自己反映言語 Open C++ とその分散処理への適用の実際

千葉 滋　益田 隆司

Open C++ は自己反映計算に基づいた拡張可能な C++ 言語である．Open C++ では，プログラマは C++ 言語を使って，C++ 言語のメソッド呼出しや変数アクセスの実現機構を拡張できる．これらを拡張することで，さまざまな言語プリミティブを Open C++ 上で実現することができる．例えば遠隔手続き呼出しや，分散共有オブジェクトなどの，分散処理のための言語プリミティブが実際に実現可能である．Open C++ の拡張はメタオブジェクトによって定義される．Open C++ は，分散処理のための言語プリミティブの実装を容易にするために，Object Communities というメタオブジェクトのクラスライブラリを用意している．Open C++ はオブジェクト指向に基づいた自己反映計算を採用しているので，このように言語の拡張自体がライブラリ化できる．

1 はじめに

分散処理や並列処理の研究の進展に伴い，それらの処理のための言語が次々に提案されてきた [2]．これらの言語は，通信と同期のために特別な言語プリミティブを用意している．例えば Ada の rendezvous [25] や，遠隔手続き呼出し [4]，Orca の共有データオブジェクト [1] などがあげられる．これらの言語プリミティブは広い分野のアプリケーションに適用可能であるが，全ての分野で最適であるわけではない．したがって，さまざまなアプリケーションの開発に対応するためには，同一言語内で，複数の言語プリミティブが使用可能であることが望まれる．

本論文では，複数の言語プリミティブを利用できるように，言語仕様自体が拡張可能になっている言語の設計について述べる．このような拡張可能言語は，始めから多数の言語プリミティブを提供している，マルチパラダイム言語よりも優れている．なぜなら，後者は言語仕様が巨大になり，しばしば習得が困難であるが，前者は言語の基本仕様が簡潔であり，必要な言語プリミティブだけを利用できるので，理解しやすい．また拡張可能言語では，プログラマがまったく新しい言語プリミティブを追加することもできる．

本論文が述べる言語 Open C++ [8] は，言語仕様を拡張可能な C++ [22] である．プログラマは適当な言語プリミティブを C++に追加することができる．現在，分散処理や並列処理に対応するためにいくつかの拡張をほどこした C++が多数存在するが，Open C++は，自身の言語仕様を拡張することで，それらの言語と同等の機能を提供できることを最終的にめざしている．OpenC++を用いると，プログラマは，作成するアプリケーションに適した言語プリミティブを持つ C++の変種を容易に作成することができる．

Open C++の拡張は，オブジェクト指向の自己反映計算 (reflection) [21] [14] に基づいてなされる．オブ

A Reflective Language Open C++ and Its Application to Distributed Computing.

Shigeru Chiba, 東京大学理学部情報科学科, Dept. of Info. Sci., Univ. of Tokyo.

Takashi Masuda, 東京大学理学部情報科学科, Dept. of Info. Sci., Univ. of Tokyo.

コンピュータソフトウェア, Vol.11, No.3(1994), pp.33–48.

1993 年 8 月 23 日受付.

ジェクトはメタオブジェクトを持つ．メタオブジェクトは，そのオブジェクトへのメソッド呼出しと変数アクセスの実現機構[†1]を定める．プログラマは適切なメタオブジェクトを選択することで，そのオブジェクトへのメソッド呼出しと変数アクセスの実現機構を変更することができる．それらの実現機構を変更することで，さまざまな言語プリミティブが，Open C++上に実装される．メタオブジェクトは C++で記述されるので，プログラマは新しいクラスを定義して，望みの実現機構を定めるメタオブジェクトを容易に生成することができる．言語拡張の研究は 1960 年代にも活発におこなわれたが [29]，当時の研究と本研究の違いは，当時の研究は主に言語の文法の拡張に注目しており，一方，本研究は言語プリミティブの実現機構—すなわち言語の意味—の拡張を考えている点である．Open C++では，言語の文法はそのままに，プログラム中の文が持つ意味を変更することで，言語の拡張をおこなう．

本論文では，さらに，実際に Open C++上に実装された分散処理のための言語プリミティブについて，拡張の具体例として紹介する．個々の言語プリミティブを実現するメタオブジェクトのクラスは，全体として，Object Communities と呼ばれる 1 つのクラスライブラリとしてまとめられている．このためコードの再利用性が高く，プログラマは，このクラスライブラリを利用してサブクラスを定義することで，容易に独自の言語プリミティブを Open C++上に実装することができる．これはオブジェクト指向に基づいた自己反映計算を採用したことによる利点である．

以下では，第 2 章で自己反映計算に基づいた言語の拡張について，その概論を述べ，他の研究成果に対する本研究の位置付けを示す．第 3 章では，本論文で述べる言語 Open C++の概要を説明し，合わせてその実装方法を述べる．第 4 章では，Open C++上に実装された言語プリミティブをいくつか紹介し，それらが具体的にどのように実装されているかを示す．第 5 章では，Open C++で書かれたプログラムの実行時の性能の測定結果を示す．第 6 章では，ライブラリを使った従来の言語拡張の技法との比較を論ずる．最後に第 7 章でまとめと今後の課題を示す．

†1 メソッド呼出しと変数アクセスの際に，実際にどのような計算処理をおこなうか．

2 自己反映計算による言語拡張

自己反映計算とは，一般には，計算システムにそれ自身のモデルを計算対象として与え，計算システムが自身を動的に変更・修正できるようにするための技術である．このような技術の理論的な裏づけを探るために，3-Lisp [21], Brown [26], Blond [9], ACT/R [28] などの言語が作成されている．自己反映計算の実際的な用途は主に 2 つに分類される．

- 実行時システムの最適化：
  言語処理系やＯＳ，ネットワーク通信などのシステムは，平均して最も実行効率がよくなるように実装されるが，一般に，任意の場合に最良の実行効率を達成することはできない．そこで，自己反映計算を用いてシステムの実装アルゴリズムなどを実行中に動的に変更して，実行効率を改善する．例として，Window システム Silica [19] や並列言語 ABCL/R [27] [16] などがある．分散言語 AL-1/D [17] では，プログラム実行時の統計情報を自己反映計算を使って集めることができる．また生成コードの最適化のやり方を変更できる Scheme コンパイラ Intrigue [13] などもある．
- システムの機能拡張：
  元から用意されている機能だけでなく，ユーザの必要に応じて，システムに機能を追加，拡張できるようにする場合，自己反映計算は有効な道具となる．システム自身のモデルを与え，これを通して拡張を定義できるので，自己反映計算を用いると，ad hoc でない系統だったやり方でシステムを拡張することができる．CLOS MOP [12] や Apertos オペレーティングシステム [30] などが例としてあげられる．

本論文で扱う自己反映計算は，後者の，システムの機能拡張に分類される．本論文で述べる言語 Open C++では，自己反映計算を用いて，C++言語が元々言語プリミティブとして用意していない機能を，容易に言語に追加することができる．Open C++は，メソッド呼出しと変数アクセスの実現機構に関する部分

の言語仕様を，プログラマにメタオブジェクトとして公開する．プログラマは，プログラムの中からメタオブジェクトを選択することで，言語処理系 (コンパイラ) がそれらの実現機構をどう変更するかを指示できる．このとき言語処理系の再構築などは不要である．自己反映計算を用いているので，言語を拡張する機能は，言語自身の機能の一部として，明確に定義される．処理されるプログラムとは独立に言語処理系に拡張のための特別なプログラムを与えたり，あるいは複雑な pragma 命令[†2]を用いるというような，ad hoc な手法でなく，簡潔でわかりやすい方法で言語の拡張をおこなうことができる．

