ソフトウェア工学 117−10
（1997．11．21）

自動分散型オブジェクト指向数値計算プログラムの生成システム

上原均　畠山正行
E-mail:{uehara,masayuki}@cis.ibaraki.ac.jp
茨城大学工学部情報工学科

要約

本研究では逐次型オブジェクト指向シミュレーション・システム（OOSS）を分散型OOSSへ，人の手を煩わせずに再構成することを実現した．これは逐次型OOSSを解析した結果から，分散型OOSSの記述に必要な同期制御などの分散処理の記述全てを自動的に生成できることが判明したためである．本研究ではこの再構成処理によって構築された分散型OOSSを自動分散型OOSSと呼ぶ．本研究では再構成処理を行うソースコード・ジェネレータを開発し，それによる逐次型OOSSから分散型OOSSへの再構成実験にも成功している．本研究の成果は分散型OOSSの構築に難渋しているユーザ層にとって大きな価値を持つ．

Source Code Generator for Automatic Distributed Object Oriented Simulation Systems

Hitoshi Uehara　Masayuki Hatakeyama
E-mail:{uehara,masayuki}@cis.ibaraki.ac.jp
Department of Computer & Infomation Sciences, Ibaraki University

Abstract

The aim of the present paper is to re-construct a distributed object oriented simulation system (OOSS) without the user's load from a sequential OOSS. To realize this aim, we have designed and implemented a kind of the source code generator. This source code generator analyzes the sequential OOSS source code as the input, and generates the descriptions that are needed for distributed OOSS. We have succeeded in re-constructing the distributed OOSS from the sequential OOSS using this source code generator. This realization of our study leads much valuable merits for the users that must implement the distributed OOSS.

1　はじめに

科学技術計算の分野において，オブジェクト指向パラダイムに基づく数値シミュレーション，オブジェクト指向シミュレーション・システム（OOSS : Object-Oriented Simulation System）[1] [2] の構築が盛んになりつつある．これらは，高い計算負荷への対処や実行効率の向上などを行うために，いずれ分散型のOOSSへと発展することが予測できる．

しかし分散型OOSSを含む分散システムの構築は，逐次システムのそれよりも難しい．その理由の一つとして，通信処理やデッドロック問題，並行処理を行う上で不可欠な同期制御といった，分散システムにおける諸問題を解決するための処理（以降，これらの処理をまとめて「分散処理」と表現する）の記述が不可欠であることがあげられる．

これら分散処理を記述するには，プログラマが分散システムの知識を持っていることが不可欠である．しかし科学技術計算の分野では分散システムの知識を持たないその分野のドメインユーザがプログラマとなるケースがしばしばある．そのようなケースではシステム構築以前に分散システムについて学習しなければならないため，開発効率が大きく低下してしまう．またその学習自体もユーザにとっての重い負担となる．この結果，ユーザが分散型システムの構築を断念することすらある．

しかし我々が逐次型OOSSと分散型OOSSを比

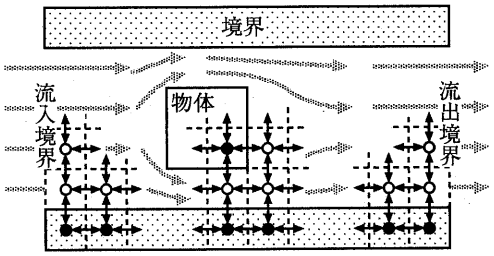

図 1: 二次元空間の流体シミュレーションの例

較検討した結果，分散型 OOSS に必要な分散処理は限定されおり，かつ自動処理またはその記述を自動生成できる程度であるため，逐次型 OOSS を解析することでそれらの分散処理を実現できることがわかった．この分散処理の自動化および記述の自動生成を実現すれば，プログラマによる分散処理の記述を必要とせずに分散型 OOSS を構築できる．

