マルチメディア通信と分散処理 41－2
（１９８９．３．１３）

並行オブジェクト指向言語を用いた待ち行列網シミュレーション

吉田 隆一
九州工業大学 情報工学部

所 真理雄
慶応義塾大学 理工学部

並行オブジェクト指向言語 ConcurrentSmalltalk の応用例として、待ち行列網シミュレーションに関して述べる。オブジェクト指向計算モデルの並行記述及び並列実行の能力を生かすことにより、待ち行列網システムに本来備わっている並列性を抽出することができる。行列網システムを構成する要素を表す各々のオブジェクトに高度の自律性を与えるため、シミュレーション時計を各々のオブジェクトに分散させた。これらの時計間の同期機構をオブジェクト間のメッセージ交換により実現した。待ち行列網シミュレーションの記述を通して、並行オブジェクト指向言語の並行記述の有効性が確かめられた。

Queueing Network Simulation Using a Concurrent Object-Oriented Language

Takaichi Yoshida,
Dept. of Artificial Intelligence,
Kyushu Institute of Technology
680-4, Kawazu, Iizuka, 820 Japan

Mario Tokoro
Dept. of Electrical Engineering,
Keio University
3-14-1, Hiyoshi, Yokohama, 223 Japan

As one of applications of a concurrent object-oriented language, ConcurrentSmalltalk, we describe queueing network simulation. We can extract parallelism in a queueing network system explicitly by making use of potencial capability for concurrent description and parallel execution of an object-oriented computing model. In order to give the great individual autonomy to each object comprising a queueing network system, simulation clocks are distributed over each object. Synchronization mechanism for those clocks is realized by passing messages. Through description of queueing network simultion, it is found that the concurrent object-oriented language is useful for concurrent description.

1. はじめに

待ち行列網システムは離散事象シミュレーションの非常に大きな応用分野である。待ち行列網システムをモデル化する方法の一つとして図1に示すようなフロー·グラフを用いて表現する方法が挙げられる[13]。例えばサービス窓口のような待ち行列網の構成要素はこのグラフのノードで表現され、窓口間の客の動きはノードからノードへ送られる事象として表現される。各ノードの機能は到着事象を受け取り、それを処理し、退去事象を生成し、その事象をリンクにより接続された他のノードに送ることである。オブジェクト指向計算モデルにおいては、ノードはオブジェクトで表現され、事象生起はメッセージ交換により表現される。

待ち行列シミュレーションを並行オブジェクト指向言語を用いて記述し、並列実行するにはこのフロー·グラフによるモデル化が以下の理由により適している。

① 現実の待ち行列網は容易にフロー·グラフに写像できる。また、プログラミングにおいては、各ノードの動作を記述するプログラムを個別に用意し、ノード間の接続を記述するだけでよい。これらはオブジェクト内に定義する。

② 多くの場合、フロー·グラフは静的である。即ち、動的なノード·オブジェクトの生成、消滅がなく、相互にメッセージを交換するオブジェクトは固定されている。従って、並列実行環境において、実行効率のよいオブジェクトのプロセッサへの割り当てが期待できる。

待ち行列網システムは本来高度の並列性を有する[14]。即ち、フロー·グラフの各ノードは、事象が到着するとこれを処理するが、それは他の全てのノードと並列に行われる。離散事象シミュレーションに対する従来のアプローチは、例えばGPSS[9]、Simscript[12]、Simula[2, 7]のようなシミュレーション言語を用いてシミュレーション·プログラムを記述し、このプログラムを汎用機上で処理していた。しかしながら、これらの言語はこの並列性を利用していない。

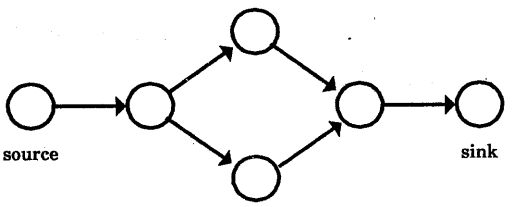

図1　フロー·グラフ

それに対して、オブジェクト指向計算モデルは並行記述と並列実行能力を潜在的に有している。シミュレーション·プログラムを並行オブジェクト指向言語を用いて記述することにより、待ち行列網モデルに固有の並列性を抽出することができる。並列実行環境においては、オブジェクトは各々の計算機に分散され、シミュレーション·プログラムの並列実行が可能となり、シミュレーションの実行時間の短縮が期待できる。

