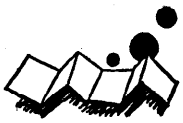
解 説

オブジェクト指向とシミュレーション†



山 本 喜 一††

1. はじめに

オブジェクト指向プログラミング（以下，OOP と略記する）とシミュレーションとは，歴史的にも密接な関連をもっていることが知られている．本稿ではまず初めに OOP とシミュレーションの関係について述べる．次に，簡単な離散型シミュレーションの OOP による実現例としてジョブショップモデルを取り上げる．その際に，Smalltalk-80 による離散型シミュレーションの実現に基づいて，その利点と欠点についても述べる．

次に，連続型シミュレーションに対する応用例をあげる．連続型シミュレーションにおいては，シミュレーションモデルおよび結果の視覚的な表現が重要視されており，OOP を利用してこのようなシステムを実現した例を報告する．シミュレーションの視覚化に関しては，離散型シミュレーションにおいても同様な試みが行われており，これからのシミュレーションの方向を示すものと考えられる．

Simula の開発経緯[8]によれば，離散型シミュレーションにおいて，対象とするシステムの構成要素を表現する手段として**クラス**を導入している．Simula におけるクラスはシステム構成要素のもつデータ構造と，その動作を記述する**動作規則** (action rule) とを合わせもった**型紙**（テンプレート）として定義され，型紙から new 文によって生成される**オブジェクト**（現在の OOP での用語では**インスタンス**一個体）が個々の構成要素となっている[16]．特に離散型システムのためのシミュレーションモデルを作るためには，システムの構成要素のそれぞれが互いに関連をもって動く様子を記述する必要があるが，各要素を一つのまとまったかたまりとしてとらえることによって，モデリングの見通しを良くすることができる．たとえば，工場には同じような機械が何台も設置されていたり，病院には数人の医師がいることが多い．このようなときには，個々の個体である機械や医師をそれぞれ別のものとして直接記述するよりも，それらが共通にもつ性質，すなわちデータ構造と動作規則，を一つのモジュールとして記述するほうが便利である．この考え方がクラスの概念の基礎になっている．

OOP におけるオブジェクトは，一般に Simula のオブジェクトよりも範囲が広く，Smalltalk-80 ではシステムの要素をすべてオブジェクトとしてとらえることから，クラスもまたオブジェクトとなっている．Actor 理論[24),25)] に基づくオブジェクトと Simula 流のオブジェクトのもう一つの相違は，前者ではメッセージのやりとりによって処理の進行を記述するのに対し，後者では手続き呼出しによってそれを記述するところにあり，前者がオブジェクト間の並列処理までを考慮しているのに対し，後者ではコルーチンによる疑似並列処理を行ってはいるけれども基本的には逐次処理を行うに過ぎない．

歴史的にみても，1967 年に発表された Simula によってクラスおよびオブジェクトの概念が取り入れられてから OOP の基本概念として使われ，現在 OOP を実現している代表的な言語である Smalltalk-80 にもそれらの概念が全面的に取り入れられている．この二つの言語にはもちろん多くの違いがあるが，クラスとオブジェクトについての基本的な概念は同じと考えてよい[20)]．また，C++[13)] はCを基礎言語とした Simula と呼んでも差し支えないほどである．

2. シミュレーションへの適用

2.1 離散型シミュレーションと OOP

離散型シミュレーションの対象となるシステムは，そのシステムの状態変化が時間軸に関して離散的であるとみなせるようなシステムである．このようなシステムにおける状態変化は，ある時刻に瞬間的に起こるものと仮定でき，このような状態変化を引き起こす出

† Object Oriented Programming and Simulation by Yoshikazu YAMAMOTO (Institute of Information Science, KEIO University).

