修士論文 2004 年度 (平成 16 年度)

社会シミュレーションのための モデル作成環境

～図解によるシミュレーション・モデルの作成～

慶應義塾大学大学院 政策・メディア研究科

青山　希

修士論文要旨 2004 年度 (平成 16 年度)

社会シミュレーションのためのモデル作成環境

～図解によるシミュレーション・モデルの作成～

論文要旨

本論文では，社会シミュレーションのためのモデルを，ソースコードではなく図解によって記述するモデル作成環境を提案する．現在，マルチエージェント・シミュレーションの多くは，最終的に C 言語や Java 言語などのプログラミング言語による実装を行うことで実現されている．そのため，プログラミング言語の知識がないとモデルが作成できないという問題がある．また，モデル作成者が，本質的なモデリングの作業よりも，ソースコードを記述する作業に目を奪われがちになってしまう．これらを解決するため，本論文では，シミュレーションのためのモデル作成をソースコードではなく図解によって行い，図解からソースコードを自動生成する方法を提案する．

図解によるモデルの作成を可能にするため，本論文では，その支援環境である「Component Builder」(CB) を提案する．提案するモデル作成環境は，モデルを記述するための記法として，UML と，独自に定義したアクション記述言語である「Action Block Language」(ABL) を採用する．CB はこれらを用いて図に記述されたモデルを，実行可能なプログラムのソースコードに自動で変換する機能を提供する．モデル作成者は，CB を利用することでソースコードを記述せずにモデルを作成することができる．また，ソースコードよりも抽象度の高い図解でモデルを作成することで，社会シミュレーションの対象をモデル化する作業に集中することができる．

本論文では，提案するモデル作成環境の有効性を明らかにするため，提案する環境を用いて既存モデルの再現を行った．再現するモデルとしては,「繰り返し囚人のジレンマ」モデルを取りあげた．さらに，9 名の被験者を対象として，提案するモデル作成環境を利用し，小規模なモデルを作成する試用実験を行った．これら二つの面から評価を行った結果，本論文が提案するモデル作成環境の有効性が実証された．

キーワード

1. シミュレーション, 2. オブジェクト指向, 3.UML, 4. アクション, 5.MDA

慶應義塾大学大学院　政策・メディア研究科

青山　希

Abstract of Master's Thesis Academic Year 2004

Modeling Tool for Social Simulation

-Development of Simulation Model with Diagrams-

Summary

This thesis presents modeling tools for social simulation. Current existing tools for multi-agent simulation only support modelers to make simulation models with coding. Therefore, we have to take care of the implementation with program codes in addition to modeling a target world of simulation. To solve the problem, we propose a new paradigm and development tools which translate simulation models from diagrams to program codes.

We propose "Component Builder"(CB), which consists of several tools to develop simulation models by drawing diagrams. These tools provide functions to design a model with diagrams in UML and "Action Block Language"(ABL). ABL is modeling language for describing actions in a simulation model. With the proposed tools, modelers can develop a simulation model without coding. Thus, they can concentrate on essence of a simulation model.

We experimented the performance of the proposed tools. We applied the tools to an existing model which is the model of iterated prisoner's dilemma. We also made modelers who have few programming skills to use the tools. As a result, the performance of the proposed tools is justified.

Key Word

1.Simulation, 2.Object-Oriented, 3.UML, 4.Action, 5.MDA

Keio University Graduate School of Media and Governance

Nozomu Aoyama

目 次

図 目 次

第1章　研究の目的と概要

1.1　研究の目的

本論文の目的は，社会シミュレーションのモデル作成における試行錯誤を支援し，その結果をそのままシミュレーションとして実行することができる環境を構築することである．そのために，モデル化の作業を概念モデリングの段階から支援し，ソースコードではなく図解によってモデルを記述することで，シミュレーションの対象をモデル化する作業に集中できる環境を提案する．

コンピュータ・シミュレーションの多くは，モデル作成を支援するツールを使う使わないにかかわらず，最終的にプログラミング言語による実装を行うことで実現されている．そのため，そのプログラミング言語の文法を知らなければ，コンピュータ・シミュレーションを研究や学習に利用することができない．

モデル作成の過程でソースコードを記述することは，その過程における試行錯誤を妨げる原因ともなる．既存のシミュレーション環境では，プログラミング言語による実装の作業はコストが高く，ミスも起こりやすいため，どうしてもそこでの作業に目を奪われがちになる．このようなモデル化の作業において本質的でないことがらに目を奪われることで，概念モデリングの段階に立ち戻って，試行錯誤しながらモデルを作成することが難しくなってしまう．

このような問題に対し，本論文ではモデルの記述方法として，UML(Unified Modeling Language) と社会シミュレーションに特化したアクション記述言語「Action Block Language」を用いた，図解による記述を採用する．さらに，図解によるモデル作成を支援する環境として「Component Builder」を提案する．本章では，その前提となる概念を整理した後に，本論文における提案の概要を提示することにしたい．

1.2　コンピュータ・シミュレーションによる社会の理解

一口にシミュレーションといっても，シミュレーションには多くの種類があり，また利用する目的も様々である．本論文では，その中で社会現象の理解を目的としたコンピュータ・シミュレーションの利用に焦点を当てることにする．

コンピュータ・シミュレーションは「社会の特徴についての『理解』を深める」(Gilbert and Troitzsch, 1999) ための道具として利用することができる．シミュレーションとは,「対象とするシステムの要素やその関係性を抽出したモデルを操作することによって，システ

ムの動作などを調べるために行われるもの」(井庭および福原, 1998) である．内部の動作がよくわかっていないシステムがあったときに，その構成要素をモデル化し，シミュレーションを行うことで，対象となるシステムの理解につなげることができる．

さらに，コンピュータ・シミュレーションは，その結果が社会を理解するのに役立つだけでなく，シミュレーションのモデルを作成する過程そのものも，社会の理解を深める手助けとなる．モデルをコンピュータ上で動作するプログラムのソースコードに変換するためには，完全に整合性の取れた仮説を立てて，それをモデルとして表現しなければならない．そのためにモデルを推敲する作業の中で，モデル作成者はシミュレーションの対象に対する新たな視点を得て，社会を理解する手助けとすることができる．

また，本論文では，コンピュータ・シミュレーションの中でも特に，マルチエージェント・シミュレーションに焦点をあてる．まず，エージェントとは何かということについて Axelrod は「エージェントは，他のエージェントや外部環境と相互作用する能力を備えている．また，自分の周りで起きたことに反応し，ある程度の目的を持って行動する.」(Axelrod and Cohen, 1999) と述べている．マルチエージェント・シミュレーションは，システムの構成要素であるエージェントの相互作用を通して，システム全体の振る舞いを理解するための手法である．山影らは「マルチエージェントシミュレーションはコンピュータの中に人工社会を作る方法でもあり，そして同時にその社会を分析する方法でもある.」(山影および服部, 2002) と述べている

1.3 シミュレーションのためのモデル

モデルという言葉について，Wilson は「“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の明示的な解釈である．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，実態，プロセス，属性，およびそれらの関係についての記述である．モデルは規範的，記述的のどちらでもありうるが，何よりも役立つものでなければならない」(Wilson, 1990) という定義を行っている．本論文でも，このような広い意味で「モデル」という言葉を使うことにしたい．

本論文で扱うモデルは全て，先の定義に照らし合わせると，シミュレーションによるシステムの理解に「役立つ」モデルである．まず，概念モデルは,「関心領域の明確化」,「概念の図式化」,「構造と論理の定義」,「設計の前提作業」のためのモデルである (Wilson, 1990)．設計モデルは，概念モデルをどのようにしてシミュレーションを行うプログラムとして実現するか，という「How」を定義するためのモデルである．本論文で言う設計モデルとは，UML で記述された構造や振る舞いのモデルと，シミュレーションの中で行われる処理の論理的な構造を記述したモデル[1]のことである．また，概念モデル，設計モデルとして記述するモデルだけでなく，シミュレーションをして結果を観察するためのソー

[1]本論文で提案するアクション記述言語は，シミュレーションの中で行われる具体的な処理の論理的な設計を記述し，実行可能なソースコードを生成するためのものである．これは論理構造の設計と，実装の定義を同時に行う言語によるプログラム記述であるとも言えるが，本論文ではまとめて設計モデルと呼ぶことにした．

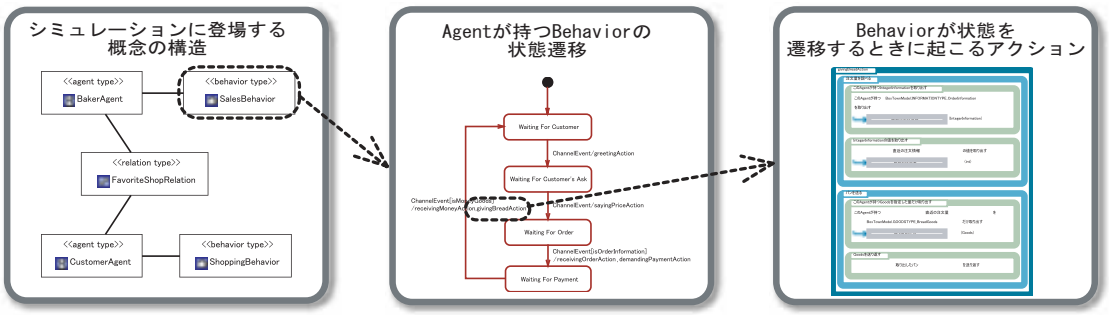


図 1.1: シミュレーションのモデル作成に必要な 3 つのモデル (第 3 章より)

スコードも実装のモデルとして捉える．モデル作成者は本論文で提案するツールにより，分析，設計の様々な場面で，様々な記法を使って必要なモデルを作成していく．本論文で，特に何のためのモデルであるかを規定せずに「モデル」という言葉を使用しているときは，これらをまとめた，シミュレーションのためのモデル全体を指していることとする．

1.4 図解によるモデル作成

本論文では，ソースコードではなく図解による，社会シミュレーションのためのモデル作成を提案する (第 3 章)．

図解によるモデルの記述から，実行可能なプログラムのソースコードを，ツールによって生成するために，モデル作成者は 3 種類の設計モデルを記述する必要がある．3 種類のモデルとは，シミュレーションに登場する概念の構造，エージェント (Agent) が持つ振る舞い (Behavior) の状態遷移，Behavior が状態遷移するときに起こるアクション，の 3 つである (図 1.1)．モデル作成者はこれらを記述した図から，ソースコードへの変換を，ツールによって自動で行うことができる．

シミュレーションの中で実行される具体的な処理を記述するために，モデル作成者は，本論文で提案する社会シミュレーションに特化したアクション記述言語「Action Block Language」(ABL) を利用することができる．これによりモデル作成者は，シミュレーションの中で行いたい処理の論理的な構造を設計し，その設計を実現するための手段を ABL に定義されている語彙の中から選択することで，処理を記述することができる．

図解によるモデルの記述から，実行可能なプログラムのソースコードを自動で生成することで，モデル作成者はソースコードを記述せずにモデルを作成することができる．そのため，特定のプログラミング言語の知識がないモデル作成者でも，コンピュータ・シミュレーションを研究や学習に利用することが可能である．また，ソースコードよりも抽象度の高い図解でモデルを作成することで，シミュレーションの対象をモデル化する作業に集中することができ，モデル作成における試行錯誤がし易くなる．

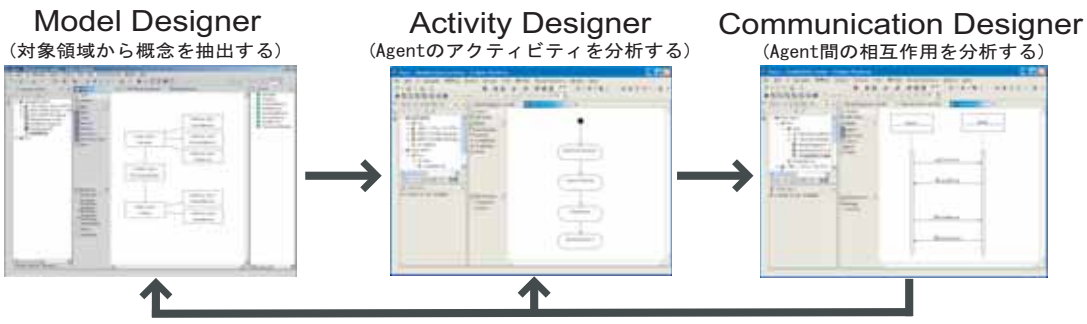


図 1.2: 概念モデリングフェーズをサポートするツール (第 4 章より)

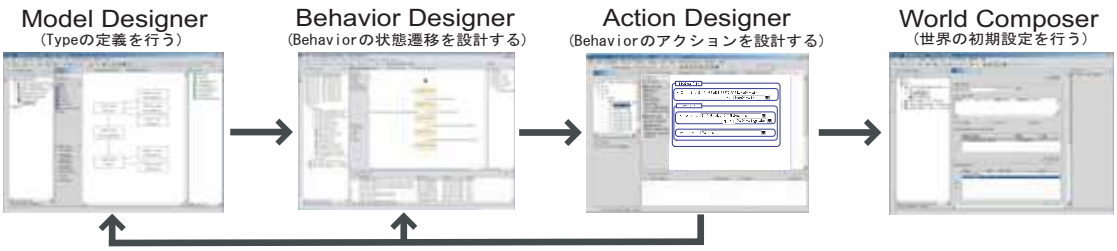


図 1.3: シミュレーションデザインフェーズをサポートするツール (第 4 章より)

1.5 モデル作成環境「Component Builder」(CB)

本論文では，図解によるモデル作成を支援するための環境である「Component Builder」(CB) を提案する (第 4 章)．CB は 6 種類[2]のツールで構成されるモデル作成環境である．CB は概念モデリング，およびシミュレーションデザインのフェーズで様々なモデルの作成を支援する機能を提供している．

まず CB には，概念モデリングフェーズをサポートするための 3 種類のツールが含まれている．モデル作成者は，これらのツールを使用して，対象領域からの概念の抽出，Agent のアクティビティの分析，Agent 間の相互作用の分析，を反復的に行いながら概念モデルを作成していく (図 1.2)．

次に CB には，シミュレーションデザインフェーズをサポートするための 4 種類のツールが含まれている．モデル作成者はこれらのツールを使用して，シミュレーションに登場する Type の設計，Behavior の状態遷移の設計，Behavior のアクションの記述，を反復的に行いながら設計モデルを作成していく (図 1.3)．

図解によるモデルの作成と，作成した設計モデルからソースコードへの変換を行うために，CB は様々な機能を提供している．これらを効率よく実現するため，CB は作図やソー

[2]CB には，概念モデリングフェーズを支援する 3 種類のツールと，シミュレーションデザインフェーズを支援する 4 種類のツールが含まれているが，Model Designer は概念モデリング，シミュレーションデザインの両方のフェーズで使用するため，CB は合計で 6 種類のツールで構成されている．

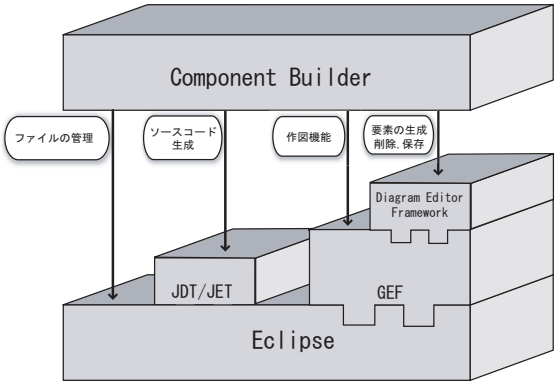


図 1.4: Component Builder のアーキテクチャ(第 4 章より)

スコード生成のためのフレームワークを利用した，図 1.4 のようなアーキテクチャで実装されている．

第2章　研究の背景

2.1　既存のシミュレーション環境とその課題

これまでにも，マルチエージェント・シミュレーションを支援するための言語やツールはいくつか提案されてきている．North (2002) では既存のシミュレーション環境を図 2.1 のように比較している．

「Swarm」(Swarm Simulation System) は，それらの中で最も有名でよく利用されているツールである．Swarm はシミュレーションのためのクラスライブラリを提供している (Langton et al., 1998)．また，Swarm と同様のコンセプトの「Repast」(REcursive Porus Agent Simulation Toolkit) もシミュレーションのためのクラスライブラリを提供している (Collier, 2003).「Ascape」は前の二つとは異なり，シミュレーションのためのフレームワークを提供するものである．Ascape を利用すると，Swarm や Repast に比べてソースコードの記述量が少なくてすむと言われている (Parker, 2000; Parker, 2001)．

このほかに，汎用のプログラミング言語を用いるのではなく，シミュレーションのための簡易言語を定義している支援ツールもある．シミュレーションの原理を教育するのに有効であるといわれている「StarLogo」では，LOGO[1]を拡張した簡易言語を定義している．Swarm と StarLogo の中間レベルを実現しようとしている「KK-MAS」(KK Multi-Agent Simulator) では，Visual Basic 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; 山影および服部, 2002)．

これらの支援ツールはどれもシミュレーションの実装を支援するためのもので，シミュレーションの設計を支援するものではない．シミュレーション研究のプロセスは，シミュレーションのモデルを設計する段階と，モデルを作成する段階に分けることができる (Gilbert and Troitzsch, 1999)．既存のツールが提供するライブラリやフレームワークを利用することで，モデル作成の段階におけるソースコードの記述量は大幅に削減される．しかし，これらのツールはモデルを作成する段階における，現実世界のモデル化や，モデルをソースコードに変換する作業を支援するものではない．シミュレーションによる複雑な現象の理解をより効果的に支援するためには，モデル作成を設計から実装まで支援するツールが必要である．

また，既存の支援ツールを利用した場合，最終的には特定のプログラミング言語による実装を行わなければならない．そのため，プログラミングの知識がない人や，特定の言語の知識しかない人にはシミュレーションを作成したり，人が作成したシミュレーションを

[1]LOGO は，子供を対象とした教育用のプログラミング言語である．

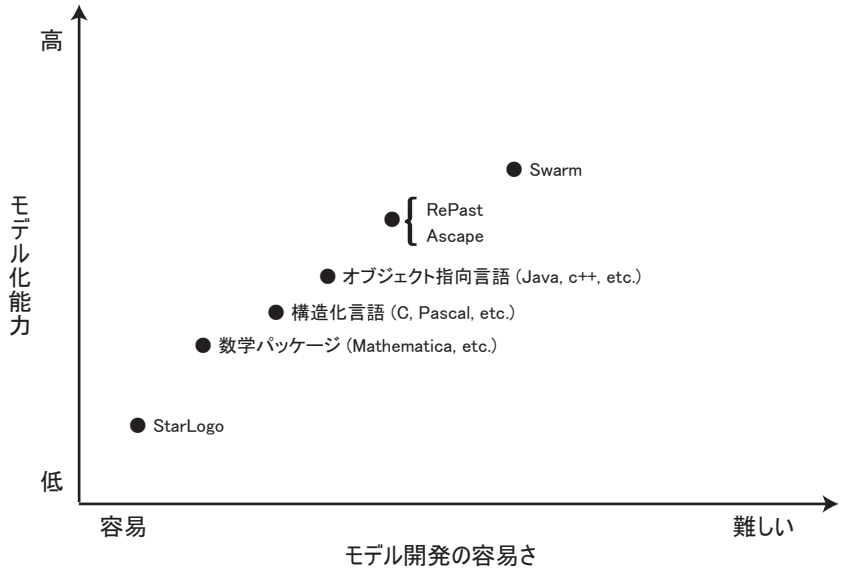


図 2.1: 既存シミュレーション環境の比較 (North, 2002 より)

理解することができない．プログラミングの知識がない人にとっては，どんなに便利なライブラリがあったとしても，文法の知識がないために，それらを使いこなすことができないのである．同じように，たとえプログラミングの知識があったとしても，支援ツールが依存しているプログラミング言語の経験がなければツールを利用することはできない．

モデル作成において，プログラムを記述することは，理論を洗練させモデルに対する理解を深めるために有効である (Gilbert and Troitzsch, 1999)．しかしその際，特定のプログラミング言語の文法を理解することは，必ずしも必要なことではなく，むしろ文法などのプログラミング言語固有の事柄を気にせずに，シミュレーションを作成できることが望ましい．このような問題を解決するためには，プログラミング言語の文法を気にせずに，プログラムの本質である，構造や振る舞い，処理の論理的な構造を記述するだけで，モデルを作成できる環境が必要である．

本論文ではソフトウェア工学のアプローチと，それをシミュレーションの分野に応用した Boxed Eonomy Project のアプローチを踏まえて，これらの問題の解決を試みる．

2.2 ソフトウェア工学のアプローチ

ソフトウェア開発の発展は抽象化レベル向上の歴史である (Mellor and Balcer, 2002)．ソフトウェア工学のアプローチは，プログラムを記述する言語の抽象度を引き上げ，下位のレベルの手段を意識せずにプログラムを記述できるようになることを目指してきた．この考え方はシミュレーションのためのプログラム作成にも応用することができる．

もうひとつここでとりあげたいのは，言語の抽象度を上げる試みと同時に発展してきた，

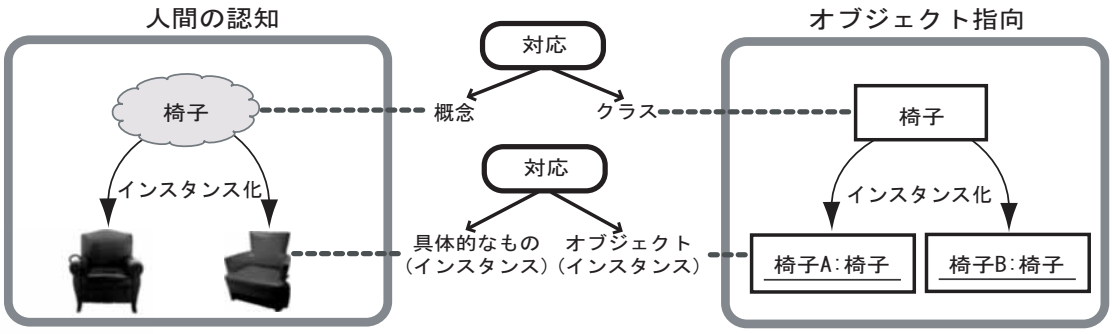


図 2.2: 人間の認知の仕組みとオブジェクト指向

ソフトウェアを開発するためのプロセスである．シミュレーションのためのモデルを作成するための開発プロセス定義することで，モデル作成の生産性を向上させることができる．

2.2.1 オブジェクト指向

オブジェクト指向とは，データ構造と振る舞いが一体となったオブジェクトの集まりとしてソフトウェアを組織化することである (Rumbaugh, 1992)．データ構造と振る舞いを一体とすることで，物理的，あるいは概念的なモノのひとつひとつをモデル化の単位として利用することができる．これはつまり，オブジェクト指向によるモデル化の過程が，人間の認知の仕組みに近く，より直感的なモデル化ができることを意味している (図 2.2)．

シミュレーションにオブジェクト指向の考え方を導入することで，現実世界のモデル化が直感的にできるだけでなく，作成したモデルを作成者以外の人と共有することが容易になる．なぜなら，オブジェクト指向の考え方は人間の認知の仕組みに近いため，人が作ったモデルを理解するのも容易になるためである．このことは，オブジェクト指向という考え方が考案された背景とも深い関係がある．オブジェクト指向の起源は，1960 年代にノルウェーで開発された Simula という言語にさかのぼる (Dahl and Nygaard, 1966)．Simula は現実世界の様々な現象をシミュレートするための言語であった．

2.2.2 UML:統一モデリング言語

UML(Unified Modeling Language:統一モデリング言語) は，オブジェクト指向のモデルを記述するための標準化された記法である．ソフトウェア主体のシステムの成果物をビジュアル化，仕様化，構築，文書化することを目的としている (Object Management Group, 2000; Object Management Group, 2004)．また，UML は表現力の豊かな言語で，システムの開発から配置に必要なすべてのビューに対応している (Booch et al., 1999)．

UML は特定のプログラミング言語や開発プロセスには依存していないため，いろいろな開

発プロセスにおいて成果物として利用することができる．これはUMLが一般的なプログラミングをすべてサポートすることを目指しているためである (Object Management Group, 2000)．

UMLには，モデルを図で記述するための記法以外にも，オブジェクトが持つ制約として記述するための言語「OCL」(Object Constraint Language)(Object Management Group, 2003)や，状態遷移と共に起きるアクションをモデル化するための仕様であるアクションセマンティクス (Object Management Group, 2004) が定義されている[2]．最新のUML2.0[3]では，これらを使って実行可能なプログラムをUMLで記述することが目指されている．

2.2.3 開発プロセスと支援ツール

ソフトウェア工学の分野では，プログラミング言語による実装を行う前に，問題領域の分析とソフトウェアの設計をどのようにするかという課題に関して，これまでに多くの議論がなされてきた．

ソフトウェア開発の現場で古くから採用されてきたソフトウェアプロセスとして，ウォータフォール型の開発プロセスがあげられる．このプロセスは，ソフトウェア開発のプロセスを要求把握，分析，設計，実装，テストという段階に分け，それぞれの作業を水が流れるように順番に行っていくというものである．

これに対しJacobsonらは，要求把握，分析，設計，実装，テストのワークフローを，反復的に行う開発プロセスを「RUP」(Rational Unified Process) としてまとめている (Jacobson et al., 2000)．UPと同じく反復的な開発を推奨している手法に，Kent Beckらの「XP」(eXtreme Programing) があげられる (Beck, 2000)．XPは厳密な開発プロセスを定義するものではないが，やはり反復的な開発を推奨している．

開発プロセスには，プロセスの各フェーズでそれぞれ作成すべき成果物が定義されている．ここで言う成果物とは，各フェーズで作成される図や，それに関連する文章，スケッチのことである．開発プロセスの各フェーズに必要な成果物が定義されていることで，開発プロセスを利用する人にとっては，次に何をすればよいのかという明確な指針を得ることができる．またそれらの成果物は，開発のドキュメントとしても有用である．成果物の作成者以外の開発メンバーや外部の人間は，成果物を見ることで作成者の意図を理解することができる．

開発プロセスにのっとってソフトウェアを開発するために，プロセスをサポートするツールは必要不可欠である (Jacobson and Jacobson, 1995; Jacobson and Jacobson, 1995)(図2.3)．開発にツールを利用することで，プロセスの一部を自動化することが可能になる．例えば，分析プロセスの成果物から開発プロセスの成果物の雛形を生成するといったよう

[2]アクションセマンティクスに関しては，現在のところ抽象的なアクション言語の仕様が策定されているだけで，具象文法は定義されていない．

[3]UML2.0は現在正式リリースを控え，暫定の仕様がWebで公開されている段階である (Object Management Group, 2004; テクノロジックおよび長瀬, 2004)．

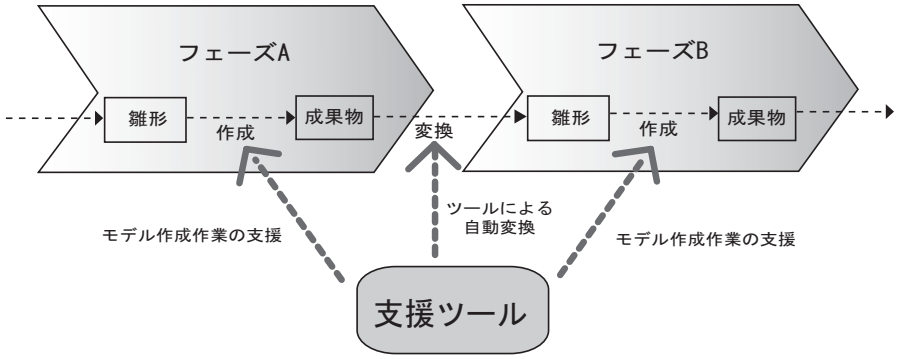


図 2.3: 開発プロセスとツールの関係

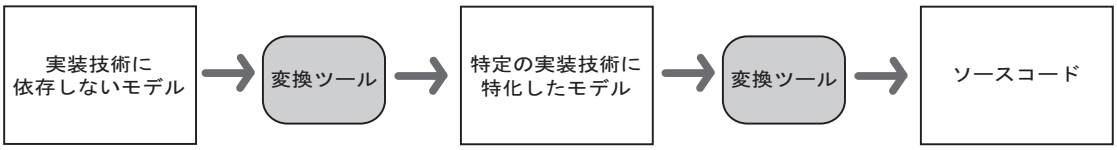


図 2.4: MDA の開発プロセス

なことが可能である．このような利点を享受するためには，開発プロセスを定義する段階でツールによる自動化を念頭におかねばならない (Jacobson et al., 2000)．

2.2.4 MDA:モデル駆動アーキテクチャ

MDA(Model Driven Architecture:モデル駆動アーキテクチャ) とは，モデリング言語 (UML) を単なる設計のための言語としてではなく，プログラミング言語として利用するソフトウェア開発の方法論である (Frankel, 2003; Kleppe et al., 2003; Mellor and Balcer, 2002)．MDA によるソフトウェア開発では，ソフトウェアアーキテクチャ，プログラミング言語などの「実装技術に依存しないモデル」(Platform Independent Model:PIM) から，特定の実装技術に依存するモデル (Platform Specific Model:PSM) を生成する．さらに，特定の実装技術に依存するモデルから，実行可能なプログラムのソースコードを生成する．モデルからモデルの生成や，モデルからソースコードの生成は，ツールによって自動で行われる (図 2.4)．以下に MDA によるソフトウェア開発の利点を述べる．

まず第一の利点は，開発者がソースコードではなく，より抽象度の高い言語で記述されたモデルに焦点をおいて開発をすることができることである．MDA によるソフトウェア開発では，実装に依存しない概念モデルから，特定の実装に依存する設計モデルやソースコードへの変換はツールによって自動的に行われる．そのため，開発者は特定の技術による実装の手段を気にせず，システムが解決すべき問題に集中することができる．

第二の利点は，MDA による開発の成果はいかなる実装技術にも依存しないということ

である．実装に依存しないモデルによって開発を行うことで，ソフトウェア開発者はシステムが採用する実装技術に関する詳しい知識を持つ必要がなくなる．また，実装に依存するモデルへの変換は自動で行われるため，変換のルールだけ変更すれば異なる実装技術への移行が可能である．

第三の利点は，最終的な成果物であるソフトウェアと，それに関するドキュメントの間の同期をはかるのが容易になることである．これまでのソフトウェア開発において，設計のためのモデル図は，実装の指針となる単なる紙としての役割しか果たさなかった．そのため，実装とテストの段階で問題が発見されると，ソースコードは修正されるがそれに対応する設計のためのモデル図は修正されず，納品の際にまとめて修正を行うことが多かった．MDA によるソフトウェア開発では，モデルからソースコードが生成されるため，常にモデルとソースコードの一貫性は保たれる．

第四の利点は，ソフトウェア開発における作業量の減少である．MDA による開発では，実装に依存しないモデルから，実装技術に依存するモデルを作成する設計の作業は，ツールによって自動で行われる．また，実装技術に依存するモデルから，ソースコードを作成する作業も同様である．さらにツールで自動化することによって，実際にコーディング量が減るだけではなく，実装の際に混入するバグがなくなるためデバッグの作業も減少する．

2.3 Boxed Economy Project のアプローチ

Boxed Economy Project[4]は，1999 年に慶應義塾大学湘南藤沢キャンパスの学生有志によって結成された研究プロジェクトである．Boxed Economy Project 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枠組みを提案してきた．筆者も 2002 年よりこのプロジェクトに参加し，ツールや方法論に関する議論，開発を行ってきた．

本節では本論文の背景となる，Boxed Economy Project のこれまでのアプローチについて述べる．なお，これらのアプローチの詳細については，井庭 (2003) および Iba (2004) を参照してほしい．

2.3.1 モデル・フレームワーク

Boxed Economy Project は，複雑な社会・経済モデルを記述・操作するための基本的な枠組みとして，オブジェクト指向モデルの導入を提案している．その上で，基本となるモデルの要素をモデル・フレームワーク「Boxed Economy Foundation Model」(BEFM) として定義し，シミュレーションのモデル作成のための指針を提供している (Iba, 2004; 井庭ほか, 2003; 井庭, 2003)．

モデル・フレームワークとは，モデルを記述する際に頻繁に登場する要素とその構造を

[4]Boxed Economy Project の活動の内容は，Web(http://www.boxed-economy.org/) に公開されている．

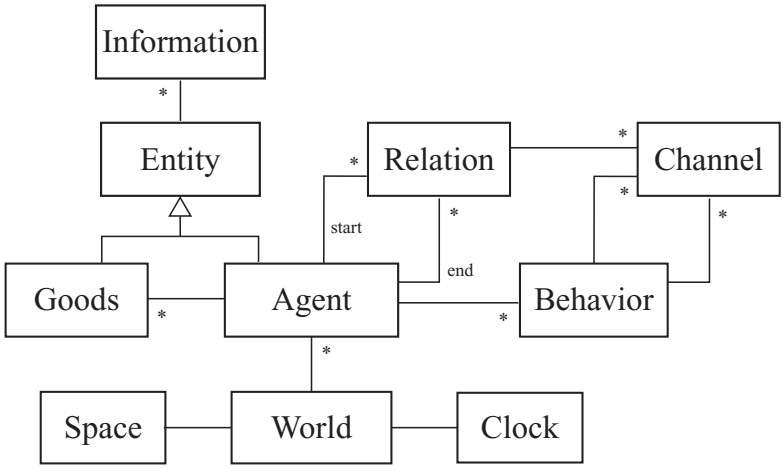


図 2.5: BEFM 概念モデル・フレームワークのクラス図

定義したものである．BEFM には社会・経済のモデルフレームワークとして,「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」の 2 種類が定義されている．

「概念モデル・フレームワーク」とは，モデル化しようとしている対象が，どのようなものでどのような構造を持っているかを洗い出し，概念モデルとして記述するための枠組みである．BEFM 概念モデル・フレームワークには図 2.5 のような概念の構造が定義されている．BEFM の主な特徴は，エージェント (Agent) 間の相互作用を，財 (Goods) とそれに付随する情報 (Information) のやりとりとしてモデル化している点と，エージェントの行動 (Behavior) を外部化して,「Agent が Behavior を持っている」という構造になっている点である．そうすることで「振る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る舞いのルール自体が変化する」というモデル化も可能になっている．

「シミュレーションモデル・フレームワーク」とは，概念モデルをシミュレーションとして実装するための設計モデルを記述するための枠組みである．概念モデル・フレームワークで記述したモデルの移行をシームレスに行うために，シミュレーションモデル・フレームワークは概念モデル・フレームワークと一貫性を持つように設計されている．シミュレーションモデル・フレームワークには，モデルに存在する要素を分類するための識別子として「Type」というクラスが用意されている (図 2.6)．Type の導入により，モデル要素の柔軟な検索・識別・取得が可能となる．

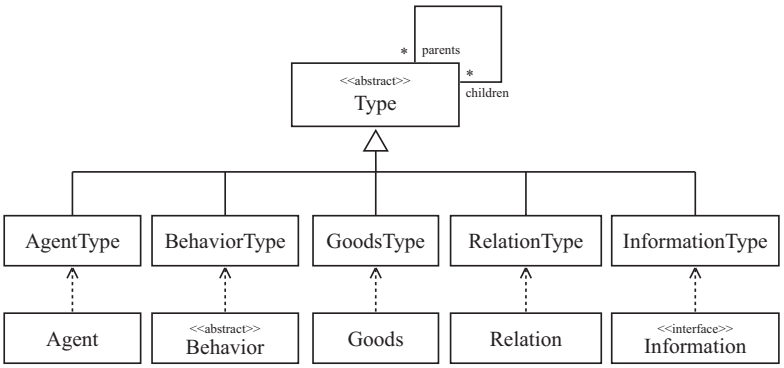


図 2.6: Type とその関連クラス

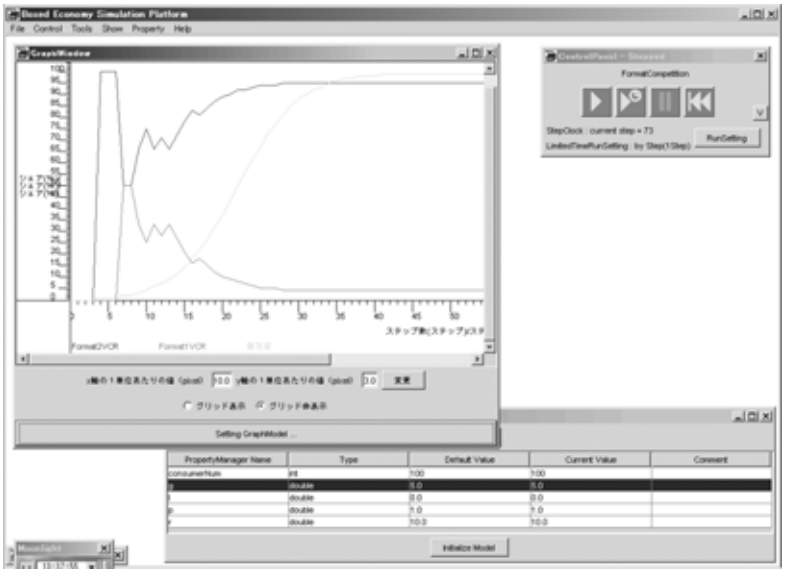


図 2.7: BESP の実行画面

2.3.2 シミュレーション・プラットフォーム

Boxed Eoconomy Project は，社会シミュレーションを実行，分析するためのソフトウェア「Boxed Economy Simulation Platform」(BESP) (Iba, 2004; 井庭, 2003) を提案している[5](図 2.7)．BESP は BEFM に基づいて記述されたプログラムをプラットフォーム上で実行し，グラフやファイルへの出力などさまざまな方法でシミュレーションの結果を分析する機能を提供している．

