Object Storage System and Programming Transparency

河野真治 渡滋 所真理雄
Shinji Kono, Shigeru Watari, Mario Tokoro

e-mail:kono@csl.sony.co.jp
Sony Computer Science Laboratory Inc.
3-14-13, Higashigotanda, Shinagawa-ku, Tokyo 141, Japan

1990 年 12 月 23 日

あらまし　Muse Object Storage System: Moss では、Reflective Operating System である Muse のメタ階層を利用し、データベースとしての処理とオブジェクト指向言語としての処理を分離する。これにより見通しのよいプログラミングを可能にするとともに、データベースデザインのダイナミックな変更を可能にしている。さらに、今までのオブジェクト指向言語では、メッセージパッシングにより直接に実装する必要があった制約や Active Value、空間的な位置関係なども、元のオブジェクトの記述の簡潔さを損なうことなく行なうことができる。

Abstract　Muse Object Storage System: Moss is an object filing system for reflective operating system: Muse. The reflective architecture of Muse makes it possible to separate database relating processing and normal object oriented processing. This system makes possible a clear programming style for persistent object and dynamic configuration of database design. In classical object oriented language, direct descriptions are necessary for a constraint, active value or spatial relationship. In Moss, object relationships are abstracted as a meta calculation.

1 Requirements

プログラミング環境や実際のアプリケーションでは、プログラムの変更や計算の結果は、どこかにとっておかねばならない。この操作は通常ファイルシステムや、データベースによって行なわれる。しかし、ファイルシステムやデータベース上の操作は、通常のプログラム言語での操作とは通常異なっていて、例えば、データの中から特定の文字列を探すという操作でも、プログラム言語ではやさしい操作だが、ファイルシステムやデータベースでは複雑なコマンドを発行する必要がある。一つの原因は、プログラム中での豊富なデータ構造が、ファイルシステムやデータベースには用意されていないからである。これは、インピーダンスミスマッチと呼ばれている。オブジェクト指向データベースは、通常のプログラム言語を少し制限した形でデータベース操作を記述として用いる、このミスマッチを解決するデータベースの視点からの一つの試みである。しかし、ユーザが、そのような言語をまったく通常のプログラミング言語として操作できるようにはなっていない。逆に、通常のプログラミング言語を用いてデータベースを構築することは一般的に非常に難しいシステムプログラミングである。

このようなデータベースとプログラムのギャップは、インピーダンスミスマッチのようなプログラムとデータベースの結合の問題としてとらえるだけでは不十分である。むしろ、より一般的に Transparency (透過性) の問題としてとらえる必要がある。例えば、ネットワークを通して操作していることを意識せずにコマンドなどを操作できる時、そのコマンドはネットワークに対して Transparent である。同様に、プログラミングをデータの Persistency(持続性) を意識せずにおこなえる時、Persistency に対して Transparent であるということができるだろう。ここでは、オブジェクト指向システムの中で、「とっておく」

という操作を、「記憶階層の中でのオブジェクトのPara Existence(同時多重存在)」としてとらえ、Persistency に対する Programming Transparency を実現することを考える。

Moss (Muse Object Storage System) は、分散オブジェクト指向 OS である Muse[6] 上のオブジェクトファイルシステムであり、Muse の特長であるメタ階層を用いて実現される。このオブジェクト指向 OS の狙いとしては以下の点があげられる。

Easy to use on distributed system 分散環境でのプログラミング、オブジェクト操作、通信がやさしくできること。

Accessibility and Availability オブジェクトに対するアクセスがいつでも可能であり、できる限り応答性が良いこと。

Transparency and Controllability 様々な性質に対して透過性が良く、しかも、制御が効くこと。

Modularity いろいろな段階の抽象化が、オブジェクトの形で可能なこと。

この Moss では、これらの要求のうち、特に storage system に対するものとして、以下の点を目標とする。

Persistency Transparent オブジェクトは persistent(持続性) と volatile(可壊性) の区別なく操作できる。

Dynamic Persistency オブジェクトの persistency の実行時の変更や、データベースの設計変更が可能である。

Expressiveness オブジェクト間の関係もオブジェクト指向にそった方法で表現できる。

Multiple View 複雑なオブジェクトの構造を、複数のより簡単なオブジェクトの構造に射影する。(図 1).

これらの目標を、オブジェクト指向の枠組の中で実現することが望ましい。

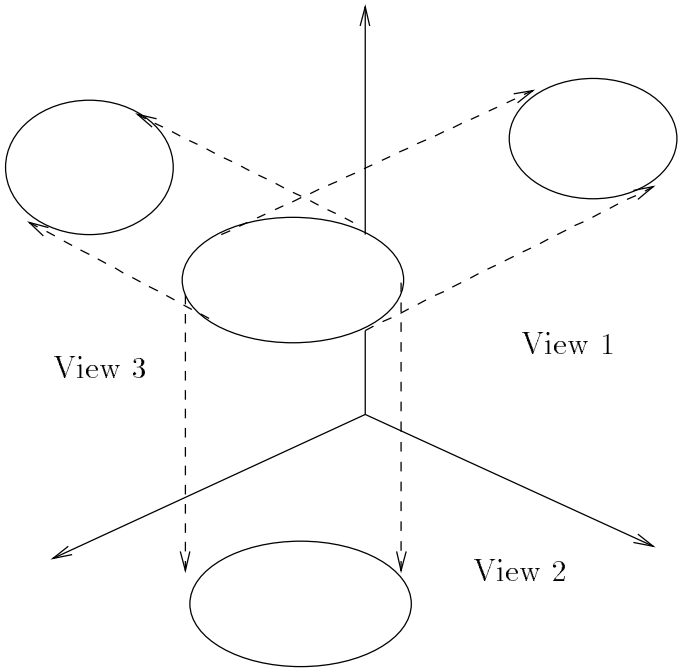

図 1: Multiple Views

2 Problems on Transparency and Controllability

分散環境や、様々な記憶階層、複数ユーザなどの環境で動作するアプリケーションなりプログラムは、通常のデータ構造とアルゴリズムの他に構造を持っている。このような構造は、アプリケーションの動作とは直交している。この構造は、一般的にオブジェクト構造がどのような集合に属しているかを表すものである。このような構造をここではオブジェクトのドメインと呼ぶ。例えば、

