オブジェクト指向分散環境 OZ++ システム第一版の実現

西岡 利博* 濱崎 陽一
三菱総合研究所 電子技術総合研究所

中川 祐* 塚本 享治
富士ゼロックス情報システム 電子技術総合研究所

* 開放型基盤ソフトウェアつくば研究室研究員

OZ++ は、ネットワーク上でのソフトウェアの自動配送を実現するソフトウェアの開発 / 実行環境である。オブジェクト指向を採用しており、共有できるソフトウェアの最小単位はクラスである。クラスには全世界でユニークなクラス ID が付与され、これに基づいてネットワーク上でクラスが検索され、配送される。OZ++ の主な特徴は、ソフトウェアを進化する対象として考えることにより、(a) ソフトウェアの新旧のバージョンが区別でき、かつ混在して利用できることと、(b) 新しいバージョンが積極的に利用できるように、クラスのインタフェースに基づくバージョン管理を採用していることである。本稿では OZ++ システム第一版の構成とおおよその性能について報告し、今後の改良の見通しについて論ずる。

The Implementation of the OZ++ system version 1

Toshihiro Nishioka* Yoichi Hamazaki
Mitsubishi Research Institute Electrotechnical Lab.

Yu Nakagawa* Michiharu Tsukamoto
Fuji Xerox Informations Systems Electrotechnical Lab.

* Researcher, Tsukuba Laboratory, Open Fundamental Software Technology Project

OZ++ is an object-oriented software developing/executing environment realizing the automatic distribution of the software. The minimum unit of shared software is a class. A globally unique class ID is given to each class. This ID is used as a key of searching and distribution of a class over the network. Because software evolves, OZ++ has following features: (a) different versions of a software can be distinguished and co-exist. (b) the versioning system is based on class interface to use the newest version as possible. This paper reports the current implementation and rough estimations of performance of OZ++ system version 1 and argues the possible improvements.

1 はじめに

近年、WWW の普及によって、ネットワーク上での情報の共有が進んでいる。誰もが情報の提供者となれるので、ネットワーク上で提供される情報の量はますます増加し、互いに参考にすることで、情報の質も向上している。ソフトウェアもネットワーク上で共有することによって、供給量も質も向上し、結果として再利用が進むことが期待できる。これまでにも匿名 FTP などの手段によってネットワーク上で優れた品質のソフトウェアが数多く提供されている。

しかし、特にソフトウェアの場合には次のような問題がある。

- ハードウェア / ソフトウェア環境の微妙な相違により、インストールに複雑な問題が発生することがある。
- ソフトウェアは常に進化するので、新旧のバージョンや、互いに関連のないパッチが発生しやすく、相互に利用する上で支障となる。
- 未知のソフトウェアを不用意に利用することで、セキュリティを脅かすおそれがある。

OZ++ は、これらのソフトウェア特有の問題を考慮しつつネットワーク上でのソフトウェアの自動配送を実現する環境である [2]。本稿では、OZ++ システム第一版の構成や性能の概要を述べる。以下、2 節では、OZ++ システム第一版の全体構成について、3 節ではその実現方式について述べる。4 節ではその性能の評価と可能な改善について論ずる。

2 OZ++ システム第一版の構成

OZ++ では、オブジェクト指向並列プログラミング言語である OZ++ 言語で記述されたプログラムをコンパイルし、分散環境上で実行することができる。OZ++ 言語のメソッド起動はネットワークを経由することができ、その引数や返り値としてオブジェクトを渡すことができる。

OZ++ システム第一版は、次のソフトウェア要素から構成されており (図 1 参照)、Sun OS 4.1.3 上で動作する。

2.1 実行環境

OZ++ のオブジェクトは、エグゼキュータと呼ばれる OS プロセス上でマルチスレッドで動作する [5]。エグゼキュータが OS やハードウェアの違いを吸収するので、OZ++ プログラムは、インストールに伴う問題を避けられる。ニュクリアスは、各マシン上にある OS プロセスで、エグゼキュータを管理する。OZ++ システム第一版では、これらのバイナリは gcc 2.5.8 によってコンパイルされ、静的にリンクされている (表 1参照)。