BESP は，さまざまなシミュレーションに対応するため，コンポーネントを組み合わせることでシミュレーションを利用することができる，拡張可能なソフトウェアとして設計

[5]BESP のチュートリアルとして，Boxed Economy Project (2003); Boxed Economy Project (2004) が公開されている．

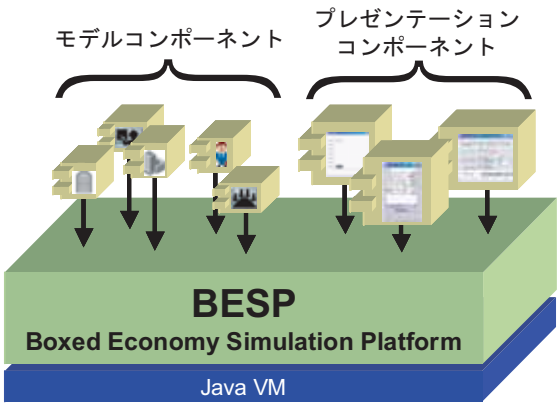


図 2.8: BESP のアーキテクチャ

されている (図 2.8)．BESP にインストールすることができるコンポーネントは,「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 2 種類である．モデルコンポーネントは，BEFM に基づいて実装されたシミュレーションのためのソフトウェアコンポーネントである．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経過や結果を画面上に表示したり，GUI からシミュレーションの設定をするためのコンポーネントである．

コンポーネントの作成者は，利用したいシミュレーションのモデルと，そのプレゼンテーションをプログラムとして作成することで，BESP 上でシミュレーションを行うことができる．BESP は，シミュレーションを行う上で頻繁に行われる時間 (ステップ) の管理や，Agent 間のやりとりをフレームワークの機能として提供している．このため，コンポーネント作成者は，一からシミュレーションのプログラムを作成するよりも容易に実装を行うことができるだけでなく，より信頼性の高いシミュレーションを行うことができる．

2.3.3 モデル作成のプロセス

Boxed Eoconomy Project は，概念の分析からシミュレーションの実行・検証までを反復的に行う，モデル作成のプロセスも提案している (Iba, 2004; Iba et al., 2004; Aoyama et al., 2004)．提案されているプロセスは,「概念モデリングフェーズ」,「シミュレーションデザインフェーズ」,「実行・検証フェーズ」の 3 つのフェーズから構成されている (図 2.9)．以下では，それぞれのフェーズにおける作業の概要について述べる．

概念モデリングフェーズ

概念モデリングフェーズはシミュレーションの対象となる領域を分析し，概念モデルとして記述するフェーズである．モデル作成者は BEFM の概念モデル・フレームワークに

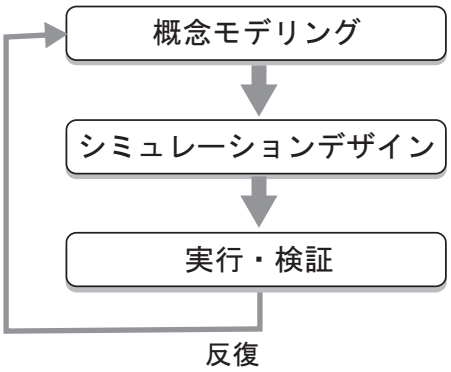


図 2.9: モデル作成のプロセス

基づいて，エージェントや情報，財などのモデルを構成する要素とその構造を記述する．

概念モデリングフェーズの成果物は，以下の 3 種類の図であらわされた概念モデルである．

- 概念モデルクラス図
- アクティビティ図
- コミュニケーションシーケンス図

概念モデルクラス図は，UML のクラス図を使って概念モデルの静的な構造を記述するものである．アクティビティ図は UML に定義されている記法であり，ここではシミュレーションの流れを記述するために利用する．コミュニケーションシーケンス図は，シミュレーションにおけるエージェント間の相互作用を，UML のシーケンス図を使って記述するものである．

シミュレーションデザインフェーズ

シミュレーションデザインフェーズは，概念モデリングフェーズで抽出した概念モデルから，BESP 上で実行可能なソースコードを作成するために，シミュレーションの設計，実装を行うフェーズである．モデル作成者は BEFM のシミュレーションモデル・フレームワークにしたがって設計モデルを作成する．

シミュレーションデザインフェーズの成果物は以下のものである．

- 設計モデルクラス図
- ステートチャート図
- シミュレーションの初期設定

- シミュレーションのソースコード

設計モデルクラス図は，UML のクラス図を使ってシミュレーションの静的な構造を記述するものである．具体的には，概念モデルで抽出した要素を，拡張可能性，再利用性の観点から再検討し，必要ならば分割，統合などの設計判断を行って Type として定義する．また，概念モデルで抽出した Type に加えて，実装が必要な Information の定義を行う．ステートチャート図は UML に定義されている記法であり，Behavior の状態遷移を記述するためのものである．シミュレーションの初期設定は，具体的には作成する World が持つ初期値と，初期状態で配置されている Agent の数，設定されている Relation の構造の記述である．シミュレーションのプログラムは，BEFM の設計モデル・フレームワークにおいて，サブクラスを作成して実装することが規定されている World, Behavior, Information の具体的なソースコードである．

実行・検証フェーズ

実行・検証フェーズは，実際にシミュレーションを行って，その正当性を検証するフェーズである．ここでいう正当性とは，シミュレーションにバグがなく，意図したとおりに動いているかということであり，モデルが問題解決に対して妥当であるかではない．実行・検証フェーズの成果物は，正しく動作することが検証されたシミュレーションのソースコードである．

2.4 本論文の位置づけ

本論文で提案するモデル作成環境は，Boxed Economy Project が提案するプロセスのうち，概念モデリングフェーズとシミュレーションデザインフェーズを支援する．Boxed Economy Project は，第 2.1 節であげたような既存のシミュレーション環境の問題点を解決するために，モデル・フレームワーク，モデル・フレームワークに沿ってモデルを作成するためのプロセス，作成されたモデルを実行するためのシミュレーション・プラットフォームを定義，開発してきた．しかし，ただプロセスが定義されているだけでは，プロセスをモデル作成に適用することは難しい．なぜなら，プロセスを利用するためには，ツールによるサポートが必要不可欠なためである．

このモデル作成環境は，プロセスの定義とほぼ同時に開発されてきた．プロセスを評価するためにはツールが必要不可欠であり，ツールを開発するためにはプロセスが定義されていなければならない．これらは鶏と卵の関係であり，どちらが先というものではない．よいプロセスとツールを開発するためには，それらを平行して行わなければならない (Booch et al., 1999)．

本論文における成果として，提案するモデル作成環境により，モデル作成に必要なソースコードの実装を大幅に自動化できたことがあげられる．提案するモデル作成環境は，モ

デルを記述したUMLなどの図からソースコードを生成する．このツールを用いると，シミュレーションの構造や振る舞い，処理を図に描くだけで，ソースコードをほとんど書かずに社会シミュレーションのモデルコンポーネントを作成することができる．また，特定の言語の知識がなくても，人が作ったモデルを理解することができる．

また，ツールによる自動化を行うために，モデル作成プロセスにも改良が加えられた．図解によってモデルを記述するために，本論文ではシミュレーションのモデル作成に特化したアクション記述言語を提案する．UML以外に，このアクション記述言語をプロセスに組み入れることで，一般的なプログラミング言語よりも，抽象度の高い図解でモデルを作成することができる．そのため，シミュレーションの問題領域だけに焦点をあててモデル作成を行うことができるようになった．

第3章　図解によるモデル作成

本章では，本論文で提案する図解によるモデル記述の詳細について述べる．まず，図解でモデルを記述する意義について述べ，次に，図解で記述したモデルをソースコードに変換するためには，図解に何を記述する必要があるのかを述べる．最後に，プログラムの処理を図解で記述するための，社会シミュレーションに特化したアクション記述言語について述べる．

3.1　図解によるモデル作成の意義

3.1.1　プログラミング言語の文法に依存しないモデル作成

これまでシミュレーションのモデル作成には，支援ツールを使う使わないに限らず，最終的にはC言語やJava言語などのプログラミング言語による実装が必要であった．その結果，作成されたモデルが特定のプログラミング言語に依存し，そのプログラミング言語の知識がない人には，モデルを作成したり，他人が作ったモデルを理解したりすることができなかった．

一方，UMLやアクション記述言語を利用して図解でモデルを記述すると，プログラミング言語の文法を気にせずに，モデルを作成することができる．そのため，特定のプログラミング言語の知識がない人でも，モデルを作成したり，他人が作ったモデルを理解することが容易になる．また，既存のシミュレーション環境が依存する言語に精通した人がモデルを作成する場合でも，自分が作成したモデルを，モデル作成者以外の人に説明することが容易になる．

3.1.2　作業量の軽減

図で記述されたモデルからソースコードを自動生成することにより，モデル化以外の本質的でない作業を削減することができる．設計のモデルは，それぞれの図によって定義されたルールに応じて，実行可能なソースコードへと変換される．その中にはプログラミング言語によるソースコードの記述に手間のかかる作業も含まれている．実装作業の中で特に設計モデルからの変換に手間がかかるのは，状態遷移のプログラミングである．ステートチャート図による設計をプログラムとして実現するためには，状態の数に応じて複雑なプログラムを記述する必要がある．このような作業は，シミュレーションを行うプログラ

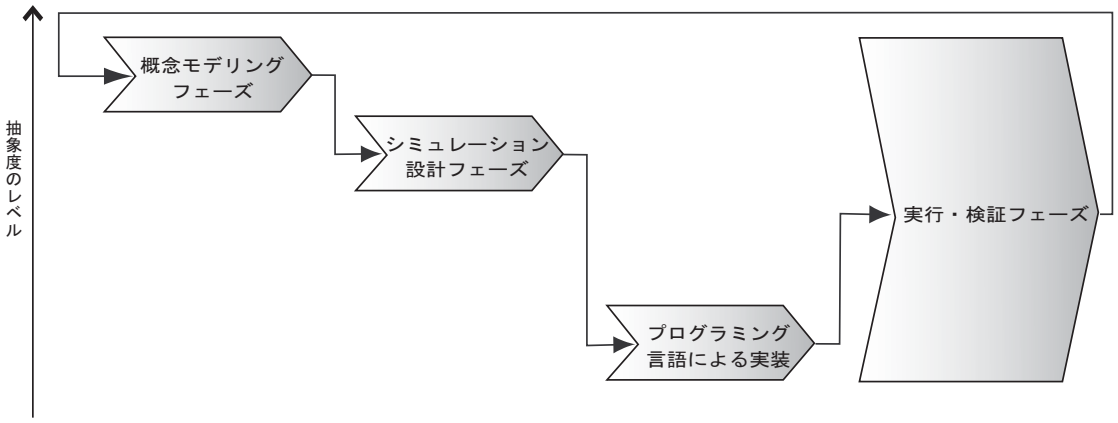


図 3.1: プログラミングをする場合のプロセスと作業の抽象度

ムを作成する上では必要なことであるが，対象領域をモデル化するという点で本質的な作業は状態遷移を設計する作業であり，その実装は本質的な作業ではない．ソースコードを設計モデルから自動生成することで，このようなモデル化と関係のない作業を減らし，モデル化の作業に集中することができる．

3.1.3 反復的なモデル作成のために

シミュレーションのモデル作成は，シミュレーションの正当性，妥当性[1]を検証するために，反復的に行われる (Gilbert and Troitzsch, 1999; 井庭, 2003; Iba et al., 2004)．シミュレーションが実際に動くまでに，モデル作成者は，概念モデリング，シミュレーションデザインを経て，設計をシミュレーションのソースコードに変換する作業を行う必要がある．しかし，作成したモデルは，最初から作成者の意図どおりに正しく動作するとは限らない．様々な原因からバグは発生し，多くの場合，数回の修正作業を行う必要がある．また，作成者の意図どおりに動いたとしても，シミュレーションを行った結果モデルの妥当性に問題があることがわかり，概念モデリングからやりなおすような場合もある．

モデルを作成する上で，分析，設計のモデルをソースコードに変換する作業は，円滑な反復の妨げとなる．最終的にソースコードを記述してシミュレーションのモデルを作成する場合，モデル作成者はプログラミング言語の文法や，実装のための技術などモデルの本質とは直接関係しない部分を気にする必要がある．このような場合，実装から設計や分析の段階に戻る際，一度頭を切り替えてモデルの本質を検討し，次にもう一度頭を切り替えて再検討した部分を実装しなければならない (図 3.1)．

図に記述された分析，設計のモデルからソースコードを自動生成することで，前述した

[1] ここで言う正当性 (verification) とはシミュレーションが正しく実装されていてモデル作成者の意図したとおりに動いているかということであり，妥当性 (validatation) とはモデルの振る舞いがモデル化の対象となる現象と一致しているかということである．詳しくは Gilbert and Troitzsch (1999) を参照してほしい．

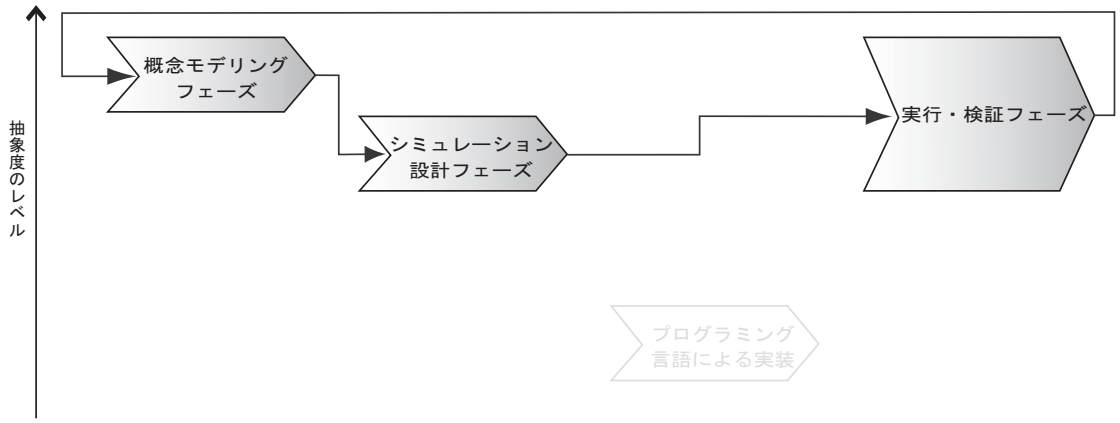


図 3.2: プログラミングをしない場合のプロセスと作業の抽象度

ような切り替えのコストは取り除くことが可能である (図 3.2)．ソースコードを自動生成することで，モデル作成者は，シミュレーションの本質的な部分だけに集中してモデルを作成することができる．

3.1.4 信頼性の向上

シミュレーションのためのモデルを作成する上でモデルの信頼性は非常に重要な課題である．シミュレーションのプログラムは，通常のソフトウェアと違い，どのような振る舞いがあるべき正しい振る舞いなのかを事前に予測することができない．そのため，バグの発見が通常のソフトウェアよりも困難であり，シミュレーションのモデル作成には高い信頼性が求められる．

しかし，ソフトウェアにバグはつき物である (Jones, 1996)．コンピュータ・シミュレーションもその例外ではない．たとえ図に記述したモデルからソースコードを生成したとしても，バグを完全になくすことは不可能である．なぜなら，バグにも種類があり (Beizer, 2003)，分析，設計の段階で混入するバグもあるからである．

しかし，設計を実現するためのソースコードを自動生成することで，実装の段階で起きるバグはなくすことが可能である．設計のモデルをソースコードに変換する作業を人手で行った場合，たとえ設計が正しかったとしても，そこにはバグが混入する可能性がある．ソースコードを自動生成することで人手による作業を減らし，より信頼性の高いプログラムを作成することが可能になる．

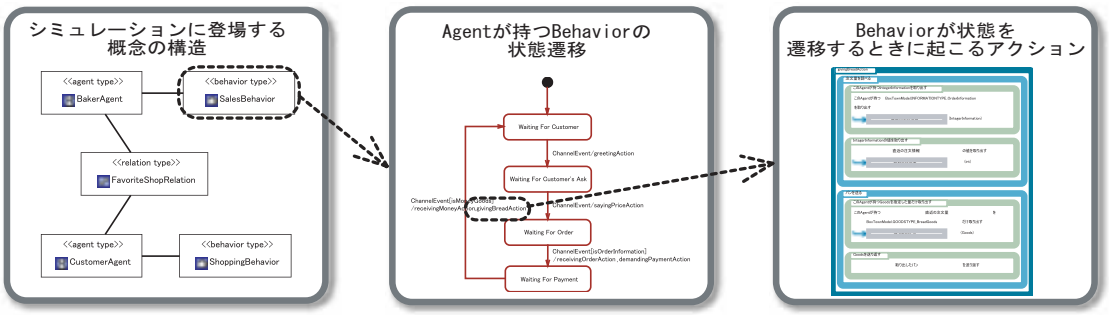


図 3.3: シミュレーションのモデル作成に必要な 3 種類のモデル

3.2 モデル作成に必要な記述

図に記述されたモデルから実行可能なプログラムのソースコードを生成するためには，対象領域を完全に定義したモデルが必要である．本節では，コンピュータによって自動でソースコードを生成するために，モデル作成者は設計モデルで何を記述する必要があるのかについて述べる．

一般的なソフトウェア開発の場合，設計モデルから実行可能なプログラムのソースコードを生成するためには，3 種類のモデルが記述されている必要がある．3 種類のモデルとは，クラス図で表される静的な構造，ステートチャート図で表される動的な振る舞い，オブジェクトが状態を遷移するときに起きるアクションの記述である (Mellor and Balcer, 2002)．

これらの 3 種類のモデルは，シミュレーションのモデルを作成する際に必要な設計のモデルにマッピングすることができる．3 種類のモデルに対応するシミュレーションの設計モデルが記述されていれば，そこからシミュレーションの実行可能なプログラムのソースコードを生成することが可能である (図 3.3)．BEFM のモデル・フレームワークで設計モデルを記述する場合，それらが何に対応し，どのような方法でそれらが記述できるのかについて以下に述べる．

3.2.1 静的な構造の記述

シミュレーションのモデル作成において記述するべき静的な構造は，シミュレーションに登場する概念の構造にあたる．概念の構造は，BEFM ではモデルに登場する Agent，Relation，Behavior，Goods，Information を定義することで表現される．これらは UML のクラス図を用い，Agent や Behavior などの BEFM の要素をクラスとして定義することで記述することができる (図 3.4)．

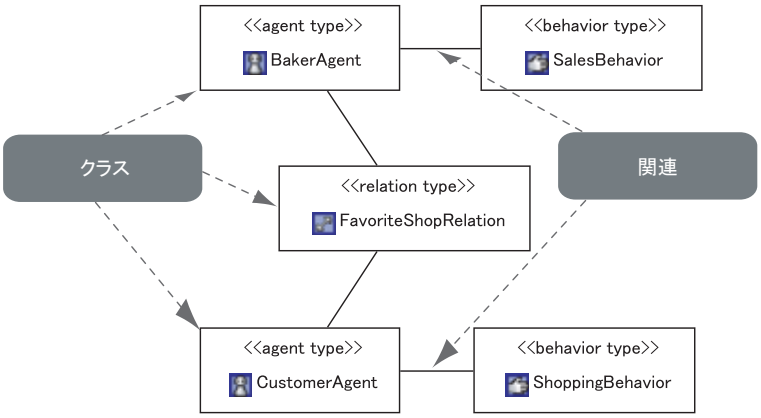


図 3.4: シミュレーションに登場する概念の構造

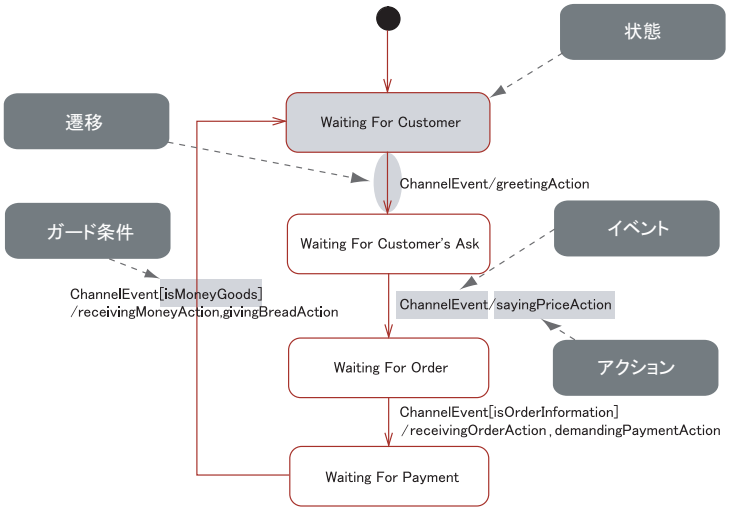


図 3.5: Agent が持つ Behavior の状態遷移

3.2.2 動的な振る舞いの記述

シミュレーションのモデル作成において記述するべき動的な振る舞いは Agent の状態遷移である．BEFM では，Agent の状態遷移はそれぞれの Agent が持つ Behavior の状態遷移としてモデル化される．これらは UML のステートチャート図を用いて記述することができる (図 3.5)．

3.2.3 処理の記述

シミュレーションのモデル作成において記述するべき処理は，Agent が行動するときの具体的な処理である．これは BEFM では，Agent が持つ Behavior が状態遷移するときに

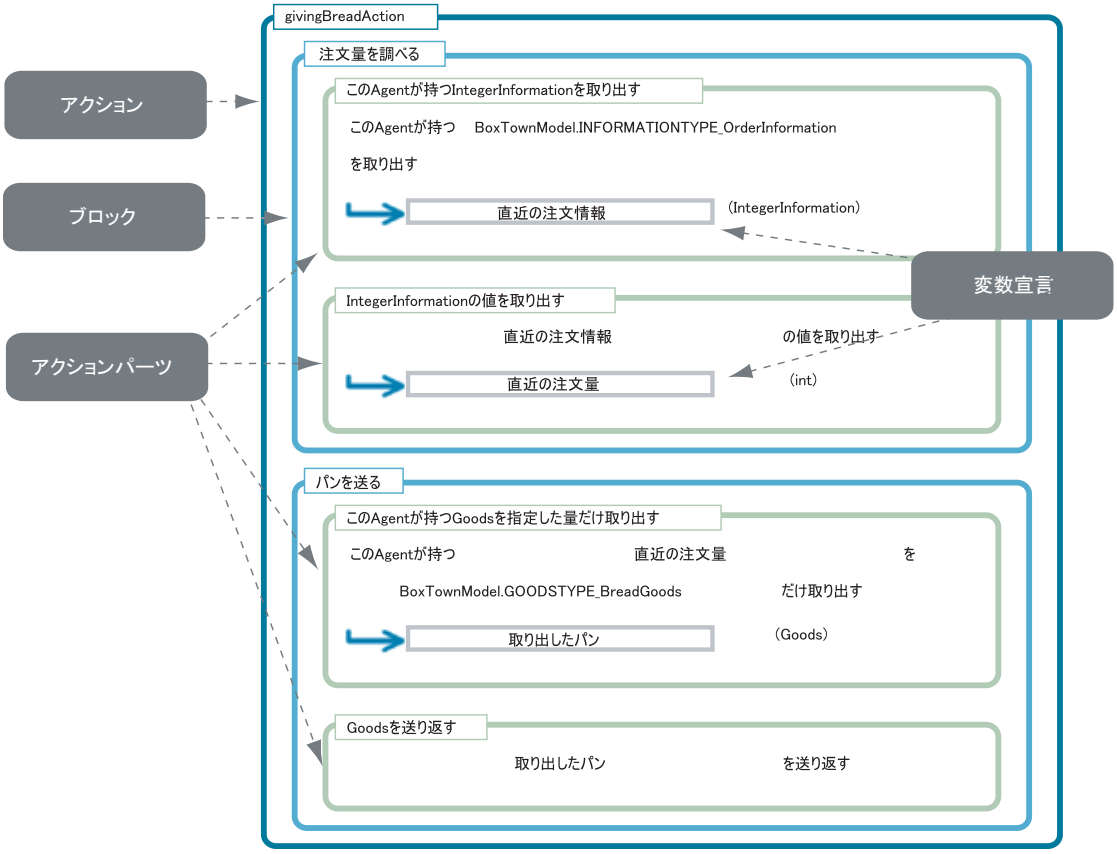


図 3.6: Behavior が状態を遷移するときに起こるアクション

起こるアクションにあたる．これらは後述するアクション記述言語を利用して図で記述することができる (図 3.6)．

3.3　アクション記述言語「Action Block Language」(ABL)

3.3.1　社会シミュレーションのモデル作成に特化した言語

プログラミング言語の種類に依存しない汎用的なアクション記述言語を定義しただけでは，プログラミング言語の知識のないモデル作成者には状態が遷移するときに起きる処理を記述することはできない．例えば，OAL(Object Action Language)(Project Technology, 2002) のようなアクションを記述するための汎用的な言語[2]を使って，実行可能なプログラムのソースコードを生成しようとした場合，モデル作成者は処理をかなり詳細まで記述する必要がある．具体的にどのような変数を定義して，どのメソッドを呼ぶのか，といった

[2]アクションを記述するための特定のプログラミング言語に依存しない汎用的な言語には OCL，OAL の他にも SMALL，TALL などがある．

ことを特定のプログラミング言語で記述するのと同じ粒度まで記述しなければならないのである．そのような汎用的なアクション記述言語を開発したとしても，モデル作成者は新たな言語を覚えなければならないだけで，プログラミング言語の知識がなければ処理を記述するのは難しい．

このような問題を解決するため，本論文では社会シミュレーションのモデル作成に特化したアクション記述言語「Action Block Language」(ABL) を提案する．ABL は，BEFM で定義されている Agent，Relation，Information などの概念を基にして作成されている．また，ABL には「この Agent が持つ Goods を全て取り出す」というような，シミュレーションのモデル作成において頻出する処理の集まりが，語彙として定義されている．

ABL を使うことでモデル作成者は，より抽象度の高い記述でアクションのモデル化を行うことができる．例えば，これまでモデル作成者はアクションを実装する際,「this. getAgent(). removeAllGoods(GoodsType type)」というメソッドを呼ぶ，というプログラムを考える必要があった (図 3.7 [a])．汎用的なアクション記述言語を使っても，特定のプログラミング言語に依存しないというだけで，考えなければならないことはあまり変わらない (図 3.7 [b])．ABL を使うと,「この Agent が持つ Goods を全て取り出す」というシミュレーションの中で行いたい処理の目的を考え，それに対応する語句を ABL に定義されている語彙の中から選択することでアクションを記述することができる (図 3.7 [c])．

3.3.2 Action Block Language(ABL) の文法

ABL の文法には，制御構造と，論理的な構造を扱える仕組みが必要である．モデル作成者はときとして複雑な処理をアクションとして記述する．そのような場合，モデル作成者はアクションで行いたい処理をただ並べて書くのではなく，制御構造や論理的な構造を使い処理を階層化して記述する必要がある．

このために，ABL には「構造を記述するための要素」と「文を記述するための要素」の 2 種類の要素が文法として用意されている.「文を記述するための要素」はプログラミング言語における式の呼び出しなどの一般的な文や，制御文を記述するための要素である.「構造を記述するための要素」は制御構造と論理的な構造を扱うための要素である.「構造を記述するための要素」は下位の要素として,「文を記述するための要素」と他の「構造を記述するための要素」を持つことができる．

ABL には，制御構造を扱うための文法が定義されている．制御構造を表す要素の下位の要素として具体的な処理や，他の制御構造を定義することで，下位の要素を「繰り返す」もしくは「条件が真ならば実行する」という記述をすることができる．

また，ABL には制御構造以外に論理的な構造を記述するための文法が定義されている．論理的な構造はプログラムとしては特に意味を持たないが，これを使うことで処理のまとまりを階層化して記述することができる．モデル作成者は，まず論理的な処理の階層構造を設計し，次に設計した構造を実現するための手段として処理を記述していく．このよう

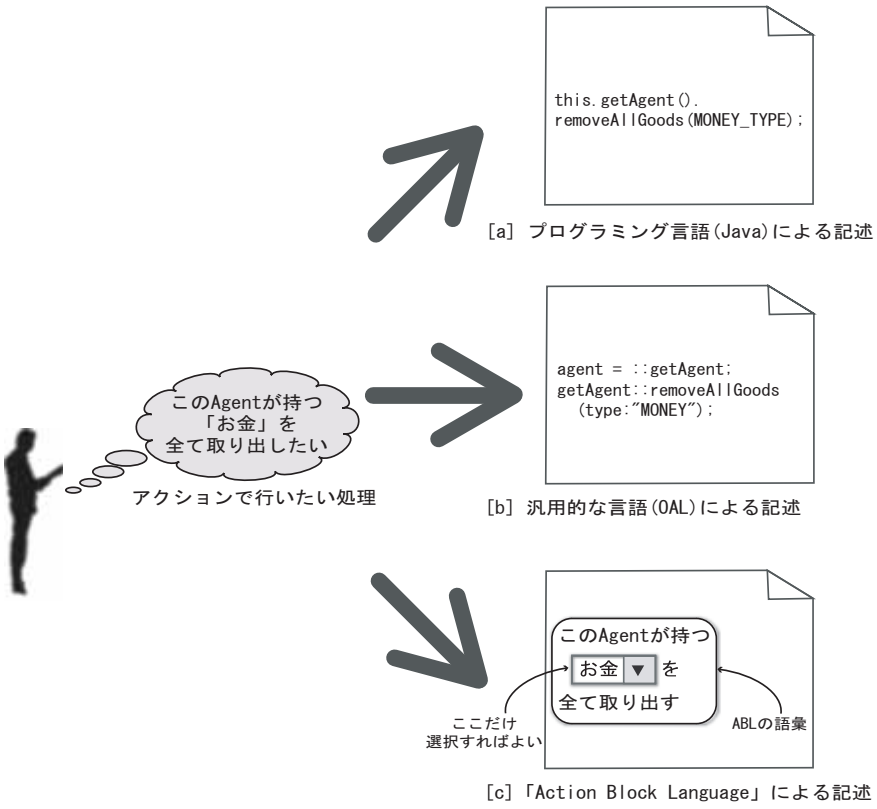


図 3.7: アクション記述の比較

な考え方は HCP チャート (花田, 1983) にも共通する考え方である．

複雑なアクションを構造化して記述するために，アクション記述言語には 6 種類の「構造を記述するための要素」と，8 種類の「文を記述するための要素」が定義されている．以下にそれぞれの詳細を述べる．

構造を記述するための要素

(1) メソッド

プログラミング言語におけるメソッドと同じ役割を持つ．複雑なアクションを構造化したり，アクションの中で重複する記述をサブルーチンとして定義したりするために使用される．

(2) ブロック

処理の論理的なまとまりを構造として記述するために利用される．プログラミング言語における一般的なブロックの定義とは違い，変数のスコープには関係しない．ブロックには名前をつけることができ，ブロックの中に定義された下位の要素の目的を記述することができる．

(3) 条件分岐

条件分岐はプログラミング言語における if 文に相当する構造である．条件分岐には，条件式として真偽値の変数を設定することができる．

(4) 集合操作

集合操作は，集合のそれぞれの要素に対して何らかの処理を行いたいときに利用することができる構造である．集合操作には，操作を行いたい対象の集合と，集合の要素をどのような型のオブジェクトとして扱うかを指定することができる．

(5) 回数繰り返し

回数繰り返しは何らかの処理を数回繰り返して行いたい場合に利用することができる構造である．回数繰り返しには，下位の処理を何回繰り返すかを変数や値を使って設定することができる．

(6) 条件付繰り返し

条件付繰り返しは何らかの処理を一定の条件を満たすまで繰り返して行いたい場合に利用することができる構造である．条件付繰り返しには，繰り返しの継続条件を設定することができる．

文を記述するための要素

(1) アクションパーツ

アクションパーツは，ABL に語彙として定義された，アクション記述において頻出する処理を利用するための文である．アクションパーツにはそれぞれ「この Agent が持つ Information を取り出す」というような，BEFM の用語で記述された名前がつけられている．また，アクションパーツにはそれぞれ引数が定義されている．アクションパーツに引数を渡すことで，アクションパーツで操作する対象のオブジェクトを動的に変更したり，

設定する値を動的に変更したりすることができる．どのようなアクションパーツが語彙として提供されているのかについては次節に詳しく述べる．

(2) 変数/操作

アクションパーツとして用意されていない操作を行うための文である．呼び出したいメソッドを選択し，必要ならば結果を格納する変数を定義することができる．アクション記述の中では，モデル作成者が自ら定義したクラスやメソッドを利用するために使われる．

(3) 論理式

論理式を作成し，その結果を変数として定義するための文である．論理式の中では，真偽値だけでなく，定義済みの他の変数を使うこともできる．

(4) 算術式

実数[3]の値を式として設定し，その結果を変数として定義するための文である．算術式の中では，値だけでなく，定義済みの他の変数を使うこともできる．

(5) 文字列

アクション記述の中で使用する文字列を変数として定義するための文である．変数には，新たな文字列を値として格納するだけでなく，新たな文字列と他の変数に格納された文字列をつなげた結果を格納することもできる．

(6) return 文

return 文はプログラミング言語における return 文と同じく，メソッドが返す値を決定してメソッドの処理を中途脱出するための文である．

(7) continue 文

continue 文はプログラミング言語における continue 文と同じく，繰り返される処理を途中で止め，繰り返しの最初に戻って処理を行うための文である．そのため continue 文は集合操作，回数繰り返し，条件付繰り返しの中でしか使用することができない．

[3] 正確には浮動小数点数だけがあつかえる．

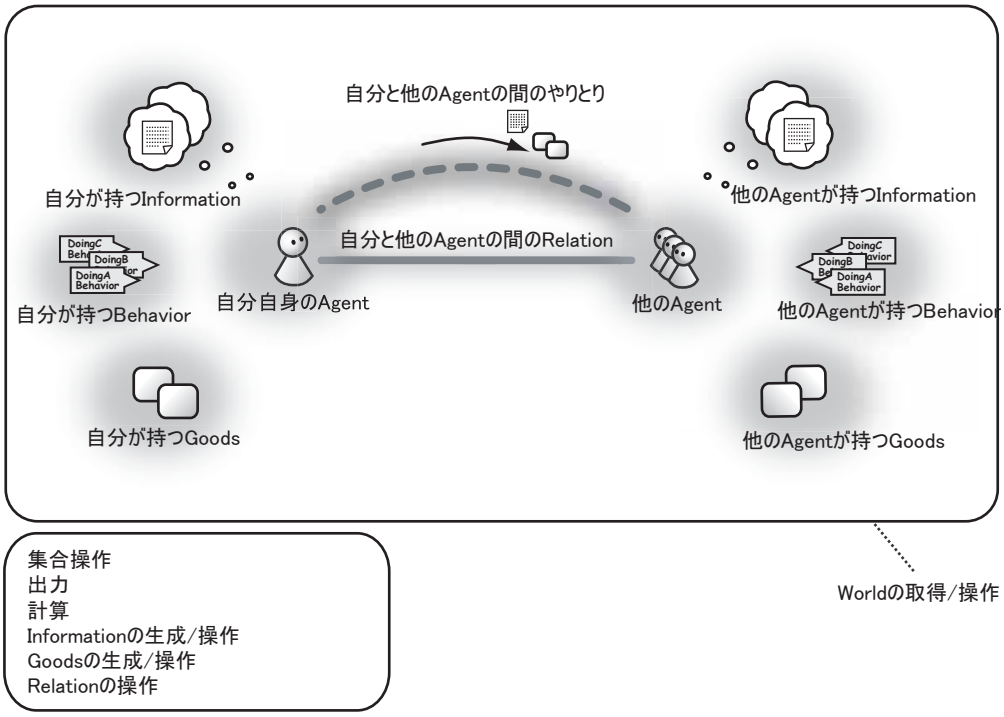


図 3.8: アクションパーツのカテゴリ

(8) break 文

break 文はプログラミング言語における break 文と同じく，繰り返される処理を途中で止め，繰り返しを中途脱出するための文である．そのため break 文は集合操作，回数繰り返し，条件付繰り返しの中でしか使用することができない．

3.3.3 Action Block Language(ABL) の語彙

ABL には語彙として 100 個以上のアクションパーツが用意されている．アクションパーツにはそれぞれ「この Agent が持つ Information を取り出す」というような，BEFM の用語で記述された名前がつけられている．モデル作成者はアクションを記述するために，制御構造やブロックを使って処理の構造を設計し，その中に文としてアクションパーツを配置する．

アクションパーツはいくつかのカテゴリに分けて定義されている (図 3.8)．100 を超える数のアクションパーツから利用したいものを迅速に探すためには，カテゴリによる分類が必要である．

以下はカテゴリごとに分類されたアクションパーツの一覧である．なお，これらのアクションパーツのがどのような引数を持ち，どのような Java のソースコードに対応してい

るかについては，付録 A を参照してほしい．

- カテゴリ「自分自身の Agent」
  - この Agent が指定した Type の Behavior を持っているか調べる
  - この Agent が指定した Type の Goods を持っているか調べる
  - この Agent が指定した Type の Information を持っているか調べる
  - この Agent が指定した Type の Relation を持っているか調べる
  - この Agent の Type を取得する
  - この Agent を消滅させる
  - 自分自身の Agent を取得する
- カテゴリ「自分が持つ Behavior」
  - この Agent が持つ Behavior を削除する
  - この Agent が持つ Behavior を取得する
  - この Agent が持つ Behavior を親 Type を指定して再帰的に取得する
  - この Agent が持つ Behavior を全て取得する
  - この Agent が持つ指定した Type の Behavior を全て取得する
  - この Agent に Behavior を追加する
  - この Behavior の Type を取得する
  - この Behavior の現在の状態名を取得する
- カテゴリ「自分が持つ Information」
  - この Agent が持つ DoubleInformation の値を減らす
  - この Agent が持つ DoubleInformation の値を設定する
  - この Agent が持つ DoubleInformation の値を増やす
  - この Agent が持つ DoubleInformation を取得する
  - この Agent が持つ Information を削除する
  - この Agent が持つ Information を取得する
  - この Agent が持つ Information を全て取得する
  - この Agent が持つ IntegerInformation の値を減らす
  - この Agent が持つ IntegerInformation の値を設定する
  - この Agent が持つ IntegerInformation の値を増やす