†† 慶應義塾大学情報科学研究所

来ごとを**事象** (event) とよぶ．離散型のシステムではある事象から次の事象までの間には，システムの状態変化がないものと考える．

システムに生ずる事象に着目して，その事象によって引き起こされる状態変化と，他の事象との因果関係とを記述する事象ルーチン (event routine) によってシミュレーションモデルを作るものが SIMSCRIPT にみられる，**事象に基づくアプローチ**である[18]．このアプローチを直接 OOP に適用することはできないが，事象をオブジェクトと考えることによってモデル化することは可能である．ただし，直感的な意味での OOP の考え方とは離れるので，混乱を避けるためにここでは述べない．

いわゆるジョブショップのようなシステムでは，システムに最初から存在している機械のようなサーバと，そのシステムに外から入ってきて加工されて出ていく原料や部品のような“もの”とに分けて考えると，システムをモデル化しやすい．システム内を動き回って加工される“もの”を，トランザクションとよび，その動きの様子をモデルとして記述するものが，GPSS にみられる**トランザクションフローに基づくアプローチ**である[17]．トランザクションとサーバをそれぞれオブジェクトと考えれば，基本的には以下に述べるプロセスアプローチと同様に考えることができる．

トランザクションもサーバもシステムの構成要素であり，それらの相互作用によってシステムの状態が変わっていくと考え，構成要素のデータ構造と動作規則によってモデルを記述するのが，Simula による**プロセスアプローチ**である[17]．Simula におけるこのプロセスこそ，OOP でのオブジェクトであり，その相違は表面的には手続き呼び出しを使うかメッセージ送受を使うかだけである．

2.2 ジョブショップのシミュレーション

OOP を実現している代表的な言語である Smalltalk-80 による離散型シミュレーションの例として，**図-1** に示すジョブショップを取り上げる．

この工場には，パラメタ λ_1 の指数分布 $f(t)$ に従う時間間隔で部品が到着し，機械 A で加工される．加工の時間は，パラメタ λ_2 の指数分布 $g(t)$ に従うものとする．加工の終わった部品は工場から出ていくが，確率 p で不良品と判定され，機械 B で再処理を受ける．再処理のための時間は，パラメタ λ_3 の指数分布 $h(t)$ に従うものとする．再処理を受けた部品は，再び機械 A で加工されるものとする．機械 A と B の前

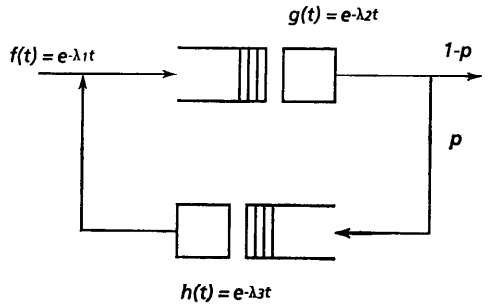


図-1 ジョブショップの例

には，それぞれ待ち行列ができるものとし，工場の外から到着する部品も，機械 B から戻ってくる部品も区別なく，到着順に機械 A の待ち行列に並ぶものとする．

Smalltalk-80 に用意されている離散型シミュレーションのための Class は，**図-2** に示すとおりであり，文献 4), 5) にそれぞれの Class のソースコードが示されてはいるが，Smalltalk-80 のシステムには含まれていないので，必要な Class を自分で入力しなければならない．

図-2 の左から，Histogram は統計値をヒストグラムとして出力するためのもの，Stream のサブクラスとして定義されている ProbabilityDistribution 以下の各 Class は確率分布関数に従う乱数を生成するためのものである．

実際にシミュレーションモデルを作るときに，利用者が考えなければならないのは，次の三つの Class Resource, SimulationObject, Simulation である．Resource はシステムの構成要素のうちいわゆる資源に相当するもので，資源の利用による構成要素の同期をとるために使われる ResourceCoordinator と，たとえば駐車場のように車種によって占有する広さが違ってもうまく扱えるように工夫した ResourceProvider とに分かれている．SimulationObject はシステムの構成要素を表すための Class で，図-1 の例の場合には工場に入って加工される部品 Parts と機械 Machine をそのサブクラスとして定義している．SimulationObject のサブクラスである EventMonitor は，事象のトレースを取るための Class である．Simulation は離散型シミュレーションを実行する主プログラムに相当する Class で，シミュレーションの初期設定から実行の制御までを行う．実際にはこの Class のサブクラスである Factory を作り，そこで

