システムソフトウェアと
オペレーティング・システム 74−24
マルチメディア通信と分散処理 81−24
（1997. 2. 28）

ORBを用いた分散処理システムの開発支援環境

橋本 圭介　　貫井 春美

株式会社 東芝 研究開発センター
システム・ソフトウェア生産技術研究所

分散処理システムのプラットフォームとして，CORBA(Common Object Request Broker Architecture)準拠のORB(Object Request Broker)を用いることにより，透過性や柔軟性の高いシステムの構築が容易となり，生産性向上が期待される．反面，信頼性や性能の確保といった問題が予想される．報告者らは，このような課題を解決し，ORBによる分散処理システムの開発を一貫して支援するシステム(方法論とツール)の研究開発を進めている．その中心機能である「分散アプリケーション記述言語」は，信頼性を損なう要因となるORB特有の作法や，異なるORBの差を隠蔽し，性能や信頼性を測定するための機能を自動的に組み込むプログラミング言語である．本稿では，分散アプリケーション記述言語の言語仕様と試作結果，記述例を基にした評価について述べ，実用性を考察する．

A Developing Support Environment for Distributed Systems using ORB

Keisuke Hashimoto, Harumi Nukui

Systems and Software Engineering Lab., R&D Center, Toshiba Corp.

It's expected to be able to develop the distributed system which have flexibility and transparency by using ORB(Object Request Broker) compliance with CORBA(Common Object Request Broker Architecture). But it's suspected to occur problems concerned with performance and reliability.

We are now researching and developing the support system to solve above problems. Distributed Application Described Language(DADL) is a main function of this system. Application developers can describe programs using DADL, without consciousness of the particular point of ORB programming and the differences among many ORB systems. Further more, the application programs described by DADL is automatically plugged in mechanism to keep performance and reliability.

In this paper, we describe the basic specification, prototyping, and evaluation of DADL and consider availability of it.

1 はじめに

情報処理システムは，異機種分散環境での分散処理システムとなる傾向にある．この構築にあたっては，標準的な基盤環境(プラットフォーム)が必要となる．

異機種分散システムのプラットフォームとして，OMG(Object Management Group)が標準化を続けているCORBA(Common Object Request Broker Architecture)に準拠したORB(Object Request Broker)がある．ORBをプラットフォームとすることにより，透過性や柔軟性の高いシステムの構築が行ないやすくなり，分散処理システム開発の生産性向上が望めるという利点がある．反面，信頼性や性能の確保において，問題が予想される．

報告者らは，上記課題を解決し，ORBをプラットフォームとした分散処理システムの開発を円滑に行なうための支援環境に関する研究・開発を進めている．

本稿では，この支援環境の中心である「分散アプリケーション記述言語」の機能と試作結果，および，実用性について報告する．

2 ORBをプラットフォームとした分散処理システム開発の利点と問題点

ORBをプラットフォームとして分散処理システムを開発する場合，次のような利点がある．

- **透過性の実現が容易**
  ネットワークや動作環境を意識しない，透過性の高いシステムの開発が行なえる．
- **柔軟性の高いシステムが構築できる**
  ネットワーク上のオブジェクトの配置を変更しても，アプリケーションプログラムには影響を及ぼさない．したがって，アプリケーションを実際に動作させてから，オブジェクトの配置を変更し，アプリケーションの性能を向上させることが行ないやすい．

このような利点により，生産性の向上が期待できるので，分散処理システムのプラットフォームとしてORBを用いることは，有効である．しかし，次のような問題点もある．

- **信頼性や性能確保に問題がある**
  - 上記利点は，ORBが通信部分を隠蔽することにより実現されている．これは，大きな利点である反面，実際にどのように通信が行なわれているかの把握が困難になり，通信に関する不具合の解析や性能の保証・チューニングが難しくなる面があるという問題点にもなっている．
  - オブジェクトの生成・利用・削除などの手順やオブジェクト間のリクエスト送受信時におけるメモリ管理などで，ORB特有の留意点が多い．信頼性の高いシステムを開発するためには，十分留意する必要がある．

3 開発支援環境のシステム構成

分散処理システムの開発において，性能や信頼性を確保するために，テストやチューニングの重要性が高い．報告者らは，ORBをプラットフォームとした分散処理システムの開発支援環境として，上記のフェーズに注目し，図1に示すような，開発の下流工程を一貫して支援するためのツールと方法論の研究・開発を進めている．