  - この Agent が持つ IntegerInformation を取得する
  - この Agent に Information を記憶させる

- カテゴリ「自分が持つ Goods」
  - この Agent が持つ Goods の Type を全て取得する
  - この Agent が持つ Goods の量の合計を親 Type を指定して取得する
  - この Agent が持つ Goods の量を Type を指定して取得する
  - この Agent が持つ Goods を指定した量だけ取り出す
  - この Agent が持つ Goods を親 Type と量を指定して取り出す
  - この Agent が持つ Goods を親 Type を指定して全て取り出す
  - この Agent が持つ指定した Type の Goods を全て取り出す
  - この Agent に Goods を持たせる

- カテゴリ「自分と他の Agent の間の Relation」
  - この Agent から他のエージェントへ Relation をむすぶ
  - この Agent が持つ Relation の Type を全て取得する
  - この Agent が持つ Relation を Type を指定して全て取得する
  - この Agent が持つ Relation を削除する
  - この Agent が持つ Relation を取得する
  - この Agent が持つ Relation を親 Type を指定して全て削除する
  - この Agent が持つ Relation を親 Type を指定して全て取得する
  - この Agent が持つ Relation を相手の Agent を指定して取得する
  - この Agent が持つ指定した Type の Relation を全て削除する
  - この Agent と他の Agent の間で双方向の Relation をむすぶ
  - この Agent と他人の間の双方向の Relation を削除する
  - この Agent の Relation を全て取得する
  - 現在アクティブな Channel を開いている関係を取得する
  - 他の Agent からこの Agent へ Relation をむすぶ
  - 他の Agent から他の Agent へ Relation をむすぶ
  - 他の Agent が持つ Relation の Type を全て取得する
  - 他の Agent が持つ Relation を Type を指定して全て取得する
  - 他の Agent が持つ Relation を削除する

- 他の Agent が持つ Relation を取得する
- 他の Agent が持つ Relation を親 Type を指定して全て削除する
- 他の Agent が持つ Relation を親 Type を指定して全て取得する
- 他の Agent が持つ Relation を相手の Agent を指定して取得する
- 他の Agent が持つ指定した Type の Relation を全て削除する
- 他の Agent の Relation を全て取得する
- 他人と他人の間で双方向の Relation をむすぶ
- 他人と他人の間の双方向の Relation を削除する

- カテゴリ「自分と他の Agent の間のやりとり」
  - Goods や Information を送ってきた相手にメッセージを送る
  - Goods を送り返す
  - Information を送り返す
  - RelationType と一人一人に送る量を指定して全員に Goods を送る
  - RelationType と一人一人に送る量を指定して全員に Goods を送る (Channel はキープする)
  - RelationType を指定してメッセージを送る
  - Relation を指定して Goods の量が N 以上だったら Goods を N だけ送る
  - Relation を指定して Goods の量が N 以上だったら Goods を N だけ送る（Channel はキープする）
  - Relation を指定して Goods を送る
  - Relation を指定して Goods を送る（Channel はキープする）
  - Relation を指定して一人に Information を送る
  - Relation を指定して一人に Information を送る（Channel はキープする）
  - 現在アクティブな Channel をつないだ元の Behavior を取得する
  - 現在アクティブな Channel をつないだ先の Behavior を取得する
  - 現在アクティブな Channel を取得する
  - 現在アクティブな Channel を閉じる
  - 最後に受け取った Goods が指定された Type のものか調べる
  - 最後に受け取った Goods を持つ
  - 最後に受け取った Goods を取得する
  - 最後に受け取った Information が指定された Type のものか調べる

  - 最後に受け取った Information を DoubleInformation として取得する
  - 最後に受け取った Information を IntegerInformation として取得する
  - 最後に受け取った Information を記憶する
  - 最後に受け取った Information を取得する
  - 指定した RelationType の Relation を持つ Agent 全員に Information を送る
  - 指定した RelationType の Relation を持つ Agent 全員に Information を送る (Channel はキープする)
  - 指定した RelationType の Relation 一つを取り出し Goods を送る
  - 指定した RelationType の Relation 一つを取り出し Information を送る
  - 指定した RelationType の Relation 一つを取り出し Information を送る（Channel はキープする）
  - 全ての Channel を取得する
- カテゴリ「他の Agent」
  - 新しい Agent を作る
  - 他の Agent が指定した Type の Behavior を持っているか調べる
  - 他の Agent が指定した Type の Goods を持っているか調べる
  - 他の Agent が指定した Type の Information を持っているか調べる
  - 他の Agent が指定した Type の Relation を持っているか調べる
  - 他の Agent の Type を取得する
  - 他の Agent を消滅させる
- カテゴリ「他の Agent が持つ Behavior」
  - 他の Agent が持つ Behavior を削除する
  - 他の Agent が持つ Behavior を取得する
  - 他の Agent が持つ Behavior を親 Type を指定して再帰的に取得する
  - 他の Agent が持つ Behavior を全て取得する
  - 他の Agent が持つ指定した Type の Behavior を全て取得する
  - 他の Agent に Behavior を追加する
- カテゴリ「他の Agent が持つ Information」
  - 他の Agent が持つ DoubleInformation の値を減らす
  - 他の Agent が持つ DoubleInformation の値を設定する

  - 他の Agent が持つ DoubleInformation の値を増やす
  - 他の Agent が持つ DoubleInformation を取得する
  - 他の Agent が持つ Information を削除する
  - 他の Agent が持つ Information を取得する
  - 他の Agent が持つ Information を全て取得する
  - 他の Agent が持つ IntegerInformation の値を減らす
  - 他の Agent が持つ IntegerInformation の値を設定する
  - 他の Agent が持つ IntegerInformation の値を増やす
  - 他の Agent が持つ IntegerInformation を取得する
  - 他の Agent に Information を記憶させる
- カテゴリ「他の Agent が持つ Goods」
  - 他の Agent が持つ Goods の Type を全て取得する
  - 他の Agent が持つ Goods の量の合計を親 Type を指定して取得する
  - 他の Agent が持つ Goods の量を Type を指定して取得する
  - 他の Agent が持つ Goods を指定した量だけ取り出す
  - 他の Agent が持つ Goods を親 Type と量を指定して取り出す
  - 他の Agent が持つ Goods を親 Type を指定して全て取り出す
  - 他の Agent が持つ指定した Type の Goods を全て取り出す
  - 他の Agent に Goods を持たせる
- カテゴリ「Goods の生成/操作」
  - Goods に Information を付与する
  - Goods に指定された Type の Information が付随しているか調べる
  - Goods に付随する Information を取得する
  - Goods に付随する Information を全て取得する
  - Goods に付属する Information を削除する
  - Goods の Type を取得する
  - Goods の量を取得する
  - Goods の量を小数のオブジェクトとして取得する
  - Goods の量を小数の値として取得する
  - Goods の量を整数のオブジェクトとして取得する

  - Goods の量を整数の値として取得する
  - Goods を消費する
  - 新しい Goods を作る
- カテゴリ「Information の生成/操作」
  - DoubleInformation の値を取得する
  - DoubleInformation をつくる
  - IntegerInformation の値を取得する
  - IntegerInformation をつくる
- カテゴリ「Relation の操作」
  - Relation の Type を取得する
  - Relation の関係元を取得する
  - Relation の関係先を取得する
- カテゴリ「World の取得/操作」
  - World が持つ Clock を取得する
  - World が持つ Space を取得する
  - World に存在する全ての Agent を取得する
  - World に定義された AgentType を取得する
  - World に定義された BehaviorType を取得する
  - World に定義された GoodsType を取得する
  - World に定義された InformationType を取得する
  - World に定義された RelationType を取得する
  - World の説明を取得する
  - World の名前を取得する
  - World を取得する
  - 現在のステップ数を取得する
  - 指定した AgentType に対応する TimeEvent の優先度を取得する
  - 世界に存在する Agent の中から指定した Type の Agent を一人取得する
  - 世界に存在する指定した Type の Agent を親 Type を指定して全て取得する
  - 世界に存在する指定した Type の Agent を全て取得する

  - 世界に定義されている整数型 (int) のパラメータの値を取得する
  - 乱数ジェネレータを取得する
  - 乱数ジェネレータを名前を指定して取得する
- カテゴリ「計算」
  - 0 以上 1 未満の実数の乱数を生成する
  - 実数配列の合計と平均を計算する
  - 整数の乱数を生成する
  - 整数配列の合計と平均を計算する
- カテゴリ「集合操作」
  - Agent の集合からランダムで N 人の Agent を取り出す
  - Agent の集合からランダムで一人を選ぶ
  - Behavior の集合からランダムで一つを選ぶ
  - Goods の集合からランダムで一つを選ぶ
  - Information の集合からランダムで一つを選ぶ
  - Relation の集合からランダムで N 個の要素を取り出す
  - Relation の集合からランダムで一つを選ぶ
  - マップに要素を追加する
  - マップのキーを取得する
  - マップの要素を取得する
  - マップの要素を集合として取得する
  - リストの N 番目を削除する
  - リストの N 番目を取得する
  - 集合からリストを作る
  - 集合から要素を削除する
  - 集合に要素を追加する
  - 集合の要素数を取得する
- カテゴリ「出力」
  - デバッグとしてログを出力する
  - 警告としてログを出力する
  - 情報としてログを出力する
  - 標準出力に出力する

第4章　モデル作成環境「Component Builder」(CB)

本論文では，図解によるモデル作成を支援する環境である「Component Builder」(CB)を提案する．CB は 6 種類のツールから構成されている[1]．各ツールはモデルを図解で記述し，必要に応じてソースコードを生成する機能を提供している．本章では CB を構成するツール群の詳細と，それらを実現するための設計について述べる．

4.1　概念モデリングのためのツール

CB には概念モデリングフェーズをサポートするための 3 種類のツール「Model Designer」,「Activity Designer」,「Communication Designer」が含まれている．モデル作成者はこれらのツールを使用して，対象領域からの概念の抽出，Agent のアクティビティの分析，Agent 間の相互作用の分析，を反復的に行いながら概念モデルを作成していく (図 4.1)．以下ではこれらの 3 種類のツールの詳細について述べる．

4.1.1　Model Designer

Model Designer は UML のクラス図を使ってシミュレーションモデルの静的な構造を記述するためのツールである (図 4.2)．概念モデリングフェーズでは，モデル作成者はこのツールを使って，シミュレーションの対象領域から概念を抽出しクラスとして記述する．概念モデリングフェーズで抽出するべき要素は，Agent，Relation，Information，Goods，Behavior の 5 種類である．

モデル作成者は，まず画面左側のパレットからクラス図に描きたい BEFM の要素を選択し，キャンバス上に配置する (図 4.3 左側)．クラスの間に関連を引きたいときは，パレットから「Association」を選択し，両端のクラスをそれぞれ選択することで関連を引くことができる (図 4.3 右側)．

[1]CB には，概念モデリングフェーズを支援する 3 種類のツールと，シミュレーションデザインフェーズを支援する 4 種類のツールが含まれているが，Model Designer は概念モデリング，シミュレーションデザインの両方のフェーズで使用するため，CB は合計で 6 種類のツールで構成されている．

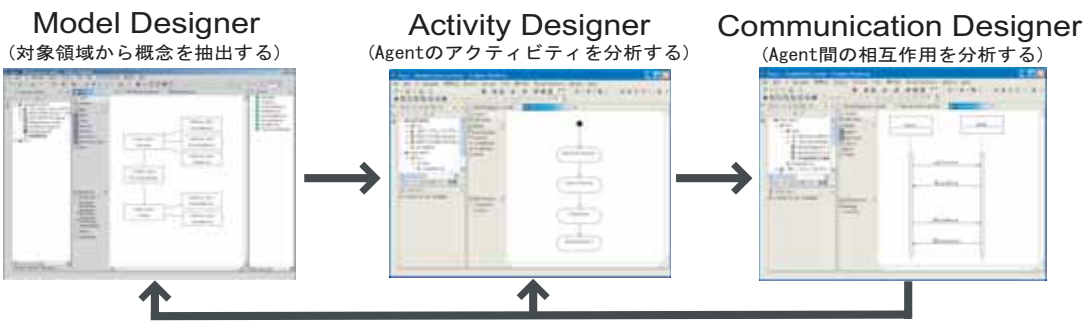


図 4.1: 概念モデリングフェーズをサポートするツール

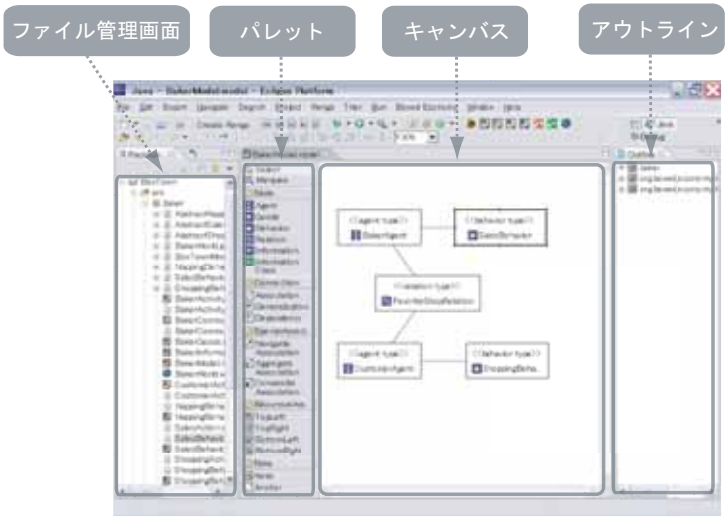


図 4.2: Model Designer

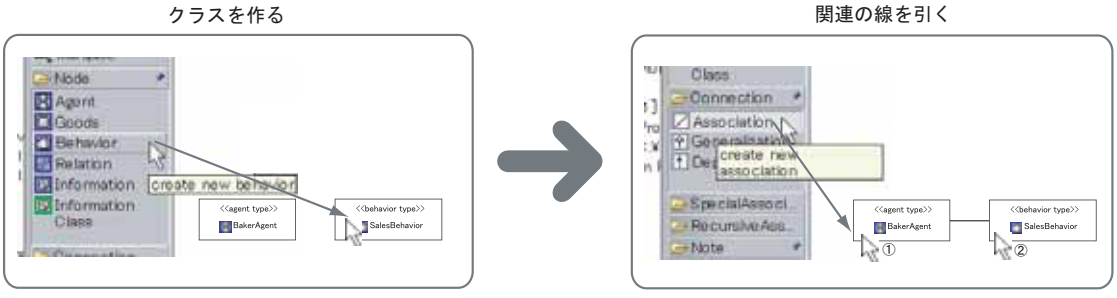


図 4.3: クラス図の作成プロセス

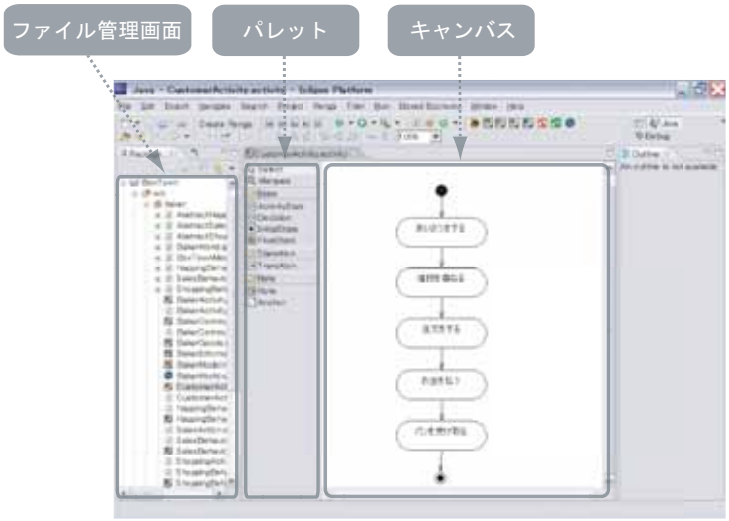


図 4.4: Activity Designer

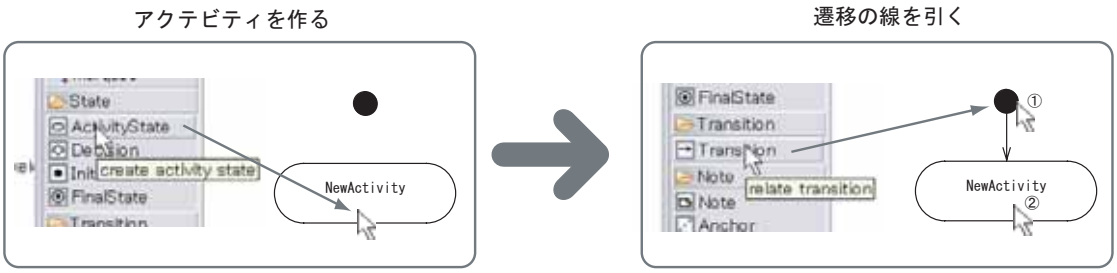


図 4.5: アクテビティ図の作成プロセス

4.1.2 Activity Designer

Activity Designer は，UML のアクティビティ図を使って Agent のアクティビティを分析するためのツールである (図 4.4)．分析したアクティビティと遷移は，シミュレーションデザインフェーズで Behavior の状態遷移を設計する際に利用される．

モデル作成者はまず，画面左側のパレットから初期状態やアクティビティを選択し, キャンバス上に配置する (図 4.5 左側)．次にパレットから「Transition」を選択し, 両端の要素をそれぞれ選択することで遷移の線を引くことができる (図 4.5 右側)．

4.1.3 Communication Designer

Communication Designer は，UML のシーケンス図を使って Agent 間の相互作用を分析するためのツールである (図 4.6)．モデル作成者は，Model Designer で定義した静的な構造と，Activity Designer で分析した Agent のアクティビティをもとに，Agent がどのよ

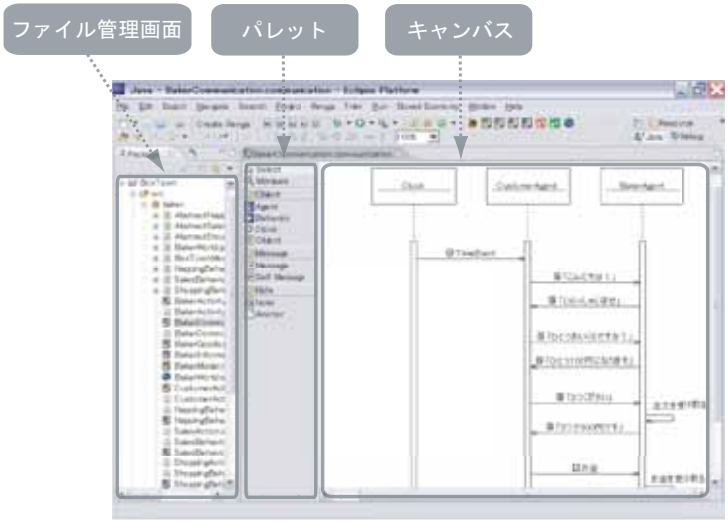


図 4.6: Communication Designer

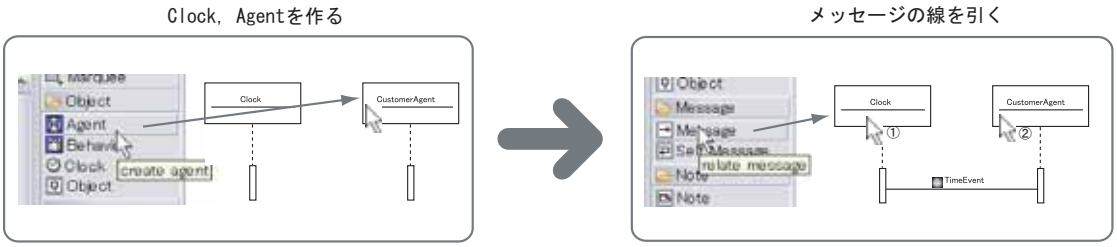


図 4.7: シーケンス図の作成プロセス

うな順番で活動を行い，その中でどの Goods や Information を送りあうのかを記述する．

モデル作成者はまず，Clock[2] とシーケンス図に登場する Agent を記述するために，それぞれの要素をパレットから選択して，任意の場所に配置する (図 4.7 左側)．次に，Agent 同士の相互作用を記述するために，パレットから「Message」を選択し，相互作用の線を引いていく (図 4.7 右側)．

4.2 シミュレーションデザインのためのツール

CB にはシミュレーションデザインフェーズをサポートする 4 種類のツール「Model Designer」,「Behavior Designer」,「Action Designer」,「World Composer」が含まれている．モデル作成者はこれらのツールを使用して，まずシミュレーションに登場する Type

[2]Clock は BEFM に定義されている時間を管理する概念である．BEFM シミュレーションモデル・フレームワークでは，Clock が発信する TimeEvent(時計信号) に反応して，それぞれの Agent が持つ Behavior が起動する．詳細な定義については井庭 (2003) を参照してほしい．

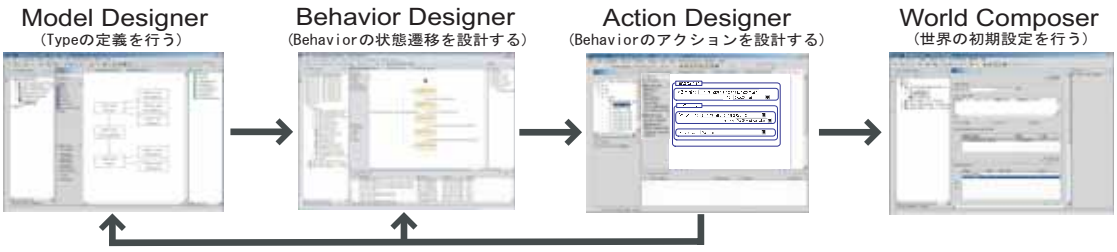


図 4.8: シミュレーションデザインフェーズをサポートするツール

の設計，Behavior の状態遷移の設計，アクションの設計，を反復的に行いながら設計モデルを作成していく．次に，シミュレーションの初期設定を行い，対応するソースコードを生成することで，モデル作成者は BESP 上で実行可能なモデルコンポーネントを得ることができる (図 4.8)．シミュレーションデザインをサポートする 4 種類のツールは，設計モデルからシミュレーションのプログラムを生成する機能を提供している．以下ではこれらのツールの詳細について述べる．

4.2.1 Model Designer

概念モデリングフェーズで使用した Model Designer は，シミュレーションデザインフェーズでは，概念モデルで記述した静的な構造を Type の構造として定義するために使用する．概念モデルとして抽出した要素を，拡張可能性，再利用性の観点から再検討し，必要ならば分割，統合などの設計判断を行って Type として定義する．

Model Designer は，定義された Type の構造からソースコードを自動で生成する機能を提供している．また，Model Designer には定義された Behavior を Behavior Designer で開く機能がある．これらの機能は，キャンバス上で右クリックすると現れるメニューから利用することができる．

4.2.2 Behavior Designer

Behavior Designer は，UML のステートチャート図を使って Behavior の状態遷移を設計するためのツールである (図 4.9)．

モデル作成者はまず，パレットから任意の「State」を選択し，キャンバス上に配置する (図 4.10 左側)．次にパレットから「Transition」を選択し，両端の要素をそれぞれ選択することで遷移の線を引くことができる (図 4.10 右側)．

Behavior Designer には，図中で定義されたアクションを設計するために，Action Designer を起動する機能がある．キャンバス上で右クリックすると現れるメニューから Behavior のアクションを Action Designer で開くことができる．このとき同時に，ステート

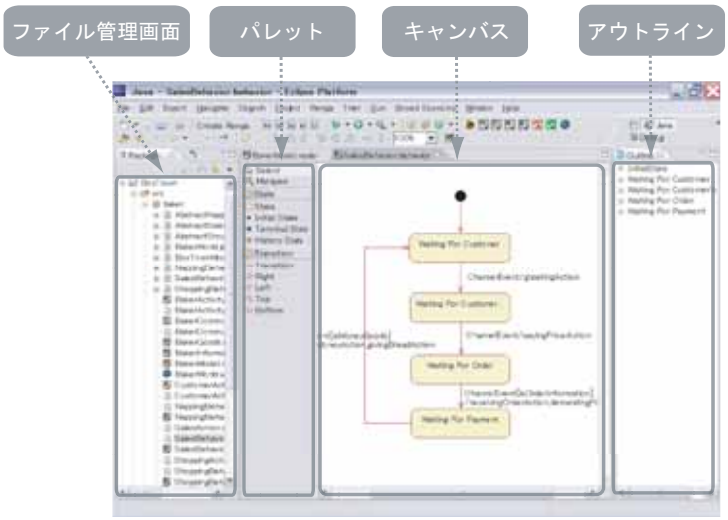


図 4.9: Behavior Designer

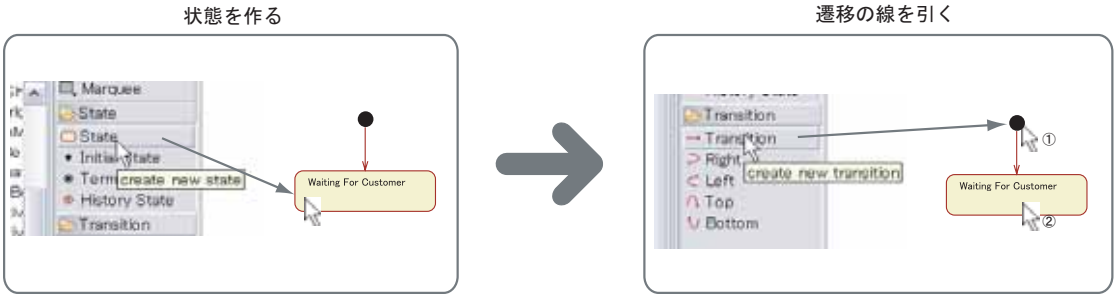


図 4.10: ステートチャート図の作成プロセス

チャート図で記述された状態遷移に対応するソースコードが，自動で生成される．

4.2.3 Action Designer

Action Designer は Behavior に定義されたアクションを設計するためのツールである (図 4.11)．モデル作成者は，このツールを使ってアクション記述言語を GUI で記述することができる．

モデル作成者はまず，パレットから構造を記述するための要素を選択し，論理構造や制御構造をメソッドの中に記述していく (図 4.12 左側)．構造を記述し終えたら，使用するアクションパーツのカテゴリをパレットから選択し，配置する場所を決める (図 4.12 右側)．次に，配置場所を決めると開くダイアログから使用するアクションパーツを選択する (図 4.13)．記述したアクションは対応するソースコードに自動で変換することができる．

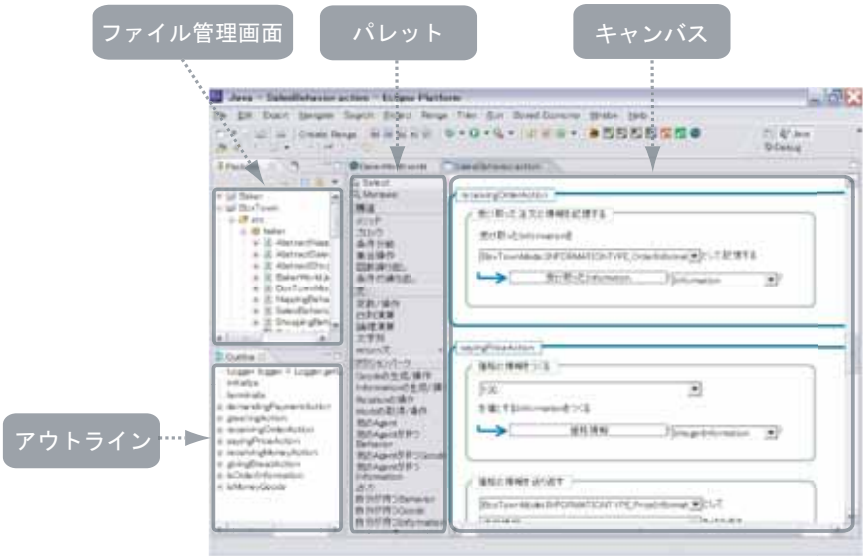


図 4.11: Action Designer

図 4.12: アクションの作成プロセス

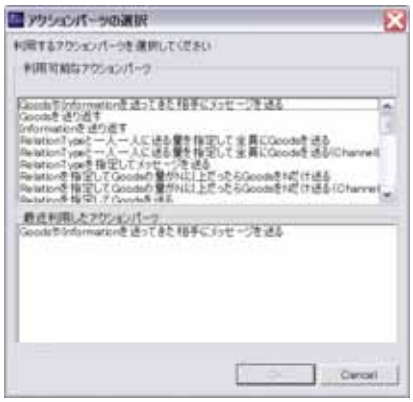


図 4.13: アクションパーツ選択ダイアログ

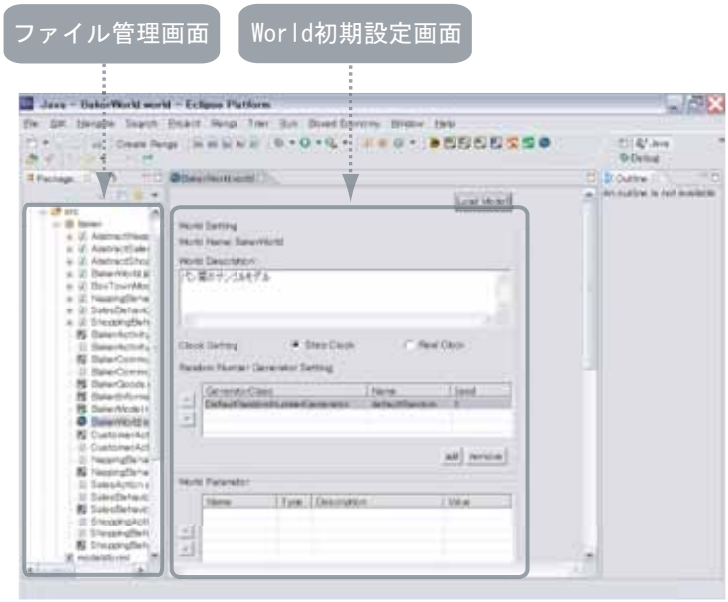


図 4.14: World Composer

4.2.4 World Composer

World Composer [3]は，シミュレーションの初期設定を行うためのツールである (図 4.14)．

モデル作成者は，このツールを使って，World の各種設定 (図 4.15(1) から (4))，Agent の初期配置の設定 (図 4.15(5)), Relation の初期構造の設定 (図 4.15(6)) を行うことができる．これらの設定は，World のソースコードに自動で変換することができる．

[3]World Composer だけは他のツールと違い「Desginer」という名前にはなっていない．World Composer は他のツールで「Design」したモデルを「Compose」してモデルコンポーネントとして実行できるようにするツールだからである．

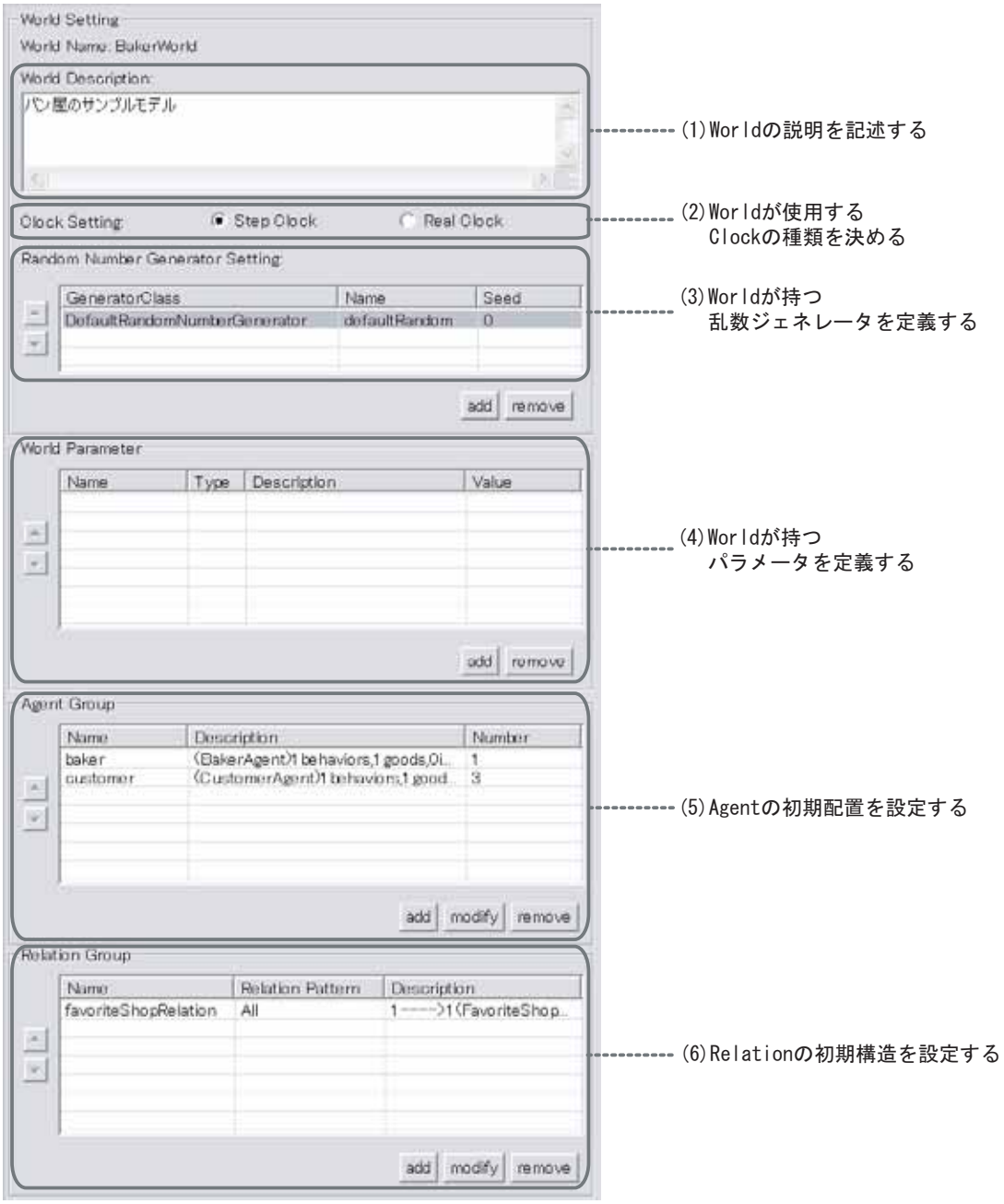


図 4.15: World の初期設定のプロセス

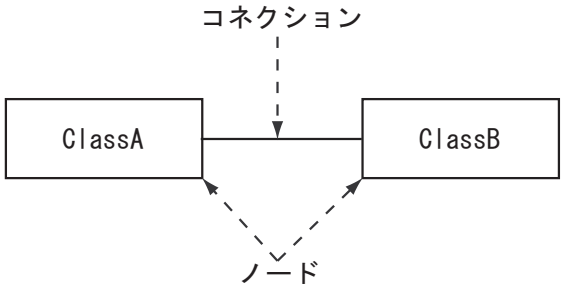


図 4.16: ノードとコネクション

4.3 Component Builder の設計と実装

CB は円滑なモデル作成と作成したモデルの共有を支援するために，作図やソースコード生成の機能だけでなく様々な機能を提供している．本節では，CB が持つ様々な機能と，それらを効率よく実装するために CB が採用しているアーキテクチャについて述べる．

4.3.1 Component Builder の機能

図解エディタとしての機能

CB を構成するツール群は，図解によるモデリングをサポートするために以下のような共通の機能を持っている．ここで言う「ノード」とは，クラス，状態など大きさと位置を持つものであり,「コネクション」とはノードの間に引かれる線のことである (図 4.16).

