

ODMG 標準へのマルチタイプオブジェクト機能の導入

佐 藤 秀 樹

概 要

実世界の实体は、一生の間に異なるアスペクトの獲得・喪失を繰り返す。こうしたアスペクトの動的変化を扱うため、複数の型を持つオブジェクト、すなわちマルチタイプオブジェクトがオブジェクト指向モデリングの枠組みに不可欠である。本研究では、マルチタイプオブジェクト機能をオブジェクト指向モデリングの枠組みに導入し、ODMG 標準に準拠する INADA と Object Query Language (OQL) への組込みを行なう。INADA は拡張 C++ 永続プログラミング言語であり、ODMG C++ Binding の具体化である。また、OQL はオブジェクトに対する連想アクセスを可能とする高水準問合せ言語である。本論文では、マルチタイプオブジェクト機能の実装の詳細とその有効性を示す。OQL へのマルチタイプオブジェクト機能の組込みは、その言語仕様を拡張することなく実現される。著者が知る限り、本研究の他に、OQL のような高水準問合せ言語にマルチタイプオブジェクト機能に類する機能の組込みを行なった研究はない。

キーワード オブジェクト指向データベース (Object-Oriented Data Base), オブジェクト指向データモデル (Object-Oriented Data Model), マルチタイプオブジェクト (Multiple Type Object), 永続オブジェクト (Persistent Object), ODMG (Object Database Management Group)

1 まえがき

オブジェクト指向モデリングの枠組みは実世界の实体をモデル化するための高い能力を持っている。しかし、实体は一生の間に異なるアスペクトの獲得・喪失を繰り返し、实体の性質は時間と共に動的に変化するため、オブジェクト指向モデリングによる实体のモデル化は必ずしも容易でない。例えば、研究所に働く人が、仕事を辞めることなく、大学に入学することとする。この人は、大学では学籍番号を持つ学生であり、会社では従業員として従業員番号を持つ。この人がこの世界に登場した時点で、将来に渡ってこの人が持つアスペクトの全

てを正確に予測・設計できないことに注意を要する。オブジェクト指向データベースシステムは長期間に渡ってデータベースに存在するオブジェクトを管理し、複数の応用間でオブジェクトの共有を可能とする。このため、上記のモデリング問題は、オブジェクト指向データベースではより重大となる。この問題の克服に向けて、オブジェクト指向データモデルを拡張する研究も行なわれてきた [1], [2], [3], [4], [5], [6], [7], [8], [9]。

オブジェクト指向データベースは、オブジェクト指向プログラミング方法論とそれらを具体化したオブジェクト指向プログラミング言語 [10], [11] から発展してきた。その黎明期には、オブジェクト指向データベースの基礎としてのオブジェクト指向データモデルの形式化に関する研究は遅れていた。しかし、近年、関係データモデル [12], [13] にならったオブジェクト指向データモデルの代数論的研究が行われるようになった。また、Object Database Management Group (ODMG) はオブジェクト指向データベースの標準化作業を継続している [14], [15], [16], [17]。ODMG-93 標準 [14], [15] では、以下のようにマルチタイプオブジェクト機能が引き続く改定項目として位置付けられていた。

オブジェクトは動的に型を獲得・喪失する。オブジェクトがそのインスタンス型の下位型のインスタンスに動的に変わるのが、この最も単純な場合である。型プログラマはこの変更を実装するシステム演算子を置き換え、システムは事前／事後演算子を提供する。これは、複雑な完全性制約のために使われる。

ODMG-93 標準の改定版である ODMG 2.0 [16], ODMG 3.0 [17] では、マルチタイプオブジェクト機能の記述は消失している。マルチタイプオブジェクト機能が多くの応用にとって有効であることより、これは残念な事である。著者は、世界中に広く普及することが期待されている ODMG 標準において、マルチタイプオブジェクト機能を必須機能とすることを提案したい。