主プログラムに相当する仕事をさせている．Simulation のサブクラスである StatisticsWithSimulation は SimulationObject のインスタンスの滞留時間（シミュレーションモデル内に存在していた時間）についての統計を取るための Class である．

SimulationObjectRecord は先のインスタンスの滞留時間を求めるために使われる Class である．最後の DelayedEvent とそのサブクラスである WaitingSimulationObject は，それぞれ事象通知 (Event Notice) と資源での待ちを示すために Simulation から使われる．

これらのクラスを使えば，シミュレーションモデルを作るために利用者が実際にコーディングしなければならないのは図-2 の Factory, Machine, Parts に対応する Class だけである．それぞれの Class のプログラムを**図-3, 4, 5** に示す．Factory のクラスメソッド example が，この例題のシミュレーションを実行するためのもので，実際には最後の注釈 "Factory example" を do it すればよい．プログラムの詳細については，図中に多くの注釈を入れてあるのでここでは特に説明を加えない．このシミュレーションの事象トレースの一部を**図-6** に示し，部品の滞留時間についての統計値の出力の一部を**図-7** に示す．

Smalltalk-80 でのこれらの Class は文献 5) にもあるように G. M. Birtwhistle による DEMOS とよぶ Simula によって定義したクラスに基づいている．Birtwhistle によれば DEMOS は離散型シミュレーションをプロセスアプローチによってモデル化する際の方法を教育するために，Simula の class simulation のサブクラスとして定義したもので，Simula をそのまま使うよりは便利な機能をもたせてはいるが，汎用性という面からはやや劣っており，Simula のコードを直接書く必要もある．筆者の個人的な意見としては，DEMOS は Simula の上に作られているからこそ，必要に応じて Simula のコードを自由に書けるので不便はないが，Smalltalk-80 上の実現では対応する Simula の部分がないので，シミュレーションのためのシステムとしては力不足である．たとえば，Resource に対する割込み処理を記

