加する。生成直後に Agent の init メソッドを呼ぶ。
createAgent	なし	Agent	デフォルトの Type を使用したエージェントを生成して、World に追加する。モデル内にエージェントが 1 種類しか登場しない場合や、Type によるプライオリティ設定の必要がない場合に、使用できる。
destoryAgent	Agent agent	なし	World から Agent を削除する。Agent の死亡・消滅を意味する。存在しない Agent を削除しようとした時、Exception を投げる。
getAgent	AgentType type	Agent	引数の Type の Agent を返す。該当する Agent が複数存在する場合、一番最初に追加された Agent を返す。存在しない場合、Exception を投げる。
getAgents	AgentType type	Collection	引数の Type の Agent の Collection を返す。もし存在しなければ、空の Collection を返す。
getAgentsRecursively	AgentType type	Collection	引数の Type 及びその子 Type の Agent の Collection を返す。存在しない場合、空の Collection を返す。
getAllAgents		Collection	World に追加されている全ての Agent の Collection を返す。

B.1.4 タイプとプライオリティの設定

メソッド名	引数	戻り値	説明
installAgentType	String name	AgentType	AgentType を生成して返す。Type の生成には必ずこれらのメソッドを用いること。
installAgentType	String name, int priority	AgentType	プライオリティをもつ AgentType を生成して返す。Type によってエージェントの行動順を制御したい場合に使用するメソッドである。Type の生成には必ずこれらのメソッドを用いること。
installGoodsType	String name	GoodsType	GoodsType を生成して返す。Type の生成には必ずこれらのメソッドを用いること。
installBehaviorType	String name	BehaviorType	BehaviorType を生成して返す。Type の生成には必ずこれらのメソッドを用いること。
installRelationType	String name	RelationType	RelationType を生成して返す。Type の生成には必ずこれらのメソッドを用いること。

B.1.5 乱数ジェネレータの追加/取得

メソッド名	引数	戻り値	説明
installRandom NumberGenerator	RandomNumber Generator random	なし	引数の RandomNumberGenerator を World の randomMap に追加する。もし、同じ name の RandomNumberGenerator が既にあれば、上書きされる。
getRandomNumber Generator	String name	RandomNumber Generator	引数の name の RandomNumberGenerator を検索して返す。存在しない場合、null を返す。
getRandomNumber Generator	なし	RandomNumber Generator	Default の RandomNumberGenerator を検索して返す。存在しない場合、null を返す。

B.2 Agent クラス

B.2.1 行動の追加/取得

メソッド名	引数	戻り値	説明
addBehavior	BehaviorType type	なし	Behavior を追加する。追加された Behavior は開始状態となる。この時、追加された Behavior には所有者である Agent がセットされる。
getBehavior	BehaviorType type	Behavior	引数の Type である Behavior を返す。該当する Behavior が複数存在する場合、一番初めに追加された Behavior を返す。存在しない場合、Exceptionが投げられる。
getBehaviors	BehaviorType type	Collection	引数の Type である Behavior を Collection として全て返す。存在しない場合、空の Collection を返す。
getBehaviorsRecursively	BehaviorType type	Collection	引数の Type およびその子である Behavior を Collection として全て返す。存在しない場合、空の Collection を返す。
removeBehavior	Behavior behavior	なし	Behavior を終了して削除する。Behavior 終了には、特にトラブルが発生しやすいため、注意して使用すること）

B.2.2 所有財の追加/取得

メソッド名	引数	戻り値	説明
addGoods	Goods goods	なし	財を所有財に追加する。
removeGoods	GoodsType type, double quantity	Goods	指定された種類の財を指定量引き出して、返す。指定された種類の財が指定量存在しない場合は、Exception が投げられる。type は、引き出す財の種類。quantity は、引き出す財の量。
removeGoodsRecursibely	GoodsType type, double quantity	Collection	指定された種類の財を指定量引き出して、返す。指定された種類に下位種類があれば再帰的に検索するので、返されるのは Goods ではなく Collection である。指定された種類の財が 1 種類でも指定量ぶん存在しない場合、Exception が投げられる。
removeAllGoods	GoodsType type	Goods	指定された種類の財をすべて引き出して、返す。指定された種類の財が存在しない場合、Exception が投げられる。
removeAllGoodsRecursively	GoodsType type	Collection	指定された種類の財を全て引き出して、返す。指定された種類に下位種類があれば再帰的に検索するので、返されるのは Goods ではなく Collection である。指定された種類の財が全く存在しない場合、Exception が投げられる。
getQuantity	GoodsType type	GoodsQuantity	指定された種類の財の量を取得する。指定された種類の財が無ければ 0 を表す GoodsQuantity インスタンスが返される。取得した GoodsQuantity から int、double を得るには、GoodsQuantity のもつ getValueAsInt()、getValueAsDouble() を使用すること。
getQuantityRecursively	GoodsType type	GoodsQuantity	指定された種類の財の量を取得する。指定された種類に下位種類があれば再帰的に検索する。指定された種類の財が無ければ 0 を表す GoodsQuantity インスタンスが返される。取得した GoodsQuantity から int、double を得るには、GoodsQuantity のもつ getValueAsInt()、getValueAsDouble() を使用すること。
hasGoods	GoodsType type	boolean	指定された種類の財を持っているかどうか、真偽を返す。
getGoodsTypes	なし	Collection	この Agent が持つ全ての財の種類を返す。

B.2.3 情報の追加/取得

メソッド名	引数	戻り値	説明
putInformation	Information key, Information value	なし	エージェントに情報を追加する。key は、取り出す時の検索キー（情報）。value は、追加したい情報。
getInformation	Information key	Information	引数の Information をキーとする Information(clone ではない) を返す。キーが見つからなかった場合、Exception が投げられる。
removeInformation	Information key	Information	引数の Information をキーとする Information(clone ではない) を削除して、返す。キーが見つからなかった場合、Exception が投げられる。

B.2.4 関係の追加/取得

メソッド名	引数	戻り値	説明
addRelation	RelationType relationType, Agent target	なし	関係を追加する。エージェントは同じ種類の関係を複数もつことができる。relationType は、追加したい関係の Type。target は、関係先の Agent。
addRelation	Agent target	なし	デフォルトの Type を利用して関係を追加する。モデル内で関係を 1 種類しか使わない時や、全てのエージェントが関係を 1 種類しかもたない時に使用できる。
getRelation	RelationType type	Relation	引数の Type である関係を返す。該当する関係が複数存在する場合、一番最初に追加された関係を返す。存在しない場合、Exception が投げられる。
getRelation	RelationType type, Agent agent	Relation	引数の Type、かつ関係先が引数の Agent である関係を返す。
getRelations	RelationType type	Collection	引数の Type である関係を全て返す。存在しない場合、空の Collection が返される。
getRelationsRecursively	RelationType type	Collection	引数の Type 及びその Type の子の Type である全ての関係を返す。存在しない場合、空の Collection が返される。
removeRelation	Relation relation	なし	引数の関係を削除する。
removeRelations	RelationType type	なし	引数の Type の関係全てを削除する。
removeRelationsRecursively	RelationType type	なし	引数の Type 及びその子 Type の関係全てを削除する。
getRelationTypes	なし	Collection	この RelaitonManager のもつ関係の Type を返す。存在しない場合、Exception が投げられる。

B.3 Behavior クラス

B.3.1 エージェント /世界の参照

メソッド名	引数	戻り値	説明
getAgent	なし	Agent	その行動をもっているエージェントを返す。Behavior を継承したクラス内で後述したエージェントメソッドを使用するには、このメソッドでエージェントを参照すること。
getWorld	なし	World	その行動をもっているエージェントが存在する世界を返す。Behavior を継承したクラス内でエージェントを生成するといったワールドメソッドを使用するには、このメソッドで世界を参照すること。

B.3.2 財の送信

メソッド名	引数	戻り値	説明
sendGoods	Goods goods	なし	既に開いている経路を利用して、財を送信する。開いている経路がない場合、Exception を投げる。
sendGoods	Relation relation, BehaviorType behaviorType, Goods goods, boolean keep	なし	関係先の単数のエージェントに対して、経路を開き財を送信する。relation は、送信したいエージェント単体に対する関係。behaviorType は、送信先のエージェントがもつ、送信財を受け取る行動の Type。goods は、送信したい財。keep は経路を keep するかどうかの真偽値。
sendGoods	Relation relation, BehaviorType behaviorType, Goods goods	なし	関係先の単数のエージェントに対して、経路を開き財を送信する。この時、keep=false な経路を自動的に開設するところが、上のメソッドと異なる。
sendGoods	RelationType relationType, BehaviorType behaviorType, GoodsType goodsType, double goodsQuantity, boolean createGoods, boolean keep	int	指定した関係先の全てのエージェントに対して、経路を開いて財を送信する。戻り値は送り先エージェントの数である。relationType は、送信したい各エージェントに対する関係の Type。behaviorType は、送信先の各エージェントがもつ、送信財を受け取る行動の Type。goodsType は、送信したい財の Type。goodsQuantity は、送信したい財の量。createGoods は、送信したい財を新しく生成するかどうかの真偽値。false の場合、送信財は送信元のエージェントの所有財から取り出される。keep は経路を keep するかどうかの真偽値。
sendGoods	RelationType relationType, BehaviorType behaviorType, GoodsType goodsType, double goodsQuantity, boolean createGoods	int	指定した関係先の全てのエージェントに対して、経路を開き財を送信する。戻り値は送り先のエージェント数である。この時、Keep=false な経路を自動的に開設するところが、上のメソッドと異なる。

B.3.3 情報の送信

メソッド名	引数	戻り値	説明
sendInformation	Information information	なし	既に開いている経路を利用して、情報を送信する。開いている経路がない場合は例外を投げる。
sendInformation	Relation relation, BehaviorType behaviorType, Information information, boolean keep	なし	関係先の単数のエージェントに対して、経路を開き情報を送信する。relation は、送信したいエージェント単体に対する関係。behaviorType は、送信先のエージェントがもつ、送信情報を受け取る行動の Type。information は、送信したい情報。keep は経路を keep するかどうかの真偽値。
sendInformation	Relation relation, BehaviorType behaviorType, Information information	なし	関係先の単数のエージェントに対して、経路を開き情報を送信する。この時、Keep=false な経路を自動的に開設するところが、上のメソッドと異なる。
sendInformation	RelationType relationType, BehaviorType behaviorType, Information information, boolean keep	int	指定した関係先の全てのエージェントに対して、経路を開き情報を送信する。戻り値は送り先エージェントの数である。relationType は、送信したい各エージェントに対する関係の Type。behaviorType は、送信先の各エージェントがもつ、送信情報を受け取る行動の Type。information は、送信したい情報の Type。keep は経路を keep するかどうかの真偽値。
sendInformation	RelationType relationType, BehaviorType behaviorType, Information information, boolean keep	int	指定した関係先の全てのエージェントに対して、経路を開き情報を送信する。戻り値は送り先エージェントの数である。この時、Keep=false な経路を自動的に開設するところが、上のメソッドと異なる。

B.3.4 財/情報の受信

メソッド名	引数	戻り値	説明
getReceivedGoods	なし	Goods	最後に送られてきた財の参照を返す。再び財/情報が送られてくるまで、このメソッドは同じ参照を返す。
getReceivedInformation	なし	Information	最後に送られてきた情報の参照を返す。再び財/情報が送られてくるまで、このメソッドは同じ参照を返す。

付 録C　モデル・パターン カタログ

C.1　モデル・パターンの分類

本カタログは、Boxed Economy Foundation Modelに基づくモデルにおいて繰り返し登場するモデル・パターンを記述したものである。取り上げるモデル・パターンは、大きく分けて次のような分類ができる。

- エレメンタリーなモデル・パターン
- コミュニケーションのモデル・パターン
- 行動変化のモデル・パターン
- アクティベーションのモデル・パターン

各分類に属するモデル・パターンの一覧は次のようになる。

モデル・パターンの分類	モデル・パターン名
エレメンタリーなモデル・パターン	Agent Creation (p. 236) Relation Creation (p. 238) Related Agent Creation (p. 240) Agent Destruction (p. 242) Goods Creation (p. 244) Information Creation (p. 246)
コミュニケーションのモデル・パターン	Information Sending (p. 248) Blank Information Sending (p. 252) Internal Information Sending (p. 256) Immediate Reply (p. 260) Collect Immediate Replies (p. 264) Appointed Destination Reply (p. 268) Super BehaviorType Calling (p. 272)
行動変化のモデル・パターン	Behavior Creation (p. 276) Behavior Destruction (p. 278) Behavior Switching (p. 280) Temporary Behavior Creation (p. 282) Requested Behavior Attachment (p. 284) Forced Behavior Attachment (p. 288)
アクティベーションのモデル・パターン	TimeEvent Distributer Agent (p. 290) TimeEvent Filtering (p. 294) TimeEvent Distributer Behavior (p. 296) Time-Consuming Behavior (p. 298)

C.2　パターンにおけるクラス名・オブジェクト名について

基本動作、設計、サンプルコードに記載されているエージェント名などは、ここでの説明用の名称がつけられている。モデルにおいて現れるときには、そのコンテクストにあった名前をつける必要がある。

C.3　設計におけるオブジェクト図について

本カタログの「設計」の部分は、オブジェクト図を変形した図で記述している。この図は、モデルにおいて登場するすべてのモデル要素を記述しているため、正しいオブジェクト図とはいえない。なぜなら、本来オブジェクト図は、あるシステムのある特定の時間における状態のスナップショットを記述するものだからである。Type オブジェクトによるモデル要素の表現の関係から、本カタログでは、このような変形版オブジェクト図を用いる。

状態遷移図では、そのパターンに関する処理を行う action については、その目的をわかりやすくするための名前をつけている (そうでない場合には、nextAction という名前をつけてある)。これらの action 名も、モデルのコンテクストに合った名前に付けなおすことが望ましい。

C.4　サンプルコードについて

World クラスを継承した「〜World」は、initializeWorld メソッドと initializeAgents メソッドをオーバーライドする。本カタログに示すサンプルはすべて、以下のような initializeWorld メソッドを想定している。

【〜World クラス】

```
...
  public void initializeWorld() {
     super.initializeWorld();

     //時計に StepClock を設定
     this.setClock(new StepClock());
  }
...
```

initializeAgents メソッドは、モデルによってその処理の内容が異なるため、カタログごとにサンプルコードを掲載している。「〜World」の全体像は、次のようになる (AgentCreationWorld クラスの例)。

【AgentCreationWorld クラス】

```
package org.boxed_economy.agentcreation.model;

import org.boxed_economy.besp.container.BESP;
import org.boxed_economy.besp.model.fmfw.*;
import org.boxed_economy.components.stepclock.StepClock;


public class AgentCreationWorld extends World {

   //World の初期化

   public void initializeWorld() {
      super.initializeWorld();

      //時計に StepClock を設定
      this.setClock(new StepClock());
   }


   //Agent の初期化

   public void initializeAgents() {

      //AgentCreator エージェントの生成
      Agent agentCreator = createAgent(AgentCreationModel.AGENTTYPE_AgentCreator);

      //そのエージェントへの CreateAgentBehavior の追加
      agentCreator.addBehavior(AgentCreationModel.BEHAVIORTYPE_CreateAgent);
   }
}
```

C.5　バリエーションについて

ふつう、同じ目的を満たすようなモデルは、いくつか存在する。モデル・パターンでは典型的なサンプルモデルをあげているが、常にこれが最良であるというわけではない。「バリエーション」の項では、代替的な案について、若干補足している。これらを参考に、適用する文脈に適した形で変形して利用することが期待される。

なお、サンプルモデルでは、登場するエージェントが、それぞれ異なる AgentType をもつというモデル化をしているが、多くの場合、そのようにする必要はない。説明の便宜上のものだと考えてほしい。

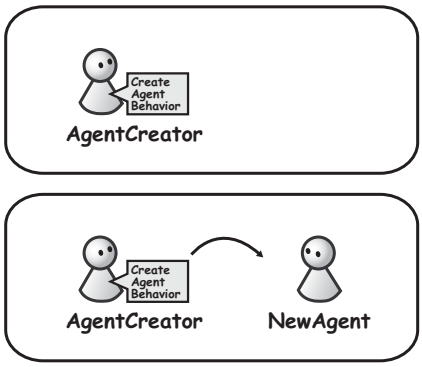
Agent Creation

■ 目的

新しいエージェントを生成する。

■ 動機

人口が増減するモデルや、組織が形成・解体されるモデルでは、シミュレーション実行中に、新しいエージェントを生成し、世界に追加する必要がある。このエージェント生成処理を内生化したい場合には、モデル内のいずれかのエージェントが、生成処理を行う必要がある。

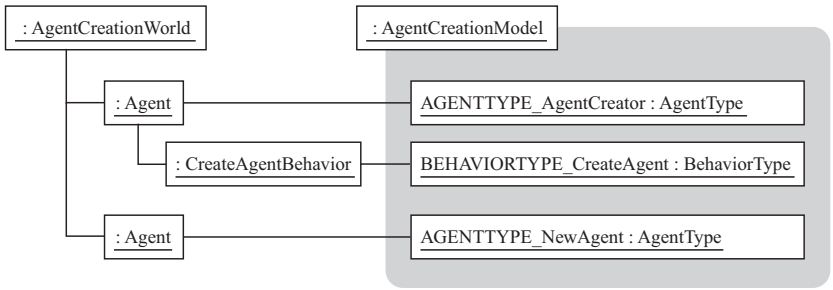


■ 基本動作

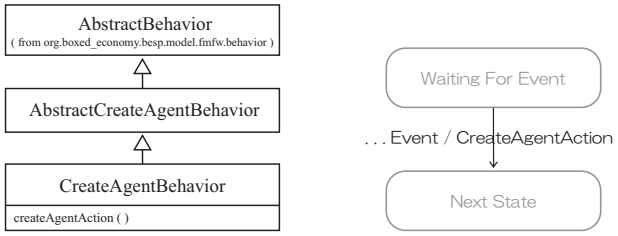
AgentCreator エージェントは CreateAgentBehavior を持っており、この CreateAgentBehavior によって、NewAgent エージェントを生成する。