本論文は、マルチタイプオブジェクト機能の ODMG 標準への組込みを提案する。先ず、ODMG C++Binding の具体化である拡張 C++永続プログラミング言語 INADA へのマルチタイプオブジェクト機能の組込みを述べる。次に、データベース・オブジェクトの宣言的・非手続き的問合せ言語である ODMG Object Query Language (OQL) へのマルチタイプオブジェクト機能の組込みについて述べる。このために、OQL の言語仕様の拡張は必要ない。さらに、オブジェクト・ビュー、集合オブジェクトの多重実装法、SGML 文書データベースなどの応用を使って、マルチタイプオブジェクト機能の有効性を示す。著者が知る限り、本研究の他に、OQL のような高水準問合せ言語にマルチタイプオブジェクト機能に類する機能の組込みを行なった研究はない。

2 マルチタイプオブジェクト

2.1 背景

一般に、実世界の実体をモデル化するオブジェクトは、応用に応じた形式あるいは情報を持つ。人間をモデル化する場合を想定する。ある人が大学に入学すると、この人は大学生となる。大学では、この人は名前、年齢、学籍番号、部屋番号、部屋の電話番号といった属性を持つ。同時に、この人は学外では非常勤従業員として働くことができ、名前、年齢、所属部門名、オフィスの電話番号、給与額といった属性を持つ。この人は基本的に同一人物であり、両環境において名前と年齢は同じであるが、電話番号は異なる。

上記の実体のモデリングに対する第1の解は、データベース設計者がオブジェクトを操作する全ての応用が使用する型に対するオブジェクト構造をオブジェクトの作成前に設計することである。言い換えれば、オブジェクトは全ての情報を含み、ユーザはオブジェクトのアクセス時に自分の応用に適した特定の形式あるいは型にオブジェクトの情報を射影する。設定されない属性値として、ユーザは未定義値 (null) あるいはフラグを使う。例えば、全ての人間は University-Company 型のオブジェクトとしてモデル化されることとする。学生ではあるが、従業員ではない人をモデル化する場合、従業員としての所属部門名、オフィスの電話番号、給与額の属性値は null 値に設定される。

データベース設計者が事前に全ての応用を知らなければならない点を除けば、この解法は機能する。しかし、モデル化対象となる実世界の実体の特性からは、別の問題が生じる。一般に、誰もある人の未来を正確に予測不可能である。少数のオブジェクトでは使用されるが、多くのオブジェクトでは使われない属性の場合、null 値あるいはフラグを用いる解は多大な記憶空間を浪費する。結論的には、オブジェクト構造の設計は、動的であるべきである。従って、漸次的なデータベース設計やソフトウェア・モデリングを実現するため、オブジェクトに特性を追加する機構が必要となる。

複数量を持つオブジェクトのモデル化に使われる多重継承は、基本的にこれとは異なる。多重継承では、導出クラスは上位クラスの全ての属性を持つことになる。さらに、オブジェクトがいったん作成されると、その生存期間においてその型を変更できない。結果として、多重継承は、ある人が学生になり、仕事を辞める場合をモデル化できない。

2.2 設計方針

前記の考察から、INADA にマルチタイプオブジェクト機能を導入する上で、以下の原則を設定する。

- 永続オブジェクトは、その生存期間において任意の型を動的に獲得・喪失できる。また、永続オブジェクトは、任意の時点において0個以上の型を持つ。
- 永続オブジェクトは、複数の型に対して単一のオブジェクト識別子 OID (Object Identifier)

を持つ。この OID は、識別子ならびに参照として使われる。

- 属性名/メソッド名の有効範囲はオブジェクトではなく、型である。従って、オブジェクトは同一の属性名/メソッド名を持つ複数の型に属する。
- マルチタイプオブジェクトの型システムは、C++の型システムに準拠する。言い換えれば、マルチタイプオブジェクト概念は、C++の型システムを変更することなく導入される。
- C++コードは、INADA において利用可能である。INADA は既存の C++コードを変更することなく使用する。

2. 3 ODMG C++Binding に対する拡張

INADA では、永続オブジェクトは `d_Ref<T>` 型の永続ポインタ (OID) を使って操作される。ここで、`T` は永続化可能クラスである。これは、ODMG 標準に完全に準拠する。例えば、`Person` クラス (型) の永続オブジェクトを参照する永続ポインタ `p` は、次のように宣言される。

```
d_Ref<Person> p;
```