分散環境 プログラムのどの部分が、どのサイト (ネットワーク上の特定の計算機) で動作しているか。

記憶階層 現在扱っているデータがどの記憶階層にあるか。ディスク、メモリ、キャッシュなどがある。

複数ユーザ 現在使っているデータやプログラムは、どのユーザのものでどのユーザが使っているものか。

粒度 動作しているプロセスの大きさが大きなものか小さいものか。

持続性 今使っているデータが、どのような記憶階層にいるか。ディスク上にあるものか、メモリ

上にあるかによって、そのデータの持続性は異なる。

これらのドメインの管理は、オペレーティングシステムと密接に結び付いている。プログラミングやアプリケーションの使い方が、これらのドメインの違いに独立である時、そのプログラミングやアプリケーションは、そのドメインに関して Transparent であると呼ぼう。例えば、Sun の NFS (Network File System) は、その上で動作するアプリケーションやユーザが、意識せずにネットワーク上のファイルを使うことができる。これは、ネットワークに対して Transparent である。これに対して、ftp (File Transfer Protocols) などは、ユーザが意識してネットワークを使う。これは Transparent ではない。

本来 Transparency が高いほど、ユーザにとっては使いやすい。しかし、Transparent であるということはドメインに対する制御を放棄していることでもある。例えば OS のシステムコールがすべて Transparent である場合は、ドメインに関することが、まったく制御できなくなってしまう。オブジェクトのドメインは、余計なものというわけではなく、それを制御することにより、より高度な機能や高速性、より大きな問題を取り扱うようにすることができる。

分散環境 どのようなサイトで、どれだけのサイトを使ってプログラムを動作させるか。

記憶階層 メモリをキャッシュに使うかバッファに使うか。キャッシュの invalidate や load をアプリケーションが選択する。

複数ユーザ どのユーザに対してアクセスを許可するか。どのユーザのオブジェクトを共有するか。

粒度 一つのアプリケーションを分割して並行実行するか、いくつかまとめて並行実行するか。あるいは、一つのプログラムを一つのサイトで動かす。

持続性 バージョン管理をするかしないか。今あるオブジェクトの持続性を放棄するかしないか。

これらの制御は、もちろん Transparent な環境でおこなうことはできない。つまり、Transparency がない状況ではじめて制御が可能になる。このように、あるアプリケーションやプログラムは、Transparent な状況とそうでない状況の二つの見方が必要である。この二つの状況は、抽象化レベル (Level of Abstraction) の差として考えることができる。Muse Operating System では、これらの抽象化レベルをメタ階層にマップして考えることができる。これらの抽象化レベルの移動と操作は、メタ階層間をオブジェクトが移動していると考えることもできる。また、OS のカーネルの中に、様々なネットワークドライバや、ディスクインタフェースを取り込むことを kernelize というが、この抽象化レベルの移動は概念的な kernelize/de-kernelize に相当するということもできる。

さらに、この抽象化レベルもまたアプリケーションドメインと考えることができ、このドメインに対する Transparency を vertical transparency と呼ぶ。

3 Examples

まず、いくつか例題を考えよう。ここでは、Unix Like File System, Printer, Calendar, Software CAD Database を取りあげる。これらの例題は、オブジェクト指向による簡潔な記述により解決され、なおかつ、豊富な能力 (信頼性、柔軟性) を持たなければならない。

3.1 Unix Like File System

Muse の世界ではほとんどあらゆるものが平坦なオブジェクトとして現れるが、既存の階層型ファイルシステムに相当するものを考える。これは、特定のデバイスに属する持続性オブジェクトと、そのディレクトリとして実現される。ファイルシステムは高速にアクセスされるキャシュされた部分と持続性を持つ部分の二つの記憶階層を持つ。

3.2 Printer

プリンタは物理的なオブジェクトとして固定された一つの場所に一つだけ存在する。しかし、個々のワークステーション上からもアクセスされる。プリンタへの入力は、複数の箇所で同時に入力される。

3.3 Calendar

カレンダーは、端末などにカレンダーを表示するプログラムである。カレンダーは日付と曜日の対応

を知らせるだけなく、特定の個人の予定(マーク)を持っている。これらのマークは、ネットワーク上に存在するすべてのカレンダーからアクセスすることができる。

3.4 Software CAD Database

Software CAD データベースは、複雑な設計、自在な変更、高速性を要求されるデータベースである。一つ一つのレコードは比較的大きく、複合オブジェクト (composite object) となっている。