■ 設計

【全体像】



【CreateAgentBehavior】



■ サンプルコード

【AgentCreationWorld クラス】

```
…
public void initializeAgents() {
    //AgentCreator エージェントの生成
    Agent agentCreator = createAgent(AgentCreationModel.AGENTTYPE_AgentCreator);

    //そのエージェントへの CreateAgentBehavior の追加
    agentCreator.addBehavior(AgentCreationModel.BEHAVIORTYPE_CreateAgent);
}
…
```

【CreateAgentBehavior クラス】

```
…
protected void createAgentAction() {
    //新しいエージェントの生成
    Agent createAgent = this.getWorld().createAgent(AgentCreationModel.AGENTTYPE_NewAgent);
}
…
```

■ バリエーション

ここでのサンプルでは、生成したエージェントに行動をもたせていないため、このエージェントは何もしない。何らかの行動をさせたい場合には、このエージェントに行動を付加する必要がある (→ Forced Behavior Attachment 参照)。

■ 関連するパターン

Related Agent Creation: 新しくエージェントを生成して、そのエージェントに関係を結ぶ。

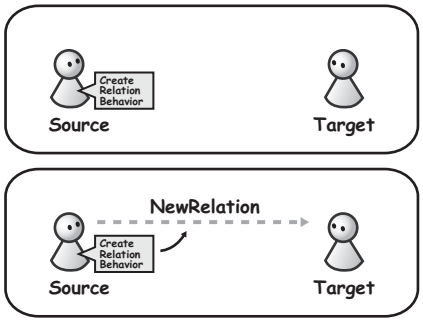
Relation Creation

■ 目的

他のエージェントとの関係を生成する。

■ 動機

エージェント間の関係が変化したり、新しい関係性が生じるモデルでは、シミュレーション実行中に、エージェント間の関係を生成して結ぶ必要がある。この関係生成処理を内生化したい場合には、モデル内のいずれかのエージェントが、この生成処理を行う必要がある。

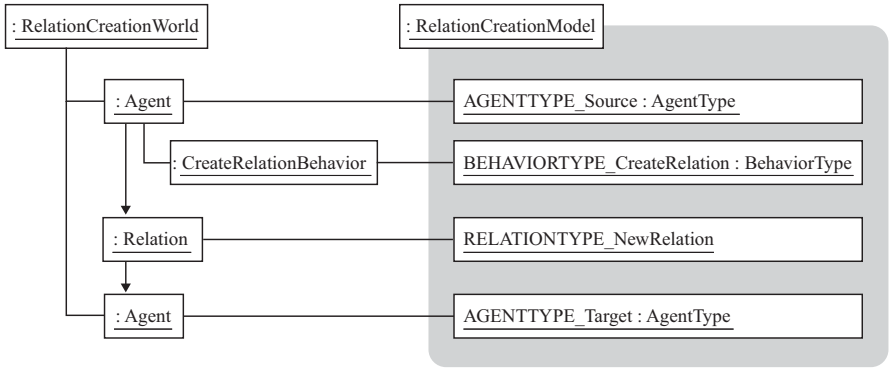


■ 基本動作

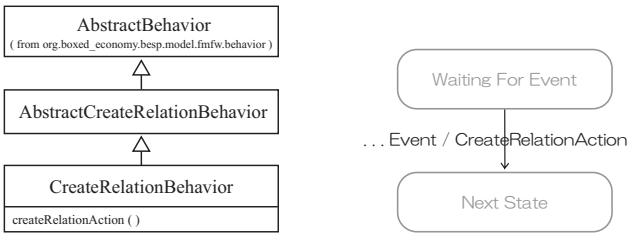
Source エージェントと Target エージェントが登場する。Source エージェントは CreateRelationBehavior を持っており、この CreateRelationBehavior によって NewRelation を生成し、Target エージェントと関係を結ぶ。

■ 設計

【全体像】



【CreateAgentBehavior】



■ サンプルコード

【RelationCreationWorld クラス】

```
…
public void initializeAgents() {
   //Source エージェントと Target エージェントの生成
   Agent source = createAgent(RelationCreationModel.AGENTTYPE_Source);
   Agent target = createAgent(RelationCreationModel.AGENTTYPE_Target);

   //Source エージェントへの行動の追加
   source.addBehavior(RelationCreationModel.BEHAVIORTYPE_CreateRelation);
}
…
```

【CreateRelationBehavior クラス】

```
…
protected void createRelationAction() {
   //関係を結ぶ相手の特定
   Agent target = this.getWorld().getAgent(RelationCreationModel.AGENTTYPE_Target);
   //相手と関係を結ぶ
   this.getAgent().addRelation(RelationCreationModel.RELATIONTYPE_NewRelation, target);
}
…
```

■ バリエーション

ここでのサンプルでは、関係を結ぶエージェントの特定化に、AgentType を用いている。このほかの代替案としては、(1) エージェントを特定化するための情報を受取り、それを用いて指定する、(2) 世界に存在するエージェントを調べて、その中から選択して指定する、という方法が考えられる。

■ 関連するパターン

Related Agent Creation: 新しくエージェントを生成して、そのエージェントに関係を結ぶ。

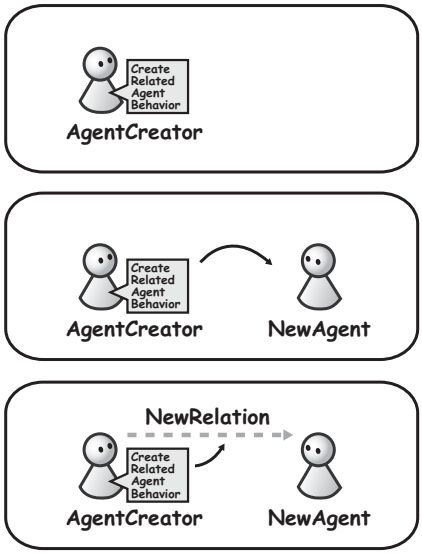
Related Agent Creation

■ 目的

新しいエージェントを生成し、関係を結ぶ。

■ 動機

エージェントの生成を伴うモデルでは、生成処理を行うエージェントと、新しく生成されたエージェントの間に、なんらかの関係を持たせたいということがしばしばある。例えば、個人エージェントが子供を生む場合は、それらの間に親子関係を結ぶことになるだろう。また、個人もしくは組織エージェントが、新しい組織を形成したり、内部組織を分化させる場合にも、生成処理を行ったエージェントとなんらかの関係を結ぶことが想定される。

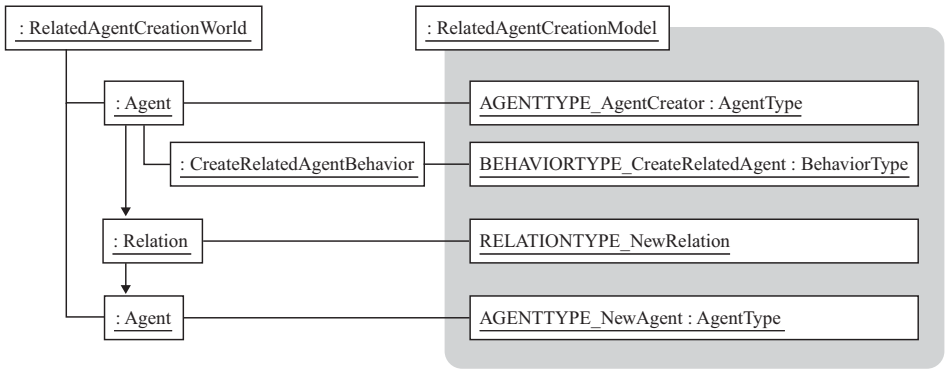


■ 基本動作

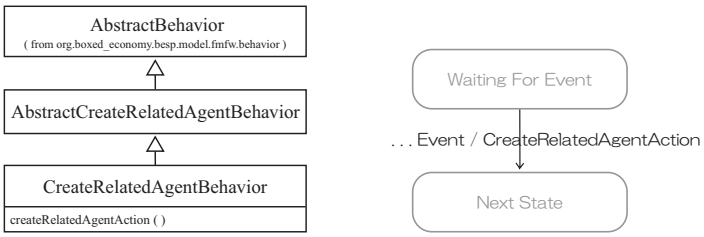
AgentCreator エージェントは CreateRelatedAgentBehavior を持っている。この CreateRelatedAgentBehavior によって、NewAgent エージェントを生成した後、NewRelation を結ぶ。

■ 設計

【全体像】



【CreateRelatedAgentBehavior】



■ サンプルコード

【RelatedAgentCreationWorld クラス】

```
 …
public void initializeAgents() {
   //AgentCreator エージェントの生成
   Agent agentCreator = createAgent(AgentCreationModel.AGENTTYPE_AgentCreator);

   //そのエージェントへの CreateAgentBehavior の追加
   agentCreator.addBehavior(RelatedAgentCreationModel.BEHAVIORTYPE_CreateRelatedAgent);
}
 …
```

【CreateRelatedAgentBehavior クラス】

```
 …
protected void createRelatedAgentAction() {
   //新しいエージェントの生成
   Agent createAgent =
      this.getWorld().createAgent(RelatedAgentCreationModel.AGENTTYPE_NewAgent);

   //自分から新しいエージェントへの関係の設定
   this.getAgent().
      addRelation(RelatedAgentCreationModel.RELATIONTYPE_NewRelation, createdAgent);
}
 …
```

■ バリエーション

ここでのサンプルでは、生成したエージェントに行動をもたせていないため、このエージェントは何もしない。何らかの行動をさせたい場合には、このエージェントに行動を付加する必要がある (→ Forced Behavior Attachment 参照)。

■ 関連するパターン

Agent Creation: 関係をもたないエージェントを生成する。
Relation Creation: すでに存在するエージェントと関係を結ぶ。

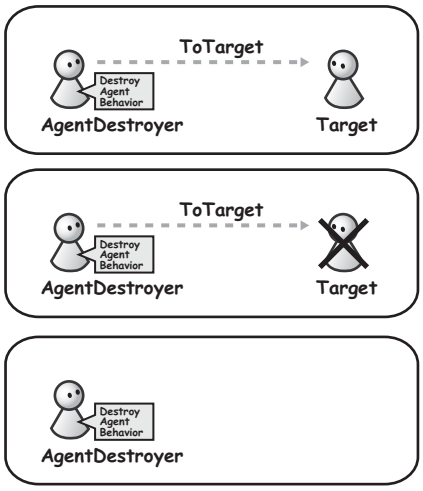
Agent Destruction

■ 目的

他のエージェントを消滅させる。

■ 動機

人口が増減するモデルや、組織が形成・解体されるモデルでは、シミュレーション実行中に、エージェントを消滅させる必要がでてくる。例えば、個人エージェントの死亡や、組織エージェントの解散などが、これにあたる。このエージェント消滅処理を内生化したい場合には、モデル内のいずれかのエージェントが、消滅処理を行う必要がある。

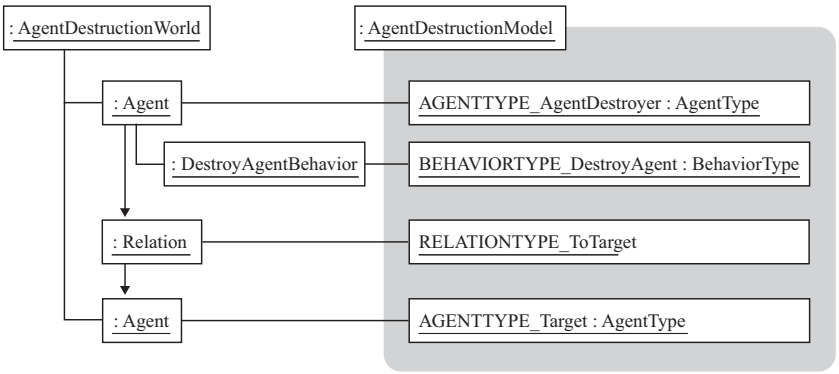


■ 基本動作

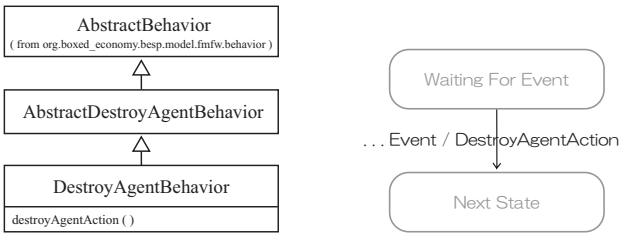
AgentDestroyer エージェントは DestroyAgentBehavior を持っている。この DestroyAgentBehavior によって、Target エージェントを消滅させる (このとき、関係は自動的に消滅する)。

■ 設計

【全体像】



【DestroyAgentBehavior】



■ サンプルコード

【AgentDestructionWorld クラス】

```
 …
public void initializeAgents() {

   //AgentDestroyer エージェントの生成
   Agent agentDestroyer = createAgent(AgentDestructionModel.AGENTTYPE_AgentDestroyer);

   //そのエージェントへの DestroyAgentBehavior の追加
   agentDestroyer.addBehavior(AgentDestructionModel.BEHAVIORTYPE_DestroyAgent);

   //消滅させる Target エージェントの生成
   Agent target = createAgent(AgentDestructionModel.AGENTTYPE_Target);

   //AgentDestroyer エージェントから Target エージェントへの関係を結ぶ
   agentDestroyer.addRelation(AgentDestructionModel.RELATIONTYPE_ToTarget, target);
}
  …
```

【DestroyAgentBehavior クラス】

```
 …
protected void destroyAgentAction() {
   //削除するエージェントの特定
   Agent target = this.getAgent()
      .getRelation(AgentDestructionModel.RELATIONTYPE_ToTarget).getTarget();

   //target のエージェントの削除
   this.getWorld().destroyAgent(target);
}
  …
```

■ バリエーション

ここでのサンプルでは、消滅させるエージェントの特定化に、AgentType を用いている。このほかの代替案としては、(1) エージェントを特定化するための情報を受取り、それを用いて指定する、(2) 世界に存在するエージェントを調べて、その中から選択して指定する、という方法が考えられる。

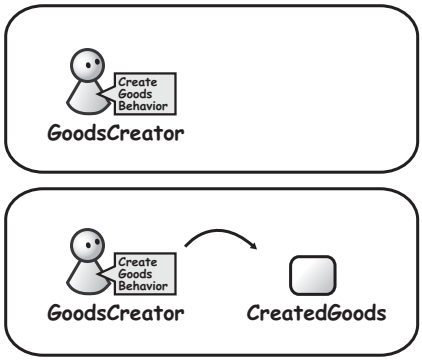
Goods Creation

■ 目的

財を生成する。

■ 動機

商品を生産したり調達したりするモデルでは、シミュレーション実行中に、財を生成する必要がある。この財生成処理を内生化したい場合には、モデル内のいずれかのエージェントが、生成処理を行う必要がある。

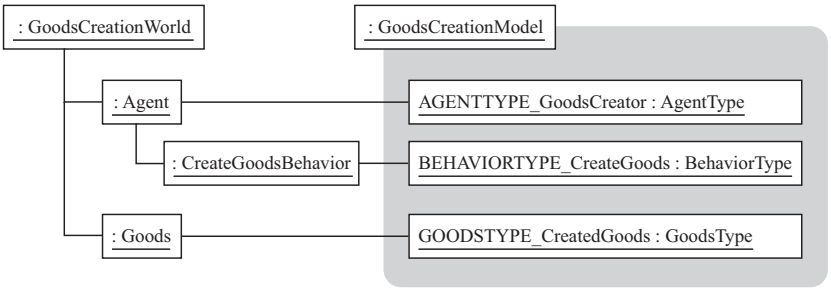


■ 基本動作

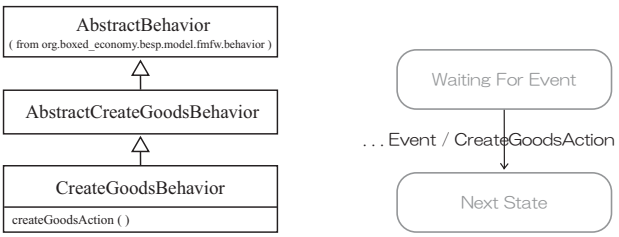
GoodsCreator エージェントは CreateGoodsBehavior を持っている。この CreateGoodsBehavior によって、CreatedGoods を生成する。

■ 設計

【全体像】



【CreateGoodsBehavior】



■ サンプルコード

【GoodsCreationWorld クラス】

```
…
public void initializeAgents() {
   //GoodsCreator エージェントの生成
   Agent goodsCreator = createAgent(GoodsCreationModel.AGENTTYPE_GoodsCreator);

   //そのエージェントへの CreateGoodsBehavior の追加
   goodsCreator.addBehavior(GoodsCreationModel.BEHAVIORTYPE_CreateGoods);
}
…
```

【CreateGoodsBehavior クラス】

```
…
protected void createGoodsAction() {
   //Goods の作成
   Goods createdGoods =
      this.getWorld().createGoods(GoodsCreationModel.GOODSTYPE_CreatedGoods, 1.0);
}
…
```

■ バリエーション

このサンプルで行っているのは、財を生成することのみである。この後に行う動作として考えられるのは、(1) 財を他のエージェントに送るという動作であり、それは、Behavior の sendGoods( ) で行う。あるいは、(2) 自分の所有財として保管するという動作も考えられ、それは、Agent の addGoods( ) で行う。

