# バーチャルタイムアルゴリズムの改良†

福井 眞吾††

離散事象シミュレーションを並列に行うアルゴリズムのひとつにバーチャルタイムアルゴリズムがある．このアルゴリズムの効率を向上させる2つのアルゴリズムを提案する．バーチャルタイムアルゴリズムでは，シミュレーション対象間の同期を積極的にとることはせず，楽観的な予測に基づいて計算を進め，情報に矛盾が発生すると，矛盾が発生する以前の状態にロールバックする．したがって，効率を向上させるには「ロールバックの発生を少なくする」「ロールバックの処理コストを低くする」ことが重要である．第一の改良アルゴリズムは，発信者識別可能性と2つのオブジェクト間のメッセージ到着順序の保存性を仮定して，ロールバックする際に正しくないメッセージを一度に削除することによって，オブジェクトがありえないメッセージ系列を信じる可能性を減少させる．第二のアルゴリズムは，一度計算した結果をロールバックの際に削除せずに状態木として保存することにより，シミュレーション対象のプログラムの再実行を防止する．前者のアルゴリズムはアンチメッセージの発生を少なくすることにより，ロールバックの発生を少なくする効果を持ち，後者のアルゴリズムはロールバック発生時の再実行を防止することにより，ロールバックの処理コストを減少させる効果を持つ．提案したアルゴリズムを2つの例題に適用し有効性を確認した．

## 1. はじめに

近年，CPU やメモリなどのハードウェアコストの低下と，分散型オペレーティングシステムの実用化によって，商用レベルのマルチプロセッサ型計算機が登場するようになった．このような計算機を用いることによって処理速度の大幅な向上が望める分野の1つに離散事象シミュレーションがある．

並列処理を前提とした離散事象シミュレーション方式として，シミュレーション対象にローカルクロックを持たせ同期機構を用いて時刻を一様に保つアルゴリズムがある[3),6)]．これに対し，矛盾が発生するまで同期をとらないことにより並列性を積極的に利用するアルゴリズムが提案されている[5)]．このアルゴリズムはバーチャルタイムアルゴリズムと呼ばれる．

バーチャルタイムアルゴリズムでは，シミュレーション対象をローカルクロックを持ち互いにメッセージを交換し合うオブジェクトでモデル化する．メッセージにはタイムスタンプが付与されている．このタイムスタンプとローカルクロックを比較することで時間的矛盾を検出する．矛盾が検出されると計算がロールバックされ，オブジェクトの状態は矛盾が発生する以前の状態にもどされる．いったん送信されたメッセージを取り消すためにアンチメッセージが用いられる．したがって，バーチャルタイムアルゴリズムの処理効率を向上させるには，

- ロールバックの発生をなるべく少なくする
- ロールバックの処理コストを低くする

ことが重要である．

本論文では，まずアンチメッセージの発生原因を考察し，メッセージの到着順序と発信者識別性に関する仮定を導入し，アンチメッセージの発生可能性を低下させるアルゴリズムを提案する．次にアンチのパターンとそれに伴うプログラムの再実行を考察し，一度行った計算結果を保存することでアンチメッセージによる計算を軽減するアルゴリズムを提案する．本論文で提案する2つのアルゴリズムは互いに独立したものであり，単独で採用することも両方同時に採用することもできる．

以下，第2章ではバーチャルタイムアルゴリズムを説明する．第3章と第4章ではバーチャルタイムアルゴリズムの効率について考察し，2つの改良アルゴリズムを提案する．第5章では評価実験について述べる．第6章では提案したアルゴリズムの利点と限界について論じる．第7章ではまとめと今後の課題を示す．

## 2. バーチャルタイムアルゴリズム

バーチャルタイムアルゴリズムでは，各々のオブジェクトはローカルクロックを持ち，これが正しい時刻を示していると信じて処理を進める．処理を行った結果，内部状態の変化とメッセージ送信が発生する．各オブジェクトでの処理は同期をとらずに行われるの

† Improvement of the Virtual Time Algorithm by SHINGO FUKUI (Application System Research Laboratory, C&C Systems Research Laboratories, NEC Corporation).