並列実行の効果を最大限に生かすため、シミュレーション時計は分散される。各ノード·オブジェクトは自分自身のシミュレーション時計を持ち、実時刻の与えられたある一時点において異なるノード·オブジェクトは異なるシミュレーション時計の値を持つことがあり得る。この方法により、ノード·オブジェクトはそのノードにおける最も早い事象時刻まで自己のシミュレーション時計を進めることができ、大きな自律性が与えられる。

同期機構は、実際のシステムに生起する事象の時刻順の系列をシミュレーション·システム内に実現するためにシミュレーション時計を管理しなければならない。この同期機構をノード間のメッセージ交換により実現する方法は[3, 4, 5, 14]などに提案されているが、我々は同期に用いられるメッセージの削減を目的とした新しい同期機構を既に[19]において提案した。同期メッセージは必要なノード·オブジェクトのみに、かつ、必要な時のみに送信されるので、メッセージ交換の頻度は少なくて済む。

本稿ではシミュレーション·システムを並行オブジェクト指向言語の一つであるConcurrentSmalltalk[18]を用いて実装する方法を述べ、シミュレーション·システムを実現するにあたり並行オブジェクト指向言語を用いることが有効であることを確かめる。

2 並行オブジェクト指向モデル

2.1 並行オブジェクト指向モデルの利点

分散処理される待ち行列網シミュレーションの記述に並行オブジェクト指向モデルを用いる利点として以下の点が挙げられる。

① シミュレーション･プログラムの記述の容易性

シミュレートされるシステムの構成要素(例えば、フロー･グラフのノード)はオブジェクトという単一の枠組みを用いて表現される。また、ノード間の相互作用である事象生起や同期操作はメッセージ交換という統一されたインタフェースにより表現できる。3章で述べる同期機構は、広域変数やブロードキャストの必要がなく、メッセージによる局所的な同期に適する。

② クラス階層と継承の利用。

例えば同期操作のような全てのノード･オブジェクトに共通の機能を上位クラスに定義し、個々のノード･オブジェクトに固有の機能を下位クラスに定義する。シミュレーション･システムは上位のクラスを用意することになる。

③ オブジェクト指向モデルに固有の並列性の利用。

オブジェクトは自己完備なモジュールであり、従って、オブジェクト指向モデルは並行記述と並列実行能力を潜在的に有する。この特徴を用いることにより、待ち行列網モデルの並列性を自然に抽出することができる。ノード･オブジェクトのようなオブジェクトは、システムにおける振る舞いのシナリオにあたる一連の動作を繰り返し実行している。例えば、各ノードは到着事象を受け取ると、それを処理し、退去事象を他のノードに送出するという一連の動作を繰り返す。この繰り返し行われる一連の動作がシミュレーションの対象となるシステムから我々が抽出できる並列性の単位となる。個々のノードの動作を個別に記述することにより、待ち行列網システム全体の並列性を表現することができる。

2.2 オブジェクト指向並行計算

Simula は最初のオブジェクト指向言語であり、離散事象シミュレーションのための言語である。

Simula においては、シナリオに当たる一連の動作を実行する process object は概念的には並列に動作し、この動作のモードは quasi-parallel[6] と呼ばれる。しかしながら、quasi-parallel の概念は co-routine の拡張概念に過ぎず、任意の時点において常に1つの process object のみが活性である。更に、プログラマは co-routine 間の制御の移動を記述しなければならない。従って、並行記述には難点があり、また、並列実行には適さない。

Smalltalk-80[8]においては、プログラム実行の制御はメッセージ交換により他のオブジェクトに移動する。換言すれば、Smalltalk-80のメッセージ交換はプロシジャ･コールと同一のセマンティクスを持つと定義でき、計算は逐次的に行われる。

Smalltalk-80を用いて並列性を有する問題を記述しようとすると、プロセスという概念が用いられる。しかしながら、この選択はオブジェクトとプロセスという2つの異なるレベルのモジュールを用いるので、問題のモデル化はプログラマにとって煩わしい作業となる。更に、プログラマは2種類の実行制御の移動、即ち、メッセージ交換とプロセス･スケジューリング、を考慮しなければならない。これは記述性および理解性を損ねる。プログラマは単一レベルのモジュールのみを用いて問題を記述すべきであり、またそのようなモジュールは必要に応じて並列に実行されるべきである。