永続オブジェクトの作成には、オブジェクトの割当て場所を引数とする `new` 演算子が使われる。`Person` 型の永続オブジェクトは、次のように作成される。

```
d_Ref<Person> p=new(database, "Person")Person("Sato", 48, ...);
```

ここで、`database` は `d_Database` オブジェクトへの永続ポインタであり、オブジェクトが作成されるデータベースを参照する。C++の型システムが、永続オブジェクトの作成に使われる。次の文は `Part` 永続オブジェクトを作成し、その OID を `d_Ref<Person>` 型の `p` に代入しようとする。この文が誤りであることは、コンパイル時に検出される。

```
d_Ref<Person> p=new(database, "Part")Part("wheel", 4, ...);
```

INADA の永続オブジェクトは、C++の揮発オブジェクトと同じように扱われる。例えば、

```
p->Name();
```

は、`Person` 型が `Name()` メソッドを持てば、正しい文であり、

```
p->PartID();
```

は、`Person` 型が `PartID()` メソッドを持たないならば、C++の型システムによりコンパイル時に誤りとして検出される。

マルチタイプオブジェクト機能をサポートするため、INADA 構文として `transforms` と `as`¹⁾ [18] を導入する。これらは、永続オブジェクトへの任意の型の追加、任意の型による永続オブジェクトのアクセス、永続オブジェクトからの任意の型の削除に使われる。

永続オブジェクトへの任意の型の追加

構文 `transforms` は、既存の永続オブジェクトに任意の型を追加するために使われる。この構文は、`new` 演算子とともに使われる。例えば、次は前記の永続オブジェクトに `Employee` 型を追加する。

`d_Ref<Employee> e=new(database, "Employee") Employee("sale", 1853, ...)` transforms `p`; INADA では、`e` は `p` と同一の値を持つ、すなわち `e=p` となる。これは、両方が同じ永続オブジェクトを指しているからである。これが、INADA と `aspect2)` 機構 [4] を含む他の類似研究との違いの 1 つである。

任意の型による永続オブジェクトのアクセス

永続ポインタによる永続オブジェクトのアクセスでは、オブジェクトをアクセスする型はポインタ型によって決まる。次の例を考える。

```
p->TelNumber(); // (1)
```

```
e->TelNumber(); // (2)
```

`Person` 型が自宅の電話番号を返す `TelNumber()` メソッドを持ち、`Employee` 型が職場の電話番号を返す `TelNumber()` メソッドを持ち、この 2 つの番号が異なる場合、(1) の結果は (2) の結果とは異なる。すなわち、(1) は `p` の型が `d_Ref<Person>` であるから `Person` 型によりオブジェクトをアクセスし、(2) は `e` の型が `d_Ref<Employee>` であるから `Employee` 型によりオブジェクトをアクセスする。`Person` 型が `boss()` メソッドではなく `spouse()` メソッドを持ち、`Employee` 型が `spouse()` メソッドではなく `boss()` メソッドを持つならば、コンパイル時に C++ の型システムにより次の 2 つの文が誤りであることが検出される。

```
p->boss();
```

```
e->spouse();
```

特定の型による永続オブジェクトのアクセスのため、INADA の `as` 構文を使う。次の文の各々は、`Employee` 型と `Person` 型によりオブジェクトをアクセスする。

```
p as Employee ->boss();
```

```
e as Person ->spouse();
```

`p` が指す永続オブジェクトが `Employee` 型を持っていない。この場合、最初の文の実行は `nil` オブジェクトの識別子を返す。アクセスされるオブジェクトがその型を持つか否かはコンパイル時にはわからない。この処理は、実行時に行われる。

`as` の別の使い方は、クラス定義中に "`this as TypeName`" 形式で現れる。これは、第 3 節において述べる。

永続オブジェクトからの任意の型の削除

永続オブジェクトからの任意の型の削除は、次の文により行われる。

```
delete Employee of e;
```

この文は、ポインタ `e` が指す永続オブジェクトから `Employee` 型を削除する (オブジェクトがその型を持つならば)。ここでは、動的な型チェックが必要となる。オブジェクトがその型を持つか否かはコンパイル時にはわからない。このため、その型を持たない永続オブジェクトからの削除に対して、実行時誤りが起きる。