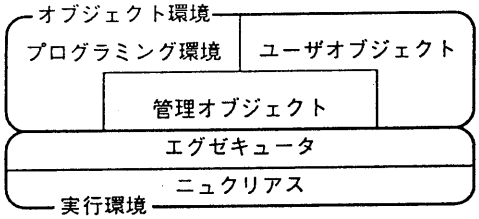

図 1: OZ++ システムの構成

	executor	nucleus
ソースコード	22.2KLOC	3.3KLOC
実行可能ファイル	1304KB	272KB
メモリ負荷	12MB	360KB

表 1: 実行環境

2.2 分散オブジェクト管理システム

一般の OZ++ オブジェクトが動作するためには、分散オブジェクト管理システムと呼ばれる一連のオブジェクトが必要である。これらは管理オブジェクトと呼ばれ、次の種類がある。

- オブジェクトマネージャ (OM)

  各エグゼキュータにひとつずつあって、そのエグゼキュータ上のすべてのオブジェクトの状態を管理する。

- クラスオブジェクト

  クラスに対して、全世界でユニークなクラス ID を発行する。クラス ID をキーとしてクラスを管理し、要求に応じてネットワークを越えて配送する。ひとつのエグゼキュータには高々ひとつだけ存在する。

- ネームディレクトリ

  オブジェクトに名前を与え、名前で参照するための、名称空間である。

	OM	Class Obj.
ソースコード	3.9KLOC	4.2KLOC
クラス数	32	34
イメージ	30KB	2200KB
	Name Dir.	School Dir.
ソースコード	0.54KLOC	1.7KLOC
クラス数	6	8
イメージ	19KB	6.5KB

表 2: 管理オブジェクト

- スクールディレクトリ

  OZ++ では、プログラム中で (クラス ID ではなく) クラス名でクラスを参照できるようにするために、スクールを導入している。スクールとは局所的なクラス名空間で、クラス名からクラス ID への対応表である。スクールディレクトリは、スクールを共有するために、スクールに名前を与え、名前で参照するための名称空間である [4]。

各オブジェクトの規模を表2に示す。ここで、イメージとは、オブジェクトのファイル上での大きさを示す。

2.3 プログラミング環境

ワークベンチと呼ばれるプログラミング環境のプラットフォームが提供されている [1]。ワークベンチからは、スクールを指定して、コンパイラフロントエンドや各種のブラウザが起動できる。他に、デバッグのためのツールが用意されている。

3 OZ++ システム第一版の実現方式

この節では、OZ++ システムの特徴的な部分について、その実現方式を述べる。

3.1 クラス管理の実現方式

クラスの情報を、そのバージョンごとに管理する仕組みについて述べる。

3.1.1 クラスのバージョン管理

OZ++ では、ソフトウェアを進化する対象として考えており、各クラスはクラスオブジェクトによってバージョンを区別して管理されている。クラスの各メソッドとインスタンス変数は、その公開レベルに応じて、四層のクラスパートに分かれている [2] (図2参照)。これらのすべてのバージョンにバージョン ID が与えられる。バージョン ID はクラス ID の一種である。