そこで我々は，プログラマによる分散処理の記述を必要とせずに分散型 OOSS を構築することを目的として研究を行い，本稿で述べるソースコード・ジェネレータにより，その目的を達成した．このソースコード・ジェネレータでは逐次型 OOSS のソースコードを入力として，それに分散システムとして必要な分散処理の記述を加えることで，分散型 OOSS のソースコードへと再構成する．これにより，分散型 OOSS の開発効率の大幅な向上が望める．

2 OOSS の対象世界とその分析

2.1 OOSS の対象世界

本研究では OOSS の対象世界を流体力学の分野に限定する．これは流体力学が科学技術計算の典型例の一つであり，他の分野への応用結果が多少なりとも推測できると判断したためである．

流体の OOSS を図 1を用いて説明する．

1. 内部状態が均一な部分空間の代表点をノードオブジェクト（図中の丸印）とする．
2. ノードは部分空間の状態を属性として持ち，近隣のノードと参照関係（矢印）にある．計算空間全体はノードの集合として表現する．
3. 境界や物体はノードの一種（黒丸印）として表現する．
4. ノードは自身の属性値と近隣のノードの属性値から自身の次の時刻の属性値を計算する．

この例のような計算空間の離散表現と計算法は流体シミュレーションでは一般的に用いられる．

前述のノードは近隣のノードの属性値以外の，属性値および処理記述の全てをそのノード内に保持している．よって近隣のノードの属性値を取得するメッセージ通信処理を分散システム対応とするだけで，ノード単位での分散計算が可能になる．

分散型 OOSS では，数値計算処理を並行化することによる，実行速度の向上が望まれる．この並行処理はオブジェクト指向システムでは非同期メッセージ通信によって実現できる [3].

2.2 逐次 OOSS と分散 OOSS の違い

前節で少し述べたが，同じ OOSS を逐次型システムとした場合と分散型システムとした場合，その流体の数値計算処理自体には違いはない．我々は逐次型 OOSS と分散型 OOSS を比較した結果から，両者の相違点を以下の分散処理の有無と結論づけた．

1. 分散システムを構成するプロセス群の生成/終了処理
2. 分散オブジェクトの生成（配置）処理
3. 分散オブジェクト間通信処理
4. 分散オブジェクト間通信におけるデッドロックの回避処理
5. 非同期通信メソッド関連の同期制御処理

上記から明らかなように，分散型 OOSS に必要な分散処理は 1. 以外は分散オブジェクト関連，特にメッセージ通信の分散システムへの対応であり，かつその処理内容には関係ない．このメッセージ通信処理はもともと定型的な処理であるため，分散システムに対応するように自動処理する，あるいはその記述を生成することは可能である．

また 1. の生成処理ではユーザからのホストマシンなどの分散環境情報の提供が必要であるが，プロセス生成などの処理自体は自動化できる．また終了処理も，分散システムを構成するプロセス群全てを終了することにすれば，その自動化は容易である．

3 自動分散型 OOSS

3.1 自動分散型 OOSS の実現可能性

2.2節で述べたように分散型 OOSS に必要な分散処理は限定されており，かつ自動処理あるいはその

記述を自動的に生成できる程度のものである．この点から我々は，それらの分散処理は逐次型 OOSS を解析することで実現できる，と考えた．

またそれとは別に我々は，分散オブジェクト関連の分散処理がメソッドの処理内容に関係ないことに着目し，それらの分散処理を逐次型 OOSS の記述を変更せずに利用可能とすることで，逐次型 OOSS から分散型 OOSS へと再構成できる，とも考えた．これは例えば，逐次型 OOSS ではローカルなメモリ領域を確保するだけのオブジェクト生成処理をリモートでのメモリ領域の確保へ変更することで，それを利用するための記法を逐次型 OOSS のそれから変更せずに，その動作を分散型 OOSS のものとできる，ということである．

