ソフトウェア工学 124－2
（1999．10．21）

リフレクションを利用したCORBA APIの改善

藤枝 和宏　渡部 卓雄　落水 浩一郎

北陸先端科学技術大学院大学 情報科学研究科

〒923-1292 石川県能美郡辰口町旭台 1-1
Phone: 0761-51-1260, Fax: 0761-51-1149
E-mail: {fujieda,takuo,ochimizu}@jaist.ac.jp

概要

本論文では、リフレクションの可能なプログラミング言語を利用して、CORBAのアプリケーション開発手順を簡略化する手法と、利用方法の複雑なCORBAのAPIを簡略化する手法を示す。リフレクションを利用することで、必要なスタブとスケルトンをアプリケーションの実行時に生成して取り込むことが可能になり、アプリケーションの開発手順を簡略化できる。さらに、リフレクションを利用することで、CORBAの遅延同期呼び出しやDynAny APIを利用する際のプログラミングの手間を削減できる。

Improvement of CORBA API Using Reflection

Kazuhiro Fujieda　Takuo Watanabe　Koichiro Ochimizu

School of Information Science,
Japan Advanced Institute of Science and Technology

Asahidai 1-1, Tatsunokuchi, Ishikawa 923-1292
Phone: +81-761-51-1260, Fax: +81-761-51-1149
E-mail: {fujieda,takuo,ochimizu}@jaist.ac.jp

Abstract

In this paper, we show the usefulness of the capabilities of a reflective programming language in developing CORBA applications. We show two examples. One can make application development process in CORBA simpler and easier. Stubs and skeletons in CORBA applications can be generated and incorporated at runtime with the use of reflection. Another can make programming with complicated CORBA APIs simpler and easier, too. We can reduce programming efforts of the deferred synchronous invocation API and the DynAny API.

1 はじめに

CORBA(Common Object Request Broker Architecture) は、OMG(Object Management Group) によって策定された分散オブジェクト指向環境の標準規格である [1]。CORBA を利用すると、ネットワークを介して提供するサービスを分散オブジェクトとして実装することができて、クライアントは、分散オブジェクトのメソッド[1]起動を介してそのサービスを利用できる。

分散オブジェクトとそのクライアントは、オブジェクト指向モデルではない言語を含め、さまざまなプログラミング言語で記述できる。オブジェクト指向言語を利用する場合には、クラス定義の構文を用いて分散オブジェクトの実装を記述し、その言語のオブジェクトと同じ構文で分散オブジェクトを操作可能である。

分散オブジェクトとプログラミング言語のオブジェクトの対応付けは、スタブとスケルトンと呼ばれるプログラムによって実現される。これらは、分散オブジェクトの仕様を記述した IDL(Interface Description Lanugage) ファイルから IDL コンパイラによって生成される。CORBA のアプリケーション開発では、開発者の記述したファイルだけでなく、スタブやスケルトンを格納したファイルも関係するため、アプリケーションの構築手順が非常に煩雑になっている。

著者らは、この問題を解決する一つのアプローチとして、リフレクションの可能なプログラミング言語を利用して、IDL ファイルの定義を格納したインタフェースリポジトリを元に、アプリケーションの実行時に必要なスタブやスケルトンを生成して組み込む手法を提案している [2]。本論文では、この CORBA のアプリケーション実行環境についてより詳しく述べるとともに、リフレクションを利用することでプログラミングが複雑な CORBA の API を簡略化できることを示す。

以下、本稿の構成は 2 節で CORBA のアプリケーション開発手順の問題点を論じ、それを解決する、インタフェースリポジトリを利用して、必要なスタブとスケルトンを実行時に生成する実行環境を提案する。3 節では、この実行環境をオブジェクト指向スクリプト言語 Python の提供しているリフレクションを利用して実現する方法を示す。4 節では、リフレクションの可能なプログラミング言語が提供している、メソッド起動時の振る舞いをカスタマイズする能力を利用して、CORBA の二つの API、遅延同期呼び出しと DynAny API の実行を簡略化する方法を示す。5 節では、他の研究で用いられているスタブやスケルトンを動的に生成するアプローチと本研究で用いているアプローチを比較する。6 節でまとめと今後の課題を述べる。