一方、次の文は INADA において可能である。

`delete e ;`

この文はポインタ `e` が指すオブジェクトを削除し、同時にそのオブジェクトが持つ全ての型も削除する。

2. 4 OQL へのマルチタイプオブジェクトの組込み

ODMG OQL [14], [15], [16], [17] は、オブジェクトに対する連想アクセスのための高水準問合せ言語である。マルチタイプオブジェクトをアクセスするため、型の指定が OQL において必要となる。このため、永続ポインタ・クラス `d_Ref<T>` のメソッド `as(TypeName)` がパス式で使われる。メソッド `as(TypeName)` は通常のメソッドと同様に使えるので、マルチタイプオブジェクトの組込みのために、OQL 言語仕様の拡張は必要ない。併せて、マルチタイプオブジェクトの OQL への組込みに伴って、OQL コンパイラに付加的な処理が課せられることもない。

表 1 は、マルチタイプオブジェクトの問合せ例を示す。Q1 では、Persons は Person 型のエクステンションである。Q1 の `where` 節では、`as("Employee")` メソッドが選択条件を示すためにパス式中で使われている。同様に、メソッド `as(TypeName)` は `select` 節、`from` 節、`order_by` 節、`group_by` 節、`having` 節でも使用できる。マルチタイプオブジェクトのアクセス型は、`as(TypeName)` メソッドにより指定されるが、これはマルチタイプオブジェクトのアクセス型を明確に示しており、かつ OQL 言語仕様に合っている。

表 1 マルチタイプオブジェクトに対する問合せコマンド

<p>Q1: 30 歳より若い上司を持つ人を求めよ。</p> <pre>select p from Persons p where ((Employee) p.as("Employee")).boss().age < 30</pre>
<p>Q2: 給料が上司より多い人の上司の名前を求めよ。</p> <pre>select ((Employee) p.as("Employee")).boss().name from Persons p where ((Employee) p.as("Employee")).salary < ((Employee) p.as("Employee")).boss().salary</pre>
<p>Q3: 人の従業員アスペクトの集合を <code>v_employee</code> と定義せよ。</p> <pre>define query v_employee() as select ((Employee)p.as("Employee")) from Persons p</pre>
<p>Q4: <code>v_employee</code> を使って、給料が上司より多い人の上司の名前を求めよ。</p> <pre>select ve.boss().name from v_employee ve where ve.salary < ve.boss().salary</pre>

OQL コンパイラにより, Q1 の問合せは次のような INADA コードに翻訳される。

```
d_Ref<Person> p;
d_iterator<d_Ref<Person>> iter=Person::Persons->create_iterator( );
d_Ref<d_Bag<d_Ref<Person>>> result=new(d_Database::transit_memory,
    "d_Bag<d_Ref<Person>>") d_Bag<d_Ref<Person>>;
while (iter.next(p)) {
    if ((Person) p.as ("Employee"))->bos( )->age < 30) {
        result->insert_element(p);
    }
}
```

Q2 の問合せのように, as(TypeName) 句が複数現れると, 記述が複雑であると感じられる。この問題点の対処に, 構文糖衣は必要はない。代りに, Q3 のように define_query 文 [14], [15], [16], [17] が使用できる。そして, Q4 のように, define_query 文が定義するコレクションを使えば, 元の問合せは簡潔に記述できる。

3 実装の詳細

本節では, INADA におけるマルチタイプオブジェクト機能の実装の詳細を示す。INADA 自体はトランスレータ, クラス・ライブラリ, 実行時ルーチンからなる。トランスレータは INADA プログラムを C++コードに翻訳し, C++コードは次に C++コンパイラにより翻訳される。

3. 1 永続ヒープと永続ポインタ

INADA の永続オブジェクトは, データの動的な割当て・削除が行われる永続ヒープ (Persistent Heap; PH) を使って実装される。永続ヒープはプロセスの一部あるいは仮想アドレス空間であり, ローカルサイトあるいはリモートサイトの 2 次記憶上のファイルにマップされる。これは, メモリーマップ型の I/O 機構に基づく分散ページ-オブジェクト・サーバー WAKASHI [19] により実装される。図 1 は, 永続ヒープの構成を示す。ONT は, 名前と永続ヒープに割当てられる永続オブジェクトを結びつける。永続ヒープ上の全てのオブジェクトに, 名前が付けられる必要はない。ORT はハッシュ表である。OS は, 永続オブジェクトの保存のために領域を割当てて。ORT と OS は, 拡張可能である。