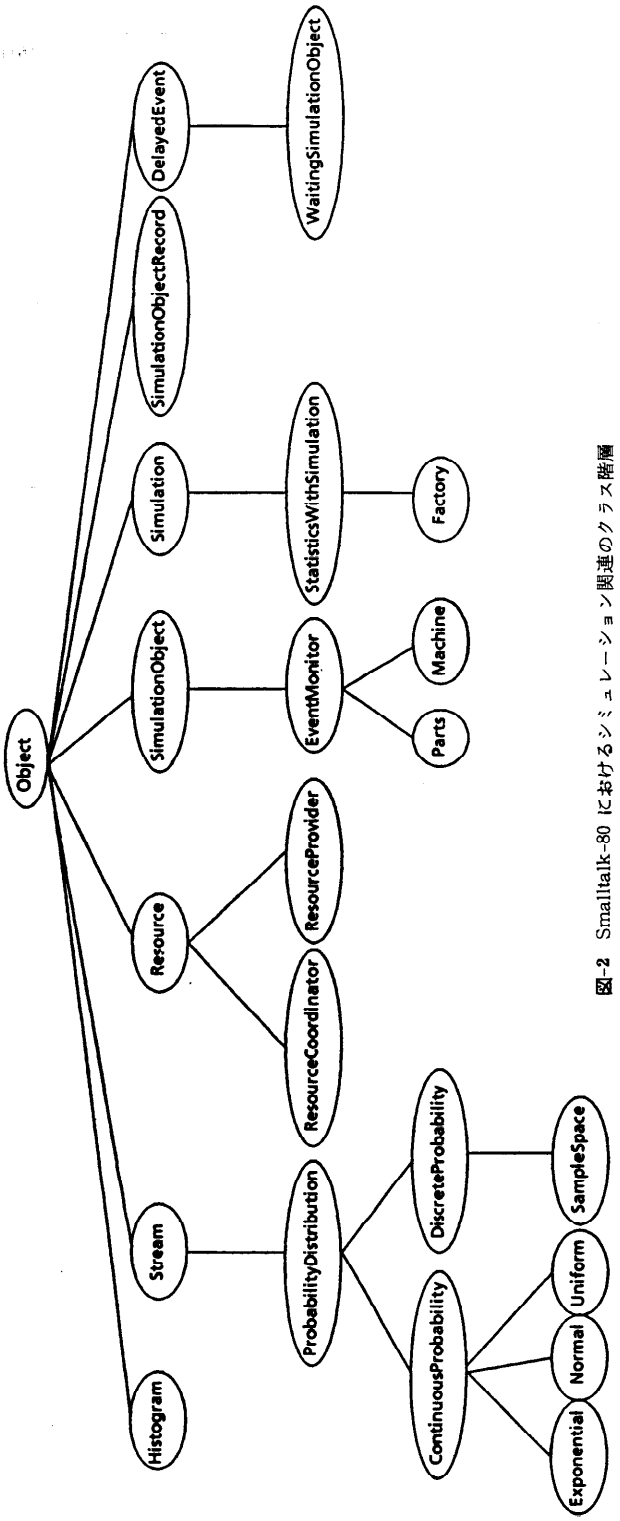


図-2 Smalltalk-80 におけるシミュレーション関連のクラス階層

```
StatisticsWithSimulation subclass: #Factory
        instanceVariableNames: 'rejectProb '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulation - Example'


Factory methodsFor: 'accessing'

rejectionProb
        "不良品確率 p を返す."
        ↑rejectProb

Factory methodsFor: 'initialization'

defineArrivalSchedule
        self scheduleArrivalOf: Parts       "平均5.0の到着時間間隔で部品を作る."
                accordingTo: (Exponential mean: 5.0).
        self scheduleArrivalOf: ((Machine name: 'MachineA')
                                                setMeanProcessingTime: 3.0)
                at: 0.0.                    "平均処理時間3.0の機械Aを作る."
        self scheduleArrivalOf: ((Machine name: 'MachineB')
                                                setMeanProcessingTime: 6.0)
                at: 0.0.                    "平均処理時間6.0の機械Bを作る."

defineResources
        "ResourceCoodinator として MAParts と MBParts を作る."
        self coordinate: 'MAParts'.
        self coordinate: 'MBParts'.

setRejectionProb: aFloat
        "不良品の確率 p を設定する."
        rejectProb ← aFloat

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Factory class
        instanceVariableNames: ''


Factory class methodsFor: 'example'

example
        "ジョブショップシミュレーションの実行."
        | aFactory aFile |
        Prob ← Random new.              "一様乱数系列を作る."
        aFile ← Disk file: 'factory.events'.        "EventMonitor のためのファイルを作る."
        Parts file: aFile.
        aFactory ← (self new setRejectionProb: 0.3) startUp.
                "不良品確率を 0.3 としてシミュレーションを開始する."
        [aFactory time < 100.0] whileTrue: [aFactory proceed].
                "時刻が 100 になるまで実行を続ける."
        aFactory printStatisticsOn: (Disk file: 'factory.data')
                "部品についての統計をファイルに書き出す."
        "Factory example"
```

図-3 Class Factory

述するには，Simulation 以下のいくつかの Class にメソッドやインスタンス変数をかなり追加しなければならない．また，統計収集のための機能もまだまだ不足で，施設の稼動率や待ち行列に関する統計を得るにはかなりのコードを加えねばならない．

