第5章　シミュレーション・プラットフォームの提案

5.1　シミュレーション・プラットフォームとは

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

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

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

5.1.1　研究プロセスを一貫して支援する統合環境の提供

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

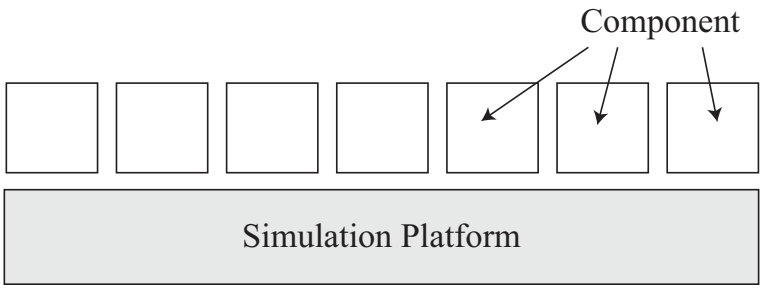

図 5.1: シミュレーション・プラットフォームの基本構造

5.1.2 モデル部品の再利用と並行開発を支援する仕組みの提供

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

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

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

5.1.3 シミュレーション環境の再利用と拡張を支援する仕組みの提供

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

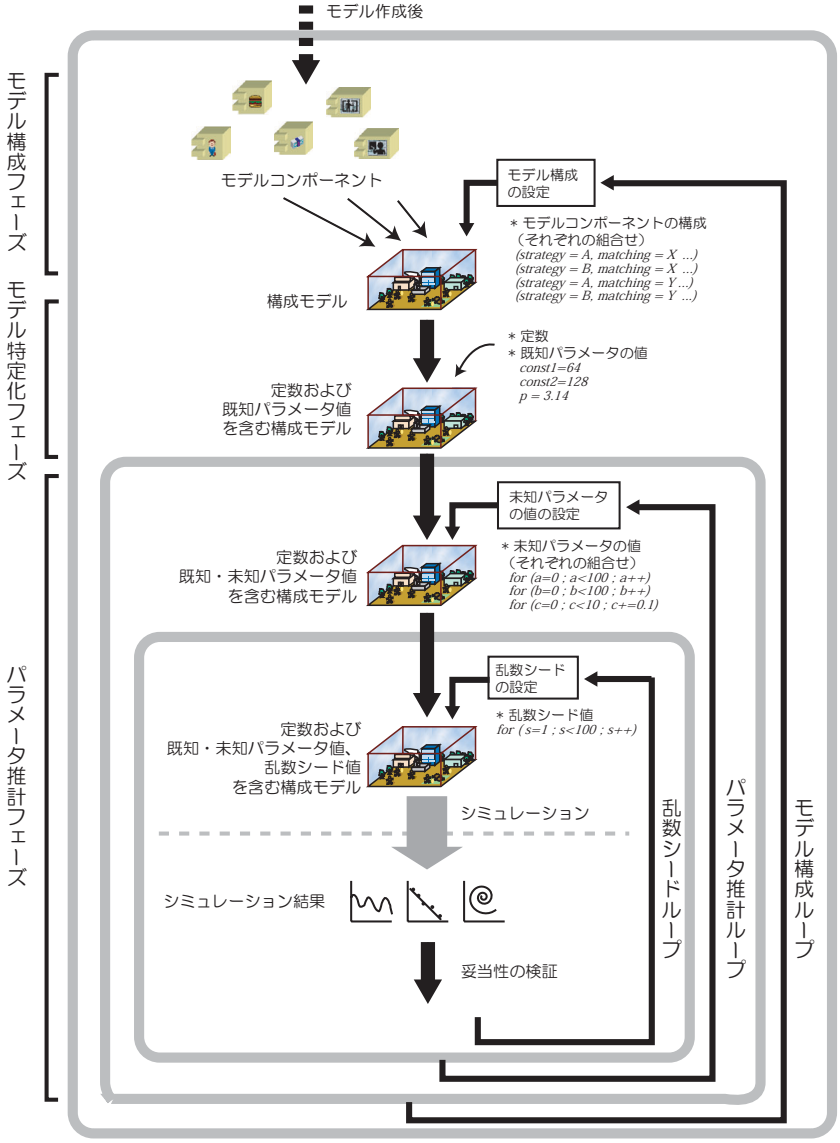

図 5.2: 探索的モデルビルディング (Iba et al., 2000)