4 Disadvantage of Objects

ここで提示したTransparencyの問題は、オブジェクト指向の考え方を導入するだけで解決することができるだろうか。

本来オブジェクト指向計算は並列処理の基本的要素として定義されてきた。しかし、計算機が複雑になるにつれオブジェクトとメッセージパッシングでは簡単には表現できないことが多くあることが分かってきた。例えば、 shared memory, thread, constraints, FIFO, bounded buffer などのより柔軟な通信などは、メッセージパッシングで表現すると、非常に低レベルの表現になってしまい、オブジェクト指向本来の抽象化ができなくなってしまう。また、オブジェクトに対するアクセスがいつでも可能であり、できる限り応答性が良いこと、様々な性質に対して透過性が良く制御が効くこと、などを実現するにも問題がある。

これは、並列オブジェクト指向計算が、以下のような制限を持っているためである。

直列性 :一つのオブジェクトは、一回にひとつしかメッセージを処理できない。⇒ オブジェクトが仕事をしている時にはアクセスできない。

単独性 :一つのオブジェクトは一つの所に一つしかいることができない。⇒ オブジェクトが、手元にいないことが多い。

所有性 :オブジェクト間の関係は、所有されるか所有するかしかない。ここでの所有とは参照を持っていることを指す。⇒ 制約などの複雑な関係は、メッセージパッシングで細かく記述しなければならない。

原子性 :オブジェクトはそれ以上分割できない。⇒ 例えば、メソッド単位の共有ができない。

単純な通信機構 : データ駆動的な変更もメッセージを使って、より原始的に実装しなくてはならない。また、フロー制御のような通信機能の抽象化がない。

実在性 :オブジェクトは現実世界の物体の表現を基にしているため、関係や抽象名詞を表現しにくい。

さらに、分散環境では、次のような問題が加わる。

分散オブジェクトポインタ オブジェクトを指し示す識別子を大域的なものとして定義する必要がある。しかし、分散環境上で大域的な識別子を維持するにはコストがかかる。

メッセージの先送り メッセージを送信したオブジェクトが、自分のサイトにない場合、メッセージをそのオブジェクトがいる位置に先送りする必要がある。

本来オブジェクト指向は、人間が現実世界を認識しているもっとも素朴な方法を、そのまま分散処理に当てはめたものなので、人間には理解しやすいが、実際の分散処理では欠点になってしまう。しかし、プログラムを作り、操作するのは人間であるから、これらのコストや欠点を考慮してもなおかつオブジェクト指向パラダイムを使う利点はある。つまり、オブジェクトは分散環境でのプログラムを抽象化する道具として役に立つ。しかし、実際の実装と制御をオブジェクト指向の枠組そのものを使って行なうことは望ましくない。

ではオブジェクト指向の見通しの良さを維持したまま、Transparency, Availability, Controllability を得るにはどうすれば良いのだろうか。Moss では、Muse オペレーティングシステムの持つメタ階層を利用してオブジェクトのもつ欠点を克服しようとするものである。

5 Muse Object Storage System

Moss がその上に構築される OS である Muse は、メタ階層を持つオブジェクト指向オペレーティングシステムである。Muse のメタ階層は 3 Level であ

り、オブジェクトレベル、メタレベル、メタメタレベルとなっている。Moss もこの階層に沿って構築される。(図 2). 現在のプロトタイプの構成では、

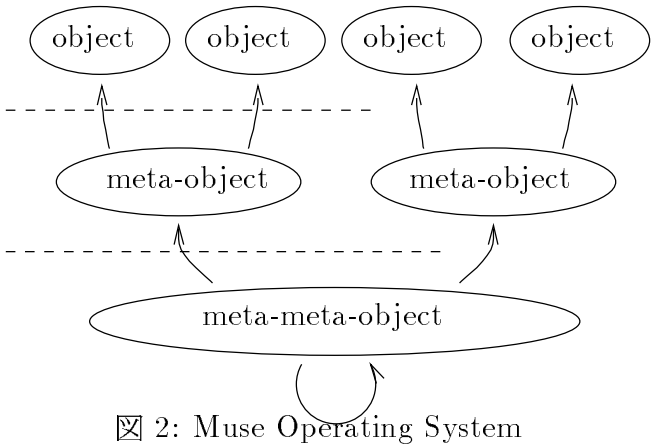

図 2: Muse Operating System

メタメタ階層が一番下のレベルでありハードウェア依存の部分はここで吸収される。メタ階層が通常のオペレーティングシステムのプロセスデータなどに相当する。この階層は、また、通常のオブジェクトの持つメッセージ送信の仕組みを決めるメーラや、メッセージキューなどがあり、オブジェクトの計算機構を実現するメタオブジェクトになっている。この部分により Muse のリフレクション機能 [4] を実現する。オブジェクトは、リフレクティブな操作に対してもメタインタプリタにより実行されるのではなく、ネィティブコードにより実行される。

Moss では、Muse のメタ階層を利用し、データベースとしての処理とオブジェクト指向言語としての処理を分離する。これにより見通しのよいプログラミングを可能にするとともに、データベースデザインのダイナミックな変更を可能にしている。さらに、今までのオブジェクト指向言語では、メッセージパッシングにより直接に実装していた制約や Active Value、空間的な位置関係なども、元のオブジェクトの記述の簡潔さを損なうことなく行なうことができる。メタの部分を考えないオブジェクトの部分では、プログラミングは通常のオブジェクト指向プログラミングで行なわれる。例えば、Moss では Printer は、単に一つのオブジェクトであり、それに対する要求はメッセージパッシングで行なわれる。また、Software CAD のレコードは普通のオブジェクトで実現され、自由に変更できる。

さらに、メタ階層を利用してユーザのレベル (オブジェクトレベル) では、オブジェクトの性質 (欠点) をそのまま維持し、わかりやすい表現を考える。メタのレベルでは、これらの欠点を補正するための工夫を付加する。逆にオブジェクトレベルはこれらのメタレベルの複雑さを無視したものになる。これは実際の複雑なシステムをより簡単な反映に射影することであり、これをを Object Projection と呼ぶ。(図 3).