†† 日本電気(株) C&C システム研究所応用システム研究部

で，オブジェクトごとに異なったローカルクロックを持つことになる．

メッセージには受信されるべき時刻がタイムスタンプとして付与されている．メッセージの発信順序と受信順序の関係について何の仮定もおいていないので，一般に発信順やタイムスタンプからメッセージの到着順序を決定することはできない．

オブジェクトはメッセージキューと状態スタックを持っている．メッセージキューには受信したメッセージがタイムスタンプの時刻順にソートされて格納される．状態スタックにはメッセージに対応する処理を実行した後のオブジェクトの内部状態がその間に送信されたメッセージのアンチメッセージとともに格納されている．**図1**の場合，このオブジェクトに送信されたメッセージは M1〜M4 の4個で，タイムスタンプの値はこの順である．S0 は状態の初期値である．メッセージ M1 を実行した結果 Ma, Mb という2つのメッセージが送信され状態は S1 に変化している．M2, S2, M3, S3 も同様である．M4 はメッセージキューには格納されているがまだ実行されていないメッセージである．

**図2**に示すようにバーチャルタイムアルゴリズムは，メッセージとアンチメッセージからメッセージキューを構成する「メッセージ管理部」と，メッセージに対応してオブジェクトのプログラムを実行する「実行管理部」の2つから構成される．

メッセージ管理部の動作は次のとおりである．メッセージが到着するとタイムスタンプに従ってメッセージキューの適切な位置に挿入する．挿入する位置より後ろのメッセージが既に実行されている場合には，アンチメッセージを送信して既に送信されたメッセージを取り消し，オブジェクトの状態を以前の状態にもどし，状態スタック内の対応する情報を廃棄する．

アンチメッセージが到着すると，該当するメッセージをメッセージキューから削除する．そのメッセージが既に実行されてしまっている場合にはロールバックが行われる．受信順序は保証されないのでメッセージよりも先にそのアンチメッセージが到着することがあるが，その場合にはアンチメッセージをメッセージキューに格納しておき，通常のメッセージが到着したときにそのアンチメッセージを削除する．

実行管理部は，メッセージキューの中で未実行かつタイムスタンプが最小のメッセージに対応するオブジェクトのプログラムを実行し，新しい内部状態と実行途中に送信されたメッセージのアンチメッセージを状態スタックに保存する．

アンチの波及効果は原因となったアンチのタイムスタンプよりさかのぼることはない．したがって連鎖反

| メッセージキュー | | M1 | M2 | M3 | M4 |
|---|---|---|---|---|---|
| 状態スタック | S0 | S1 | S2 | S3 | |
| | | $\overline{Ma}$ | $\overline{Mc}$ | $\overline{Md}$ | |
| | | $\overline{Mb}$ | | $\overline{Me}$ | |

（$\overline{\phantom{x}}$ はアンチメッセージを表す）

**図1** オブジェクトの内部データ構造例
Fig. 1 An example of the object internal data structure.

```
<メッセージ管理部>
  新メッセージの処理
    if (新メッセージのアンチメッセージがメッセージキューにある)
      then
        {アンチメッセージを削除する}
      else
        {while (新メッセージのタイムスタンプ<
                  カレントメッセージのタイムスタンプ)
           do   {ロールバック処理を呼び出す
                  カレントメッセージを1つ前のメッセージにする}
         タイムスタンプ順の正しい位置に新メッセージを挿入する}
  アンチメッセージの処理
    if (アンチ対象のメッセージがメッセージキューにない)
      then
        {アンチメッセージをメッセージキューに挿入する}
      else
        {while (アンチ対象のメッセージがカレントメッセージあるいは
                  カレントメッセージより前にあるメッセージである)
           do   {ロールバック処理を呼び出す
                  カレントメッセージを1つ前のメッセージにする}
         対応するメッセージを削除する}
  ロールバック処理
    状態スタックのトップにあるアンチメッセージを送信する
    状態スタックをポップしスタックトップの情報を破棄する
    オブジェクトの状態を状態スタックのトップにある状態にする
<実行管理部>
  メッセージの受信
    if (カレントメッセージの次のメッセージがある)
      then {カレントメッセージを1つ後ろのメッセージにする
            カレントメッセージに対応するプログラムを実行する
            送信したメッセージのアンチメッセージを
              状態スタックにプッシュする
            オブジェクトの新状態を状態スタックにプッシュする}
```