本論文で述べる Open C++と，従来の他の自己反映言語との違いは，Open C++がシステム記述言語であり，高い実行効率を維持しつつ，自己反映計算による言語拡張を可能にしようとしている点である．一般に，自己反映計算はシステムの実行効率の低下をもたらす．例えば，言語コンパイラの場合，言語仕様を自己反映計算によって拡張可能にすると，生成コードの最適化が困難になる．このため Open C++の設計では，自己反映計算の必要性と実行効率のバランスが重視された．Open C++では，実行効率の低下を抑えるために，自己反映計算によって変更可能な部分を，分散処理や並列処理の言語プリミティブを Open C++上に実装するのに必要な最小限の部分に制限している．C++に自己反映計算を導入した研究は他に Meta-Information-Protocol (MIP) [5] があるが，MIP では言語の実現機構の変更を許しておらず，自己反映計算の範囲を型情報の参照だけに制限している．

さらに Open C++では，自己反映計算による言語拡張で，分散処理のプリミティブが実装できることを具体的に示している．オブジェクト指向言語に自己反映計算を導入し，何らかの言語拡張を可能にした言語はすでにいくつか提案されているが [14]，Open C++の特徴はさまざまなパラダイムの分散言語の実現に自己反映計算を応用できることを明らかにした点である．自己反映計算に関する研究では，自己反映計算の実現機構の研究と同時に，その具体的な応用目的を探究することが重要である．

†2 コンパイラの実装に依存した，特別なコンパイル制御命令.

3 Open C++の概要

Open C++は自己反映計算に基づいて言語機能を部分的に拡張可能にした C++言語である．Open C++のプログラムは，一部の自己反映計算のための宣言を除き，C++のプログラムと同一である．本章では，まず Open C++のメタレベルアーキテクチャについて述べ，続いて Open C++の言語仕様の概要と，言語処理系の実現方法の簡単な解説をおこなう．

3.1 メタレベルアーキテクチャ

Open C++はメソッド呼出しと変数アクセスの実現機構を C++オブジェクトで表現し，処理系のメタシステムとしてプログラマに公開している．実現機構を表現しているオブジェクトを，メタオブジェクト (metaobject) と呼ぶ．メタオブジェクトは C++オブジェクトであり，通常のオブジェクトとまったく同様に定義される．メソッド呼出しと変数アクセスの実現機構が変更されるオブジェクトは，1 つのメタオブジェクトを持つ．オブジェクトとメタオブジェクトの関係は 1 対 1 である．メタオブジェクトはオブジェクトを制御し，そのオブジェクトへのメソッド呼出しや，そのオブジェクト内の変数アクセスを，拡張された実現機構で実行する．

オブジェクトとメタオブジェクトはメタ階層を形成する．オブジェクトが属す階層をベースレベル，メタオブジェクトが属す階層をメタレベルと呼ぶ．ここで，メタオブジェクトも C++オブジェクトであるので，メタオブジェクトを持たせ，メソッド呼出しと変数アクセスの実現機構を変更することができる．この場合，メタオブジェクトのメタオブジェクトのことを，メタメタオブジェクトと呼び，メタメタオブジェクトはメタメタレベルを作ると考える．このように，オブジェクトとメタオブジェクトの関係は無限の階層構造を作る．Open C++では，必要に応じて宣言することで，このメタ階層を自由に増やすことができる．

メタオブジェクトは，メソッド呼出しと変数アクセ

スの実現機構だけを表現し，他の実現機構，例えば多重継承の実現機構などは表現しない．したがってOpen C++では，CLOS MOP などと異なり，多重継承の規則などは拡張できない．これは，不要な拡張性を導入したために，全体の実行効率が低下することを避けるためである．既存の多くの分散言語が提供する同期と通信の言語プリミティブは，メソッド呼出し (手続き呼出し) または変数アクセスを拡張したものなので，Open C++ではメソッド呼出しと変数アクセスの実現機構が変更できれば充分と考える．

3.2 Base-Level Directives

オブジェクトにメタオブジェクトを持たせるには，いくつかの宣言をプログラム中に埋め込まなければならない．例えば，クラス X のオブジェクトがクラス M のメタオブジェクトを持つことを宣言するには，

```
//MOP reflect class X : M;
```

とする．オブジェクトがメタオブジェクトを持つと実行効率が低下するので，オブジェクトごとにメタオブジェクトを持つか否かを指定できる．メタオブジェクトを持つ場合はクラス名の前に refl_ をつけて，クラス refl_X のオブジェクトとして，例えば以下のように生成する．

```
X* obj = new refl_X;
```

もしクラス refl_X でなく，ただの X としてオブジェクトを生成すると，そのオブジェクトはメタオブジェクトを持たない．そのようなオブジェクトへのメソッド呼出しや変数アクセスは，標準のままの方法で実行され，実行効率も低下しない．メタオブジェクトを持つオブジェクトと，そうでないオブジェクトを区別するため，前者を特に，自己反映オブジェクト (reflective object) と呼ぶ．なお，メタオブジェクトを持つか否かは，オブジェクト生成時に決定し，動的に変えることはできない．

プログラマは，自己反映オブジェクトの個々のメソッドや変数についても，メタオブジェクトによって制御されるか否かを指定できる．すなわち自己反映オブジェクトへの全てのメソッド呼出しや変数アクセスが，メタオブジェクトによって制御されるわけではない．メタオブジェクトによって制御されるメソッドや変数のことを，reflect メソッド，reflect 変数と呼ぶ．メタオブジェクトによって制御されないメソッドや変数は，たとえ自己反映オブジェクトのものであっても，標準のままの方法で実行され，実行効率も非自己反映オブジェクトと同じである．reflect メソッド・変数の指定は予約語 reflect: を用いて次のようにおこなう．

```
class X : {
public:
    int f(int p);
    int i;
    ...
//MOP reflect:
    int g(int p);
    int j;
};
```

これは，クラス X のメソッド g() と変数 j が reflect メソッド・変数であることを宣言している．メソッド f() と変数 i は，普通の C++メソッド・変数である．

3.3 Simple MOP

自己反映オブジェクトの reflect メソッドや reflect 変数への呼出し，アクセスは，メタオブジェクトの記述に従って実行される．メタオブジェクトプロトコル (MOP)[†3]は，メタオブジェクトの記述の方法を定めるプロトコルである．Open C++のMOP は，Open C++の処理系に詳しくなくても，容易に言語の拡張ができるように，非常に簡潔なものになっている．本節では，Open C++の MOP について述べる．

メタオブジェクトのクラスは，必ずクラス MetaObj を継承しなければならない．クラス MetaObj は，メソッド呼出しや変数アクセスの実現に必要な，さまざまなメソッドを定義している．これらのメソッドの一部をサブクラスで再定義することで，メソッド呼出し

†3 メタオブジェクトプロトコルは，正確には，オブジェクト指向によって設計された，メタプロトコルであると定義できる．ここでメタプロトコルとは，あるプロトコル (例えば，メソッド呼出し．これはある手続きを起動するためのプロトコルである) の振舞いや実装を定義するプロトコルを指す．なお，MOP は「モップ」と読む．

や変数アクセスの実現機構を変更することができる．以下では，クラス MetaObj で定義されている主なメソッドを紹介する．

reflect メソッドが呼ばれると，そのメソッド呼出しはトラップされ，代わりにメタオブジェクトのメソッド Meta_MethodCall() が実行される．このメソッドは，実際に呼ばれた reflect メソッドの実行手順を，インタプリタ風に定義する．reflect メソッドは，このメタオブジェクトのメソッドの中から明示的に実行されない限り，実行されない．以下はクラス MetaObj におけるメソッド Meta_MethodCall() の定義である．これは，呼ばれた reflect メソッドを元の C++ でのメソッド呼出しとまったく同じように実行する．

```
void MetaObj::Meta_MethodCall(Id
                method_id, Id category,
          ArgPac& args, ArgPac& reply){
    Meta_HandleMethodCall(method_id,
                          args, reply);
}
```