我々は以上の二つの考えを組み合わせることで，逐次型 OOSS から分散型 OOSS への自動的な再構成が可能ではないか，という見解に達した．本研究ではこの見解の正当性を立証するために，逐次型 OOSS のソースコードから分散型 OOSS のソースコードを再構成するソースコード・ジェネレータを開発する．この実現方法を選択したのは「逐次型 OOSS と同じ記法で利用できる分散処理」の記述の生成が再構成処理の主な処理となると判断したためである．この開発では OOSS の記述言語を C++，その動作環境を小規模 LAN と想定した．これは OOSS の記述言語および動作環境としてそれらが一般的であろうと考えたためである．

なお以降では 2.2節であげた分散処理をプログラマがソースコードに記述しないことを「分散無記述性」と表現する．そして分散無記述性が達成された逐次型 OOSS のソースコードから再構成された分散型 OOSS を「自動分散型 OOSS」と呼ぶ．これはプログラマの手を煩わせずに，2.2節で列挙された，分散型 OOSS に必要な分散処理が自動的に処理されることに由来する．ソースコード・ジェネレータの目的は，この自動分散型 OOSS の実現と位置づけることができる．ソースコード・ジェネレータでどのような自動分散型 OOSS を実現するかについては以降の節で述べる．

3.2 自動分散型 OOSS のシステム構成

本研究ではソースコード・ジェネレータによって実現する自動分散型 OOSS のシステム構成として，マスター・スレーブ方式を採用する．これは並行処理を行う上で必要な同期制御を，マスター側で集中的かつ容易に行える点を評価したからである．

この自動分散型 OOSS のマスターでは自動分散型 OOSS 全体の管理と同期制御の処理を行い，一方のスレーブでは分散オブジェクトを配置し，そのメソッドの実行処理を行う．スレーブでは，分散オブジェクト間通信機構を並行動作させることで，複数のメッセージを並行処理する．

また同期制御のために，マスターに非同期通信メソッドの実行終了を監視するメカニズムを設け，非同期処理メソッドの呼び出しはこのマスター側からのみに限定する．この監視メカニズムは通常の処理と並行して非同期通信メソッドの終了を監視する．

また自動分散型 OOSS 全体はマスターの起動を起点として自動的に起動するものとする．この起動処理は，マスターが各ホスト上にスレーブを生成し，各スレーブはマスターの管理下で自身以外のスレーブとの通信経路を確立することで行われる．また自動分散型システムの終了はマスターおよび全スレーブを終了させるものとする．これらの処理は自動分散型 OOSS 内部で自動処理されるものとする．

このマスターのソースコードには，プログラマが記述した逐次型 OOSS のソースコードを極力流用する．一方，スレーブのソースコードは相当する記述が逐次型 OOSS のソースコードにないため，ソースコード・ジェネレータで生成する．

3.3 分散オブジェクト

前節で述べた自動分散型 OOSS に配置される分散オブジェクトのオブジェクトモデルには，オブジェクト内部に複数のアクティビティを許す多重スレッドモデル [3] を採用する．

また分散環境でオブジェクトを一意に識別するための分散オブジェクト ID を新たに与えるものとする．この分散オブジェクト ID から C++の OID への変換手段もあわせて提供する．

この分散オブジェクトは proxy-object[4] による方式で実現するものとする（図 2参照）．ローカルな proxy-object がリモートオブジェクトとの通信処理を隠蔽しつつ，リモートオブジェクトと同等のメソッドを提供する．この proxy-object による方式は 1) ネイティブコードによる効率的な実行ができる，2) 異機種分散環境へ対応しやすい，ことから選択した．

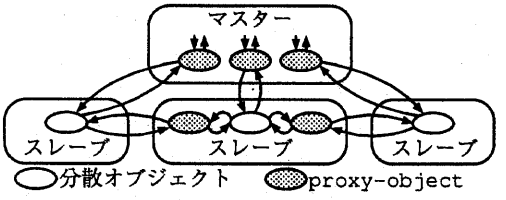

図 2: proxy-object による分散オブジェクトの実現

本研究で開発するソースコード・ジェネレータにより，2.2節で述べた分散オブジェクト関連の分散処理の記述を隠蔽して，この proxy-object のクラス定義とそれに対応したスレーブの記述を生成する．これについては次節以降で詳述する．

