ソフトウェア工学 110－9
（1996．7．26）

オブジェクト指向言語における離散時間シミュレーションのための クラスライブラリの設計と実現

後藤邦夫† 野呂昌満††
南山大学経営学部情報管理学科

ジョブショップなどの離散時間シミュレーションの優れた記述方法として，Simula，Simscript II.5などに見られるプロセスアプローチがあり，それらの処理系では並行プロセスの記述が可能である．本論文では，Objective-Cなどの並行処理を記述できない逐次処理オブジェクト指向言語処理系において，いくつかのクラスを追加するだけで，疑似的に並行プロセス記述を可能にする方法を提案し，それらのクラスを用いたシミュレーションの具体例を示す．このアプローチの利点は，シミュレーションの記述性がシミュレーション言語に比較して劣らないこと，プロセス間の相互作用の種類の追加が容易に行なえるなど拡張性に富むこと，言語仕様の変更を要しないこと，言語のもとの機能をすべて利用できること(例えばシステムで利用できるグラフィック使用者インタフェースなどのクラスライブラリの利用が可能である)，クラスに基づく逐次処理オブジェクト指向言語に適用可能であることである．

Design and Implementation of Class Libraries in Object-oriented Programming Languages for Discrete-time Simulations

KUNIO GOTO† and MASAMI NORO††
Department of Information Systems and Quantitative Sciences, Nanzan University

This paper concerns with design and implementation issues on class libraries for discrete event simulations in an object-oriented language which does not have concurrent programming facilities. Without modification to existing language specification and its processor, the designed class libraries enable easy description of quasi-parallel processes. Compared with three possible and typical programming languages for simulation, Pascal, Simscript II.5, and C++ with task library, the implemented system is shown to be useful for writing practical simulation programs from the viewpoints of computational time, ease of programming, capability of model representation for complicated systems, interface to other library functions, and extensibility of process interaction facilities. Although we used Objective-C as a target language, our approach can be applied to any other class-based object-oriented languages.

1. はじめに

離散時間シミュレーションは，特定の出来事に注目したとき，システムの状態が離散時間で変化するとみなせる場合に用いられる．その状態変化を起こす出来事を事象(event)と呼ぶ．事象の生起時刻の系列には実データを用いるか，モデルで仮定した確率分布に従う擬乱数で人為的に発生させたものを用いる．各事象の因果関係と，各事象によって引き起こされるシステムの状態変化を表現することが，離散時間シミュレーションプログラムの記述である．また，離散時間であっても，微小時間 δt 毎の状態変化を追うことにより，連続時間シミュレーションとして用いることが可能である．また，長時間のシミュレーション実行による統計量の出力が目的である場合には事象の間の時間は省略され，これを非同期実行と呼ぶ．アニメーションにより，システムの挙動を表示することが主な目的であったりする場合は，実時間，あるいは実行速度が実時間に及ばない場合は，実時間に比例したシミュレーション実行を行なうことが望ましく，それを同期実行と呼ぶ．

手続き指向プログラミング言語や，SIMSCRIPTでは，直接，事象に基づいた記述を行なう方法をとり，事象に基づくアプローチと呼ぶが，事象の種類が多数で，因果関係が複雑な場合は，プログラムの記述が容易ではなく，また，それ故に誤りをおかしやすい．従って，実際の計算処理は，事象に基づいたものであっても，シミュレーションの記述方法としては，より人間の思考方法に近いものが望ましい．そこで，GPSSに見られるように，システム内を動き回るものを，トランザクションと呼び，その動きを記述するトランザクションフローに基づくアプローチがある．また，Simula，SIMSCRIPT II.5に見られる，システムの構成要素の動きを考え，それらのデータ構造と動作の流れをひとまとめにしたものをプロセスと呼び，プロセスの相互作用により，システムの状態が自然に変化してゆくという考えに基

† 本著者担当部分は(財)永井科学技術財団の研究助成による．
†† 両著者の本研究の一部は1995年度南山大学パッヘI-A研究奨励金の助成による．

づく記述方法はプロセスアプローチと呼ばれる．このようなプロセスは，オブジェクト指向プログラミングにおけるオブジェクトに他ならない．また，最初のオブジェクト指向言語であるSimulaがシミュレーションを目的として開発されたものであることから見ても，オブジェクト指向言語がシミュレーションを記述することに適していることは，明らかである[1)]．

GPSSは汎用とはいえ，トランザクションやサーバの細かな相互作用を記述するためには工夫が必要となる．Simscript II.5も同様であり，また，オブジェクト指向言語としての要件を十分に満たしていない．一方，並行プロセスの記述が容易であるオブジェクト指向言語として，Simula, Smalltalk-80, Modula3, C++ (taskライブラリ)などがあるが，それらの処理が可能なオブジェクト指向言語処理系が利用できない場合でも，逐次処理オブジェクト指向言語でプロセスアプローチに基づくシミュレーションが記述できる方法を本論文で提案する．具体的には，逐次処理オブジェクト指向言語の基本クラスにいくつかのクラスを追加するだけで実現する．この方法の主な利点は，