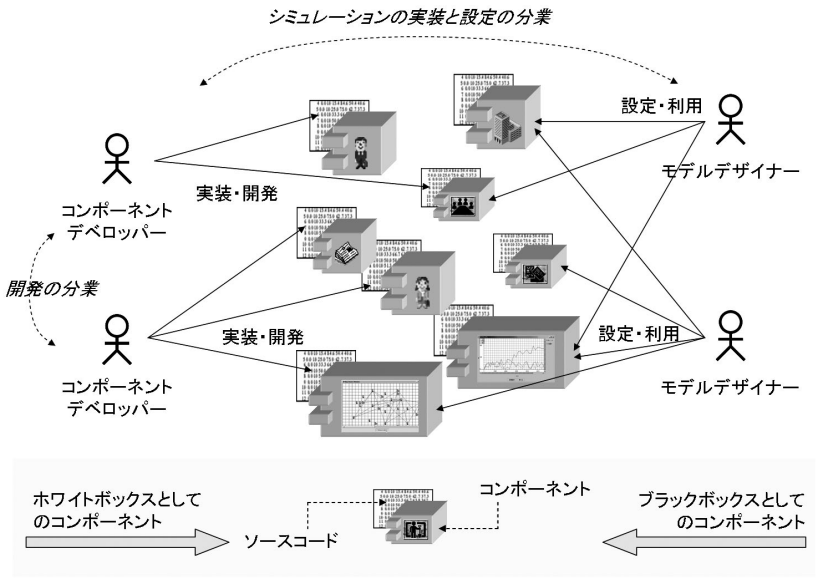

図 5.3: コンポーネントによる実装と設定の分離

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

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

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

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

5.2.1 基本アーキテクチャ

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

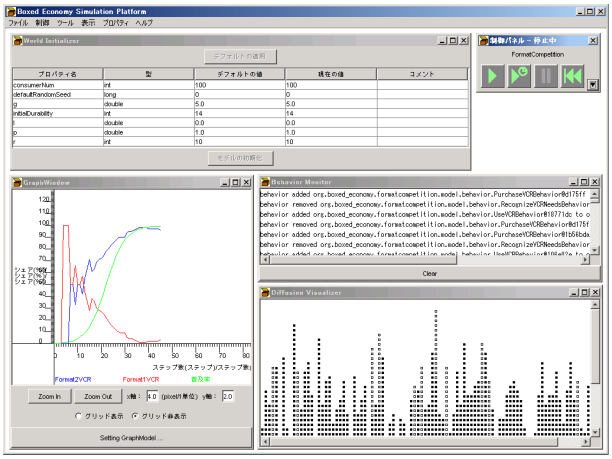

図 5.4: Boxed Economy Simulation Platform (BESP)