このメソッドの引数は順に，呼ばれた reflect メソッドの整数識別子，そのカテゴリ識別子，実引数列，返り値に対応する．カテゴリ識別子は，プログラマが呼ばれた reflect メソッドにカテゴリ名を与えている場合，それを表す．詳しくは文献[6]を参照されたい．引数の中で重要なのは，実引数列と返り値を表す引数 args，reply である．引数 args は，トラップしたメソッド呼出しの実引数列を表す．具体的には，メソッド呼出しフレームの実引数列部分のコピーである．一方，引数 reply はトラップしたメソッド呼出しの返り値を表す．メソッド終了時に reply に格納された値が，トラップしたメソッド呼出しの呼出し元に返される．どちらも ArgPac (クラス) 型の C++オブジェクトである．

クラス MetaObj が定義するメソッド Meta_MethodCall() は，呼ばれている reflect メソッドを実際に実行するだけである．実行にはメソッド Meta_HandleMethodCall() が用いられる．このメソッドはクラス MetaObj が定義するメソッドで，サブクラスで再定義することはできない．このメソッドは，与えられた識別子に対応する reflect メソッドを，与えられた実引数列を使って実行する．実行された reflect メソッドの返り値は，3番目の ArgPac 型の引数に格納される．このメソッドは，指定の reflect メソッドを最後まで実行するので，メタオブジェクトは，reflect メソッドを途中まで実行して止める，といった処理を定義することはできない．

クラス MetaObj は，変数アクセスがトラップされたときに実行されるメソッドも定義している．サブクラスでこれらのメソッドを再定義することで，変数アクセスの実現機構を変更できる．これらのメソッドは，reflect メソッド呼出しがトラップされたときに実行されるメソッドと，概ね同じである．

reflect 変数が参照されると，その参照はトラップされ，メタオブジェクトのメソッド Meta_Read() が実行される．このメソッドは，実際に値が参照される変数を指定する．このメソッドが指定した変数の値が reflect 変数の値となる．クラス MetaObj によるこのメソッドの定義は以下のようである．

```
void MetaObj::Meta_Read(Id var_id,
        Id category, ArgPac& reply){
    Meta_HandleRead(var_id, reply);
}
```

引数は順に，アクセスされた変数の整数識別子，カテゴリ識別子，実際に参照される変数，である．このメソッド終了時には，最後の引数 reply に，実際に参照される変数のアドレスが格納されていなければならない．引数 reply に値そのものでなく，参照先のアドレスを格納するのは，参照される変数がオブジェクトである場合に対応するためである．クラス MetaObj における Meta_Read() は，メソッド Meta_HandleRead() を用いて，単純に引数 var_id に対応する変数のアドレスを引数 reply に格納する．

reflect 変数に値が代入されるときには，メタオブジェクトのメソッド Meta_Assign() が実行される．以下はクラス MetaObj での定義である．

```
void MetaObj::Meta_Assign(Id var_id,
          Id category, ArgPac& arg){
    Meta_HandleAssign(var_id, arg);
```

```
}
```

このメソッドは代入する値へのポインタを受け取り，代入を実行する．引数は順に，アクセスされた変数の整数識別子，カテゴリ識別子，代入する値へのポインタ，である．このメソッドは，クラス MetaObj で定義される別のメソッド Meta_HandleAssign() を使って，実際に代入をおこなう．

上記のメソッドの他に，クラス MetaObj は reflect 変数の値を直接読み出して ArgPac 型のオブジェクトに変換したり，また ArgPac 型のオブジェクトを適当な型に変換して reflect 変数に直接代入するメソッドも定義している．この他にも，メソッド識別子や変数識別子から，対応するメソッド名や変数名を引くメソッドなども定義している．クラス MetaObj のサブクラスで，これらのメソッドを使い，適当にメソッド Meta_MethodCall()，Meta_Read()，Meta_Assign() を再定義することで，さまざまな言語プリミティブを実現するメタオブジェクトを定義できる．

3.4 MOP による拡張

MOP による Open C++ の拡張の具体例として，簡単なメソッド呼出しの実現機構の拡張を紹介する．ここに示すクラス VerboseMeta のメタオブジェクトは，reflect メソッドが実行される度に，実行されたメソッド名を表示する．以下は，クラス VerboseMeta で再定義されたメソッド Meta_MethodCall() である．

```
void VerboseMeta::Meta_MethodCall(
        Id mid, Id category,
        ArgPac& args, ArgPac& reply){
    printf("<%s> was called.\n",
           Meta_GetMethodName(mid));
    Meta_HandleMethodCall(mid, args,
                          reply);
}
```

このメソッドは呼ばれたメソッドの名前を始めに表示し，その後，そのメソッドを実行する．

ここで示したクラス VerboseMeta のメタオブジェクトは任意のクラスのオブジェクトのメタオブジェクトになりえる．Open C++のメタレベルでは，ベースレベルのオブジェクトのクラスや，メソッドの種類，変数の型は抽象化されていて，それらに依存せずにプログラムを記述できる．例えば，メソッドの返り値や引数列は，その引数の個数や型にかかわらず，単一の ArgPac 型で表現される．またメソッド名もコンパイラが割り当てた整数値で表現される†4．このため Open C++のメタレベルのプログラムは，ベースレベルの複数のクラスから共通に使え，再利用性が高い．また，同じベースレベルのクラスの中でも，例えば単一の Meta_MethodCall() の記述で，異なる型や個数の引数列を持つメソッド全体の実現機構を定義することができる．

このような抽象化は，自己反映計算の用語で reify (具象化) および reflect (反映) という．ベースレベルからメタレベルに処理が移るとき，ベースレベルでは直接扱えない情報が，一階 (first class) のデータに変換されて，メタレベルで扱えるようになる．この変換のことを reify と呼ぶ．一方，メタレベルからベースレベルに処理が戻るときには，一階のデータに変換されていた情報がシステムの内部状態に反映される．この処理のことを reflect と呼ぶ．例えば，ベースレベルではメソッドの個々の引数を扱うことはできても，引数列全体を 1 つの値として扱うことはできない．しかしメタレベルでは，これが reify されて，引数列全体が 1 つの ArgPac 型の値で表現される．

3.5 Open C++の処理系

現在の Open C++コンパイラは，C++コンパイラへの前処理系である．以下ではこの Open C++コンパイラによる C++上での自己反映計算の実現の概要を述べる．

例として，クラス X のメソッド f() が reflect メソッドである場合を考える．このクラスの定義は以下のようである．

```
class X {
public:
```

†4 メソッドごとにメタレベルでの処理をかえる場合は，メソッド識別子でなくカテゴリ識別子を使う．カテゴリ識別子はプログラマがメソッドに対して明示的に指定できる．

```
//MOP reflect:
    int f(int i);
};
//MOP reflect class X : MetaObj;
```

最後の行は，クラス X のオブジェクトはクラス MetaObj のメタオブジェクトを持つことを宣言している．Open C++ コンパイラは，この宣言を見つけると，以下のようなクラス X のサブクラス refl_X を暗黙のうちに生成する．

```
class refl_X
: public X, private MetaMsgReceiptor {
public:
    int f(int);
    void Base_HandleMethodCall(Id, Id,
                    ArgPac&, ArgPac&);
    MetaObj* metaobj;
    . . .
};
```