(1) 拡張した言語を用いたときのシミュレーションの記述性が既存のシミュレーション言語に比較し，劣らない．
(2) プロセス間の相互作用の種類の追加が容易に行えるなど，拡張性に富み，拡張に際して言語仕様の変更を要しない．
(3) システムで利用できるグラフィック使用者インタフェースなどのクラスライブラリの利用が可能である．
(4) ほとんどの逐次処理オブジェクト指向言語に適用可能である．

次節では，本論文で提案するシミュレーション支援システムの設計について述べ，第3節でその実現方法，4節で本システムを用いたシミュレーションの記述方法，5節では，グラフィック使用者インタフェースの利用とシミュレーションの実時間に比例した実行のための拡張，そして第6節では本システムの評価を行なう．

2. シミュレーション支援システムの設計

本節では，離散時間シミュレーション支援システムの設計について述べる．本システムの第1の目的は単一CPUでのプロセスの見かけの並行性(以後疑似並行性と呼ぶ)を実現することである．

2.1 プロセス疑似並行実行機構

プロセスを1つのオブジェクトで表現する際には，プロセス内でのシミュレーション時間経過はオブジェクト内に記述し，各プロセスが並行に動くことが望ましい．簡単な例として同じオブジェクトあるいはシミュレーション言語のルーチンで記述された2つのプロセスの実行と集中時刻管理ルーチンの管理する事象予定表の遷移を図1a, 1bに示す．

プロセス1はシミュレーション時刻0秒で生成され，処理Aを実行後，例えば乱数で生成された4秒後に処理Bを実行，更に5秒後に終了する．(時間の単位は任意であるが本論文では秒とする．)プロセス2は時刻2秒で生成され，処理A，3秒待ち，処理B，2秒待ちの後に終了する．図1aは波線で示された各プロセスでのシミュレーション時間の経過のみを示したものである．図1bでは，それに加えて太線で各プロセスにおけるA，Bの実際の処理時間と時刻管理ルーチン(Simscriptでは、翻訳時に結合されるもの．本論文で提案するObjective-Cクラスライブラリでは時刻管理オブジェクトのメソッド)内部での処理に伴う実時間の経過を示す．残る実時間の経過はプロセスと時刻管理ルーチン間の通信に伴うものであり矢印で示されている．実時間軸の細線部分では，シミュレーション時間の経過はなく，括弧内に示された一定の値をとる．

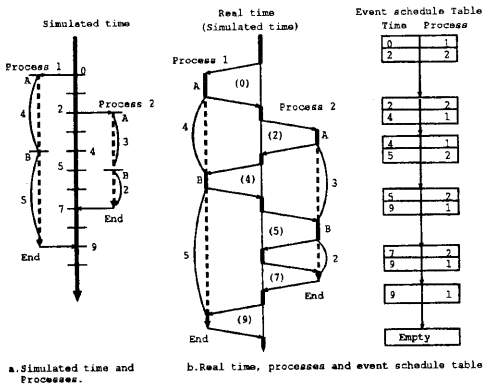

図1．プロセスの擬似並行実行例

時刻管理ルーチンでは，図1b右の表のように，システムの状態が変わる事象の予定を時刻とプロセス番号の組で表現し管理する．

次に図1の例に従いプロセス記述法を述べる．シミュレーションの時計管理，実行制御が言語の機能として組み込まれているSIMSCRIPT II.5の場合，翻訳時にそのためのライブラリルーチンが結合されるため，図2aのようにプロセスルーチン内で直接時間経過が記述できる．ところが逐次処理オブジェクト指向言語においては，あるオブジェクトから別のオブジェクトにメッセージを送ると，メッセージを受けたオブジェクトの対応するメソッドの実行が終了するまで，実行制御はメッセージを送ったオブジェクトに戻らない．すなわち1つのプロセスにメッセージを送ったオブジェクトは，そのメッセージの処理に時間がかかる場合，その実行が終るまで，他のプロセスにメッセージを送ることができないため，これらの言語を用いてプロセス指向アプローチに従ってシミュレーションを記述することは不可能である．そこで言語仕様を変更することなく上記の欠点を補うため，シミュレーションを支援するためのいくつかのクラスを定義し，図1の例に対応するプロセス1，2が図2bのクラスExampleProcessの記述で行なえるようにした．

```
Process ExampleProcess
'' Some instructions for A
 work EXPONENTIAL.F(MEAN,1) unit
'' Some instructions for B
 work EXPONENTIAL.F(MEAN,1) unit
end
```