2.3 連続型シミュレーションへの適用

今まで行われてきている連続型シミュレーションは，基本的には微分方程式あるいは差分方程式系によって表されたシステムを単位時間方式（微小時間 Δt 刻みで数値積分を行う）によってシミュレートするものであった．OOP を用いた連続型シミュレーションの例として最も著名なものはおそらく文献 1), 2) の制約に基づいたシミュレーションシステムであろう．

従来の連続型シミュレーションシステムや言語では，本質的にはアナログコンピュータの加算器や積分器などの結線を，サブルーチン呼び出しの連続として記述し，それに基づいて数値計算を行うものであった．これに対して，制約に基づくシステムでは，画面上に表示されたオブジェクト間の静的な制約条件を記述することによって，その制約を満足するように系を安定化するという操作をシステムが行っている．利用者としてはオブジェクト間の制約条件を記述するだけであるが，システム側にはこの制約を満足しつつ安定化するためのアルゴリズムをあらかじめ組み込んでおかねばならず，部品としてどれだけ多くのオブジェクトを用意しておくかによって，使い勝手に大きな差が出てくる．

3. シミュレーションの視覚化

シミュレーションの視覚化という側面について考えてみると，今まで特に連続型のシミュレーションにおいては，シミュレーション結果が本質的には連続量であることからその表示にさまざまな工夫がなされてきた．多くの場合には，数値として得られたシミュレーション結果を元にグラフを作り，それを印刷したり画面に表示する方法がとられていた．この場合にも，初期のものではたとえば飛行機の航跡だけを図示するというように，シミュレーション結果の抽象的な表示に過ぎなかったが，コンピュータグラフィックスの技術進歩にともなって，実際の飛行機が飛ぶ様子をできるだけ忠実に再現するというように工夫されてきている．いずれにしても，このような視覚化においては，シミュレーションの実行と結果の表示

```
EventMonitor subclass: #Machine
    instanceVariableNames: 'name meanProcessingTime '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Simulation - Example'

Machine methodsFor: 'simulation control'

tasks
    | partsRequest |
    [ true ] whileTrue:          "シミュレーション終了まで繰り返す."
        [partsRequest ← (self name = 'MachineA')
                                    ifTrue: [self acquireResource: 'MAParts'.]
                                    ifFalse: [self acquireResource: 'MBParts'.].
                "自分が機械Aならば ResourceCoodinator MAParts,
                そうでなければ MBParts を要求する."
        self holdFor: (Exponential mean: meanProcessingTime) next.
                "指数分布に従う処理時間だけ休止する."
        self resume: partsRequest          "部品の実行を再開する."
        ]

Machine methodsFor: 'accessing'

name
    "自分の名前を返す."
    ↑name

Machine methodsFor: 'initialization'

setMeanProcessingTime: aFloat
    "平均処理時間を設定する."
    meanProcessingTime ← aFloat

setName: aName
    name ← aName

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Machine class
    instanceVariableNames: ''

Machine class methodsFor: 'instance creation'

name: aName
    "引数で与えた名前の機械を作る."
    | aMachine |
    aMachine ← super new.
    aMachine setName: aName.
    ↑aMachine
```

図-4 Class Machine

は別のフェーズと考えられており，表示のためだけのシステムを独立に作り上げる方法が普通である．

一方，OOP を用いたシステムでは，ThingLab[1] をはじめとしてモデルの作成段階から**視覚的なオブジェクト**（ビジュアルオブジェクト）あるいは**アイコン**を使って，シミュレーションの実行過程や結果をそのまま画面上で見られるようにしたものが多いが，シミュレーションの結果としてのオブジェクトの挙動をアニメーションとして表示するシステムも報告されている[15]．シミュレーションの重要な目的の一つにシステムの挙動の観察があげられるが，このためにはシステムの動く様子が実際に画面上で見られることが教育上からも有用であり，これからも多くのシステムが作られることが期待される．