**図2** バーチャルタイムアルゴリズム
Fig. 2 The virtual time algorithm.

応的に過去のメッセージがすべてアンチされることはない．アンチが発生する時刻の下限は GVT (Global Virtual Time) で抑えられる．GVT は各オブジェクトのローカルクロックと，伝送途中および未処理のメッセージのタイムスタンプの最小値で決定される．したがって GVT は単調非減少である．GVT が進まないのは，CPU タイムを与えられないオブジェクトが存在するかあるいは発信されてから長い間相手に達しないメッセージが存在する場合なので，シミュレーションが十分長い時間実行されれば GVT は必ず進むといえる．アンチされる可能性のあるメッセージは GVT より大きなタイムスタンプ持つものだけであるので，各オブジェクトは GVT の直前以後の情報のみを保存しておけばよい．GVT を決定するアルゴリズムとしてグローバルスナップショット[2] がある．

## 3. アルゴリズムの改良—その1—

### 3.1 ロールバックの発生原因

ロールバックが発生する原因は2つある．

原因1：ローカルクロックより前のメッセージが到着する

原因2：ローカルクロックより前のメッセージがアンチされる

原因2はそのメッセージを送信したオブジェクトでロールバックが発生したことに起因するので原因1に吸収される．原因1はメッセージキューが常に正しいと信じて処理を進めることから発生する．

メッセージキューには複数のオブジェクトからのメッセージがマージされて保持されている．メッセージの受信者は発信者間のローカルクロックのずれを観測できないので，発信者が異なるメッセージの時間的ずれを受信者側で回避することは困難である．

では次に1つの発信者から来るメッセージ系列について考える．発信側オブジェクトが複数個のメッセージを送信したとする．1対1の関係でも到着順序は保証されないのでランダムな順序で到着する．すべてのメッセージが到着した段階では受信側に正しいメッセージ系列が構成されるが，途中段階では誤った系列が構成される．受信側がこの途中段階の系列に基づいて計算すると，欠落しているメッセージが到着したときにロールバックが発生しアンチメッセージが生成される．

次に，送信側でロールバックが発生した場合を考える（図3）．古いタイムスタンプを持つメッセージMの到着によって送信側の状態スタックの Sk～Sm がロールバックされると，アンチメッセージ $\overline{Mc}$, $\overline{Md}$, $\overline{Me}$ が送信される（図3送信側）．到着順序はランダムなので受信側では一部のメッセージだけがアンチされた状態が出現する（図3受信側）．これも誤った系列なのでロールバックの発生原因になる．実際にはアンチメッセージの順序が入れ替わるだけではなく，元のメッセージ Ma～Me と，そのアンチメッセージ $\overline{Mc}$～$\overline{Me}$ と，ロールバックのあとで送信側が送る新しいメッセージが順不同で到着する可能性があり，ロールバックの可能性が増大する．

### 3.2 順序保存性を利用するアルゴリズム

逐次的な手続きをプロセスやスレッドで並列に実行するタイプのプログラミング言語[1),4),7),8)] の多くは，1対1のメッセージ通信の順序保存を保証している．また，順序保存性のない環境でも，オブジェクトごとのローカルな番号付によって実現できる．したがって，順序保存性がもともと存在する環境では積極的に利用すべきであり，存在しない環境でも実現コストが十分に小さければ利用すべきである．

送信側のロールバックによってアンチメッセージが発生した場合，順序保存性がありかつ送信者を識別できる場合には，送信側が古い順にアンチメッセージを送信すると，受信側では最初のアンチメッセージが到着した時点で，そのメッセージよりも新しいタイムスタンプを持つ同一オブジェクトからのメッセージを一

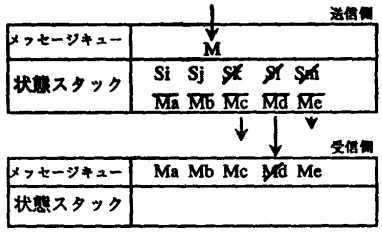


図3 ロールバック発生時の内部データ構造例
Fig. 3 Example internal data structure when a rollback occures.