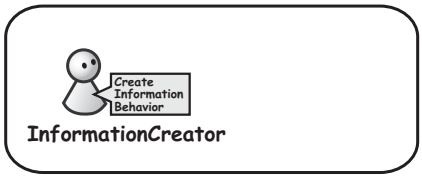
Information Creation

■ 目的

情報を作成する。

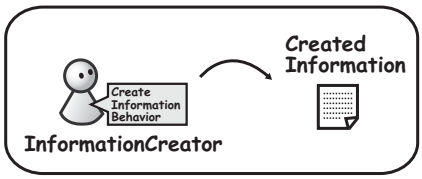
■ 動機

エージェント間の相互作用を行う場合や、何らかのデータを記憶したい場合には、シミュレーション実行中に、エージェントが情報を作成する必要がある。



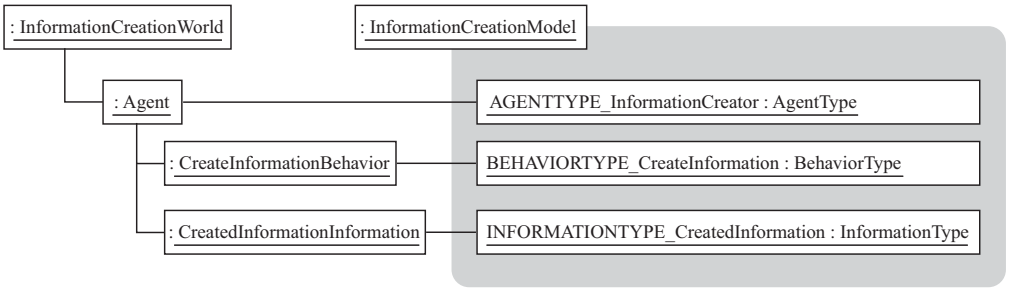
■ 基本動作

InformationCreator エージェントは、CreateInformationBehavior もっており、この CreateInformationBehavior によって CreatedInformation を作成する。

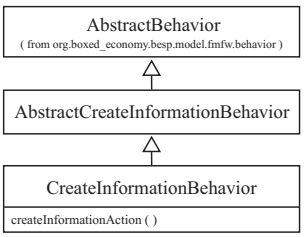


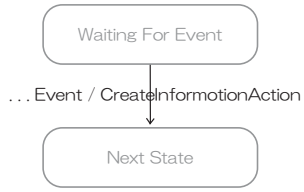
■ 設計

【全体像】



【CreateInformationBehavior】





■ サンプルコード

【InformationCreationWorld クラス】

```
…
public void initializeAgents() {
   //InformationCreator エージェントの生成
   Agent informationCreator =
      createAgent(InformationCreationModel.AGENTTYPE_InformationCreator);

   //そのエージェントへの CreateInformationBehavior の追加
   informationCreator.addBehavior(
      InformationCreationModel.BEHAVIORTYPE_CreateInformation);
}
…
```

【CreateInformationBehavior クラス】

```
…
protected void createInformationAction() {
   //情報の生成
   CreatedInformation createdInformation = new CreatedInformation();

   //情報の追加
   this.getAgent().putInformation(createdInformation);
}
…
```

【CreatedInformation クラス】

```
…
   public class CreatedInformation implements Information {
      //ここに情報の形式、および設定・取得のためのメソッド等を書く
      …
   }
…
```

■ 関連するパターン

Information Sending: 情報を送る。

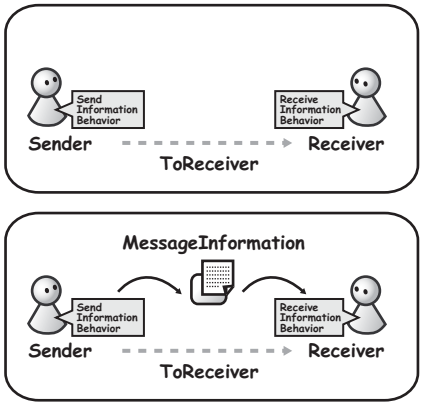
Information Sending

■ 目的

他のエージェントに、Information を送信する。

■ 動機

エージェントが、他のエージェントに何らかのメッセージを送ったり、質問をしたりするようにしたい。

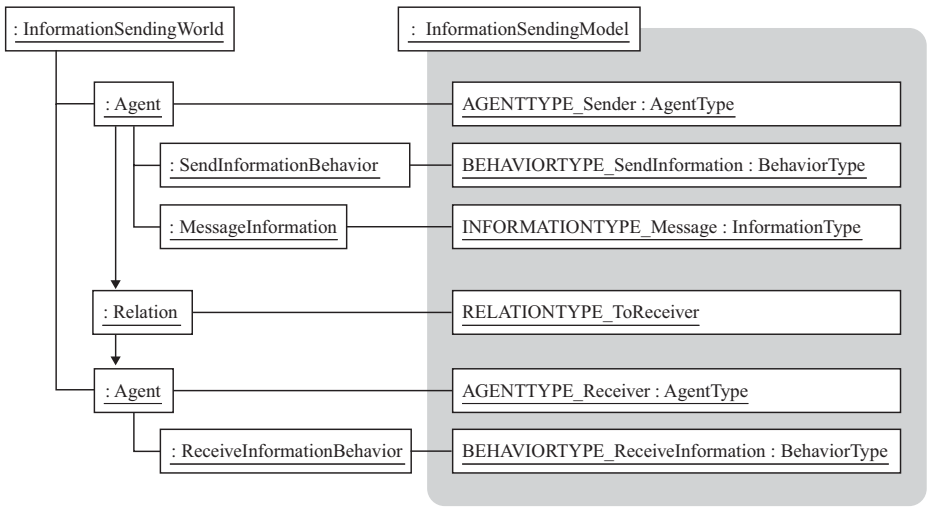


■ 基本動作

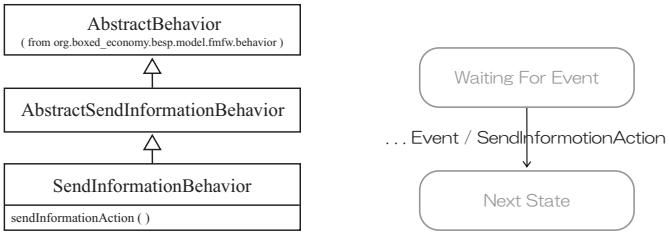
Sender エージェントと Receiver エージェントが登場する。Sender エージェントは、SendInformationBehavior をもっている。SendInformationBehavior は、(ここでは) MessageInformation を生成し、Receiver エージェントに送信する。Receiver エージェントは、ReceiveInformationBehavior でそれを受ける。

■ 設計

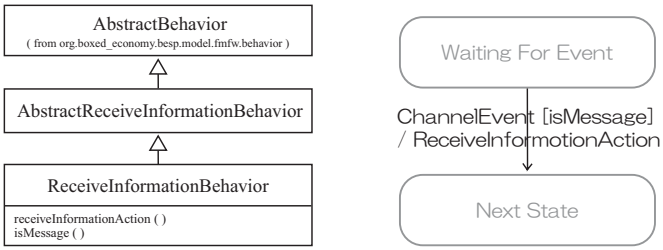
【全体像】



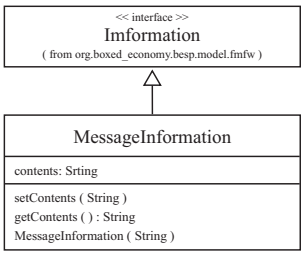
【SendInformationBehavior】



【ReceiveInformationBehavior】



【MessageInformation】



■ サンプルコード

【InformationSendingWorld クラス】

```
…
public void initializeAgents() {
   //Sender エージェントの生成
   Agent sender = createAgent(InformationSendingModel.AGENTTYPE_Sender);
   //Sender エージェントへの SendInformationBehavior の追加
   sender.addBehavior(InformationSendingModel.BEHAVIORTYPE_SendInformation);

   //Receiver エージェントの生成
   Agent receiver = createAgent(InformationSendingModel.AGENTTYPE_Receiver);
   //Receiver エージェントへの ReceiveInformationBehavior の追加
   receiver.addBehavior(InformationSendingModel.BEHAVIORTYPE_ReceiveInformation);

   //Sender エージェントから Receiver エージェントへの関係の追加
   sender.addRelation(InformationSendingModel.RELATIONTYPE_ToReceiver, receiver);
}
…
```

【SendInformationBehavior クラス】

```
…
protected void sendInformationAction() {
   //情報の生成 (ここでは、メッセージの内容は文字列)
   MessageInformation message = new MessageInformation("Hello!");

   //生成した情報の送信
   this.sendInformation(InformationSendingModel.RELATIONTYPE_ToReceiver,
      InformationSendingModel.BEHAVIORTYPE_ReceiveInformation, message);
}
…
```

【ReceiveInformationBehavior クラス】

```
…
protected void receiveInformationAction() {
   //受信した情報を取得
   Information receivedInformation = getReceivedInformation();
   //メッセージの内容を取得 (ここでは文字列)
   String messageContents =
      ((MessageInformation)receivedInformation).getContents();
}

protected boolean isMessage(Event e) {
   //送られてきた情報が、MessageInformation であれば true を返す。
   return this.getWorld().getInformationType(getReceivedInformation())
      == InformationSendingModel.INFORMATIONTYPE_Message;
}
…
```

【MessageInformation クラス】

```
…
public class MessageInformation implements Information {
   //ここでは、内容は文字列
   String contents;

   //コンストラクタ (引数 = 文字列)
   public MessageInformation(String contents) {
      this.contents = contents;
   }

   //内容を返す
   public String getContents() {
      return contents;
   }

   //内容を設定する
   public void setContents(String contents) {
      this.contents = contents;
   }
}
…
```

■ バリエーション

このサンプルでは、文字列の内容を記録する MessageInformation という情報を送っているが、それ以外の情報形式でも構わない。文脈に応じて自由に設計することができる。

また、このサンプルでは、どのような情報が送られてきたのかを、InformationType で判断させているが、情報の内容で判別させることもできる。

■ 関連するパターン

Information Creation: Information を生成する。
Blank Information Sending: 内容を伴わないシグナルを送る。

Blank Information Sending

■ 目的

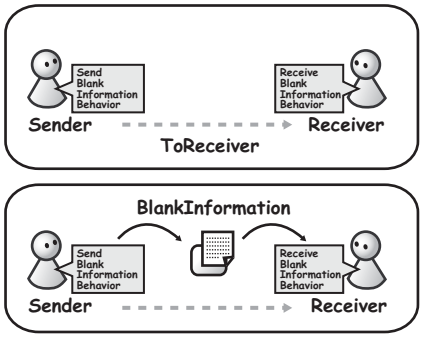
他のエージェントに、内容が空の Information を送信する。

■ 動機

単に合図や質問を送りたいときには、相手に情報の種類が伝わればよいということがしばしばある。このような場合には、内容を伴わない Information を送るだけで済ましたい。

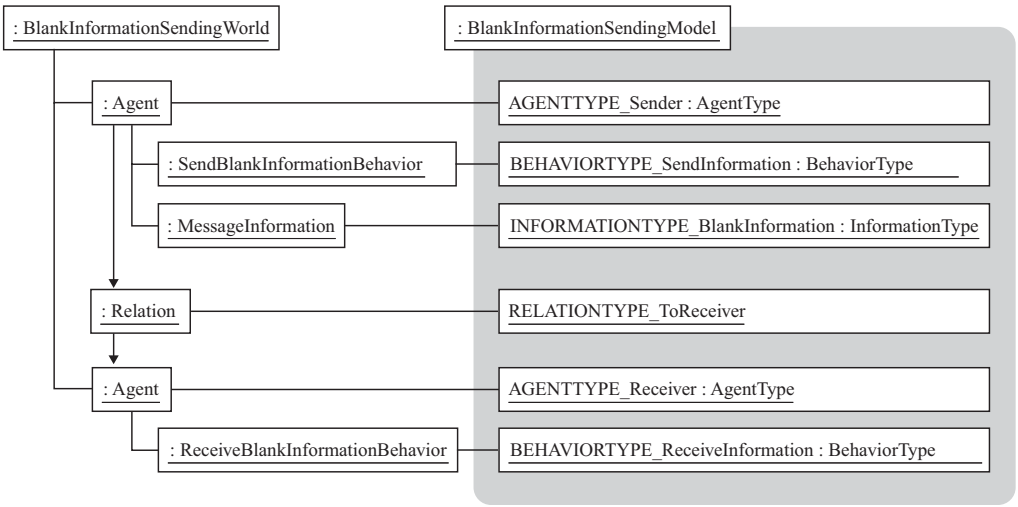
■ 基本動作

Sender エージェントと Receiver エージェントが登場する。Sender エージェントは、SendBlankInformationBehavior をもっている。SendBlankInformationBehavior では、内容を伴わない BlankInformation を生成し、Receiver エージェントに送信する。Receiver エージェントは、ReceiveBlankInformationBehavior でそれを受ける。どのような情報が送られてきたのかの識別は、BlankInformation の InformationType で判断する。

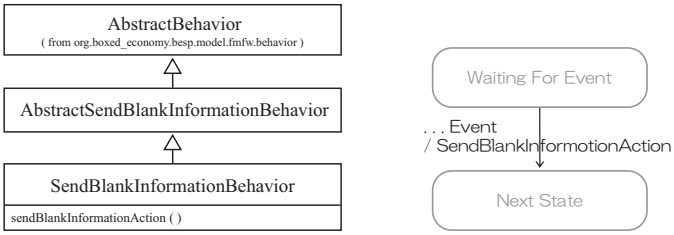


■ 設計

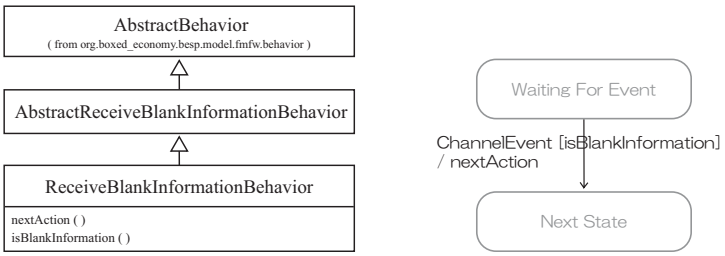
【全体像】



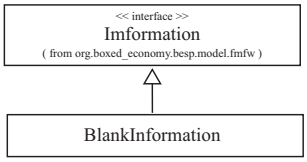
【SendBlankInformationBehavior】



【ReceiveBlankInformationBehavior】



【BlankInformation】



■ サンプルコード

【BlankInformationSendingWorldクラス】

```
…
public void initializeAgents() {
   //Sender エージェントの生成
   Agent sender = createAgent(BlankInformationSendingModel.AGENTTYPE_Sender);
   //Sender エージェントへの SendInformationBehavior の追加
   sender.addBehavior(BlankInformationSendingModel.BEHAVIORTYPE_SendBlankInformation);

   //Receiver エージェントの生成
   Agent receiver = createAgent(BlankInformationSendingModel.AGENTTYPE_Receiver);
   //Receiver エージェントへの ReceiveInformationBehavior の追加
   receiver.addBehavior(
      BlankInformationSendingModel.BEHAVIORTYPE_ReceiveBlankInformation);

   //Sender エージェントから Receiver エージェントへの関係の追加
   sender.addRelation(BlankInformationSendingModel.RELATIONTYPE_ToReceiver, receiver);
}
…
```

【SendBlankInformationBehavior クラス】

```
…
protected void sendBlankInformationAction() {
   //空の情報の生成
   BlankInformation blankInformation = new BlankInformation();

   //その情報の送信
   this.sendInformation(BlankInformationSendingModel.RELATIONTYPE_ToReceiver,
      BlankInformationSendingModel.BEHAVIORTYPE_ReceiveBlankInformation,
       blankInformation);
}
…
```

【ReceiveBlankInformationBehavior クラス】

```
…
protected void nextAction() {

   //BlankInformation を受け取ったら行うアクション

}

protected boolean isBlankInformation(Event e) {
   //送られてきた情報が、BlankInformation であれば true を返す。
   return this.getWorld().getInformationType(getReceivedInformation())
      == BlankInformationSendingModel.INFORMATIONTYPE_BlankInformation;
}
…
```

【BlankInformation クラス】

```
…
public class BlankInformation implements Information {
   //内容は空でよい
}
```

■ 関連するパターン

Information Creation: Information を生成する。
Information Sending: 内容を伴う Information を送信する。

Model Patterns

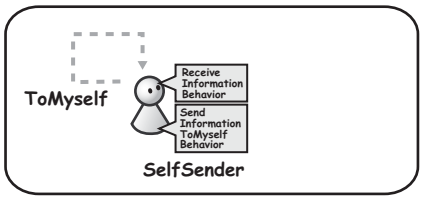
Internal Information Sending

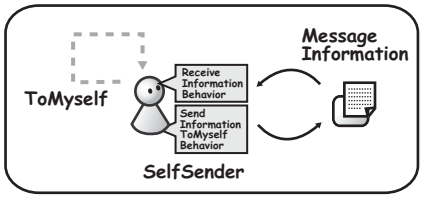
■ 目的

自分自身 (エージェント) に対して、Information を送信する。

■ 動機

エージェントが複数の行動をもっている場合、それらの行動を連携させたいことがある。例えば、他のエージェントとのコミュニケーションを行う行動が集めた情報を、意思決定・戦略行動が利用するという場合などである。行動間の単なる情報共有であれば、情報を記憶し、それを取り出すことで共有できるが、行動をアクティベートして何らかの処理をさせたい場合には、直接情報を送る方がわかりやすい。