3.4 proxy-object のクラス定義

本節では，本研究のポイントである分散無記述性を実現するために我々が案出した，proxy-object のクラス定義方法について述べる．

proxy-object の定義方法としては，proxy-object のクラスをプログラマが定義したクラスとは別の型として定義する方式が考えられる．事実，HORB[5] はこの方式を採用している．しかしこの方式では，プログラマが proxy-object を分散オブジェクトについての記述手段として明白に意識して用いなければならないので，分散無記述性は実現できない．

そこで我々は分散無記述性を実現するために，従来の方法とは異なる proxy-object の定義方法を案出した．この方法では，プログラマが定義したクラスとその proxy-object のクラスを「入れ替え」て定義する．具体的には，

1. プログラマが定義したクラス名を変更し，
2. proxy-object のクラスをプログラマが定義したクラス名で定義，

する．これを図 3と図 4で説明すると，

1. クラス A, B, C を定義しており，
2. クラス名でクラス定義以外の記述と関連，

づけられている（図 3参照）状態において

1. クラス名 A, B, C を A', B', C' に変更し，
2. proxy-object クラスをクラス名 A, B, C で定義，

する（図 4参照）というものである．クラス定義以外の記述で宣言しているクラス名は変更していないため，proxy-object クラスがプログラマが定義した

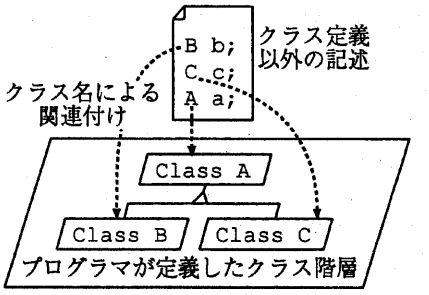

図 3: クラス名による関連

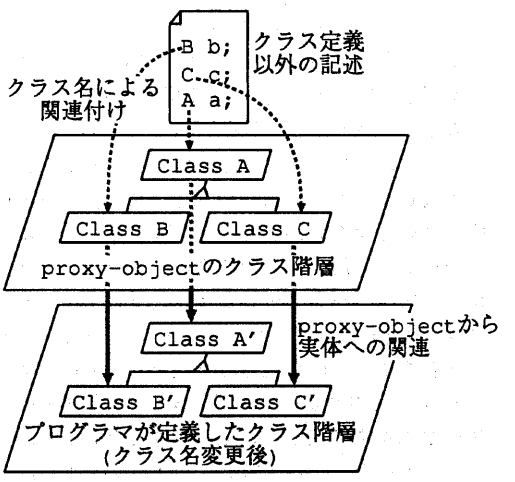

図 4: proxy-object の導入

クラスの代りに関連づけられる．クラス間の関連もこれと同様である．

この一連の処理によって，(クラス名以外の) プログラマによる記述を変更せずに，2.2節で述べた分散型 OOSS に必要な分散オブジェクト関連の分散処理を隠蔽する proxy-object の導入を実現する．この proxy-object のクラス階層はプログラマが定義したクラス階層と相似な階層とする．これはメソッドの継承関係を proxy-object の階層において自然に再現するためである．

分散オブジェクトの配置は，この proxy-object のコンストラクタ内部で最適なホストを自動選択してする．プログラマが定義したクラスでコンストラクタを定義されていなかったとしても，proxy-object ではこれらの配置処理を行うためにコンストラクタを定義する．また proxy-object のクラス定義では，分散オブジェクトを生成しないダミーコンストラクタも定義する．このダミーコンストラクタは，階層的なコンストラクタ呼び出しによる不要な

分散オブジェクトの生成を防ぐために使用される．デストラクタも階層的に呼び出されるが，これはproxy-object 内部のフラグ処理で回避する．

通常の proxy-object のメソッドはリモートオブジェクトへの通信処理のみを隠蔽しているが，本研究の proxy-object の定義方法ではローカルオブジェクトに対しても proxy-object を介してアクセスすることになるため，該当するオブジェクトがローカルである場合には proxy-object のメソッド内部において通常の C++のメソッド呼び出しを行う．