視覚化のアイデアと問題点については文献 22) に述べられているが，実際に実現されている例としてハードウェアのシミュレーションを視覚的に行うことができる杉本，阿部によるシステム[14]がある．また，物理現象（特に，力学系）のシミュレーションを行う例として Smith によるシステム[11),12)]がある．このシステムでは，ボールを初速 v で投げ上げたときのバウンドの様子を，反発係数を変えて見比べたり，壁の間を跳ね返らせて摩擦による減衰を観察できるようにするなどの工夫がみられる．これらの例では，物理や電気の実験室の環境を画面上に実現する方向でシステムが作られている．その際，実験室に置かれているバネやボール，電池，電流計などの道具や部品をそれぞれオブジェクトとして表現し，そのオブジェクトと視覚的な要素（画面に表示されている視覚オブジェクト）とを最初から結びつけて実現している．このようなバインディングはまさに OOP の得意とするところであり，視覚オブジェクトを物理的な“もの”と一体化して考えることによって，モジュラリティの高いシステムを実現することができる[9]．さらに，最近のシステムでは視覚オブジェクトをいかに本物らしく見せるかにも注意が払われており，この点からもシミュレーションによるアニメーションシステムと言っても過言ではなくなりつつある．

離散型シミュレーションに関しては前述のような部品化が簡単にはできないことから，ある程度の汎用性をもったシミュレーションシステムを作り上げることが難しい．OOP ではないが，ジョブショップモデルに関しては SIMSCRIPT II.5 によって実現したグラフィックモデリングツールが市販されている．汎用性を狙ったものとしては，文献 6),7) および 19),21) の

```
EventMonitor subclass: #Parts
        instanceVariableNames: 'currentMachine '
        classVariableNames: 'PartsCounter '
        poolDictionaries: ''
        category: 'Simulation - Example'


Parts methodsFor: 'simulation control'

moveToNormal
        "処理すべき機械をAとする."
        currentMachine ← 'MachineA'

moveToRepair
        "処理すべき機械をBとする."
        currentMachine ← 'MachineB'

tasks
        |completed|
        completed ← false.
        [completed] whileFalse:  "完成するまで以下を繰り返す."
                [self produceResource: 'MAParts'.       "機械Aでの処理を要求する."
                (completed ← self passed)               "良品ならば終わり."
                        ifFalse: [self moveToRepair.            "次の処理は機械B."
                                        self produceResource: 'MBParts'.        "機械Bでの処理を要求する."
                                        self moveToNormal               "次の処理は機械A."
                                        ]
                ]

Parts methodsFor: 'accessing'

passed
        "良品ならば true を返す."
        ↑Simulation active rejectionProb < Prob next

setLabel
        "連番を1進め, トレースに使う."
        PartsCounter ← PartsCounter + 1.
        label ← PartsCounter printString

Parts methodsFor: 'initialization'

initialize
        "最初にサービスを受ける機械をAとする."
        super initialize.
        currentMachine ← 'MachineA'

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Parts class
        instanceVariableNames: ''


Parts class methodsFor: 'initialization'

file: aFile
        "Event trace のためのファイルを作り, 連番を0にする."
        super file: aFile.
        PartsCounter ← 0
```

図-5 Class Parts

システムがあるが，対象システムの動作をアニメーションとして見るという点においては，連続型のシステムにみられるほどの迫力がない．OOP の思想を忠実に実現しようとしたならば，ある程度用途を限ったシミュレーションシステムを作り上げる方向のほうが，将来的には有望であろう．