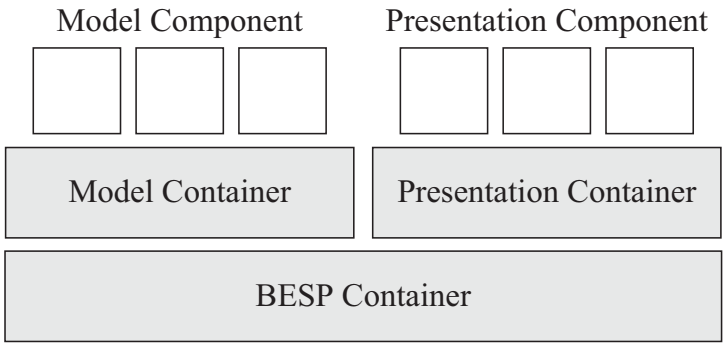

図 5.5: BESP の内部構造

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

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

モデルコンポーネント

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

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

プレゼンテーションコンポーネント

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

モデルコンテナ

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

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

プレゼンテーションコンテナ

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

図 5.6: Control Panel プレゼンテーションコンポーネント

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

BESP コンテナ

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

5.2.2 提供されるプレゼンテーションコンポーネント

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

Control Panel

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

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

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

World Initializer

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

Data Collector

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

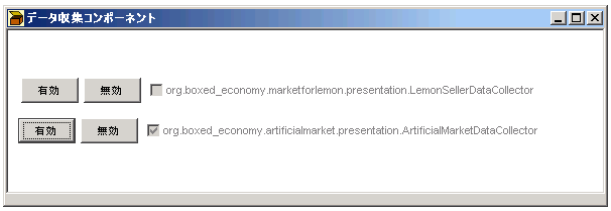

図 5.9: DataCollector プレゼンテーションコンポーネント

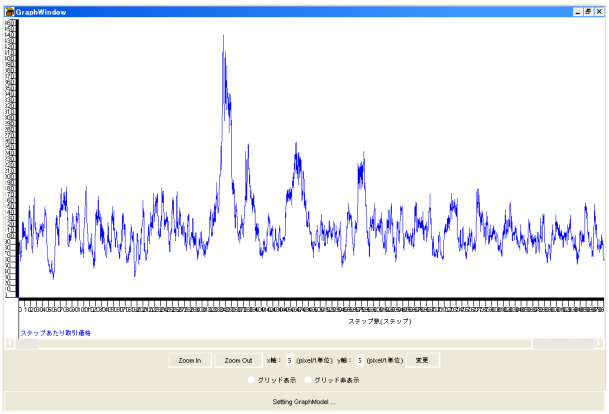

図 5.10: Graph プレゼンテーションコンポーネント

Data Collector

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

Relation Viewer