マルチタイプオブジェクト機能の実装のため, ORT エントリは永続ヒープの先頭からのオ

プロジェクトのオフセット、型識別子 (TypeID), OID, NextID を保持する (図 2 (a) 参照)。各々は4バイト長である。さらに、TypeID 表 (図 2 (b) 参照) は、INADA コードを C++コードに翻訳するために必要である。この表はデータベースに対して1つ存在し、2 次記憶に保存される。

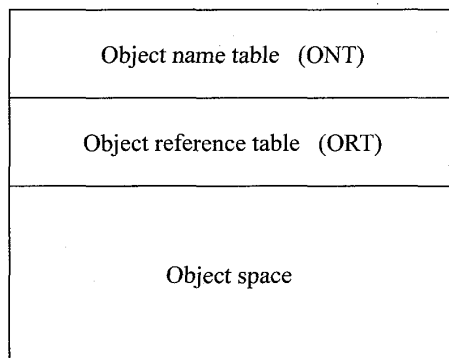


図 1 永続ヒープの物理的セグメント配置

Offset Address		
TypeID	OID	NextID

(a) ORT エントリ

0	CLASSI
1	CLASSJ
2	CLASSK
⋮	⋮
⋮	⋮
⋮	⋮

(b) TypeID 表

図 2 ORT エントリと TypeID 表

永続オブジェクトは、永続ポインタを使って操作される。永続ポインタは2つのフィールドから成る (図 3 参照)。最初の1バイトは、そのポインタが参照する永続オブジェクトが割当てられる永続ヒープの識別子 `ph_id` を表わす。残りの3バイトは、オブジェクトに対する ORT エントリ `ort_id` を保存する。32 ビット仮想アドレス空間における C++ のポインタ変数と同様に、永続ポインタは4バイト長である。揮発オブジェクトと永続オブジェクトはポインタ型が異なる点を除けば構造的に同一であり、かつそれらのサイズ互換性も保証されている。

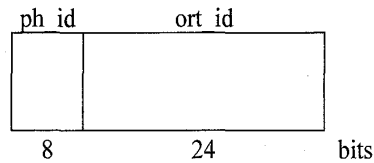


図3 永続ポインタのフォーマット

3. 2 手続き

永続オブジェクトに対する型の作成・追加・操作・削除等の手続きは、次の通りである。

Procedure 1：型の作成

- (1) `d_Ref<TypeName> varname` 文があれば、TypeID 表を調べて、`TypeName` が登録されているか否かを確認する。登録されていないければ、その型を登録する。`TypeName` に対する TypeID を得る。
- (2) `new` 演算子の引数として指定された永続ヒープ上の ORT に空エントリを探す。
- (3) OS 上に領域を割当て、永続ヒープの先頭からのオフセット、TypeID、そのエントリに対する OID を求める。OID 値は、そのエントリを指す。エントリの NextID は、null に設定される。

Procedure 2：型の追加

- (1) TypeID 表を調べて、`TypeName` が登録されているか否かを確認する。登録されていないければ、その型を登録する。`TypeName` に対する TypeID を得る。
- (2) 型が追加されるオブジェクトを指す OID から NextID チェインを辿っていき、NextID の値が null であるエントリを探す。
- (3) `new` 演算子の引数として指定された永続ヒープ上の ORT に空エントリを探す。
- (4) OS 上に領域を割当て、永続ヒープの先頭からのオフセット、TypeID、そのエントリに対する OID を求める。エントリの NextID は、null に設定される。(2)で求めた NextID が(3)で探したエントリを指すようにする。

Procedure 3：型による永続オブジェクトの操作

- (1) オブジェクトが操作される型の TypeID を TypeID 表から得る。型は `as` 構文により指定されるか、あるいは暗黙的に永続ポインタ変数の宣言で指定される。
- (2) その TypeID を持つ ORT エントリを探し、その仮想アドレスを返す。見つからなければ、nil オブジェクトの仮想アドレスを返す。