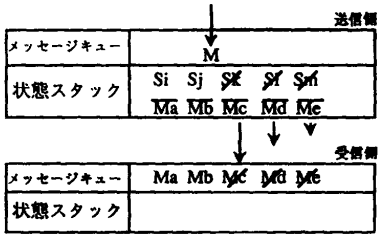


図4 不正なメッセージの一括削除
Fig. 4 Deletion of incorrect messages.

```
<メッセージ管理部>
  新メッセージの処理
    while(新メッセージのタイムスタンプ<
          カレントメッセージのタイムスタンプ)
      do {作業用スタックに状態スタックのトップにあるアンチメッ
          セージをプッシュする
          状態スタックをポップしスタックトップにある情報を廃棄
          カレントメッセージを1つ前のメッセージにする}
    while(作業用スタックが空でない)
      do {作業用スタックのトップにあるアンチメッセージを送信
          作業用スタックをポップしスタックトップの情報を廃棄}
      オブジェクトの状態を状態スタックのトップにある状態にする
      タイムスタンプ順の正しい位置に新メッセージを挿入する
  アンチメッセージの処理
    if (アンチ対象のメッセージがメッセージキューにある) then {
      while(アンチ対象のメッセージがカレントメッセージあるいは
            カレントメッセージより前にあるメッセージ)
        do {作業用スタックに状態スタックのトップにあるアンチメッ
            セージをプッシュする
            状態スタックをポップしスタックトップにある情報を廃棄
            カレントメッセージを1つ前のメッセージにする}
      while(作業用スタックが空でない)
        do {作業用スタックのトップにあるアンチメッセージを送信する
            作業用スタックをポップしスタックトップの情報を破棄}
      オブジェクトの状態を状態スタックのトップにある状態にする
      対応するメッセージとそれ以後の同一発信者からの
            メッセージを削除}
```

**図 5** 順序保存性と発信者識別性を仮定するアルゴリズム
Fig. 5 The improved algorithm which assumes that the arriving order of messages is preserved and a receiver can recognize the message sender.

括して削除できる．したがって誤った中間状態が発生しなくなる．これは，アンチメッセージがそれ以前のメッセージを追い抜くことがなく，またアンチの後の新たなメッセージがアンチを追い抜くことがないからである．**図 4** の例では受信側のオブジェクトは $\overline{Mc}$ を受信した時点で Md, Me を同時に削除できる．

順序保存性と発信者識別性がある場合のアルゴリズムを**図 5** に示す．メッセージ管理部ではメッセージの一括削除および古い順のアンチメッセージの送信が行われる．実行管理部は元のアルゴリズムと同一なので省略されている．

## 4. アルゴリズムの改良—その 2—

### 4.1 ロールバックのコスト

ロールバックをゼロにはできないので，ロールバックが発生した場合の処理量を低く抑えることが重要である．ロールバックに関係する処理には，オブジェクトを昔の状態にもどす処理と，変化後のメッセージ系列に基づいてオブジェクトのプログラムを実行し時間を進める処理がある．昔の状態にもどす処理は状態の代入とアンチメッセージの送信のみである．これに対し時間を進める処理は，シミュレーションの主要部分であるオブジェクトのプログラムを実行する処理なので作業量が多い．したがって，ロールバックの負荷を下げるには時間を進める処理を軽減することが重要である．

バーチャルタイムアルゴリズムではロールバック時に状態スタックの情報を廃棄するため，ロールバック後のメッセージ系列が既に実行したことがある系列であってもオブジェクトのプログラムを再実行しなければならず効率的でない．メッセージの系列が過去に実行されたものと同一になるのは次の 2 つの場合がある．.