- ノード (クラス，状態，アクティビティなど) の管理
  - ノードの作成
  - ノードの削除
  - ノードの移動
  - ノードのサイズ変更
- コネクション (関連，汎化，遷移など) の管理
  - コネクションの作成
  - コネクションの削除
  - コネクションのつなぎかえ
  - コネクションの曲がり方の選択
- その他

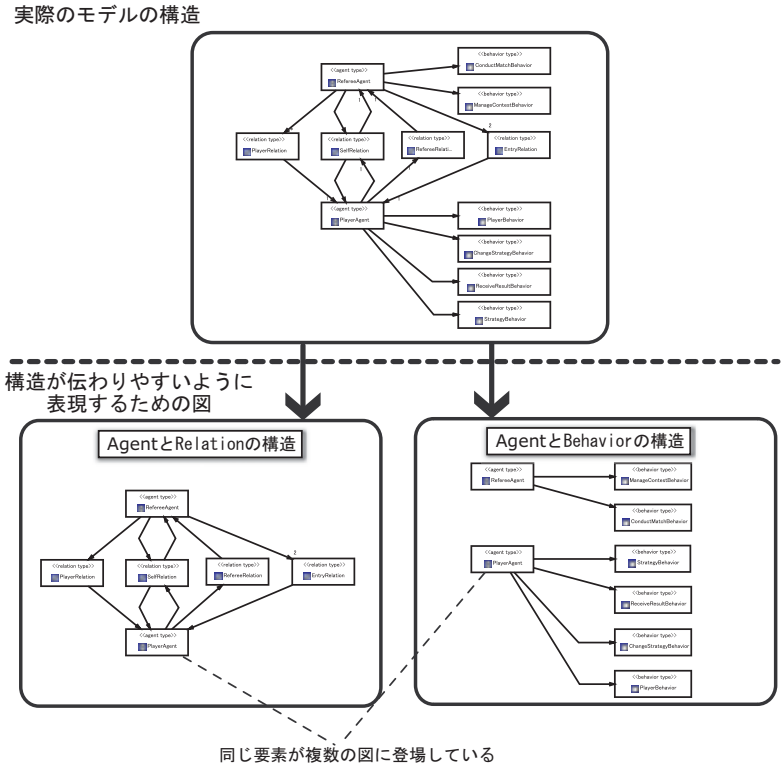


図 4.17: 構造を複数の図で表現する例

– 無限回の Undo/Redo

概念モデリングフェーズとシミュレーションデザインフェーズで記述するクラス図では，ひとつの構造をいくつかの視点から表現したい場合がある (図 4.17)．このような要求に対応するために，CB は，構造を複数の図で表現し，それぞれの図から同一の要素を参照する機能を持つ．

ソースコード生成の機能

CB は以下のソースコードを生成する機能を持つ．

- Type を定義するソースコード (モデル全体で 1 つの Java ファイル)
- Behavior の状態遷移を定義するソースコード (1Behavior につき 1 つの Java ファイル)
- Behavior のアクションを定義するソースコード (1Behavior につき 1 つの Java ファイル)

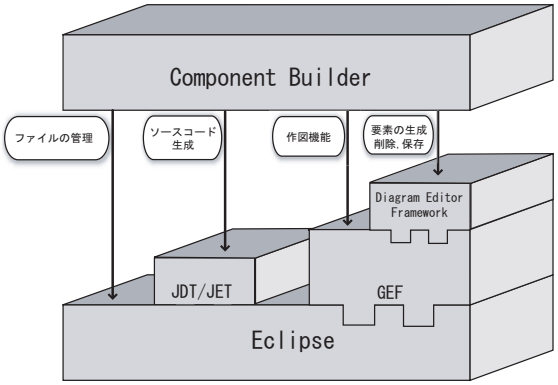


図 4.18: Component Builder のアーキテクチャ

- World の初期設定を定義するソースコード (1 つの初期設定につき 1 つの Java ファイル)

ファイル管理の機能

モデル作成を円滑に行うためには，ファイルやフォルダの作成，削除，移動などが容易にできる必要がある．さらに，モデルを複数人で作成したり，外部の研究者との共有するために，CB は以下のような機能を持つ．

- ネットワーク共有
- モデル全体のインポート/エクスポート

ドキュメンテーションのサポート

作成したモデルを，外部の研究者に公開するためには，作成した図を様々な形式でドキュメント化する必要がある．以下は CB が持つドキュメンテーションのための機能である．

- 図の印刷
- 図のファイルへの出力

4.3.2 Component Builder のアーキテクチャ

要求された機能を効率よく実装するために，CB は図 4.18 のようなアーキテクチャの上で実現されている．

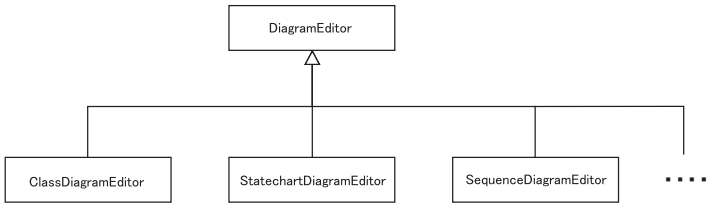


図 4.19: Diagram Editor Framework

Eclipse Platform

Eclipse(Eclipse Project, 2004) は，開発環境を構築するための汎用的なプラットフォームである．Eclipse は Eclipse Project がオープンソースで開発しており，開発環境を構築するための多彩な機能がプラットフォーム上で提供されている．CB が持つ以下のような機能は，Eclipse の機能を利用して実現されている．

- ファイル管理
- モデル全体のインポート/エクスポート
- ファイルのバージョン管理とネットワーク共有

GEF と Diagram Editor Framework

CB は作図の機能を実現するために作画ツール用の汎用的なフレームワーク「GEF」(Graphical Editor Framework)(GEF, 2004) を利用している．GEF も，Eclipse Project がオープンソースで開発しているフレームワークである．GEF はノードとコネクションを，GUI を使って操作していくための枠組みを提供している．GEF を利用すると，ノードやコネクションの種類とそれらに対する操作を定義するだけで，作成した図を印刷する，作図の際に行った操作を Undo/Redo する，といった基本的な機能を実現することができる．

CB を開発するにあたり，我々はそれぞれのツールに共通の部分を抽象化して，汎用的な図解ツール用のフレームワークを作成した．CB を構成するツールの中で，特に UML を記述するためのツールはクラス，状態など，それぞれに記述する要素が違うだけで,「要素をつくる」,「削除する」,「図を保存する」といった仕組みは共通している.「Diagram Editor Framework」は，これらの共通する機能を抽象化した図解ツール作成のためのフレームワークである．それぞれのツールは，このフレームワークに基づいて作成されている．このような仕組みを用意することで，複数のツールを効率よく実装することができた．

ソースコードの生成

Comoponent Builder はソースコード生成を行うために，以下の二つのフレームワークを利用している．これらはどちらも Eclipse プラットフォーム上で構築されたソースコード生成のための仕組みである．

- JDT(Java Developer Tool)(JDT, 2004)
- JET(Java Emitter Templates)(EMF, 2004)

JDT は，ソースコードを構文木として構築していくためのフレームワークである．CB は，Benavior の状態遷移，Behavior のアクション，World のそれぞれに対応するソースコードの生成にこれを利用している．それらを実現するためにこれを利用している理由は，JDT が JET に比べ自由度が高く，複雑なソースコードを生成するのに適しているためである．

JET は，JSP(Sun, 2004) のようなスクリプト形式のテンプレートから，SQL，XML，Java プログラムなどの構造化されたテキストを出力するための仕組みである．CB は Type の定義を行うソースコードの生成にこれを利用している．それらを実現するためにこれを利用している理由は，JET が JDT に比べ開発が容易で，形がほぼ決まっているソースコードを生成するのに適しているためである．

永続化の仕組み

CB を利用したモデル作成において，ファイルに保存するべき情報は以下の 2 種類である

- 図を表示するための座標の情報 (位置，大きさ)
- 図に表現されている要素の情報 (名前など)

ひとつの構造を複数の図から参照する機能を実現するために，CB は座標の情報をそれぞれの図に対応するファイルに保存し，要素の情報はモデル全体でひとつのファイルに保存している (図 4.20)．

それぞれのファイルは XML 形式で保存される．これは，XML 形式で保存をしておけば，XSLT などの技術を用いて図で記述したモデルを HTML やテキスト形式，CSV 形式などのドキュメントへの変換が可能になるためである．

画像ファイルへの出力形式

CB は，作成した図を画像ファイルに変換する形式として，SVG(W3C, 2004) という XML ベースの画像形式を採用している．保存した SVG ファイルは，市販のソフトウェアを使って EPS や JPEG などに変換することができる．なお，本論文中で使用している UML や ABL の図は全て，CB で作成した図を画像ファイルとして出力したものである．

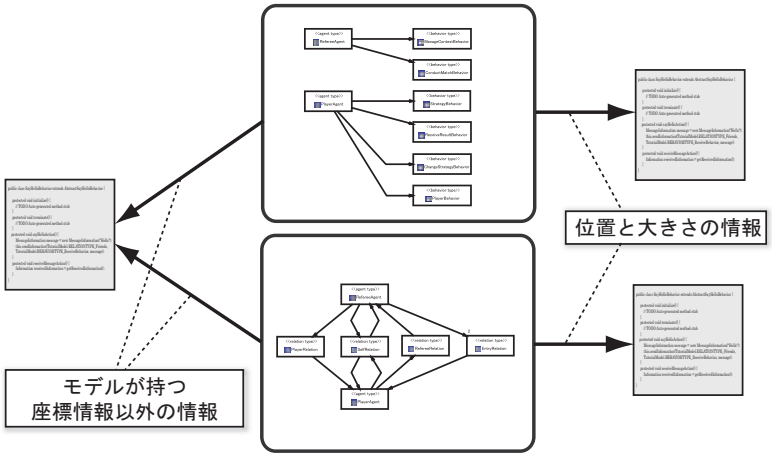


図 4.20: 情報を保存するファイルを分割する仕組み

第5章　評価と考察

本章では，既存モデルの再現と，試用実験の二つの面から提案するモデル作成環境の評価について述べる．既存モデルとしては「繰り返し囚人のジレンマモデル」を取り上げた．試用実験では，提案するモデル作成環境を利用して小規模なモデルの作成を行った．

5.1　既存モデルへの適用

5.1.1　繰り返し囚人のジレンマモデルの概要

「囚人のジレンマ」は，利害が対立する利己的な主体の間で，どのように協調関係が形成されていくのかを調べるための枠組みであり，政治学や経済学，社会学の分野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．このモデルでは，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，自分の行動 (手) と相手の行動 (手) の組み合わせによって，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 1 回限りのゲームであるが，これを繰り返し行うという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 (Axelrod, 1997)．本論文で再現したモデルは，繰り返しゲームを行う中で，Agent がより優れた戦略を模倣していくシミュレーションである (図 5.1)．シミュレーションの結果や考察など詳しくは，井庭 (2003) を参照してほしい．

5.1.2　Component Builder による繰り返し囚人のジレンマモデルの再現

概念モデリングフェーズのモデル

概念モデリングフェーズでは，対象領域からの概念の抽出，Agent のアクティビティの分析，Agent 間の相互作用の分析，を反復的に行いながら概念モデルを作成していく．

まず，Model Designer を使って対象となるモデルから Agent，Relation，Behavior，Goods[1]，Information を抽出する．

「繰り返し囚人のジレンマ」モデルでは,「PlayerAgent」(プレイヤー) と「RefereeAgent」(レフェリー) の 2 種類の Agent をモデル化する必要がある．その 2 種類の Agent の間には，

[1]「繰り返し囚人のジレンマ」モデルでは，Goods を使用する必要がなかった．しかし，他のモデルを作成する場合は，この段階で必要に応じて Goods もモデル化する．

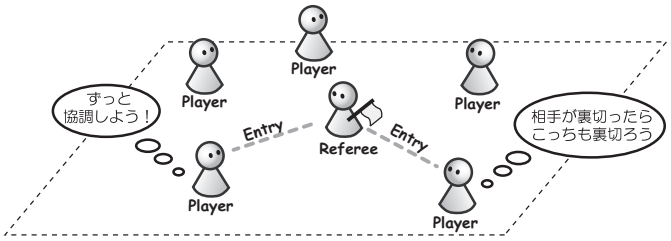


図 5.1: 「繰り返し囚人のジレンマ」モデルのイメージ

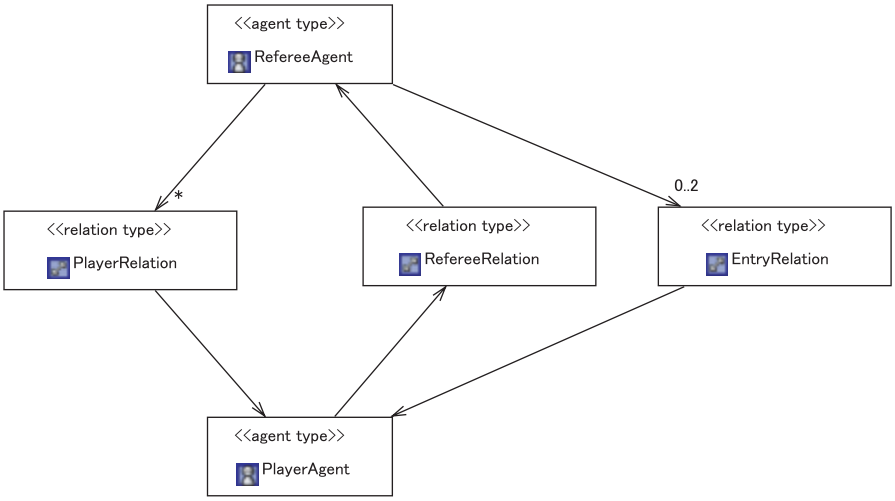


図 5.2: 「繰り返し囚人のジレンマ」概念モデル (全体の構造)

レフェリーからプレイヤーに向かう「PlayerRelation」(プレイヤーとの関係) と，プレイヤーからレフェリーに向かう「RefereeRelation」(レフェリーとの関係)，さらに，現在対戦中のプレイヤーをレフェリーが認識するために動的に引かれる「EntryRelation」(エントリー関係)，の 3 種類の Relation がある (図 5.2)．

レフェリーは，総当たり戦の全体 (コンテスト) を管理する「ManageContestBehavior」(コンテストを管理する行動) と，コンテストを構成する 1 対戦を管理する「ConductMatch-Behavior」(対戦を運営する行動) の 2 種類の行動を持つ．プレイヤーは,「PlayerBehavior」(ゲームを行う行動) と,「StrategyBehavior」(戦略行動),「ChangeStrategyBehavior」(戦略を変更する行動) の 3 種類の行動を持つ (図 5.3)．StrategyBehavior は抽象的な定義であり，具体的にはプレイヤーは全て C(協調) を出す戦略や，交互に C(協調) と D(裏切り) を繰り返す戦略など，具体的な戦略に基づく Behavior を持っている．ChangeStrategyBehavior も同様に，具体的な戦略変更のルールが具体的なサブクラスでモデル化される．

Agent が持つ Information は，レフェリーが持つ「MatchPairTableInformation」(対戦

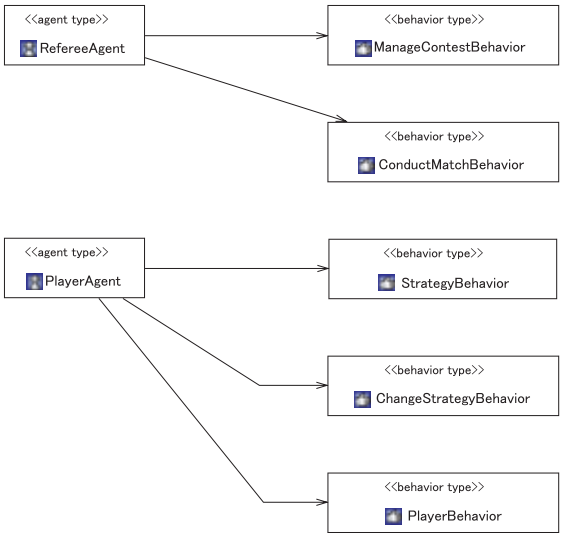


図 5.3: 「繰り返し囚人のジレンマ」概念モデル (Behavior の構造)

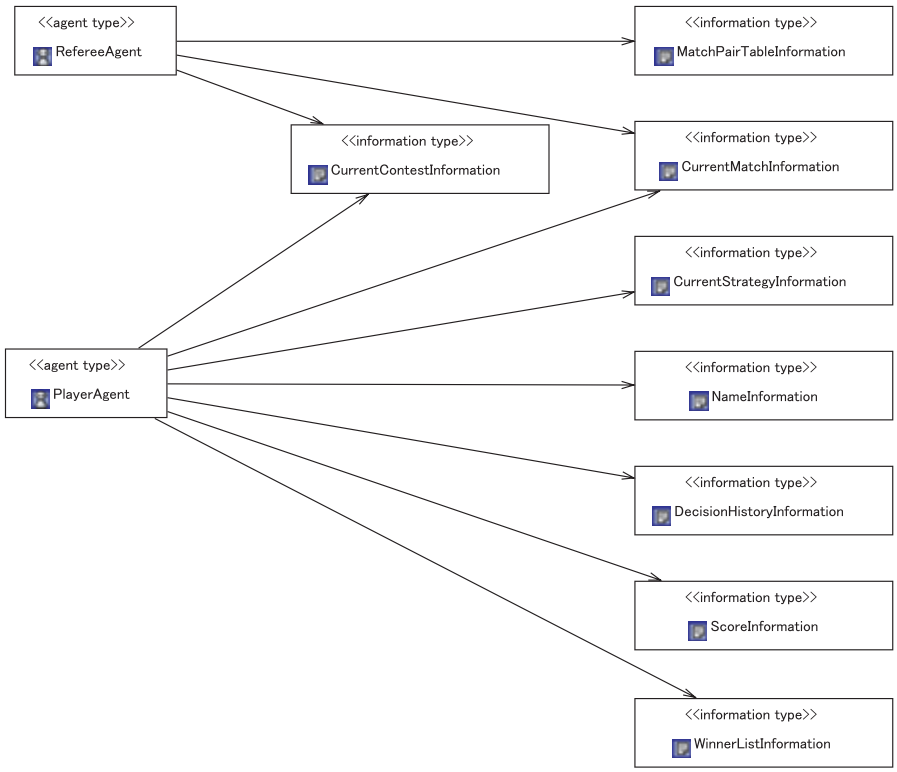


図 5.4: 「繰り返し囚人のジレンマ」概念モデル (Information の構造)

表の情報) や，レフェリーとプレイヤーの双方が持つ「CurrentContestInformation」(現在のコンテストの情報),「CurrentMatchInformation」(現在の対戦の情報)，プレイヤーの

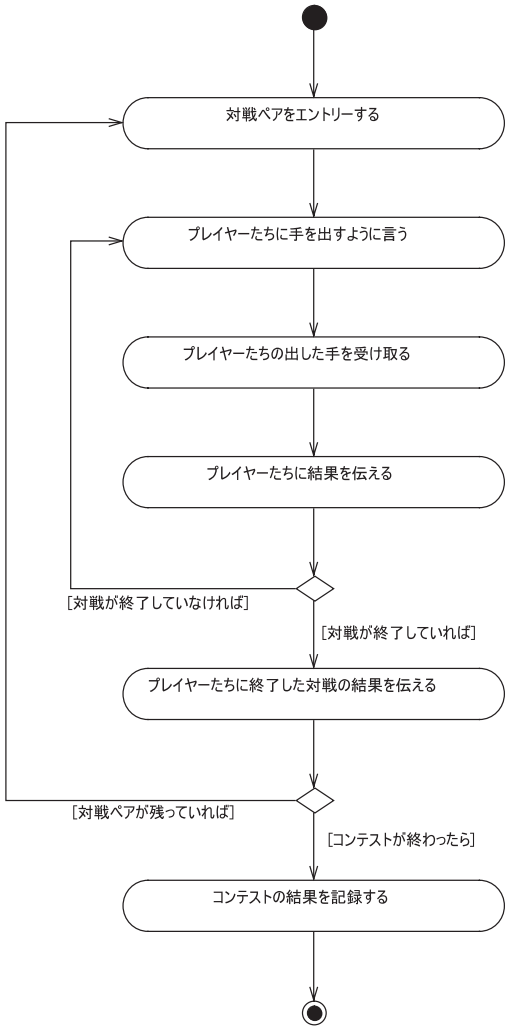


図 5.5: 「繰り返し囚人のジレンマ」概念モデル (レフェリーのアクティビティ)

みが持つ「ScoreInformation」(得点の情報) などがある (図 5.4).

これら以外にも，StrategyBehavior と ChangeStrategyBehavior のサブクラスとして定義される具体的な戦略や，Behavior 間でやりとりされる Information などについて作図を行う必要がある.

次に，Activity Designer を使って，それぞれの Agent のアクティビティを分析する．レフェリーのアクティビティは図 5.5 のようになる．レフェリーは，総当たり戦のペアの数だけ，対戦を繰り返し，1 つの対戦の中では双方のプレイヤーに手を出させて，ゲームの結果を知らせる，ということをゲーム回数分繰り返す．また，それぞれの対戦，コンテストが終わったときには，関係するプレイヤー全員にその結果を報告する．同じようにプレイヤーのアクティビティについても分析を行う必要がある.

次に，Communication Designer を使って，Agent 間の相互作用を分析する．相互作用

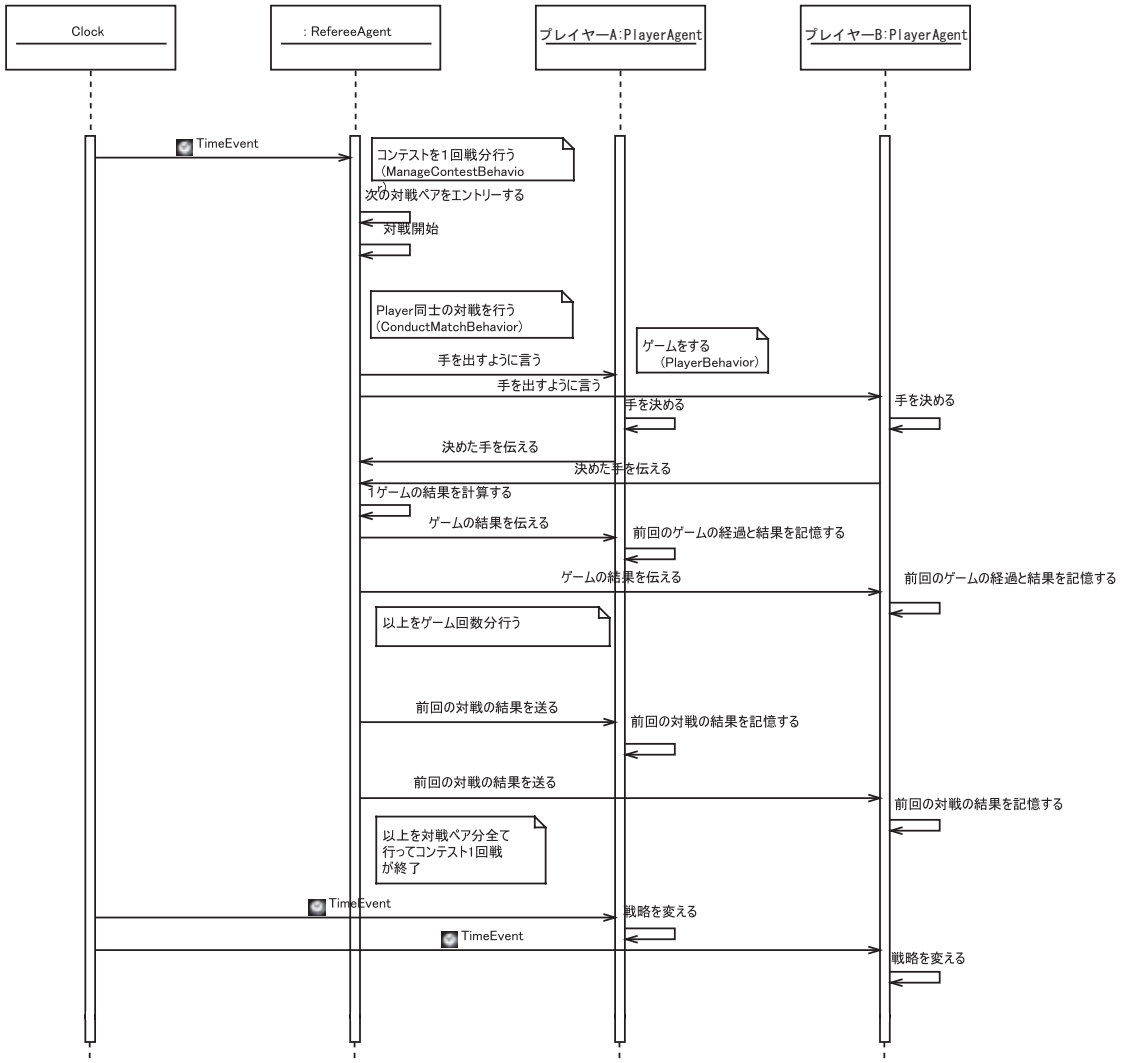


図 5.6: 「繰り返し囚人のジレンマ」概念モデル (Agent の相互作用)

のシーケンス図は図 5.6 のようになる．アクティビティ分析で分析したそれぞれの Agent のアクティビティが，時系列でどのように相互作用していくのかと，相互作用の際にどの Information がやりとりされているのかを，これで明らかにすることができる．また，アクティビティのどこからどこまでをシミュレーションの 1 ターンとするのかを決めるのも，この段階の重要なポイントである．

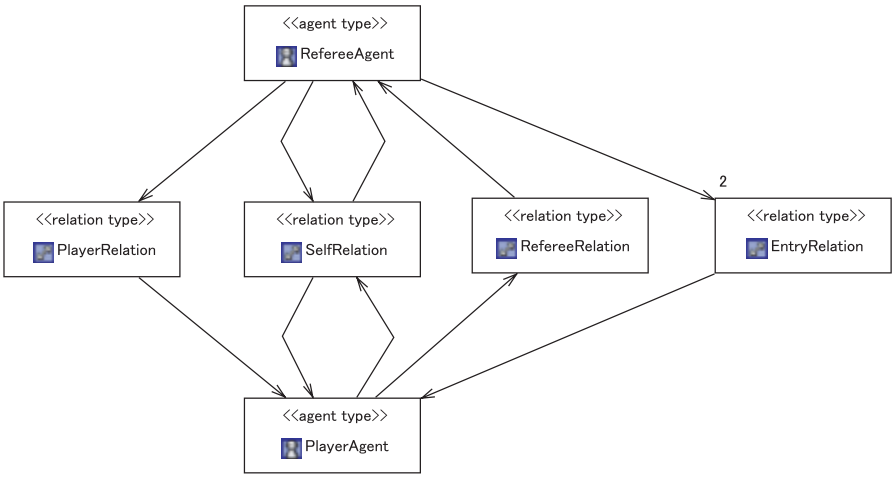


図 5.7: 「繰り返し囚人のジレンマ」設計モデル (全体の構造)

シミュレーションデザインフェーズのモデル

シミュレーションデザインフェーズでは，シミュレーションに登場する Type の設計，Behavior の状態遷移の設計，アクションの設計，を反復的に行いながら設計モデルを作成していく．

まず，概念モデリングフェーズで作成したモデルを基に Type と Information の実装クラス[2]を設計する．Agent と Relation の構造は，概念モデルとほとんど変わらないが，設計上必要となる，自分自身への Relation [3]を追加している (図 5.7)．Behavior の設計では，プレイヤーの行動の状態遷移が，複雑になってしまうのを避けるため,「ReceiveResultBehavior」(コンテストと対戦の結果を受け取る行動) を PlayerBehavior から分離している (図 5.8)．Information の設計では，概念モデルで分析した Information をどのような実装クラスを用意して実現するのかを記述する (図 5.9)．

[2]BEFM のシミュレーションモデルフレームワークに用意されていない複雑な Information の実装クラスについては，現在のところ CB は，Information が持つ属性と操作の雛形を作成する機能しか提供していない．操作の中身は Java でプログラムを記述する必要がある．この点は今後の検討課題である．

[3]自分自身への Relation はある Agent が持つ 2 種類の Behavior が相互作用したい場合に設計上必要となる．このような BEFM で頻出する設計手法は井庭 (2003); 岡部および井庭 (2002) にパターンとしてまとめられている．

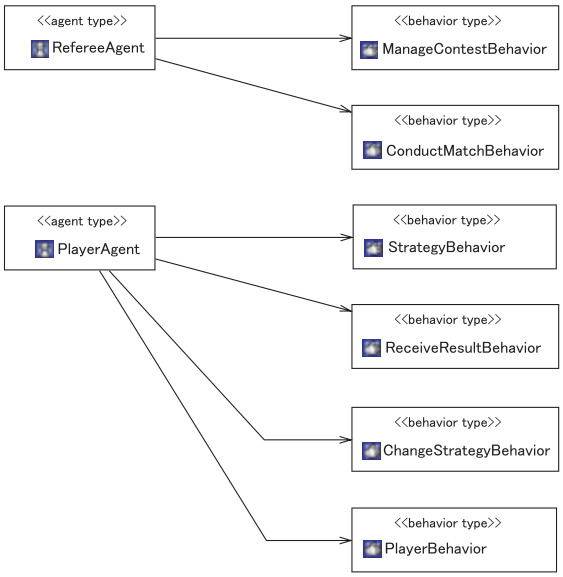


図 5.8: 「繰り返し囚人のジレンマ」設計モデル (Behavior の構造)

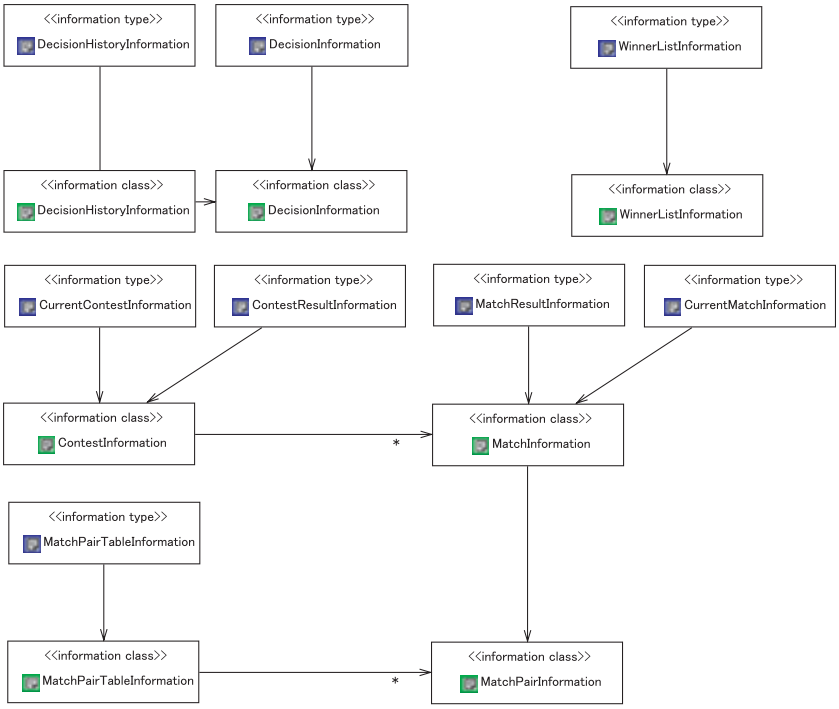


図 5.9: 「繰り返し囚人のジレンマ」設計モデル (Information の構造)

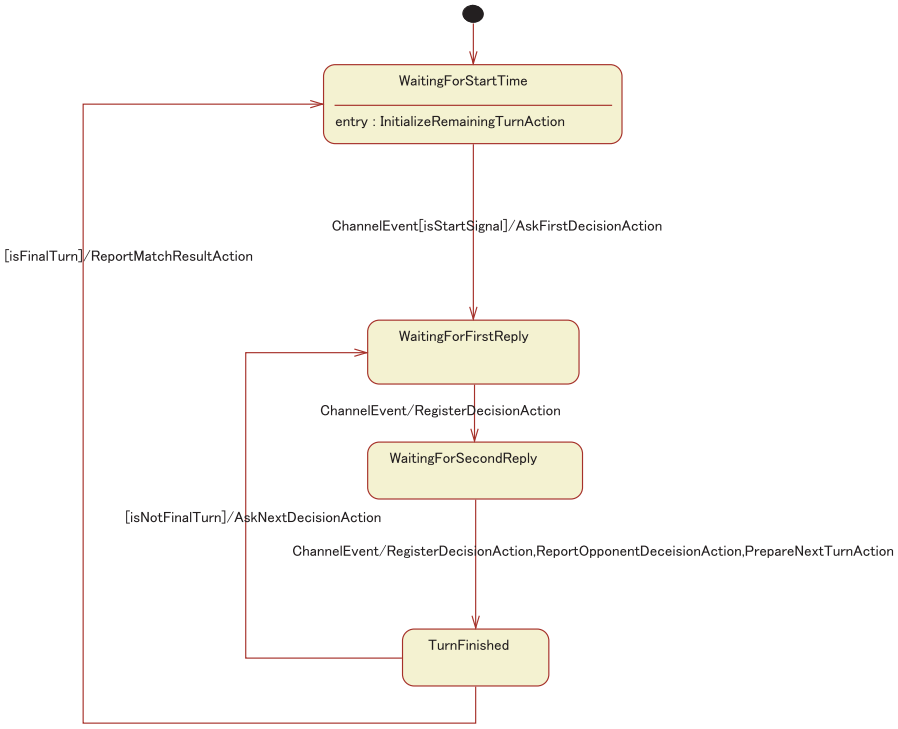


図 5.10: 「繰り返し囚人のジレンマ」設計モデル (レフェリーが対戦を管理する行動)

次に Behavior Designer を使って，Behavior の状態遷移を設計する．ConductMatchBehavior(レフェリーが対戦の運営をする行動) を例に挙げると，レフェリーがゲームを設定された回数分だけ繰り返して，1 回の対戦を運営する振る舞いのルールを，図 5.10 のようなステートチャート図で，モデル化することができる．これ以外にも，ManageContestBehavior，PlayerBehavior，ReceiveResultBehavior，ChangeStrategyBehavior のサブクラス，StrategyBehavior のサブクラスなどについても同様に，ステートチャート図による設計を行う必要がある．

次に Action Designer を使って，Behavior のアクションを設計する．ここで例に挙げているのは，ConductMatchBehavior の「ReportMatchResult」(対戦の結果を報告するアクション) と「AskNextDecision」(次の手をプレイヤーに尋ねるアクション) の二つである．

ReportMatchResult アクションでは，終了した対戦の情報を Agent(レフェリー) の記憶から取り出し，自分自身が持つ「ManageContestBehavior」(コンテストを管理する行動) と，プレイヤーの「PlayerBehavior」(ゲームを行う行動) に結果を送る (図 5.11)．AskNextDecision アクションでは，次の手を尋ねるためのメッセージを Information として作成し，プレイヤーの「PlayerBehavior」(ゲームを行う行動) にメッセージを送る (図 5.12)．

図 5.11: 「繰り返し囚人のジレンマ」設計モデル (レフェリーが対戦結果を通知するアクション)

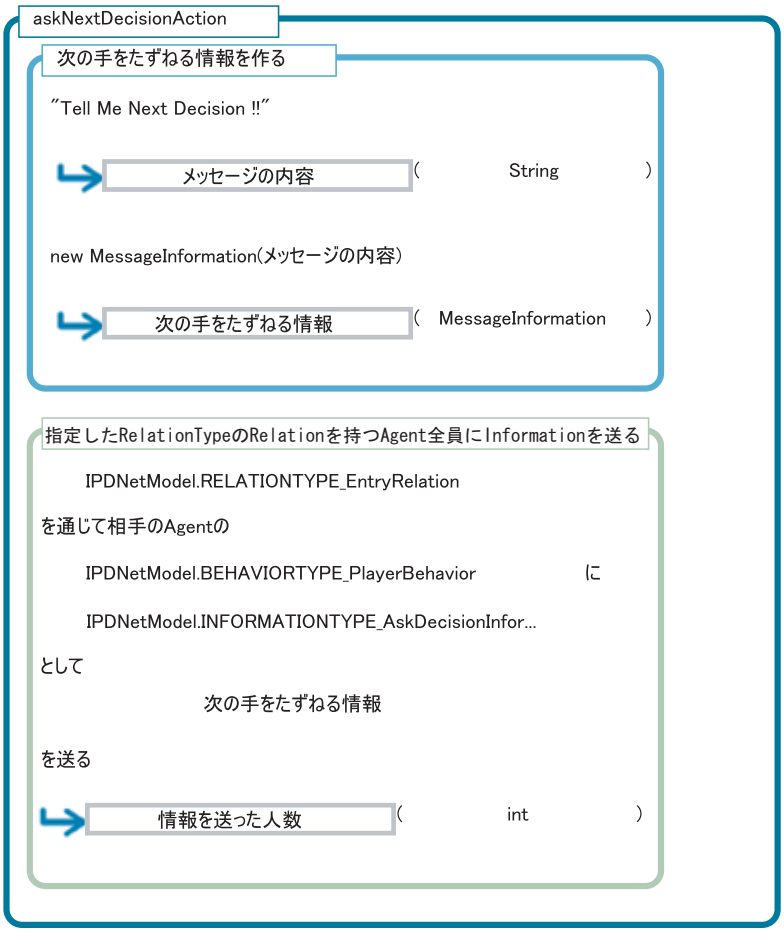


図 5.12: 「繰り返し囚人のジレンマ」設計モデル (レフェリーが次の手をプレイヤーに尋ねるアクション)

5.1.3 結果と考察

提案するツールを利用することで,「繰り返し囚人のジレンマ」モデルを再現した結果,モデルを構成する 6899 行のソースコードのうち，6184 行のソースコードを図から自動で生成することができた (表 5.1)．図の内訳は，クラス図 7 枚，ステートチャート図 15 枚，ABL によるアクション記述 15 枚である[4].

このように，提案するツールを利用することで、ほとんどソースコードを書かず，図解でモデルを作成することができた。なお、現在唯一，モデル作成者がソースコードを書かなければならないのは，複雑な Information の構造を記述する部分である。このことは今後の検討課題であるが、文字列や数値など単純な Information をやりとりするだけのモデルならば 1 行もソースコードを書かずにシミュレーションを作成することができる。

表 5.1: モデル作成に必要な記述の比較

	CB を使わないモデル作成	CB によるモデル作成
World	2318 行のソースコード	2 つの World 設定
Type の定義	245 行のソースコード	7 枚のクラス図
Behavior の状態遷移	1729 行のソースコード	15 枚のステートチャート図
Action の記述	1892 行のソースコード	15 枚の ABL によるアクション記述
Information の実装	715 行のソースコード	715 行のソースコード
合計	6899 行のソースコード	37 枚の図 2 つの World 設定 715 行のソースコード

[4] この再現実験の内容は Aoyama et al. (2004) ですでに報告されているものである.