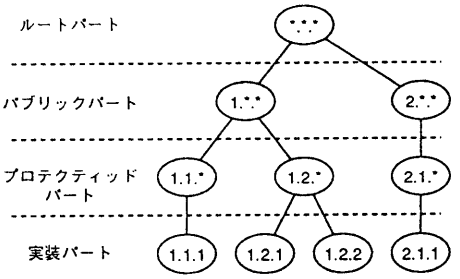

図 2: クラスのバージョンの関係

3.1.2 コンフィギュレーション

インスタンスはパブリックパートのバージョン ID (パブリック ID と呼ぶ) を指定して生成される。一般に、あるクラスのインスタンスは、そのクラスが継承している祖先クラスのインスタンス変数も含むので、インスタンス生成の際には、そのクラスのバージョンだけでなく、祖先クラスのバージョンも決定する必要がある [2]。こうして決定された、そのクラスとそのすべての祖先クラスの実装パートのバージョンの組はコンフィギュアドクラスと呼ばれ、ユニークな ID (コンフィギュアドクラス ID = CCID) が与えられる。CCID もクラス ID の一種である。

パブリック ID と CCID の組をコンフィギュレーションと呼ぶ。各パブリックパートはデフォルトのコンフィギュレーションを持つことができ、インスタンス生成の際は、特に指定がなければこれが使われる。

3.1.3 クラスファイル

パブリックパートやコンフィギュアドクラスなど、クラス ID を持つものを、クラスパートと呼ぶ。クラスパートは、エグゼキュータやコンパイラなどが直接利用するためのファイル群 (クラスファイル) を持っている。

各エグゼキュータはホームディレクトリを持っており、その構成を図3に示す。'classes' はクラスオブジェクトが管理するディレクトリであり、クラスファイルもここで管理される。OZ++ シス

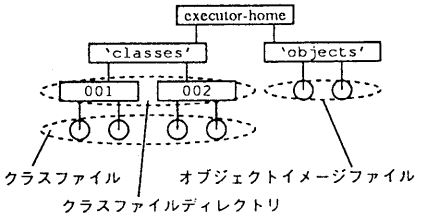

図 3: エグゼキュータのホームディレクトリの構成

テム第一版では、およそ 420 のクラスがリリースされている (この数にはジェネリッククラスから生成されたクラスも含まれている)。これらが、およそ 1,800 のクラスファイルディレクトリに収められた、総計 6,500 のクラスファイルとして提供される。

3.1.4 クラス配送

クラス配送とは、クラスオブジェクトが、要求に応じて、クラスファイルを転送することである [3]。デフォルトでは、クラスファイルをひとつずつ宛先のステーションに転送していくが、クラスディレクトリごとひとつのファイルにアーカイブしてから送信することもできる。

3.2 OZ++ プログラムの実行

OZ++ プログラムを実行するための基本的なメカニズムの重要な部分について述べる。

3.2.1 オブジェクトの構造

オブジェクトのメモリレイアウトは図4のようになっている。インスタンスを構成している各クラスの部分を、オブジェクトパートと呼ぶ (以下、単にパートと呼ぶ)。OZ++ 言語では、クラスは型でもあり、祖先クラスは上位型である。オブジェクトは、その型に応じて、これらのパートのどれかを指すポインタで実現されている。例えば、図のオブジェクトをクラス B 型の値としてアクセスしているときには、パート B へのポインタを参照している。

祖先クラスのバージョンはインスタンスを生成しないと確定しないので、祖先クラスのインスタンス変数領域の構成や大きさも、インスタンスを生成するまで確定しない。例えば、クラス B のメソッドは、クラス B のインスタンスについても、

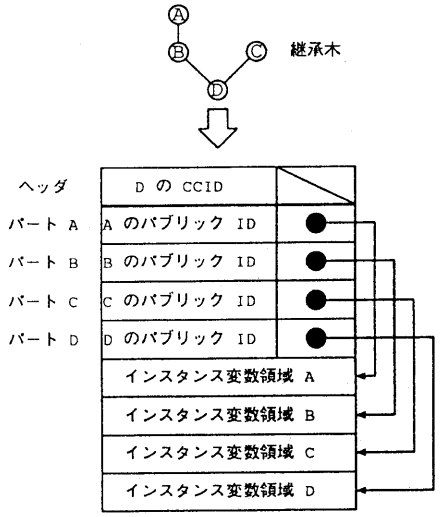

図 4: オブジェクトのメモリレイアウト

クラス D のインスタンスについても同様に動作しなければならないが、そのためにはパート B から見たパート A のインスタンス変数の位置が常に一定でなければならない。このため、インスタンス変数領域は各パートからポインタで参照し、あるパートから見た、そのパートの祖先クラスのパートへのオフセットが常に等しくなるように、各パートをレイアウトする。このため、OZ++ 言語では反復継承がサポートされていない (共通の祖先クラスを継承しても、よく似た二つのクラスを継承したように扱われる)。