ケース 1：ローカルクロックより前のメッセージが到着したのちにそのメッセージがアンチされた

ケース 2：ローカルクロックよりも前のメッセージがアンチされたあとで同じメッセージが到着した

ケース 1 の状況を図示すると**図 6** のようになる．まず，ローカルクロックより前のタイムスタンプを持つメッセージ M が到着する（図 6 (a)）．M をメッセージ系列に挿入し，M の直前の状態までロールバックする．このとき状態スタックの情報が削除される（図 6 (b)）．この段階で実行処理が行われると M, M2, と順にメッセージが受信されて状態スタックが伸びる．ここで $\overline{M}$ が到着すると再びロールバックが発生してメッセージキューは M が来る前と同じ状態になり再実行が必要に

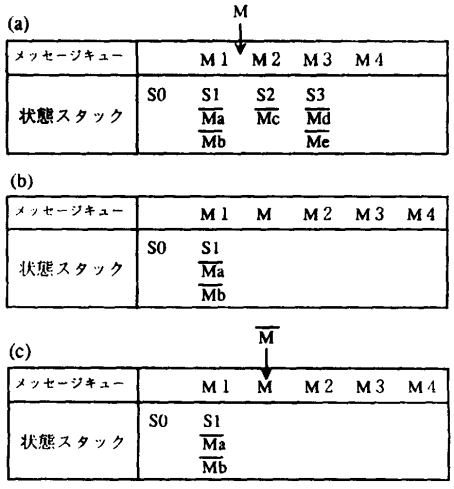


**図 6** 同一メッセージ列の発生
Fig. 6 Generation of the same message stream.

なる（図6（c））．この状況はオブジェクトのプログラム内容に関わりなくどのようなオブジェクトをシミュレートする場合にも発生する可能性がある．

ケース2の場合も同様に再実行が必要になる．この状況の発生はオブジェクトのプログラム内容に大きく依存する．詳しくは第6章で論じる．

### 4.2 状態保存アルゴリズム

木構造で状態を保持するとロールバック前の状態を保存でき，同じメッセージ系列が発生した場合に再実行することなく以前の状態にもどれる．図7は図6を状態木を用いて表した図である．状態木のノードは，受信したメッセージ，送信したメッセージのアンチメッセージ，新しい状態から構成される．網掛けされている状態ノードが最新のノードである．ケース1の場合には次のように動作する．メッセージMが到着すると状態木のノード S2〜S3 がアンチされ S1 の状態までもどる(図7（b）)．次にメッセージMを加えた新しいメッセージ系列に対応した状態系列 (S4, S5, …) が作成される（図7（c））．$\overline{M}$が到着すると，S4, S5 がロールバックされ，Mが削除される（図7（d））．M2〜M4 が再び実行される際には S2, S3 が再利用される．ケース2の場合にも同様に状態が再利用される．

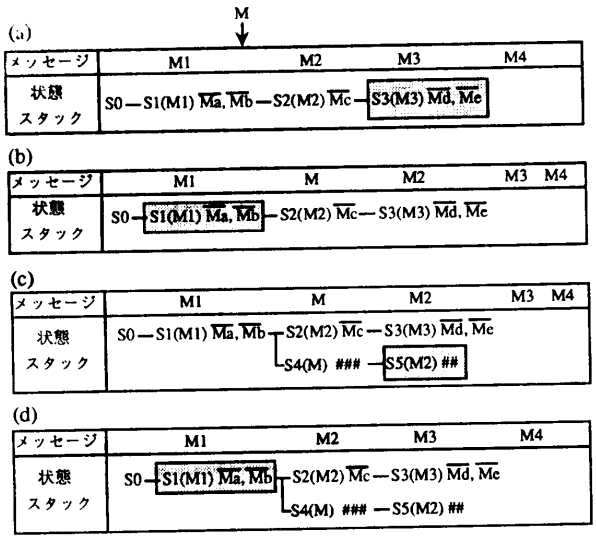


図7 状態木による再実行の回避
Fig. 7 Avoidance of re-execution making use of a status tree.