[1]CORBA ではオペレーションと呼ばれる

2 開発手順の問題点とその解決策

CORBA のアプリケーション開発は、分散オブジェクトのインタフェースを IDL(Interface Description Language) で記述したファイルを作成することから始まる。IDL ファイルを作成したら、CORBA の IDL コンパイラを実行して、IDL ファイルからスタブとスケルトンを生成する。**スタブ**は分散オブジェクトを操作する際に必要なプログラムであり、**スケルトン**は分散オブジェクトの実装に必要なプログラムである。

CORBA のアプリケーション開発に広く用いられている C++や Java では、これらのプログラムをアプリケーションに取り込むために、コンパイルとリンク (Java ではサーバとクライアントへのクラスファイルの配置) が必要である (図 1)。そのため、CORBA のアプリケーション開発はアプリケーションを実行可能にするまでの手順が煩雑であり、特に IDL ファイルの変更が生じる場合にはターンアラウンドが長くなる。

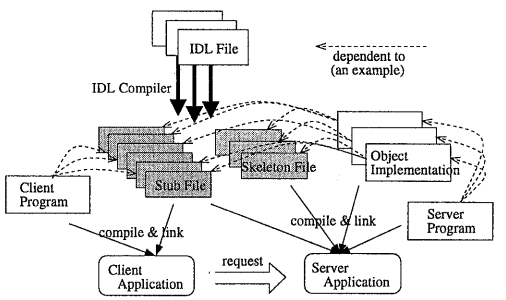

図 1: CORBA のアプリケーション開発手順の概要

CORBA のアプリケーションをコンパイルやリンクの作業の不要なスクリプト言語を用いて記述することで、この問題はある程度解消できる。たとえば、オブジェクト指向スクリプト言語**Python**[3] では、**Fnorb**[4] という CORBA の実装を利用することで、分散オブジェクトの実装とクライアントの両方を Python で記述することができる。ただし、Fnorb においても、IDL コン

パイラを用いてIDLファイルからスタブとスケルトンを生成し、アプリケーションに取り込む必要がある。

IDLコンパイラを用いてIDLファイルからスタブとスケルトンを生成する手法には、サーバとクライアントで同じIDLファイルの異なるバージョンを誤って用いたときに、インタフェースの不整合を起こす問題がある。アーキテクチャやプログラミング言語がサーバとクライアントで異なる場合には、スケルトンとスタブが異なるIDLコンパイラで別々に生成されるため、この問題が起こりやすい。スクリプト言語は、別の言語で記述されたサーバのテストプログラムの記述に用いられることが多く、この問題を引き起こす可能性が高い。

そこで、これらの問題を解決するために、CORBAのインタフェースリポジトリに格納された分散オブジェクトの仕様から、アプリケーションの実行時に必要なスタブやスケルトンを生成して取り込む手法を提案する。この手法により、スタブやスケルトンを格納したファイルを直接扱う必要がなくなるため、開発手順が簡略化される。また、サーバとクライアントがインタフェースリポジトリに格納された定義から、必要なスタブとスケルトンを直接生成するのでインタフェースの不整合を防ぐことができる(図2)。

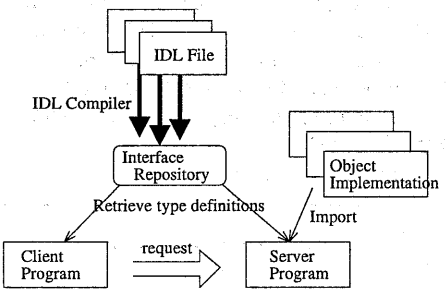

図2: インタフェースリポジトリを利用した実行環境

3 リフレクションを利用したスタブとスケルトンの実行時生成

実行時にスタブとスケルトンを生成してアプリケーションに取り込むには、実行時にデータ型やクラス定義を生成してアプリケーションに取り込む必要がある。これは、リフレクションを提供しているプログラミング言語を利用することで可能になる。本節では、オブジェクト指向スクリプト言語Pythonの提供しているリフレクション能力を利用して、インタフェースリポジトリに格納された定義から、実行時にスタブ/スケルトンを生成して取り込む手法を示す。