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

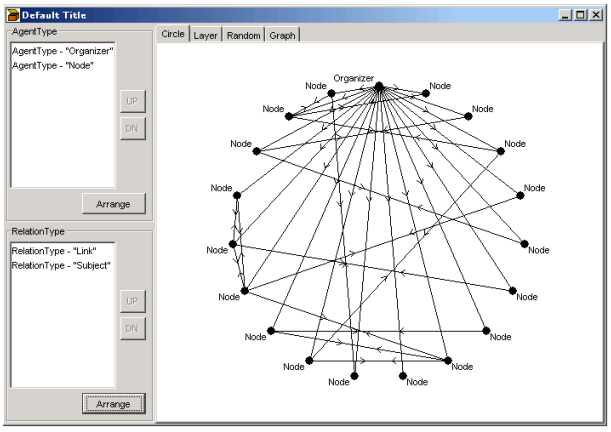

図 5.11: RelationViewer プレゼンテーションコンポーネント

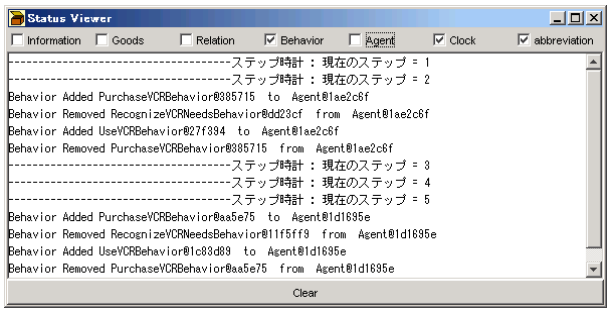

図 5.12: StatusViewer プレゼンテーションコンポーネント

Status Viewer

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

Board

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

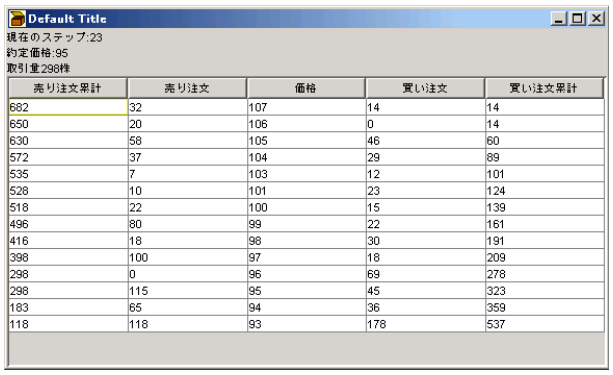

図 5.13: Board プレゼンテーションコンポーネント

5.3 提案シミュレーション・プラットフォームにおける設計と実装の支援

5.3.1 プログラミングの軽減の仕組み

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

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

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

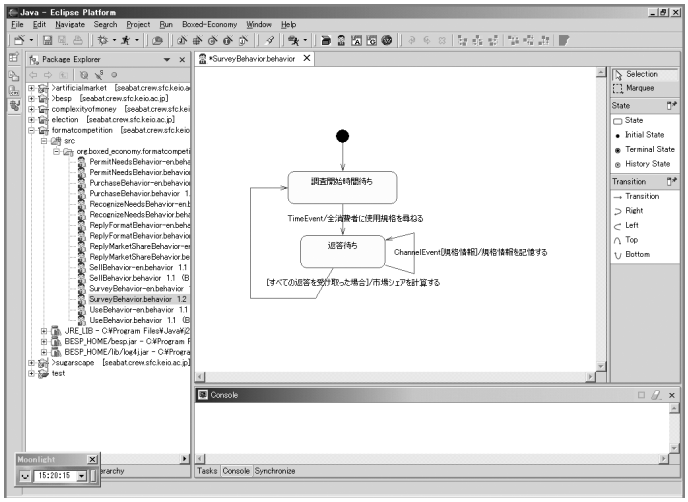

図 5.14: Component Builder

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

5.3.2 支援ツール: Component Builder

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

Behavior の作成

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

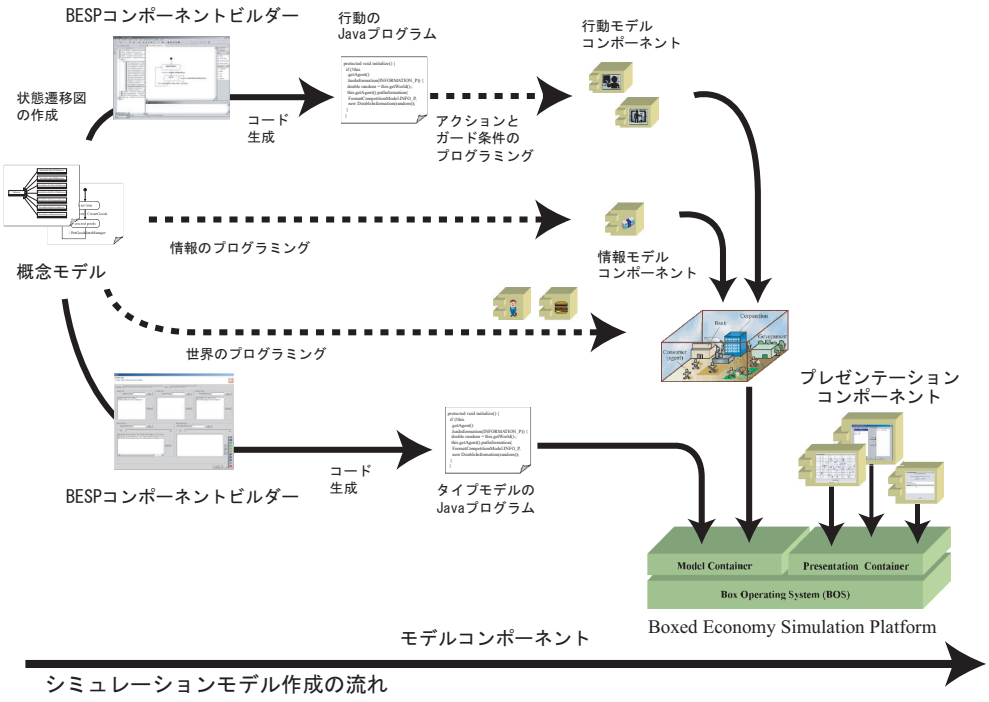

図 5.15: BESP の支援ツールを用いたシミュレーションの作成の流れ

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

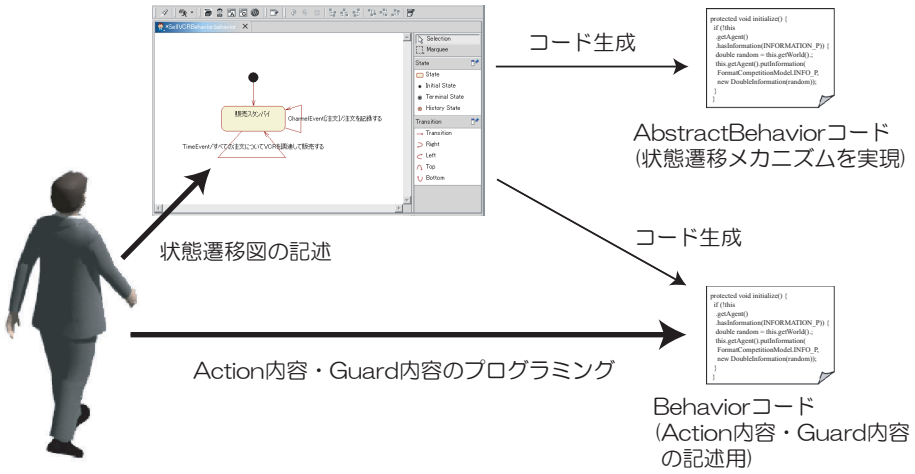

図 5.16: Component Builder を用いた行動の作成

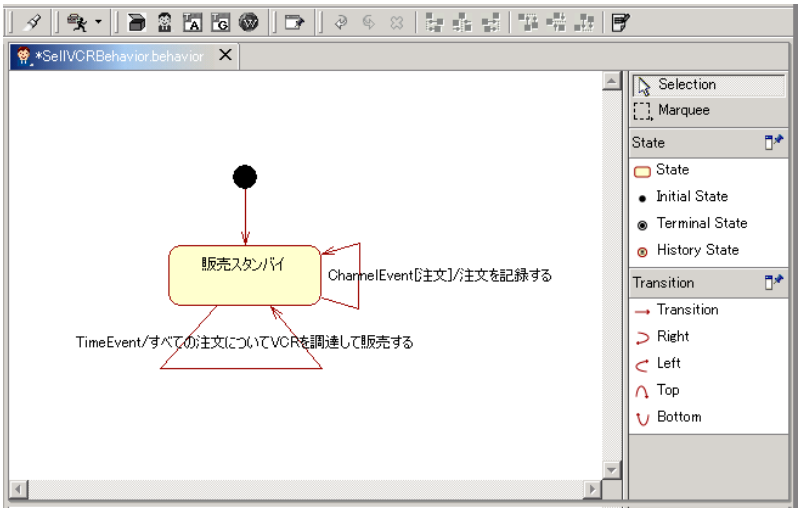

図 5.17: Component Builder 上で作成した状態遷移図

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

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

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

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

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

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

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

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

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

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

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

```
    //states setting

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

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

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

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

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

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

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

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

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

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


public class SellVCRBehavior extends AbstractSellVCRBehavior {

    private Map orders = new HashMap();

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

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

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

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

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

}
```