クラス refl_X がクラス X の他に継承しているクラス MetaMsgReceiptor は，メタオブジェクトとのインタフェースを定義した内部的に使われるクラスである．メタオブジェクトは，全ての自己反映オブジェクトをこのクラスのオブジェクトとして扱う．クラス refl_X はオブジェクトへのメソッド呼出しや変数アクセスをトラップして，メタオブジェクトの適切なメソッドを実行する．クラス X の自己反映オブジェクトは，このクラス refl_X のオブジェクトとして生成されることを思い出して欲しい．このクラスはメタオブジェクトへのポインタ metaobj を持ち，さらに reflect メソッド f() を再定義する．したがって自己反映オブジェクトのメソッド f() を呼ぶと，この再定義されたメソッドが呼ばれる． 図 1 に，reflect メソッドが呼ばれたときの処理の手順を示す．再定義されたメソッド f() は，実引数列を ArgPac 型のオブジェクトに変換して，メタオブジェクトのメソッド Meta_MethodCall() を呼び出す．メタオブジェクトのメソッドが終了すると，返り値を ArgPac 型のオブジェクトから取り出して，int 型に変換し，それを返り値として終了する．最初の変換が reify 処理に相当し，最後の変換が reflect 処理に相当する．下に再定義されたメソッド f() を示す．

```
int refl_X::f(int i){
    ArgPac args, reply;
    args.PushInt(i);
    metaobj->Meta_MethodCall(3,
          Category_none, args, reply);
    int r = reply.PopInt();
    return r;
}
```

再定義されたメソッド f() は，遠隔手続き呼出しのクライアント側のスタブ (stub)[4] によく似ている．クライアント側のスタブは，引数をネットワークメッセージに変換し，サーバにそのメッセージを送信する．その後，サーバから返答メッセージを受信し，そこから呼び出した手続きの返り値を取り出す．スタブがおこなう前者の変換を marshaling と呼び，後者の変換を unmarshaling と呼ぶ．スタブの marshaling が reify に，unmarshaling が reflect に対応するといえる．遠隔手続き呼出しにおいて，任意の型のデータの効率のよい (un)marshaling 処理は容易ではなく，多くの議論[10][31] がある．Open C++ コンパイラの reify/reflect 処理では，基本データ型についてはコンパイラが自動的におこなうが，クラス型のオブジェクトについてはプログラマが reify/reflect 処理用のメソッドを用意する，という方針を採用している．

メタオブジェクトは，クラス MetaObj が定義するメソッド Meta_HandleMethodCall() を使って，ベースレベルのオブジェクトの reflect メソッドを実行するかもしれない．このメソッド Meta_HandleMethodCall() は，実際にはクラス refl_X で定義されているメソッド Base_HandleMethodCall() を呼ぶことで reflect メソッドを実行する．Open C++ コンパイラが生成するこのメソッドは以下のようである．

```
void refl_X::Base_HandleMethodCall(
          Id method_id,
          ArgPac& args, ArgPac& reply){
    switch(method_id){
    . . .
    case 3 : {
        int p1 = args.PopInt();
        int r = X::f(p1);
```

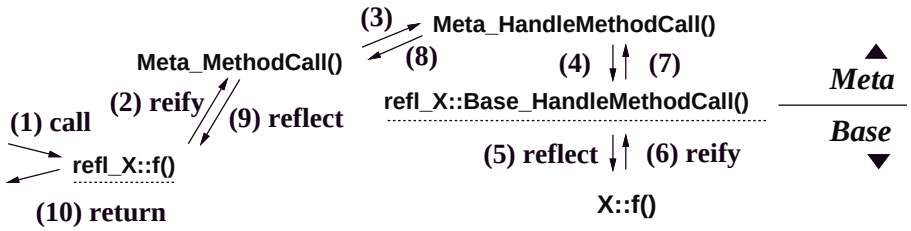

図 1　メタシステムの概要 (コンパイラは下線のメソッドを生成する)

```
        reply.PushInt(r);
        } break;
    . . .
    }
}
```

このメソッドは，与えられたメソッド識別子 `method_id` によって分岐し，それぞれのメソッドの型に応じて，与えられた実引数列 `args` から引数を取り出し，実際にクラス `X` で定義されているメソッドを呼び出し，返り値を引数 `reply` に格納する．このメソッドでは，引数を取り出す処理が reflect に相当し，返り値を格納する処理が reify に相当する．

上の例では，`reflect` メソッドの実現方法を述べたが，`reflect` 変数の実現方法もほぼ同様である．2つの間の違いは，`reflect` 変数の実現の場合，サブクラス `refl_X` で，メソッドを再定義するのでなく，同名の変数を別な型の変数として定義している点である．以下の例を考える．

```
class X {
public:
//MOP reflect:
    int i;
};
//MOP reflect class X : MetaObj;
```

クラス `X` の整数変数 `i` は `reflect` 変数である．Open C++ コンパイラが生成するサブクラス `refl_X` は下のようである．

```
class refl_X
: public X, private MetaMsgReceiptor {
public:
    i_in_X i;
    MetaObj* metaobj;
    . . .
};
```

型 `i_in_X` はコンパイラが生成するクラス型である．このクラス型は，代入や型変換演算子 (cast operator) などを再定義して，変数 `i` へのアクセスがあったときに，これをトラップし，適切なメタオブジェクトのメソッドを呼び出すようになっている．これは，`reflect` メソッドの場合に，サブクラスで再定義されたメソッド (先の例では `refl_X::f()`) がおこなう処理に対応する†5．

4　Open C++の分散処理への応用

Open C++ は，メソッド呼出しと変数アクセスの実現機構をプログラマが拡張するための機構を提供するが，それ自体は具体的な拡張の枠組を提供するわけではない．Open C++ では，言語を拡張するための機構と，それを用いて実際にどのように拡張するかが，分離されている．言語を拡張するための機構の仕様は，MOP によって明確に定められているが，具体的な拡張の指針については，Open C++ の言語仕様で定められていない．

本章では，Open C++ の上でさまざまな分散処理

†5 Open C++ コンパイラは，変数の読み出しをトラップするのに，クラス型 `i_in_X` から整数型への暗黙の型変換が起こることを利用している．しかしながら式によっては，暗黙の型変換が起こらないことも多い．その場合，コンパイルエラーとなるので，現在の言語仕様では，`()` 演算子をつけて，トラップを起こさなければならない．例えば，本文中の例の場合，`obj->i` ではなく，`obj->i()` のように記述しなければならない．この問題は，Open C++ コンパイラがより細かい意味解析をおこなって，適切なプログラム変換をするようになれば解決される．

のための言語プリミティブを実装するための枠組について述べる．この枠組は，新しい言語プリミティブを実現するために，メタオブジェクトのクラスをどのように定義するかの指針を与える．

4.1 Object Communities

Object Communities は，Open C++上で分散処理のための言語プリミティブを実装するための枠組である．この枠組では，言語プリミティブは異なるホスト計算機上に存在するいくつかの拡張されたオブジェクトによって実装されると考える．例えば遠隔手続き呼出しは，標準的には，クライアント側のスタブ関数を使って実装される．Object Communities では，これは，サーバオブジェクトと，スタブ関数に相当するスタブオブジェクトの組によって実装される．サーバオブジェクトとスタブオブジェクトは，それぞれ C++オブジェクトをメタオブジェクトで拡張することで実現する．組になったオブジェクトのグループのことを，この枠組では object community と呼ぶ．