3.1 スタブ/スケルトンの実行時生成

リフレクションの可能なプログラミング言語では、プログラム自身を操作するプログラムや、プログラムの実行環境を操作するプログラムを記述することができる。前者を**Linguistic Reflection**、後者を**Behavioral Reflection**と呼ぶ[5]。Pythonではこの両者が利用可能である。スタブとスケルトンを実行時に生成するために必要な能力はLinguistic Reflectionである。

Pythonのプログラムの構成要素、すなわちモジュールやクラスやメソッドなどは、Pythonのオブジェクト[2]として実装されている。これらのオブジェクトの属性を参照、変更することで、プログラムの参照、改変が可能である。newモジュールやexec文を利用して、新たにこれらのオブジェクトを生成することもできる。

このPythonのLinguistic Reflectionの能力を利用して、インタフェースリポジトリに格納された分散オブジェクトの仕様を元に、IDLコンパイラが生成するものと同様なスタブやスケルトンを生成することで、必要なスタブとスケルトンを実行時に生成することが可能になる。

インタフェースリポジトリは、IDLファイルの内容を構文木の形で格納し、構文木の各ノードを分散オブジェクトとして提供するCORBAで規定されたサーバである。インタフェースリポジトリを利用することで、CORBAのアプリケーションは実行時にIDLファイルで定義された内容を参照することが可能になる。Fnorbを含むほとんどのCORBAの実装では、インタフェースリポジトリの実装と、IDLファイルの内容をインタフェースリポジトリに格納可能なIDLコンパイラが提供されている。

IDLファイルで記述される分散オブジェクトの仕様には、分散オブジェクトのインタフェースと、インタフェースで用いられる配列や構造体などのユーザ定義型の定義が含まれる。これらの定義はインタ

[2]正確には組込み型の値だが、オブジェクトと同様に扱える

フェースリポジトリに格納される際に、リポジトリIDが割り当てられる。インタフェースリポジトリから定義を取り出す際には、リポジトリIDか定義の絶対スコープ名(図3の*Interface1*インタフェースでは::*Module1*::*Interface1*)のどちらかが必要である。

実行時にアプリケーションに必要な必要なスタブとスケルトンを生成するためには、少なくとも、アプリケーションが利用する分散オブジェクトのインタフェースの定義のリポジトリIDか絶対スコープ名が必要である。クライアントに必要なスタブを生成する際には、クライアントの取得したオブジェクトリファレンスに含まれているリポジトリIDを利用することができる。

```
module Module1 {
    struct AtrType {
      string name;
      long value;
    };
    interface Interface1 {
      attribute AtrType attr1;
    };
};
```

図3: IDLファイルの例

3.2 オブジェクトリファレンスを利用したリポジトリIDの取得

分散オブジェクトを利用するクライアントは、最初にCORBAのAPIかNamingServiceを用いて分散オブジェクトのオブジェクトリファレンスを取得する。Fnorbを含むほとんどのCORBAの実装では、オブジェクトリファレンスに分散オブジェクトの型を表すIDとして分散オブジェクトのインタフェースのリポジトリIDが含まれている。このリポジトリIDを元にインタフェースリポジトリから定義を取り出して、対応するスタブを生成することができる。インタフェースの定義の参照関係をたどることで、継承しているインタフェースやユーザ定義型など、インタフェースの利用に必要なスタブを順次生成することができる(図4)。生成したスタブは、分散オブジェクトを操作するプログラムから利用できるように、オブジェクトリファレンスの取得を実行したプログラムの名前空間に挿入する。