5.2 試用実験

本節では，ABL によるアクション記述の有効性を実証するために行った試用実験の内容と結果について述べる．

5.2.1 前提となる被験者の能力

ABL によるアクション記述を行うためには，以下のような前提技能が求められる．

- 変数，制御構造などのプログラミングにおける基本的な概念の理解
- アクションの中で行いたい目的を，それを実現するための手段に分解して記述する論理的な思考能力
- BEFM の用語や構造の理解

このような前提を踏まえて，実験は一度以上プログラミングを学んだことがあり，BEFM の概略を理解している学生を対象として行った．しかし，実際には，一度授業などでプログラミングを学んだというだけでは，被験者によってプログラミングの概念の理解にばらつきがあり，それが試験の結果からもみてとれる．

論理的な思考能力については，被験者の能力が十分であるかを評価するのが難しく (松澤ほか, 2004)，また，その能力を身につけている被験者を探すのが困難であった．そのため，実験では，アクションの論理設計を記述した HCP チャート (花田, 1983) を配布して，論理的な設計はすでにできているものとした．

5.2.2 対象となるモデル

実験では，簡単な商取引をシミュレートしたサンプルモデルである「パン屋モデル」(Iba et al., 2004; Boxed Economy Project, 2004) [5]を被験者に作成してもらった．

パン屋のモデルには,「パン屋さん」と,「お客さん」が Agent として登場し，お客さんはパン屋さんに対して「お気に入りのお店」という Relation を持っている (図 5.13)．このサンプルモデルの詳細に関しては付録 B を参照してほしい．

5.2.3 実験の形式

実験では，まず，それぞれの学生のプログラミングの経験とモデル作成の経験を調べるための，事前アンケートを行った．次に,「パン屋」モデルのクラス図，ステートチャート図と，アクションの論理的な設計を記述した HCP チャートを配布した．被験者には，そ

[5] このモデルは，慶應義塾大学湘南藤沢キャンパスの授業「企業と市場のシミュレーション」でもサンプルとして使われているものである．

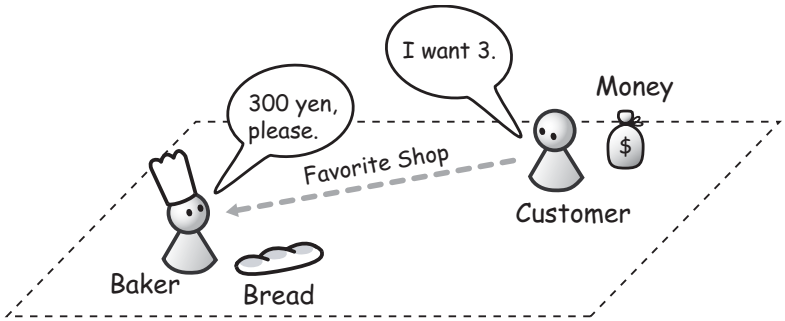


図 5.13: 対象となるモデルのイメージ

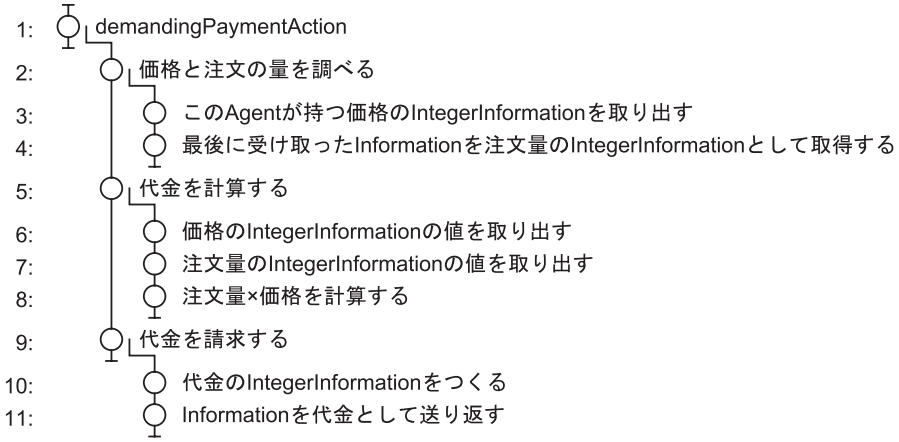


図 5.14: 配布した HCP チャート

れらをもとにアクションを記述してもらい，かかった時間を計測した．図 5.14 は配布した HCP チャートの一部である．配布したクラス図，ステートチャート図については付録 B を参照してほしい．さらに，Java による実装もできる学生には，Java による実装にも挑戦してもらい，かかった時間を計測した．最後に Action Designer を利用した感想を記述してもらった．

5.2.4 実験の結果と考察

実験の結果，被験者たちのプログラミング能力，モデル作成能力，Action Designer を利用したモデル作成にかかった時間，Java によるモデル作成にかかった時間について表 5.2 のような結果が得られた．この結果を履修者のプログラミング経験とモデル作成経験の多さで分類すると，表 5.3 のようになる．この表ではプログラミング関連の授業を 2 回以上受けたことがある被験者を「プログラミング経験が多い」とし，モデル作成を 2 回以上行ったことがある被験者を「モデル作成経験が多い」とし，被験者を 3 つのカテゴリに

分類した．なお，被験者の感想の詳細については付録 C を参照してほしい．

まず，この結果からプログラミング経験，モデル作成経験とも少なく，Java による実装ができない被験者でも，Action Designer を使えばアクションを記述できることがわかる．感想の中にも「苦手な Java の文法部分を理解できる」というような記述がみられた．

次に，Action Designer を使用していても論理的な思考能力，BEFM に対する理解，プログラミングの概念の理解は必要になるということが，試験の結果からあらためて明らかになった．プログラミング経験や，モデル作成の経験が少ない被験者は，それらが多い被験者に比べて作業にかかる時間が多かった．また，プログラミング経験，モデル作成の経験ともに少ない被験者の感想には,「困ったときに自分が何をしたかったのかがわからなくなるときがある」,「苦手な変数でつまづいた」,「アクションパーツを探すのに苦労した」というような記述が見られた．これは，Action Designer を使いこなすためには，アクションの中で行いたい処理の目的を，ABL に用意されているアクションパーツに分解していくための論理的な思考能力，アクションパーツを探すための BEFM に対する理解，変数,

表 5.2: 試用実験の結果

被験者	プログラミング関連授業の履修経験	Action Designer を使わないモデル作成経験	Action Designer によるモデル作成にかかった時間	Java によるモデル作成にかかった時間 (-は作成できず)
A	2 回	2 回	63 分	85 分
B	1 回	1 回	120 分	-
C	1 回	1 回	107 分	-
D	1 回	1 回	122 分	-
E	1 回	1 回	66 分	-
F	2 回	3 回	60 分	60 分
G	5 回	0 回	102 分	-
H	4 回	0 回	121 分	-
I	5 回	5 回	86 分	73 分

表 5.3: 試用実験の結果 (分類済み)

被験者の分類	Action Designer によるモデル作成にかかった平均時間	Java によるモデル作成にかかった平均時間 (-は作成できず)
プログラミング経験が多くモデル作成経験も多い (被験者 A，F，I)	70 分	72 分
プログラミング経験は多いがモデル作成経験は少ない (被験者 G，H)	111 分	-
プログラミング経験が少なくモデル作成経験も少ない (被験者 B，C，D，E)	103 分	-

制御構造，データ構造などのプログラムの概念の理解，が必要となるためだと思われる．

また，Action Designer が論理的な設計を助ける働きを持つことが被験者の感想からわかった．感想でも「これをするために，○○をする[6]というのがわかりやすい」,「プログラムの構造が可視化できるので助かる」,「頭の中で整理しやすかった」というような意見が得られた．

試験の結果，プログラミング，モデル作成ともに経験が多い被験者にも，プログラムの信頼性の点で Action Designer が有用であることがみてとれた．プログラミング，モデル作成の経験がともに多い被験者は，Action Designer を使う，使わないにかかわらず，作業時間はほとんど変わらなかった．しかし，彼らの感想の中には,「バグが減らせる」,「Reference がなくても不安なく書ける」というような意見があった．これは，アクションの設計からソースコードを生成することで，実装の段階でのケアレスミスをなくすことができるためだと考えられる．

被験者の感想から，Action Designer と ABL の課題も明らかになった．一番多かったのは「最初アクションパーツを探すのに手間取った」という意見である．これにはツールに不慣れであったり，BEFM の理解が曖昧であることも関係するが，ツールのインターフェイスや，アクションパーツの分類にも問題があると思われる．この点は今後の検討課題としたい．

[6]論理的な階層構造を作る, という意味だと思われる．

第6章　まとめ

本論文では，社会シミュレーションのためのモデルを，図解によって記述するモデル作成環境「Component Builder」(CB) を提案した．提案するモデル作成環境は，UML と，独自に定義したアクション記述言語「Action Block Language」(ABL) を用いて図に記述されたシミュレーションの設計を，実行可能なプログラムのソースコードに自動で変換する機能を提供する．このモデル作成環境の有効性を明らかにするため，本論文では既存モデルの再現と，作成環境の試用実験を行った．その結果，モデル作成者はソースコードを記述せずに，図解によってシミュレーションのためのモデルを作成できることが明らかになった．

本論文の試みは，社会にある複雑な現象を理解するために，より多くの人が，より有効にコンピュータ・シミュレーションを利用できるようにすることを目指すものである．図解でモデルを記述することで，特定のプログラミング言語の知識がなくても，モデルを作成したり，人が作ったモデルを理解することができる．また，抽象度の高い図解によってモデルを記述することで，モデル作成者は社会現象のモデル化に集中することができる．このことは，モデル化の過程における新たな発見を導くものである．

提案するモデル作成環境は，プログラミング教育にも利用することができる．論理的な思考能力を育てることはプログラミング教育の重要な目的のひとつである．また同様に，データ構造，アルゴリズム，オブジェクト指向など，プログラムにおける概念に対する理解もプログラミング教育の重要な目的である．評価の章でも述べたとおり，提案するモデル作成環境ではプログラミング言語の文法の知識は必要がないが，論理的な思考能力，プログラムの概念の理解は必要とされる．このことは逆に，それらの能力を身につけるために，提案する環境が有用であることを示唆している．それらの能力がないとモデルが作成できないために，その必要性はモデル作成者の心に強烈に残る．必要性を理解することは，何かを学ぶために非常に重要なことである．このようなことを踏まえて，今後この環境を利用したプログラミング教育の可能性についても模索していきたい．

本論文の成果は，シミュレーションのためのモデル作成を手助けしているに過ぎない．提案してきたモデル作成環境が真の意味で社会に貢献するのは，これを利用して，複雑な現象に対する新たな発見がなされたときである．そのためにも，今後，本論文で提案してきたモデル作成環境の普及に努めると共に，そのさらなる発展を目指していきたい．

謝辞

研究を遂行し修士論文としてまとめるにあたり，非常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，指導教官主査の大岩元先生には，学部の頃から様々な形でご指導いただきました．研究のためのソフトウェア開発，論文作成などで行き詰っているときに「そういえば大岩先生がおっしゃっていた」とあとから身にしみて感じるようなことが多々ありました．

副査の服部隆志先生には，研究会の場で本論文におけるソフトウェア開発を行う上で必要な，オブジェクト指向設計の基礎を教えていただきました．

本論文の内容の多くは Boxed Economy Project のメンバーとの共同研究の成果によるものです．プロジェクトのリーダーである井庭崇先生，メンバーの津屋隆之介さん，山田悠さんとはどのようなモデル作成環境が必要なのかということについて，長い時間をかけて一緒に議論していただきました．また，津屋さんには出来上がったソフトウェアを繰り返しテストいただき，Boxed Economy Project のメンバーではありませんが，井庭研のみなさんにも Component Builder のテストや評価のための実験に参加していただきました．

Component Builder の開発は，CreW Project のメンバーであり，Boxed Economy Project のメンバーでもある松澤芳昭さん，武田林太郎さんとともに 2002 年から繰り返し行われてきました．松澤さんには技術的な面以外にも考え方，仕事の仕方などいろいろな面でご指導いただきました．武田さんとは，特に Action Designer の開発において,「何を作るか」,「どうつくるか」について徹底的に議論し，共に開発を行ってきました．さらに，本論文をまとめるためには，土台となる BESP や BEFM を井庭先生や松澤さん，津屋さんと共に開発してきた中鉢欣秀さん，浅加浩太郎さん，海保研さんら Boxed Economy Project の OB の業績が不可欠でした．

修士課程での研究活動の中で，CreW Project の先輩，後輩である，齋藤俊則さん，岡田健さん，明石敬さんにはいろいろな形でご指導，ご支援をいただきました．また，同期の杉浦学さん，川村昌弘さんとは，学部の頃からほんとうにいろいろな研究活動を共にしてきました．これまで彼らと刺激しあってきたからこそどうにかがんばってこれました．一人だったらもっと手を抜いてしまっていたような気がします．

最後に，父 仁と母 恵子，祖母 歌子，妹 芽は研究活動をいろいろな面で支えてくれました．

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。

青山　希

参考文献

[Aoyama et al., 2004] N. Aoyama, R. Takeda, T. Iba, and H. Ohiwa (2004). Development Tools of Simulation Models with MDA in *International Workshop on Massively Multi-Agent Systems*.

[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.

[Axelrod and Cohen, 1999] R. Axelrod and M. D. Cohen (1999). *Harnessing Complexity.* ロバート・アクセルロッド, マイケル・D・コーエン,『複雑系組織論』, 高木 晴夫, 寺野 隆雄 (翻訳), ダイヤモンド社 , 2003.

[Beck, 2000] K. Beck (2000). *eXtreme Programming explained* , Addison-Wesley. ケントベック,『XP エクストリーム・プログラミング入門–ソフトウェア開発の究極の手法』, 飯塚 麻理香, 永田 渉 (訳), 長瀬 嘉秀 (監訳), ピアソンエデュケーション, 2000.

[Beizer, 2003] B. Beizer (2003). *Software Testing Techniques* , Van Nostrand Reinhold. ボーリスバイザー,『ソフトウェアテスト技法–自動化、品質保証、そしてバグの未然防止のために』, 小野間 彰, 山浦 恒央 (訳), 日経 BP 出版センター, 2003.

[Booch, 1996] G. Booch (1996). *OBJECT SOLUTIONS : Managing the Object-Oriented Project* , Addison-Wesley. グラディブーチ,『オブジェクトソリューション–オブジェクト指向プロジェクトの管理』, 石川 克己 (訳), ピアソンエデュケーション , 2002.

[Booch et al., 1999] G. Booch, J. Rumbaugh, and I. Jacobson (1999). *The Unified Modeling Language User Guide* , Addison-Wesley. グラディ・ブーチ,『UML ユーザーガイド』, オージス総研オブジェクト技術ソリューション事業部 (訳), ピアソンエデュケーション, 1999.

[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).

[Boxed Economy Project, 2004] Boxed Economy Project. 『社会シミュレーションデザイナーズガイド』. フジタ未来経営研究所 (2004).

[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, 水野貴之, 桜井麻里, 千葉寛之, 水野貴之, 関富登志 (訳), 近代科学社, 2000.

[Carlson, 2003] D. Carlson (2003). *Modeling XML Applications with UML Practical e-Business Applications* , Addison-Wesley. デビッドカールソン,『UML による XML アプリケーションモデリング–実践的 e - ビジネスアプリケーション』, 依田 光江 (訳), 依田 智夫 (監訳), ピアソンエデュケーション, 2002.

[Collier, 2003] N. Collier (2003). Repast: An extensible framework for agent simulation. *The University of Chicago's Social Science Research* , http://repast.sourceforge.net/.

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

[EMF, 2004] EMF. Java Emitter Templates. http://www.eclipse.org/emf/ (2004).

[Eclipse Project, 2004] Eclipse Project. Eclipse. http://www.eclipse.org/ (2004).

[Frankel, 2003] D. S. Frankel (2003). *Model Driven Architecture: Applying MDA to Enterprise Computing* , Wiley Publishing. 『MDA モデル駆動アーキテクチャ』, 日本アイ・ビー・エム TEC-J MDA 分科会 (訳), エスアイビーアクセス, 2003.

[GEF, 2004] GEF. Graphical Editing Framework. http://www.eclipse.org/gef/ (2004).

[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.

[花田, 1983] 花田 收悦 (1983). 『プログラム設計図法』 , 企画センター.

[原田, 1987] 原田 賢一 (1987). 『構造エディタ』 , 共立出版.

[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).

[広瀬ほか, 2002] 広瀬 通孝, 田村 善昭, 小木 哲朗 (2002). 『シミュレーションの思想』 , 東京大学出版会.

[井庭, 2003] 井庭 崇 (2003). Ph.D. Thesis, 慶應義塾大学政策・メディア研究科, 『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』.

[Iba, 2004] T. Iba (2004). A Framework and Tools for Modeling and Simulating Societies as Evolutionary Complex Systems. in *2nd. International Conference of the European Social Simulation Association.*

[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, 2003] 井庭 崇, 中鉢 欣秀, 松澤 芳昭, 海保 研, 武藤 佳恭 (2003). Boxed economy foundation model: 社会・経済のエージェントベースモデリングのためのフレームワーク. **情報処理学会論文誌*: 数理モデル化と応用*** .

[Iba et al., 2004] T. Iba, Y. Matsuzawa, and N. Aoyama (2004). From Conceptual Models to Simulation Models: Model Driven Development of Agent-Based Simulations. in *9th Workshop on Economics and Heterogeneous Interacting Agents.*

[JDT, 2004] JDT. Java development tools. http://www.eclipse.org/jdt/ (2004).

[Jacobson and Jacobson, 1995] I. Jacobson and S. Jacobson (1995). Beyond Methods and CASE: The software engineering process with its integral support environment in *Object Magazine*, January.

[Jacobson and Jacobson, 1995] I. Jacobson and S. Jacobson (1995). Designing an integrated SEPSE in *Object Magazine*, September.

[Jacobson et al., 2000] I. Jacobson, J. Rumbaugh, and G. Booch (2000). *The Unified Software Development Process* , Addison-Wesley. イヴァー ヤコブソン, ジェームズ ランボー, グラディ ブーチ, 『UML による統一ソフトウェア開発プロセス–オブジェクト指向開発方法論』, 日本ラショナルソフトウェア (訳), 藤井 拓 (監訳), 翔泳社, 2000.

[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.

[Kleppe et al., 2003] A. Kleppe, J. Warmer, and W. Bast (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise* , Addison-Wesley. 『MDA(モデル駆動型アーキテクチャ) 導入ガイド—UML を基盤としたオブジェクト指向設計・開発手法』, テクノロジックアート (訳), 長瀬 嘉秀 (監訳), インプレス, 2003.

[児玉, 2004] 児玉 公信 (2004). 『UML モデリングの本質』 , 日経 BP 社.

[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).

[Object Management Group, 2003] Object Management Group. UML2.0 OCL Specifications. http://www.omg.org/cgi-bin/doc?ptc/2003-10-14 (2003).

[Object Management Group, 2004] Object Management Group. UML2.0 Specifications. http://www.omg.org/technology/documents/ (2004).

[松澤ほか, 2004] 松澤 芳昭, 杉浦 学, 大岩 元 (2004). プログラム設計教育における HCP チャートのレビュー手法. **情報処理学会第** *66* **回全国大会** *1C*(2-4).

[Mellor and Balcer, 2002] S. J. Mellor and M. J. Balcer (2002). *Executable UML: A Foundation for Model-Driven Architecture* , Addison-Wesley. スティーブ J. メラー, マーク J. パルサー, 『Executable UML』, 二上貴夫, 長瀬 嘉秀 (監訳), 翔泳社, 2003.

[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.

[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.

[岡部および井庭, 2002] 岡部 明子, 井庭 崇 (2002). 社会・経済シミュレーションのモデル・パターン：複雑系における動的な変化を記述する , 『第 2 回情報科学技術フォーラム Forum on Information Technology (FIT2003)』.

[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.

[Project Technology, 2002] Project Technology. Bridge Point Action Language Manual. http://www.projtech.com/pdfs/bp/oal.pdf (2002).

[Rumbaugh, 1992] J. Rumbaugh (1992). *Object-Oriented Modeling and Design* , Prentice Hall. J. ランボー, M. ブラハ,W. プレメラニ,F. エディ,W. ローレンセン, 『オブジェクト指向方法論 OMT -モデル化と設計』, 羽生田栄一 (監訳), トッパン, 1992.

[Perdita Stevens and Rob Pooley, 2000] Perdita Stevens and Rob Pooley (2000). *Using UML: Software Engineering with Objects and Components* , Addison-Wesley. ペルディタ スティーブンス, ロブ プーリー, 『オブジェクト指向とコンポーネントによるソフトウェア工学–UML を使って』, 児玉公信 (監訳), ピアソンエデュケーション, 2000.

[Sun, 2004] Sun. JavaServer Pages. http://java.sun.com/products/jsp/ (2004).

[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』.

[テクノロジックおよび長瀬, 2004] テクノロジック アート, 長瀬 嘉秀 (2004). 『UML2 ハンドブック』 , 翔泳社.

[W3C, 2004] W3C. Scalable Vector Graphics. http://www.w3.org/Graphics/SVG/ (2004).

[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.

[山影および服部, 2002] 山影 進, 服部 正太 (2002). 『コンピュータの中の人工社会: マルチエージェントシミュレーションモデルと複雑系』 , 共立出版.

付 録A　Action Block Language(ABL) に用意されているアクションパーツの詳細

ここでは Action Block Language(ABL) に語彙として用意されているアクションパーツの詳細な定義を紹介する．ABL の文法や，アクションパーツの一覧表については本文中，第 3 章の第 3.3.3 節を参照してほしい．

アクションパーツの定義は全て以下のような形式で記述されている．

アクションパーツの名前

- 引数 (引数がないアクションパーツにはこの項目はない)
  1 番目の引数の型　1 番目の引数の意味
  2 番目の引数の型　2 番目の引数の意味
- 対応するソースコード (CB が生成する BESP 上で実行可能な Java 言語のソースコード)

```
....
```

A.1　カテゴリ「自分自身の Agent」

この Agent が指定した Type の Behavior を持っているか調べる

- 引数
  BehaviorType　調べる対象となる BehaviorType
- 対応するソースコード

```
boolean 指定した Type の Behavior を持っているか
    = getAgent().getBehaviors([BehaviorType]).isEmpty();
```

この Agent が指定した Type の Goods を持っているか調べる

- 引数
  GoodsType　調べる対象となる GoodsType
- 対応するソースコード

```
boolean 指定した Type の Goods を持っているか
    = getAgent().hasGoods([GoodsType]);
```

この Agent が指定した Type の Information を持っているか調べる

- 引数
  InformationType　調べる対象となる InformationType
- 対応するソースコード

```
boolean 指定した Type の Information を持っているか
    = getAgent().hasInformation([InformationType]);
```

この Agent が指定した Type の Relation を持っているか調べる

- 引数
  RelationType　調べる対象となる RelationType
- 対応するソースコード

```
boolean 指定した Type の Relation を持っているか
    = getAgent().getRelations([RelationType]).isEmpty();
```

この Agent の Type を取得する

- 対応するソースコード

```
AgentType 取得した AgentType = getAgent().getType();
```

この Agent を消滅させる

- 対応するソースコード

```
getAgent().destroy();
```

自分自身の Agent を取得する

- 対応するソースコード

```
Agent 自分自身の Agent = getAgent();
```

A.2　カテゴリ「自分が持つ Behavior」

この Agent が持つ Behavior を削除する

- 引数
  Behavior　削除する Behavior
- 対応するソースコード

```
getAgent().removeBehavior([Behavior]);
```

この Agent が持つ Behavior を取得する

- 引数
  BehaviorType　取得する Behavior の Type
- 対応するソースコード

```
Behavior 取得した Behavior = getAgent().getBehavior([BehaviorType]);
```

この Agent が持つ Behavior を親 Type を指定して再帰的に取得する

- 引数
  BehaviorType　取得する Behavior の Type
- 対応するソースコード

```
Collection 取得した Behavior の集合 = getAgent().getBehaviorsRecursively([BehaviorType]);
```

この Agent が持つ Behavior を全て取得する

- 対応するソースコード

```
Collection 取得した Behavior の集合 = getAgent().getAllBehaviors();
```

この Agent が持つ指定した Type の Behavior を全て取得する

- 引数
  BehaviorType　取得する Behavior の Type
- 対応するソースコード

```
Collection 取得した Behavior の集合 = getAgent().getBehaviors([BehaviorType]);
```

この Agent に Behavior を追加する

- 引数
  BehaviorType　追加する Behavior の Type
- 対応するソースコード

```
getAgent().addBehavior([BehaviorType]);
```

この Behavior の Type を取得する

- 対応するソースコード

```
BehaviorType 取得した BehaviorType = getType();
```

この Behavior の現在の状態名を取得する

- 対応するソースコード

```
String 現在の状態の名前 = getState().getName();
```

A.3　カテゴリ「自分が持つ Information」

この Agent が持つ DoubleInformation の値を減らす

- 引数
  InformationType　値を減らす Information の Type
  double　減らす量
- 対応するソースコード

```
InformationType 更新する Information の Type = [InformationType];
DoubleInformation 更新前の DoubleInformation
  = (DoubleInformation)getAgent().getInformation(更新する Information の Type);

//DoubleInformation の値を減らす
DoubleInformation 更新された DoubleInformation
  = new DoubleInformation(更新前の DoubleInformation.getValue() - [double]);
getAgent().putInformation(更新する Information の Type, 更新された DoubleInformation);
```

この Agent が持つ DoubleInformation の値を設定する

- 引数
  InformationType　値を設定する Information の Type
  double　設定する値
- 対応するソースコード

```
DoubleInformation 更新された DoubleInformation = new DoubleInformation([double]);
getAgent().putInformation([InformationType], 更新された DoubleInformation);
```

この Agent が持つ DoubleInformation の値を増やす

- 引数
  InformationType　値を増やす Information の Type
  double　増やす量
- 対応するソースコード

```
InformationType 更新する Information の Type = [InformationType];
DoubleInformation 更新前の DoubleInformation
  = (DoubleInformation)getAgent().getInformation(更新する Information の Type);

//DoubleInformation の値を増やす
DoubleInformation 更新された DoubleInformation
  = new DoubleInformation(更新前の DoubleInformation.getValue() + [double]);
getAgent().putInformation(更新する Information の Type, 更新された DoubleInformation);
```

この Agent が持つ DoubleInformation を取得する

- 引数
  InformationType　取得する Information の Type
- 対応するソースコード

```
DoubleInformation 取得した DoubleInformation
  = (DoubleInformation)getAgent().getInformation([InformationType]);
```

この Agent が持つ Information を削除する

- 引数
  InformationType　削除する Information の Type
- 対応するソースコード

```
Information 削除した Information = getAgent().removeInformation([InformationType]);
```

この Agent が持つ Information を取得する

- 引数
  InformationType　取得する Information の Type
- 対応するソースコード

```
Information 取得した Information = getAgent().getInformation([InformationType]);
```

この Agent が持つ Information を全て取得する

- 対応するソースコード

```
Map 取得した Information の Map = getAgent().getInformations();
```

この Agent が持つ IntegerInformation の値を減らす

- 引数
  InformationType　値を減らす Information の Type
  int　減らす量
- 対応するソースコード

```
InformationType 更新する Information の Type = [InformationType];
IntegerInformation 更新前の IntegerInformation
  = (IntegerInformation)getAgent().getInformation(更新する Information の Type);

//IntegerInformation の値を減らす
IntegerInformation 更新された IntegerInformation
  = new IntegerInformation(更新前の IntegerInformation.getValue() - [double]);
getAgent().putInformation(更新する Information の Type, 更新された IntegerInformation);
```

この Agent が持つ IntegerInformation の値を設定する

- 引数
  InformationType　値を設定する Information の Type
  int　設定する値
- 対応するソースコード

```
IntegerInformation 更新された IntegerInformation = new IntegerInformation([int]);
getAgent().putInformation([InformationType], 更新された IntegerInformation);
```

この Agent が持つ IntegerInformation の値を増やす

- 引数
  InformationType　値を増やす Information の Type
  int　増やす量
- 対応するソースコード

```
InformationType 更新する Information の Type = [InformationType];
IntegerInformation 更新前の IntegerInformation
  = (IntegerInformation)getAgent().getInformation(更新する Information の Type);

//IntegerInformation の値を増やす
IntegerInformation 更新された IntegerInformation
  = new IntegerInformation(更新前の IntegerInformation.getValue() + [double]);
getAgent().putInformation(更新する Information の Type, 更新された IntegerInformation);
```

この Agent が持つ IntegerInformation を取得する

- 引数
  InformationType　取得する Information の Type
- 対応するソースコード

```
IntegerInformation 取得した IntegerInformation
    = (IntegerInformation)getAgent().getInformation([InformationType]);
```

この Agent に Information を記憶させる

- 引数
  InformationType　記憶のキーにする InformationType
  Information　記憶する Information
- 対応するソースコード

```
getAgent().putInformation([InformationType],[Information]);
```

A.4　カテゴリ「自分が持つ Goods」

この Agent が持つ Goods の Type を全て取得する

- 対応するソースコード

```
Collection この Agent が持つ Goods の Type の集合 = getAgent().getGoodsTypes();
```

この Agent が持つ Goods の量の合計を親 Type を指定して取得する

- 引数
  GoodsType　量を取得する Goods の親 Type
- 対応するソースコード

```
GoodsQuantity Goods の量 = getAgent().getQuantityRecursively([GoodsType]);
```

この Agent が持つ Goods の量を Type を指定して取得する

- 引数
  GoodsType　量を取得する Goods の Type
- 対応するソースコード

```
GoodsQuantity Goods の量 = getAgent().getQuantity([GoodsType]);
```

この Agent が持つ Goods を指定した量だけ取り出す

- 引数
  GoodsType　取り出す Goods の Type
  double　取り出す量
- 対応するソースコード

```
Goods 取り出した Goods = getAgent().removeGoods([GoodsType],[double]);
```

この Agent が持つ Goods を親 Type と量を指定して取り出す

- 引数
  GoodsType　取り出す Goods の親 Type
  double　　取り出す量
- 対応するソースコード

```
Collection 取り出した Goods の集合
  = getAgent().removeGoodsRecursively([GoodsType],[double]);
```

この Agent が持つ Goods を親 Type を指定して全て取り出す

- 引数
  GoodsType　取り出す Goods の親 Type
- 対応するソースコード

```
Collection 取り出した Goods の集合 = getAgent().removeAllGoodsRecursively([GoodsType]);
```

この Agent が持つ指定した Type の Goods を全て取り出す

- 引数
  GoodsType　取り出す Goods の Type
- 対応するソースコード

```
Goods 取り出した Goods = getAgent().removeAllGoods([GoodsType]);
```

この Agent に Goods を持たせる

- 引数
  Goods　この Agent に持たせる Goods
- 対応するソースコード

```
getAgent().addGoods([Goods]);
```

A.5　カテゴリ「自分と他の Agent の間の Relation」

この Agent から他のエージェントへ Relation をむすぶ

- 引数
  RelationType　結ぶ Relation の Type
  Agent　結ぶ相手の Agent
- 対応するソースコード

```
getAgent().addRelation([RelationType],[Agent]);
```

この Agent が持つ Relation の Type を全て取得する

- 対応するソースコード

```
Collection 取得した RelationType の集合 = getAgent().getRelationTypes();
```

この Agent が持つ Relation を Type を指定して全て取得する

- 引数
  RelationType　取得する Relation の Type
- 対応するソースコード

```
Collection 取得した Relation の集合 = getAgent().getRelations([RelationType]);
```

この Agent が持つ Relation を削除する

- 引数
  Relation　削除する Relation
- 対応するソースコード

```
getAgent().removeRelation([Relation]);
```

この Agent が持つ Relation を取得する

- 引数
  RelationType　取得する Relation の Type
- 対応するソースコード

```
Relation 取得した Relation = getAgent().getRelation([RelationType]);
```

この Agent が持つ Relation を親 Type を指定して全て削除する

- 引数
  RelationType　削除する Relation の親 Type
- 対応するソースコード

```
getAgent().removeRelationsRecursively([RelationType]);
```

この Agent が持つ Relation を親 Type を指定して全て取得する

- 引数
  RelationType　取得する Relation の親 Type
- 対応するソースコード

```
Collection 取得した Relation の集合 = getAgent().getRelationsRecursively([RelationType]);
```

この Agent が持つ Relation を相手の Agent を指定して取得する

- 引数
  RelationType　取得する Relation の Type
  Agent　相手の Agent
- 対応するソースコード

```
Relation 取得した Relation = getAgent().getRelation([RelationType],[Agent]);
```

この Agent が持つ指定した Type の Relation を全て削除する

- 引数
  RelationType　削除する Relation の Type
- 対応するソースコード

```
getAgent().removeRelations([RelationType]);
```

この Agent と他の Agent の間で双方向の Relation をむすぶ

- 引数
  RelationType　結ぶ Relation の Type
  Agent　結ぶ相手の Agent
- 対応するソースコード

```
結んだ Relation の Type = [RelationType];
関係先の Agent = [Agent];

//双方向に結ぶ
getAgent().addRelation(結んだ Relation の Type, 関係先の Agent);
関係先の Agent.addRelation(結んだ Relation の Type,getAgent());
```

この Agent と他人の間の双方向の Relation を削除する

- 引数
  Agent　Relation を削除する相手の Agent
  RelationType　削除する Relation の Type
- 対応するソースコード

```
相手の Agent = [Agent];
削除した Relation の Type = [RelationType];

//自分からの Relation を削除する
自分からの Relation = getAgent().getRelation(削除した Relation の Type,target);
getAgent().removeRelation(自分からの Relation);

//相手からの Relation を削除する
相手からの Relation = target.getRelation(削除した Relation の Type,getAgent());
target.removeRelation(相手からの Relation);
```

この Agent の Relation を全て取得する

- 対応するソースコード

```
Collection 取得した Relation の集合 = getAgent().getAllRelations();
```

現在アクティブな Channel を開いている関係を取得する

- 対応するソースコード

```
Relation 現在アクティブな Channel を開いている関係 = getActiveChannel().getParentRelation();
```

他の Agent からこの Agent へ Relation をむすぶ

- 引数
  Agent　　結ぶ相手の Agent
  RelationType　結ぶ Relation の Type
- 対応するソースコード

```
[Agent].addRelation([RelationType],getAgent());
```

他の Agent から他の Agent へ Relation をむすぶ

- 引数
  AgentA　　関係元の Agent
  RelationType　結ぶ Relation の Type
  AgentB　　関係先の Agent
- 対応するソースコード

```
[AgentA].addRelation([RelationType],[AgentB]);
```

他の Agent が持つ Relation の Type を全て取得する

- 引数
  Agent　対象となる Agent
- 対応するソースコード

```
Collection 取得した RelationType の集合 = [Agent].getRelationTypes();
```

他の Agent が持つ Relation を Type を指定して全て取得する

- 引数
  Agent　　対象となる Agent
  RelationType　取得する Relation の Type
- 対応するソースコード

```
Collection 取得した Relation の集合 = [Agent].getRelations([RelationType]);
```

他の Agent が持つ Relation を削除する

- 引数
  Agent　対象となる Agent
  Relation　削除する Relation
- 対応するソースコード

```
[Agent].removeRelation([Relation]);
```

他の Agent が持つ Relation を取得する

- 引数
  Agent　対象となる Agent
  RelationType　取得する Relation の Type
- 対応するソースコード

```
Collection 取得した Relation = [Agent].getRelation([RelationType]);
```

他の Agent が持つ Relation を親 Type を指定して全て削除する

- 引数
  Agent　対象となる Agent
  RelationType　削除する Relation の Type
- 対応するソースコード

```
[Agent].removeRelationsRecursively([RelationType]);
```

他の Agent が持つ Relation を親 Type を指定して全て取得する

- 引数
  Agent　対象となる Agent
  RelationType　取得する Relation の親 Type
- 対応するソースコード

```
Collection 取得した Relation の集合 = [Agent].getRelationsRecursively([RelationType]);
```

他の Agent が持つ Relation を相手の Agent を指定して取得する

- 引数
  AgentA　　対象となる Agent
  RelationType　取得する Relation の Type
  AgentB　　相手の Agent
- 対応するソースコード

```
Relation 取得した Relation = [AgentA].getRelation([RelationType],[AgentB]);
```

他の Agent が持つ指定した Type の Relation を全て削除する

- 引数
  Agent　　対象となる Agent
  RelationType　削除する Relation の Type
- 対応するソースコード

```
[Agent].removeRelations([RelationType]);
```

他の Agent の Relation を全て取得する

- 引数
  Agent　対象となる Agent
- 対応するソースコード

```
Collection 取得した Relation の集合 = [Agent].getAllRelations();
```

他人と他人の間で双方向の Relation をむすぶ

- 引数
  RelationType　結ぶ Relation の Type
  AgentA　　Relation を結ぶ一方の Agent
  AgentB　　Relation を結ぶ他方の Agent
- 対応するソースコード

```
結んだ Relation の Type = [RelationType];
一方の Agent = [AgentA];
他方の Agent = [AgentB];

//Relation を双方向に結ぶ
一方の Agent.addRelation(結んだ Relation の Type, 他方の Agent);
他方の Agent.addRelation(結んだ Relation の Type, 一方の Agent);
```

他人と他人の間の双方向の Relation を削除する

- 引数
  - **AgentA** Relation を持つ一方の Agent
  - **AgentB** Relation を持つ他方の Agent
  - **RelationType** 削除する Relation の Type
- 対応するソースコード

```
一方の Agent = [AgentA];
他方の Agent = [AgentB];
削除した Relation の Type = [RelationType];

//一方からの Relation を削除する
一方からの Relation = 一方の Agent.getRelation(削除した Relation の Type, 他方の Agent);
一方の Agent.removeRelation(一方からの Relation);

//他方からの Relation を削除する
他方からの Relation = 他方の Agent.getRelation(削除した Relation の Type, 一方の Agent);
他方の Agent.removeRelation(他方からの Relation);
```

A.6　カテゴリ「自分と他の Agent の間のやりとり」

Goods や Information を送ってきた相手にメッセージを送る

- 引数
  String　メッセージの内容
- 対応するソースコード

```
返事の Information = new MessageInformation([String]);
sendInformation(返事の Information);
```

Goods を送り返す

- 引数
  Goods　送り返す Goods
- 対応するソースコード

```
sendGoods([Goods]);
```

Information を送り返す

- 引数
  InformationType　送り返すときにキーとなる InformationType
  Information　送り返す Information
- 対応するソースコード

```
sendInformation([InformationType],[Information]);
```