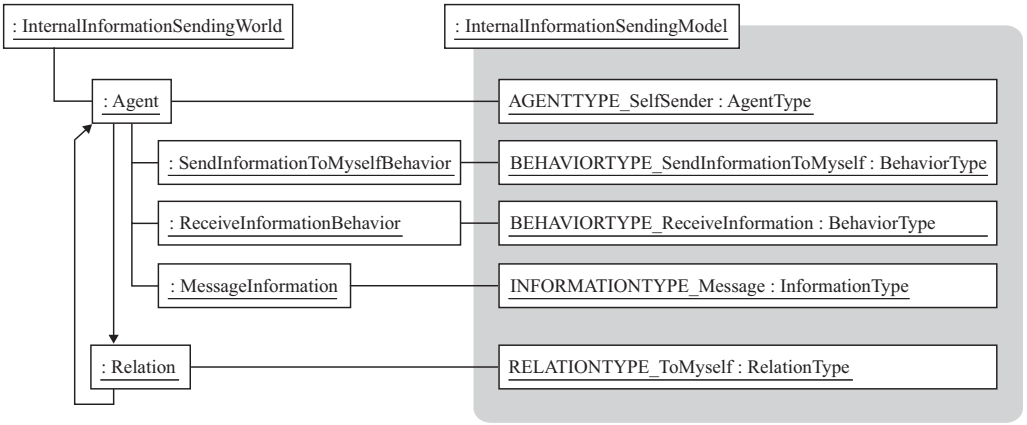


■ 基本動作

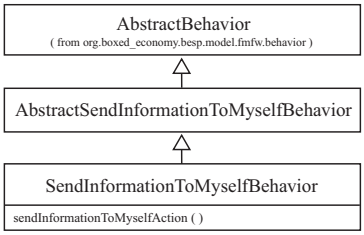
SelfSender エージェントは、SendInformationBehavior と ReceiveInformationBehavior をもっている。SendInformationBehavior では、MessageInformation を生成し、自分自身に送信する。SelfSender エージェントは、ReceiveInformationBehavior でそれを受け取る。

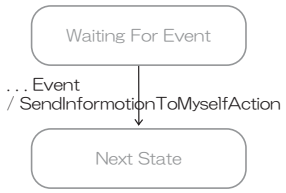
■ 設計

【全体像】

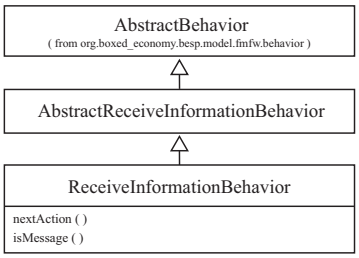


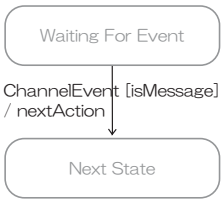
【SendInformationBehavior】



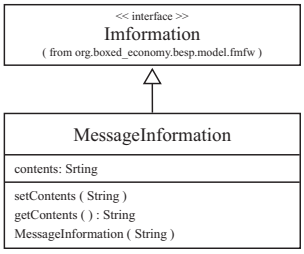


【ReceiveInformationBehavior】





【MessageInformation】



■ サンプルコード

【InternalInformationSendingWorld クラス】

```
…
public void initializeAgents() {
   //SelfSender エージェントの生成
   Agent selfSender = this.createAgent(
      InternalInformationSendingModel.AGENTTYPE_SelfSender);

   //SelfSender エージェントに、SendInformationBehavior、
   //および ReceiveInformationBehavior を追加する
   selfSender.addBehavior(
      InternalInformationSendingModel.BEHAVIORTYPE_SendInformation);
   selfSender.addBehavior(
      InternalInformationSendingModel.BEHAVIORTYPE_ReceiveInformation);

   //SelfSender の自分自身への Relation の追加
   selfSender.addRelation(
      InternalInformationSendingModel.RELATIONTYPE_ToMyself,
      selfSender);
}
…
```

【SendInformationBehavior クラス】

```
…
protected void sendInformationToMyselfAction() {
   //送信する情報の作成
   MessageInformation message = new MessageInformation("Fight!");

   //作成した情報を自分に送信する
   sendInformation(
      InternalInformationSendingModel.RELATIONTYPE_ToMyself,
      InternalInformationSendingModel.BEHAVIORTYPE_ReceiveInformation,
      message);
}
…
```

【ReceiveInformationBehavior クラス】

```
…
protected void receiveInformationAction() {
   //受信した情報を取得
   Information receivedInformation = getReceivedInformation();
   //メッセージの内容を取得 (ここでは文字列)
   String messageContents =
      ((MessageInformation)receivedInformation).getContents();
}

protected boolean isMessage(Event e) {
   //送られてきた情報が、MessageInformation であれば true を返す。
   return this.getWorld().getInformationType(getReceivedInformation())
      == InformationSendingModel.INFORMATIONTYPE_Message;
}
…
```

【MessageInformation クラス】

```
…
public class MessageInformation implements Information {
   //ここでは、内容は文字列
   String contents;

   //コンストラクタ (引数 = 文字列)
   public MessageInformation(String contents) {
      this.contents = contents;
   }

   //内容を返す
   public String getContents() {
      return contents;
   }

   //内容を設定する
   public void setContents(String contents) {
      this.contents = contents;
   }
}
…
```

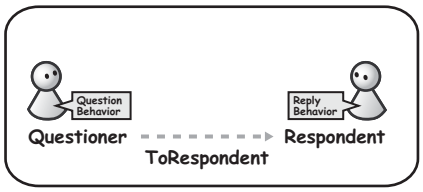
■ 関連するパターン

Information Creation: Information を生成する。
Information Sending: 他のエージェントに情報を送る。
Blank Information Sending: 内容を伴わないシグナルを送る。

Immediate Reply

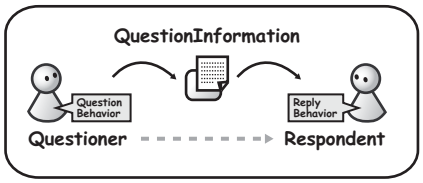
■ 目的

他のエージェントに質問し、直ちに返答を受ける。



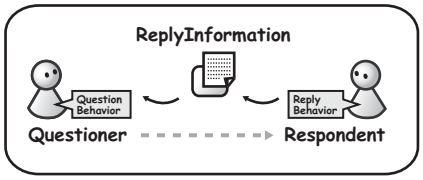
■ 動機

他のエージェントの属性等について知りたい場合に、質問をして問い合わせることがある。



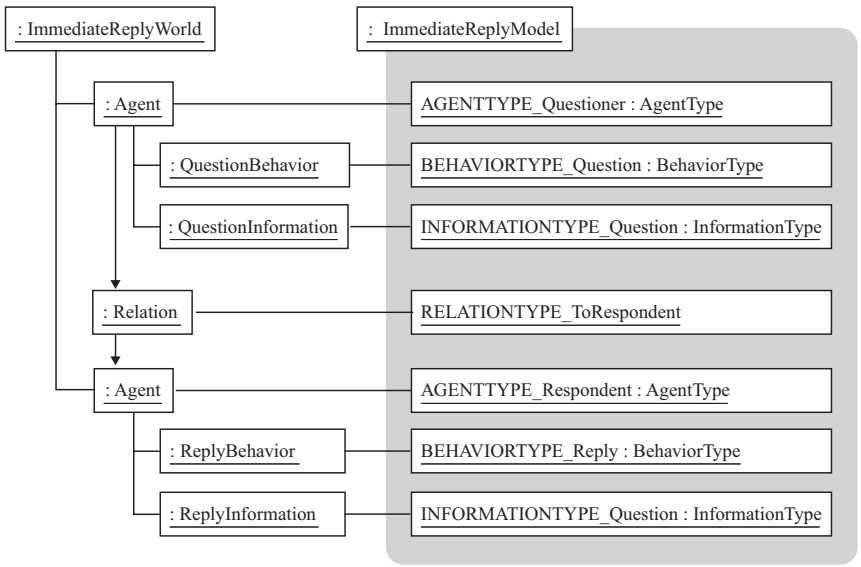
■ 基本動作

Questioner エージェントと Respondent エージェントが登場する。Questioner エージェントは、QuestionBehavior をもっており、これによって QuestionInformation を生成し (ここでは内容は空とする)、Respondent エージェントに送信する。Respondent エージェントは、ReplyBehavior でそれを受けて、直ちに ReplyInformation を送り返す (ここでは文字列の内容をもつとする)。

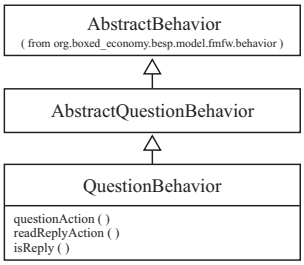


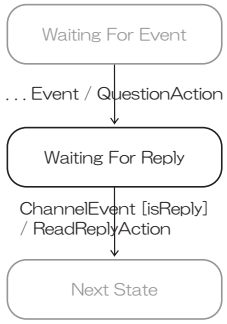
■ 設計

【全体像】

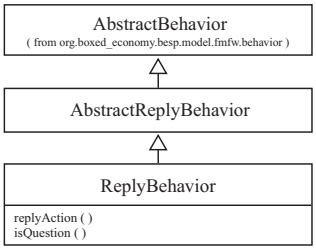


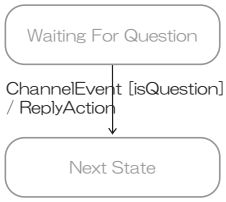
【QuestionBehavior】



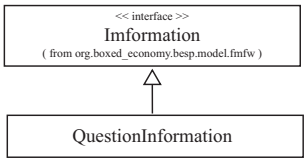


【ReplyBehavior】

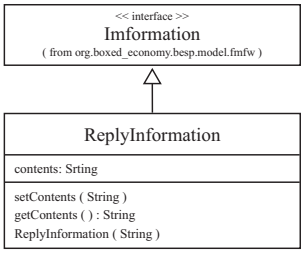




【QustionInformation】



【ReplyInformation】



■ サンプルコード

【ImmediateReplyWorld クラス】

```
…
public void initializeAgents() {
   //Questioner エージェントの生成
   Agent questioner = createAgent(ImmediateReplyModel.AGENTTYPE_Questioner);
   //Questioner エージェントへの QuestionBehavior の追加
   questioner.addBehavior(ImmediateReplyModel.BEHAVIORTYPE_Question);

   //Respondent エージェントの生成
   Agent respondent = createAgent(ImmediateReplyModel.AGENTTYPE_Respondent);
   //Respondent エージェントへの ReplyBehavior の追加
   respondent.addBehavior(ImmediateReplyModel.BEHAVIORTYPE_Reply);

   //Questioner エージェントから Reply エージェントへの関係の追加
   questioner.addRelation(ImmediateReplyModel.RELATIONTYPE_ToRespondent, respondent);
}
…
```

【QuestionBehavior クラス】

```
…
protected void questionAction() {
   //質問 (空の情報) の生成
   QuestionInformation questionInformation = new QuestionInformation();

   //質問の送信
   this.sendInformation(ImmediateReplyModel.RELATIONTYPE_ToRespondent,
      ImmediateReplyModel.BEHAVIORTYPE_Reply, questionInformation);
}

protected void readReplyAction() {
   //送られてきた返信内容の取得
   Information receivedInformation = getReceivedInformation();
   String replyContents = ((ReplyInformation)receivedInformation).getContents();
}

protected boolean isReply(Event e) {
   //送られてきた情報が ReplyInformation であれば true を返す
   return this.getWorld().getInformationType(getReceivedInformation())
      == ImmediateReplyModel.INFORMATIONTYPE_Reply;
}
…
```

【ReplyBehavior クラス】

```
…
protected void replyAction() {
   //返信情報の作成
   ReplyInformation replyInformation = new ReplyInformation("My answer is Yes.");

   //その返信情報の送信 (現在開いている Channel へ送信)
   this.sendInformation(replyInformation);
}

protected boolean isQuestion(Event e) {
```

```
   //送られてきた情報が QuestionInformation であれば true を返す
   return this.getWorld().getInformationType(getReceivedInformation())
      == ImmediateReplyModel.INFORMATIONTYPE_Question;
}
…
```

【QuestionInformation クラス】

```
…
public class QuestionInformation implements Information {
   //ここでは、質問の内容は空
}
```

【ReplyInformation クラス】

```
…
public class ReplyInformation implements Information {
   //ここでは、内容は文字列
   String contents;

   //コンストラクタ (引数 = 文字列)
   public ReplyInformation(String contents) {
      this.contents = contents;
   }

   //内容を返す
   public String getContents() {
      return contents;
   }

   //内容を設定する
   public void setContents(String contents) {
      this.contents = contents;
   }
}
…
```

■ バリエーション

このサンプルでは、質問を送る行動 (QuestionBehavior) が返答を受け取ったが、これらを分離することもできる。例えば、ReceiveReplyBehavior のように、受取り専用の行動をつくることができる。その場合には、ReplyBehavior は、返答情報の送り先として、明示的に ReceiveReplyBehavior を指定する必要がある。もし事前に ReplyBehavior が返信先を知っていたくない場合 (行動コンポーネントのの独立性のため) には、QuestionBehavior が返信先を指定することもできる (Appointed Destination Reply パターン)。

■ 関連するパターン

Information Creation: Information を生成する部分で使用。
Blank Information Sending: 空のメッセージを送信する場合。
Collect Immediate Replies: 複数のエージェントから、直ちに返答を集める場合。この Immediate Reply パターンを内包している。

Collect Immediate Replies

■ 目的

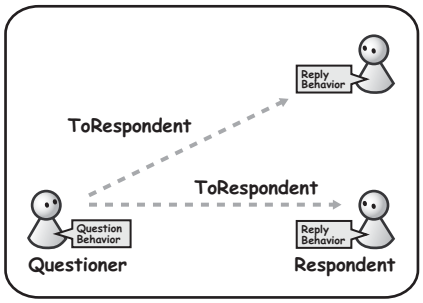
複数のエージェントに質問し、直ちに返された返答を集める。

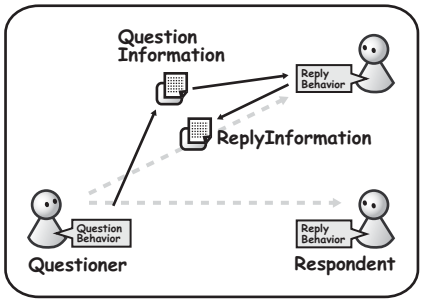
■ 動機

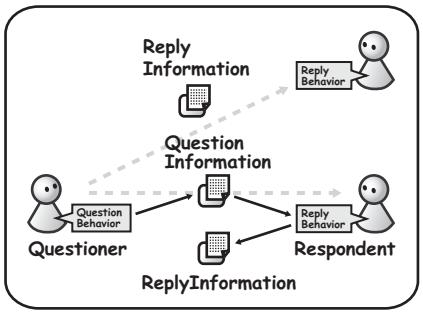
複数のエージェントの属性等を知りたい場合に、質問をして問い合わせることがある。例えば、注文を集める場合、データの合計・平均を調査したい場合などがこれにあたる。

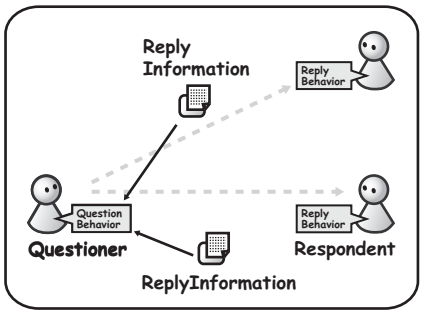
■ 基本動作

Questioner エージェントと複数の Respondent エージェントが登場する。Questioner エージェントは、QuestionBehavior をもっており、これによって QuestionInformation を生成し (内容は空でもよい)、連続して Respondent エージェントに送信していく。Respondent エージェントは、ReplyBehavior でそれを受けて、直ちに ReplyInformation を送り返す。ここで、送り返された ReplyInformation は、一度、Questioner のスタックにたまり、次の Respondent エージェントの ReplyBehavior が実行されて、その返信も、ためられる。すべての Respondent エージェントの返答が終った時点で、Questioner エージェントは、スタックにたまったものをまとめて処理する。