これらの不都合は Concurrent-Smalltalk[18]、Orient84/K[16]、ABCL/1[21]、Act-1[15] などの並行オブジェクト指向言語を用いることで回避できる。このような言語において、送信側オブジェクトは、メッセージ送信後、受信側オブジェクトからの返答を待たずに、受信側オブジェクトと並列に実行を続けることができる。並列実行環境においては、問題の並列性は並列計算機の並列実行能力により自然に実行される。本稿では最も典型的なオブジェクト指向言語の一つである Smalltalk-80 と上位互換性を持つ ConcurrentSmalltalk に関して述べる。

ConcurrentSmalltalk は Smalltalk-80 に並行記述の機能を取り入れたプログラミング言語/システムである。並行記述および並行実行を実現するために、ConcurrentSmalltalk には以下の並行構文のための方式が採用されている。

① メッセージ送信後、返答を待たずに実行の継続や他のメッセージ送信ができる。ここではこれを非同期メッセージ送信と呼ぶ。

② 受信側オブジェクトは返答を返した後にも実行を継続できる。メソッドの実行途中で返される返答を acknowledge と呼ぶ。

③ 返答が必要な時にそれを待つ同期機構が導入されている。

ConcurrentSmalltalk は、アトミック･オブジェク

トとノンアトミック・オブジェクトの2種類のオブジェクトを持つ。アトミック・オブジェクトにおいては、メッセージは逐次的に受け付けられ、FIFO順に1つづつ実行される。このように、新たに到着した要求メッセージは、それ以前に到着した全ての要求メッセージが処理されるまで待たされる。これにより相互排除が実現できる。それに対して、ノンアトミック・オブジェクトは、新たな要求メッセージを受信すると、このメッセージは他のメッセージとは独立に処理される。このようにノンアトミック・オブジェクト内の幾つかのメソッドは並行に実行される。

3. シミュレーションの同期機構

各ノード・オブジェクトの自律性を最大限に引き出すために、全体時計を置かずに、シミュレーション時計をノード・オブジェクトに分散させる方式をとる。全体時計を置くと、事象の処理が全体時計に従って逐次化されてしまう。各ノード・オブジェクトの持つシミュレーション時計の値は事象生起の因果律を保証するために、単調非減少しなければならない。このシミュレーション時計を管理する同期機構が各ノード・オブジェクトに必要となる。以下にこの同期機構の概略に関して述べる。但し、ここでは、フロー・グラフに有向閉路が存在する場合、及び到着事象に割り込みがある場合は考慮しない。これらに関しては[19]を参照されたい。

到着事象を表すメッセージを"到着メッセージ"と名付ける。これはリンクで接続されたノード・オブジェクト間で授受され、到着する客を表すオブジェクトを引数として持ち、到着時刻を表す時刻印が付される。

ノード・オブジェクトは到着メッセージを受信すると、その客を待ち行列に追加する。その後この到着メッセージより過去の時刻印を持った到着メッセージの受信が無いことが確認できたならば、待ち行列から最小の到着時刻印を持つ客を選択し、その客に対してサービスを施す。客を送り出す際には、ノード・オブジェクトは到着メッセージを出力リンクにより接続されたノード・オブジェクトに送信する。

同期機構の核心となるのは、最小の到着時刻印を持つ客を選択する際に、それより過去の時刻印を持った到着メッセージの受信がないこいとを確認することである。単一の入力リンクしか持たないノード・オブジェクトは、手前のノード・オブジェクトから送信して来る到着メッセージの時刻印の系列が単調非減少している限り、到着メッセージを到着順に処理しても、事象の生起時刻に逆転は起こらない。

複数入力リンクを持つノード・オブジェクトは、異なるノード・オブジェクトから受信する到着メッセージを、それらの時刻印の順に処理を行わなければならない。このため、このようなノード・オブジェクトは最小到着時刻印を持つ客の選択に際して、到着メッセージを未だ送信していない手前ノード・オブジェクトがあるとき、それらのノード・オブジェクトの現在時刻を知るために、それらの時刻を問い合わせるメッセージを送信する。これを"時刻問い合わせメッセージ"と名付ける。このメッセージは、この時までに受信した到着メッセージの最小時刻印より過去の時刻印持つ到着メッセージの受信があるか否かの判断を行うために、手前のノード・オブジェクトに対して時刻を問い合わせるものである。このために、時刻問い合わせメッセージには、この時までに受信した到着メッセージの最小時刻印に等しい値を時刻印として与える。