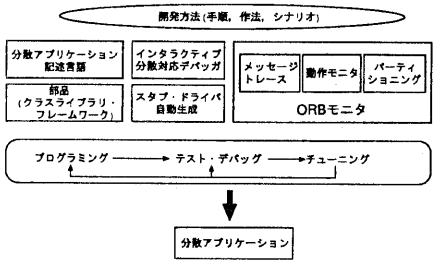

図 1: 開発支援環境・システム構成図

「分散アプリケーション記述言語」は，分散処理システムを記述するための抽象度の高いプログラミング言語である．この言語で記述されたアプリケーションは，テストやチューニングに必要な情報，すなわち，アプリケーションの動作状況やリソースの利用状況に関する情報を出力する．この情報に基づき，「ORBモニタ」がアプリケーションの動作を解析し，オブジェクト間のメッセージ交換の確認や，ボトルネックの抽出を可能とする．この情報を基に，パーティショニング(オブジェクトの再配置)を行ないながらテスト，チューニングを進め，結果をプログラミングフェーズにフィードバックする．このサイクルを繰り返すことにより，システムが完成する．

以下では，この支援環境の中心である，分散アプリケーション記述言語とその処理系について述べる．

4 分散アプリケーション記述言語

ORBをプラットフォームとした分散処理システムの開発を支援する上で，言語レベルでの支援として必要な機能として，次のようなものが挙げられる．

- **抽象度の高いレベルでのプログラミング**
  ORB を利用する場合に特有な手順や留意点を隠蔽する.
- **実システム (ORB) の隠蔽**
  分散オブジェクトのライフサイクルに関する手順や，メモリ管理に関する手順などが ORB 製品によって異なっているので，この違いを吸収する.
- **性能・信頼性の作り込み**
  性能や信頼性の低下を抑制する．また，チューニングに必要な情報を提供する.

分散アプリケーション記述言語は，ORB を利用する場合に特有な手順や留意点を隠蔽し，かつ，ORB ごとの違いを吸収するような言語となっている．したがって，ORB 特有な点や，ORB ごとの相違点を意識することなく，分散処理システムが開発できる．また，3節で述べたように，この言語で記述されたアプリケーションは，テストやチューニングに必要な情報を出力する.

4.1 従来の ORB アプリケーション開発手順

ORB アプリケーションの従来の開発手順を以下に示す.

(1) IDL(Interface Definition Language) を用いて，インタフェース定義を行なう.
(2) IDL コンパイラにより，インタフェース定義をコンパイルする.
(3) (2) により，実装言語 (例えば C++) で書かれたインプリメンテーション定義のスケルトンが得られ，これに処理の中身を追加する.
(4) (3) をコンパイルし，(2) で生成されるライブラリとリンクする.
(5) オブジェクトを ORB に登録する.
(6) このようにして作成されたいくつかのオブジェクトが，お互いにやり取りすることにより，一つのシステムとして動作する.

このように，2 種類の言語を使用し，両者を連携させる必要があり，開発効率の面で問題がある．また，オブジェクト間のやり取りの状況の確認が容易ではないという問題もある.

4.2 分散アプリケーション記述言語を用いた開発手順

アプリケーション開発者は，C++ などを用いた開発と同様，分散アプリケーション記述言語を用いて，インタフェース定義 (クラス定義) とインプリメンテーション定義 (処理内容) の両方を，単一の言語によって行なう．記述されたアプリケーションソースは，図 2に示すような処理を経て，ORB 対応の分散アプリケーションに変換される.

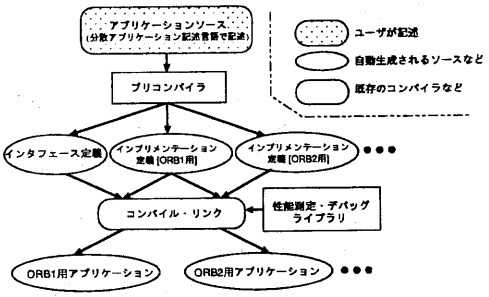

図 2: 分散アプリケーション記述言語・処理系

分散アプリケーション記述言語で記述されたアプリケーションソースは，まず「プリコンパイラ」で処理される．プリコンパイラは，クラス定義の部分からインタフェース定義を生成する．さらに，メソッドの記述の部分に対し，必要に応じて変換を施し，インプリメンテーション定義を生成する．この時，指定によって，アプリケーションの動作状況やリソースの利用状況に関する情報を出力するプローブルーチンが，インプリメンテーション定義に埋め込まれる.