RelationType と一人一人に送る量を指定して全員に Goods を送る

- 引数
  RelationType　送るために使う Relation の Type
  BehaviorType　送り先の Behavior の Type
  GoodsType　送りたい Goods の Type
  double　送りたい Goods の量
- 対応するソースコード

```
sendGoods([RelationType],[BehaviorType],[GoodsType],[double],false);
```

RelationType と一人一人に送る量を指定して全員に Goods を送る (Channel はキープする)

- 引数
  - **RelationType** 送るために使う Relation の Type
  - **BehaviorType** 送り先の Behavior の Type
  - **GoodsType** 送りたい Goods の Type
  - **double** 送りたい Goods の量
- 対応するソースコード

```
sendGoods([RelationType],[BehaviorType],[GoodsType],[double],false,true);
```

RelationType を指定してメッセージを送る

- 引数
  - **String** メッセージの内容
  - **RelationType** 送るために使う Relation の Type
  - **BehaviorType** 送り先の Behavior の Type
- 対応するソースコード

```
メッセージの Information = new MessageInformation([String]);
sendInformation([RelationType],[BehaviorType],メッセージの Information);
```

Relation を指定して Goods の量が N 以上だったら Goods を N だけ送る

- 引数
  - **GoodsType** 送りたい Goods の Type
  - **double** 送りたい Goods の量
  - **Relation** 送るために使う Relation
  - **BehaviorType** 送り先の Behavior の Type
- 対応するソースコード

```
送信する Goods の Type = [GoodsType];
送信する量 = [double];

//Goods を送る
if(getAgent.getQuantity(送信する Goods の Type).getValueAsDouble >= 送信する量){

  送信する Goods = getAgent().removeGoods(送信する Goods の Type, 送信する量);
   sendGoods([Relation],[BehaviorType],送信する Goods,false);
}
```

Relation を指定して Goods の量が N 以上だったら Goods を N だけ送る（Channel はキープする）

- 引数
  - **GoodsType**　送りたい Goods の Type
  - **double**　送りたい Goods の量
  - **Relation**　送るために使う Relation
  - **BehaviorType**　送り先の Behavior の Type
- 対応するソースコード

```
送信する Goods の Type = [GoodsType];
送信する量 = [double];

//Goods を送る
if(getAgent.getQuantity(送信する Goods の Type).getValueAsDouble >= 送信する量){

  送信する Goods = getAgent().removeGoods(送信する Goods の Type, 送信する量);
   sendGoods([Relation],[BehaviorType], 送信する Goods,true);
}
```

Relation を指定して Goods を送る

- 引数
  - **Relation**　送るために使う Relation
  - **BehaviorType**　送り先の Behavior の Type
  - **Goods**　送る Goods
- 対応するソースコード

```
sendGoods([Relation],[BehaviorType],[Goods],false);
```

Relation を指定して Goods を送る（Channel はキープする）

- 引数
  - **Relation**　送るために使う Relation
  - **BehaviorType**　送り先の Behavior の Type
  - **Goods**　送る Goods
- 対応するソースコード

```
sendGoods([Relation],[BehaviorType],[Goods],true);
```

Relation を指定して一人に Information を送る

- 引数
  - **Relation** 送るために使う Relation
  - **BehaviorType** 送り先の Behavior の Type
  - **InformationType** 送るときのキーとする InformationType
  - **Information** 送る Information
- 対応するソースコード

```
sendInformation([Relation],[BehaviorType],[InformationType],[Information],false);
```

Relation を指定して一人に Information を送る（Channel はキープする）

- 引数
  - **Relation** 送るために使う Relation
  - **BehaviorType** 送り先の Behavior の Type
  - **InformationType** 送るときのキーとする InformationType
  - **Information** 送る Information
- 対応するソースコード

```
sendInformation([Relation],[BehaviorType],[InformationType],[Information],true);
```

現在アクティブな Channel をつないだ元の Behavior を取得する

- 対応するソースコード

```
Behavior Channel をつないだ元の Behavior = getActiveChannel().getBehaviorA();
```

現在アクティブな Channel をつないだ先の Behavior を取得する

- 対応するソースコード

```
Behavior Channel をつないだ先の Behavior = getActiveChannel().getBehaviorB();
```

現在アクティブな Channel を取得する

- 対応するソースコード

```
Channel アクティブな Channel = getActiveChannel();
```

現在アクティブな Channel を閉じる

- 対応するソースコード

```
getActiveChannel().close();
```

最後に受け取った Goods が指定された Type のものか調べる

- 引数
  GoodsType　判定対象の GoodsType
- 対応するソースコード

```
boolean 受け取った Goods が指定された Type のものかどうか = receivedGoodsEquals([GoodsType]);
```

最後に受け取った Goods を持つ

- 対応するソースコード

```
Goods 受け取った Goods = getReceivedGoods();
getAgent().addGoods(受け取った Goods);
```

最後に受け取った Goods を取得する

- 対応するソースコード

```
Goods 最後に受け取った Goods = getReceivedGoods();
```

最後に受け取った Information が指定された Type のものか調べる

- 引数
  InformationType　判定対象の InformationType
- 対応するソースコード

```
boolean 受け取った Information が指定された Type のものかどうか
  = receivedInformationEquals([InformationType]);
```

最後に受け取った Information を DoubleInformation として取得する

- 対応するソースコード

```
DoubleInformation 最後に受け取った Information
  = (DoubleInformation)getReceivedInformation();
```

最後に受け取った Information を IntegerInformation として取得する

- 対応するソースコード

```
IntegerInformation 最後に受け取った Information
  = (IntegerInformation)getReceivedInformation();
```

最後に受け取った Information を記憶する

- 引数
  InformationType　記憶するときにキーとする InformationType
- 対応するソースコード

```
Information 受け取った Information = getReceivedInformation();
getAgent().putInformation([InformationType],受け取った Information);
```

最後に受け取った Information を取得する

- 対応するソースコード

```
Information 最後に受け取った Information = getReceivedInformation();
```

指定した RelationType の Relation を持つ Agent 全員に Information を送る

- 引数
  RelationType　送るために使う Relation の Type
  BehaviorType　送り先の Behavior の Type
  InformationType　送るときのキーとする InformationType
  Information　送る Information
- 対応するソースコード

```
情報を送った人数
  = sendInformation(
      [RelationType],[BehaviorType],[InformationType],[Information],false);
```

指定した RelationType の Relation を持つ Agent 全員に Information を送る (Channel はキープする)

- 引数
  - **RelationType** 送るために使う Relation の Type
  - **BehaviorType** 送り先の Behavior の Type
  - **InformationType** 送るときのキーとする InformationType
  - **Information** 送る Information
- 対応するソースコード

```
情報を送った人数
  = sendInformation(
      [RelationType],[BehaviorType],[InformationType],[Information],true);
```

指定した RelationType の Relation 一つを取り出し Goods を送る

- 引数
  - **RelationType** 送るために使う Relation の Type
  - **BehaviorType** 送り先の Behavior の Type
  - **Goods** 送る Goods
- 対応するソースコード

```
Relation Goods の送信に使った Relation = getAgent().getRelation([RelationType]);
sendGoods(relation,[BehaviorType],[Goods],false);
```

指定した RelationType の Relation 一つを取り出し Information を送る

- 引数
  - **RelationType** 送るために使う Relation の Type
  - **BehaviorType** 送り先の Behavior の Type
  - **InformationType** 送るときのキーとする InformationType
  - **Information** 送る Information
- 対応するソースコード

```
Relation Information の送信に使った Relation = getAgent().getRelation([RelationType]);
sendInformation(relation,[BehaviorType],[InformationType],[Information],false);
```

指定した RelationType の Relation 一つを取り出し Information を送る（Channel はキープする）

- 引数

RelationType	送るために使う Relation の Type
BehaviorType	送り先の Behavior の Type
InformationType	送るときのキーとする InformationType
Information	送る Information

- 対応するソースコード

```
Relation Information の送信に使った Relation = getAgent().getRelation([RelationType]);
sendInformation(relation,[BehaviorType],[InformationType],[Information],true);
```

全ての Channel を取得する

- 対応するソースコード

```
List アクティブな Channel のリスト = getAllChannels();
```

A.7　カテゴリ「他の Agent」

新しい Agent を作る

- 引数
  AgentType　新たに生成する Agent の Type
- 対応するソースコード

```
Agent 新しい Agent = getWorld().createAgent([AgentType]);
```

他の Agent が指定した Type の Behavior を持っているか調べる

- 引数
  Agent　　対象となる Agent
  BehaviorType　対象となる BehaviorType
- 対応するソースコード

```
boolean 指定した Type の Behavior を持っているか
  = [Agent].getBehaviors([BehaviorType]).isEmpty();
```

他の Agent が指定した Type の Goods を持っているか調べる

- 引数
  Agent　　対象となる Agent
  GoodsType　対象となる GoodsType
- 対応するソースコード

```
boolean 指定した Type の Goods を持っているか = [Agent].hasGoods([GoodsType]);
```

他の Agent が指定した Type の Information を持っているか調べる

- 引数
  Agent　　対象となる Agent
  InformationType　対象となる InformationType
- 対応するソースコード

```
boolean 指定した Type の Information を持っているか
  = [Agent].hasInformation([InformationType]);
```

他の Agent が指定した Type の Relation を持っているか調べる

- 引数
  Agent　　対象となる Agent
  RelationType　対象となる RelationType
- 対応するソースコード

```
boolean 指定した Type の Relation を持っているか
  = [Agent].getRelations([RelationType]).isEmpty();
```

他の Agent の Type を取得する

- 引数
  Agent　対象となる Agent
- 対応するソースコード

```
AgentType 取得した AgentType = [Agent].getType();
```

他の Agent を消滅させる

- 引数
  Agent　対象となる Agent
- 対応するソースコード

```
[Agent].destroy();
```

A.8　カテゴリ「他の Agent が持つ Behavior」

他の Agent が持つ Behavior を削除する

- 引数
  Agent　　対象となる Agent
  Behavior　削除する Behavior
- 対応するソースコード

```
[Agent].removeBehavior([Behavior]);
```

他の Agent が持つ Behavior を取得する

- 引数
  Agent　　対象となる Agent
  BehaviorType　取得する Behavior の Type
- 対応するソースコード

```
Behavior 取得した Behavior = [Agent].getBehavior([BehaviorType]);
```

他の Agent が持つ Behavior を親 Type を指定して再帰的に取得する

- 引数
  Agent　　対象となる Agent
  BehaviorType　取得する Behavior の親 Type
- 対応するソースコード

```
Collection 取得した Behavior の集合 = [Agent].getBehaviorsRecursively([BehaviorType]);
```

他の Agent が持つ Behavior を全て取得する

- 引数
  Agent　対象となる Agent
- 対応するソースコード

```
Collection 取得した Behavior の集合 = [Agent].getAllBehaviors();
```

他の Agent が持つ指定した Type の Behavior を全て取得する

- 引数
  Agent　　対象となる Agent
  BehaviorType　取得する Behavior の Type
- 対応するソースコード

```
Collection 取得した Behavior の集合 = [Agent].getBehaviors([BehaviorType]);
```

他の Agent に Behavior を追加する

- 引数
  Agent　　対象となる Agent
  BehaviorType　追加する Behavior の Type
- 対応するソースコード

```
[Agent].addBehavior([BehaviorType]);
```

A.9　カテゴリ「他の Agent が持つ Information」

他の Agent が持つ DoubleInformation の値を減らす

- 引数

InformationType	減らす Information の Type
Agent	対象となる Agent
double	減らす量

- 対応するソースコード

```
Information 更新する Information の Type = [InformationType];
DoubleInformation 更新前の DoubleInformation
  = (DoubleInformation) [Agent].getInformation(更新する Information の Type);

//DoubleInformation の値を減らす
DoubleInformation 更新された DoubleInformation
  = new DoubleInformation (更新前の DoubleInformation.getValue() - [double]);
[Agent].putInformation(更新する Information の Type, 更新する Information の Type);
```

他の Agent が持つ DoubleInformation の値を設定する

- 引数

double	設定する値
Agent	対象となる Agent
InformationType	設定する Information の Type

- 対応するソースコード

```
更新された DoubleInformation = new DoubleInformation([double]);
[Agent].putInformation([InformationType], 更新された DoubleInformation);
```

他の Agent が持つ DoubleInformation の値を増やす

- 引数

InformationType	増やす Information の Type
Agent	対象となる Agent
double	増やす量

- 対応するソースコード

```
更新する Information の Type = [InformationType];
DoubleInformation 更新前の DoubleInformation
  = (DoubleInformation) [Agent].getInformation(更新する Information の Type);

//DoubleInformation の値を増やす
DoubleInformation 更新された DoubleInformation
  = new DoubleInformation(更新前の DoubleInformation.getValue() + [double]);
[Agent].putInformation(更新する Information の Type, 更新された DoubleInformation);
```

他の Agent が持つ DoubleInformation を取得する

- 引数
  Agent　　対象となる Agent
  InformationType　取得する Information の Type
- 対応するソースコード

```
DoubleInformation 取得した DoubleInformation
  = (DoubleInformation)[Agent].getInformation([InformationType]);
```

他の Agent が持つ Information を削除する

- 引数
  Agent　　対象となる Agent
  InformationType　削除する Information の Type
- 対応するソースコード

```
Information 削除した Information = [Agent].removeInformation([InformationType]);
```

他の Agent が持つ Information を取得する

- 引数
  Agent　　対象となる Agent
  InformationType　取得する Information の Type
- 対応するソースコード

```
Information 取得した Information = [Agent].getInformation([InformationType]);
```

他の Agent が持つ Information を全て取得する

- 引数
  Agent　対象となる Agent
- 対応するソースコード

```
Map 取得した Information の Map = [Agent].getInformations();
```

他の Agent が持つ IntegerInformation の値を減らす

- 引数
  - **InformationType** 減らす Information の Type
  - **Agent** 対象となる Agent
  - **int** 減らす量
- 対応するソースコード

```
InformationType 更新する Information の Type = [InformationType];
IntegerInformation 更新前の IntegerInformation
  = (IntegerInformation)[Agent].getInformation(更新する Information の Type);

IntegerInformation 更新された IntegerInformation
  = new IntegerInformation(更新前の IntegerInformation.getValue() - [int]);
[Agent].putInformation(更新する Information の Type, 更新された IntegerInformation);
```

他の Agent が持つ IntegerInformation の値を設定する

- 引数
  - **int** 設定する値
  - **Agent** 対象となる Agent
  - **InformationType** 設定する Information の Type
- 対応するソースコード

```
IntegerInformation 更新された IntegerInformation = new IntegerInformation([int]);
[Agent].putInformation([InformationType], 更新された IntegerInformation);
```

他の Agent が持つ IntegerInformation の値を増やす

- 引数
  - **InformationType** 増やす Information の Type
  - **Agent** 対象となる Agent
  - **int** 増やす量
- 対応するソースコード

```
InformationType 更新する Information の Type = [InformationType];
IntegerInformation 更新前の IntegerInformation
  = [Agent].getInformation(更新する Information の Type);

IntegerInformation 更新された IntegerInformation
  = new IntegerInformation(更新前の IntegerInformation.getValue() + [int]);
[Agent].putInformation(更新する Information の Type, 更新された IntegerInformation);
```

他の Agent が持つ IntegerInformation を取得する

- 引数
  Agent　　対象となる Agent
  InformationType　　取得する Information の Type
- 対応するソースコード

```
IntegerInformation 取得した IntegerInformation
  = (IntegerInformation)[Agent].getInformation([InformationType]);
```

他の Agent に Information を記憶させる

- 引数
  Agent　　対象となる Agent
  InformationType　　記憶するときのキーとする InformationType
  Information　　記憶する Information
- 対応するソースコード

```
[Agent].putInformation([InformationType],[Information]);
```

A.10　カテゴリ「他の Agent が持つ Goods」

他の Agent が持つ Goods の Type を全て取得する

- 引数
  Agent　対象となる Agent
- 対応するソースコード

```
Collection 他の Agent が持つ Goods の Type の集合 = [Agent].getGoodsTypes();
```

他の Agent が持つ Goods の量の合計を親 Type を指定して取得する

- 引数
  Agent　対象となる Agent
  GoodsType　量を取得する Goods の親 Type
- 対応するソースコード

```
GoodsQuantitiy Goods の量 = [Agent].getQuantityRecursively([GoodsType]);
```

他の Agent が持つ Goods の量を Type を指定して取得する

- 引数
  Agent　対象となる Agent
  GoodsType　量を取得する Goods の Type
- 対応するソースコード

```
GoodsQuantity Goods の量 = [Agent].getQuantity([GoodsType]);
```

他の Agent が持つ Goods を指定した量だけ取り出す

- 引数
  Agent　対象となる Agent
  GoodsType　取り出す Goods の Type
  double　取り出す量
- 対応するソースコード

```
Goods 取り出した Goods = [Agent].removeGoods([GoodsType],[double]);
```

他の Agent が持つ Goods を親 Type と量を指定して取り出す

- 引数
  Agent　　対象となる Agent
  GoodsType　取り出す Goods の親 Type
  double　　取り出す量
- 対応するソースコード

```
Collection 取り出した Goods の集合 = [Agent].removeGoodsRecursively([GoodsType],[double]);
```

他の Agent が持つ Goods を親 Type を指定して全て取り出す

- 引数
  Agent　　対象となる Agent
  GoodsType　取り出す Goods の親 Type
- 対応するソースコード

```
Collection 取り出した Goods の集合 = [Agent].removeAllGoodsRecursively([GoodsType]);
```

他の Agent が持つ指定した Type の Goods を全て取り出す

- 引数
  Agent　　対象となる Agent
  GoodsType　取り出す Goods の Type
- 対応するソースコード

```
Goods 取り出した Goods = [Agent].removeAllGoods([GoodsType]);
```

他の Agent に Goods を持たせる

- 引数
  Agent　対象となる Agent
  Goods　持たせる Goods
- 対応するソースコード

```
[Agent].addGoods([Goods]);
```

A.11　カテゴリ「Goodsの生成/操作」

Goods に Information を付与する

- 引数
  Goods　対象となる Goods
  InformationType　付与するときのキーとなる InformationType
  Information　付与する Information
- 対応するソースコード

```
[Goods].putInformation([InformationType],[Information]);
```

Goods に指定された Type の Information が付随しているか調べる

- 引数
  Goods　対象となる Goods
  InformationType　対象となる InformationType
- 対応するソースコード

```
boolean Goods に指定された Type の Information が付随しているか
  = [Goods].hasInformation([InformationType]);
```

Goods に付随する Information を取得する

- 引数
  Goods　対象となる Goods
  InformationType　取得する Information の Type
- 対応するソースコード

```
Information 取得した Information = [Goods].getInformation([InformationType]);
```

Goods に付随する Information を全て取得する

- 引数
  Goods　対象となる Goods
- 対応するソースコード

```
Map 取得した Information のマップ = [Goods].getInformations();
```

Goods に付属する Information を削除する

- 引数
  Goods　　対象となる Goods
  InformationType　削除する Information の Type
- 対応するソースコード

```
Information 削除した Information = [Goods].removeInformation([InformationType]);
```

Goods の Type を取得する

- 引数
  Goods　対象となる Goods
- 対応するソースコード

```
GoodsType 取得した Goods の Type = [Goods].getType();
```

Goods の量を取得する

- 引数
  Goods　対象となる Goods
- 対応するソースコード

```
GoodsQuantity 取得した Goods の量 = [Goods].getGoodsQuantity();
```

Goods の量を小数のオブジェクトとして取得する

- 引数
  Goods　対象となる Goods
- 対応するソースコード

```
Double 取得した Goods の量
  = [Goods].getGoodsQuantity().getValueAsDoubleObject();
```

Goods の量を小数の値として取得する

- 引数
  Goods　対象となる Goods
- 対応するソースコード

```
double 取得した Goods の量
  = [Goods].getGoodsQuantity().getValueAsDouble();
```

Goods の量を整数のオブジェクトとして取得する

- 引数
  Goods　対象となる Goods
- 対応するソースコード

```
Integer  取得した Goods の量
  = [Goods].getGoodsQuantity().getValueAsIntegerObject();
```

Goods の量を整数の値として取得する

- 引数
  Goods　対象となる Goods
- 対応するソースコード

```
int 取得した Goods の量
  = [Goods].getGoodsQuantity().getValueAsInt();
```

Goods を消費する

- 引数
  Goods　対象となる Goods
- 対応するソースコード

```
getWorld().consumeGoods([Goods]);
```

新しい Goods を作る

- 引数
  GoodsType　新しく生成する Goods の Type
  double　生成する量
- 対応するソースコード

```
Goods 新しい Goods = getWorld().createGoods([GoodsType],[double]);
```

A.12　カテゴリ「Information の生成/操作」

DoubleInformation の値を取得する

- 引数
  DoubleInformation　対象となる DoubleInformation
- 対応するソースコード

```
double 取り出した少数値 = [DoubleInformation].getValue();
```

DoubleInformation をつくる

- 引数
  double　新しい DoubleInformation の値
- 対応するソースコード

```
DoubleInformation 新しい DoubleInformation = new DoubleInformation([double]);
```

IntegerInformation の値を取得する

- 引数
  IntegerInformation　対象となる IntegerInformation
- 対応するソースコード

```
int 取り出した整数値 = [IntegerInformation].getValue();
```

IntegerInformation をつくる

- 引数
  int　新しい IntegerInformation の値
- 対応するソースコード

```
IntegerInformation 新しい IntegerInformation = new IntegerInformation([int]);
```

A.13　カテゴリ「Relation の操作」

Relation の Type を取得する

- 引数
  Relation　対象となる Relation
- 対応するソースコード

```
RelationType Relation の Type = [Relation].getType();
```

Relation の関係元を取得する

- 引数
  Relation　対象となる Relation
- 対応するソースコード

```
Agent 関係元の Agent = [Relation].getSource();
```

Relation の関係先を取得する

- 引数
  Relation　対象となる Relation
- 対応するソースコード

```
Agent 関係先の Agent = [Relation].getTarget();
```

A.14　カテゴリ「World の取得/操作」

World が持つ Clock を取得する

- 対応するソースコード

```
Clock 取得した Clock = getWorld().getClock();
```

World が持つ Space を取得する

- 対応するソースコード

```
Space 取得した Space = getWorld().getSpace();
```

World に存在する全ての Agent を取得する

- 対応するソースコード

```
Collection 取得した Agent の集合 = getWorld().getAllAgents();
```

World に定義された AgentType を取得する

- 引数
  String　取得する AgentType の名前
- 対応するソースコード

```
AgentType 取得した AgentType = getWorld().getAgentType([String]);
```

World に定義された BehaviorType を取得する

- 引数
  String　取得する BehaviorType の名前
- 対応するソースコード

```
BehaviorType 取得した BehaviorType = getWorld().getBehaviorType([String]);
```

World に定義された GoodsType を取得する

- 引数
  String　取得する GoodsType の名前
- 対応するソースコード

```
GoodsType 取得した GoodsType = getWorld().getGoodsType([String]);
```

World に定義された InformationType を取得する

- 引数
  String　取得する InformationType の名前
- 対応するソースコード

```
InformationType 取得した InformationType = getWorld().getInformationType([String]);
```

World に定義された RelationType を取得する

- 引数
  String　取得する RelationType の名前
- 対応するソースコード

```
RelationType 取得した RelationType = getWorld().getRelationType([String]);
```

World の説明を取得する

- 対応するソースコード

```
String 取得した World の説明 = getWorld().getDescription();
```

World の名前を取得する

- 対応するソースコード

```
String 取得した世界の名前 = getWorld().getName();
```

World を取得する

- 対応するソースコード

```
World 取得した World = getWorld();
```

現在のステップ数を取得する

- 対応するソースコード

```
StepClock 時計 = (StepClock)getWorld().getClock();
long 現在のステップ数 = clock.getStep();
```

指定した AgentType に対応する TimeEvent の優先度を取得する

- 引数
  AgentType　優先度を取得する AgentType
- 対応するソースコード

```
int 優先度の数値 = getWorld().getPriority([AgentType]);
```

世界に存在する Agent の中から指定した Type の Agent を一人取得する

- 引数
  AgentType　取得する Agent の Type
- 対応するソースコード

```
Agent 取得した Agent = getWorld().getAgent([AgentType]);
```

世界に存在する指定した Type の Agent を親 Type を指定して全て取得する

- 引数
  AgentType　取得する Agent の親 Type
- 対応するソースコード

```
Collection 取得した Agent の集合 = getWorld().getAgentsRecursively([AgentType]);
```

世界に存在する指定した Type の Agent を全て取得する

- 引数
  AgentType　取得する Agent の Type
- 対応するソースコード

```
Collection 取得した Agent の集合 = getWorld().getAgents([AgentType]);
```

世界に定義されている整数型 (int) のパラメータの値を取得する

- 引数
  String　パラメータの名前
- 対応するソースコード

```
//値を取り出す準備
World にアクセスするためのオブジェクト
  = (ObjectObject)WrapperObject.getInstance(getWorld());

//値を取り出す
FieldObject 値を表すオブジェクト = object.getValue((Integer)object.getField([String]));
int 取り出した整数（int）の値 = value.intValue();
```

乱数ジェネレータを取得する

- 対応するソースコード

```
RandomNumberGenerator 乱数ジェネレータを取得する = getWorld().getRandomNumberGenerator();
```

乱数ジェネレータを名前を指定して取得する

- 引数
  String　乱数ジェネレータの名前
- 対応するソースコード

```
RandomNumberGenerator 乱数ジェネレータを取得する
  = getWorld().getRandomNumberGenerator([String]);
```

A.15　カテゴリ「計算」

0 以上 1 未満の実数の乱数を生成する

- 対応するソースコード

```
double 生成した乱数 = getWorld().getRandomNumberGenerator().generate();
```

実数配列の合計と平均を計算する

- 引数
  double[]　合計と平均を計算するための配列
- 対応するソースコード

```
double 合計 = 0;
double[] 実数配列 = [double[]];

//合計を計算する
for(double i = 0; i < 実数配列.length; i++){
合計 = 合計 + 整数配列 [i];
}

//平均を計算する
double 平均 = 合計 / 実数配列.length;
```

整数の乱数を生成する

- 引数
  int　乱数の最大値
- 対応するソースコード

```
int 生成した乱数 = getWorld().getRandomNumberGenerator().generate([int]);
```

整数配列の合計と平均を計算する

- 引数
  int[]　合計と平均を計算するための配列
- 対応するソースコード

```
int 合計 = 0;
int[] 整数配列 = [int[]];

//合計を計算する
for(int i = 0; i < 整数配列.length; i++){
合計 = 合計 + 整数配列[i];
}

//平均を計算する
double 平均 = 合計 / 整数配列.length;
```

A.16　カテゴリ「集合操作」

Agent の集合からランダムで N 人の Agent を取り出す

- 引数
  Collection　抽出元の集合
  int　取り出す Agent の数
- 対応するソースコード

```
List 抽出元の Agent のリスト = new ArrayList([Collection]);
int 抽出する要素の数 = [int];
List 選ばれた Agent のリスト = new ArrayList();

//ランダムで要素を選ぶ
for(int i = 0; i < 抽出する要素の数; i++){
   int 生成した乱数
    = getWorld().getRandomNumberGenerator().generate(抽出元の Agent のリスト.size());

   Agent 取り出した Agent = (Agent) 抽出元の Agent のリスト.remove(生成した乱数);
  選ばれた Agent のリスト.add(取り出した Agent);
}
```

Agent の集合からランダムで一人を選ぶ

- 引数
  Collection　抽出元の集合
- 対応するソースコード

```
List Agent のリスト = new ArrayList([Collection]);

//ランダムで一人選ぶ
int インデックス番号
  = getWorld().getRandomNumberGenerator().generate(Agent のリスト.size());
Agent ランダムで選ばれた Agent = (Agent)Agent のリスト.get(インデックス番号);
```

Behavior の集合からランダムで一つを選ぶ

- 引数
  Collection　抽出元の集合
- 対応するソースコード

```
List Behavior のリスト = new ArrayList([Collection]);

//ランダムで一つ選ぶ
int インデックス番号
  = getWorld().getRandomNumberGenerator().generate(Behavior のリスト.size());
Behavior ランダムで選ばれた Behavior = (Behavior)Behavior のリスト.get(インデックス番号);
```

Goods の集合からランダムで一人を選ぶ

- 引数
  Collection　抽出元の集合
- 対応するソースコード

```
List Goods のリスト = new ArrayList([Collection]);

//ランダムで一つ選ぶ
int インデックス番号
  = getWorld().getRandomNumberGenerator().generate(Goods のリスト.size());
Goods ランダムで選ばれた Goods = (Goods)Goods のリスト.get(インデックス番号);
```

Information の集合からランダムで一人を選ぶ

- 引数
  Collection　抽出元の集合
- 対応するソースコード

```
List Information のリスト = new ArrayList([Collection]);

//ランダムで一つ選ぶ
int インデックス番号
  = getWorld().getRandomNumberGenerator().generate(Information のリスト.size());
Information ランダムで選ばれた Information
  = (Information)Information のリスト.get(インデックス番号);
```

Relation の集合からランダムで N 個の要素を取り出す

- 引数
  Collection　抽出元の集合
  int　取り出す Relation の数
- 対応するソースコード

```
List 抽出元の Relation のリスト = new ArrayList([Collection]);
int 抽出する要素の数 = [int];
List 選ばれた Relation のリスト = new ArrayList();

//ランダムで要素を選ぶ
for(int i = 0; i < 抽出する要素の数; i++){
   int 生成した乱数
    = getWorld().getRandomNumberGenerator().generate(抽出元の Relation のリスト.size());

   Relation 取り出した Relation
    = (Relation)抽出元の Relation のリスト.remove(生成した乱数);
  選ばれた Relation のリスト.add(取り出した Relation);
}
```

Relation の集合からランダムで一人を選ぶ

- 引数
  Collection　抽出元の集合
- 対応するソースコード

```
List Relationのリスト = new ArrayList([Collection]);

//ランダムで一つ選ぶ
int インデックス番号
  = getWorld().getRandomNumberGenerator().generate(Relationのリスト.size());
Relation ランダムで選ばれた Relation
  = (Relation)Relationのリスト.get(インデックス番号);
```

マップに要素を追加する

- 引数
  Map　　対象となるマップ
  Object　キーとなるオブジェクト
  Object　追加する要素
- 対応するソースコード

```
[Map].put([Object],[Object]);
```

マップのキーを取得する

- 引数
  Map　対象となるマップ
- 対応するソースコード

```
Set キーのセット = [Map].keySet();
```

マップの要素を取得する

- 引数
  Map　　対象となるマップ
  Object　要素を取得するキー
- 対応するソースコード

```
Object 取得した要素 = [Map].get([Object]);
```

マップの要素を集合として取得する

- 引数
  Map　対象となるマップ
- 対応するソースコード

```
Collection マップの要素の集合 = [Map].values();
```

リストの N 番目を削除する

- 引数
  List　対象となるリスト
  int　何番目を削除するか
- 対応するソースコード

```
Object 削除した要素 = [List].remove([int]);
```

リストの N 番目を取得する

- 引数
  List　対象となるリスト
  int　何番目を取得するか
- 対応するソースコード

```
Object 取得した要素 = [List].get([int]);
```

集合からリストを作る

- 引数
  Collection　対象となる集合
- 対応するソースコード

```
List 新しいリスト = new ArrayList([Collection]);
```

集合から要素を削除する

- 引数
  Collection　対象となる集合
  Object　削除する要素
- 対応するソースコード

```
[Collection].remove([Object]);
```

集合に要素を追加する

- 引数
  Collection　対象となる集合
  Object　追加する要素
- 対応するソースコード

```
[Collection].add([Object]);
```

集合の要素数を取得する

- 引数
  Collection　対象となる集合
- 対応するソースコード

```
int 要素数 = [Collection].size();
```

A.17　カテゴリ「出力」

デバッグとしてログを出力する

- 引数
  Logger　ログの出力に使う Logger
  String　ログの内容
- 対応するソースコード

```
[Logger].debug([String]);
```

警告としてログを出力する

- 引数
  Logger　ログの出力に使う Logger
  String　ログの内容
- 対応するソースコード

```
[Logger].warn([String]);
```

情報としてログを出力する

- 引数
  Logger　ログの出力に使う Logger
  String　ログの内容
- 対応するソースコード

```
[Logger].info([String]);
```

標準出力に出力する

- 引数
  String　出力する内容
- 対応するソースコード

```
System.out.println([String]);
```

付 録B　パン屋モデルの詳細

ここでは，本文，第 5 章の第 5.2 節で実験に使用したサンプルモデルである「パン屋モデル」の詳細を紹介する．ここで紹介しているのは以下の分析・設計モデルと，それらから CB によって自動生成することができるソースコードである．

- 概念モデリングフェーズ
  - シミュレーションに登場する概念を分析するクラス図
  - Agent のアクティビティを分析するシーケンス図
  - Agent 間の相互作用を分析するシーケンス図
- シミュレーションデザインフェーズ
  - Type を設計するするクラス図とそこから生成されるソースコード
  - Behavior の状態遷移を設計するするステートチャート図とそこから生成されるソースコード
  - Behavior のアクションを設計するする ABL の図とそこから生成されるソースコード
  - World の初期設定とそこから生成されるソースコード

このモデルは実際に BESP 上で実行することができる．図 B.1 はパン屋モデルの BESP 上での実行画面である．

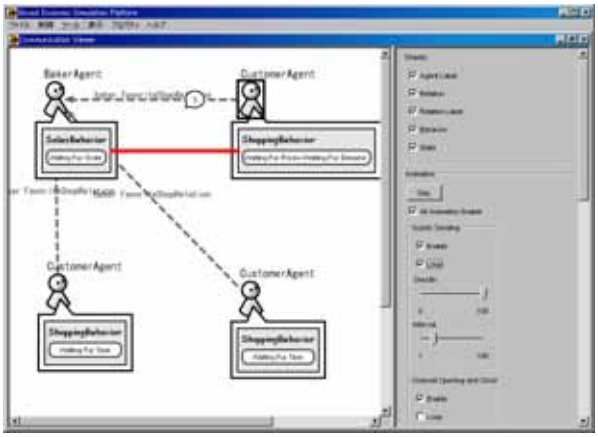


図 B.1: パン屋モデルの実行画面

B.1　概念モデリングフェーズ

(1) シミュレーションの登場する概念の抽出

パン屋モデルに登場する Agent は「BakerAgent」(パン屋) と「CustomerAgent」(お客さん) の 2 種類である．その間には「FavoriteShopRelation」(お得意先の関係) がある (図 B.2)．これらの Agent は 2 種類の Goods(図 B.3) と 2 種類の Information(図 B.4) をやりとりして取引を行う．

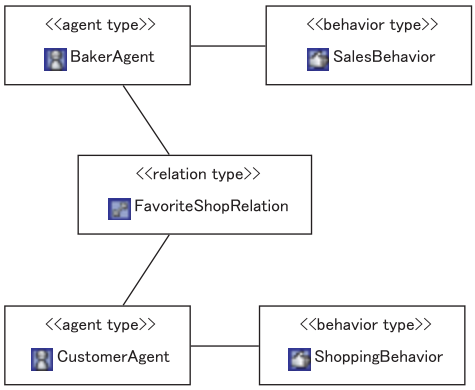
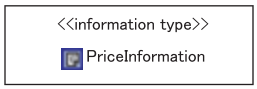


図 B.2: Agent,Relation,Behavior の構造



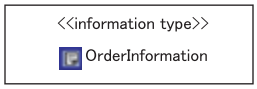


図 B.3: Goods の構造　　図 B.4: Information の構造

(2) Agent のアクティビティの分析

パン屋モデルにおいてアクティビティ分析対象となるのは，BakerAgent と CustomerAgent である．CustomerAgent は，まずパン屋さんに挨拶をして，価格を尋ね，注文，支払いを行って，最後にパンを受け取る (図 B.5)．BakerAgent は，お客さんからの問いかけに応じて，挨拶を返し，パンの価格を答え，支払いを請求し，お金を受け取って最後にパンをお客さんに渡す (図 B.6)．

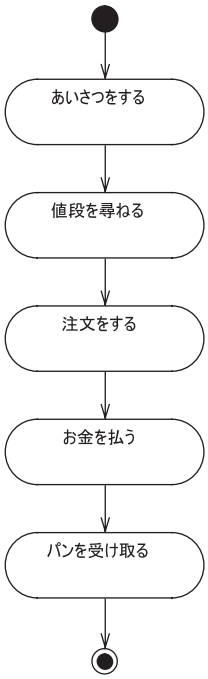


図 B.5: CustmerAgent のアクティビティ

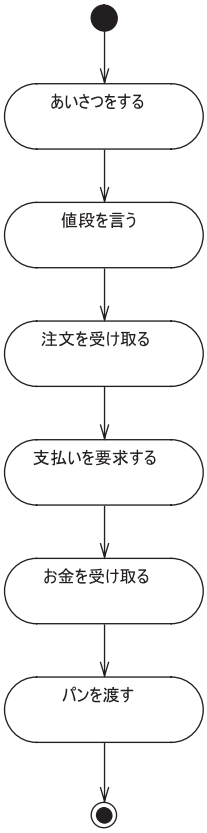


図 B.6: BakerAgent のアクティビティ

(3) Agent 間の相互作用の分析

ここでは，アクティビティの分析をもとに，BakerAgent と CustomerAgent がどのような流れで Goods と Information をやり取りしているのかを明らかにする．BakerAgent と CustomerAgent は,「PriceInformation」(価格の情報) と「OrderInformation」(注文の情報),「MoneyGoods」(お金) と「BreadGoods」(パン) を順にやり取りしていく (図 B.7).