時刻問い合わせメッセージを受信したノード・オブジェクトは、このメッセージの時刻印より過去の時刻印を持った到着メッセージの送信がないことが確認できたならば、この旨を返答として返す。この確認がこのノード・オブジェクト内でできなければ、さらに手前のノード・オブジェクトに対して時刻問い合わせメッセージを転送する。転送した時刻問い合わせメッセージに対する確認の返答が全て返送されると、元の時刻問い合わせメッセージの送信元へ確認の返答を返す。時刻問い合わせメッセージの送信元はこの確認の返答により、この時点での最小時刻印を持つ客の選択を行える。

シミュレーション時計をより先へ進めることと、時刻問い合わせメッセージの頻度を減少させることを目的として、時刻問い合わせメッセージの確認の返答の値に、その返答を返すノード・オブジェクが次に送信するであろう到着メッセージの時刻印の最下限を与える。この値として少なくとも現在時刻を与えることができる。この返答の値を表に保持しておく。この表を"ノード時刻表"と名付ける。この表の値は、時刻問い合わせメッセージの返答と到着メッセージの時刻印により更新し、常に知り得るかぎり最新の値を保持する。少なくともノード時刻表の時刻の最小値までは任意にシミュレーション

時計を進めすことができ、時刻問い合わせメッセージの頻度を削減できる。

4. 実装

4.1 シミュレーション·システムのクラス

シミュレーション·システムに用意されるクラスとこれらの機能について述べる。プログラマはこれらのクラスをスーパー·クラスとしてシミュレーション·プログラムを記述することができる。

① class Simulation
プログラマはこのクラスのサブクラスに、シミュレーションの対象となるシステムの全体の構造や制御、例えば、ノード間の接続やシミュレーションの開始や終了などを記述する。

② class NodeObject
プログラマはフローグラフのノードをこのクラスのサブクラスとして定義する。即ち、このサブクラスがノード·オブジェクトのクラスである。このクラスは、シミュレーション時計や到着事象の待ち行列などを表すインスタンス変数を持つ。更に、このクラスには次節で述べる同期操作を実現するメソッドが定義される。

③ class Queue
このクラスのインスタンスは各ノード·オブジェクトの持つ到着客の待ち行列を表す。このクラスのインスタンスには、客が時刻印順にソートされる。また、待ち行列長や待ち時間に関する統計が取られる。ここでの待ち行列は客の送り出し側ノード·オブジェクトと受け取り側ノード·オブジェクトで時刻が異なるため、現実の系の待ち行列を直接表現する物ではなく、むしろ分散された事象リストとしての意味合いが強い。

④ class Customer
このクラスのインスタンスは到着メッセージの引数としてノード·オブジェクト間に受け渡される。

4.2 同期機構の実現

前章で述べた同期機構は class NodeObject に記述されるが、ここでは ConcurrentSmalltalk を用いて実現する例を述べる。図2~4はその具体的な記述例である。各ノード·オブジェクトはシミュレーション時計 currentTime、到着客の待ち行列 queue、及び手前ノードの時刻を保持するノード時刻表 predecessorList を持つ。

send: aCustomer to: aNOde

客のノードからの退去を表す。
客 aCustomer を引数とした到着メッセージを行き先ノード aNode へ非同期メッセージとして送信する。即ち、

aNode arrival: aCustomer &

を実行する。ここで、& は非同期メッセージ送信を表し、返答を待たずに次の文を実行できる。

arrival: aCustomer

客のノードへの到着を表す。
到着客 aCustomer を待ち行列に追加し、ノード時刻表の送り元ノード·オブジェクトの時刻を到着客の時刻印に更新する。

(1) このノードの入力リンクが1つの場合
待ち行列から客を取り出し、シミュレーション時計を

max(現在時刻, 客の到着時刻印)

に更新し、客へのサービスを開始する。メソッド startTask: はクラス NodeObject のサブクラスに定義され、このノードの振る舞いが記述される。このシミュレーション·システムの利用者のプログラミングはここが中心となる。
客の待ち行列への追加、取り出しにより待ち行列の長さ、客の待ち時間などに関する統計を取ることができる。