状態木を用いるアルゴリズムを図8に示す．メッセージ管理部は変更がないので省略されている．実行管理部では再利用できるノードがないか捜し，見つかった場合にはアンチメッセージに対応するメッセージを再送信している．状態木を用いた場合メモリ消費が問題になるが元のアルゴリズムと同様に GVT 以前の状態木は廃棄できるのでメモリの無制限な消費は防げる．

```
<実行管理部>
  メッセージの受信
    if (カレントメッセージの次にメッセージがある)
      then {
        カレントメッセージを1つ後ろのメッセージにする
        if (カレントノードの子ノードの中にカレントメッセージを
            受信しているノードがある)
          then {
            その子ノードをカレントノードとする
            カレントノードのアンチメッセージに対応する
                メッセージを送信する
            オブジェクトの状態をカレントノードの状態にする}
          else {
            新しい状態ノードを作成しカレントノードの子ノードとする
            この新しいノードをカレントノードとする
            カレントメッセージに対応するプログラムを実行する
            カレントメッセージをカレントノードに登録する
            送信したメッセージのアンチメッセージを
                カレントノードに登録する
            オブジェクトの状態をカレントノードに登録する}}
```

図8 状態ノードを再利用するアルゴリズム
Fig. 8 The improved algorithm which reuses the status nodes.

## 5. 評価実験

提案した方式の効果を調べるために，並列オブジェクト指向言語[4]を用いてアルゴリズムを記述し，2つの例題についてシミュレーションを行った．アンチメッセージの数と，状態を再利用できた回数を各アルゴリズムについて比較した．また，メモリ消費量の比較のために，GVT より古い状態記憶の廃棄を行った場合の保存ノード数を測定した．

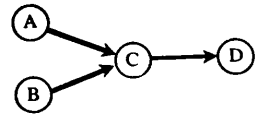


図9 例題1：輪状の関係がない場合
Fig. 9 Example 1: There are no circular relations among objects.

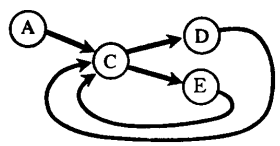


図10 例題2：洗車問題
Fig. 10 Example 2: The car wash problem.

図9に示すように，オブジェクトは通常複数個のオブジェクトからメッセージを受信し，他のオブジェクトにメッセージを送信している．一般的にはCの出力が直接的，間接的にA，Bの動作に影響を与えると考えられるが，まず影響を与えない場合を考える．

A，Bが発生するメッセージ系列について考える．メッセージは実行管理部の処理によって発生する．1回の処理で発生されるメッセージの個数はオブジェクトのプログラム内容に依存する．通常，個数の変動は小さく，一定個数が発生される場合がほとんどである．また，オブジェクトの入力に時間的な変動があっても，メッセージキューのバッファリグで吸収され，メッセージ発生の時間的な変動は小さいと考えられる．

アンチメッセージはメッセージ管理部の処理により発生する．一回の処理で発生するアンチメッセージの個数は，処理した入力がキューのどの位置に来るかで決まる．ローカルクロックより後の位置に来た場合は0個であり，前の位置に来た場合はそのタイムスタンプとローカルクロック差に比例した個数が一度に発生する．したがってアンチの個数の変動はメッセージと比較して大きくなる．入力の到着位置はローカルクロックに近い位置ほど可能性が高く離れるほど低いので，発生個数は少ない個数ほど確率が高い分布になると考えられる．

以上の点を考慮してメッセージ系列の発生を次のように行った．まず一定の確率でメッセージ，アンチメッセージのいずれを発生するか決定する．メッセージの場合にはメッセージを1つ発生する．アンチメッセージの場合には指数分布に従う乱数によって個数を決定し，その個数分のアンチメッセージを連続して発生する．指数分布では大きな値が発生する可能性があり，GVT に近いアンチメッセージによる大幅なロールバックを表現することができる．今回の実験ではメッセージとアンチメッセージの選択確率を 0.85 対 0.15 とし，一度にアンチされるメッセージの個数は平均3の指数分布とした．一回の選択におけるメッセージの発生個数の期待値が 0.85，アンチメッセージの発生個数の期待値が 0.45 となり，前者の期待値が大きいことで GVT の増加傾向を表現した．