3.2.2 セル

OZ++ では、オブジェクトは均等ではない。すべてのオブジェクトはセルと呼ばれる単位に分割されて管理される。セルの中には、セルを代表するオブジェクトが必ずひとつだけあり、これをグローバルオブジェクトと呼ぶ。グローバルオブジェクトでないオブジェクトを**ローカルオブジェクト**と呼ぶ (図5参照)。セルには全世界でユニークな ID を与える。これを**グローバルオブジェクト ID** と呼ぶ。

グローバルオブジェクトはグローバルオブジェクト ID によってグローバル参照される。グローバル参照はネットワークを越えられる。ローカルオブジェクトは、マシンアドレスを使ってローカ

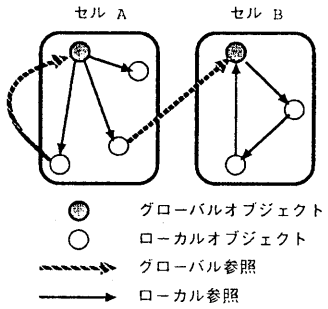

図 5: セルとグローバルオブジェクト

ル参照される。他のセルのオブジェクトをローカル参照することはできない。グローバルオブジェクトを同じセルからローカル参照することはできる。ローカル参照へのメソッド起動を**ローカルメソッド起動**と呼び、グローバル参照へのメソッド起動を**グローバルメソッド起動**と呼ぶ。グローバルメソッド起動では、ネットワークアドレスの解決や、引数 / 返り値のコピーなどが行われる。

3.2.3 オブジェクトイメージファイル

OZ++ のオブジェクトは、セル単位で永続化できる。永続オブジェクトは、オブジェクトイメージファイルに保持される (図 3)。オブジェクトイメージファイルは、メソッド起動を受けたときなど、必要に応じてメモリ上にロードされる。

3.2.4 メソッドの起動

メソッドは、先頭の引数にレシーバオブジェクトへのポインタ (これをレシーバポインタと呼ぶ) が渡されるような関数として実現される。

メソッドサーチには以下の情報が必要である。

- 起動しようとしているメソッドが、どの祖先クラスで最初に宣言されており、そのクラスの何番目のメソッドなのかという情報は、コンパイル時に与えられる。
- メソッドは再定義されるかもしれないので、実際にどのクラスで定義されている何番目のメソッドを起動すべきかは、コンフィギュアドクラスごとに異なる。前項の情報からこれを得るための情報 (これを**実行時クラス情報**と呼ぶ) はネットワークを通じて供給される。エグゼキュータはこれを (CCID) をキーとするハッシュ表で管理する。実行時クラス情報自体は単なる二重配列である。
- クラスの実行可能コードは、インデックスからメソッドの先頭アドレスが分かるような配列 (**関数ポインタ表**と呼ぶ) と、関数本体とが組となって、ネットワークを通じて供給される。エグゼキュータはこれらを実装パートのバージョン ID をキーとするハッシュ表で管理する。

メソッドサーチは以下のアルゴリズムである。

1. オブジェクトヘッダ中の CCID をキーとしてハッシュ表を検索し、実行時クラス情報を得る。なければネットワークから取得してハッシュ表に登録する。
2. 実行時クラス情報を検索して、実装パートのバージョン ID と関数ポインタ表上のインデックスを得る。
3. バージョン ID をキーとしてハッシュ表を検索し、関数ポインタ表を得る。なければネットワークから取得してハッシュ表に登録する。
4. インデックスを用いて関数ポインタ表から求めるメソッドの先頭アドレスを得て、そこにジャンプする。