Object Communities はこのような枠組にそった言語プリミティブの実装を支援するために，以下のような基本機能をクラスライブラリの形で提供する．

- **グループ化**：メタオブジェクトは他のメタオブジェクトと，異なるホスト計算機にまたがったグループ (object community) を作ることができる．
- **ネットワーク通信**：メタオブジェクトは，同一グループ内で，ネットワークを通じて他のメタオブジェクトと通信することができる．１対１通信の他，同報通信 (broadcast) が可能である．同報通信で送られたメッセージは，一意に順序付けされ，全てのメタオブジェクトが同じ順序で受け取る．
- **並行性**：メタオブジェクトは，マルチスレッド機能を使って，オブジェクト内部に並行性を与えることができる．例えばオブジェクトへのメソッド呼出しをトラップしたときに，そのメソッド呼出しをおこなっているスレッドを止めて別なスレッドを作り，そのオブジェクトの別なメソッドを並行に実行することができる．

このクラスライブラリは，この他に，いくつかの基本的な言語プリミティブを実現するためのメタオブジェクトのクラスも提供する．提供されているクラスからの差分を記述することで，容易に新しいメタオブジェクトのクラスを定義し，新しい言語プリミティブを実装できる．

このような，クラスライブラリが標準で提供する，分散共有オブジェクトなどの言語プリミティブは，メタレベルのプログラミングにも利用できる．これはメタオブジェクトも C++オブジェクトであり，メタレベルのプログラミングもベースレベルのそれと違いがないからである．非常に複雑な言語プリミティブを実装する場合，すでに実装してある他の言語プリミティブを使って記述できれば，実装がより容易になる．この場合，メタレベルで使われる言語プリミティブを実現するのは，メタメタオブジェクトとなる．Open C++では通常，ベースレベルとメタレベルしか存在しないが，このように必要に応じてメタメタレベルを作ることができる．

4.2 追加された MOP

上で述べた機能をメタオブジェクトに与えるために，Object Communities は，そのクラスライブラリの一部として，クラス `MetaObj` のサブクラス `OcCoreMetaObj` を定義している．Object Communities の枠組に沿って言語プリミティブを実現するメタオブジェクトは，この `OcCoreMetaObj` のサブクラスとして定義される．このクラスは，object community

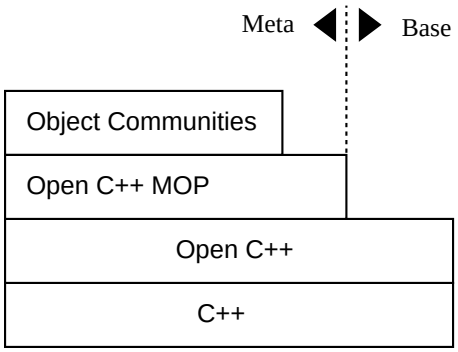

図 2 Open C++ のシステム階層

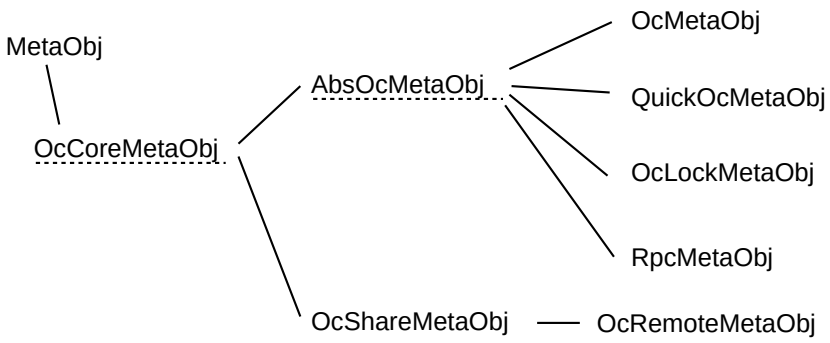

図 3 Object Communities のクラス階層 (下線を引いたクラスは抽象クラス)

を操作するためのメソッド，オブジェクト内の並行性に関係するメソッド，そして通信のためのメソッドを提供する．MOP という観点から Object Communities を見ると，Object Communities は Open C++ が標準で提供する MOP にいくつかの分散処理に関係したプロトコルを追加しているといえる (図 2)．このことから，Object Communities の代わりに，特定のネットワーク・ハードウェアに適した機能拡張のための枠組や，並列処理のための言語プリミティブを実装する枠組を，Open C++ MOP 上に構築することも，同様にして可能であることがわかる．

Object Communities は，`OcCoreMetaObj` の他に，いくつかの基本的な言語プリミティブを実現するメタオブジェクトのクラスも提供している．図 3 に全体のクラス階層を示す．図に示したクラスのうち，`OcShareMetaObj` は分散共有オブジェクトを実現するメタオブジェクトの，`OcRemoteMetaObj` はネットワーク透明なメソッド呼出しを実現するメタオブジェクトのクラスである．クラス `RpcMetaObj` は遠隔手続き呼出しを実現する．3 つのクラス `OcMetaObj`, `QuickOcMetaObj`, `OcLockMetaObj` は，著者らが設計した，グループウェアのような分散アプリケーションの開発のための，独自のデータ共有機構を実現する．なお図中で下線を引いたクラスは，サブクラスで共通のメソッドを定義したクラスで，実装上，内部的に使われる．

以下では，遠隔手続き呼出しと分散共有オブジェクトを例にとり，Object Communities に基づいてどのように言語プリミティブを実現するかを簡単に述べる．

4.2.1 遠隔手続き呼出し

遠隔手続き呼出しを実現するメタオブジェクトのクラスは `RpcMetaObj` である．まず始めにこれを使った遠隔手続き呼出しの記述を示す．例えば，2 つの整数を受け取って，その和を返すサービスをおこなうサーバを実装する場合を考える．プログラマは，まずこのサービスを実行するオブジェクトを定義する．

```
class Adder {
public:
//MOP reflect:
    int add(int a, int b){
        return a + b;
    }
};
```

次にこのクラスのオブジェクトのメタオブジェクトを宣言する．メタオブジェクトのクラスは `RpcMetaObj` である．

```
//MOP reflect class Adder:RpcMetaObj;
```

最後に，クライアント側とサーバ側のプログラムで，それぞれ次のようにクラス Adder の自己反映オブジェクトを生成する．

```
// クライアント側
refl_Adder client("adder");
// サーバ側
refl_Adder server("adder", IS_SERVER);
```

クライアント側はクラス `refl_Adder` のオブジェクト `client` を，サーバ側は同じクラスのオブジェクト `server` を生成する．ともに第 1 引数はサービスの名

前，第2引数はサーバ側かクライアント側かの区別，である[†6]．クライアント側は第2引数を省略することができる。クライアント側のオブジェクト client はスタブオブジェクトあるいは proxy [20] の役割をはたす．クライアント側で，オブジェクト client のメソッド add() を以下のように呼び出すと，

```
result = client.add(3, 4);
```

サーバ側のオブジェクト server が実行した add(3,4) の結果の返り値が result に代入される．

クラス Adder のオブジェクトを，遠隔手続き呼出しを実現するように拡張しているクラス RpcMetaObj のメタオブジェクトは，次のような処理をおこなう．まずメタオブジェクトは，生成時に第 1 引数で渡されるサービス名の object community を作ってそこに入る．サーバ側とクライアント側のメタオブジェクトは，同名の object community に入ることで，互いに相手を見つけ，通信路が設定される．さらに，このメタオブジェクトのメソッド Meta_MethodCall() は再定義されていて，クライアント側のオブジェクトのメソッド呼出しをトラップすると，メタオブジェクトはそのメソッドを実行する代わりに，メソッドの識別子と実引数列をサーバ側のメタオブジェクトに送信し，サーバ側に指定のメソッドを実行させる．そして返り値を受信して，受信した値を，トラップしたメソッドの返り値として，メソッド Meta_MethodCall() を終了する．同一 object community 間の通信機能は，Object Communities がクラス OcCoreMetaObj のメソッドとして用意しており，サーバ側のメタオブジェクトがクライアント側からメッセージを受信すると，クラス OcCoreMetaObj のある決まったメソッドが実行される．クラス RpcMetaObj では，このメソッドを再定義しており，サーバ側のメタオブジェクトがメソッド識別子と実引数列をクライアント側から受信すると，メソッド Meta_HandleMethodCall() を使い，受け取った実引数列でサーバ側のオブジェクトのメソッドを実行し，その返り値をクライアント側のメタオブジェクトに送信するようになっている．