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

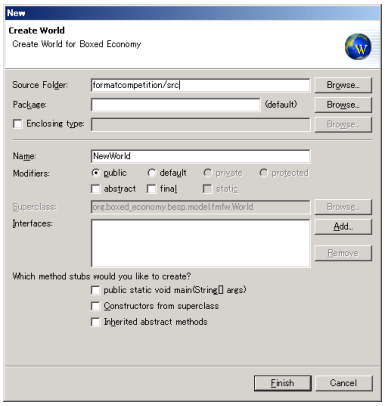

図 5.21: BESP Component Builder の World 生成ウィンドウ

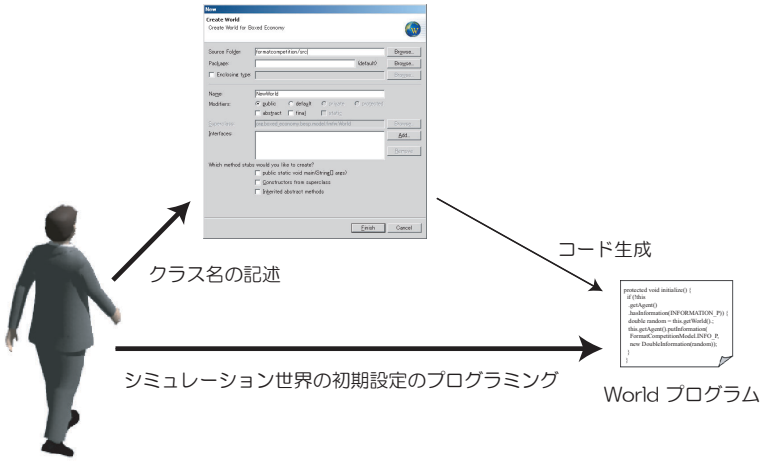

図 5.22: Component Builder を用いた世界の作成

World の作成

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

Model の作成

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

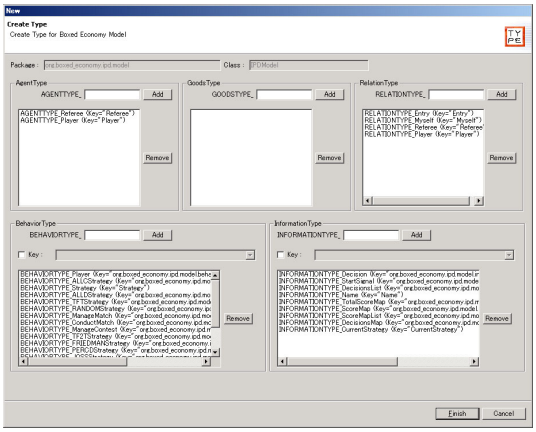

図 5.23: BESP Component Builder の Model 設定生成ウィンドウ

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

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

5.4 先行研究との比較

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

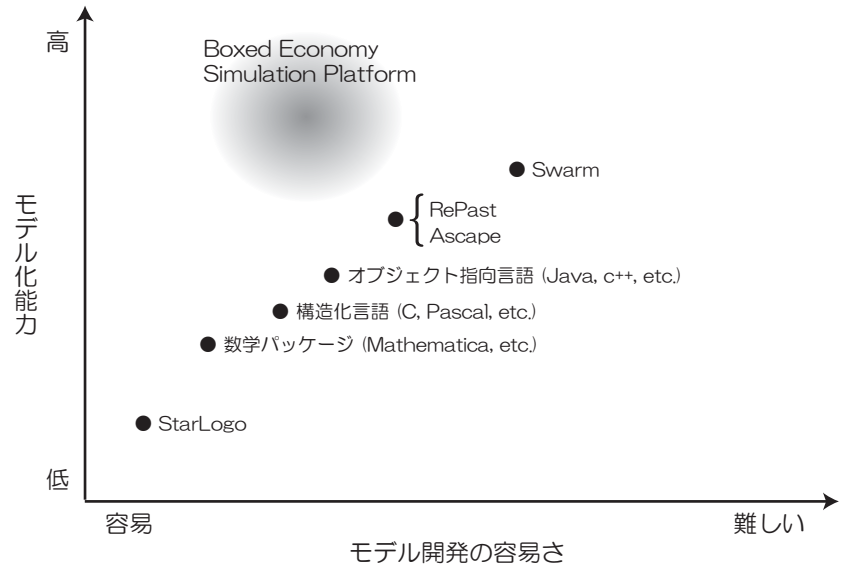

図 5.25: 既存シミュレーションシステムとの比較 (North(2002) を元に改変)

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

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

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

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

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