以上から、一回のメソッドサーチには、二回のハッシュ表検索と、三回の配列検索、および最悪で二回のネットワークアクセスが含まれる。

3.2.5 型の調整

メソッドは、祖先クラスのメソッドを再定義していることがあるので、レシーバポインタが常にそのメソッドが定義されている (オブジェクト) パートを指しているとは限らない。そこでメソッドの先頭には、レシーバポインタを自身が定義されているパートを指すように調整するコード (**型の調整**と呼ぶ) が挿入されている。すなわち、先頭のパートから順にパブリック ID を比較し、一致したパートにポインタを合わせる。

3.2.6 スレッドスケジューリング

エグゼキュータは、ユーザレベルのプリエンプティブなマルチスレッドスケジューリング機能を提供している。動作中のスレッドは、終了するか、待ち状態になるか、または与えられたタイムスライスを使い切ると、実行が中断する。

OS	SunOS 4.1.3
マシン	Sun SS2
OZ++ 動作中の平均負荷	1.5 プロセス未満
ネットワーク	イーサネット
ネットワーク容量	10Mbps
平均コリジョン	0.3% 未満
C/C++	gcc 2.5.8 (-O4)

表 3: 計測条件

スケジューラ (C と比較)	1%
スレッド切替え	7%
スレッドの生成と消滅	1.7 msec/thread

表 4: マルチスレッドのオーバヘッド

4 OZ++ システム第一版の性能

OZ++ システムの基本性能を計測した。計測条件は表3の通りである。スケジューリングタイムスライスは 20msec (デフォルト値) である。OZ++ は GC を行うが、GC は避けて測定した。

ただし、マルチユーザ OS 上で、ネットワークを利用している状態で計測するので、厳密な測定ではない。典型的な利用状況でのおおよその傾向をつかむことを目指す。

4.1 スケジューリング

マルチスレッドであることのオーバヘッドを測定した。結果を表4に示す。

スケジューラのオーバヘッドは、単一スレッドのプログラムの実行時間の、C の同等の処理との差を計測した。スレッド切替のオーバヘッドは、あるメソッドを、2 スレッドで並行実行する時間と、逐次に二回実行する時間の差を計測した。

コンパイル言語であるので、基本的な計算は高速である。

4.2 インスタンス変数アクセス

インスタンス変数書き込みの所要時間を、C++ の同等の処理 (仮想継承した祖先クラスのインスタンス変数書き込み) と比較した。80 % ほど OZ++ の方が遅い。

4.3 ローカルメソッド起動

一定回数のメソッド起動を行うプログラムを C/C++ と OZ++ でそれぞれコンパイルして実行し、消費時間を測定した。実装パートのメソッドの起動ではメソッドサーチをしないので、やや高速である。結果を表5に示す。

C の関数：実装パートのメソッド	1：20
C++ の仮想関数	
：実装パート以外のメソッド	1：30

(いずれも、起動と復帰だけにかかる時間)

表 5: ローカルメソッド起動の性能比較

4.3.1 実装パートのメソッド起動

オーバヘッドの大部分は型の調整である。表の数値は親クラスのないクラスのメソッドの場合であり、継承している祖先クラスがひとつ増えると、消費時間も 30 % 増加する。これについて、次のような高速化を検討している。

- 実装パートのメソッドで、祖先クラスのメソッドを再定義していないものは、必ず同じクラスのメソッドから呼ばれるので、型の調整は不要である。そのように改良して同じプログラムで計測した結果、比率は 1：4 に減った。
- 祖先クラスのメソッドを再定義しているものであっても、実装パートのメソッドであれば、レシーバポインタがサブクラスの (オブジェクト) パートを指していることはない。この性質を利用して、レシーバポインタを、渡された時点で指しているパートから、そのパートの祖先クラスであるパートの数だけオブジェクトヘッダに向かって調べるようにすれば、無関係なパートの分だけメモリアクセスを減らせる。