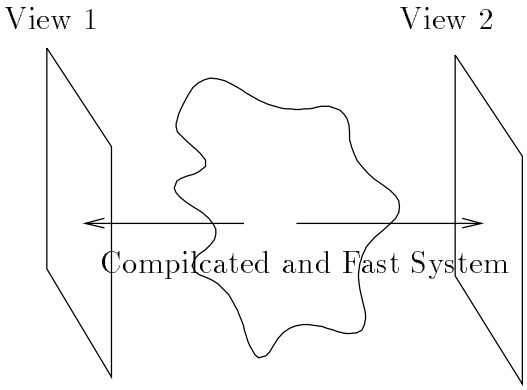

図 3: Object Projection

まず Moss では、オブジェクトの一箇所に一つしかいることができないという性質を緩和し、複数の場所に同時に存在できることにする (Para Existence)。ユーザのレベルからは、単に一つのオブジェクトにしか見えない。これは、コピーや、マイグレーションよりは、キャッシュに近い操作であり (というか、本質的にはキャッシュそのもの)、分散した同一性を表している。本来キャッシュというシステム内でしか使われないものをユーザに方法論として提供するところに意味がある。システムの性能をあげるキャッシュは、これとは別に行なわれる。

これらのユーザレベルでの比較的簡単なオブジェクトは、より細かいレベルのオブジェクトに分解し、メタオブジェクトを使って再構成される (Object Decomposition)。これらは、実際にオブジェクトとして実装される時もあるし、適当なハードウェアや、共有メモリなどを用いて実現された機能を表している時もある。これにより、オブジェクト間の関係などをシステムの方から見た場合でも、見通し良く構成できる。

つまり、以下の三種類の要素が Moss を特徴づける。

Object Projection 複雑なオブジェクト相互関係をメタ階層を利用して抽象化する。

Para Existence 分散キャッシュの技術を用いて実

現する複数の場所に同時にいるオブジェクト。これを用いてダイナミックなオブジェクトの持続性を実現する。

Object Decomposition 一つのオブジェクトを複数の部分に分割して、使用記述または実装をおこなう。

例えば Moss の持続性オブジェクトは、ディスクなどの高信頼性記憶階層と、高速記憶階層との間の Para Existence として実現される。

5.1 Para Existence

複数のオブジェクトが、同じメッセージを受け取り、共通の状態を見せるように協調動作している時、これらのオブジェクトは一つのオブジェクトに見える。例えば、Printer は物理的なプリンタと画面上に見えるアイコンの二つの存在があるが、ユーザは通常これを同一視する。しかし、これらは、複数のオブジェクトからなっており、アイコンや物理的なプリンタはプリンタオブジェクトの一つのユーザ射影に過ぎない。(例えば、プリンタが故障で入れ換えられても、そのネットワークでのプリンタオブジェクトは換わらない) このような射影を提供する中心のオブジェクトが複数存在し、そのメタオブジェクトがその Coherency を保証する。(図 4)

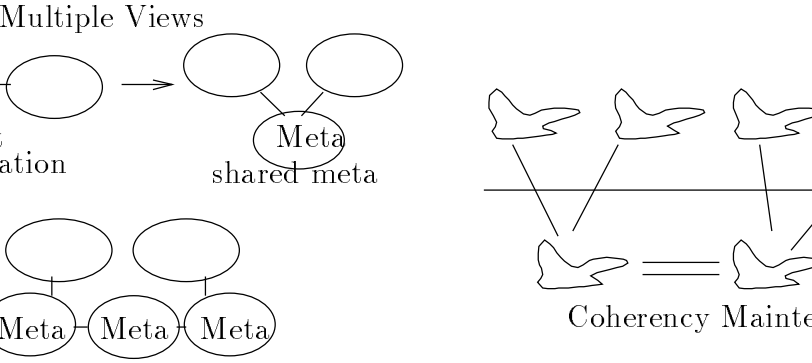

図 4: Para Existence

キャッシュとバッファは、様々な異なる記憶階層で用いられる。例えば、遅いが大容量の主記憶と高速だが高価なキャッシュメモリ、ディスク上のデータとメモリ上のデータ、ディスク装置内のバッファとディスクインタフェースのバッファ。これらはオペレーティングシステム実装上は良くしられたテクニックであるが、ユーザがこれを制御することは通常はできない。そこで、普通はプログラマ自身がこれらのキャッシュやバッファを作ることになる。しかも、これらのキャッシュやバッファはアプリケーションごとに特殊化する必要がある。これらのキャッシュとバッファの機能の中心は、アクセスしやすいコピーを手元に置き、それとオリジナルとの Coherency を維持する所にある。

Pare-Existence は、これらの機能を抽象化したものである。外からは一つのオブジェクトと見ることにより通信に関する Transparency を確保する。バッファやキャッシュとの違いは、主従の関係が対称である点である。

これを実現するためのメタ機構には、以下のようなものをがある。

centralized 一つの固定した l マスタがいて、それ以外のコピーは、そのマスタに対して常に問い合わせる。

token 書き込み権を持つトークンがあり、コピーの間を要求にしたがって巡回する。書き込み権を持たないコピーの変更はトークンを持つコピーに転送される。

negotiation 変更は、複数のコピーで同時におこなわれる。書き込みの衝突が起きた時、メソッドのロールバックまたは手動の調整を行なう。

これらの機能は、Para Existence の関係にあるオブジェクトにより共有されたメタオブジェクトにより実現される。もし、Para Existence の関係にあるオブジェクトが物理的にはなれたドメインにある時には、それらのメタオブジェクトを共有することはできず、メタオブジェクトどうしのメッセージ交換により Coherency の維持がおこなわれる。