†6 自己反映オブジェクトの構築子 (constructor) は，オブジェクト自身の構築子に与える引数と，メタオブジェクトの構築子に与える引数の両方を取る．例の場合では，クラス Adder の構築子はないので，メタオブジェクトの構築子の引数だけを与える．

4.2.2 分散共有オブジェクト

分散共有オブジェクトとは，異なるホスト計算機間でデータを共有するためのオブジェクトである．任意のホスト計算機から，このオブジェクトが持つ変数に分散透明にアクセスできる．Object Communities の枠組では，分散共有オブジェクトは，各ホスト計算機上に作成された複製オブジェクトの集まりによって実装される．複製オブジェクトは共有されるデータの複製を自分が持つ変数に格納する．各ホスト計算機から分散共有オブジェクトの変数にアクセスする場合は，この複製オブジェクトの変数がアクセスされる．複製オブジェクトは，自分の変数に代入が起きたときには，同じデータを共有する他の複製オブジェクトにその代入を伝播させる．

例として，名前を表す文字列と年齢を表す整数を持つクラス Person を分散共有オブジェクトにする．

```
class Person {
public:
//MOP reflect:
    argString name;
    int       age;
};
//MOP reflect class Person
                    : OcShareMetaObj;
```

クラス refl_Person のオブジェクトは，クラス OcShareMetaObj のメタオブジェクトによって，複製オブジェクトとなる．このオブジェクトの reflect 変数 name と age は共有データの複製値を常に保持するようにメタオブジェクトによって制御される．変数 name の型 argString は，C++の文字列を表現するための型である．これは Open C++が標準ライブラリとして提供するクラス型で，reify/reflect 処理の際に，文字列全体が正しく ArgPac 型のオブジェクトに格納されるようにする．

このクラスのオブジェクトを生成するときは，

```
refl_Person boy("jack");
```

のようにメタオブジェクトの構築子に，分散共有オブジェクトの名前 (例では jack) を渡す．メタオブジェクトは生成時に，この名前の object community に入

表 1 空メソッド呼出しの平均遅延時間 (μsec.)

引数の個数	0	1	5	5× double
C++ メソッド	0.3	0.6	1.3	2.1
C++ virtual メソッド	0.8	1.0	1.8	2.2
reflect メソッド	1.8	6.3	13.8	21.7
reflect/virtual 比	2.3	6.3	7.7	9.9

SPARC 40MHz (28.5MIPS) and Sun C++ 2.1

る．これによって，メタオブジェクトは同じデータを共有している他の複製オブジェクトを知ることができる．このメタオブジェクトのメソッド Meta_Assign() は再定義されており，オブジェクトの変数に代入があったときに，代入された値を ArgPac 型のオブジェクトの形で取り出し，これを同じ object community のメタオブジェクトに同報通信で通知する．通知を受け取ったメタオブジェクトは，この代入を自分のオブジェクトの変数に対しておこなう．

Object Communities のクラスライブラリは分散共有オブジェクトを指す特別なポインタ†7も提供している．このポインタを通して分散共有オブジェクトにアクセスすると，複製が同一ホスト計算機上にない場合，自動的に複製が生成される．このポインタを使うと，複製オブジェクトを明示的に生成する必要がなくなり，より自然に分散共有オブジェクトを使うことができる．

5 実行時速度の測定

Open C++ は現在，SunOS 上で利用可能で，Object Communities を使ってネットワーク上で動作するアプリケーションを作成することができる†8．Object Communities は，ネットワークプロトコルとして TCP/IP を用い，UNIX の socket インタフェースを使用する．本章では，現在の版の Open C++と Object Communities で作成されたアプリケーションの実行速度の測定結果を紹介する．

5.1 自己反映計算による性能低下

自己反映計算の導入に伴う実行効率の低下を考察することは，実用的な自己反映計算処理系を作成する場合に重要なことである．Open C++での reflect メソッド・変数の実行は，部分的にメタオブジェクトが解釈実行するので，普通のメソッドや変数の実行に比べて遅くなっている．本節では，メソッド呼出しと変数アクセスに関する実行速度の測定結果を示す．

表 1 は 3 種類の空メソッド呼出しの遅延時間を示している．測定には，SPARC station 2 (SunOS 4.1.1) および Sun C++ 2.1 を用いた．遅延時間は引数の数を変えて測定した．右端のものを除き，引数は全て int 型である．右端のものは，double 型である．引数なしのものは返り値がないが，他のものは int 型の値を返す．ただし右端のものは，double 型の値を返す．メソッドの種類は，普通の C++メソッド，C++ virtual メソッド，そして reflect メソッドの 3 つである．普通の C++メソッドとは，コンパイル時にメソッド名とメソッド本体の結合が静的に解決されるメソッドである．C++ virtual メソッドとは，メソッド名とメソッド本体の結合が，実行時に表を引いて動的に解決されるメソッドである．reflect メソッドの呼出しは，クラス MetaObj のメタオブジェクトによって制御される．このメタオブジェクトは，通常の C++メソッド呼出しの実現機構と同じものを実装する．

表の最下段は，virtual メソッドと reflect メソッドの速度比を表している．速度比は，引数の数の増加に伴って大きくなる．引数がないときのオーバヘッドは非常に小さくなっている．これから，reflect メソッド呼出しのオーバヘッドの大半が，reify/reflect 処理，すなわち，実引数列を ArgPac 型の C++オブ

†7 C++ の smart pointer [22] の技術を使って実現されている．

†8 Open C++の現処理系は utsun.s.u-tokyo.ac.jp (133.11.11.11) から anonymous ftp で入手可能である．

ジェクトに変換する処理にあることがわかる．

表 2 整数型変数の平均アクセス時間 (μsec.)

変数	C++	reflect
read	0.1	3.5
wirte	0.1	3.5

SPARC 40 MHz (28.5 MIPS) and Sun C++ 2.1

次に表 2 に整数 (int) 型の変数へのアクセス時間の測定結果を示す．測定には，空メソッド呼出しの遅延時間の測定と同じ環境を用いた．'C++' は通常の整数型インスタンス変数のアクセス時間，'reflect' は reflect 変数のアクセス時間を示す．reflect 変数のアクセスは，クラス MetaObj のメタオブジェクトにより制御される．このメタオブジェクトは，通常の C++変数アクセスの実現機構と同じものを実装するので，reflect 変数のアクセスは C++の通常の変数アクセスと同じように実行される．

表 1 は，reflect メソッドの実行は，virtual メソッドの実行と比較して，約 6 から 8 倍遅いことを示している．また表 2 は，reflect 変数のアクセスは，普通の変数アクセスに比べて約 35 倍遅いことを示している．この結果はけっして小さくないが，Open C++が分散処理のために使われる限り，これらの実行速度の悪化は無視できる．なぜなら，現在の平均的なネットワークの性能では，メッセージの伝送時間はおよそ数百マイクロ秒から数ミリ秒であるからである．また，プログラマは自己反映オブジェクトを，一部の機能拡張が必要なオブジェクトに限定できるので，適切に設計されたアプリケーションでは，ほとんどのオブジェクトは普通の C++オブジェクトとなり，自己反映計算による実行速度の低下は全体として小さくなる．著者らは，自己反映計算の技術は実用的なプログラミングに充分適用可能な水準に達していると考える．自己反映計算による実行効率の低下は今のところ回避できないが，分散処理など，うまく応用領域を選べば充分無視できる．