またオブジェクト間通信でのデッドロックは，proxy-object とスレーブのメッセージ処理メカニズムの内部で静的・構造的に回避する．

この proxy-object のクラス定義をファイルへ出力する際には，プログラマが記述したクラスの定義ファイルを違うファイル名に変更し，その代りに proxy-object のクラス定義をプログラマが記述したファイル名で出力する．なお proxy-object のクラス定義ファイルではプログラマが記述したクラス定義ファイルも読み込む．この一連のファイル操作により，ソースコードで読み込まれるクラス定義ファイル名を変更せずにすむ．

また分散オブジェクト ID から C++の OID へ変換する変換関数の定義も，proxy-object の定義と同時にソースコード・ジェネレータによって生成される．変換関数は，分散オブジェクト ID で示される分散オブジェクトへアクセスするための，ローカルな proxy-object の C++の OID を返す．

3.5 非同期通信と同期制御

本研究では，分散オブジェクト間の同期通信に加えて，並行処理を実現するために非同期通信とそれに関連する同期制御を実現する．

この実現ではメソッドの戻り値および引数に，

- 戻り値を持たず，
- 引数が無い，または定数のみ，

といった制限を設けて，これらを同時に満たすメソッドのみを非同期通信メソッドとして扱う．

それに関連する同期制御は以下の三つの処理を proxy-object 内部で行うことで実現する．

proxy-object 毎の遅延: ある proxy-object を介して非同期通信メソッドを実行した場合はその実行が終了するまで，その proxy-object を介したメソッド実行を遅延する．

非同期通信メソッド実行時の遅延: 一つ以上の非同期通信メソッドを実行している場合には，それら全てが終了するまで，同期通信メソッドの実行を遅延する．

同期制御コードによる同期: 各非同期通信メソッドに任意の同期制御コードを定めておき，異なる同期制御コード値を持つ非同期通信メソッドは並行処理しない．

これらは，2.1節で述べた数値計算を分散型 OOSS で並行処理するために必要な同期制御方法を検討した結果から，必要十分と判断して採用した．

4 ソースコード・ジェネレータ

本研究では，2.2節で述べた分散オブジェクト関連の分散処理の記述を proxy-object 関連の記述として生成する．そのため，ソースコード・ジェネレータで行う処理は以下のようになる．

- proxy-object クラス定義の生成
- スレーブ・ソースコードの生成
- 自動分散型 OOSS の初期化/終了処理の設定

それらの処理は以下のような手順で行う．

1. プログラマが記述したクラス定義の解析 (クラス名，継承関係，メソッド情報等の取得)
2. 前項で取得されたクラス定義情報から proxy-object クラスを定義
3. 前項で得られた proxy-object クラスの定義へ，ダミーコンストラクタなどの定義を追加
4. プログラマが記述したファイルのバックアップ (`File` → `File.org` としてバックアップ)
5. プログラマが記述したクラス定義ファイル中のクラス名の変更，およびファイル名の変更 (`F.cc` → `F_dist.cc`， `F.h` → `F_dist.h` と変更)
6. プログラマが記述したファイル名で，proxy-object のクラス定義を出力
7. スレーブのソースコード (`main` 関数と通信処理関数の記述を含む `sub_main.cc`) を生成
8. 分散オブジェクト ID から C++の OID への変換関数を定義
9. 逐次型 OOSS の `main` 関数を変更し，自動分散型 OOSS のマスターの `main` 関数とする

2. の処理では，非同期通信メソッドを決定する対話的処理とその同期制御コードの決定も行われる．8. の処理では，変換関数をマスターとスレーブ共用のヘッダファイルとして生成するauto_func.h 内で定義する．このヘッダファイルのソースコード中への読み込みは自動的に行われる．9. の処理では，逐次型 OOSS のmain 関数の冒頭に初期化関数の呼び出しを追加することで，自動分散型 OOSS のマスターのmain 関数とする．この初期化関数ではプロセスの生成処理などを行う．この初期化に必要な分散環境情報は，自動分散型 OOSS 起動時にマスタープログラムの引数として与えられる．初期化関数への分散環境情報の引き渡し処理の記述も 9. の処理で行われる．また，この初期化関数で終了時処理関数として自動分散型 OOSS の終了関数も設定する．