(2) このノードの入力リンクが複数ある場合

(2.1) 全ての手前ノード·オブジェクトからの客が待ち行列内にあれば、(1)と同様に、待ち行列から客を取り出し、シミュレーション時計を更新し、客へのサービスを開始する。

(2.2) 待ち行列に客のない手前ノード·オブジェクトがある場合には、メソッド inquiry:time:をコールし、この時点での待ち行列の先頭の客の到着時刻印に等しい時刻印を持つ時刻問い合わせメッセージを次々に非同期送信する。このメソッドからの返値が nil でない場合、(1)と同様に、待ち行列から客を取り出し、シミュレーション時計を更新し、客へのサービスを開始する。
返値が nil の場合、客を待ち行列に残したままこの到着メッセージの処理を終わり、次の到着メッセージの受信を待つ。

timeInquiry: timeStamp

時刻問い合わせメッセージを処理する。

(1) ソース･ノードなどこのノード･オブジェクトに対して到着メッセージを送信するノード･オブジェクトのないものは、時刻問い合わせメッセージの受信に際して、その時刻印 timeStamp が自己のシミュレーション時計の値より小さくないときは、その時計の値を返値とする。さもないときは、nil を返答として返す。

(2) 一般のノードの場合、まず返値とすべき値 ackTime を求める。これにはノード時刻表の時刻の最小値と、自己のシミュレーション時計の値の小さくない方が与えられる。次にメッセージの時刻印 timeStamp と ackTime の値を比較して、ackTime の値が小さくなければ、その値を返値とする。

(3) ackTime の値が小さいときは、このノード･オブジェクトに対し到着メッセージを送るべき手前ノード･オブジェクトのなかでその時刻が、このメッセージの時刻印 timeStamp より小さなノード･オブジェクトに対して、メソッド inquiry:time: により時刻問い合わせメッセージを非同期送信を用いて転送する。inquiry:time: から nil 以外の値が返された場合、元の時刻問い合わせメッセージの送信元へ、改めて ackTime を求め確認の返答を返す。この時、返値は次の客がこのノードから退去する時の時刻を越えないようにすべきである。また、nil が返された場合、nil を返答として返す。

inquiry: aCollection time: timeStamp

ノード時刻表、またはその一部である aCollection のなかで、その時刻がこのメッセージの時刻印 timeStamp より小さなノード･オブジェクトを選び、それらへ時刻問い合わせメッセージを非同期に送信する。即ち、

```
sendToList do:
[:pred | pred name timeInquiry: timeStamp &]
```

非同期メッセージ送信を行うと、CBox オブジェクトと呼ばれるオブジェクトが返値となる。将来レシーバがこの非同期メッセージに対する返答を返

```
Object subclass: #NodeObject
    instanceVariableName: 'currentTime queue
                           predecessorList'

send: aCustomer to: aNode
    aCustomer setTime: currentTime.
    aNode arrival: aCustomer &

arrival: aCustomer
    |sendToList firstCustomer predecessor ack|
    queue add: aCustomer.
    predecessor ← predecessorList detect: [:each | each name
                                  = = aCustomer source].
    predecessor setTime: aCustomer time.
    predecessorList size = 1
        ifFalse: [  "複数入力リンクを持つ場合"
            sendToList ← self notSendPredecessor.
            "到着メッセージを送っていないノードの選択"
            sendToList notEmpty
                ifTrue: [
                    ack ← self inquiry: sendToList
                              time:aCustomer time.
                    ack isNil ifTrue: [↑nil]
                ]
        ]
    firstCustomer ← queue removeFirst.
    currentTime ← currentTime max: firstCustomer time.
    self startTask: firstCustomer
```

図2 class NodeObject の一部 (その1)

```
timeInquiry: timeStamp
    |sendToList ackTime ack|
    (predecessorList isEmpty
        ifTrue: [   "ソース･ノードの場合"
            timeStamp < = currentTime
                ifTrue: [↑currnetTime]
                ifFalse: [↑nil]
        ].
    ackTime ← predecessorList first.
    predecessorList do:
        [:pred | ackTime ← ackTime min: pred time].
    ackTime ← ackTime max: currentTime.
    timeStamp < ackTime
        ifTrue: [↑ackTime]
        ifFalse: [
            ack ← self inquiry: predecessorList
                      time: timeStamp.
            ack isNil ifTrue: [↑nil].
            ackTime ← predecessorList first.
            predecessorList do:
                [:pred | ackTime ← ackTime min: pred time].
            ackTime ← ackTime max: currentTime.
            ackTime > = arrivalQueue first time
                "返値の時刻が客の時刻印を追い越さないように"
                ifTrue: [↑ackTime]
                ifFalse: [
                    timeStamp < = currentTime
                        ifTrue: [↑currentTime]
                        ifFalse: [↑nil]
                ]
        ]
```