次に，インタフェース定義とインプリメンテーション定義をコンパイル・リンクすることにより，各 ORB 用のアプリケーションが作成される.

このアプリケーションを実際に動作させることにより，テストやチューニングに必要な情報が出力され，これにより，動作状況の確認などの支援が行なえる.

4.3 分散アプリケーション記述言語の文法

4.3.1 言語仕様

分散アプリケーション記述言語は，C++ の言語仕様を基本として，これに以下のような仕様の追加と制約を加えている.

- **ORB オブジェクトの生成・削除などを行なう命令を持つ**
  ORB オブジェクトの生成や削除は，実際には多少複雑な手順を踏む必要があるが，記述言語では，ORB オブジェクトの生成は new，削除は delete と書くだけでよい.
- **予約語の追加**
  string, sequence など，IDL で追加されたデータ型と，ORB オブジェクトに接続するための "bind" などの予約語が追加されている.
- **引数としてポインタは渡せない**
  IDL を用いたインタフェース定義では，ポインタを引数として受け渡すような定義はできない．分散アプリケーション記述言語も，この仕

様に従っている.

4.3.2 記述例

分散アプリケーション記述言語を用いた記述例として，次のようなクラス foo を考える.

- value というインスタンス変数を持ち，数値を保持する.
- value の値を変更するメソッド set() を持つ.
- value の値を取り出すメソッド get() を持つ.

このクラスを，分散アプリケーション記述言語で記述すると，図 3のようになる.

```
class foo {
  private:
    long value;
  public:
    void set(in long x);
    long get();
};

void foo::set(in long x) {
    value = x;
}

long foo::get() {
    return value;
}
```

図 3: クラス定義の例

このように，引数の定義部分に in, inout, out という引数の方向を，IDL と同様に明示する必要がある以外は，C++ での記述とほとんど違いがない.

一方，この foo オブジェクトを利用する部分は，図 4のように記述する.

```
foo p;
p = new foo();
p.set(x); // x は既に値がセットされている
          //long 型の変数とする
```

図 4: クライアントの記述の例

2 行目の "new" で，foo オブジェクトが生成され，3 行目で set() メソッドが実行されている.

ここで，ORB を利用する場合に特有な手順の隠蔽の例として，ORB オブジェクトの生成について説明する. これは，COSS(Common Object Services Specification) のライフサイクルの規定によれば，

(1) FactoryFinder オブジェクトにアクセスし，生成したいオブジェクト用の Factory オブジェクトを探す.
(2) Factory オブジェクトに対して，生成要求を出す.
(3) オブジェクトが生成され，オブジェクトリファレンスが返される.

という手順によって行なわれることになっているが，実際には，ORB によって手順や表記方法が異なっている.

しかし，分散アプリケーション記述言語では，図 4の 2 行目にあるように，単に "new" とすればよい. このように，COSS で規定されている手順，あるいは，ORB ごとの手順の違いを，分散アプリケーション記述言語が隠蔽している.

5 分散アプリケーション記述言語処理系

5.1 プリコンパイラの構成

分散アプリケーション記述言語で記述されたソースは，図 2のような処理を経て ORB 対応の分散アプリケーションに変換される. ここでは，プリコンパイラについて説明する.

プリコンパイラの構成を，図 5に示す.

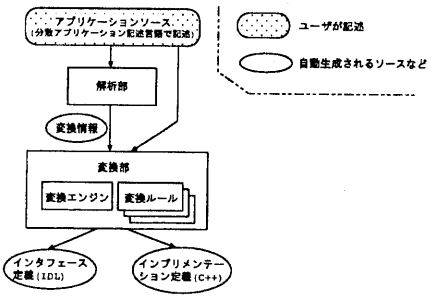

図 5: プリコンパイラの構成

解析部 分散アプリケーション記述言語で記述されたアプリケーションソースの構文解析を行ない，変換に必要な情報を出力する.

変換部 解析部が出力した変換情報を基に，分散アプリケーション記述言語で記述されたソースを，IDL を用いたインタフェース定義と C++ を用いたインプリメンテーション定義に変換する. どのような変換を行なうかは，変換ルールの形で与える. 各 ORB 用の変換ルールを用意することにより，複数の ORB 用のアプリケーションを生成することができる.