Open C++の実行時のオーバヘッドの原因の大半は，reify/reflect 処理にあるが，これは遠隔手続き呼出しの (un)marshaling 処理とほぼ同じ処理であり，ネットワークでデータを送るためには不可欠である．したがって，Open C++のオーバヘッドは，他の分散言語などでも共通して現れる．例えば Sun の RPC システム [23] などでは，1 つの整数 (int) 型引数の marshaling 処理に数マイクロ秒必要とする．これは対応する Open C++の reify 処理よりも遅いが，これは Sun のシステムが汎用的なライブラリから構成されていて，1 回の marshaling 処理に数回の手続き呼出しを伴うからである．

5.2 Object Communities の実行時性能

Object Communities のクラスライブラリが提供する分散システム上のグループ管理や通信の機能は，1 個のサーバプロセスによって実現されている．それぞれのメタオブジェクトはこのサーバとだけ通信し，メタオブジェクト同士が直接ネットワーク経由で通信することはない．これによって実装を単純化している．本節では，このサーバプロセスを介したメタオブジェクト間の通信速度の測定結果を示す．

表 3 は，空の遠隔手続き呼出しの遅延時間を示す．Object Communities を使って Open C++上に実装された遠隔手続き呼出しと，Sun RPC[23] を使って実装された遠隔手続き呼出しとを比較した．Sun RPC では，スタブを自動生成する rpcgen というツールを用いた．Sun RPC の測定では，ネットワークプロトコルとして TCP/IP と UDP/IP を用いた場合の両方を測定した．Object Communities は TCP/IP を用いている．測定は Ethernet で結合された SPARC station ELC を用いた． 2台の計算機間の ICMP パケット (64byte) の往復時間は 1 msec. であった．測定は 0, 4byte, 4Kbyte の 3 種類の大きさの引数を使っておこなった．サーバプロセス，クライアントプロセ

表 3 遠隔手続き呼出しの遅延時間 (msec.)

引数の大きさ	0	4-byte	4-Kbyte
Object Communities	4.5	4.5	14.1
Sun RPC+UDP	2.1	2.1	6.4
Sun RPC+TCP	2.6	2.6	19.5

SPARC 33MHz (21.0MIPS), Packet round-trip 1 msec.

ス，および Object Communities のサーバは，全て別々の計算機で動作している．

Object Communities による遠隔手続き呼出しは，Sun RPC に比べ約 2 倍遅い．これはメッセージがクライアントとサーバ間で直接やり取りされるのではなく，一度 Object Communities のサーバを経由するからである．したがって Object Communities のクラスライブラリにメタオブジェクト間で直接メッセージをやり取りする機能を追加すれば，Sun RPC と同等の速度を実現できると思われる．TCP/IP を使った Sun RPC は，引数 4Kbyte の場合，Object Communities に比べて遅くなっているが，これは Sun RPC の実装の問題であると考えられる．

表 4　同報通信の遅延時間 (msec.)

計算機の数	1	2	3	4	5
0-byte	2.5	2.5	2.5	2.7	2.5
4-byte	2.6	2.5	2.5	2.7	2.6
4-Kbyte	13.7	15.2	20.6	26.8	32.5

Packet round-trip 1 msec.

表 4 は，Object Communities の同報通信の遅延時間を示す．これは，同報通信のメッセージを送信してから，そのメッセージを自分自身が受け取るまでの遅延時間を表す．測定は，メッセージを受信する計算機の数とメッセージの大きさを変えて，SPARC station 上でおこなわれた．測定値は 100 回の連続した同報通信の遅延時間の平均値である．

測定結果は，メッセージの大きさがネットワークの帯域幅に比べて充分小さいとき，計算機の数が遅延時間に影響しないことを示している．実際のアプリケーションでは，個々のメッセージの大きさはそれほど大きくないと考えられるので，この性質は実用上，非常によい性質であるといえる．サーバを用いた方法では，計算機の数が増えたときにサーバがボトルネックになる可能性があるが，５台程度の場合は，ボトルネックになることはない．メッセージの受信順序が全体で一意になるような同報通信を実現する方法としては，Object Communities のようにサーバを使う方法[11] の他に，ISIS[3] のようにサーバを使わない方法もある．著者らの UNIX TCP/IP 上での経験では，サーバを使わない方法では，計算機の数の増加とともに遅延時間が線形的に増加することがわかっている[7]．したがって小規模な同報通信の機能を，OS 外の機能として実現する場合には，サーバを使う方法の方が優れているといえる．

6　考察

Open C++上では，新しい言語プリミティブはメタオブジェクトのクラスライブラリとして実現される．このため，一見すると，従来の C++クラスライブラリでもその言語プリミティブと同等な機能を実現できるように思える．本章では，従来の C++クラスライブラリと，Open C++のメタオブジェクトのクラスライブラリの違いについて考察する．そして類似の研究との比較をおこなう．

C++のクラスライブラリと Open C++のクラスライブラリの違いは，ライブラリが提供する機能のインタフェースである．Open C++では，言語の実現機構を変更できるので，提供する機能を言語の構文によりよく融和させることができる．例えば遠隔手続き呼出しの場合，Open C++では，メソッド呼出しの実現機構を変更するので，プログラマは遠隔手続き呼出しを，通常のメソッド呼出しと同様の構文で記述できる．C++で同様のことをするには，Arjuna[18] や Sun `rpcgen`[23] のようなスタブ生成器が必要である．Open C++をスタブ生成器付の C++と考えることもできるが，Open C++はスタブ生成をメタレベルアーキテクチャという観点から考察し，より汎用的に，遠隔手続き呼出し以外の言語プリミティブにも利用できるようにしている．

新しい言語プリミティブをクラスライブラリとして提供できるように，メソッド呼出しをトラップ可能にするというアイデアは，新しいものではなく，他にも例えば GNU C++の wrapper[24] などがある．これはメソッド呼出しを，他の演算子と同様に，多義化 (operator overload) できるようにする機能である．これを使うと C++の範囲内で，例えば遠隔手続き呼出しを Open C++のように通常のメソッド呼出しと同じ構文で実行できるようにできる．しかしながら

wrapper はメソッドの引数列や返り値を抽象化する機能が充分でないので，引数列や返り値が異なるのメソッドごとに別々の wrapper を用意しなければならない．Open C++のメタレベルでは，例えば引数列は個数や型にかかわらず `ArgPac` 型の値として扱えるが，wrapper では基本的に引数列はそのままなので，任意の引数列に適用できるような記述は wrapper ではできない．wrapper はこのような欠点を補うために synthetic type を用意しているが，これは Open C++の reify 処理だけに相当し，reflect 処理に相当する処理がない．

Open C++はメタレベルのプログラムの汎用性を高めるために，ベースレベルの情報をより抽象度の高い形で操作できるようにしている．Open C++の抽象化機能は，多相関数に似て，より包括的な記述を可能にし，プログラムの再利用性を高める．この点が，他の ad hoc な言語の拡張機構と異なり，Open C++が自己反映計算の概念に基づいて設計されていることの利点である．

7 まとめと今後の課題

本論文では，自己反映計算の機能を導入した C++である Open C++について述べた．Open C++では，メタオブジェクトを変更することで，新しい言語プリミティブを言語上で利用可能にできる．また，本論文では，Open C++上に分散処理のためのプリミティブを実装するための枠組 Object Communities について述べた．この枠組が提供するメタオブジェクトのクラスライブラリを用いることで，容易に分散処理のための新しい言語プリミティブを Open C++上に実装できる．Open C++を用いると，さまざまな分散処理のための C++の変種を容易に作成することができる．