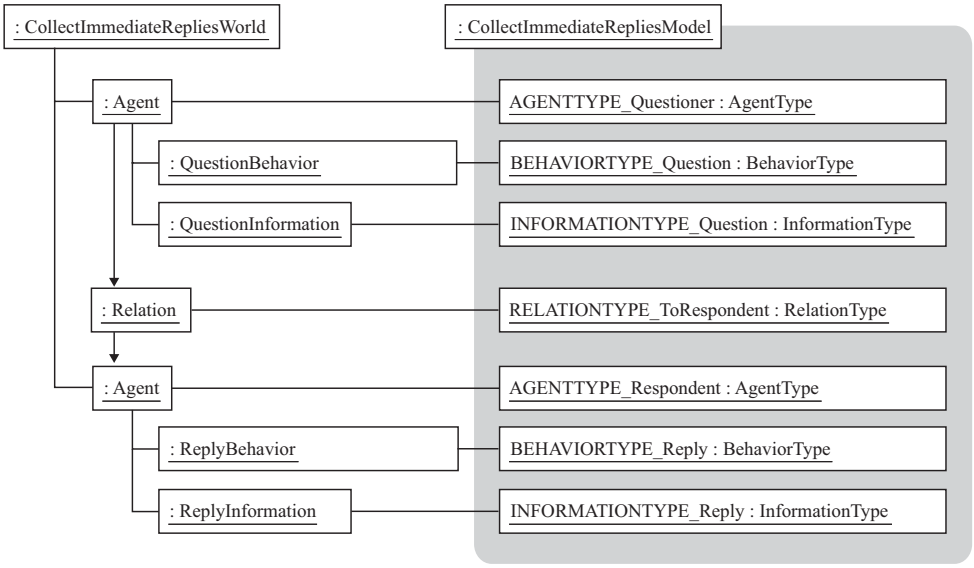




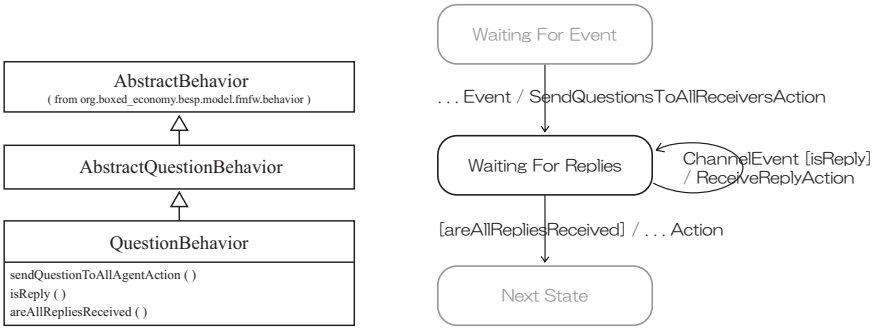


■ 設計

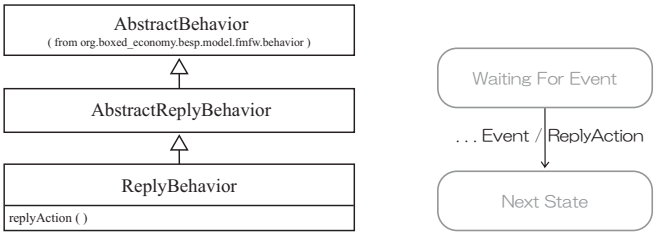
【全体像】



【QuestionBehavior】



【ReplyBehavior】



■ サンプルコード

【CollectImmediateRepliesWorld クラス】

```
…
public void initializeAgents() {

   //Questioner エージェントの生成
   Agent questioner =
      this.createAgent(CollectImmediateRepliesModel.AGENTTYPE_Questioner);

   //Questioner エージェントへの QuestionBehavior の追加
   questioner.addBehavior(CollectImmediateRepliesModel.BEHAVIORTYPE_Question);

   //Respondent エージェントの生成（複数人）
   for (int i = 0; i < 10; i++) {
      //Respondent エージェントの生成
      Agent respondent =
         this.createAgent(CollectImmediateRepliesModel.AGENTTYPE_Respondent);

      //Respondent エージェントへの ReplyBehavior の追加
      respondent.addBehavior(
         CollectImmediateRepliesModel.BEHAVIORTYPE_Reply);

      //Questioner エージェントの Respondent エージェントへの Relation の追加
      questioner.addRelation(
         CollectImmediateRepliesModel.RELATIONTYPE_ToRespondent,respondent);
   }
}
…
```

【QuestionBehavior クラス】

```
…
   //返信の残り数のカウンタ（クラス変数）
   private int receiveCount = 0;
…

protected void sendQuestionToAllAgentAction() {
   // ToRespondent 関係のエージェント全員に対して、
   // QuestionInformation を送り、Respondent の人数を receiveCount に記録する
   this.receiveCount = this.sendInformation(
      CollectImmediateRepliesModel.RELATIONTYPE_ToRespondent,
      CollectImmediateRepliesModel.BEHAVIORTYPE_Reply,
      new QuestionInformation());
}

protected void receiveReplyAction() {
   private List replies = new ArrayList();

   //受け取った ReplyInformation を記録する
   replies.add(this.getReceivedInformation());

   //返信の残り数を減らす
   this.receiveCount--;
}

protected boolean isReply(Event e) {
   //ReplyInformation を受け取った場合に true を返す
```

```
   return this.getWorld().getInformationType(this.getReceivedInformation())
      == CollectImmediateRepliesModel.INFORMATIONTYPE_Reply;
}

protected boolean areAllRepliesReceived(Event e) {
   //QuestionInformation を送った数だけ、返信を受け取った場合に true を返す。
   return this.receiveCount == 0;
}
 …
```

【ReplyBehavior クラス】

```
 …
protected void replyAction() {
   //ReplyInformation を送り返す
   this.sendInformation(new ReplyInformation());
}

protected boolean isQuestion(Event e) {
   //QuestionInformation を受け取った場合、true を返す
   return this.getWorld().getInformationType(this.getReceivedInformation())
      == CollectImmediateRepliesModel.INFORMATIONTYPE_Question;
}
 …
```

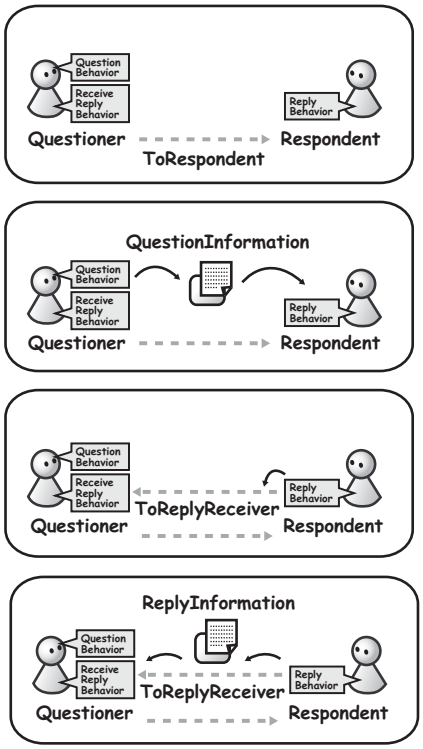
■ 関連するパターン

Information Creation: Information を生成する部分で使用。
Blank Information Sending: 空のメッセージを送信する場合。
Immediate Reply: 一人のエージェントに対して。Collect Immediate Replies パターンは、この Immediate Reply パターンを内包している。

Appointed Destination Reply

■ 目的

質問を送ったエージェントに、指定した宛先 (Appointed Destination) に返答してもらう。



■ 動機

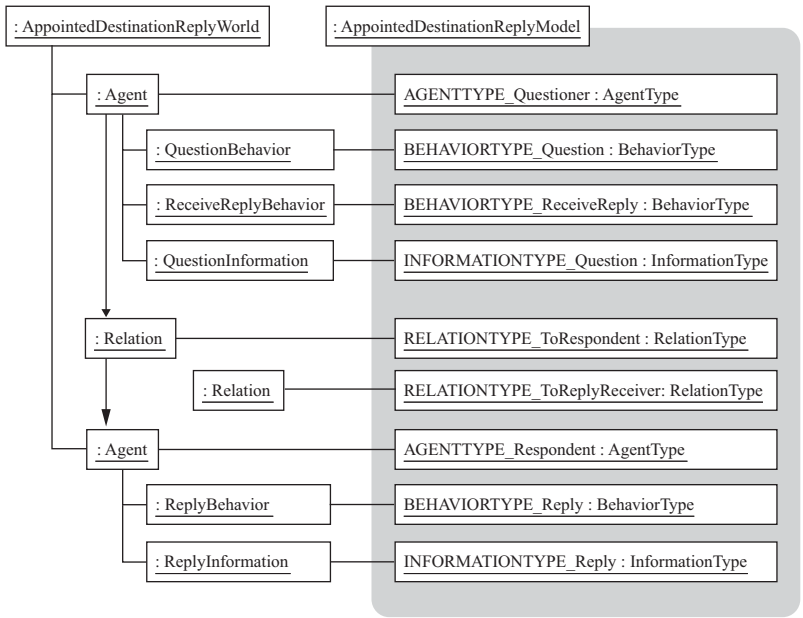
他のエージェントに質問して返答を求める際に、質問した行動以外の行動に返信を求めたいことがある。例えば、質問を受けてから返答するまでの間に時間を要する場合や、返信内容によって返信先が異なる場合などがある。

■ 基本動作

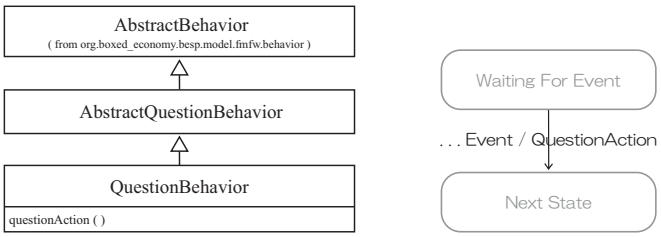
Questioner エージェントと Respondent エージェントが登場する。Questioner エージェントは、QuestionBehavior をもっており、これによって、返答先の Agent と BehaviorType を含む QuestionInformation を生成し、Respondent エージェントに送信する。Respondent エージェントは、ReplyBehavior でそれを受けて、その返答先に ReplyInformation を送る。ReplyBehavior は、ReceiveReplyBehavior のことを事前に知っている必要はない。

■ 設計

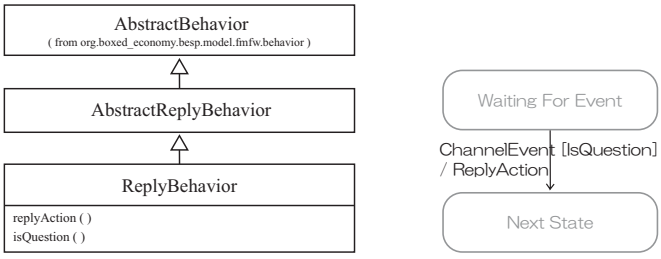
【全体像】



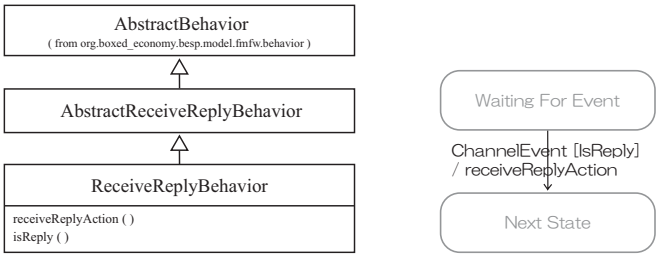
【QuestionBehavior】



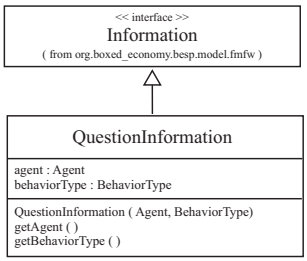
【ReplyBehavior】



【ReceiveReplyBehavior】



【QuestionInformation】



■ サンプルコード

【AppointedDistinationReplyWorld クラス】

```
…
public void initializeAgents() {
   //Questioner エージェントの生成
   Agent questioner = createAgent(AppointedDestinationReplyModel.AGENTTYPE_Questioner);
   //Questioner エージェントへの QuestionBehavior と ReceiveReplyBehavior の追加
   questioner.addBehavior(AppointedDestinationReplyModel.BEHAVIORTYPE_Question);
   questioner.addBehavior(AppointedDestinationReplyModel.BEHAVIORTYPE_ReceiveReply);

   //Respondent エージェントの生成
   Agent respondent = createAgent(AppointedDestinationReplyModel.AGENTTYPE_Respondent);
   //Respondent エージェントへの ReplyBehavior の追加
   respondent.addBehavior(AppointedDestinationReplyModel.BEHAVIORTYPE_Reply);

   //Questioner エージェントの Respondent エージェントへの Relation の追加
   questioner.addRelation(
      AppointedDestinationReplyModel.RELATIONTYPE_ToRespondent, respondent);
}
…
```

【QuestionBehavior クラス】

```
…
   protected void questionAction() {
   //質問情報の作成
   QuestionInformation question = new QuestionInformation(this.getAgent(),
      AppointedDestinationReplyModel.BEHAVIORTYPE_ReceiveReply);

   //質問情報を送る
   this.sendInformation(AppointedDestinationReplyModel.RELATIONTYPE_ToRespondent,
      AppointedDestinationReplyModel.BEHAVIORTYPE_Reply, question);
}
…
```

【ReplyBehavior クラス】

```
…
protected void replyAction() {
   //受け取った情報を取得する
   QuestionInformation question = (QuestionInformation) this.getReceivedInformation();

   //質問情報から返信先のエージェントと行動を調べる
   Agent appointedAgent = question.getAgent();
   BehaviorType appointedBehaviorType = question.getBehaviorType();

   //指定された返信先エージェントに関係を結ぶ
   this.getAgent().addRelation(
      AppointedDestinationReplyModel.RELATIONTYPE_ToReplyReceiver, appointedAgent);

   //指定された返信先エージェントの指定された行動に返信を送る
   this.sendInformation(AppointedDestinationReplyModel.RELATIONTYPE_ToReplyReceiver,
      appointedBehaviorType, new ReplyInformation());
}

protected boolean isQuestion(Event e) {
```

```
   //QuestionInformation を受け取った場合に true を返す
   return this.getWorld().getInformationType(this.getReceivedInformation())
      == CollectImmediateRepliesModel.INFORMATIONTYPE_Question;
}
…
```

【ReceiveReplyBehavior クラス】

```
…
protected void receiveReplyAction() {
   //受信した情報を取得
   Information receivedInformation = getReceivedInformation();
}

protected boolean isReply(Event e) {
   //送られてきた情報が ReplyInformation であれば true を返す
   return this.getWorld().getInformationType(getReceivedInformation())
      == InformationSendingModel.INFORMATIONTYPE_Reply;
}
…
```

【QuetionInformation クラス】

```
…
public class QuestionInformation implements Information {
   private Agent agent;
   private BehaviorType behaviorType;

   //コンストラクタ（引数 = Agent, BehaviorType）
   public QuestionInformation(Agent agent, BehaviorType behaviorType) {
      this.agent = agent;
      this.behaviorType = behaviorType;
   }

   public Agent getAgent() {
      return this.agent;
   }

   public BehaviorType getBehaviorType() {
      return this.behaviorType;
   }
}
…
```

■ 関連するパターン

Information Creation: Information を生成する。
Collect Immediate Replies: 複数のエージェントから直ちに返答を集める。

Super BehaviorType Calling

■ 目的

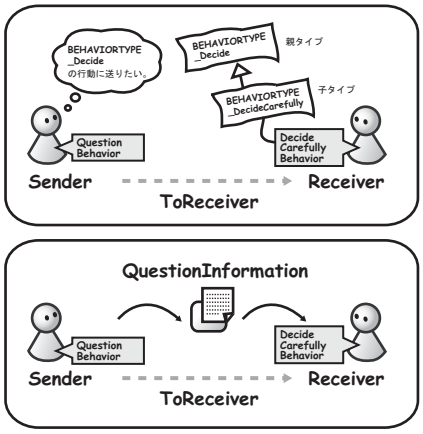
具体的行動ではなく、その行動の親 BehaviorType で指定したい。

■ 動機

戦略行動のように、同種の行動であるが内容が異なるという行動を扱いたいことがある。これらの行動は、シミュレーションの実行中に、状況に応じて切り替えることが多い。そのため、他の行動が、これらの行動と連携・コミュニケーションをはかりたい場合には、指定の仕方に工夫が必要となる。

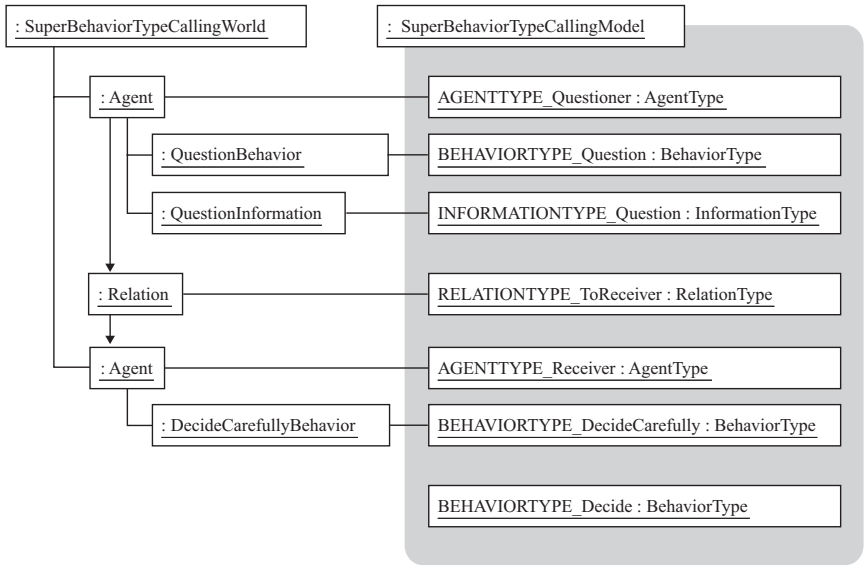
■ 基本動作

Questioner エージェントと Receiver エージェントが登場する。Questioner エージェントは、QuestionBehavior をもっており、それによって QuestionInformation を作成し、Receiver エージェントに送信する。そのとき、送信先は、Receiver エージェントの「DecideBehaviorType」の行動になっている。Receiver エージェントは、この DecideBehaviorType の子タイプ「DecideCarefullyBehaviorType」の行動 (DecideCarefullyBehavior) をもっており、この DecideCarefullyBehavior で、QuestionInformation を受け取る。