4.3.2 実装パート以外のメソッド起動

オーバヘッドの大部分はメソッドサーチである。中でも最も大きいのは、ハッシュ表のエントリの参照カウンタを増減する際にエントリをロックする処理である。これを省くためには、リファレンスカウンタに頼らない GC を採用する方法が考えられる。そうするとメソッド起動自体は 2-3 倍高速化されることが分かっているが、GC の頻度はかえって増えるので、評価が難しい。

その他に、ハッシュ表検索を省く方法を実験した。オブジェクトヘッダと各オブジェクトパートに領域を設け、最初のハッシュ表検索時に、求めた実行時クラス情報や関数ポインタ表へのポインタを入れる。これにより、ハッシュ表検索を、最

宛先	一回あたり消費時間
同一エグゼキュータ内	0.5msec
同一 station 上	8.3msec
他の station	5.3msec

表 6: グローバルメソッド起動の性能

C++	0.17 μ sec
local object	1.2 msec
global object (同じ executor)	14 msec
global object (同じ station)	29 msec
global object (他の station)	56 msec

表 7: インスタンス生成の性能

初の一回と GC の直後に限定できる。実験の結果、20 % 程度高速化できることが分かった。

4.4 グローバルメソッド起動

引数と返り値を持たないグローバルメソッド起動 / 復帰一回の消費時間を計測した。結果を表6に示す。他のエグゼキュータへのメソッド起動では、OS のプロセス切替えがない分、他のステーションへの起動の方が速い [6]。

4.5 インスタンス生成

親クラスを持たない、十分小さなインスタンスをひとつ作るための消費時間を計測した。結果を表7に示す。C++ が単にメモリ領域を確保するだけなのに対し、OZ++ では、動的にバージョンを決定するので、クラスオブジェクトにコンフィギュレーションを問い合わせ、実装パートを決定し、図4 の構造を作り上げる。グローバルオブジェクトの生成では、さらに、新たなヒープの確保などの処理が含まれる。

4.6 クラス配送

実行可能コードを含む、典型的なクラスパート (表8) を配送するのに必要な時間を計測した。結果を表 9 に示す。

一括転送の結果から、この条件でのネットワーク上での実質的な配送性能は 320Kbps 程度と見られる。これには、ファイル転送デーモンの消費時間の他に、ファイル転送デーモンとのプロセス切替えや、クラスファイルディレクトリの作成などが含まれている。他のステーションへの転送では、一括転送の方がプロセス切替が少なくて済むが、同一ステーションでは、ファイルコピーは OZ++ のスレッドでできるので、ファイル単位の方がプロセス切替が少なく、高速である。

種別	実装パート
ファイル数	9
総バイト数	98342
一括転送時の総バイト数	112640

表 8: クラス配送実験に使用したクラスパート

配送先	ファイル単位	一括転送
同一 station	2.8 sec	3.8 sec
他の station	3.3 sec	2.8 sec

表 9: クラス配送の性能

4.7 その他の性能上の指標

エグゼキュータの起動後、必要なクラスをロードし、アプリケーションを起動できる状態になるまでの所要時間を計測した。結果を表10に示す。ここで、起動時にクラスの配送を受ける場合に転送されるデータ量を、表11に示す。

オンデマンドに転送するので、ネットワーク負荷はそれほど大きくない。そこで、完全にオンデマンドにせず、要求される前に予測して配送する高速化を検討している。例えば、コンフィギュアドクラスを配送する場合は、続いてその実装パートが配送される可能性が高く、一緒に配送することが考えられる。