5 試作と動作実験

5.1 ソースコード・ジェネレータの試作

ソースコード・ジェネレータの試作は SparcStation20(Solaris 2.5) 上で行い，C++で記述した．プロセス内部での並行処理は，Solaris スレッドライブラリで実現した．この試作システムでは逐次型 OOSS の記述に，

1. public メソッドの引数および戻り値としてのポインタの使用禁止
2. 大域変数およびクラス変数の禁止
3. インライン展開される public メソッドの禁止
4. public メソッドの引数として使えるデータ型は char, double, float, int とその配列のみ
5. public メソッドの戻り値として使える型は char, double, float, int のみ

といった制約を課している．このうち，1と 2は分散メモリ型環境で動作することに由来する．3, 4, 5 は，試作システムの構造を簡略化するための制約であり，いずれ解消する予定である．また，これらの制約とは別に，public メソッドの引数および戻り値としてオブジェクトの参照を用いる場合には分散オブジェクト ID を用いなければならない．

この試作システムは逐次型 OOSS のmain 関数が記述されたメインファイル名と各クラス定義ファイル名をコマンドライン引数として起動する．

この試作システムで定義する proxy-object のコンストラクタでは，ある程度の数の分散オブジェクトをまとめて一つのスレーブに配置するようにしている．また被集約オブジェクトは，集約オブジェクトと同じスレーブに配置される．

このソースコード・ジェネレータの試作に合わせて，基底クラスを定義した．この基底クラスでは，分散オブジェクト ID やオブジェクト内並行制御メカニズム，同期制御に必要な属性などを隠蔽する．本試作システムを用いて自動分散型 OOSS を構築するには，必ずこの基底クラスを継承してクラスを定義しなければならない．

なお試作システムが生成するソースコード量を抑えるために，専用関数ライブラリを作成した．このライブラリには通信処理関数などが含まれる．

5.2 ソースコード生成および動作実験

前節で述べた試作システムが，自動分散型 OOSS のソースコードを生成できることを実証するために，ソースコード生成および実行実験を行った．試作システムへの入力には流体の差分法の一種である Roe スキームを用いた逐次型 OOSS のソースコードを用いた．これは文献 [6] の分散型 OOSS の原型であり，計算結果の合理性などが実証されていたためである．

この逐次型 OOSS の記述には 5.1節で述べた制約を満たさない箇所があったので，約 5 時間ほどでそれを変更し，ソースコード・ジェネレータへ入力した．その結果，対話的処理も含めて約 1 分弱で自動分散型 OOSS のソースコードが生成できた．この生成処理後のファイル構成を図 5に示す．

この得られたソースコードをコンパイルして実行した結果を図 6に示す．この実行では二つのホストマシンに 4 つのスレーブを生成して分散計算を行った．この計算では 1250 個 (50x25) 個のノードから構成される二次元の計算空間に対して 10000 回の繰り返し計算を行っている．図 6は流れ方向速度の等高線を示しているが，この計算結果は変更前の逐次型 OOSS による正当な計算結果と一致し，自動分散型 OOSS における計算が正常に行われたことが確認できた．またデッドロックなども生じなかった．