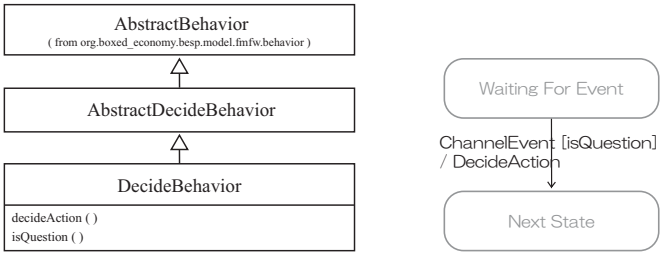


■ 設計

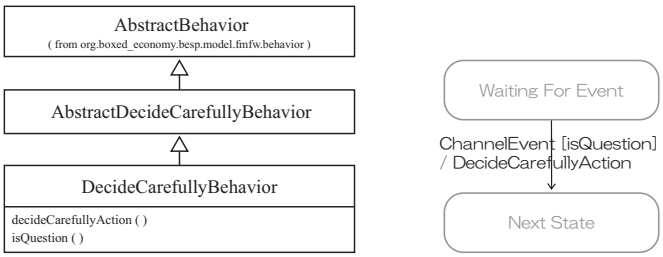
【全体像】



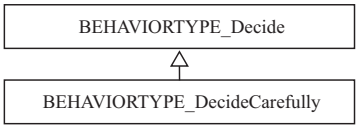
【DecideBehavior】



【DecideCarefullyBehavior】



【タイプの親子関係】



■ サンプルコード

【SuperBehaviorTypeCallingModel クラス】

```
…
private static void initializeTypes(BESPContainer container) {
   //BEHAVIORTYPE_DecideCarefully を、BEHAVIORTYPE_Decide の子タイプに設定
   BEHAVIORTYPE_Decide.addChild(BEHAVIORTYPE_DecideCarefully);
}
…
```

【SuperBehaviorTypeCallingWorld クラス】

```
…
public void initializeAgents() {
   //Questioner エージェントの生成
   Agent questioner = this.createAgent(
      SuperBehaviorTypeCallingModel.AGENTTYPE_Questioner);
   //Questioner エージェントへの QuestionBehavior の追加
   questioner.addBehavior(SuperBehaviorTypeCallingModel.BEHAVIORTYPE_Question);

   //Receiver エージェントの生成
```

```
   Agent receiver = this.createAgent(
      SuperBehaviorTypeCallingModel.AGENTTYPE_Receiver);
   //Receiver エージェントへの DecideCarefullyBehavior の追加
   receiver.addBehavior(
      SuperBehaviorTypeCallingModel.BEHAVIORTYPE_DecideCarefully);

   //Questioner エージェントの Receiver エージェントへの Relation の追加
   questioner.addRelation(
      SuperBehaviorTypeCallingModel.RELATIONTYPE_ToReceiver, receiver);
}
…
```

【QuestionBehavior クラス】

```
…
protected void questionAcrion() {

   //Receiver エージェントに、QuestionInformation を送る。
   this.sendInformation(
      SuperBehaviorTypeCallingModel.RELATIONTYPE_ToReceiver,
      SuperBehaviorTypeCallingModel.BEHAVIORTYPE_Decide,
      new QuestionInformation());
}
…
```

【DecideCarefullyBehavior クラス】

```
…
protected void decideCarefullyAction() {
   //慎重に意思決定する
}

protected boolean isQuestion(Event e) {
   //QuestionInformation を受け取った時に true を返す
   return this.getWorld().getInformationType(this.getReceivedInformation())
      == SuperBehaviorTypeCallingModel.INFORMATIONTYPE_Question;
}
…
```

■ バリエーション

このサンプルでは、DecideCarefullyBehavior だけを用意し、DecideBehavior は用意しなかった。もし複数の Decide〜Behavior があり、それらのプログラムに共通部分があるならば、これらの Behavior の親クラスとして、DecideBehavior を定義してもよい。この場合には、BehaviorType の親子関係とは別に、プログラム的な汎化関係を用いるということである。

■ 関連するパターン

Information Sending: 他のエージェントに情報を送る。
Internal Information Sending: 自分の持っている他の行動に情報を送る。

Model Patterns

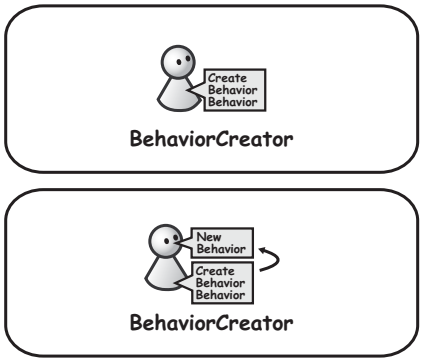
Behavior Creation

■ 目的

自分自身に新しい行動を生成・追加する。

■ 動機

エージェントの役割が動的に変化するモデルや、状況に応じた振舞いをするモデルでは、シミュレーション実行中に、そのエージェントがいままで持っていなかった行動を生成し追加する必要がある。

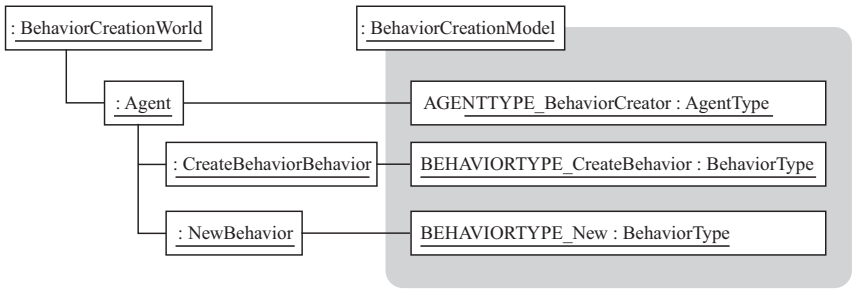


■ 基本動作

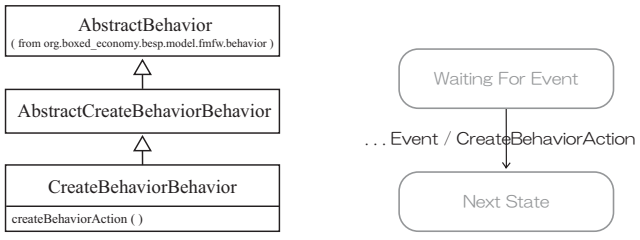
BehaviorCreator エージェントは CreateBehaviorBehavior をもっている。この CreateBehaviorBehavior は、NewBehavior を生成する。その結果、BehaviorCreator エージェントは、新たに NewBehavior を持つことになる。

■ 設計

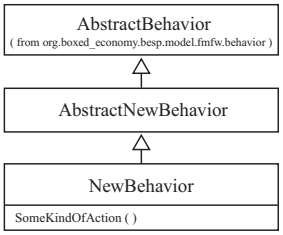
【全体像】



【CreateBehaviorBehavior】



【NewBehavior】



※この行動の内容は、文脈に合わせて作成する。

■ サンプルコード

【BehaviorCreationWorld クラス】

```
…
public void initializeAgents() {
   //BehaviorCreator エージェントの生成
   Agent behaviorCreator =
      createAgent(BehaviorCreationModel.AGENTTYPE_BehaviorCreator);

   //そのエージェントへの CreateBehaviorBehavior の追加
   behaviorCreator.addBehavior(BehaviorCreationModel.BEHAVIORTYPE_CreateBehavior);
}
…
```

【CreateBehaviorBehavior クラス】

```
…
protected void createBehaviorAction() {
   //NewBehavior の追加
   this.getAgent().addBehavior(BehaviorCreationModel.BEHAVIORTYPE_New);
}
…
```

■ 関連するパターン

Behavior Switching: 新しいエージェントが持っている行動を削除し、新しい行動を追加する。
Temporary Behavior Creation: 生成・追加する行動が、一時的に処理を行って自動消滅する場合。

Behavior Destruction

■ 目的

エージェントが持っている行動を削除する。

■ 動機

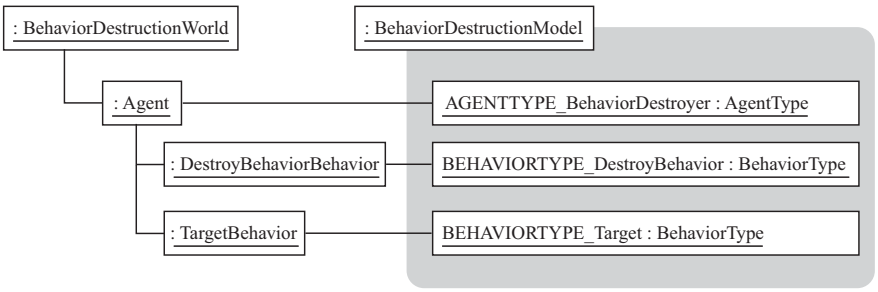
エージェントの役割が動的に変化するモデルや、状況に応じた振舞いをするモデルでは、シミュレーション実行中に、そのエージェントが持っている行動を削除したいことがある。現在もっている行動を、強制的に終了したり、同種の別の行動に置き換えたりするためである。

■ 基本動作

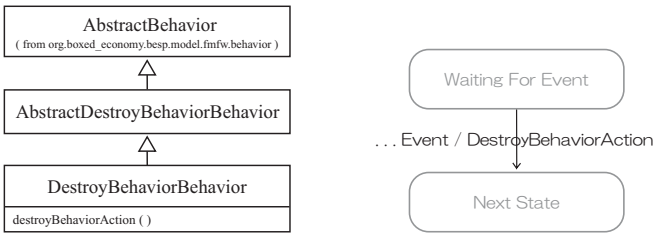
BehaviorDestroyer エージェントは TargetBehavior と DestroyBehaviorBehavior を持っている。DestroyBehaviorBehavior によって、TargetBehavior を削除する (TargetBehavior の状態にかかわらず、外部から強制的に削除する)。

■ 設計

【全体像】



【DestroyBehaviorBehavior】



■ サンプルコード

【BehaviorDestructionWorld クラス】

```
…
public void initializeAgents() {
   //BehaviorDestroyer エージェントの生成
   Agent behaviorDestroyer =
      createAgent(BehaviorDestructionModel.AGENTTYPE_BehaviorDestroyer);

   //そのエージェントへの DestroyBehaviorBehavior の追加
   behaviorDestroyer.addBehavior(
      BehaviorDestructionModel.BEHAVIORTYPE_DestroyBehavior);

   //そのエージェントへの削除対象動の追加
   behaviorDestroyer.addBehavior(
      BehaviorDestructionModel.BEHAVIORTYPE_Target);
}
…
```

【DestroyBehaviorBehavior クラス】

```
…
protected void destroyBehaviorAction() {
   //Behavior の削除
   this.getAgent().removeBehavior(this.getAgent()
      .getBehavior(BehaviorDestructionModel.BEHAVIORTYPE_Target));
}
…
```

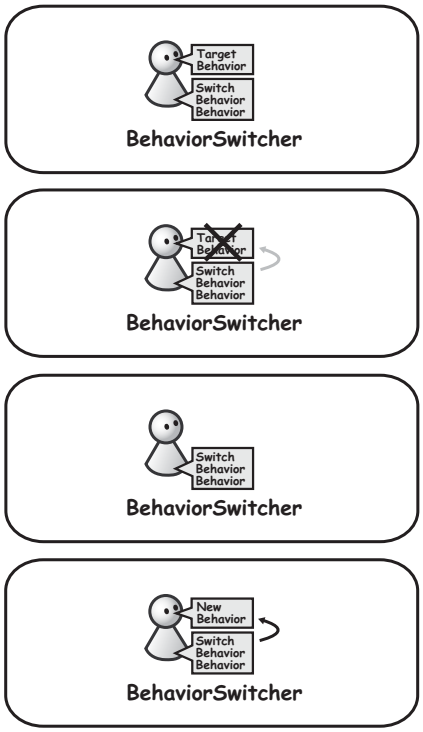
■ 関連するパターン

Behavior Switching: ある行動を他の行動に切り替える場合 (現在持っている行動を削除し、新しい行動を追加する)。

Behavior Switching

■ 目的

エージェントが現在持っている行動を、新しい他の行動に切り替える。



■ 動機

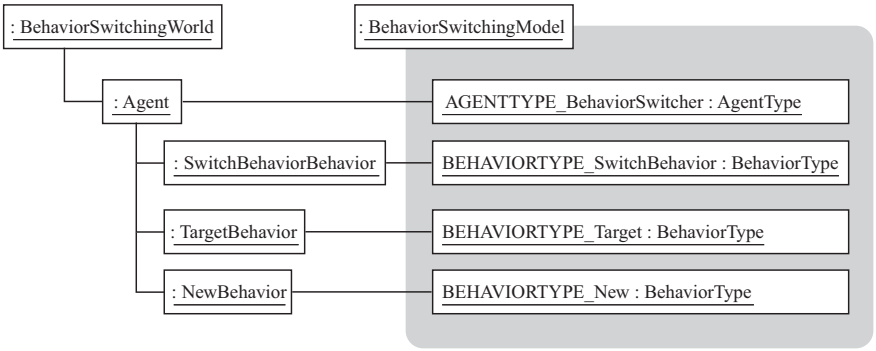
エージェントの役割が動的に変化するモデルや、状況に応じた振舞いをするモデルでは、シミュレーション実行中に、そのエージェントが持っている行動を切り替える必要がある。特に典型的な例としては、戦略 (行動) の切り替えがある。

■ 基本動作

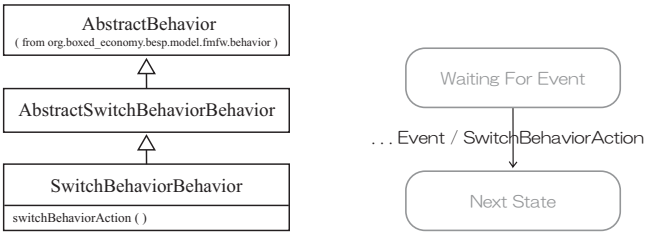
BehaviorSwitcher エージェントは TargetBehavior と SwitchBehaviorBehavior を持っている。SwitchBehaviorBehavior によって、TargetBehavior を削除し、SwitchBehaviorBehavior によって、NewBehavior を追加する。

■ 設計

【全体像】



【SwitchBehaviorBehavior】



■ サンプルコード

【BehaviorSwitchingWorld クラス】

```
…
public void initializeAgents() {
   //エージェントの生成
   Agent behaviorSwitcher =createAgent(
      BehaviorSwitchingModel.AGENTTYPE_BehaviorSwitcher);

   //そのエージェントへの SwitchBehavior 行動と、切り替え前行動の追加
   behaviorSwitcher.addBehavior(
      BehaviorSwitchingModel.BEHAVIORTYPE_SwitchBehavior);
   behaviorSwitcher.addBehavior(BehaviorSwitchingModel.BEHAVIORTYPE_Target);
}
…
```

【SwitchBehaviorBehavior クラス】

```
…
protected void switchBehaviorAction() {
   //切り替え前の行動の削除
   this.getAgent().removeBehavior(
      this.getAgent().getBehavior(BehaviorSwitchingModel.BEHAVIORTYPE_Target));
   //切り替え後の行動の追加
   this.getAgent().addBehavior(BehaviorSwitchingModel.BEHAVIORTYPE_New);
}
…
```

■ バリエーション

このサンプルでは、ソースコード中に明示的に切り替え後の BehaviorType を指定しているが、BehaviorType を情報として入手し、それに応じて切り替え後の行動を決めるということもできる。

なお、戦略行動のように、同種の行動であるが内容が異なるという行動を切り替えることがある (その場合には、Super BehaviorType Calling パターンを使って行動のアクティベーションが行われていると思われる)。このような場合には、切り替え前の行動の削除の際に、親 BehaviorType を指定して削除することができる。これにより、切り替え前の行動が具体的に何であるかを意識することなく、削除することができる。

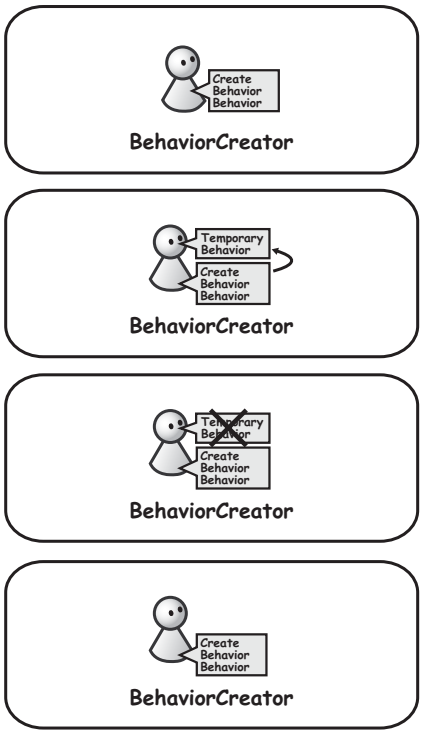
■ 関連するパターン

Behavior Destruction: 行動を削除する (切り替え前の行動を削除する際に用いる)。
Behavior Creation: 行動を生成する (切り替え後の行動を生成する際に用いる)。
Temporary Behavior Creation: 一時的に行われる行動を生成する (切り替え後の行動が一時的な行動である場合は、これを用いる)。
Super BehaviorType Calling: 具体的な BehaviorType ではなく、親 BehaviorType で指定する (行動を切り替えても、動作するために必要となる)。

Temporary Behavior Creation

■ 目的

一時的に行う行動を追加する。



■ 動機

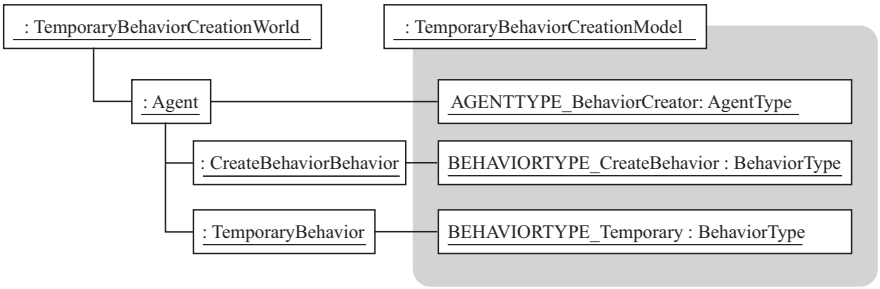
付加的な処理や例外的な処理などのように、エージェントが一時的に行う行動を追加したいことがある。そのような行動は、通常の行動とは別に動き、処理が終了したときには自働的に消滅させたい。