図9において，受信したメッセージをそのままDに送信するオブジェクトとしてCを定義してシミュレーションを行い，Cの動作を測定した．結果を図 11 に示す．図の中で「一括削除」は第3章，「再利用」は第4章で提案したアルゴリズムを表す．また「保存」はメッセージの順序が保存されている状態でシミュレートしたことを示す．一括削除アルゴリズムは順序保存性を仮定しているので，同じ条件で比較する必要がある．図 11 の結果では，実行したメッセージ数，アンチメッセージ数のいずれも提案した方式の方が少ない．特に一括削除の場合のアンチメッセージ数は，順序保存性がある条件での元のアルゴリズムの約5分の1と大幅に減少している．状態の再利用は，いずれの場合も約 10% 程度発生している．メモリ消費は平均で3倍，最大 4.3 倍であった．状態木が最大のとき，木の高さは 22 でノード数は 95 であった．このときの，子供を持つノードの子供の数の平均は 1.3 であり，高さ方向と比較して木の広がりが小さいといえる．この実験によってオブジェクトの出力が入力に影響を与えない場合に関して，アンチメッセージの減少，状態の再利用が確認できた．

次にオブジェクトの出力が入力に影響を与える場合を考える．この場合，一方の入力のアンチがループの

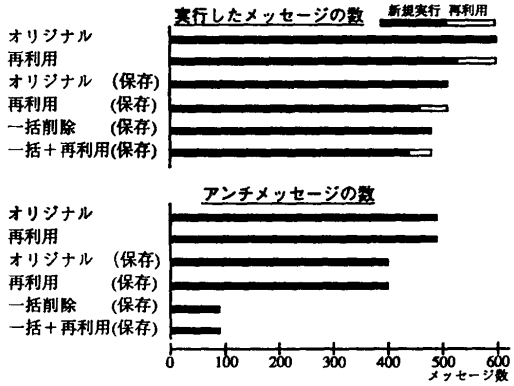


図 11 例題1のシミュレーション結果
Fig. 11 The simulation results of example 1.

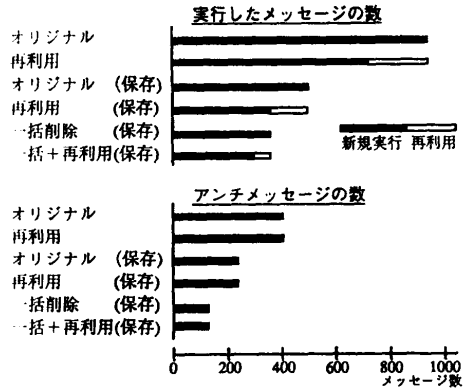


図 12 例題2のシミュレーション結果
Fig. 12 The simulation results of example 2.

影響によって他の入力のアンチを引き起こし，原因となった入力の一括削除だけでは正しい入力系列が得られず，第3章のアルゴリズムの効果が低下することが予想される．図9においてCの出力に依存してA，Bの発生分布を変化させるのは困難なので，**図10**に示す洗車問題[6),7)]のシミュレーションを行った．Aは自動車の発生源で，例題1のAと同じ確率分布に従って自動車の到着を表すメッセージとそのアンチメッセージをCに送信する．Cは洗車係のマネージャである．洗車係D，Eの中でより長く待っている方に自動車を割り当てる．洗車係は洗車が終了するとマネージャに次の自動車を割り当ててもらう．D，Eの出力が図9のBの代わりを果たしループの影響を表現している．

実験結果を**図12**に示す．アンチメッセージの減少は，順序保存性に起因するものの方が一括削除に起因するものよりも大きいが，一括削除アルゴリズムでも順序保存性があるオリジナルアルゴリズムよりさらに約45%少なくなっている．状態の再利用は，順序保存性のない場合には約20%，「一括削除＋再利用」のアルゴリズムでも約15%程度発生し，例題1よりも発生割合が高くなっている．メモリ消費は平均6倍，最大8.2倍であった．状態木が最大のとき，木の高さは23でノード数は189であった．このときの，子供を持つノードの平均の子供数は1.4であり，高さ方向と比較して木の広がりが小さいといえる．この実験によってメッセージ送受関係にループがある場合も，アンチメッセージの減少，状態の再利用が確認できた．

## 6. 議　論

第3章のアルゴリズムが保証するのは，発信者ごとに分けられたメッセージ系列が個別に間違っていないことである．メッセージ系列がとりうるパターンは，個々の発信者からのメッセージ系列のパターンの組合せで決まる．第3章で提案したアルゴリズムは個々のパターン数を減らす効果があるが，これらの系列の組合せの数を減らす効果はない．