この実験結果から，本研究で開発したソースコード・ジェネレータによる，分散無記述性を達成した逐次型 OOSS のソースコードから正常に動作する

auto_func.h	*in_dist.cc*	*right_dist.cc*
const.h	*in_dist.h*	*right_dist.h*
down.cc	**left.cc**	**roe.cc**
down.h	**left.h**	**sub_main.cc**
down_dist.cc	*left_dist.cc*	**up.cc**
down_dist.h	*left_dist.h*	**up.h**
grid.cc	**point.cc**	*up_dist.cc*
grid.h	**point.h**	*up_dist.h*
grid_dist.cc	*point_dist.cc*	**upst.cc**
grid_dist.h	*point_dist.h*	**upst.h**
in.cc	**right.cc**	*upst_dist.cc*
in.h	**right.h**	*upst_dist.h*

filename はファイル名変更されたファイル
filename は新しく生成されたファイル
filenameは変更なく流用したファイル

図 5: 生成処理後のソースコードファイル群

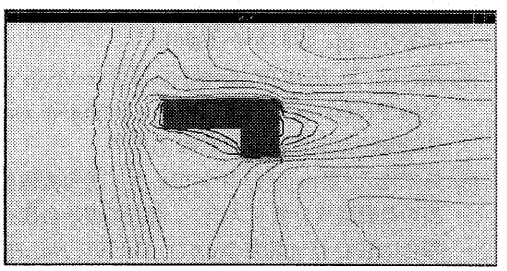

図 6: 自動分散型 OOSS の計算結果

分散型 OOSS のソースコードへの再構成，つまり自動分散型 OOSS の実現，が実証できたと判断する．

6 議論

6.1 関連研究との比較

我々の調査では本研究に類似した研究例を見いだせなかったので，分散オブジェクト指向言語/技術の研究を関連研究として取り上げる．

Argus[7] は抽象データ型言語であり，遠隔サイトの故障などの分散プログラミングに対処している．しかしデッドロックを検出あるいは回避できない[8]．Emerald[9] も抽象データ型言語であり，アクセス透過性および位置透過性を提供しているが，分散処理の幾つかがプログラマに隠されていない[8]．MPC++[10] はメタレベル・アーキテクチャを特徴とする，並列/分散向けに拡張された C++の拡張言語である．MPC++では同期/非同期呼び出しは invoke/ainvoke 関数で明示的に行う．HORB[5] は Java の分散オブジェクト実行系である．HORB ではプロキシを用いることを明示的に宣言し，非同期通信するメソッドも明示的に記述する．CORBA[11] は OMG が推進している，様々な言語で利用できる分散オブジェクト技術である．そのプログラミングでは CORBA の専用関数を用い，非同期通信なども明示的に記述する．科学技術計算の分野での標準的な通信ライブラリである MPI の C++言語対応ライブラリとして，OOMPI[12] がある．OOMPI では従来の MPI の関数群をクラスライブラリとして再構成しており，その利用には MPI の知識が必要である．

これらの関連研究では，最終的にはプログラマが何らかの分散処理の記述を行わなければならないため，分散無記述性の達成という観点では本研究が優れていると結論する．

6.2 考察

6.2.1 自動分散型 OOSS の実現について

本研究のソースコード・ジェネレータで生成する proxy-object 関連の記述では，分散オブジェクトの配置処理，通信処理，デッドロックの回避および同期制御をその内部で行っている．これらの詳細は 3.4節や 3.5節で述べた．またソースコード・ジェネレータでは，分散型 OOSS を構成するプロセス群の生成処理などを行う初期化関数の組み込みを行っている．この初期化関数内部で終了時処理の設定も行っている．この処理については 4章で述べた．以上であげた各分散処理は 2.2節で列挙した分散型 OOSS に必要な分散処理と正確に対応づけられる．またソースコード・ジェネレータへの入力である逐次型 OOSS のソースコードには一切分散処理は記述していない．また，その分散処理の動作および逐次型 OOSS のソースコードへの組み込みについては，5.2節で述べた実験結果から，正常に行われているものと判断できる．

以上から，(1) 分散無記述性を達成した逐次型 OOSS のソースコードから分散型 OOSS に必要な分散処理を実現し，(2) 分散処理の逐次型 OOSS への組み込みによる分散型 OOSS への再構成を実現している，ので，本研究の目的である逐次型 OOSS から分散型 OOSS への自動的な再構成，つまり自