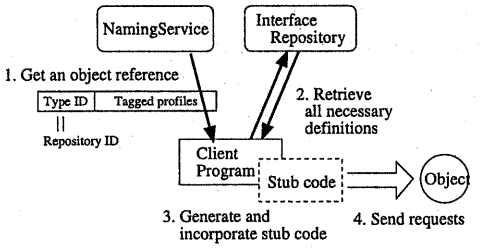

図4: オブジェクトリファレンスを元にスタブを生成する

図3のIDLファイルで定義されたインタフェース*Interface1*を利用するには、通常はIDLコンパイラを起動してスタブファイルを生成し、モジュールを取り込むimport文を用いてimport *Module1*として明示的にプログラムに取り込まなければならない。上記の手法では、IDLファイルの内容がインタフェースリポジトリに格納されているなら、IDLコンパイラの実行もimport文の実行も不要である(図5)。

```
#オブジェクトリファレンスの取得
ior = open('server.ref', 'r').read())
#文字列からオブジェクトリファレンスを得るAPI
srv = orb.string_to_object(ior)
#IDLの構造体に対応するスタブを利用して
atr =Module1.AttrType("hoge", 100)
#属性の値を設定
srv._set_attr1(atr)
```

図5: プログラミング例

Behavioral Reflectionによる名前空間の操作　名前空間を操作する際には、Pythonの提供している実行環境に関するプログラミング能力(Behavioral Reflection)を利用する。具体的には、Pythonの例外処理の実装を参照するAPIを介してスタックフレームにアクセスして、オブジェクトリファレンスの取得を行ったプログラムの名前空間を取り出す。名前空間はPythonの連想配列で実装されており、これを書き換えることで名前空間の内容を変更できる。

ところで、この手法には問題が二つある。一つは、オブジェクトリファレンスの取得を行った名前空間でしか、スタブを利用することができないことである。Python ではモジュールごとに名前空間が異なるため、関数やメソッドを介して他のモジュールにオブジェクトリファレンスを渡すと、そのモジュールでは分散オブジェクトの操作にスタブを利用できない。もう一つは、スケルトンの実行時生成に応用できないことである。

3.3 モジュール機構のカスタマイズ

上記の問題に対処するために、スタブやスケルトンのモジュールを明示して取り込む方法も提供する。これを Python のモジュール機構をカスタマイズして実現する方法を示す。

3.3.1 Python のモジュール

Python のモジュールの概念は、ディレクトリやファイルに対応している。ディレクトリ`Module1/`のファイル`__init__.py` にプログラムを記述して、`import Module1` 実行すると、このファイルで定義されたクラス`Class1` を`Module1.Class1` という名前で参照できる。このファイルが更新されたときには、組み込み関数`reload` を用いて`reload(Module1)` とするとモジュールの内容を更新できる。

3.3.2 リフレクションを利用したモジュール機構のカスタマイズ

Python では`import` 文の実装が`__import__`という組み込み関数として公開されている。この関数と`reload` 関数を再定義することにより、Python のモジュール機構の振る舞いを、Python のプログラムでカスタマイズできる (Behavioral Reflection)。モジュール機構をカスタマイズすることで、Python のモジュール機構を用いてスタブやスケルトンを実行時に生成して、アプリケーションに取り込むことができる。

具体的には、`__import__`関数を以下のように再定義する。

1. 指定されたモジュールが存在する場合には通常通り振る舞う
2. 存在しない場合には、モジュール名 (例えば *Module1*) を IDL の絶対スコープ名 (`::`*Module1*) に変換して、インタフェースリポジトリから IDL のモジュール定義を取得する。
3. モジュールに含まれる定義のスタブかスケルトンを生成する (Fnorb の IDL コンパイラに習い、モジュール名の最後が`_skel` の場合はスケルトンとする)
4. 生成したスタブやスケルトンを格納した Python のモジュールを作成する

生成されたモジュールに対して`reload` が実行された場合には、インタフェースリポジトリから定義を読み込んでスタブやスケルトンを更新するように`reload` も再定義する。

上記のように`import` 文の振る舞いをカスタマイズすると、同じ IDL のモジュールについて、IDL コンパイラの生成したスタブやスケルトンを取り込むための`import` 文と、実行時に生成して取り込むための`import` 文がまったく同じになる。この性質により、変更の可能性がない IDL のモジュールについては、IDL コンパイラを用いてスタブやスケルトンを事前に生成しておくことで、実行時の自動生成を抑制し実行コストを削減できる。