第4章のアルゴリズムが効果を発揮するのは，ロールバックが発生しても発生する前と同じメッセージを送信するオブジェクトが存在する場合である．洗車問題におけるマネージャはこのようなオブジェクトである．マネージャに仕事を求める洗車係の要求が到着しばらくして車が来ると，その洗車係に仕事が割り当てられる．この後，この車が到着する前の時刻に別の洗車係の要求が割り込んだ場合，ロールバックによって最初の洗車係への仕事割当はいったんアンチされるが，結局この洗車係に割り当てられる．この場合最初の洗車係は仕事の割当を受信した後，そのメッセージがいったんアンチされ再び同一の仕事割当を受信する．このマネージャのように，メッセージ系列に鈍感なオブジェクトがシミュレーション対象に多く含まれている場合，再実行が必要になり第4章のアルゴリズムが効果を発揮する．実際，図12の結果でも再利用の割合が高くなっている．

## 7. おわりに

バーチャルタイムアルゴリズムの処理効率を向上させるポイントとして，ロールバックの発生をなるべく少なくすることと，ロールバックの処理コストを低くすることの2点に注目し，メッセージの順序保存性と発信者識別性を仮定したメッセージ管理アルゴリズムと，状態木によって再実行を防止する実行管理アルゴリズムを提案した．オブジェクトの出力が入力に影響を与えない例題と，与える例題にアルゴリズムを適用し有効性を確認した．

アンチメッセージの発生をより少なくするにはローカルクロックのずれを補正する必要がある．このような方式の1つとして，ロールバックの発生頻度に応じてそのオブジェクトの優先度を動的に変化させ，特定のオブジェクトだけが先走るのを防ぐ方式が考えられる．その実現と評価が今後の課題である．また，第5章の実験ではシミュレーションを用いて効果を確かめたが，数理的モデルに基づく厳密な議論が今後の課題として残されている．

**謝辞**　本研究の機会を与えてくださった日本電気(株) C&C システム研究所応用システム研究部永井部長，宮下課長に感謝いたします．

## 参考文献

1) Andrews, G.R. and Schneider, F.B.: Concepts and Notations for Concurrent Programming, *ACM Trans. Comput. Surv.*, Vol. 15, No. 1, pp. 3-44 (1983).
2) Chandy, K.M. and Lamport, L.: Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Trans. Comput. Syst.*, Vol. 3, No. 1, pp. 63-75 (1985).
3) Chandy, K.M., Holmes, V. and Misra, J.: Distributed Simulation of Networks, *Computer Networks*, Vol. 3, No. 2, pp. 105-113 (1979).
4) 福井眞吾：並列オブジェクト指向言語 DUE, 第

33 回情報処理学会全国大会論文集, pp. 441-442 (1986).
5) Jefferson, D. R.: Virtual Time, *ACM Trans. Prog. Lang. Syst.*, Vol. 7, No. 3, pp. 404-425 (1985).
6) Misra, J.: Distributed Discrete-Event Simulation, *ACM Comput. Surv.*, Vol. 18, No. 1, pp. 39-65 (1986).
7) Shibayama, E. and Yonezawa, A.: Distributed Computing in ABCL/1, Yonezawa, A. and Tokoro, M. (eds.), *Object-Oriented Concurrent Programming*, pp. 91-128, The MIT Press (1987).
8) Yonezawa, A., Briot, J. P. and Shibayama, E.: Object-Oriented Concurrent Programming in ABCL/1, *Proc. of Object-Oriented Prog. Syst. Lang. Appl.*, pp. 258-268 (1986).

(平成元年5月31日受付)
(平成元年9月12日採録)

**福井 眞吾** (正会員)

1959年生. 1982年東京工業大学理学部情報科学科卒業. 1984年同大学院修士課程修了. 同年日本電気(株)入社. 現在, 同社 C&C システム研究所応用システム研究部勤務. 並列オブジェクト指向計算, 並列アルゴリズムなどの研究に従事. 並列プログラミング言語, プログラミング環境, 分散システムに興味を持つ. ソフトウェア科学会, ACM 各会員.