■ 基本動作

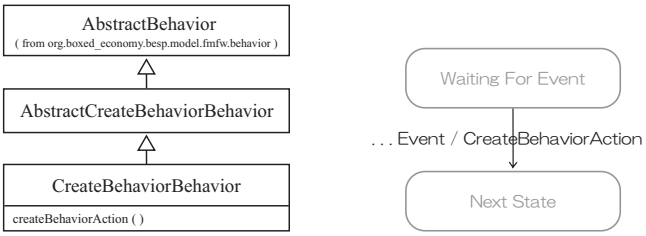
BehaviorCreator エージェントは CreateBehaviorBehavior をもっている。CreateBehaviorBehavior は TemporaryBehavior を生成・追加する。TemporaryBehavior は、処理を完了すると、終了状態に遷移して自ら消滅する。

■ 設計

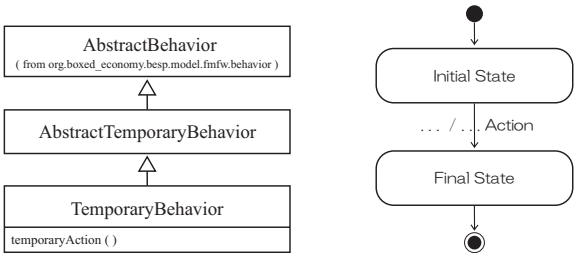
【全体像】



【CreateBehaviorBehavior】



【TemporaryBehavior】



■ サンプルコード

【TemporaryBehaviorCreationWorld クラス】

```
…
public void initializeAgents() {
   //BehaviorCreator エージェントの生成
   Agent behaviorCreator = createAgent(
      TemporaryBehaviorCreationModel.AGENTTYPE_BehaviorCreator);

   //そのエージェントへの CreateBehavior 行動の追加
   behaviorCreator.addBehavior(
      TemporaryBehaviorCreationModel.BEHAVIORTYPE_CreateBehavior);
}
…
```

【CreateBehaviorBehavior クラス】

```
…
protected void createBehaviorAction() {
   //TemporaryBehavior の追加
   this.getAgent().addBehavior(
      TemporaryBehaviorCreationModel.BEHAVIORTYPE_Temporary);
}
…
```

■ バリエーション

TemporaryBehavior が「モデル上の時間」を要する場合には、TimeEvent を必要な数だけ受けた後に終了するという拡張を行う。

■ 関連するパターン

Behavior Creation: 新しい行動を生成・追加する。
Behavior Request Attachment: 何の行動を追加するのかを、他のエージェントから指定される場合。
Time-Consuming Behavior: 遂行するのに一定の時間がかかる Behavior を表現する。

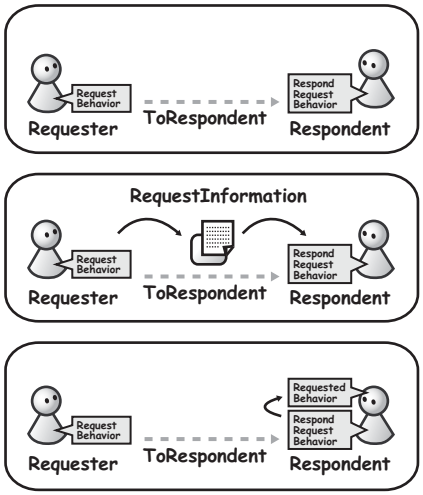
Requested Behavior Attachment

■ 目的

こちらが指定した BehaviorType の行動を、他のエージェントに生成・追加してもらう。

■ 動機

相手にコミュニケーションの手順を合わせてほしい場合がある。例えば、独特の手順で買い物をする店では、来店した顧客にその手順を知らせて、それに合せて行動してもらう必要がある。また、新しい役割を委譲する場合などにも、こちらが指定した行動を、相手にもってもらう必要がある。そのほか、環境エージェントが、対象となるエージェントの行動をアフォードさせたいことがある。

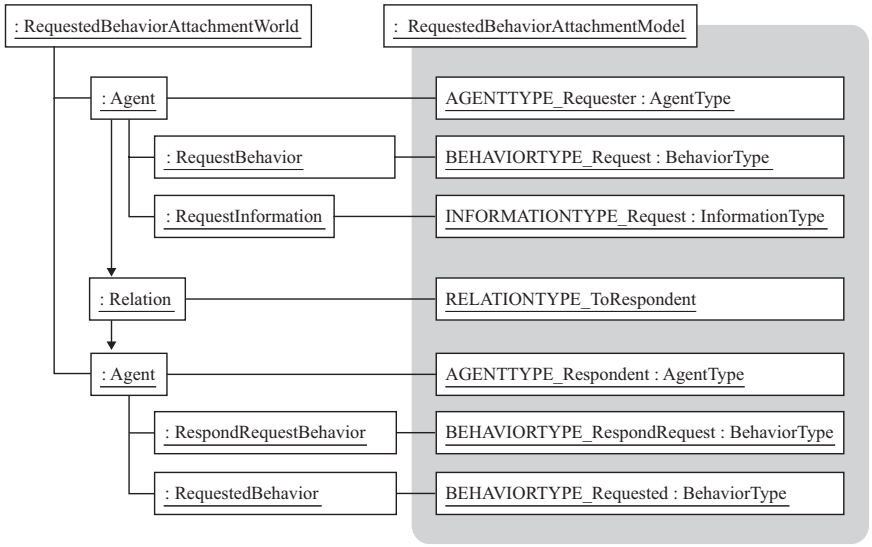


■ 基本動作

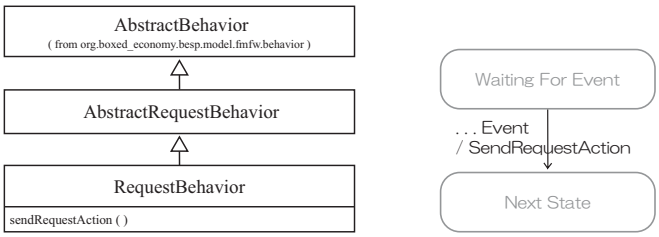
Requester エージェントと Respondent エージェントが登場する。Requester エージェントは、RequestBehavior によって、相手にもってほしい BehaviorType を相手に送信する (Type は Information の一種なので、そのまま送信することができる)。Respondent エージェントは、RespondRequestBehavior をもっており、RequestInformation を受けて RequestedBehavior を生成する。

■ 設計

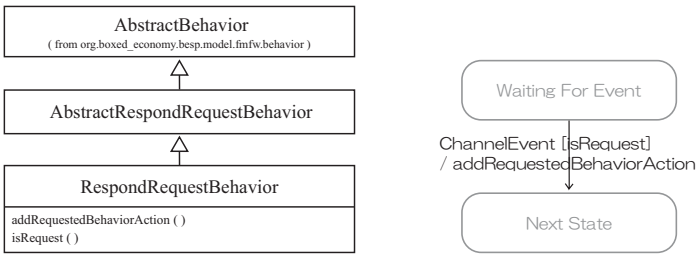
【全体像】



【RequestBehavior】



【RespondRequestBehavior】



■ サンプルコード

【RequestedBehaviorAttachmentWorld クラス】

```
…
public void initializeAgents() {
   //Requester エージェントを生成する
   Agent requester = createAgent(
      RequestedBehaviorAttachmentModel.AGENTTYPE_Requester);

   //Requester エージェントに、RequestBehavior を追加する
   requester.addBehavior(
      RequestedBehaviorAttachmentModel.BEHAVIORTYPE_Request);

   //Respondent エージェントを生成する
   Agent respondent = createAgent(
      RequestedBehaviorAttachmentModel.AGENTTYPE_Respondant);

   //Respondent エージェントに、RespondRequestBehavior を追加する
   respondent.addBehavior(
      RequestedBehaviorAttachmentModel.BEHAVIORTYPE_RespondRequest);

   //Requester エージェントに、Respondent エージェントへの Relation を追加する
   requester.addRelation(
      RequestedBehaviorAttachmentModel.RELATIONTYPE_ToRespondent, respondent);
}
…
```

【RequestBehavior クラス】

```
…
protected void sendRequestAction() {
   //Respondent エージェントへ追加したい BehaviorType を送る
   sendInformation(
      RequestedBehaviorAttachmentModel.RELATIONTYPE_ToRespondent,
      RequestedBehaviorAttachmentModel.BEHAVIORTYPE_RespondRequest,
      RequestedBehaviorAttachmentModel.BEHAVIORTYPE_Requested);
}
…
```

【RespondRequestBehavior クラス】

```
…
protected void addRequestedBehaviorAction() {
   //送られてきた BehaviorType を取得する
   BehaviorType sentBehaviorType = (BehaviorType) this.getReceivedInformation();

   //送られてきた BehaviorType を自分自身に追加する
   this.getAgent().addBehavior(sentBehaviorType);
}

protected boolean isBehaviorType(Event e) {
   //送られてきた情報が BehaviorType であれば true を返す
   return getReceivedInformation() instanceof BehaviorType;
}

…
```

■ バリエーション

このサンプルでは、持ってほしいといわれた BehaviorType の行動を無条件に追加するが、受け手側 (Respondent エージェント) の判断に基づいて、追加するかどうかを決定することもできる。

また、送られてくる BehaviorType は、具体的な BehaviorType である必要はなく、その子 BehaviorType の行動を選択して追加させることもできる。例えば、BEHAVIORTYPE_Strategy という戦略行動をもつように言われた場合、その子 BehaviorType である BEHAVIORTYPE_StrategyA や BEHAVIORTYPE_StrategyB の行動を追加することができる (その場合、Super BehaviorType Calling パターンを使って、行動のアクティベーションが行われるだろう)。

■ 関連するパターン

Behavior Creation: 行動を生成・追加する。
Information Sending: 情報を送る。
Forced Behavior Attachment: 行動を付加するように依頼するのではなく、外部から強制的に行動を付加する (この場合には、相手の意思を介在させる余地はないが、受け手側が RespondRequestBehavior を持っている必要はない)。

Model Patterns

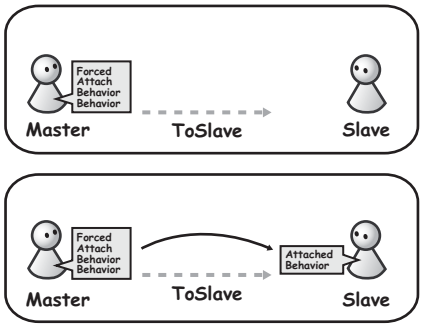
Forced Behavior Attachment

■ 目的

こちらが指定した BehaviorType の行動を、他のエージェントに強制的に追加する。

■ 動機

相手にコミュニケーションの手順を合わせてほしい場合がある。例えば、独特の手順で買い物をする店では、来店した顧客にその手順を知らせて、それに合せて行動してもらう必要がある。また、新しい役割を委譲する場合などにも、こちらが指定した行動を、相手にもってもらう必要がある。そのほか、環境エージェントが、対象となるエージェントの行動をアフォードさせたいことがある。

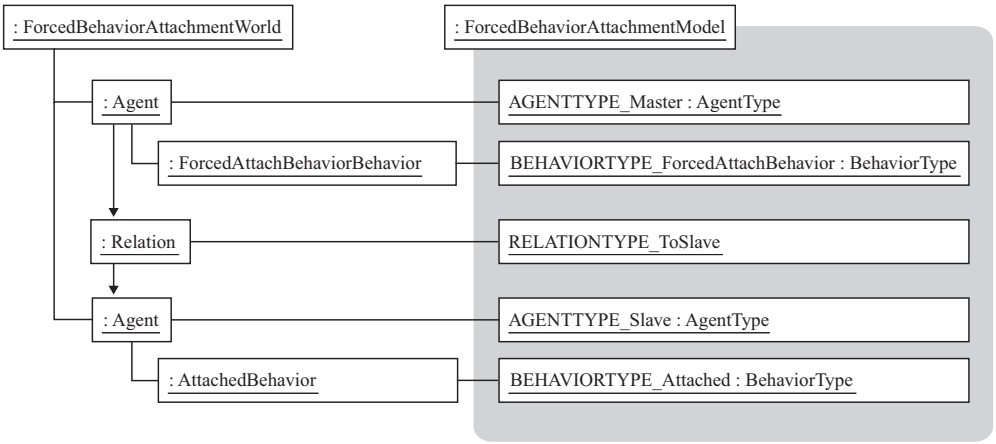


■ 基本動作

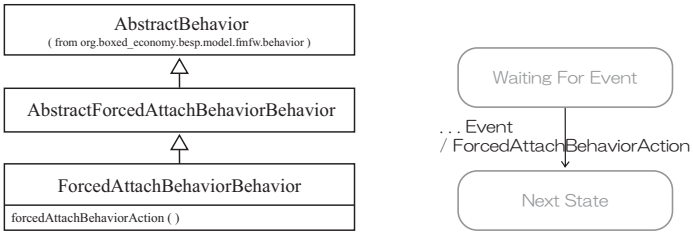
Master エージェントと Slave エージェントが登場する。Master エージェントは ForcedAttatchBehaviorBehavior をもっており、Slave エージェントに強制的に AttachedBehavior を付加する。

■ 設計

【全体像】



【AttachBehaviorBehavior】



■ サンプルコード

【ForcedBehaviorAttachmentWorld クラス】

```
…
public void initializeAgents() {
   //Master エージェントを生成する
   Agent master = createAgent(ForcedBehaviorAttachmentModel.AGENTTYPE_Master);

   //Master エージェントに、ForcedAttachBehavior を追加する
   master.addBehavior(ForcedBehaviorAttachmentModel.BEHAVIORTYPE_ForcedAttachBehavior);

   //Slave エージェントを生成する
   Agent slave = createAgent(ForcedBehaviorAttachmentModel.AGENTTYPE_Slave);

   //Master エージェントに、Slave エージェントへの Relation を追加する
   master.addRelation(ForcedBehaviorAttachmentModel.RELATIONTYPE_ToSlave, slave);
}
…
```

【ForcedAttachBehaviorBehavior クラス】

```
…
protected void forcedAttachBehaviorAction() {
   //Slave エージェントへの Relation を取得する
   Relation toSlave = this.getAgent().getRelation(
      ForcedBehaviorAttachmentModel.RELATIONTYPE_ToSlave);

   //Relation から相手を取得し、AttachedBehavior を追加する
   toSlave.getTarget().addBehavior(
      ForcedBehaviorAttachmentModel.BEHAVIORTYPE_Attached);
}
…
```

■ 関連するパターン

Behavior Creation: 行動を生成・追加する。
Requested Behavior Attachment: 外部から強制的に行動を付加するのではなく、行動を付加するように依頼する (この場合には、相手の意思を介在させる余地がある)。

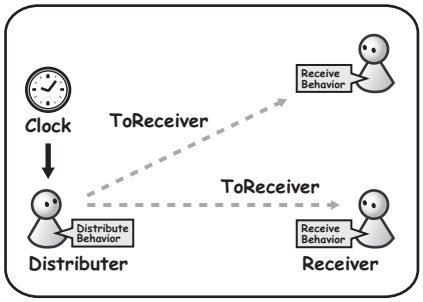
TimeEvent Distributer Agent

■ 目的

TimeEvent を一部のエージェントにだけ送りたい。

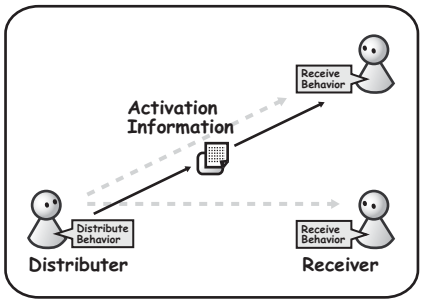
■ 動機

モデルのすべてのエージェントではなく、一部のエージェントだけを活性化させたいときがある。例えば、1 ステップに 1 人だけが動作するような場合である。配信先の決定は、ランダムに選ぶ場合もあれば、リストに従って順番に選んでいく場合もあるだろう。



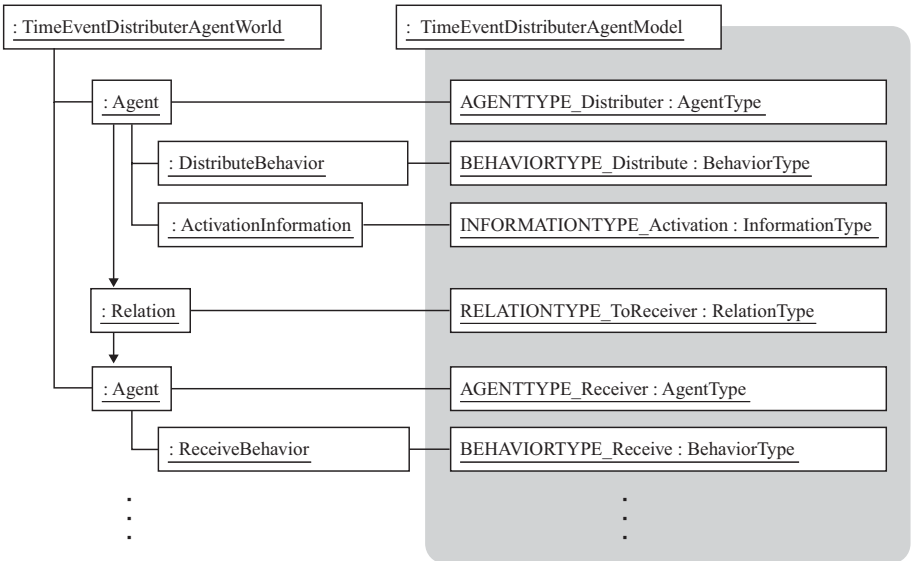
■ 基本動作

Distributer エージェントを用意する。Distributer エージェントは、TimeEvent を代表して受け取り、ActivationInformation を作成して、対象となるエージェントに配信する。活性化されるエージェントは、TimeEvent ではなく ChannelEvent で状態遷移するように記述する。