Negotiation では Para Existence オブジェクトは厳密にはキャッシュになっていない。これはより大きな単位での共有オブジェクトの制御である。例えば、

- プログラムソースなどを複数の主体により変更する時、
- 移動サイトなどで、Para Existence の状態にあるオブジェクトの通信が切れた時、

など、厳密な直列可能性よりも柔軟な処理を必要とする時、ユーザの指定により使われる。

5.2 Object Relationship

このような、Para Existence を実現するためには、より柔軟なオブジェクト間の通信が必要である。単純な通信でも様々な変化があり、それを効率よく実現していかなくてはならない。例えば、(図 5). も

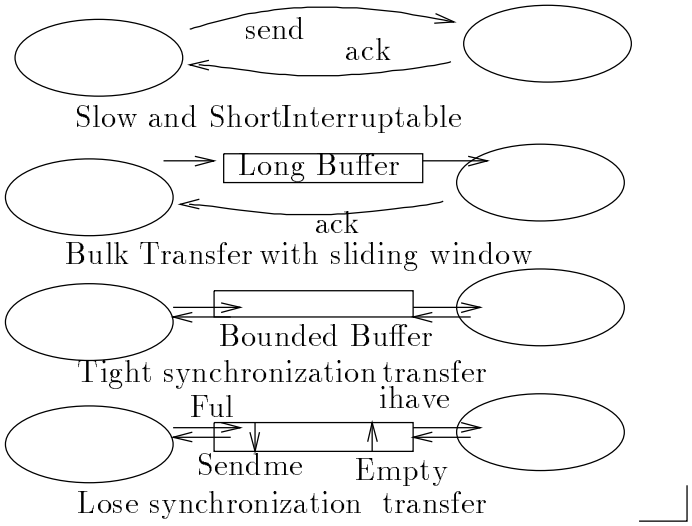

図 5: Various kind of communication

ちろんこれらの通信は、メッセージ通信を基に構築することもできる。しかし、Muse では、メッセージ通信自体に対してメーラを入れ換えるというリフレクティブな操作が可能である。これらの通信は様々な方法で実現されるが、例えばメーラとして二つのオブジェクトの状態を操作できるメタオブジェクトを使うと、本来非同期通信であるはずのメッセージパッシングを FIFO のような同期通信にすることもできる。

この場合、通信する二つのオブジェクトはメーラの部分と二つのオブジェクトの動作を制御する部分とオブジェクトのメソッドを実際に実行する部分に分割されている。

5.3 Object Decomposition

一般的にオブジェクトをより細かく解釈することにより、より高度な抽象化と制御が実現される。そこで、前もってオブジェクトを分解しておくことを考える (図 6)。分解されたオブジェクトは相互に共有されたり、排他制御を緩和したりして、効率的な通信を実現する。これらの小さいオブジェクトは実質的に元の射影されたオブジェクトのメタオブジェクトであるが、そのメタオブジェクトに対応するオブジェクトは存在しない。

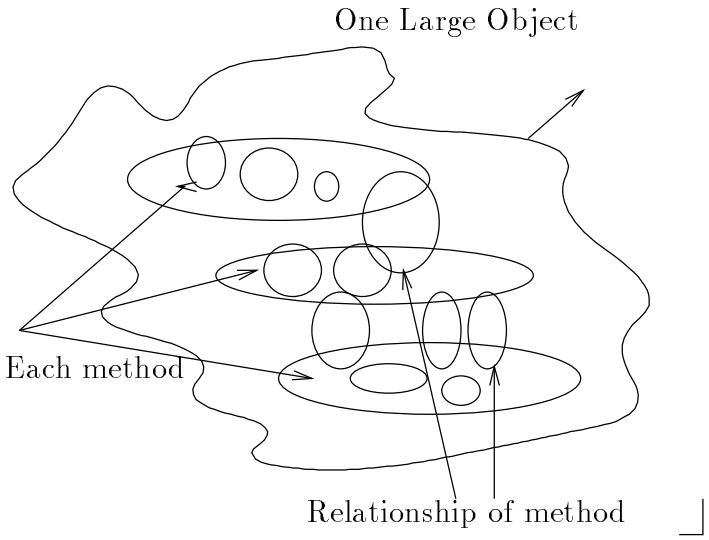

図 6: Object Decomposition

つまり、オブジェクト間の関係は、オブジェクトを持たないメタオブジェクトにより実現される (図 7)。例えば、複合オブジェクトのマイグレーションを考える。いくつかのオブジェクトが複合したものをマイグレーションする場合、その複合したオブジェクトの集合は、どこかのメタオブジェクト中に存在する。このオブジェクトが複合オブジェクトという関係を表している。マイグレーションはこのメタオブジェクトによって、複合オブジェクトに適した形で行なわれる。

メタオブジェクトとオブジェクトの間の因果関係 (causal relationship) も、このようなオブジェクトを持たないメタオブジェクトにより、オブジェクト指向的に実現することができる。このようなオブジェクト間の関係は、Muse のオブジェクト記述の中で、メタオブジェクトの持つ性質として記述されるべきである。この分解されたオブジェクトは概念的なものであり、実際にはオペレーティングシステムに用意されているメーラの機能として実現されて構わない。この場合は、分解されたより細かいオブジェクトは抽象的な仕様である。このより細かいオブジェクトに対するアクセスは、メタオブジェクトの機能に対するアクセスであり、Reflector と呼ばれるメタオブジェクトにより一元的に管理される。Reflector は例えばメーラ、スケジューラなどへのオブジェクトポインタを持っている。