a. Example in Simscript II.5

```
@implementation ExampleProcess
- methodA {
/*
 * Some instructions for A
 */
 [self dispatchAction: @selector(methodB)
                   in: exp_random(mean)];
        /* -dispatchAction must be at the end */
}
- methodB {
/*
 * Some instructions for B
 */
 [self dispatchAction: @selector(free)
                   in: exp_random(mean)];
}
@end
```

b. Example in Objective-C

図 2 プロセス記述の例

図 2b の例では，シミュレーション実行管理オブジェクトからメッセージを受けて，プロセスを初期化後直ちに -methodA が実行される．そして処理 A を終了した後に，(乱数の値が n であるとして)「n 秒後に -methodB の実行を予定せよ」というメッセージをそのプロセス自身に送る．スーパークラス Process から継承したメソッド -dispatchAction:in: では次の実行予定メソッドをそのプロセス (オブジェクト) のインスタンス変数に記録し，そのメソッドの実行予定時刻とプロセスの識別子をシミュレーション実行管理オブジェクトに伝え，実行制御を管理オブジェクトに戻す．メソッド -dispatchAction:in: は各メソッドの最後に書く．予定時刻になると，管理オブジェクトから，プロセスに「そのプロセスで次に予定されているメソッドを実行せよ」というメッセージが送られ，この場合は -methodB が実行され，同様にランダム時間後にプロセスの終了 (free) が予定される．

この機構では実行制御が処理の単位毎，すなわちメソッドの実行毎，に実行管理オブジェクトに戻されるため，図 1b のように複数のプロセスの並行実行が見かけ上可能となる．これを本論文では疑似並行実行と呼ぶ．この疑似並行実行機構を用いることにより，使用者 (本論文で提案するクラスを使用してシミュレーションプログラムを作成する者) は図 1a のシミュレーション時間だけを考えてプロセスを記述できる．

2.2 時刻管理機構

シミュレーション時刻管理機構には，複数の CPU を用いることを前提とする並列シミュレーションであればコンサーバティブ法[2]，バーチャルタイム法[3),4)] などを用いて実行の高速化をはかる必要があるが，本研究の目的は単一 CPU での疑似並行性を実現することであるから，単純な集中時刻管理機構を用いることとした．すなわち，プログラム実行中唯一の時刻管理オブジェクトにおいて予定時刻順に並べた事象の表を集中管理する．事象は予定時刻とメッセージを送るべきプロセスを表すオブジェクトの識別子の組で表現する．事象の表現に他の情報を含めてもよいが，ここでは最小限のものとした．時刻管理オブジェクトは，事象予定表の先頭から順に対応するプロセスにメッセージを送り，先頭の事象予定を表から削除することにより，シミュレーションを管理する．メッセージを受けとったプロセスは，そのインスタンス変数に記録された予定の処理を実行し，事象の因果関係に従い，そのプロセスに関連した次の事象予定を登録する．事象予定の追加はプロセスに用意された -dispatchAction:in: などのメソッドを用いて時刻管理オブジェクトにメッセージを送ることにより行う．なお，同時に起こる事象は先に予定表に書き込まれたものを優先して処理することとしたが，一般にはシミュレーションモデル固有のタイブレーク規則を記述する．

図 1b の例では，以下のように事象予定の追加と削除が行なわれている．

(1) **初期状態:** シミュレーション開始時に時刻 0 秒でプロセス 1 の開始，時刻 2 秒でプロセス 2 の開始を予定表に追加．

(2) **初期状態から 0 秒:** 時刻 0 秒でプロセス 1，時刻 2 秒でプロセス 2 の実行予定があるので，プロセス 1 にメッセージを送り，時刻 0 秒での予定を削除．プロセス 1 での処理 A の実行後，乱数により生成された 4 秒後での次の実行予定 (処理 B) を登録．

(3) 0 秒から 2 秒 時刻 2 秒でプロセス 2，時刻 4 秒でプロセス 1 の実行予定があるので，プロセス 2 にメッセージを送り，時刻 2 秒での予定を削除．プロセス 2 での処理 A の実行後，乱数により生成された 3 秒後，すなわち時刻 5 秒での次の実行予定 (処理 B) を登録．

(4) 2 秒から 4 秒 時刻 4 秒でプロセス 1，時刻 5 秒でプロセス 2 の実行予定があるので，プロセス 1 にメッセージを送り，時刻 4 秒での予定を削除．プロセス 1 での処理 B の実行後，乱数により生成された 5 秒後，すなわち時刻 9 秒での次の実行予定 (終了) を登録．