5 今後の予定

今後半年ほどの間に以下の点を充実させる予定である。

- 分散環境で安全に利用するために、マルチユーザシステムを整備し、ユーザ認証に基づくアクセス制御を導入する。認証情報を提供するサー

必要なクラスの配送方法	所要時間
すべてローカルから	60 sec
同一 station からファイル単位で	300 sec
他の station からアーカイブで	330 sec

表 10: ブート所要時間

クラスパート数	102
ファイル数	534
総バイト数	3289KB

表 11: 起動時に転送されるデータ量

バを用意し、これを簡単に利用できるアクセス制御ライブラリを用意する。

- 分散環境では、事故やサーバの移動などの影響を受けやすい。このような場合でもクライアントを適切なサーバに導くサービスとして、トレーディングディレクトリを検討中である [7]。
- 分散処理では、オブジェクトの複製やトランザクション処理のプログラミングの需要が大きい。これらを簡単に利用できるクラスライブラリを充実させる。

また、今年度の計画として以下を検討している。

5.1 セキュリティ

他のサイトで作成されたプログラムがネットワークを通じて配送され、動作することを無制限に許すことは危険を招く。OZ++ システムが広く使われるためには、多くのサイトが採用しているセキュリティ方針に外れない運用ができなければならない。現在のインターネットでは、アプリケーションゲートウェイなどを通じて、サイト内部のセキュリティを保ったままで外部のサービスを利用する方針を採用しているサイトが多い。OZ++ でも、アプリケーションゲートウェイを提供し、外部起源のオブジェクトやプロセスの振舞いを制限することを目指している。また、危険な C プログラムを含んだコードが動かないように、外部起源のクラスについては、クラスオブジェクトに特殊な再コンパイルをさせるオプションを検討している。

5.2 その他

マルチアーキティクチャ対応にする。アーキティクチャ依存部分の抽出は終了しているので、クラスファイルの管理方式などを検討し、実際のポーティングを進めていく。

6 まとめ

本稿では、ネットワーク上でソフトウェアを共有する環境である、OZ++ システムの第一版の実現とおおよその性能について述べた。常に進化し続けるソフトウェアを効果的に共有するために、複数のバージョンを同時に動作させたり、自動的に最新のバージョンを利用できるような柔軟性を持っている。クラスの自動配送機構を実現しており、メソッド起動などの要求に応じて、イーサネットの場合で、ひとつのクラスを数秒でロードし、実行できる。今後、マルチユーザ環境、分散処理クラスライブラリ、セキュリティ対策などの面を充実させていく予定である。

謝辞

本研究を通じて熱心な討論をいただいている当プロジェクトのメンバー諸氏に感謝する。

この研究は情報処理振興事業協会 (IPA) が実施している「開放型基盤ソフトウェア研究開発評価事業」の一環として行われたものである。

参考文献

[1] 音川他, “オブジェクト指向分散環境 OZ++ のワークベンチの実現”, SWoPP '95, プログラミング研究会発表予定, Aug, 1995.

[2] 新部他, “OZ++ コンパイラによるクラスの版管理”, SWoPP '94 プログラミング研究会予稿集, IPSJ, Aug. 1994.

[3] 西岡他, “オブジェクト指向分散環境 OZ++ のクラス配送機構”, 情報処理学会第 50 回全国大会, Mar. 1995.

[4] 西岡他, “オブジェクト指向分散環境 OZ++ の名称管理の設計”, 情報処理学会第 49 回全国大会, Oct, 1994.

[5] 濱崎他, “オブジェクト指向分散環境 OZ++ の実行機構の設計”, 情報処理学会第 48 回全国大会, Mar. 1994.

[6] 濱崎他, “オブジェクト指向分散環境 OZ++ の通信機構の実装”, 情報処理学会第 49 回全国大会, Oct. 1994.

[7] 大西他, “オブジェクト指向分散環境 OZ++ のトレーディングディレクトリの設計”, 情報処理学会第 51 回全国大会発表予定, Sep, 1995.