動分散型 OOSS は本研究で開発したソースコード・ジェネレータによって実現できた，と判断する．

6.2.2 本手法に適した計算スキーム

本研究の手法では通信負荷が高くなる傾向が顕著であり，分散オブジェクトの動的な位置移動による通信負荷の軽減もできない．そのため，本手法に適した計算スキームが計算負荷に比べて通信負荷が低いものであることは明白である．元々の計算スキームの通信負荷が低ければ，本手法による通信負荷の増大も問題にはなりにくい．逆に適さない計算スキームは上記の逆の計算負荷に比べて通信負荷が元々高いもの，例えば5.2節で用いている Roe スキームなどである．

本手法に適さない計算スキームで効率的な自動分散型 OOSS を構築するには，集約オブジェクトで計算空間を幾つかに分割してその大粒度オブジェクト間でのみ通信する，領域分割法が適切と判断する．しかし，大粒度オブジェクトの自動生成はほぼ不可能であるため，本研究では領域分割についてはプログラマに一任する．

6.3 今後の課題

今後の研究課題としては，自動分散型 OOSS の最適化と異機種分散環境への対応があげられる．現在，分散オブジェクトの最適配置手法として，対象世界の次元情報やオブジェクト数などから配置を最適化する手法を検討している．また現状の試作システムで生成するソースコードで用いている Solaris スレッドを POSIX スレッドへ変更することで，ソースコードの可搬性を高め，本来想定していた異機種分散環境での動作の実現を現在検討している．

7 結論

本稿では，逐次型 OOSS から分散型 OOSS への再構成手段として我々が開発した自動分散型 OOSS のソースコードジェネレータについて述べた．このソースコード・ジェネレータは，分散無記述性が実現された逐次型 OOSS のソースコードから分散型 OOSS のそれを再構成する，つまり自動分散型 OOSS のソースコードを生成する．これを用いることで，分散型 OOSS 構築時のプログラマの負担を大きく軽減できることが期待できる．本研究ではソースコード・ジェネレータの試作も行い，その動作実験の結果から，本研究で案出した手法によって自動分散型 OOSS のソースコードを生成することが十分可能であることを実証した．今後は分散オブジェクトの配置の最適化および異機種分散環境への対応を行う予定である．

参考文献

[1] 日本機械学会: 第 6 回計算力学講演会講演論文集, 日本機械学会 (1993).

[2] S.N.Atluri et al.(eds.): *Computational Mechanics*, Vol. 1, Springer (1995).

[3] 所真理雄ほか (編): オブジェクト指向コンピューティング, 岩波書店 (1993).

[4] Shapiro, M.: Structure and encapsulation in distributed systems : the proxy principle, *ICDCS*, pp. 198–204 (1986).

[5] Hirano, S: HORB: Distributed Execution of Java Programs, *World Wide Computing and its Applications*, pp. 29–42 (1997).

[6] 鈴木俊人, 畠山正行: オブジェクト指向数値計算の自動負荷分散システム, 情報処理学会研究報告 (HPC 研究報告 No.67), Vol. 97, No. 75, 情報処理学会, pp. 109–114 (1997).

[7] Liskov, B.: Distributed Programming in Argus, *CACM*, Vol. 31, No. 3, pp. 300–312 (1988).

[8] 前川 守, 所真理雄, 清水謙多郎: 分散オペレーティングシステム, 共立出版 (1991).

[9] Black, A. et al.: Distribution and Abstract Type in Emerald, *IEEE Trans. Softw. Eng.*, Vol. SE-13, No. 1, pp. 65–76 (1990).

[10] Yutaka Ishikawa: Multiple Threads Template Library – MPC++ Version 2.0 Level 0 Document – Document Revision 0.12, Technical report, Tsukuba Researh Center, Real World Computing Partnership (1997).

[11] Object Management Group: *CORBA2 Universal Networked Objects* (1995).

[12] Lumsdaine, A.: Object Oriented MPI: A Class Library for the Message Passing Interface, *POOMA'96* (1996).