(5) 4 秒から 5 秒 時刻 5 秒でプロセス 2，時刻 9 秒でプロセ 1 スの実行予定があるので，プロセス 2 にメッ

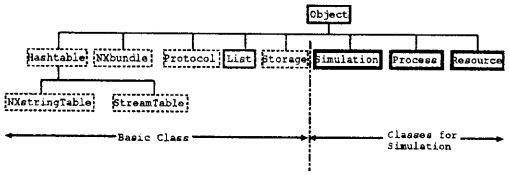

図3: クラス階層

セージを送り，時刻5秒での予定を削除．プロセス2での処理Bの実行後，乱数により生成された2秒後，すなわち時刻7秒での次の実行予定(終了)を登録．

(6) 5秒から7秒 時刻7秒でプロセス2，時刻9秒でプロセス1の実行予定があるので，プロセス2にメッセージを送り，時刻7での予定を削除．プロセス2は終了(オブジェクトが消滅)．

(7) 7秒から9秒 時刻9秒でプロセス1の実行予定があるので，プロセス1にメッセージを送り，時刻9での予定を削除．プロセス1は終了(オブジェクトが消滅)．

(8) シミュレーションの終了 事象予定表が空になったのでシミュレーションを終了する．

3. シミュレーション支援システムの実現

本節では，前節で述べたシミュレーション支援システムを実現するためのクラスライブラリの概要とシミュレーション実行制御の方法を述べる．

3.1 クラスライブラリの概要

本研究で使用したNeXTのobjective-C処理系には，Object，Listなどの基本クラス(システム定義クラス)が用意されている．それらにシミュレーション支援のために，図3に示したように前節で述べたシミュレーション実行制御，すなわち時刻管理のためのクラスSimulation，プロセス記述のためのクラスProcess，更にサービス施設の記述のためにクラスResourceの3つのクラスを追加した．クラスObjectはEiffleのAnyと同様，すべてのクラス階層の最上位のものである．基本クラスのうち，Object以外には，事象予定表の管理に便利なListクラスのみを用いた．本論文で提案するクラスライブラリの使用者はSimulation, Process, Resourceのサブクラス定義を行なうことでシミュレーションプログラムを作成する．

更にNeXTコンピュータでは，NeXTStep環境と呼ばれるグラフィック使用者インタフェース(GUI)のためのクラスライブラリが多数用意されていて，シミュレーションの視覚化には便利であるが，疑似並行実行の本質には無関係であるので，その詳細については後節で述べる．

追加した各クラスのインタフェースの概要は以下の通りである．

- **Simulation:** 生起時刻順に並べられた事象(各プロセスに対するメッセージを送る時刻)の先頭のものをとりだし，その事象の生起時刻までシミュレーション時間を進め，その事象に対応するProcessにメッセージを送る(-startSimulation).
  使用者(プログラマ)が記述するサブクラス(MySimulationと呼ぶ)においては，パラメータ入力，プログラマが定義した最初に起動すべきProcessサブクラスのインスタンスの初期化，その他の初期化を行う(-init).
- **Process:** シミュレーションにおける新たなプロセスの起動(ファクトリメソッド+activateProcessIn:by:)，インスタンスにおける処理の実行(-step)，停止(-suspendWith:)，次のメソッドの実行予定(-dispatchActionIn:in:)，再起動の要求を処理(-reactivateIn:)する．プロセス内での時間経過は次に実行される処理およびその時刻をメソッド単位で記述し，MySimulationのインスタンスにメッセージを送り予定事象として登録することによる．その処理は指定時刻にMySimulationからのメッセージにより実行される．
  使用者が記述するサブクラスにおいては，対象とするモデルにおける具体的な事象の因果関係に基づき，事象の生起，シミュレーション中での単位時間で示される待ち時間，あるいは他のプロセス，処理施設などの状態に依存する時間経過を記述する．
- **Resource:** 待ち行列におけるサービス施設と待ち行列，通信ネットワークにおけるノードなどのサービス施設と待ち行列を表現する．
  使用者が記述するサブクラスにおいては，初期化(-init)，サービス率の設定(-setServiceRate:)，サーバの数，サービス規範(FIFO,LIFO,優先権)，サービス時間(乱数)の生成方法を記述する．

単純なシミュレーションであれば，プロセスの起動と終了の機能があれば十分だが，通信ネットワークの交換機など複雑な処理を行うサービス施設は単にResourceで記述するよりProcessとして記述するほうが適切な場合があり，そのようなプロセスは処理要求がなくなったら一時停止し，処理要求が発生したときに再起動されることが望ましい．それらの場合を考慮し，Simscript II.5に見られるようにプロセスの一時停止(suspend)，再起動(reactivate)が可能になるように，SimulationおよびProcessを設計した．例えば，パケット交換機はパケットが到着するまでは休止状態(suspended)にあり，到着したパケットのプロセスから再起動(reactivate)されるようにプログラムを記述する方法が自然である．更に，必要なプロセスの相互作用があれば，適宜Simulation，Processのサブクラスに機能を追加することが可能である．