4 リフレクションを利用した API の改善

ここまでは、リフレクションを利用して CORBA のアプリケーション開発手順を簡略化する方法を論じてきた。ここまで論じてきた手法で簡略化されるのは、アプリケーションの開発手順だけであり、プログラミングについては従来とほとんど変わらない。本節では、リフレクションを提供するプログラミング言語の提供する二つの能力のうち、Behavioral Reflection を利用することで、プログラミングの複雑な CORBA の API を簡略化できることを示す。

4.1 遅延同期呼び出しの簡略化

4.1.1 遅延同期呼び出しの問題点

メソッド実行時のスタブ生成を利用することで、CORBA のオペレーションの遅延同期呼び出しの言語インタフェースを改善することができる。

CORBA では oneway を指定したオペレーション以外は、呼び出すとサーバからオペレーションの返値が戻るまで待つ同期呼び出しである。返値を待たずに別の処理を行い、後で結果を受け取る遅延同期 (Deferred Synchronous) 呼び出しもサポートされているが、現状ではこれを利用するには DII を用いなければならない。DII ではメソッド呼び出しの構文も例外処理も利用することができないため、プログラミングが非常に煩雑になる (図 6)。

```
// 引数と返値の設定
org.omg.CORBA.Request req =
    srv._request("insert");
req.add_in_arg().insert_string("Dummy");
org.omg.CORBA.Any v_any =
    req.add_in_arg();
VValueHelper.insert(v_any, v);
req.add_in_arg().insert_boolean(false);
// 遅延同期呼び出しの実行
req.send_deferred();
...
...
// 結果を受け取る
req.get_response();
// 発生した例外を処理
org.omg.CORBA.ExceptionList exlist =
   req.exceptions();
if (exlist > 0) {
  omg.org.CORBA.TypeCode extcode =
    exlist.item(0);
  if (extcode.equal(
      NameTable.AlreadyExistHelper.type()))
  {
    System.err.println(
       "\'" + name +
       "' already exists.");
  }
}
```

図 6: 遅延同期呼び出しの例

現在策定中の CORBA の新しい非同期呼び出しの規格 CORBA Messaging [6] では、IDL コンパイラにインタフェースの持つオペレーションごとに、同期、遅延同期、非同期の 3 種類のメソッド定義を生成させることで、通常のメソッド呼び出しと同じ方法で遅延同期呼び出しや非同期呼び出しを実行することが可能になっている。しかし、そのオペレーションで本当に遅延同期呼び出しが行われるかどうかにかかわらず、すべてのオペレーションについて遅延同期呼び出しのメソッド定義を生成する必要があるため、スタブが非常に大きくなるという問題がある。

4.1.2 オペレーション実行時のスタブ生成

実際に遅延同期呼び出しが行われた時点で、対応するスタブを生成することができれば、無駄なスタブを生成することなく遅延同期呼び出しを簡略化することが可能になる。これは、遅延同期呼び出しだけでなく、多数のオペレーションを持つ分散オブジェクトの一部のオペレーションしか利用しないクライアントについても、実際には利用しないメソッドの定義が生成されるのを防ぐことができるという点で有用である。

Behavioral Reflection が可能なプログラミング言語では、実行環境がオブジェクトのメソッドを呼び出す際の振る舞いをプログラムから変更することができる。この機能を利用して、メソッドの定義が存在しないときの実行環境の振る舞いを変更することで、メソッドが実行されたときにインタフェースリポジトリを参照して、オペレーションに対応するメソッドの定義を生成することが可能になる。

Python では定義されていないメソッドが呼ばれたときのインタプリタの振る舞いをカスタマイズすることができる。クラスを定義するときに、__getattr__という名前のメソッドを定義しておくと、そのクラスのインスタンスについて未定義のメソッドが呼ばれたときに、エラーの例外を発生させる変わりにこのメソッドが実行される。