5.2 解析部

解析部が出力する変換情報は，

行番号，機能名，変換に必要な情報

からなる.

行番号 分散アプリケーション記述言語で記述されたソースの何行目に関する情報かを示す.

機能名 対象部分がどのような処理を行なっているかを示す. クラス定義，メソッド定義，変数宣

言，ORB オブジェクトの生成，削除，ORB オブジェクトへの接続，メソッド呼び出しなどが挙げられる．

変換に必要な情報 変換部での変換に必要な情報を列挙する．「機能」によって，情報の内容や個数は異なる．

一例として，ORB オブジェクトの生成を考える．変換に必要な情報は，

- 変数名
- スコープ内で宣言されている変数か，引数として受け取った変数か
- 引数として受け取った変数の場合，その方向 (in, inout, out)
- 生成するオブジェクトのクラス名
- new のオペランド

が挙げられる．図 4の記述例 (p = new foo();) の場合，「変数名は "p"，スコープ内で宣言されている変数，クラス名は "foo"，オペランド (コンストラクタへ渡す引数) はない」となる．

ここで，引数として受け取った変数かどうかの情報が必要なのは，引数として受け取った変数は，その方向 (in, inout, out) によって異なる型変換を受けるので，その影響を受ける場合があるからである．

5.3 変換部

変換部は，変換エンジンと，ORB ごとにそれぞれ用意される変換ルールからなる．変換ルールは，分散オブジェクトの生成などをどのような手順で記述するかを指定したスケルトンと，これにどのように変換情報を当てはめて，ORB 用のソースを生成するかの指示を与える部分からなる．変換エンジンは，変換ルールに従ってソースの変換を行なう部分である．

5.3.1 インタフェース定義への変換

インタフェース定義としては，クラス定義や構造体の定義などを IDL を用いた記述に変換して出力する．クラス定義部分に対する変換情報として，次のような情報が得られる．

- クラス名
- クラス定義が何行目から何行目までか
- private の定義が何行目から何行目までか
- public の定義が何行目から何行目までか

IDL で記述するのは，外部インタフェース，すなわち public なもののみであるので，このうち public で定義されている部分のみを抜き出す．同時に，"class" を "interface" に変換する．したがって，図 3の記述例から生成される IDL 定義は次のようになる．

```
interface foo {
  void set(in long x);
  long get();
};
```

5.3.2 インプリメンテーション定義への変換

インプリメンテーション定義としては，ORB オブジェクトの生成，ORB オブジェクトへの接続，メソッド呼び出し，構造体変数へのアクセスなどに関して変換を行なう．

ここでは，ORB オブジェクトの生成を例に，変換を説明する．ORB オブジェクトの生成は，4.3節で述べたように，分散アプリケーション記述言語では，「p = new foo();」のように記述する．また，これに対する変換情報は，5.2節で述べたように，「変数名は "p"，スコープ内で宣言されている変数，クラス名は "foo"，オペランド (コンストラクタへ渡す引数) はない」となる．

COSS に準拠した変換ルールは，分散アプリケーション記述言語での生成部分の記述を，次のような手順に変換するというルールである．

```
Factory クラス名* factory_p;
factory_p = FactoryFinder ->
      find_factories("Factory クラス名");
変数名 = factory_p -> create_object();
```

ここで，下線を引いた部分が可変部分 (変換情報の当てはめ方の指示) である．また，Factory クラス名は，生成するオブジェクトのクラス名の後ろに "Factory" をつけた名前と定められているものとする．変換情報により可変部分をすべて埋めることができ，次のように変換される．

```
fooFactory* factory_p;
factory_p = find_factories("fooFactory");
p = factory_p -> create_object();
```

5.3.3 プローブルーチンの自動挿入

3節で述べたように，アプリケーションの動作状況やリソースの利用状況に関する情報を出力するプローブルーチンを自動的に埋め込むことも行なう．

一例として，オブジェクト間のメッセージ交換の状況をログに書き出すルーチンの挿入を考える．オブジェクト間のメッセージ交換は，メソッド呼び出しによって発生する．また，アプリケーションソース中のどこでメソッド呼び出しが行なわれるかは，変換情報として出力される．したがって，メソッド呼び出しに対する変換処理の一つとして，メッセージ交換に関する情報をログに書き出すルーチンの挿入を行なう．