今後の課題として，自己反映計算による実行時オーバヘッドを軽減することがあげられる．この問題は，自己反映計算の実用化のために重要であり，すでに，いくつかの自己反映計算の処理系がこの問題に取り組んでいる [15]．現在のネットワーク・ハードウェアでは，Open C++の自己反映計算に関係するオーバヘッドは，Open C++を分散処理に用いる限り充分小さいといえる．しかしながら，今後のハードウェアの改良や，Open C++の応用範囲を広げることを考慮すると，オーバヘッドの軽減は重要な課題である．著者らは，部分評価 (partial evaluation) の技術を用いることで，このオーバヘッドを大きく減少させることに成功している [32]．これについては，他の論文で詳しく述べる予定である．

また，この他の課題は，Open C++で拡張可能な範囲を広げることである．現在は，メソッド呼出しと変数アクセスの実現機構だけが拡張可能であるが，さらに Open C++の文法定義も部分的に拡張可能にすることを考えている．Open C++を並列処理などのために利用する場合，データ並列などを扱うために，新しい文を導入できることが望まれる．このような拡張も，自己反映計算によって扱えるようにすることが課題である．

謝辞

本研究にあたっては，東京大学の松岡 聡，増原 英彦，XEROX PARC の Gregor Kiczales の各氏，その他に筑波大学の加藤 和彦，Siemens AG の Frank Buschmann の各氏から有益な助言をいただきました．末筆ながら深く感謝いたします．また本稿にコメントして下さった査読者の方々に感謝します．

参考文献

[1] Bal, H. E., Kaashoek, M. F. and Tanenbaum, A. S.: Orca: A Language For Parallel Programming of Distributed Systems, *IEEE Trans. Softw. Eng.*, Vol. 18,No. 3(1992), pp. 190–205.

[2] Bal, H. E., Steiner, J. G. and Tanenbaum, A. S.: Programming Languages for Distributed Computing Systems, *Computing Surveys*, Vol. 21,No. 3(1989), pp. 261–322.

[3] Birman, K. P. and Joseph, T. A.: Reliable Communication in the Presence of Failures, *ACM Trans. Comp. Syst.*, Vol. 5,No. 1(1987), pp. 47–76.

[4] Birrell, A. D. and Nelson, B. J.: Implementing Remote Procedure Calls, *ACM Trans. Comp. Syst.*, Vol. 2,No. 1(1984), pp. 39–59.

[5] Buschmann, F., Kiefer, K., Paulisch, F. and Stal, M.: The Meta-Information-Protocol: Run-Time Type Information for C++, *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture*(Yonezawa, A. and Smith, B. C.(eds.)), 1992, pp. 82–87.

[6] Chiba, S.: Open C++ Programmer's Guide, Technical Report 93-3, Dept. of Information Science, Univ. of Tokyo, Tokyo, Japan, 1993.

[7] Chiba, S., Kato, K. and Masuda, T.: Exploiting a Weak Consistency to Implement Distributed Tuple Space, *Proc. of the 12th IEEE Int'l Conf. on Distributed Computing Systems*, 1992, pp. 416–423.

[8] Chiba, S. and Masuda, T.: Designing an Extensible Distributed Language with a Meta-Level Architecture, *Proc. of the 7th European Conference on Object-Oriented Programming*, LNCS 707, Springer-Verlag, 1993, pp. 482–501.

[9] Danvy, O. and Malmkjær, K.: Intensions and Extensions in a Reflective Tower, *Proc. of ACM Conf. on Lisp and Functional Programming*, 1988, pp. 327–341.

[10] Herlihy, M. and Liskov, B.: A Value Transmission Method for Abstract Data Types, *ACM Trans. Prog. Lang. Syst.*, Vol. 4,No. 4(1982), pp. 527–551.

[11] Kaashoek, M. F. and Tanenbaum, A. S.: Group Communication in the Amoeba Distributed Operating System, *Proc. of the 11th IEEE Int'l Conf. on Distributed Computing Systems*, 1991, pp. 222–230.

[12] Kiczales, G., des Rivières, J. and Bobrow, D. G.: *The Art of the Metaobject Protocol*, The MIT Press, 1991.

[13] Lamping, J., Kiczales, G., Rodriguez, L. and Ruf, E.: An Architecture for an Open Compiler, *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture*(Yonezawa, A. and Smith, B. C.(eds.)), 1992, pp. 95–106.

[14] Maes, P.: Concepts and Experiments in Computational Reflection, *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 1987, pp. 147–155.

[15] Masuhara, H., Matsuoka, S., Watanabe, T. and Yonezawa, A.: Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently, *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 1992, pp. 127–144.

[16] Matsuoka, S., Watanabe, T. and Yonezawa, A.: Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming, *Proc. of European Conf. on Object-Oriented Programming '91*, LNCS, No. 512, Springer-Verlag, 1991, pp. 231–250.

[17] Okamura, H., Ishikawa, Y. and Tokoro, M.: AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework, *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture*(Yonezawa, A. and Smith, B. C.(eds.)), 1992, pp. 36–47.

[18] Parrington, G. D.: Reliable Distributed Programming in C++: The Arjuna Approach, *Proc. of USENIX C++ Conference*, 1990, pp. 37–50.

[19] Rao, R.: Implementational Reflection in Silica, *Proc. of European Conf. on Object-Oriented Programming '91*, LNCS 512, Springer-Verlag, 1991, pp. 251–267.

[20] Shapiro, M.: Structure and Encapsulation in Distributed Systems: The Proxy Principle, *Proc. of the 6th IEEE Int'l Conf. on Distributed Computing Systems*, 1986, pp. 198–204.

[21] Smith, B. C.: Reflection and Semantics in Lisp, *Proc. of ACM Symp. on Principles of Programming Languages*, 1984, pp. 23–35.

[22] Stroustrup, B.: *The C++ Programming Language*, Addison-Wesley, 2nd edition, 1991.

[23] Sun Microsystems: *Network Programming Guide*, Sun Microsystems, Inc., 1990.

[24] Tiemann, M. D.: Solving the RPC problem in GNU C++, *Proc. of USENIX C++ Conference*, 1988, pp. 343–361.

[25] U.S. Dept. of Defense: *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, 1983.

[26] Wand, M. and Friedman, D. P.: The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower, *Meta-Level Architectures and Reflection*(Maes, P. and Nardi, D.(eds.)), Elsevier Science Publishers B.V., 1988, pp. 111–134.

[27] Watanabe, T. and Yonezawa, A.: Reflection in an Object-Oriented Concurrent Language, *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 1988, pp. 306–315.

[28] Watanabe, T. and Yonezawa, A.: An Actor-Based Metalevel Architecture for Group-Wide Reflection, *Foundation of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, 1990, Proceedings.*(de Rover et al., J. W.(ed.)), LNCS 489, Springer-Verlag, 1991, pp. 405–425.

[29] Wegbreit, B.: *Studies in Extensible Programming Languages*, Outstanding dissertations in the computer sciences, Garland Publishing, Inc., 1980. (based on the author's thesis, Harvard, 1970).

[30] Yokote, Y.: The Apertos Reflective Operating System: The Concept and Its Implementation, *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 1992, pp. 414–434.

[31] 加藤, 大堀, 村上, 益田: 高階遠隔手続き呼出しに基づいた分散 C 言語について, **コンピュータソフトウェア**, Vol. 9,No. 3(1992), pp. 65–82.

[32] 千葉, 益田: Open C++ における自己反映計算の処理系の最適化技法, **ソフトウェア科学会第 *10* 回大会論文集**, 日本ソフトウェア科学会, Jun. 1993, pp. 41–44.