4. おわりに

OOP によって現実の世界に存在しているさまざまな“もの”が，シミュレーションの世界のオブジェクトとして素直に表されることを示してきた．この意味ではオブジェクト指向言語のモジュラリティの高さと，クラスの継承の機構が強力な助けとなっている．

```
0.0         Parts 1 enters
0.0         Parts 1 wants to get service as MAParts
0.0         Machine 1 enters
0.0         Machine 1 wants to serve for MAParts
0.0         Machine 1 can serve Parts 1
0.0         Machine 1 holds for 0.831237
0.0         Machine 2 enters
0.0         Machine 2 wants to serve for MBParts
0.831237    Machine 1 resumes Parts 1
0.831237    Machine 1 wants to serve for MAParts
0.831237    Parts 1 exits
6.73078     Parts 2 enters
6.73078     Parts 2 wants to get service as MAParts
6.73078     Machine 1 can serve Parts 2
6.73078     Machine 1 holds for 4.81581
11.5466     Machine 1 resumes Parts 2
11.5466     Machine 1 wants to serve for MAParts
11.5466     Parts 2 exits
12.9224     Parts 3 enters
12.9224     Parts 3 wants to get service as MAParts
12.9224     Machine 1 can serve Parts 3
12.9224     Machine 1 holds for 0.244496
13.1669     Machine 1 resumes Parts 3
13.1669     Machine 1 wants to serve for MAParts
13.1669     Parts 3 exits
```

図-6 ジョブショップモデルの事象トレースの一部

Object	Entrance Time	Duration
Parts 1	0.0 0.831237	
Machine 1	0.0 nil	
Machine 2	0.0 nil	
Parts 2	6.73078	4.81581
Parts 3	12.9224	0.244496
Parts 4	22.7682	2.09813
Parts 5	27.6365	9.46214
Parts 6	27.9888	5.00002
Parts 7	31.8223	7.71498

図-7 ジョブショップモデルの滞留時間統計の一部

シミュレーションにおいては，モデルの作成からデバッグ，実行，結果の解析と評価にいたるまでのすべての段階において，人間とのインタラクションが重要である．この点については現在のオブジェクト指向言語が強力にサポートしているグラフィックスの機能がたいへん役に立っている．

シミュレーションを考えるときに忘れてはならないことは，モデルプログラムの実行効率である．本稿で取り上げた例の場合，富士ゼロックスの 1121 で約 15 秒の実行時間を要した．教科書の例題程度の問題での速度であることと，統計値の収集をほとんど行っていないことから，本当に実用的なシミュレーションを行うにはまだまだ不満がある．この点に関して，本稿では触れられなかったが，OOP のもう一つの側面である並列性について考慮することで，効率的な実行を実現できる可能性がある．Smalltalk-80 には並行処理の機能は含まれてはいないが，疑似的に並行処理を行うこともできる[3]ので，並列実行のシミュレーションは可能である．

OOP におけるオブジェクトの並列性に着目した言語として，Concurrent Smalltalk[23] や ABCL/1[10),26)] があり，これらの上にシミュレーションシステムを実現することが期待される．また，本当の意味でのシミュレーションの高速実行システムを作り上げるためには，文献 27) に述べられているような分散並列実行のためのアルゴリズムとシステムに関する研究がさらに必要であろう．

参 考 文 献

1) Borning, A.: ThingLab—An Object-Oriented System for Building Simulations Using Constraints, Proc. of IJCAI-77, pp. 497-498 (1977).
2) Borning, A.: The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory, ACM TOPLAS, Vol. 3, No. 4, pp. 353-387 (Oct. 1981).
3) 土居範久，瀬川 清：Smalltalk-80 による並行プログラミング，コンピュータソフトウェア，Vol. 3, No. 1, pp. 18-34 (Jan. 1986).
4) Goldberg, A. and Robson, D.: SMALLTALK-80 The Language and its Implementation, Addison-Wesley, p. 714 (1983).
5) Goldberg, A. and Robson, D.; 相磯秀夫監訳：SMALLTALK-80 —言語詳解—，オーム社，p. 569 (1987).
6) 松永賢次，山本喜一：Smalltalk-80 によるシミュレーションモデル構築システム，情報処理学会第 32 回全国大会講演論文集，pp. 473-474 (Mar. 1986).
7) 松永賢次，山本喜一：離散型シミュレーションのためのモデル構築・デバッギング支援システム第 2 回オブジェクト指向コンピューティングに関するワークショップ (Mar. 1986).
8) Nygaard, K. and Dahl, O.J.: The Development of the SIMULA Language, ACM SIGPLAN Notices, Vol. 13, No. 8, pp. 245-272 (Aug. 1978).
9) Pope, S.T.: Double Talk Project Documentation, Internal Report, PARC, p. 73 (May 1986).
10) Shibayama, E. and Yonezawa, A.: Distributed Computing in ABCL/1, Same as 26) C-76, p. 38 (Mar. 1986).
11) Smith, R.B.: The Alternate Reality Kit an Animated Environment for Creating Interactive Simulations, Internal Report, PARC, p. 8