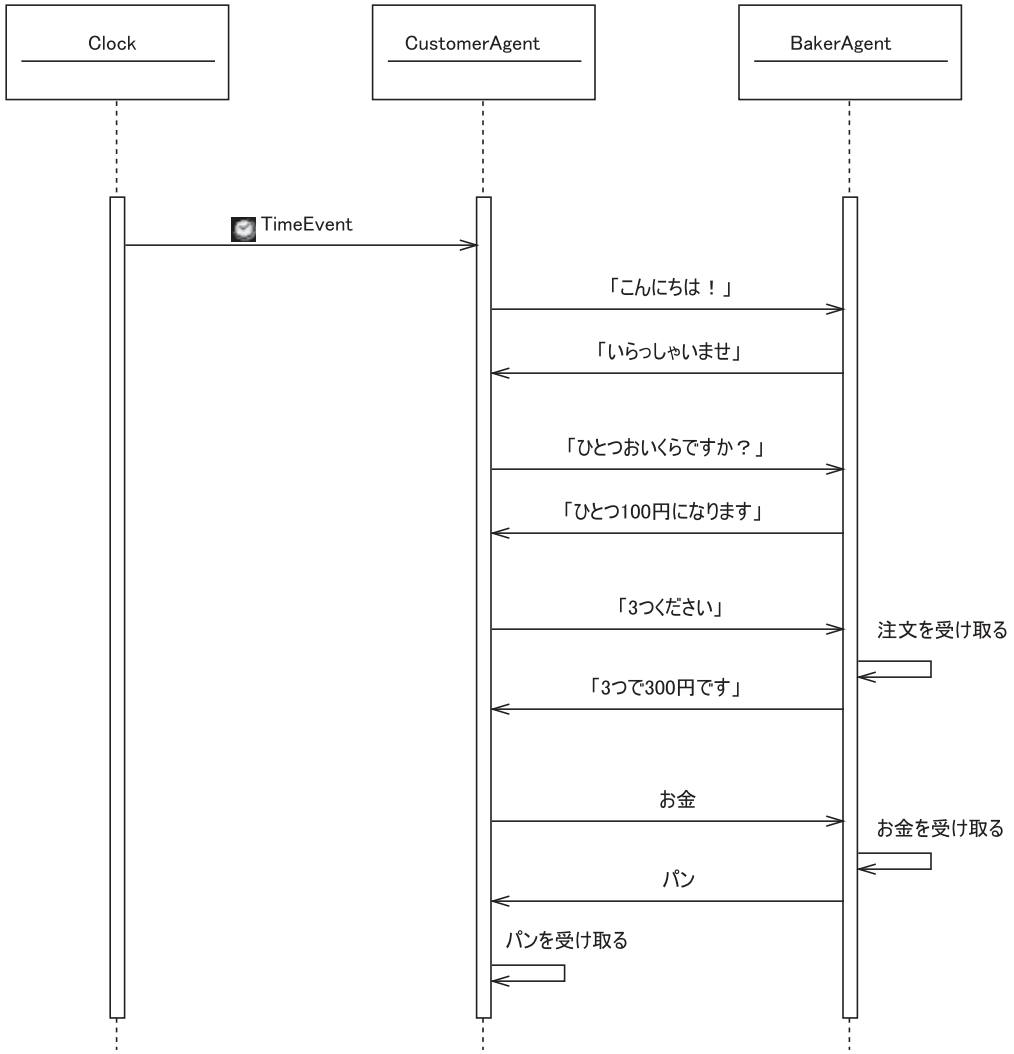


図 B.7: BakerAgent と CustomerAgent の相互作用

B.2　シミュレーションデザインフェーズ

(1) Type の設計と CB が生成するソースコード

パン屋モデルはシンプルなモデルなので，設計の段階で新たに Type を定義せずに，概念モデル (図 B.2 から図 B.4) をそのまま設計モデルとして利用することができる．

リスト 1 は，設計モデルのクラス図から CB が自動で生成するソースコードである．

リスト 1: CB が生成した Type を定義するソースコード

```
 1:  /*
 2:  * BoxTownModel.java
 3:  * (Component Builderによって生成されたソースコード)
 4:  */
 5: package baker;
 6:
 7: import …(略)
 8:
 9: /**
10:  * Model for Boxed Economy Foundation Model Framework
11:  */
12: public class BoxTownModel implements BESPPlugin {
13:
14:   public static AgentType AGENTTYPE_CustomerAgent;
15:   public static AgentType AGENTTYPE_BakerAgent;
16:   public static BehaviorType BEHAVIORTYPE_ShoppingBehavior;
17:   public static BehaviorType BEHAVIORTYPE_SalesBehavior;
18:   public static RelationType RELATIONTYPE_FavoriteShopRelation;
19:   public static InformationType INFORMATIONTYPE_PriceInformation;
20:   public static InformationType INFORMATIONTYPE_OrderInformation;
21:   public static GoodsType GOODSTYPE_MoneyGoods;
22:   public static GoodsType GOODSTYPE_BreadGoods;
23:
24:   /**
25:    * This method will be modified automatically by Model Editor
26:    */
27:   public static void initializePlugin(BESPContainer container) {
28:     installTypes(container);
29:     buildStructure(container);
30:     setPriority(container);
31:   }
32:
33:   /**
34:    * This method will be modified automatically by Model Editor
35:    */
36:   private static void installTypes(BESPContainer container) {
37:     //Get the model container
38:     ModelContainer modelContainer = container.getModelContainer();
39:
40:     AGENTTYPE_CustomerAgent
41:       = modelContainer.installAgentType("baker.CustomerAgent");
```

```
42:     AGENTTYPE_BakerAgent = modelContainer.installAgentType("baker.BakerAgent");
43:     BEHAVIORTYPE_ShoppingBehavior
44:       = modelContainer.installBehaviorType("baker.ShoppingBehavior");
45:     BEHAVIORTYPE_SalesBehavior
46:       = modelContainer.installBehaviorType("baker.SalesBehavior");
47:     RELATIONTYPE_FavoriteShopRelation
48:       = modelContainer.installRelationType("baker.FavoriteShopRelation");
49:     INFORMATIONTYPE_PriceInformation
50:       = modelContainer.installInformationType("baker.PriceInformation");
51:     INFORMATIONTYPE_OrderInformation
52:       = modelContainer.installInformationType("baker.OrderInformation");
53:     GOODSTYPE_MoneyGoods = modelContainer.installGoodsType("baker.MoneyGoods");
54:     GOODSTYPE_BreadGoods = modelContainer.installGoodsType("baker.BreadGoods");
55:   }
56:
57:   /**
58:    * This method will be modified automatically by Model Editor
59:    */
60:   private static void buildStructure(BESPContainer container) {
61:   }
62:
63:   /**
64:    * Set priority
65:    */
66:   private static void setPriority(BESPContainer container) {
67:   }
68: }
```

(2) Behavior の状態遷移の設計と CB が生成するソースコード

ここでは概念モデリングフェーズで行ったアクティビティ分析と相互作用の分析をもとに，Behavior の状態遷移を設計する．図 B.8 は，ShoppingBehavior の状態遷移を設計したステートチャート図である．これから生成されるソースコードがリスト 2 である．図 B.9 は，SalesBehavior の状態遷移を設計したステートチャート図である．これから生成されるソースコードがリスト 3 である．

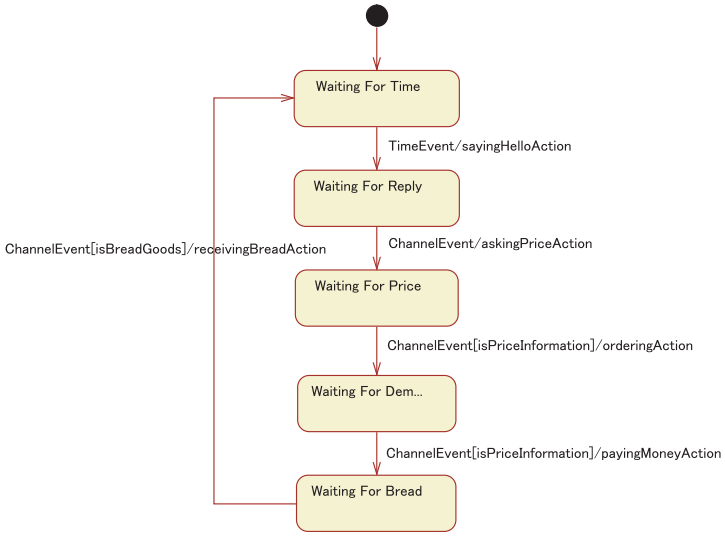


図 B.8: ShoppingBehavior の状態遷移

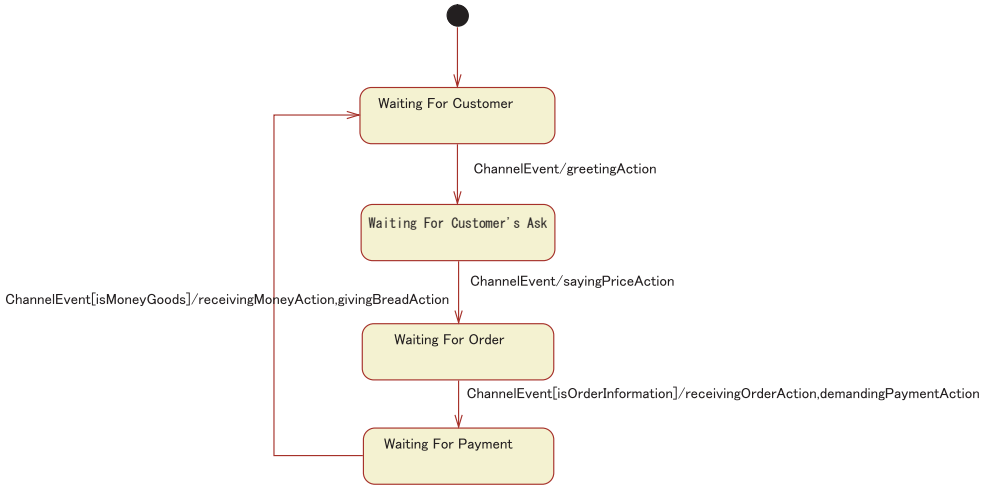


図 B.9: SalesBehavior の状態遷移

リスト 2: CB が生成する ShoppingBehavior の状態遷移のソースコード

```
 1: /*
 2:  * Created on 2004/11/21
 3:  * (Component Builderによって生成されたソースコード)
 4:  */
 5: package baker;
 6:
 7: import …(略)
 8:
 9: /**
10:  * AbstractShoppingBehavior
11:  */
12: public abstract class AbstractShoppingBehavior extends AbstractBehavior {
13:
14:   /**
15:    * This method automatically generated from Boxed Economy Behavior Editor.
16:    * Don't modify this method.
17:    */
18:   protected void initializeStateMachine() {
19:     //factory
20:     StateMachineFactory factory = this.getStateMachine();
21:
22:     //states
23:     State initialState = factory.createInitialState();
24:     CompositeState waiting_For_Time = factory
25:         .createCompositeState("Waiting For Time");
26:     CompositeState waiting_For_Reply = factory
27:         .createCompositeState("Waiting For Reply");
28:     CompositeState waiting_For_Price = factory
29:         .createCompositeState("Waiting For Price");
30:     CompositeState waiting_For_Demand_ = factory
31:         .createCompositeState("Waiting For Demand ");
32:     CompositeState waiting_For_Bread = factory
33:         .createCompositeState("Waiting For Bread");
34:
35:     //actions
36:     Action payingMoneyAction = new Action() {
37:       public void doAction() {
38:         payingMoneyAction();
39:       }
40:       public String toString() {
41:         return "payingMoneyAction";
42:       }
43:     };
44:     Action orderingAction = new Action() {
45:       public void doAction() {
46:         orderingAction();
47:       }
48:       public String toString() {
49:         return "orderingAction";
50:       }
51:     };
```

```
52:     Action askingPriceAction = new Action() {
53:       public void doAction() {
54:         askingPriceAction();
55:       }
56:       public String toString() {
57:         return "askingPriceAction";
58:       }
59:     };
60:     Action receivingBreadAction = new Action() {
61:       public void doAction() {
62:         receivingBreadAction();
63:       }
64:       public String toString() {
65:         return "receivingBreadAction";
66:       }
67:     };
68:     Action sayingHelloAction = new Action() {
69:       public void doAction() {
70:         sayingHelloAction();
71:       }
72:       public String toString() {
73:         return "sayingHelloAction";
74:       }
75:     };
76:
77:     //guard-conditions
78:     GuardCondition isPriceInformation = new GuardCondition() {
79:       public boolean isMatched(Event e) {
80:         return isPriceInformation(e);
81:       }
82:     };
83:     GuardCondition isBreadGoods = new GuardCondition() {
84:       public boolean isMatched(Event e) {
85:         return isBreadGoods(e);
86:       }
87:     };
88:
89:     //transitions
90:     Transition transitionWaiting_For_ReplyToWaiting_For_Price = factory
91:         .createTransition();
92:     Transition transitionWaiting_For_TimeToWaiting_For_Reply = factory
93:         .createTransition();
94:     Transition transitionWaiting_For_BreadToWaiting_For_Time = factory
95:         .createTransition();
96:     Transition transitionInitialStateToWaiting_For_Time = factory
97:         .createTransition();
98:     Transition transitionWaiting_For_PriceToWaiting_For_Demand_ = factory
99:         .createTransition();
100:    Transition transitionWaiting_For_Demand_ToWaiting_For_Bread = factory
101:        .createTransition();
102:
103:    //structure of states
```

```
104:     this.setInitialState(initialState);
105:     this.addState(waiting_For_Time);
106:     this.addState(waiting_For_Reply);
107:     this.addState(waiting_For_Price);
108:     this.addState(waiting_For_Demand_);
109:     this.addState(waiting_For_Bread);
110:
111:     //transitions setting
112:     transitionWaiting_For_ReplyToWaiting_For_Price
113:         .setEvent(ChannelEvent.class);
114:     transitionWaiting_For_ReplyToWaiting_For_Price
115:         .addAction(askingPriceAction);
116:     transitionWaiting_For_TimeToWaiting_For_Reply.setEvent(TimeEvent.class);
117:     transitionWaiting_For_TimeToWaiting_For_Reply
118:         .addAction(sayingHelloAction);
119:     transitionWaiting_For_BreadToWaiting_For_Time
120:         .setEvent(ChannelEvent.class);
121:     transitionWaiting_For_BreadToWaiting_For_Time
122:         .setGuardCondition(isBreadGoods);
123:     transitionWaiting_For_BreadToWaiting_For_Time
124:         .addAction(receivingBreadAction);
125:     transitionWaiting_For_PriceToWaiting_For_Demand_
126:         .setEvent(ChannelEvent.class);
127:     transitionWaiting_For_PriceToWaiting_For_Demand_
128:         .setGuardCondition(isPriceInformation);
129:     transitionWaiting_For_PriceToWaiting_For_Demand_
130:         .addAction(orderingAction);
131:     transitionWaiting_For_Demand_ToWaiting_For_Bread
132:         .setEvent(ChannelEvent.class);
133:     transitionWaiting_For_Demand_ToWaiting_For_Bread
134:         .setGuardCondition(isPriceInformation);
135:     transitionWaiting_For_Demand_ToWaiting_For_Bread
136:         .addAction(payingMoneyAction);
137:
138:     //connection of transitions
139:     transitionWaiting_For_ReplyToWaiting_For_Price
140:         .setSourceState(waiting_For_Reply);
141:     transitionWaiting_For_ReplyToWaiting_For_Price
142:         .setTargetState(waiting_For_Price);
143:     transitionWaiting_For_TimeToWaiting_For_Reply
144:         .setSourceState(waiting_For_Time);
145:     transitionWaiting_For_TimeToWaiting_For_Reply
146:         .setTargetState(waiting_For_Reply);
147:     transitionWaiting_For_BreadToWaiting_For_Time
148:         .setSourceState(waiting_For_Bread);
149:     transitionWaiting_For_BreadToWaiting_For_Time
150:         .setTargetState(waiting_For_Time);
151:     transitionInitialStateToWaiting_For_Time.setSourceState(initialState);
152:     transitionInitialStateToWaiting_For_Time
153:         .setTargetState(waiting_For_Time);
154:     transitionWaiting_For_PriceToWaiting_For_Demand_
155:         .setSourceState(waiting_For_Price);
```

```
156:     transitionWaiting_For_PriceToWaiting_For_Demand_
157:         .setTargetState(waiting_For_Demand_);
158:     transitionWaiting_For_Demand_ToWaiting_For_Bread
159:         .setSourceState(waiting_For_Demand_);
160:     transitionWaiting_For_Demand_ToWaiting_For_Bread
161:         .setTargetState(waiting_For_Bread);
162:   }
163:
164:   /**
165:    *payingMoneyAction
166:    */
167:   protected abstract void payingMoneyAction();
168:
169:   /**
170:    *orderingAction
171:    */
172:   protected abstract void orderingAction();
173:
174:   /**
175:    *askingPriceAction
176:    */
177:   protected abstract void askingPriceAction();
178:
179:   /**
180:    *receivingBreadAction
181:    */
182:   protected abstract void receivingBreadAction();
183:
184:   /**
185:    *sayingHelloAction
186:    */
187:   protected abstract void sayingHelloAction();
188:
189:   /**
190:    *isPriceInformation
191:    */
192:   protected abstract boolean isPriceInformation(Event e);
193:
194:   /**
195:    *isBreadGoods
196:    */
197:   protected abstract boolean isBreadGoods(Event e);
198: }
```

リスト 3: CB が生成する SalesBehavior の状態遷移のソースコード

```
 1: /*
 2:  * (Component Builderによって生成されたソースコード)
 3:  */
 4: package baker;
 5:
 6: import ... (略)
 7:
 8: /**
 9:  * AbstractSalesBehavior
10:  */
11: public abstract class AbstractSalesBehavior extends AbstractBehavior {
12:
13:   /**
14:    * This method automatically generated from Boxed Economy Behavior Editor.
15:    */
16:   protected void initializeStateMachine() {
17:     //factory
18:     StateMachineFactory factory = this.getStateMachine();
19:
20:     //states
21:     State initialState = factory.createInitialState();
22:     CompositeState waiting_For_Customer = factory
23:         .createCompositeState("Waiting For Customer");
24:     CompositeState waiting_For_Customer_s_Ask_ = factory
25:         .createCompositeState("Waiting For Customer's Ask ");
26:     CompositeState waiting_For_Order = factory
27:         .createCompositeState("Waiting For Order");
28:     CompositeState waiting_For_Payment = factory
29:         .createCompositeState("Waiting For Payment");
30:
31:     //actions
32:     Action demandingPaymentAction = new Action() {
33:       public void doAction() {
34:         demandingPaymentAction();
35:       }
36:       public String toString() {
37:         return "demandingPaymentAction";
38:       }
39:     };
40:     Action greetingAction = new Action() {
41:       public void doAction() {
42:         greetingAction();
43:       }
44:       public String toString() {
45:         return "greetingAction";
46:       }
47:     };
48:     Action receivingMoneyAction = new Action() {
49:       public void doAction() {
50:         receivingMoneyAction();
51:       }
```

```
52:       public String toString() {
53:         return "receivingMoneyAction";
54:       }
55:     };
56:     Action sayingPriceAction = new Action() {
57:       public void doAction() {
58:         sayingPriceAction();
59:       }
60:       public String toString() {
61:         return "sayingPriceAction";
62:       }
63:     };
64:     Action receivingOrderAction = new Action() {
65:       public void doAction() {
66:         receivingOrderAction();
67:       }
68:       public String toString() {
69:         return "receivingOrderAction";
70:       }
71:     };
72:     Action givingBreadAction = new Action() {
73:       public void doAction() {
74:         givingBreadAction();
75:       }
76:       public String toString() {
77:         return "givingBreadAction";
78:       }
79:     };
80:
81:     //guard-conditions
82:     GuardCondition isMoneyGoods = new GuardCondition() {
83:       public boolean isMatched(Event e) {
84:         return isMoneyGoods(e);
85:       }
86:     };
87:     GuardCondition isOrderInformation = new GuardCondition() {
88:       public boolean isMatched(Event e) {
89:         return isOrderInformation(e);
90:       }
91:     };
92:
93:     //transitions
94:     Transition transitionInitialStateToWaiting_For_Customer
95:       = factory.createTransition();
96:     Transition transitionWaiting_For_Customer_s_Ask_ToWaiting_For_Order
97:       = factory.createTransition();
98:     Transition transitionWaiting_For_PaymentToWaiting_For_Customer
99:       = factory.createTransition();
100:    Transition transitionWaiting_For_OrderToWaiting_For_Payment
101:      = factory.createTransition();
102:    Transition transitionWaiting_For_CustomerToWaiting_For_Customer_s_Ask_
103:      = factory.createTransition();
```

```
104:
105:     //structure of states
106:     this.setInitialState(initialState);
107:     this.addState(waiting_For_Customer);
108:     this.addState(waiting_For_Customer_s_Ask_);
109:     this.addState(waiting_For_Order);
110:     this.addState(waiting_For_Payment);
111:
112:     //transitions setting
113:     transitionWaiting_For_Customer_s_Ask_ToWaiting_For_Order
114:         .setEvent(ChannelEvent.class);
115:     transitionWaiting_For_Customer_s_Ask_ToWaiting_For_Order
116:         .addAction(sayingPriceAction);
117:     transitionWaiting_For_PaymentToWaiting_For_Customer
118:         .setEvent(ChannelEvent.class);
119:     transitionWaiting_For_PaymentToWaiting_For_Customer
120:         .setGuardCondition(isMoneyGoods);
121:     transitionWaiting_For_PaymentToWaiting_For_Customer
122:         .addAction(receivingMoneyAction);
123:     transitionWaiting_For_PaymentToWaiting_For_Customer
124:         .addAction(givingBreadAction);
125:     transitionWaiting_For_OrderToWaiting_For_Payment
126:         .setEvent(ChannelEvent.class);
127:     transitionWaiting_For_OrderToWaiting_For_Payment
128:         .setGuardCondition(isOrderInformation);
129:     transitionWaiting_For_OrderToWaiting_For_Payment
130:         .addAction(receivingOrderAction);
131:     transitionWaiting_For_OrderToWaiting_For_Payment
132:         .addAction(demandingPaymentAction);
133:     transitionWaiting_For_CustomerToWaiting_For_Customer_s_Ask_
134:         .setEvent(ChannelEvent.class);
135:     transitionWaiting_For_CustomerToWaiting_For_Customer_s_Ask_
136:         .addAction(greetingAction);
137:
138:     //connection of transitions
139:     transitionInitialStateToWaiting_For_Customer
140:         .setSourceState(initialState);
141:     transitionInitialStateToWaiting_For_Customer
142:         .setTargetState(waiting_For_Customer);
143:     transitionWaiting_For_Customer_s_Ask_ToWaiting_For_Order
144:         .setSourceState(waiting_For_Customer_s_Ask_);
145:     transitionWaiting_For_Customer_s_Ask_ToWaiting_For_Order
146:         .setTargetState(waiting_For_Order);
147:     transitionWaiting_For_PaymentToWaiting_For_Customer
148:         .setSourceState(waiting_For_Payment);
149:     transitionWaiting_For_PaymentToWaiting_For_Customer
150:         .setTargetState(waiting_For_Customer);
151:     transitionWaiting_For_OrderToWaiting_For_Payment
152:         .setSourceState(waiting_For_Order);
153:     transitionWaiting_For_OrderToWaiting_For_Payment
154:         .setTargetState(waiting_For_Payment);
155:     transitionWaiting_For_CustomerToWaiting_For_Customer_s_Ask_
```

```
156:         .setSourceState(waiting_For_Customer);
157:     transitionWaiting_For_CustomerToWaiting_For_Customer_s_Ask_
158:         .setTargetState(waiting_For_Customer_s_Ask_);
159:   }
160:
161:   /**
162:    *demandingPaymentAction
163:    */
164:   protected abstract void demandingPaymentAction();
165:   /**
166:    *greetingAction
167:    */
168:   protected abstract void greetingAction();
169:   /**
170:    *receivingMoneyAction
171:    */
172:   protected abstract void receivingMoneyAction();
173:   /**
174:    *sayingPriceAction
175:    */
176:   protected abstract void sayingPriceAction();
177:   /**
178:    *receivingOrderAction
179:    */
180:   protected abstract void receivingOrderAction();
181:   /**
182:    *givingBreadAction
183:    */
184:   protected abstract void givingBreadAction();
185:   /**
186:    *isMoneyGoods
187:    */
188:   protected abstract boolean isMoneyGoods(Event e);
189:   /**
190:    *isOrderInformation
191:    */
192:   protected abstract boolean isOrderInformation(Event e);
193: }
```

(3) Behavior のアクションの設計と CB が生成するソースコード

ここでは状態遷移を設計した際に定義したアクションの設計を行う．図 B.10 から図 B.15 は，ShoppingBehavior のアクションを設計した ABL による図である．これから生成されるソースコードがリスト 4 である．図 B.16 から図 B.22 は，SalesBehavior のアクションを設計した ABL による図である．これから生成されるソースコードがリスト 5 である．

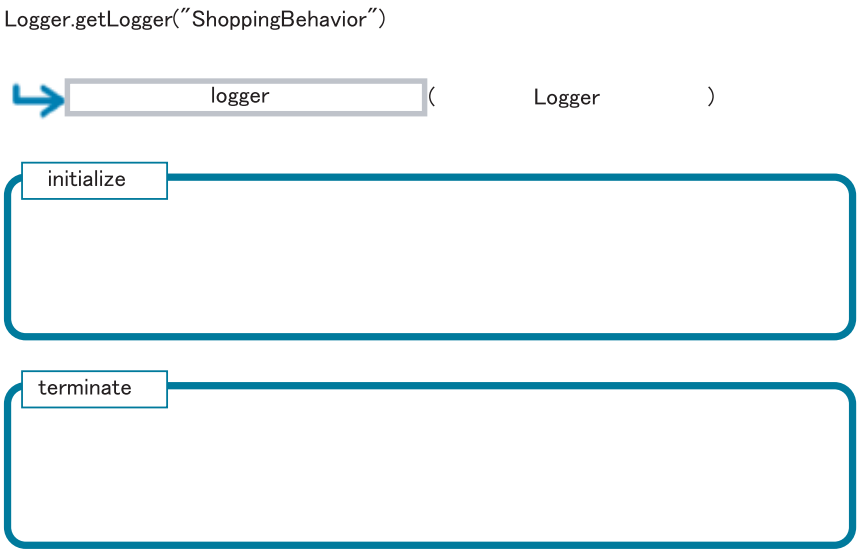


図 B.10: ShoppingBehavior のアクション (1)

図 B.11: ShoppingBehavior のアクション (2)

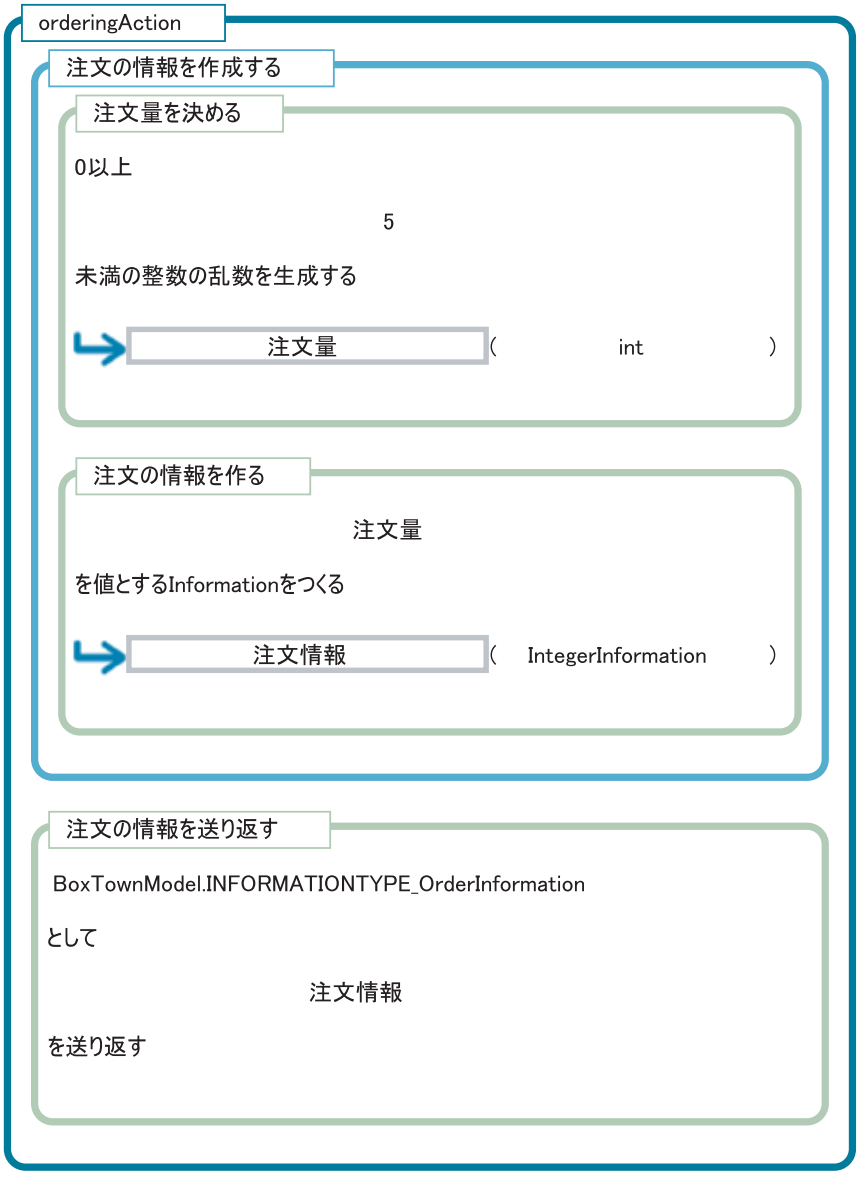


図 B.12: ShoppingBehavior のアクション (3)

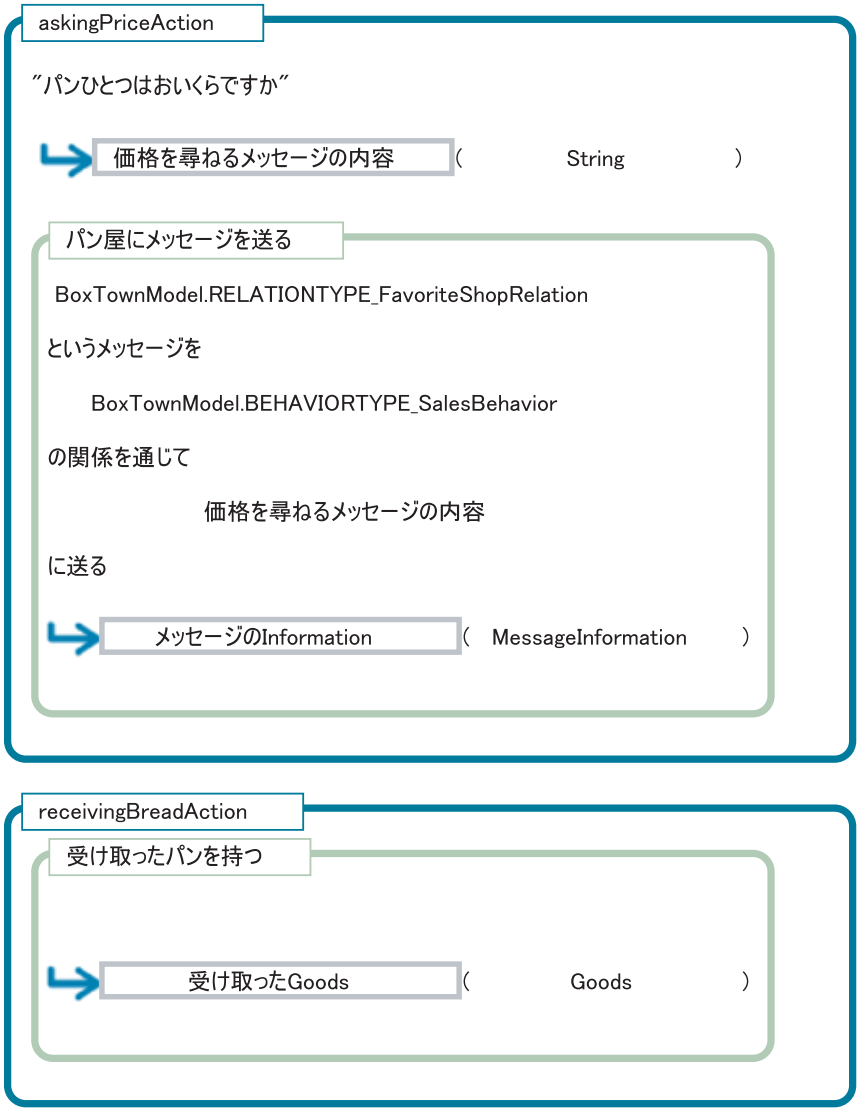


図 B.13: ShoppingBehavior のアクション (4)

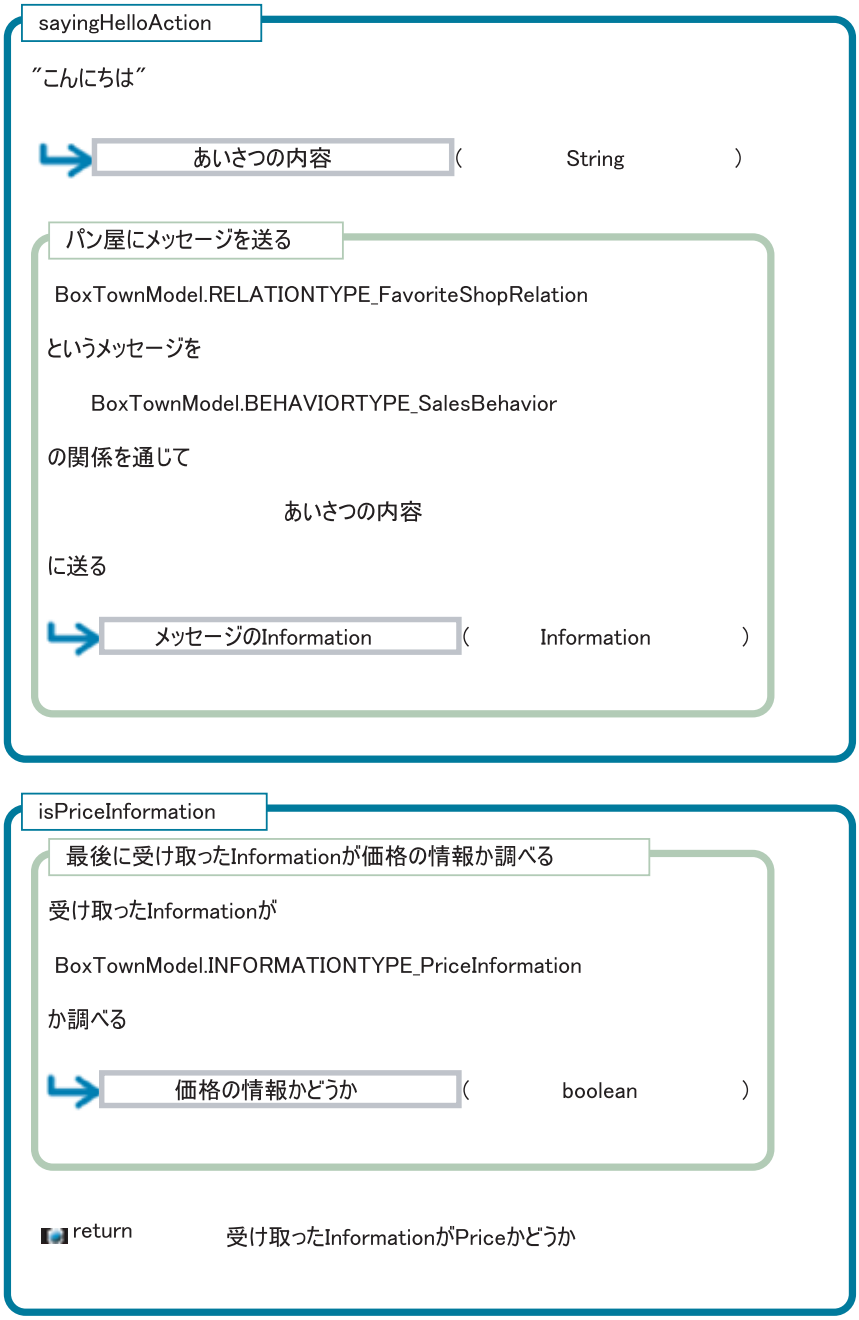


図 B.14: ShoppingBehavior のアクション (5)

図 B.15: ShoppingBehavior のアクション (6)

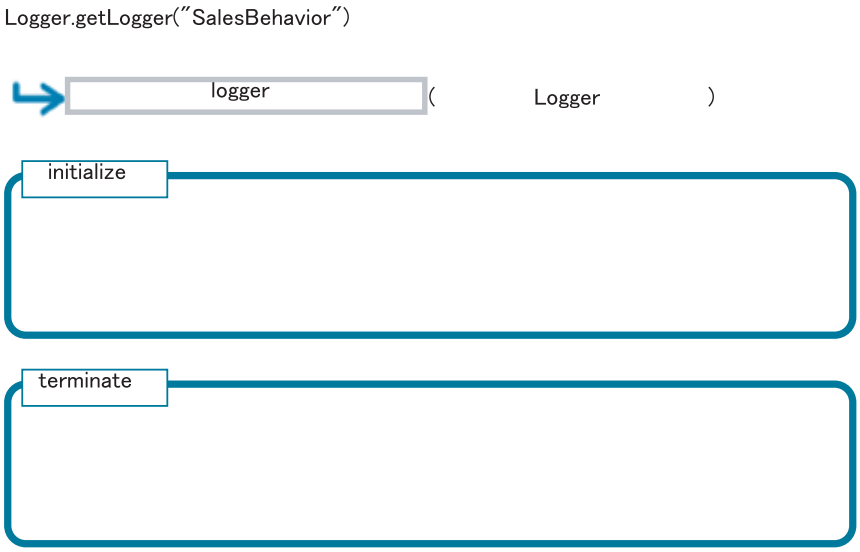


図 B.16: SalesBehavior のアクション (1)