3.2 シミュレーションの実行制御

シミュレーションの実行はモニタ主プログラム(Objective-Cの関数main())でSimulationのサブクラス

MySimulationのインスタンス(時刻管理オブジェクト)を1つだけ生成，初期化し，その後は時刻管理オブジェクトから，2節で述べた予定表に従った事象処理メッセージ送出を繰り返すことにより行う．但し，時刻管理オブジェクトにおいて，メッセージ送出部分をある特定のメソッド内の単純な繰り返しとすると，シミュレーションは処理系で可能な限りの速度で実行されるため，シミュレーション実行のグラフィック表示を実時間，あるいは実時間に比例した速度で表示することによる視覚化，また，システムに用意されたグラフィック使用者インタフェースを用いて，マウスボタンを押すなどの操作でシミュレーションの実行を一時停止したり，1ステップずつ実行させたりするといった細かな実行制御ができない．そこで実時間に比例した実行，あるいはグラフィック使用者インタフェースを利用する場合には，オペレーティングシステムに用意されているタイマ機能を用いて1つの事象だけの処理を行うメソッドを定期的に呼び出すことにした．これにより，時刻管理オブジェクト(MySimulationのインスタンス)内でウィンドウサーバから受けとったグラフィック使用者インタフェースのイベントの取り込みが可能となり，キーを押したり，マウスボタンを押す等の使用者の操作の処理，実時間と比例したシミュレーション事象の処理が可能になり，シミュレーションの一時停止，1事象ごとのステップ実行などが可能になる．

4. シミュレーションの記述

本節では，前節で述べたシミュレーションのためのクラスライブラリを用いてM/M/1待ち行列が直列に2段配置されたモデル(ポアソン分布に従い客が到着し、指数分布に従う2つのサービスを先着順に受ける)を例にとり，シミュレーションプログラムの記述方法を述べる．

使用者は，時刻管理のためにMySimulationをSimulationのサブクラスとして定義する．待ち行列に到着する客を表すCustomerと客の到着を発生させるCustomerGeneratorをProcessのサブクラスとして定義し，更に待ち行列と指数サーバを表すExpServerをResourceのサブクラスとして定義する．

以下，プログラム実行の流れにそって，要点を説明する．

主プログラムでは，MySimulationのインスタンスの生成(-alloc)，初期化(-init)に続き，シミュレーションを開始(-startSimulation)する．

MySimulationのインスタンス(theSimulation)では，CustomerGeneratorプロセスの起動(-activateProcess)と到着率，シミュレーション終了予定の設定，2つのサーバの初期化とパラメータ設定を行なう．

客の挙動は，図5に示したプロセスを表現するクラスCustomerで記述する．この例では，-startでサーバのオブジェクト識別子をtheSimulationに問い合わせた後，-stage1，-server1，-stage2，-server2，-exitの順に2段のサービスを受け，システムから出てゆく．

時間経過は，1段目の待ち行列に入ってからサービスを受け始めるまで，サービスを受け始めてから1段目のサーバから出るまで，2段目についても同様で合計4箇所ある．待ち行列が空でない場合は，すぐにサービスを受けることが出来ないので，サービス要求後，プロセスを一時停止し，サーバから，サービス開始時に再起動する．再起動の際に実行されるメソッドを[self suspendWith: @selector(メソッド名)]; で指定する．また，時間t後の特定のメソッドの実行予定は，[self dispatchAction: @selector(メソッド名) in: time]; で記述する．

一方，サーバはクラスResourceで標準的な単一サー

```
/* Customer inherits from Process */
- start
{
 theServer1 = [theSimulation getServer: 1];
 theServer2 = [theSimulation getServer: 2];
       /* get instance id of two servers. */
[self stage1]; /* go to stage1 */
return self;
}
/* Behavior of Customer */
- stage1
{
   [theServer1 requestService: self];
   [self suspendWith: @selector(stage2)];
       /* suspend until becomes head of the queue
        * get service and then go to stage 2. */
 return self;
}

- stage2
{
 [theServer1 relinquish: self];
       /* release the server of stage 1. */
 [theServer2 requestService: self];
 [self suspendWith: @selector(exit)];
       /* suspend until becomes head of the queue
        * get service and then exit. */
return self;
}

- exit
{
 [theServer2 relinquish: self];
 [self free];
       /* release the server of stage 2
        * and terminate this process */
 return self;
}
```