図3 class NodeObject の一部 (その2)

したとき、その返値はCBoxに関連付けられる。CBoxオブジェクトのメッソドblock:は返値がこれに関連付けられたとき、その引数であるブロックが評価される。これにより、このメッセージに対する返答が返される度にノード時刻表の返答元ノード･オブジェクトの時刻が返値に更新される。これらのCBoxオブジェクトはcBoxArrayに格納される。

CBoxのメソッドreceiveAndAll:はその引数となるarrayの全てのCBoxオブジェクトに返値が関連付けられるまで実行を中断する。全ての確認の返答が揃った時点で、このメッソドの実行は終了する。時刻問い合わせメッセージから1つでもnilの値を持つ返答があった場合、nilを返す。

holdFor: aTime

客へのサービスなどに際して時間を消費する。

自己のシミュレーション時計を消費時間aTimeだけ進める。他のノード･オブジェクトとは独立に実行でき同期操作は不要である。

5 まとめ

我々は[19]において、ここに述べた同期機構をSmalltalk-80を用いて実現している。並行記述、並行実行を行うため実際に次のことを行った。

```
inquiry: aCollection time: aTime
    |count sendToList cBoxArray|
    sendToList ← aCollection select: [:pred | pred time <
                                        aTime].
    sendToList notEmpty
        ifTrue: [
            count ← 0.
            cBoxArray ← Array new: 10.
            sendToList do:
                [:pred |
                count ← count + 1.
                cBoxArray at: count
                    put: ((pred name timeInquiry: aTime &)
                    block: [:ackTime | ackTime notNil ifTrue:
                    [pred setTime: ackTime]])
                ].
        (cBoxArray at: 1) receiveAndAll: cBoxdArray.
        (cBoxArray select: [cbox | cbox isNil]) notEmpty
            ifTrue: [↑ nil]
        ]

holdFor: aTime
    currentTime ← currentTime + aTime
```

図4　class NodeObjectの一部(その3)

① 1つのノード･オブジェクトに対して、到着メッセージを処理するプロセスと、時刻問い合わせメッセージを処理するプロセスを生成した。

② 上記の2つのプロセス間の同期を取るために、Semaphoreを用いた。

③ 全てのプロセスのスケジューリングのためにProcessorSchedulerを用いた。

これらの結果、同一プロセス内でのメッセージ送信による他のオブジェクトへの制御の移動が発生し、また、同一オブジェクト内のメソッドが異なるプロセスで実行される。このため、実行過程の追跡はプログラマにとって非常に困難となる。

この不都合は、受動的なオブジェクトと、そのプロシジャ(Smalltalk-80においてはメソッド)を実行するプロセスが別々に存在することによって発生する。この2つの抽象実体を用いることは極めて不自然であり、記述性、理解性を損ねる。それに対して、オブジェクトはデータ、プロシジャのみならず抽象的なプロセッサをも含んだ抽象実体である、というのが並行オブジェクト指向モデルの考え方である[17]。

本稿では、具体的な記述法に関して述べたが、ここに挙げた同期機構による同期メッセージの削減に関する簡単な評価は[19]を参照されたい。今後は更に、並列処理環境において並列処理を行った場合に、プロセッサへのオブジェクトの割り付け等を考慮し、どの程度の実行時間の短縮効果が得られるか、等の評価が必要となる。

分散型離散事象シミュレーションの記述に並行オブジェクト指向言語を用いた例には、他に[1,20]が挙げられる。これらはいずれもTime Warp[10,11]と呼ばれる機構を用いている。Time Warpにおいては、オブジェクトは到着メッセージを受信した順に処理し、もし、より過去の時刻印を持った到着メッセージを受信した場合は、その時刻までシミュレーション時計を戻し、それまでに残された副作用を元に戻す(roll back)。このため必要な履歴を全て保存しておき、既に送信したメッセージに関してはアンチ･メッセージを送信しその効果を取り消す。アンチ･メッセージを受信したオブジェクトは同様にroll backを起こし、時計を過去に戻す。