そこで、スタブを生成する際にメソッドの定義として__getattr__という名前のメソッドだけを定義しておく。実際にメソッドが呼ばれたときに、呼ばれたメソッドの名前を引数にしてこのメソッドが実行されるので、その名前に対応するメソッドの定義を生成して、それを実行すればよい。遅延同期呼び出しの場合には、CORBA Messaging に従いオペレーションの名前の前にsendp_というプレフィックスの付いたメソッドが呼ばれたときに、遅延同期呼び出し用の定義を生成する。

4.2 DynAny API

CORBA には、オブジェクトリファレンスを含む任意のデータ型を扱うことができる Any 型が用意されている。Any 型は、CORBA のデータ型を表現する TypeCode と、そのデータ型の値を表すバイト列を組にしたものである。

TypeCode とバイト列の対応を正しく扱うために、

Any 型に値を格納したり、値を取り出したりする際には、C++の場合には Any クラスと値の型の間の<<, >> 演算子を、Java の場合には値の型の Helper クラスで定義されたメソッドを使用する。ユーザ定義型の値を Any 型として扱うには、そのデータ型に対応するこれらの演算子やメソッドの定義が必要なので、スタブを取り込む必要がある。

最近の CORBA の規格では、スタブを用いずに Any 型を扱うことができる DynAny という API が定められているこの API を利用するとプログラミング言語の型システムに依存せずに、Any 型の値を扱うことができる。たとえば、図 3 で定義されている *AttrType* 構造体の値をスタブなしで生成するプログラムは、図 7 のように記述することができる。

```
using CORBA;
StructMemberSeq mems(2);
Any any_val;

// struct MyStruct を表す
// TypeCode を生成する
mems[0].name = string_dup("name");
mems[0].type = TypeCode::
  _duplicate(CORBA::_tc_string);
mems[1].name = string_dup("value");
mems[1].type = TypeCode::
  _duplicate(CORBA::_tc_long);
CORBA::TypeCode_var struct_tc =
  orb->create_struct_tc(
        "IDL:Module1/AttrType:1.0",
        "AttrType", mems);
// DynAny を利用して
// struct MyStruct 型の値を生成する
DynStruct_ptr dyn_struct =
  orb->create_dyn_struct(struct_tc);
dyn_struct->insert_string("hoge");
dyn_struct->insert_long(10000);
// Any 型に変換する
any_val = dyn_struct->to_any();
```

図 7: DynAny の例

この API を Python の属性に値を代入するときの振る舞いをカスタマイズするためのメソッド `__setattr__` を利用して簡略化すると、図 8 のように簡潔に記述することが可能になる。

```
dyn_struct = orb.create_dyn_struct()
dyn_struct.name = "hoge"
dyn_struct.value = CORBA.long(10000);
any_val = dyn_struct.to_any();
```

図 8: リフレクションを利用して簡略化した DynAny

この簡略化した DynAny API では、`create_dyn_any()` メソッドによって `DynStruct` のインスタンスが生成され、そのインスタンスの属性が変更されるときに、あらかじめ `DynStruct` クラスで定義しておいた `__setattr__` が呼び出される。この `__setattr__` は、代入された属性の名前と値を引数として呼び出されるので、名前と値の型に基づいて構造体を表す TypeCode を生成しつつ、インスタンスに新しい属性と値を挿入して構造体の値を作成していく仕組みになっている。

5 関連研究

IDL コンパイラを用いたスタブやスケルトンの生成を廃止して、インタフェースリポジトリを利用することで、CORBA の言語インタフェースを実現するアプローチとしては LuaORB[7] や CorbaScript[8] がある。

LuaORB は、リフレクションの可能なプログラミング言語 Lua のメソッド呼び出しの振る舞いをカスタマイズする機構を用いて、CORBA の言語インタフェースを実現している。分散オブジェクトのオペレーションが実行されたときに、インタフェースリポジトリのオペレーションの定義を参照して、リクエストプロトコルを生成することで、分散オブジェクトへのアクセスを可能する。本稿のアプローチとは異なりスタブは生成されない。そのため、ユーザ定義型に対応するスタブも利用することができず、ユーザ定義型を扱うプログラミングが煩雑になる。