6 試作・評価

6.1 試作

このような分散アプリケーション記述言語とその処理系を試作し，評価を行なった．試作版は，Sun-

Soft 社の NEO1.0 と IONA Technologies 社の Orbix1.3 の二つの ORB 製品に対応している．すなわち，5.3節で述べた変換ルールとして，NEO 用の変換ルールと Orbix 用の変換ルールを用意した．

ORB オブジェクトの生成を例に，変換の説明を行なう．NEO の場合の変換ルールは，分散アプリケーション記述言語での生成部分の記述を，次のような手順に変換するというルールである．

```
ODF_ObjRef<Factory クラス名> factory_p;
ODF_Find(factory_p, Factory クラス名);
変数名 = factory_p -> create();
```

NEO の場合，Factory クラス名は，生成するオブジェクトのクラス名の後ろに “Factory” をつけた名前と定められているので，変換情報により可変部分をすべて埋めることができ，次のようになる．

```
ODF_ObjRef<fooFactory> factory_p;
ODF_Find(factory_p, fooFactory);
p = factory_p -> create();
```

一方，Orbix では，Factory オブジェクトをシステムが自動的に用意するようにはなっておらず，アプリケーション開発者が明示的に記述する必要がある．ここでは，COSS の規定や NEO に合わせ，専用の Factory オブジェクトを自動生成することにした．例えば，foo クラスの定義があった場合，同時に，fooFactory クラスの定義を自動的に追加するようにした．これは，クラス定義部分に対する Orbix 用の変換ルールとして定義している．また，このようにして作成されたいろいろな種類の Factory オブジェクトを検索するための，FactoryFinder クラスも自動生成するようにした．

したがって，ORB オブジェクトの生成部分に関する Orbix 用の変換ルールは，

```
Factory クラス名* factory_p;
factory_p = FactoryFinder -> find("クラス名");
変数名 = factory_p -> create();
変数名 -> _duplicate();
```

となり，変換情報を適用することにより，次のようなソースが生成される．

```
fooFactory* factory_p;
factory_p = FactoryFinder -> find("foo");
p = factory_p -> create();
p -> _duplicate();
```

6.2 評価

この試作版を用いてアプリケーションの記述を行ない，ORB 特有の手順や ORB ごとの手順の違いなどを意識することなく，プログラムを記述できることを確認した．これにより，ORB プログラムの経験の乏しい開発者でも，効率よくアプリケーションの開発が行なえる．ただし，試作版では，メモリ管理の隠蔽やプローブルーチンの自動挿入には対応しておらず，メッセージ送受信時のメモリ管理は明示的に記述する必要がある．

6.3 効果

ORB をプラットフォームとした分散処理システムの開発に，分散アプリケーション記述言語を用いる効果として，次のような点が挙げられる．

- アプリケーションを一度記述すれば，任意の ORB 上で動作するアプリケーションを生成できる．
- ORB ごとの処理手順や記述の違いを意識する必要なく，アプリケーションが開発できる．
- 上記 2 点は，特に，異なる ORB 上で動作しているオブジェクトがやり取りするようなアプリケーションの開発において有効である．
- ORB プログラミング特有の煩雑な記述の簡易化が図れるとともに，メモリ管理の自動化などにより，作成されるアプリケーションの安全性が高まる．
- アプリケーション開発者が明示的にプローブルーチンを挿入することなく，アプリケーションの動作やその時のリソースの利用状況の確認を行なうためのデータが収集できる．

7 おわりに

本稿では，ORB をプラットフォームとした分散処理システムの開発の下流工程を一貫して支援していく開発支援環境のうち，分散アプリケーション記述言語とその処理系を中心に報告した．

今後は，実システムの開発への適用を通して評価を行ない，この結果をフィードバックして完成度を高めていく予定である．

また，プローブルーチンの自動挿入に関する検討を進め，分散処理システムのテスト支援機能の充実を図っていく．アプリケーション中のどの部分にどのようなプローブルーチンを挿入するかは，アプリケーションの種類により，また，テストのどの段階であるかにより，異なってくる．このような違いを考慮して，適切なプローブルーチンを自動挿入する方式をポイントとして検討を進めていく予定である．

参考文献

[1] 相磯秀夫監訳．共通オブジェクトリクエストブローカ ～構造と仕様～．Object Managemant Group and X/Open, 1994.

[2] OMG. IDL C++ Language Mapping Specification. 1994.

[3] 井川 他．分散オブジェクトの開発支援について．第 52 回全国大会講演論文集, Mar. 1996.