例えば、相手のオブジェクトと FIFO などの通信を行なう時には、必要な関連するメソッドをすべて

調整しなくてはならない。これは、前もって、オブジェクトにそのようなPragmaを記述しておく方法と、元のプログラムから抽出する方法の二通りが考えられる。

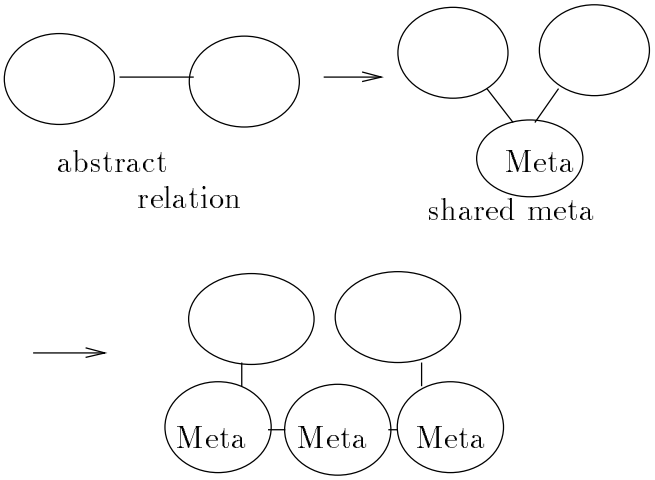

図 7: Object Relationship

5.4 Level of Abstraction

オブジェクトにいろいろな機能をつけ加えればつけ加えるほど、より細かい複雑なプログラム構造になる。また、この機能はオブジェクトとドメインの関係を制御するものであることが多いので、Transparencyを失わせる方向に複雑になってしまう。それでは、このシステムでは、どの様にすればTransparencyを維持しているのだろうか。これは、基本的に追加された機能をメタな機能とすることにより実現する。

実際、よりオブジェクトを細かい単位に分解した時にそのオブジェクトは元のオブジェクトのメタオブジェクトとなっており、追加した機能はすべてメタである。ユーザやプログラマからは、オブジェクトを普通のオブジェクトと見る限りTransparentである。通常、ユーザはそのオブジェクトにメッセージを送ることしかしないので、これでうまくいく。では、メタな機能を制御するためにはどうすれば良いのだろう。これは、まさにリフレクティブな機能であり、メタオブジェクトにメッセージを送れば良い。

つまり、メタ階層の選択が問題にしているドメインのTransparencyあるいは抽象化の階層の選択になっている。この階層の選択は概念的には、どのオブジェクトにメッセージを送るかという選択であり実際の機能としてはサポートされない。しかし、メタオブジェクトにメッセージを送るということを実現するためには、メタオブジェクトを通常のレベルから操作できるようにするために一段メタ階層を下げる必要がある。これは暗黙のde-kernelizeである。ユーザがアクセスできるメタの機能はReflectorに記述されている。(図 8).

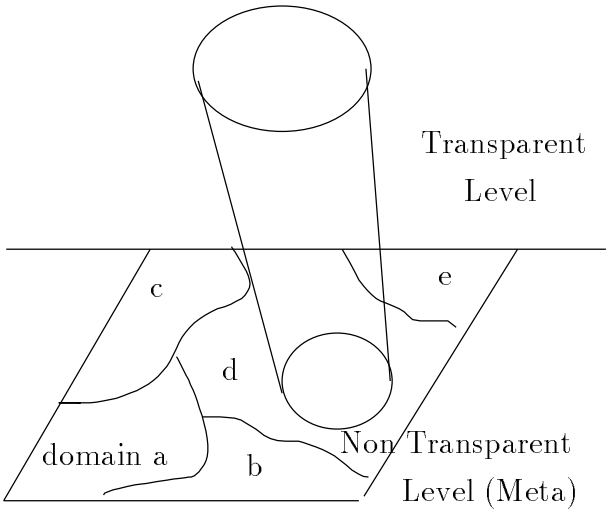

図 8: Level of Abstraction

6 Solution of Examples

さて、それでは最初に考えた例題はどのように解決されるのだろうか。

6.1 Unix Like File System

ファイルシステムは高速にアクセスされるキャシュされた部分と持続性を持つ部分の二つの記憶階層を持つ。オブジェクトはこの二つの階層に同時存在する。オブジェクトの名前と実際のオブジェクトとの対応は、一つのテーブルにより表現される。Unixのように分散したディレクトリは使わない。これは、実際にディレクトリの取る領域が全体の100分の1程度しかないので、集中させた方が効率がよいと考えられるからである。

このディレクトリオブジェクトは、そのデバイスに格納されているすべてのオブジェクトの名前の対応表を持っている。もちろん、自分自身も登録されており、適当にバージョン管理される。このディレクトリオブジェクトは複数共存してもよい。例えばユーザごと、様々な視点ごとに複数作られる。これ

らのディレクトリはすべてPara Existenceの関係にある。また、各ディレクトリは持続性があるために、さらに、二つの記憶領域でPara Existenceの関係にある。(図9)