図5 2段待ち行列シミュレーションプログラムの例

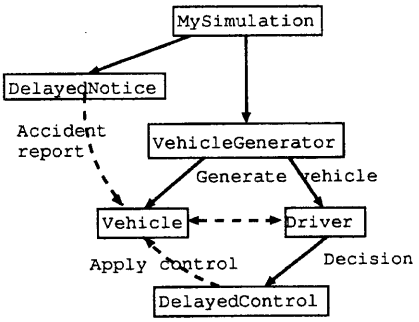

図 6: 道路交通流シミュレーションの流れ

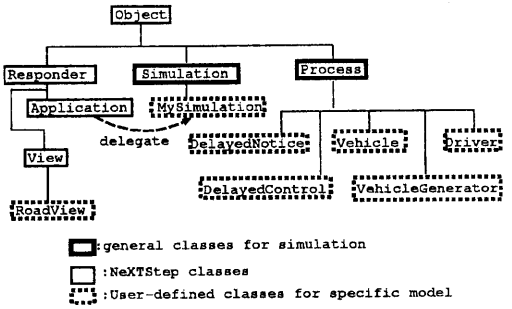

図 7: 道路交通流シミュレーションにおけるクラス階層

バ，単一待ち行列のサービス施設を定義しておき，指数分布によるサービス時間の生成部分のみをサブクラス ExpServer で定義する．この ExpServer では，サービス要求 (-requestService) があり，既にサービス中の客がなければ，すぐにサービスを要求した客に対応するプロセスを再起動 (-reactivateIn) する．また，ある客がサーバを解放 (-relinquish) した時に，列に並んでいる客があれば，その客のプロセスを再起動して，サービスを始める．各客のサービス時間は，サーバのオブジェクトで，客に対して dispatchAction メッセージを送ることにより，サーバの記述に含めることもできるが，この例では，-dispatchAction の機能を示すためにサービス時間の記述を Customer 側に含めた．

5. グラフィック使用者インタフェースの利用とシミュレーションの実時間に比例した実行

本節では，微視的道路交通流シミュレーションを例にとり，グラフィック使用者インタフェースの利用と，シミュレーションの実時間に比例した実行を可能にする方法を述べる．

図 6 のように，道路を走行する個々の運転者と自動車，その操作遅れ，自動車に装備された自動事故通報システムをプロセスで表現し，道路を走行する複数の自動車の挙動を視覚化する．

オブジェクトのクラス階層は図 7 の通りで，この例では Resource は用いない．視覚化の部分には，NeXT で用意されているライブラリクラス View を用い，そのサブクラス RoadView を定義する．各自動車の位置，速度等を微小時間 δt 毎に計算し，RoadView にメッセージを送ることにより，その位置に自動車を表す絵を描画する．これは連続系の離散時間シミュレーションによる近似である．

前節の待ち行列の例と異なるのは，プロセスの種類と相互作用が多いこと，視覚化の部分でグラフィック使用者インタフェースを用いること，実時間に比例したシミュレーション実行を行なうことである．そのため，クラス Simulation のみを拡張する．各自動車，運転者のプロセスの記述方法は，前節の例と同様である．

NeXT ではツール (InterfaceBuilder) を用いて画面上にボタンを配置し，そのボタンを使用者が押せば応用プログラムの対応するオブジェクトの対応するメソッドが起動されるように設定できる。

起動時には，システムで定義されているクラス Application の -appDidInit メソッドが最初に実行されることになっており，その -appDidInit を図 8 の MySimulation で定義したもので置き換えるため MySimulation を Application の delegate として連結する．この例では，MySimulation の -appDidInit において自動車プロセスの発生を受け持つ VehicleGenerator プロセスの起動 (-activateProcess)，シミュレーションの実時間に比例した実行モードの設定 (-setsynchronous)，実行の実時間に対する比率の設定 (-setslowdown) の後，シミュレーションを開始 (-startSimulation) する．但し，実行開始ボタンが押されるまで，実行は始まらない．描画は，各自動車のプロセスから MySimulation に要求 (-drawVehicle) があると，MySimulation から theRoadView へ要求が送られ最終的にウィンドウサーバで処理される．

Simulation において，開始 / 継続，一時停止，1 ステップ実行のボタンが押されたときに，ウィンドウサーバからボタンを押したというグラフィック使用者インタフェースのイベントがアプリケーションに送られるが，シミュレーションの事象の処理を単純にループで記述した場合，そのイベントの処理を行う時間がない．そこで，システムで用意されたタイマを用いて定期的にシミュレーション事象の処理 (メソッド -step) を行なう．これにより，シミュレーション事象の処理時間中に，ボタンが押された場合，そのボタンに対応する -start，-pause，-oneStep のいずれかのメソッドがシミュレーション事象の処理直後に実行される．この例では，DPSAddTimedEntry() という指定した関数手続きを定期的に実行する OS に依存する機能を用いたが，他の複数使用者 OS でも同様の機能を持つ関数で実現できる．