(1986).
12) Smith, R. B.: Experiences with the Alternate Reality Kit an Example of the Tension between Literalism and Magic, Proc. of CHI+GI '87 (Apr. 1987).
13) Stroustrup, B.: The C++ Programming Language, Addison-Wesley, p. 327 (1986).
14) 杉本 明, 阿部 茂: オブジェクト指向言語VEGAMSによるハードウェア・ビジュアルシミュレーション, 第2回オブジェクト指向コンピューティングに関するワークショップ (Mar. 1986).
15) 内木哲也, 丸一威雄, 所真理雄: 行動シミュレーションに基づいたアニメーションシステム Paradise, コンピュータソフトウェア, Vol. 4, No. 2, pp. 24-38 (Apr. 1987).
16) 山本喜一: SIMULA, 情報処理, Vol. 22, No. 6, pp. 477-482 (June 1981).
17) 山本喜一, 浦 昭二: 一離散型シミュレーション言語の現状と将来の展望(1)—GPSSによるモデル化, 情報処理, Vol. 22, No. 9, pp. 836-845 (Sep. 1981).
18) 山本喜一, 浦 昭二: 一離散型シミュレーション言語の現状と将来の展望(2)—SIMULA, SIMSCRIPTによるモデル化, 情報処理, Vol. 22, No. 11, pp. 1012-1023 (Nov. 1981).
19) Yamamoto, Y. and Lenngren, M.: Graphic Model Building System—GMBS—, Burger, J. and Jarny Y. eds., Simulation in Engineering Science, North-Holland, pp. 121-126 (May 1983).
20) 山本喜一, 土居範久: Simula と Smalltalk-80 の比較, 鈴木則久編: オブジェクト指向, 共立出版, pp. 119-132 (1985).
21) 山本喜一: グラフィックシミュレーションシステム, 情報処理学会研究報告, Vol. 87-OS, No. 16, pp. 1-8 (Feb. 1987).
22) 横田 実: スーパーリアルシミュレーション, 第2回オブジェクト指向コンピューティングに関するワークショップ (Mar. 1986).
23) 横手靖彦, 所真理雄: 並行オブジェクト指向言語 Concurrent Smalltalk, コンピュータソフトウェア, Vol. 2, No. 4, pp. 2-18 (Oct. 1985).
24) 米澤明憲: ACTOR 理論について, 情報処理, Vol. 7, No. 20, pp. 580-589 (July 1979).
25) 米澤明憲: オブジェクト指向プログラミングについて, コンピュータソフトウェア, Vol. 1, No. 1, pp. 29-41 (Aug. 1984).
26) Yonezawa, A., Shibayama, E. et al.: Modelling and Programming in an Object Oriented Concurrent Language ABCL/1, Research Report on Information Sciences C-75, Dept. of Information Sciences, Tokyo Institute of Technology, p. 36 (Mar. 1986).
27) 吉田隆一, 所真理雄: 並列オブジェクト指向モデルの離散系シミュレーションへの応用, 第2回オブジェクト指向コンピューティングに関するワークショップ (Mar. 1986).

(昭和63年3月1日受付)