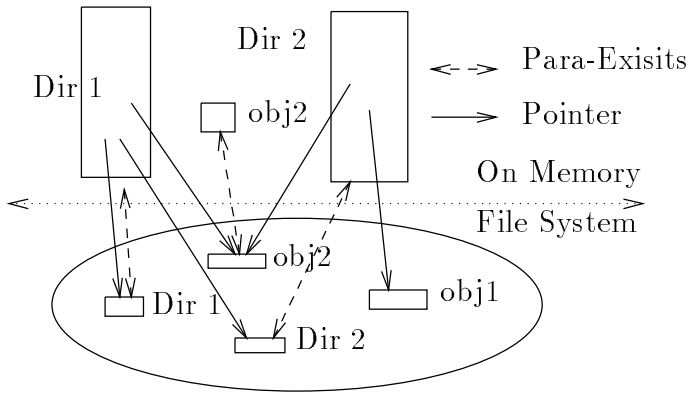

図 9: Directory Object and Para Existence

最初は、ディスク上のファイルシステムしか存在しない。あるホスト上でこのファイルシステムを使うとする場合は、まずディレクトリを自分のホストにPara Existenceにより持ってくる。ホストのメモリの状態によりこのディレクトリはすべてメモリにはいったり、一部しか入らなかったりする。このように一部だけコピーできるようにするためには、オブジェクトは適当に分割されている。これは、ディレクトリがもともとCompositeオブジェクトの性質を持っているために可能である。ホストでは、このディレクトリを使ってファイルシステム上のオブジェクトにアクセスする。このオブジェクトはもちろん普通のオブジェクトとしてコピーして使ってもよいし、ファイルとしてPara Existenceにより持ってきてもよい。この場合に、ディレクトリやファイルを変更する場合は、まず変更Para ExistenceのCoherency維持の機構にしたがって獲得する。それから、実際の変更がおこなわれる。この変更は時間によって適当に元のファイルシステムに反映されたり、Logにとられたりする。

6.2 Printer

プリンタは、ある意味ではMuseの世界に一つしかないオブジェクトである。何故なら印刷という概念は一つしかないからである。しかし、印刷するものの種類や印刷機にはいろいろな種類がある。プリンタはこれらの様々のオブジェクトの集合として機能する。

あるサイトAにプリンタaがつながっていたとしよう。このサイトでプリンタに印刷する時はプリンタaを使う。この時はプリンタaにメッセージを送れば良い。このサイトからサイトBに接続されているプリンタbに印刷したいとする。この場合サイトAからサイトBのプリンタオブジェクトに対して、Para Existenceのnegotiationを起動する。このようにすることにより、サイトAとサイトBのプリンタが一つのプリンタとして融合される。この後は、それぞれのプリンタは一つのプリンタとして動作する。例えば、サイトBのユーザがサイトAでおこなっているプロセスからプリンタを起動する時には、プリンタAのオブジェクトに要求を送ってもプリンタBに送っても構わない。印刷するプリンタは、その時の状況やメタオブジェクトへのメッセージ送信により変更される。(図10).

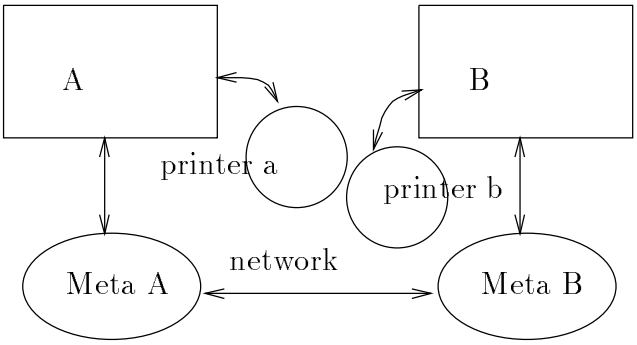

図 10: Printer Implementation

プリンタに送られた要求は、サーバクライアントモデルと異なり、クライアントからサーバにメッセージが送られるのではなく、複数のPara Existenceの状態にあるプリンタオブジェクトのcoherencyの維持として伝搬する。従って、プリンタのキューの状態などはすべてのプリンタが持っていてbroadcastなどにより調整される。

6.3 Calendar

カレンダーは中心的な存在のない同時存在の例である。カレンダーは日付と曜日の対応を知らせるだけなく、特定の個人の予定(マーク)を持っている。これらのマークは、ネットワーク上に存在するすべてのカレンダーからアクセスすることができる。これらのカレンダーはそのマークに関しては一つの同一性(identity)を持っている。

サーバクライアントモデルでは、カレンダークライアントとカレンダーサーバを作ってサーバにマークを問い合わせることになるが、ここでは、カレンダーオブジェクトを一つ作り、そのオブジェクトを複数のサイトでPara Existenceリンクを張ることにより実現する。

カレンダーに全体に対するマークをつける場合は、やはり変更権をCoherencyの機構の手順にそって獲得してから変更する。個人のマークの場合は、ユーザのホームサイトに固定したcentralizeな機構を導入する。このようにすると、サーバクライアントモデルによる実現とほぼ同様になる。

6.4 Software CAD Database

Software CAD Databaseは、通常のオブジェクトの集まりとして構成される。この場合可能な限りメモリ上にオブジェクトは置かれることになる。持続性はディスク上のオブジェクトとのPara Existenceリンクを設定することにより実現される。高速アクセスのためのインデックスは、データベースを構成するオブジェクトをより細かい単位に分解して、その一部を集合として集めることにより構成される。

6.5 Multiple Views

メタ階層は一つの抽象階層を提供していて、これは、ドメイン間の区別をしない通常のオブジェクトの部分とドメインの区別を問題にするメタオブジェクトの階層を提供している。