システムの処理速度が追い付く限りにおいて，実時間に比例したシミュレーションの実行を可能とするために，

```
@implementation MySimulation : Simulation
- appDidInit : (id) sender
{
[[theRoadView window] makeKeyAndOrderFront: self];
      /* Set up window for animation. */
[self init];
[self activateProcess: [VehicleGenerator class] in: 0.0];
[self setsynchronous: TRUE]; /* Synchronous mode. */
[self setslowdown: 20.0];    /* 20 times slower than real
time. */
[self startSimulation];
return self;
}
- drawVehicle: (id) vehicle at: (float) location
              oldat: (float) old_location
       /* Drawing method for a vehicle. */
{
[theRoadView moveVehicleatX : old_location/2.0 atY :
100.0
toX: location/2.0 toY: 100.0];
return self;
}
@end
```

図 8 同期実行とグラフィック使用者インタフェース利用のための変更

シミュレーション時間 (simclock) が実時間の経過に比例した予定の実行時間より遅れている間のみシミュレーション事象の処理を続けて繰り返すようメソッド step を変更する．実時間は一般に Unix で利用できるシステムコール gettimeofday() を用いて計算する．

6. 評　　価

本節では，本論文で述べたシミュレーション支援クラスオブジェクトの有用性をシミュレーション処理系，並行性の記述が可能なオブジェクト指向言語，汎用プログラミング言語を用いてシミュレーションを行なう場合と比較する．それぞれの例として順に，表 1 中の A，B，C，D の処理系を用いる．A，B，C，D はそれぞれ，本論文で提案する Objective-C の拡張 (以後，単に Objective-C と呼ぶ)，シミュレーション言語 Simscript II.5(以後、Simscript と呼ぶ)，AT&T C++ と task ライブラリ (以後，C++ と呼ぶ)，Pascal を用いる場合である．評価基準は，処理速度，シミュレーション記述の容易さ，拡張性，処理系のライブラリの利用のしやすさ，デバッグの容易さとし，比較結果を表 1 に示した．

シミュレーションの例として，前述の 2 段直列 M/M/1 待ち行列 (モデル 1) と，開始時に一定数のプロセスを起動，各プロセスが指数分布に従う間隔である仕事をする仮想的なモデル (モデル 2) を考え，コーディングの手間 (使用者が書く行数のみで既定義クラスの分は含まない) と実行時間を比較した．

表 1 4 つの処理系によるシミュレーションの比較
Table.1. Comparisons of simulations in four systems.

A. NeXT (16.3 SPEC marks) OS3.0J, memory 48MB, Objective-C with proposed classes.
B. SUN Sparc IPX compatible (24.4 SPEC marks) OS4.1.2, memory 40MB, Simscript II.5 release 1.7 beta.
C. SUN Sparc IPX compatible (24.4 SPEC marks) OS4.1.2, memory 40MB, AT&T C++ 1.0 with task library.
D. SUN Sparc IPX compatible (24.4 SPEC marks) OS4.1.2, memory 40MB, SUN Pascal-2.1.

a. SLOC and execution time with Model 1 (2-stage tandem M/M/1 queues)

	A	B	C	D
SLOC(lines)	202	31	189	425
Time(sec)	299	835	1624	89

Average number of processes = 14.5
arrival rate= 0.9, service rate 1 = 1.0,
service rate 2 = 1.1, simulated time = 1,000,000 sec.
Time is normalized to Sparc with SPEC marks ratio.

b. SLOC and execution time with Model 2 (artificial model)

	A	B	C	D
SLOC(lines)	102	36	48	284
Time(sec)				
10 Jobs	319	766	489	152
20 Jobs	850	2832	1085	411
100 Jobs	12746	10290	6859	6450

work rate = 0.5(sec), number of JOBS = 10/20/100,
simulated time = 1,000,000 sec.
Time is normalized to Sparc with SPEC marks ratio.

c. Other comparisons

	A	B	C	D
Use of system calls/libraries	easy	possible	easy	easy
Extensiveness	easy	difficult	difficult	easy
Debug easiness	easy	easy	difficult	difficult

記述の容易さ

表 1a，1b に見られるように，モデル 1 の Objective-C(A) でのソースプログラムの行数は，C++(C) とは同程度であり，Simscript II.5(B) の 6 倍程度で，事象駆動型で記述する Pascal(D) の 2 分の 1 程度である．Pascal では，モデルが複雑になるほど，事象の種類が増え，事象間の因果関係が複雑になるので，行数が増えるのみならず，記述に誤りが生じやすい．その傾向が特にモデル 1 では顕著である．モデル 2 は単純であるため C++ でも Simscript と同程度の少ない行数で表現でき，Objective-C では Simscript の 3 倍程度の行数を要するが，Pascal と比較すると 2 分の 1 以下である．本論文で述べた Objective-C による方法では，wait, delay などの時間経過をプロセスを記述するオブジェクトのメソッド内で表現できないため，どうして