CorbaScript は CORBA のアプリケーション記述用のスクリプト言語である。CorbaScript では、言語の名前空間とインタフェースリポジトリの名前空間が結合されており、言語側で未定義の名前にアクセスすると、対応する定義が自動的にインタフェースリポジトリから取り込まれる。このアプローチでは、スタブやスケルトンを取り込む記述が不要で、ユーザ定義型を扱うこともできる。ただし、既存の言語のリフレクションを

利用している本稿とは異なり、言語から処理系まで新規に作成しているため、クラスライブラリが存在しないこと、言語の学習コストがかかることが問題である。

6 まとめと今後の課題

本論文では、Python の提供するリフレクションの能力を用いて、CORBA のアプリケーション開発に必要なスタブとスケルトンをインタフェースリポジトリを介して実行時に生成しアプリケーションに取り込む手法と、利用方法の複雑な CORBA の API を簡略化する方法を示した。

スタブの生成と取り込みでは、Python の提供する Linguistic Reflection を利用してスタブとスケルトンを生成し、名前空間に関する Behavioral Reflection を利用してアプリケーションに取り込む二つの手法を示した。これらの手法により、IDL コンパイラを用いてスタブとスケルトンを生成する場合と比べて、アプリケーションの開発手順が簡略化し、サーバ・クライアント間のインタフェースの不整合を防ぐことが可能になる。

CORBA の API の簡略化としては、Python の提供するメソッド呼び出しに関する Behavioral Reflection を利用して、遅延同期呼び出しを簡略化する際に無駄なスタブが生成されるのを防ぐ方法、およびオブジェクトの属性の代入に関する Behavioral Reflection を利用して、Dynamic Any API を実行する際に TypeCode を生成する手間を簡略化する方法を示した。

本稿のアプローチでは、リフレクションの可能なプログラミング言語の持つ、アプリケーションの振る舞いを動的に変更できる特性が活用されていない。本稿のアプローチを元に、IDL ファイルの変更に対して、アプリケーションを止めることなくその影響を吸収する機構を実現することについて、今後検討を進めていきたいと考えている。

また、現在の実装では、リフレクションの可能な言語として Python を利用しているが、Java において Linguistic Reflection を可能にするアプローチも報告されているので [5, 9]、これらのアプローチを参考に同様の環境を Java で実現することも今後検討したい。

謝辞

本研究の遂行にあたり、(株)PFU 研究所 熊谷 章氏および東田 雅宏氏のご支援およびご討論に感謝します。また、本研究は文部省科学研究費基盤 C「ソフトウェア分散開発における共有情報の変更管理法」(課題番号 10680345) および特定領域研究 (A)(1)「発展機構を備えたソフトウェア構成原理の研究」(課題番号 09245105) の援助の下に実施された。

参考文献

[1] Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 2.2*, February 1998. formal/98-02-01.

[2] 藤枝和宏, 渡部卓雄, 落水浩一郎. リフレクションを利用した corba 言語インタフェースの改善. 日本ソフトウェア科学会第 16 回大会論文集, pp. 93–96, September 1999.

[3] Python language website. http://www.python.org.

[4] Martin Chilvers. *fnorb version 1.0*. CRC for Distributed Systems Technology, February 1999. http://www.dstc.edu.au/Fnorb.

[5] Graham Kirby, Ron Morrison, and David Stemple. Linguistic reflection in java. *Software: Practice and Experience*, Vol. 28, No. 10, pp. 1045–1077, 1998.

[6] Object Management Group. *CORBA Messaging*, May 1998. OMG TC Document orbos/98-05-06.

[7] Roberto Ierusalimschy, Renato Cerqueira, and Noemi Rodriguez. Using reflexivity to interface with CORBA. In *IEEE International Conference on Computer Languages (ICCL'98)*, May 1998.

[8] Laboratoire d'Informatique Fondamentale de Lille and Object-Oriented Concepts, Inc. *OMG CORBA Scripting Language RFP Revised Submission*, December 1998. OMG TC Document orbos/98-12-09.

[9] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. In *OOPSLA'98 Proceedings*, October 1998.