また、Para Existenceは異なるドメイン上の二つのオブジェクトを等しくする操作であり、この二つの異なるドメインが、異なるオブジェクトの見方(View)を提供する。例えば、Software CAD Databaseでは、オブジェクトはメモリ上の普通のオブジェクトとディスク上のファイルの二つのviewを持っている。

7 Concluding Remarks

通常のファイルシステムでは、ファイルに対するオープン、クローズ、書き込み、読み出しなどの基本操作を定義し、そのファイルに対する名前付けを定義することによりファイルシステムの基本が設定される。これまでのファイルシステムでは、ディスク上のデータ構造であるレコードやシリンダを基本単位としてこれらが定義されてきた。従って、その上のオブジェクトであるファイルはその構造に左右され、ブロックサイズやシンボリックリンクなどのファイルの持つオブジェクトとしての性質には関係ない要素が多数導入されTransparencyを下げる原因となっている。Mossでは、これらの操作をオブジェクトとは別のレベル(メタレベル)で定義し、通常の仕様ではその操作が見えないようにしている。

もちろん、Smalltalkなどではオブジェクトの操作は、高度にTransparentに定義されているが、ファイルシステムの場合とは逆にオブジェクトの持続性を定義することが難しい。もちろん、このオブジェクトが持続性であるという定義なり機能などをつけることはできるが、それが、OSの機能としてどういう操作であるかは明確ではなかった。例えば、実際にどういう形で持続性を実現するかはユーザには分からない。Mossでは、ディスク上でもメモリ上でもオブジェクトはオブジェクトとして存在し、それに対するアクセスは明確になっている。メモリ上のオブジェクトとディスク上のオブジェクトが同じ構造を持って見えるということが既に十分な情報をユーザに伝えている。それ以上の操作をする場合には、メタオブジェクトを用いる。

オブジェクト指向データベースでは、データベースアクセスを(型などの点で制限はあるが)一般的なメソッドを使えるようにしている。これによりデータベースプログラミングと通常のプログラムとのリンクを改善し、さらにデータベース設計の点でもより柔軟な使い方を可能にしている。Mossは、完全なデータベースとしての持続性オブジェクトではなく、より一般的なファイルシステムとしての持続性オブジェクトを実現するものである。高速検索などは初期の目標としては目指していない。より完全なデータベースはMoss及びMuseの上にさらに構築される。このために必要な機能(Atomic WriteやBuffering Control)は、ファイルシステムドライバやオペレーティングシステムのメタオブジェクトの持つ機能として用意される。

Generalized Forwarding [5], Proxy Object [3, 1]のような分散オブジェクトシステムは、オブジェクトを複数のサイトからなる空間に分散させるものである。この場合オブジェクトを指すポインタがサイト情報を含むように拡張される。このままでは、3章で述べたようなオブジェクトの持つ欠点が解決

されない。Muse/ Moss では、これをメタオブジェクトとオブジェクトの分割により解決する。特に、オブジェクトのキャッシュをシステムのレベルからユーザが制御できるレベルに移している所が Moss の特徴である。

分散したキャッシュを制御する問題は、Virtual Shared Memory [2] など、様々なキャッシュの論文に現れている。キャッシュは分散環境でのパフォーマンスを決定する重要な部分であると同時に、Transparency を維持するための機構でもある。将来的にはキャッシュはアプリケーションごとに最適に調整されるべきであり、より構造化されたものであることが要求される。Moss の Para Existence は、オブジェクトの勝手な一部を勝手にキャッシュする機構と異なり (そのような部分は、ハードウェア依存の部分でおこなわれる)、オブジェクトの全体または分割された一部をより上位の部分でキャッシュする。このキャッシュはメタオブジェクトによりオブジェクトごとに管理される。キャッシュの Coherency の維持には、Virtual Shared Memory の手法などを用いる。

サーバクライアントモデルは、分散環境でのアプリケーションを作る際に良く用いられるモデルであり、Centralize なサーバとそれにアクセスするクライアントからなる。これは、サーバが唯一の情報を持っており、クライアントがそれに対する変更や読み出しをする場合には適している (例えばウィンドウシステム)。しかし、同一ホスト内でアプリケーションを動かす場合は、通信が余計にはいる分だけ無駄になってしまう。また、分散会話システムのような複数のサイトからデータが同時に提供されるようなシステムでは、通信のリンクをリンクの二乗だけ張ることになる。これは、すべてのサイトからすべてのサイトのサーバにリンクを張るためである。Moss の方法では、このような場合でも Transparent なオブジェクトでのアプリケーションとして記述すれば良い。サイト内での通信は、共有されたメタオブジェクト内で最適化される。

参考文献

[1] D. Decouchant. Design of a distributed object manager for the smalltalk-80 system. In *OOPSLA 86*, pp. 444–452. ACM, 1986.

[2] Kai Li. *Shared virtual memory on loosely coupled multiprocessors*. PhD thesis, Yale University, 1986.

[3] Steven E. Lucco. Parallel Programming in a Virtual Object Space. In *OOPSLA 87*, pp. 26–34. ACM, 1987.

[4] Pattie Maes. COMPUTATIONAL REFLECTION. Technical Report TR-87-2, VUB AI-LAB, 1987.

[5] Paul L. McCullough. Transparent forwarding: First steps. In *OOPSLA 87*, pp. 331–341. ACM, 1987.

[6] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A Reflective Architecture for an Object-Oriented Distributed Operating System. In *Proceedings of European Conference on Object-Oriented Programming in 1989*, 1989.