もメソッドの数が増える．特にインタフェースファイル，インプリメンテーションファイルの両方でメソッド宣言が必要であることからプログラムの行数が増える．しかしメソッド宣言は定型的な記述であり，更に附属のツールを用いれば，ある程度自動生成可能であるので，読みやすさ，書きやすさからみてPascalより優れ，Simscript，C++に劣らないと言える．

実行速度

Objective-C(A)での実行時間はSPEC mark比でSPARCでの値に正規化し表に示した．すべてにおいてPascalでの実行時間がもっとも短い．モデル1(表1a)からObjective-CがPascalの次に実行時間が短く，Simscript，C++と比較して実行速度が速いことがわかる．モデル2(表1b)からは，JOB数(疑似並行実行されるプロセス数)が20以下の場合はモデル1と同様にObjective-Cでの実行速度がC++，Simscript II.5より速いが，JOB数が100と大きくなると，C++，Simscriptでの実行時間がObjective-Cのものより短くなってくることがわかる．これは，特化されたシミュレーション言語の特性，あるいは使用記憶領域が実装されている一次記憶領域を越えたため，計算機のディスク入出力能力等の差が出たものと思われる．従って，おおむねSimscriptに劣らないことから，本論文で提案したObjective-Cによる方法は処理速度の点で実用に耐え得るものであると考えられる．

定量的な評価が困難なものについては表1cにまとめ，以下で説明する．

ライブラリの利用

Objective-C，C++は，C言語の拡張であり，処理系に用意されているライブラリ，イベント，シグナルの処理機構などの，システムの資源をすべて利用できる．また，コンパイラに依存するがここでの評価に用いたPascalコンパイラでも，Cの関数を利用することが可能である．一方Simscript II.5コンパイラでもC，Fortran等の関数を用いることは可能ではあるが，X11プロトコルを用いた独自の描画ルーチンを利用することが前提となっていて，処理系のライブラリを直接用いることは困難である．従って，システムの資源をすべて利用できると言う点では，Objective-C，C++，PascalがSimscriptより優れていると言える．

拡張性

C++のtaskライブラリには，タスクの制御機構が用意されているので，拡張を必要としないかもしれないが，新しいプロセス間の相互作用を追加することはライブラリの変更なしには困難である．また，ライブラリのコードも一般には提供されていない．SIMSCRIPT II.5では，新しいプロセス間の相互作用の追加は処理系の変更なしには困難である．その点，本論文で提案した方法は，ソースコードの変更，あるいはサブクラスを定義することにより自由に拡張が可能である点が優れている．

デバッグの容易さ

評価に用いた4つのいずれにもデバッガが附属している．それを前提とすると，いずれも実行時エラーの発生箇所等は簡単に検出できるが，シミュレーションプログラムが望み通り実行されているかを確認することは，困難である．その確認の補助的な手段としては実行結果を逐次アニメーション表示することが有効であろう．その点では，Objective-CとSimscript II.5が優れている．

以上のように，シミュレーションの記述が比較的容易であること，実行速度がさほど劣らないこと，処理系のライブラリの利用が容易であること，拡張性に富むこと，アニメーションによるデバッグの補助が可能であることから，本論文で述べたObjective-C処理系は総合的に実用に耐え得るものであると言える．

7. むすび

本論文では，逐次処理オブジェクト指向言語を用いた離散時間シミュレーションのためのクラスと，それらを用いたシミュレーションの記述方法について述べた．逐次処理しかできない言語で，仮想時間での並行性の記述を可能にした．更に処理系で用意されているグラフィック使用者インタフェースの利用，実時間に比例したシミュレーション実行のための拡張を行ない，微視的道路交通シミュレーションの視覚化に十分な機能を実現した．

今後は，処理速度の向上，より多様なプロセスの相互作用の記述，グラフィック使用者インタフェースを用いたシミュレーションモデルの記述，複数CPUでの分散処理等の点で拡張を行ないたい．

参考文献

1) 山本喜一: “オブジェクト指向とシミュレーション”，情報処理, Vol. 29, No. 4 pp.374-381 (1988).
2) Misra, J.: “Distributed Discrete-Event Simulation”, ACM Comput. Surv., Vol.18, No. 1, pp.39-64 (1986).
3) Jefferson, D.R.: “Virtual Time”, ACM Trans. Prog. Lang. Syst., Vol.7, no. 3, pp.404-425(1985).
4) 松本幸則，瀧 和男: “バーチャルタイムによる並列論理シミュレーション”，情報処理学会論文誌, Vol. 33, No.3, pp.387-395 (1992).