この方式においては、roll backに備えて保存されるべき過去の履歴のために記憶容量が余分に必要となる。また、いったんroll backが発生した場合のアンチ･メッセージによる将棋倒し的roll backに

かかる実行時間が余分に必要となり、取り消されたシュミレーション実行も無駄となる。これらの余分な記憶容量と実行時間、あるいは無駄な実行時間の見積りに関する評価が必要となる。

参考文献

[1] Jean Bézivin, "Some Experiments in Object-Oriented Simulation". *Proc. OOPSLA '87*, pp. 394-405, Oct., 1987.

[2] G. M. Birtwistle, O-J Dahl, B. Myhrhang and K. Nygaard, "Simula begin -2. ed.-", *Studentlitteratur*, 1980.

[3] Randal E. Bryant, "Simulation on a Distributed System", *Proc. Distributed Computing System*, pp. 544-552, 1979.

[4] K. M. Chandy and J. Misra, "A Nontrivial Example of Concurrent Processing: Distributed Simulation", *Proc. Compsac '78*, pp. 822-826, 1978.

[5] K. M. Chandy and J. Misera, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations". *Commun. ACM*, Vol. 24, No. 11, pp. 198-206, Apr., 1981.

[6] O-J Dahl and K. Nygaard, "SIMULA – an ALGOL-Based Simulation Language", *Commun. ACM*, Vol. 9, No. 9, pp. 671-678, Sep., 1966.

[7] W. R. Franta, "The Process View of Simulation". *Elsevier North-Holland. Inc.*, 1977.

[8] A. Goldberg, D. Robson, "Smalltalk-80: The Language and Its Implementation", *Addison Wesley*, 1983.

[9] "General Purpose Simulation System V user's Manual, SH20-0851-2", *IBM Corp.*, Sep., 1979.

[10] David R. Jefferson, "Virtual Time", *ACM Trans. on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404-425, Jul., 1985.

[11] David R. Jefferson, et al., "Distributed Simulation and the Time Warp Operation System", *Proc. 11th ACM Symp. on Operating Systems Principles*, pp. 77-93, Nov., 1987.

[12] P. J. Kiviat, R. Villanueva and H. M. Markowitz, "SIMSCRIPT II.5 Programming Language", *Consolidatd Analysis Centers Inc.*, 1973.

[13] Leonard Kleinrock, "Queueing Systems, Volume 1: Theory", *John Wiley & Sons, Inc.*, 1975.

[14] J. Kent Peacock, J. W. Wong and Eric Manning, "Distributed Simulation Using a Network of Processors". *Computer Networks*, Vol. 3, No. 1, pp. 44-56, Feb., 1979.

[15] D. Theriault, "A Primer for Act-1 Language", *MIT A.I. Memo*, No. 672, Apr., 1982.

[16] Mario Tokoro and Yutaka Ishikawa, "Concurrent Programming in Orient84/K: an Object-Oriented Knowledge Representation Language", *SIGPLAN Notice,s* Vol. 21, No. 10, pp. 39-48, Oct., 1986.

[17] 所真理雄, "オブジェクト指向並列計算に関する一考察", 日本ソフトウェア科学会第5回大会論文集, 1988年9月.

[18] Yasuhiko Yokote and Mario Tokoro, "The Design and Implementation of Concurrent-Smalltalk", *Proc. OOPSLA '86*. pp. 331-340, Sep., 1986.

[19] 吉田隆一, 所真理雄, "離散系シミュレーションの分散時刻管理", コンピュータソフトウェア, Vol. 4, No. 1, pp. 23-33, 1987年1月.

[20] Akinori Yonezawa, Hiroyuki Matsuda and Etsuya Shibayama, "Discrete Event Simulation Based on an Object Oriented Parallel Computing Model", *Research Reports on Information Science, Tokyo Institute of Technology*, No. C-64, Nov., 1984.

[21] Akinori Yonezawa, Jean-Pierre Briot and Etsuya Shibayama, "Object-Oriented Concurrent Programming in ABCL/1", *Proc. OOPSLA '86*, pp.258-268, Sep., 1986.