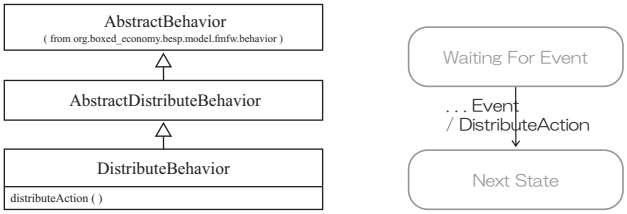


■ 設計

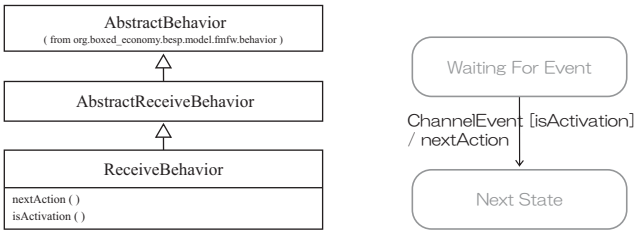
【全体像】



【DistributeBehavior】



【ReceiveBehavior】



■ サンプルコード

【TimeEventDistributerAgentWorld クラス】

```
…
public void initializeAgents() {
   //Distributer エージェントの生成
   Agent distributer = this.createAgent(
      TimeEventDistributerModel.AGENTTYPE_Distributer);
   //Distributer エージェントへの DistributeBehavior の追加
   distributer.addBehavior(TimeEventDistributerModel.BEHAVIORTYPE_Distribute);

   //Receiver エージェントの生成（複数人）
   for (int i = 0; i < 10; i++) {
      //Receiver エージェントの生成
      Agent receiver = this.createAgent(
         TimeEventDistributerModel.AGENTTYPE_Receiver);
      //Receiver エージェントへの ReceiveBehavior の追加
      receiver.addBehavior(TimeEventDistributerModel.BEHAVIORTYPE_Receive);
      //Distributer エージェントの Receiver エージェントへの Relation の追加
      distributer.addRelation(
         TimeEventDistributerModel.RELATIONTYPE_ToReceiver, receiver);
   }
}
…
```

【DistributeBehavior クラス】(ランダムに 1 人ずつに送る場合)

```
…
protected void distributeAction() {
   //Receiver エージェントへの Relation のリストを取得する
   List toReceivers = (List) this.getAgent().getRelations(
      TimeEventDistributerModel.RELATIONTYPE_ToReceiver);
   //乱数ジェネレータを用いて、リストからランダムにひとつの Relation を選び出す
   int targetIndex = this.getWorld().getRandomNumberGenerator().generate(
```

```
      toReceivers.size());
   Relation toReceiver = (Relation) toReceivers.get(targetIndex);

   //ActivationInformation を送る
   this.sendInformation(toReceiver, TimeEventDistributerModel.BEHAVIORTYPE_Receive,
      new ActivationInformation());
}
...
```

【DistributeBehavior クラス】(順番にひとりずつに送る場合)

```
...
private int targetIndex = 0;

protected void distributeAction() {
   //Receiver への Relation のリストを取得する
   List toReceivers = (List) this.getAgent().getRelations(
      TimeEventDistributerModel.RELATIONTYPE_ToReceiver);

   //リストから順番にひとつの Relation を選び出す
   if (toReceivers.size() <= targetIndex) {
      throw new ModelException("OutOfIndex : " + this);
   }
   Relation toReceiver = (Relation) toReceivers.get(targetIndex);

   //targetIndex を進める（終点の場合、始点に戻る）
   if (toReceivers.size() - 1 == targetIndex)
      targetIndex = 0;
   else
      targetIndex++;

   //ActivationInformation を送る
   this.sendInformation(toReceiver, TimeEventDistributerModel.BEHAVIORTYPE_Receive,
      new ActivationInformation());
}
...
```

【ReceiveBehavior クラス】

```
...
protected boolean isActivation(Event e) {
   //ActivationInformation を受け取った時に true を返す
   return this.getWorld().getInformationType(this.getReceivedInformation())
   == TimeEventDistributerModel.INFORMATIONTYPE_Activation;
}
...
```

■ 関連するパターン

TimeEvent Filtering: このパターンとの違いはわずかだが、決定的な差異を生み出す可能性があることに注意が必要である。例えば、全体の 20%の数のエージェントが活性化される (TimeEvent Distributer パターン) のと、各エージェントが 20%の確率で活性化する (TimeEvent Filtering パターン) というのとでは、結果が異なる場合がある。前者は、必ず全体の 20%のエージェントが活性化されるのに対し、後者は、平均すると全体の 20%のエージェントが活性化する。つまり、後者の場合は、各エージェント間の活性化確率は独立であるため、全体としてみると、活性化する人数が多いときや少ないときがある。

Model Patterns

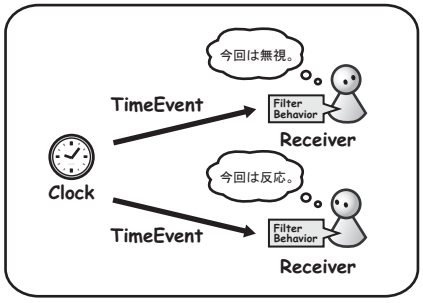
TimeEvent Filtering

■ 目的

TimeEvent を確率的に受信するようにしたい。

■ 動機

特定のエージェントを、毎回ではなく、ある確率で活性化させたいときがある。例えば、数回に 1 回だけ活性化するような行動をモデル化する場合である。

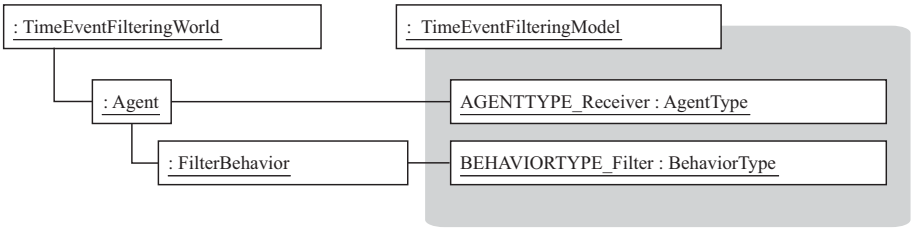


■ 基本動作

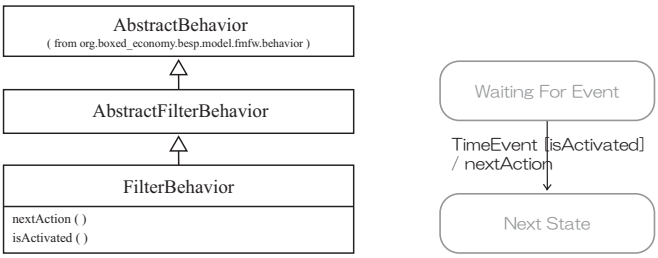
Receiver エージェントは、FilterBehavior をもっており、ある確率で TimeEvent を受け取る。

■ 設計

【全体像】



【FilterBehavior】



■ サンプルコード

【TimeEventFilteringWorld クラス】

```
…
public void initializeAgents() {
   //Receiver エージェントを生成する（複数人）
   for(int i = 0; i<10; i++){
      //Receiver エージェントの生成
      Agent receiver = createAgent(TimeEventFilteringModel.AGENTTYPE_Receiver);
      //Receiver エージェントへの FilterBehavior の追加
      receiver.addBehavior(TimeEventFilteringModel.BEHAVIORTYPE_Filter);
   }
}
…
```

【FilterBehavior クラス】

```
…
protected boolean isActivated(Event e) {
   //20 %の確率で true を返す
   return this.getWorld().getRandomNumberGenerator().generate() < 0.2;
}
…
```

■ 関連するパターン

TimeEvent Distributer Agent: このパターンとの違いはわずかだが、決定的な差異を生み出す可能性があることに注意が必要である。例えば、全体の 20%の数のエージェントが活性化される (TimeEvent Distributer パターン) のと、各エージェントが 20%の確率で活性化する (TimeEvent Filtering パターン) というのとでは、結果が異なる場合がある。前者は、必ず全体の 20%のエージェントが活性化されるのに対し、後者は、平均すると全体の 20%のエージェントが活性化する。つまり、後者の場合は、各エージェント間の活性化確率は独立であるため、全体としてみると、活性化する人数が多いときや少ないときがある。

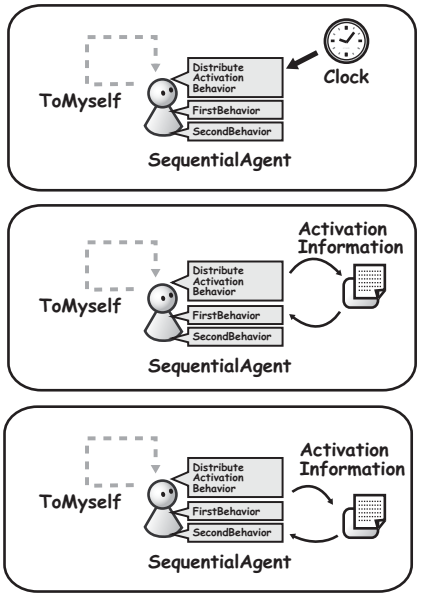
TimeEvent Distributer Behavior

■ 目的

エージェント内の Behavior に送る TimeEvent の送信順序 (プライオリティ) を制御したい。

■ 動機

エージェントが行う複数の行動を決められた順番で活性化したい場合がある。しかし、TimeEvent の送信順序 (プライオリティ) はエージェントごとに設定できるが、Behavior ごとには設定できない。そこで、Behavior の活性化を制御するための工夫が必要である。

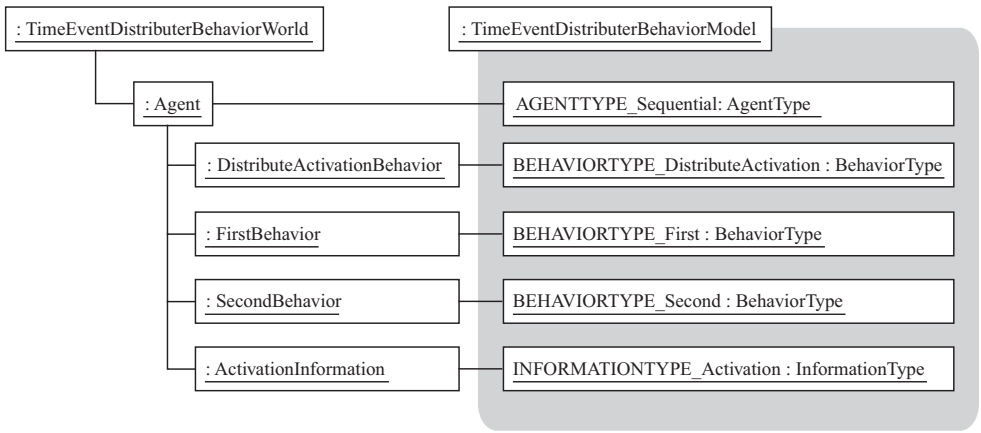


■ 基本動作

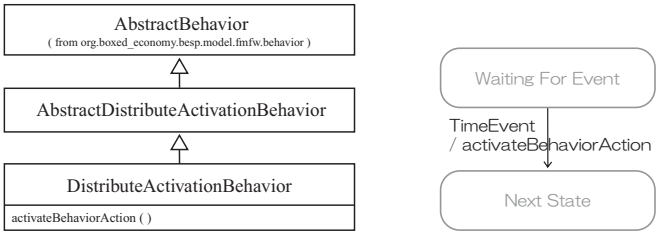
SequentialAgent は、DistributeSignalBehavior と、最初に活性化したい FirstBehavior、および 2 番目に活性化したい SecondBehavior をもっている。DistributeActivationBehavior が、TimeEvent を受け取ると、あらかじめ決められた順番で、自分自身の Behavior に ActivationInformation を配信していく。

■ 設計

【全体像】



【DistributeBehavior】



■ サンプルコード

【TimeEventDistributerBehaviorWorld クラス】

```
…
public void initializeAgents() {
   //SequentialAgent の生成
   Agent sequentialAgent = this.createAgent(
      TimeEventDistributerBehaviorModel.AGENTTYPE_Sequential);

   //SequentialAgent への DistributeSignalBehavior、FirstBehavior、
   //SecondBehavior の追加
   sequentialAgent.addBehavior(
      TimeEventDistributerBehaviorModel.BEHAVIORTYPE_DistributeActivation);
   sequentialAgent.addBehavior(TimeEventDistributerBehaviorModel.BEHAVIORTYPE_First);
   sequentialAgent.addBehavior(TimeEventDistributerBehaviorModel.BEHAVIORTYPE_Second);

   //SequentialAgent の自身への Relation の追加
   sequentialAgent.addRelation(TimeEventDistributerBehaviorModel.RELATIONTYPE_ToMyself,
      sequentialAgent);
}
…
```

【DistributeActivationBehavior クラス】

```
…
protected void activateBehaviorAction() {

   //自分の FirstBehavior に ActivationInformation を送る
   this.sendInformation(
      TimeEventDistributerBehaviorModel.RELATIONTYPE_ToMyself,
      TimeEventDistributerBehaviorModel.BEHAVIORTYPE_First,
      new ActivationInformation());

   //自分の SecondBehavior に SignalInformation を送る
   this.sendInformation(
      TimeEventDistributerBehaviorModel.RELATIONTYPE_ToMyself,
      TimeEventDistributerBehaviorModel.BEHAVIORTYPE_Second,
      new ActivationInformation());
}
…
```

■ 関連するパターン

Internal Information Sending: 自分の他の行動に情報を送る。

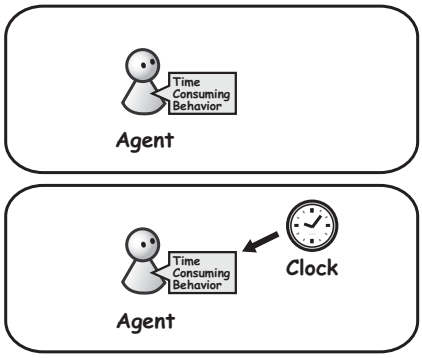
Time-Consuming Behavior

■ 目的

遂行するのに一定の時間がかかる Behavior を表現する。

■ 動機

ある行動を開始して、しばらくたってから次の動作を行わせたい場合がある。例えば、数時間ステップかかる思考や行動をモデル化する場合などである。

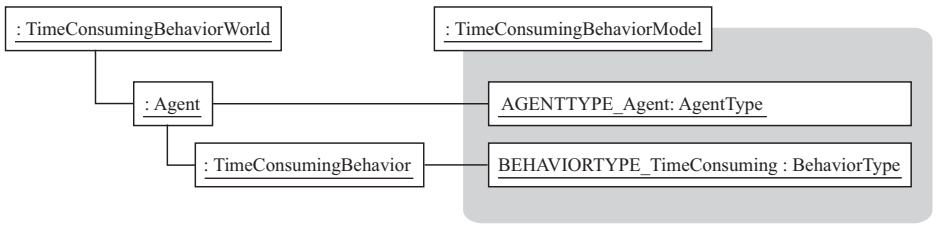


■ 基本動作

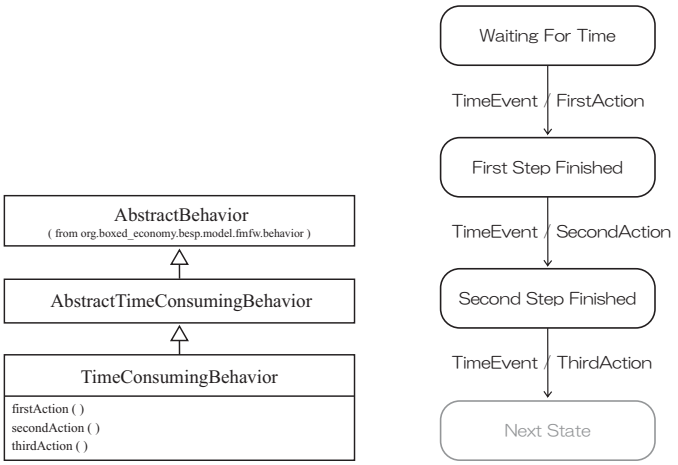
Agent エージェントは、TimeConsumingBehavior をもっている。TimeConsumingBehavior では、その一連の動作を終了するまでに、何度か TimeEvent を受ける。ここでは、Event を受けるごとに、FirstAction、SecondAction、ThirdAction を行う。

■ 設計

【全体像】



【TimeConsumingBehavior】



■ サンプルコード

【TimeConsumingBehaviorWorld クラス】

```
…
public void initializeAgents() {
   //Agent エージェントの生成
   Agent agent = this.createAgent(TimeConsumingModel.AGENTTYPE_Agent);

   //Agent エージェントに、TimeConsumingBehavior を追加する
   agent.addBehavior(TimeConsumingModel.BEHAVIORTYPE_TimeConsuming);
}
…
```

【TimeConsumingBehavior クラス】

```
…
protected void firstAction() {
   //ここで、作業の第 1 ステップを行う
}

protected void secondAction() {
   //ここで、作業の第 2 ステップを行う
}

protected void thirdAction() {
   //ここで、作業の第 3 ステップを行う
}
…
```

■ 関連するパターン

Temporary Behavior Creation: 一時的に実行する行動を追加する。