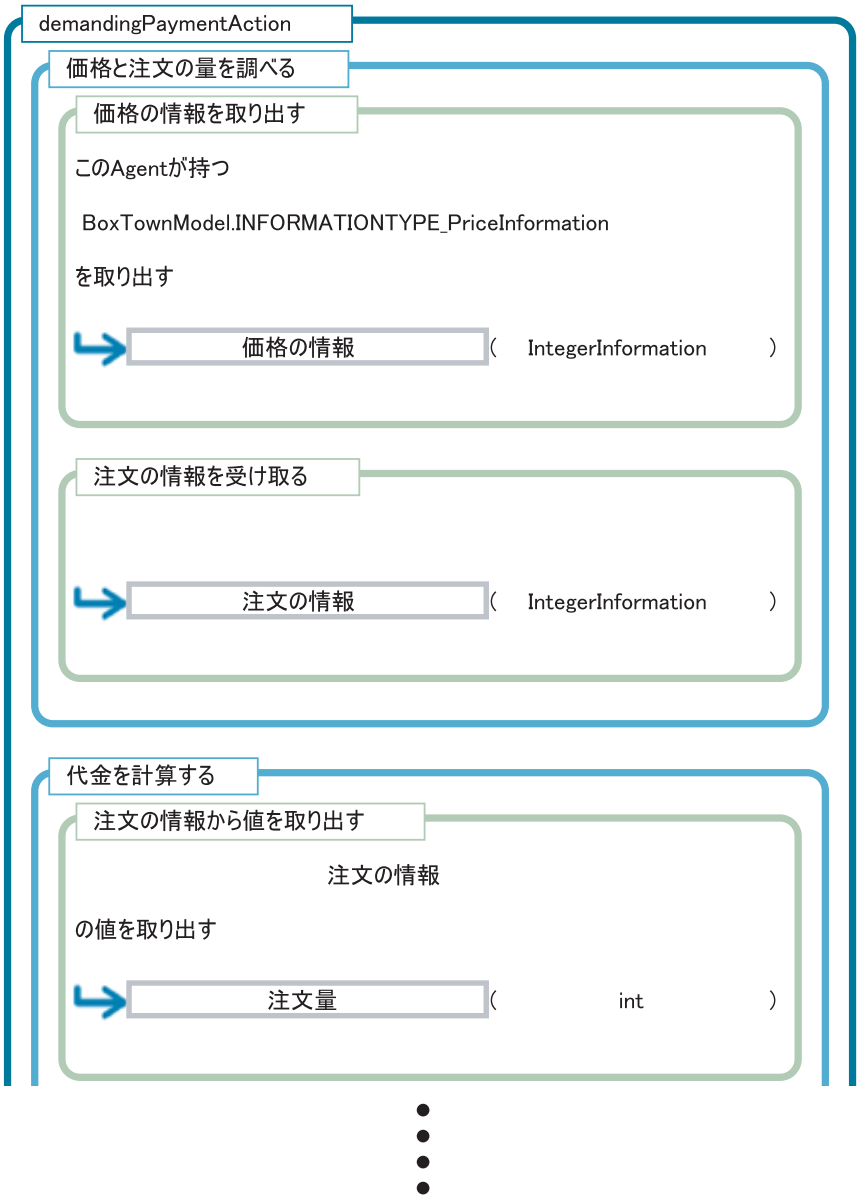


図 B.17: SalesBehavior のアクション (2)

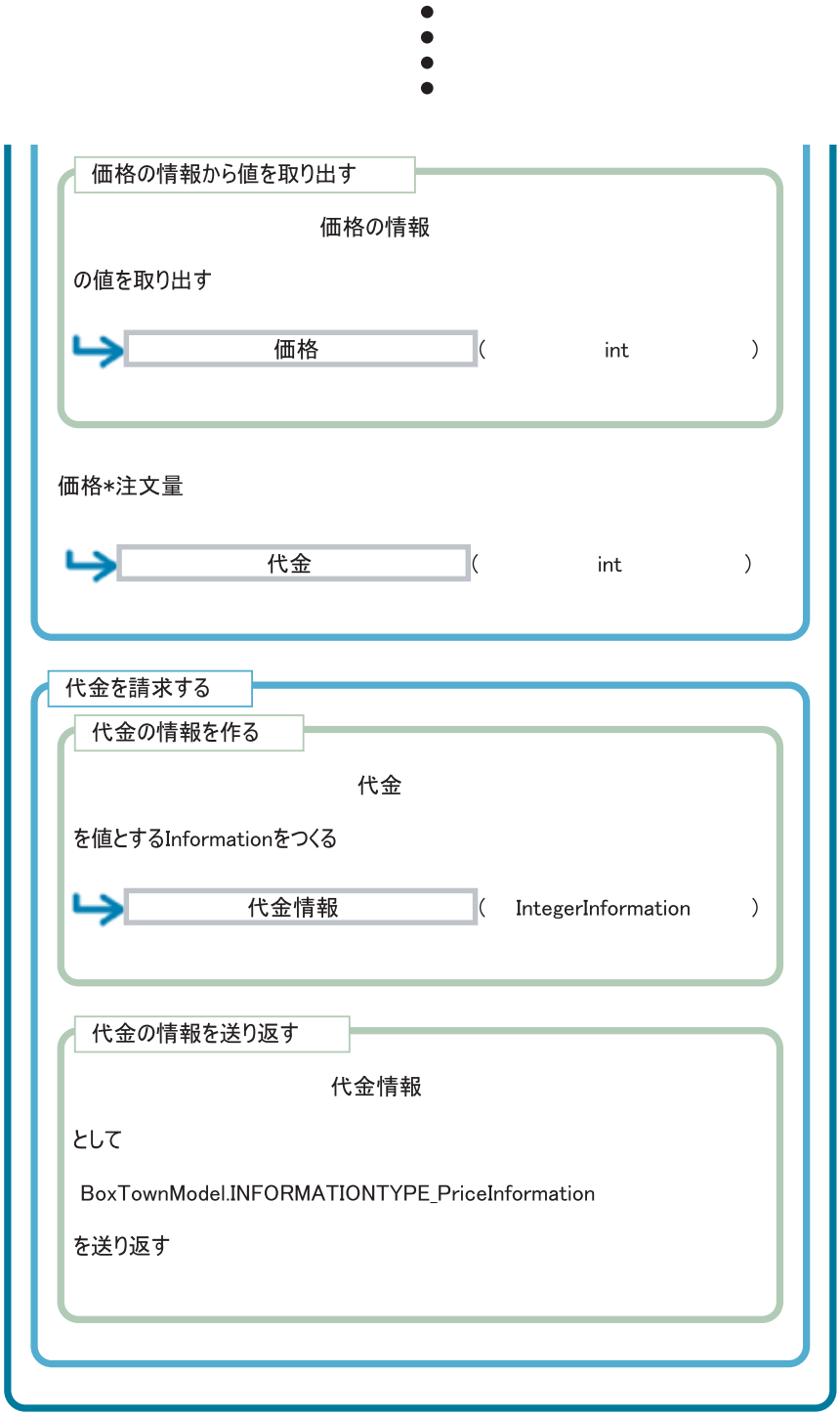


図 B.18: SalesBehavior のアクション (3)

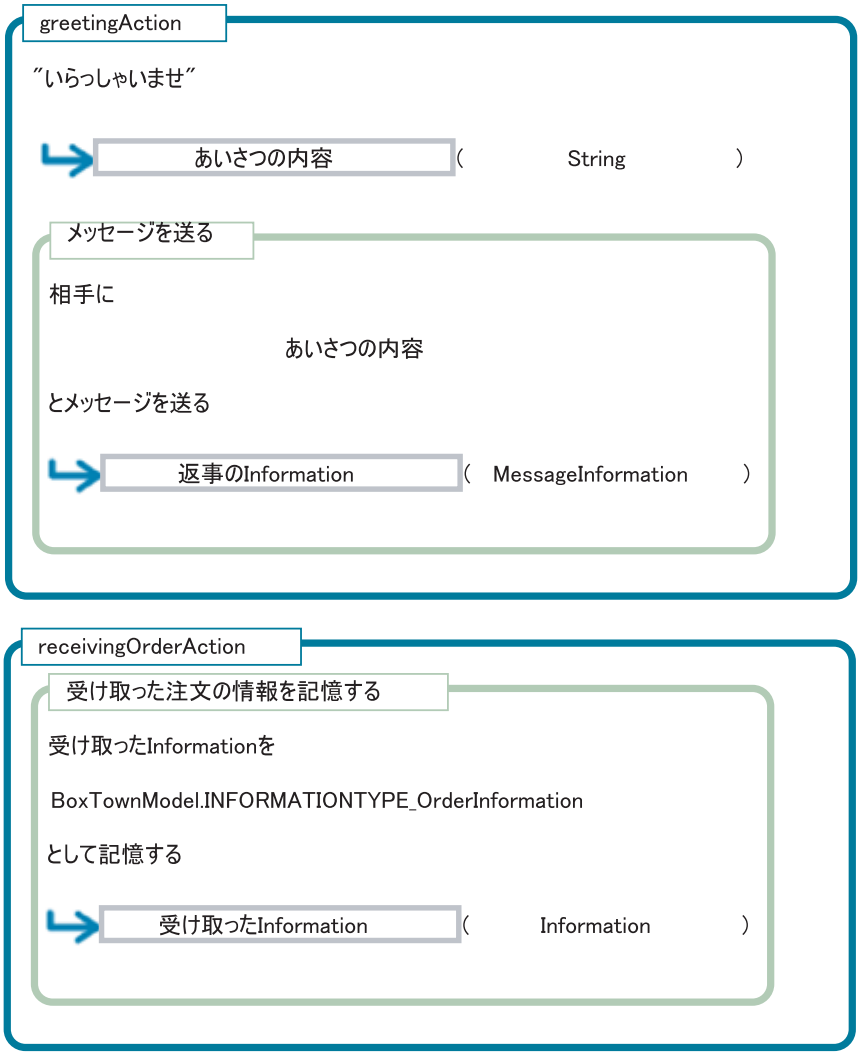


図 B.19: SalesBehavior のアクション (4)

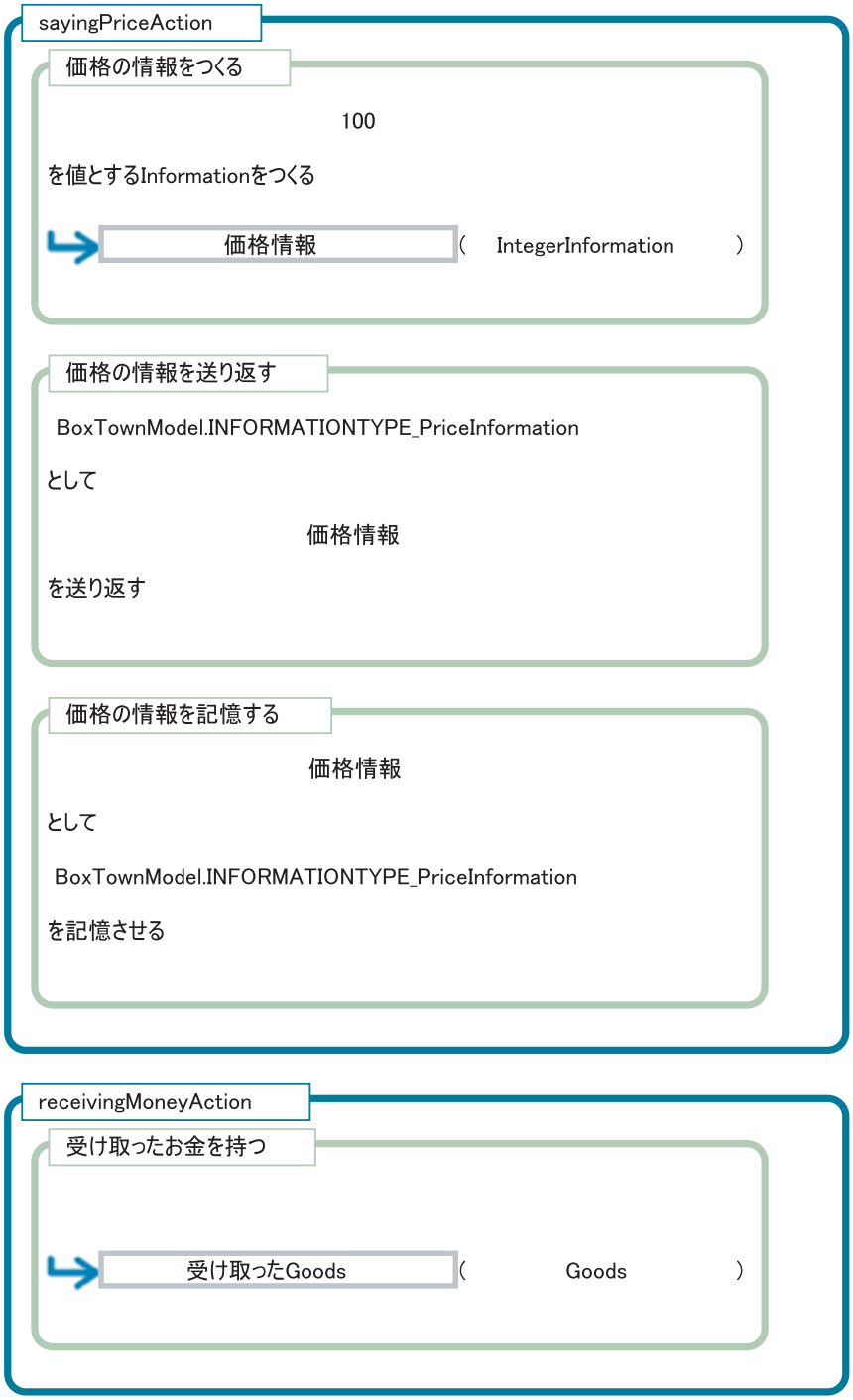


図 B.20: SalesBehavior のアクション (5)

図 B.21: SalesBehavior のアクション (6)

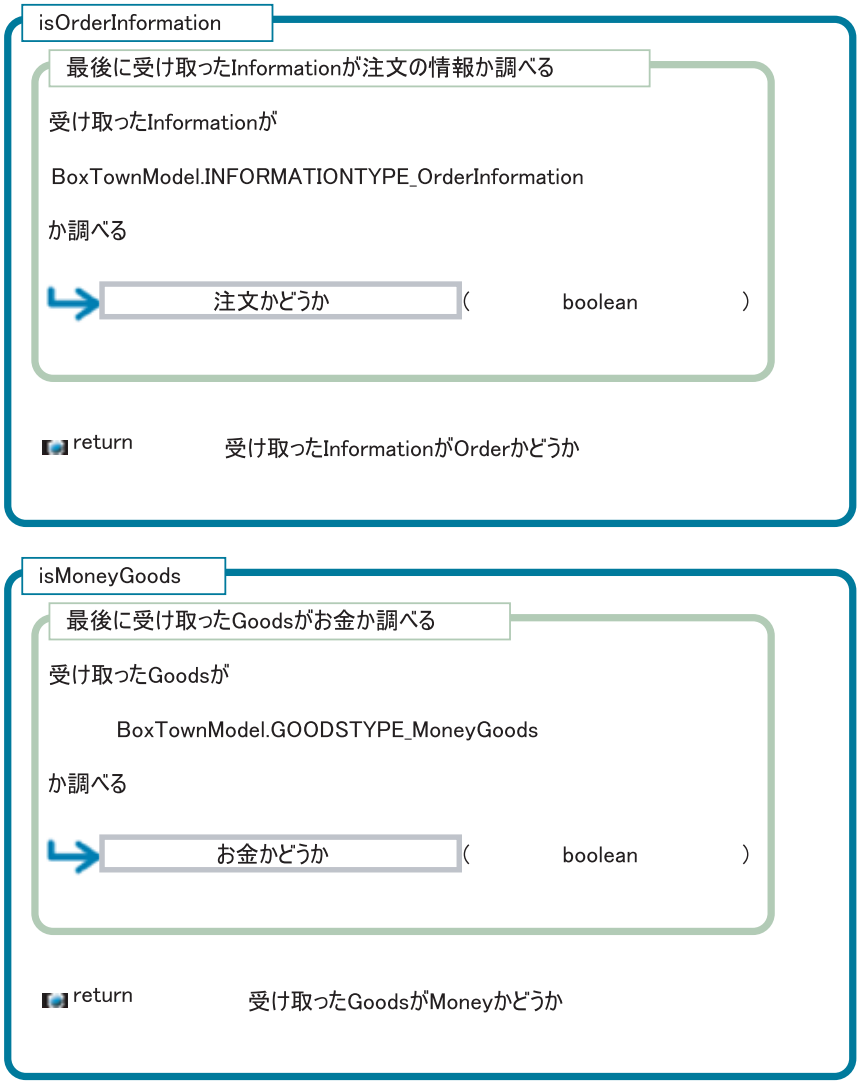


図 B.22: SalesBehavior のアクション (7)

リスト 4: CB が生成する ShoppingBehavior のアクションのソースコード

```
 1: /*
 2:  * Created on 2004/11/22
 3:  * (Component Builder によって生成されたソースコード)
 4:  */
 5: package baker;
 6:
 7: import ・・・(略)
 8:
 9: public class ShoppingBehavior extends AbstractShoppingBehavior {
10:
11:   /*****************************************
```

```
12:    * Attribute(s).
13:    ****************************************/
14:   private Logger logger = Logger.getLogger("ShoppingBehavior");
15:
16:   /****************************************
17:    * Method(s).
18:    ****************************************/
19:
20:   /**
21:    * initialize
22:    **/
23:   public void initialize() {
24:   }
25:
26:   /**
27:    * terminate
28:    **/
29:   public void terminate() {
30:   }
31:
32:   /**
33:    * payingMoneyAction
34:    **/
35:   public void payingMoneyAction() {
36:
37:     //代金を調べる
38:
39:     //  代金の情報を受け取る
40:     IntegerInformation 代金の情報 = (IntegerInformation) this
41:         .getReceivedInformation();
42:
43:     //  代金の情報から値を取り出す
44:     int 代金 = 代金の情報.getValue();
45:
46:     //代金を支払う
47:
48:     //  お金を取り出す
49:     Goods 取り出したお金 = this.getAgent().removeGoods(
50:         BoxTownModel.GOODSTYPE_MoneyGoods, 代金);
51:
52:     //  お金を送り返す
53:     this.sendGoods(取り出したお金);
54:   }
55:
56:   /**
57:    * orderingAction
58:    **/
59:   public void orderingAction() {
60:
61:     //注文の情報を作成する
62:
63:     //  注文量を決める
```

```
64:     int 注文量 = this.getWorld().getRandomNumberGenerator().generate(5);
65:
66:     //  注文の情報を作る
67:     IntegerInformation 注文情報 = new IntegerInformation(注文量);
68:
69:     //注文の情報を送り返す
70:     this.sendInformation(BoxTownModel.INFORMATIONTYPE_OrderInformation,
71:         注文情報);
72:   }
73:
74:   /**
75:    * askingPriceAction
76:    **/
77:   public void askingPriceAction() {
78:     String 価格を尋ねるメッセージの内容 = "パンひとつはおいくらですか";
79:
80:     //パン屋にメッセージを送る
81:     MessageInformation メッセージの Information = new MessageInformation(
82:         価格を尋ねるメッセージの内容);
83:     this.sendInformation(BoxTownModel.RELATIONTYPE_FavoriteShopRelation,
84:         BoxTownModel.BEHAVIORTYPE_SalesBehavior, メッセージの Information);
85:   }
86:
87:   /**
88:    * receivingBreadAction
89:    **/
90:   public void receivingBreadAction() {
91:
92:     //受け取ったパンを持つ
93:     Goods 受け取った Goods = this.getReceivedGoods();
94:     this.getAgent().addGoods(受け取った Goods);
95:   }
96:
97:   /**
98:    * sayingHelloAction
99:    **/
100:  public void sayingHelloAction() {
101:    String あいさつの内容 = "こんにちは";
102:
103:    //パン屋にメッセージを送る
104:    MessageInformation メッセージの Information1
105:      = (MessageInformation) new MessageInformation(あいさつの内容);
106:    this.sendInformation(BoxTownModel.RELATIONTYPE_FavoriteShopRelation,
107:        BoxTownModel.BEHAVIORTYPE_SalesBehavior, メッセージの Information1);
108:  }
109:
110:  /**
111:   * isPriceInformation
112:   **/
113:  public boolean isPriceInformation(Event e) {
114:
115:    //最後に受け取った Information が価格の情報か調べる
```

```
116:     boolean 価格の情報かどうか = this
117:         .receivedInformationEquals(BoxTownModel.INFORMATIONTYPE_PriceInformation);
118:     return 価格の情報かどうか;
119:   }
120:
121:   /**
122:    * isBreadGoods
123:    **/
124:   public boolean isBreadGoods(Event e1) {
125:
126:     //最後に受け取った Goods がパンか調べる
127:     boolean パンかどうか = this
128:         .receivedGoodsEquals(BoxTownModel.GOODSTYPE_BreadGoods);
129:     return パンかどうか;
130:   }
131: }
```

リスト 5: CB が生成する SalesBehavior のアクションのソースコード

```
 1: /*
 2:  * Created on 2004/11/11
 3:  * (Component Builder によって生成されたソースコード)
 4:  */
 5: package baker;
 6:
 7: import …(略)
 8:
 9: public class SalesBehavior extends AbstractSalesBehavior {
10:
11:   /*****************************************
12:    * Attribute(s).
13:    *****************************************/
14:   private Logger logger = Logger.getLogger("SalesBehavior");
15:
16:   /*****************************************
17:    * Method(s).
18:    *****************************************/
19:
20:   /**
21:    * initialize
22:    **/
23:   public void initialize() {
24:   }
25:
26:   /**
27:    * terminate
28:    **/
29:   public void terminate() {
30:   }
```

```
31:
32:   /**
33:    * demandingPaymentAction
34:    **/
35:   public void demandingPaymentAction() {
36:
37:     //価格と注文の量を調べる
38:
39:     //  価格の情報を取り出す
40:     IntegerInformation 価格の情報 = (IntegerInformation) this.getAgent()
41:         .getInformation(BoxTownModel.INFORMATIONTYPE_PriceInformation);
42:
43:     //  注文の情報を受け取る
44:     IntegerInformation 注文の情報 = (IntegerInformation) this
45:         .getReceivedInformation();
46:
47:     //代金を計算する
48:
49:     //  注文の情報から値を取り出す
50:     int 注文量 = 注文の情報.getValue();
51:
52:     //  価格の情報から値を取り出す
53:     int 価格 = 価格の情報.getValue();
54:     int 代金 = 価格 * 注文量;
55:
56:     //代金を請求する
57:
58:     //  代金の情報を作る
59:     IntegerInformation 代金情報 = new IntegerInformation(100);
60:
61:     //  代金の情報を送り返す
62:     this.sendInformation(BoxTownModel.INFORMATIONTYPE_PriceInformation,
63:         代金情報);
64:   }
65:
66:   /**
67:    * greetingAction
68:    **/
69:   public void greetingAction() {
70:     String あいさつの内容 = "いらっしゃいませ";
71:
72:     //メッセージを送る
73:     MessageInformation 返事の Information = new MessageInformation(あいさつの内容);
74:     this.sendInformation(返事の Information);
75:   }
76:
77:   /**
78:    * receivingOrderAction
79:    **/
80:   public void receivingOrderAction() {
81:
82:     //受け取った注文の情報を記憶する
```

```
 83:     Information 受け取った Information = this.getReceivedInformation();
 84:     this.getAgent()
 85:         .putInformation(BoxTownModel.INFORMATIONTYPE_OrderInformation,
 86:             受け取った Information);
 87:   }
 88:
 89:   /**
 90:    * sayingPriceAction
 91:    **/
 92:   public void sayingPriceAction() {
 93:
 94:     //価格の情報をつくる
 95:     IntegerInformation 価格情報 = new IntegerInformation(100);
 96:
 97:     //価格の情報を送り返す
 98:     this.sendInformation(BoxTownModel.INFORMATIONTYPE_PriceInformation,
 99:         価格情報);
100:
101:     //価格の情報を記憶する
102:     this.getAgent().putInformation(
103:         BoxTownModel.INFORMATIONTYPE_PriceInformation, 価格情報);
104:   }
105:
106:   /**
107:    * receivingMoneyAction
108:    **/
109:   public void receivingMoneyAction() {
110:
111:     //受け取ったお金を持つ
112:     Goods 受け取った Goods = this.getReceivedGoods();
113:     this.getAgent().addGoods(受け取った Goods);
114:   }
115:
116:   /**
117:    * givingBreadAction
118:    **/
119:   public void givingBreadAction() {
120:
121:     //注文量を調べる
122:
123:     //  注文の情報を取り出す
124:     IntegerInformation 直近の注文情報 = (IntegerInformation) this.getAgent()
125:         .getInformation(BoxTownModel.INFORMATIONTYPE_OrderInformation);
126:
127:     //  注文の情報から値を取り出す
128:     int 直近の注文量 = 直近の注文情報.getValue();
129:
130:     //パンを送り返す
131:
132:     //  パンを取り出す
133:     Goods 取り出したパン = this.getAgent().removeGoods(
134:         BoxTownModel.GOODSTYPE_BreadGoods, 直近の注文量);
```

```
135:
136:     //  パンを送り返す
137:     this.sendGoods(取り出したパン);
138:   }
139:
140:   /**
141:    * isOrderInformation
142:    **/
143:   public boolean isOrderInformation(Event e) {
144:
145:     //最後に受け取った Information が注文の情報か調べる
146:     boolean 注文かどうか = this
147:         .receivedInformationEquals(BoxTownModel.INFORMATIONTYPE_OrderInformation);
148:     return 注文かどうか;
149:   }
150:
151:   /**
152:    * isMoneyGoods
153:    **/
154:   public boolean isMoneyGoods(Event e1) {
155:
156:     //最後に受け取った Goods がお金か調べる
157:     boolean お金かどうか = this
158:         .receivedGoodsEquals(BoxTownModel.GOODSTYPE_MoneyGoods);
159:     return お金かどうか;
160:   }
161:
162: }
```

(4) World の初期設定と CB が生成するソースコード

パン屋モデルの World の初期設定では，Agent の初期配置と，初期状態での Relation の構造を設定する必要がある (図 B.23)．ここでは，Agent の初期配置は BakerAgent が 1 人と CustomerAgent が 3 人にしてある．それぞれの Agent は，パンかお金を最初から一定量持つように設定されている．また，全ての CustomerAgent は，BakerAgent に向かって FavoriteShopRelation を持つよう設定されている．

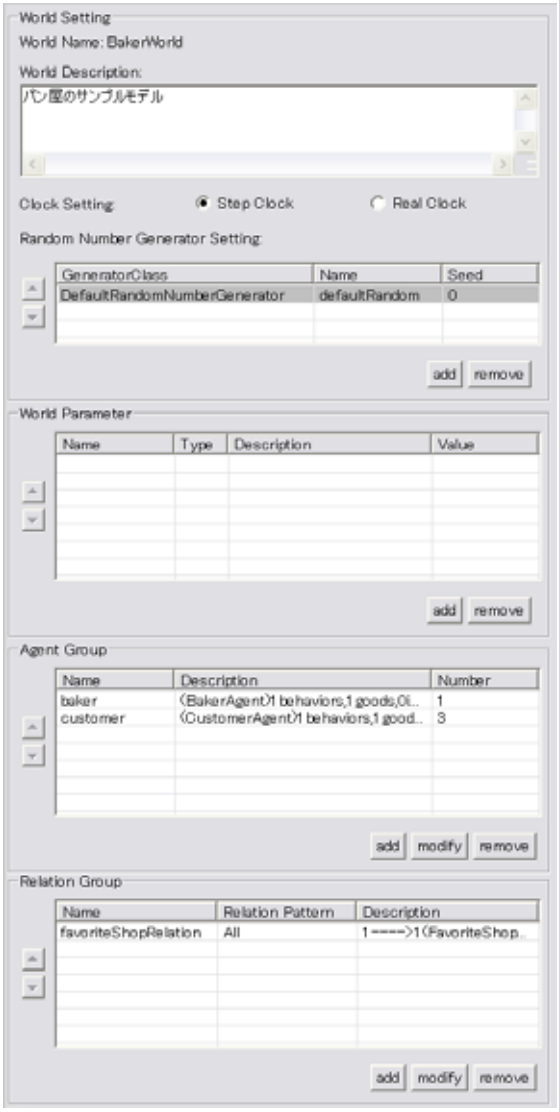


図 B.23: World の初期設定

リスト 6: CB が生成する World のソースコード

```
 1: /*
 2:  * Created on 2004/11/22
 3:  * (Component Builderによって生成されたソースコード)
 4:  */
 5: package baker;
 6:
 7: import ･･･(略)
 8:
 9: /**
10:  * BakerWorld
11:  */
12: public class BakerWorld extends World {
13:
14:   /************************************
15:    * Class variable and main method for running on the besp
16:    ************************************/
17:   private static final long serialVersionUID = 1L;
18:
19:   private static final Logger logger = Logger.getLogger(BakerWorld.class
20:       .getName());
21:
22:   public static void main(String[] args) {
23:     BESP.main(new String[] { "-model", BakerWorld.class.getName() });
24:   }
25:
26:   /************************************
27:    * Parameters
28:    ************************************/
29:
30:   //List of agents created from AgentGroup
31:   private List bakers = new ArrayList();
32:   private List customers = new ArrayList();
33:
34:   /************************************
35:    * Initialize
36:    ************************************/
37:
38:   /**
39:    * Initialize World.
40:    *
41:    * @see org.boxed_economy.besp.model.fmfw.World#initializeWorld()
42:    */
43:   public void initializeWorld() {
44:     logger.info("Initialize World.");
45:     super.initializeWorld();
46:
47:     //set the clock
48:     this.setClock(new StepClock());
49:   }
50:
51:   /**
```

```
52:    *  Initialize Agents.
53:    *
54:    * @see org.boxed_economy.besp.model.fmfw.World#initializeAgents()
55:    */
56:   public void initializeAgents() {
57:     logger.info(" Initialize Agents.");
58:     super.initializeAgents();
59:
60:     this.createAgents();
61:     this.addRelations();
62:     this.initializeByHands();
63:   }
64:
65:   /**
66:    * this method is not overrided for automaticaly.
67:    */
68:   private void initializeByHands() {
69:   }
70:
71:   /**
72:    * Create Agents.
73:    */
74:   private void createAgents() {
75:     logger.info("Create Agents.");
76:
77:     this.createBakerAgent();
78:     this.createCustomerAgents();
79:   }
80:
81:   /**
82:    * Add Relations
83:    */
84:   private void addRelations() {
85:     logger.info("Add Relations");
86:
87:     this.addFavoriteShopRelationRelations();
88:   }
89:
90:   /************************************
91:    * Subroutines to create agents
92:    ************************************/
93:
94:   /**
95:    * Create Customer Agents
96:    */
97:   private void createCustomerAgents() {
98:     logger.info("Create Customer Agents");
99:
100:    for (int i = 0; i < 3; i++) {
101:      this.createCustomerAgent();
102:    }
103:  }
```

```
104:
105:   /**
106:    * Create Baker Agent
107:    */
108:   private void createBakerAgent() {
109:     logger.info("Create Baker Agent");
110:
111:     //create agent
112:     Agent agentBaker = super.createAgent(BoxTownModel.AGENTTYPE_BakerAgent);
113:
114:     //add behavior(s)
115:     agentBaker.addBehavior(BoxTownModel.BEHAVIORTYPE_SalesBehavior);
116:
117:     //add goods
118:     Goods goodsGOODSTYPE_BreadGoods = super.createGoods(
119:         BoxTownModel.GOODSTYPE_BreadGoods, 10000);
120:     agentBaker.addGoods(goodsGOODSTYPE_BreadGoods);
121:
122:     bakers.add(agentBaker);
123:   }
124:
125:   /**
126:    * Create Customer Agent
127:    */
128:   private void createCustomerAgent() {
129:     logger.info("Create Customer Agent");
130:
131:     //create agent
132:     Agent agentCustomer = super
133:         .createAgent(BoxTownModel.AGENTTYPE_CustomerAgent);
134:
135:     //add behavior(s)
136:     agentCustomer.addBehavior(BoxTownModel.BEHAVIORTYPE_ShoppingBehavior);
137:
138:     //add goods
139:     Goods goodsGOODSTYPE_MoneyGoods = super.createGoods(
140:         BoxTownModel.GOODSTYPE_MoneyGoods, 1000000);
141:     agentCustomer.addGoods(goodsGOODSTYPE_MoneyGoods);
142:
143:     customers.add(agentCustomer);
144:   }
145:
146:   /***********************************
147:    * Subroutines to add relations
148:    ***********************************/
149:
150:   /**
151:    * add FavoriteShopRelation relations
152:    */
153:   private void addFavoriteShopRelationRelations() {
154:     logger.info("add FavoriteShopRelation relations");
155:
```

```
156:     //set the source and target agents
157:     List sourceAgents = new ArrayList(customers);
158:     List targetAgents = new ArrayList(bakers);
159:
160:     Agent targetAgent = (Agent) targetAgents.get(0);
161:
162:     //Add Relations
163:     for (int i = 0; i < sourceAgents.size(); i++) {
164:       Agent sourceAgent = (Agent) sourceAgents.get(i);
165:
166:       if (sourceAgent != targetAgent) {
167:         //add relation from source to target
168:         sourceAgent.addRelation(
169:             BoxTownModel.RELATIONTYPE_FavoriteShopRelation,
170:             targetAgent);
171:       }
172:     }
173:   }
174:
175:   /***********************************
176:    * Name and Desctription for World
177:    ***********************************/
178:
179:   /**
180:    * Returns the description of World.
181:    *
182:    * @see org.boxed_economy.besp.model.fmfw.World#getDescription()
183:    */
184:   public String getName() {
185:     return "BakerWorld";
186:   }
187:
188:   /**
189:    * Returns the name of World.
190:    *
191:    * @see org.boxed_economy.besp.model.fmfw.World#getName()
192:    */
193:   public String getDescription() {
194:     return "パン屋のサンプルモデル";
195:   }
196: }
```

付 録C　試用実験の感想

ここでは，本文中, 第 5 章の第 5.2 節で述べた試用実験における，被験者たちの感想の全文を紹介する．アルファベットで表記されている被験者の名前は，本文中の表 5.2 と対応している．なお，感想はほぼ原文のままであるが，被験者が特定できるような記述，本文中と違った意味で使われている用語などについては加筆修正を加えてある．

- 被験者 A
  - 一番感じたことは、メソッドを憶える、あるいは思い出す必要を感じなかったことである。通常 Java を打つ場合は、資料あるいは書籍などを手元において行っているが、その手間が省けることが非常に有用に感じられた。実際、Java で書いた際にはメソッドが思い出せず四苦八苦した。自分は、BEFM の基本的なメソッドは一通り書いたことがあるので、まだましであったが、初めての人には非常に苦労をするのではないかと思う。
- 被験者 B
  - アクションパーツの名前から私が想定するイメージと、青山さん[1]が抱いて作ったイメージとがなかなか一致せず、両者がうまくかみ合うまでは使いこなすのがとても難しく感じました。けれども慣れてしまえば、テンポよく進めることができました。（細かい部分は未だにわからないところがたくさんあるんですが ‥ ）
  - それでもやっぱり、アクションパーツの中身を操作途中で困ったときに、自分が何をしたかったのかわからなくなるときがあります。というのはおそらく、BEFM をしっかり把握できていないのかなぁと思ったりします。
- 被験者 C
  - 多分 2 回目はもっとずっと早くなると思います．なんとなく Action Designer の特徴をつかめてきたからこの操作はここにありそう，とか思うので．あと Java よりも頭の中で結構整理しやすかった．エラーも見やすい気がします．
- 被験者 D
  - 初めは行いたい動作[2]がどこにあるか迷いましたが，徐々に慣れました．プロ

[1] ツールの開発者である筆者のこと．
[2] アクションパーツのことだと思われる．

グラムを構造化でき，考え方，組み立て方が可視化できるので非常に助かります．Action Designer であれば，僕の心配部分である Java の文法をカバーできるなと思いました．ただ，苦手な変数の扱い方には苦労したので練習を積み重ねたいなと思っています．

- 被験者 E
  - これをするために，次に○○をする，というのがわかりやすい気がしました．これを使ったとき，最初は慣れなくて，どこにアクションパーツがあるのかどんなアクションパーツがあるのかがわからなくて戸惑いました．でも慣れるとよさがわかりました．
- 被験者 F
  - Reference がなくても，あまり不安なくかけるのがうれしい．作りたいものを整理するのが重要だと思った．
- 被験者 G
  - アクションパーツをキーワード検索したいです．ブロックごとにアクションをコピーペーストしたい．アクションパーツのメニューを絵にしてほしい．
- 被験者 H
  - まずはアクションパーツがある場所がわからずに苦労しました．どこを自分で変更する必要があるのかが，慣れるまで大変でした．ですが，覚えたあとは選択するだけで進めることができるので楽に感じました．
- 被験者 I
  - まだプログラムで書くことに慣れがあるのでアクションパーツの選択にまどろっこしさを感じる部分もありますが，構造的にプログラムを書くことや，バグを減らせる利点を考えるとすごくありがたみがあるなと思います．モデル作成をトータルでみれば，Action Designer があるほうが早くなると思います．一方で膨大な選択肢の中から 1 つのアクションパーツを選び出すのは大変だなと思います．チュートリアルなどを作って選び方のコツをナビゲートするような説明があるとみんながすぐ使えるようになると思います．