Procedure 4：型の削除

- (1) 型の TypeID を探す。見つからなければ、オブジェクトはその型を持たないため、終了する。
- (2) オフセット値が指す領域を解放する。
- (3) NextID チェインのエントリを取り除く。

(4) 再利用のため、そのエントリを無効とする。

Procedure 5：削除

- (1) NextID チェインを辿って、オフセット値が指す全ての領域を解放する。
- (2) NextID チェインを辿って、再利用のため全てのエントリを無効とする。

Procedure 1 から 5 までを使えば、クラス定義に現れる "this as TypeName" 構文以外のマルチタイプオブジェクトの機能は利用できる。this は C++ に同じである、すなわち this は永続ポインタではなく、仮想アドレスである。このため、Procedure 3 は使えない。代りに、次の手続きが必要となる。

Procedure 6："this as TypeName" の処理

- (1) "this as TypeName" 文があれば、this が現れるクラスの型に対するオブジェクトの領域を指す ORT エントリを this の仮想アドレスから計算する。
- (2) ORT エントリ中の OID フィールドから OID を得る。
- (3) Procedure 3 を行う。

図4は、CLASSI 型の永続オブジェクトの作成時、このオブジェクトへの CLASSJ 型の追加時に行われることを示す。

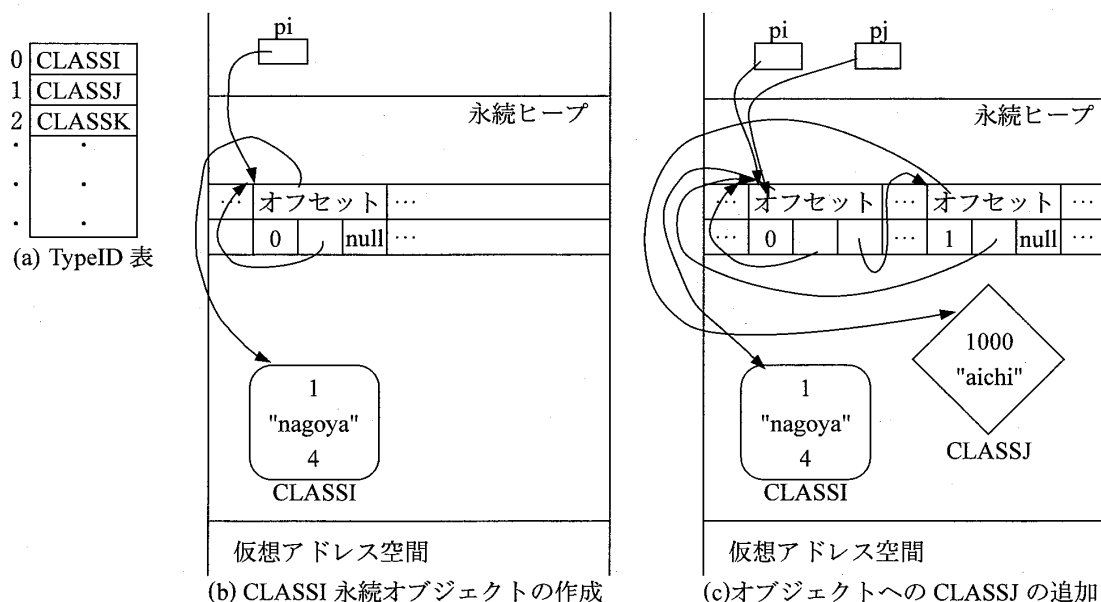


図4 オブジェクト作成と型追加

4 応用

本節では、オブジェクト・ビュー、集合オブジェクトの多重実装法、SGML文書データベースといった应用を使って、マルチタイプオブジェクト機能の有効性を示す。マルチタイプオブジェクト機能の高い潜在力を示すためにこれらの应用を選んだが、他の類似研究ではこれらの应用は見落とされている [1], [2], [3], [4], [5], [6], [7], [8], [9]。

4.1 オブジェクト・ビュー

ビューは、应用固有なデータの見方、データの安全性の単位、データ再構成の機能、スキーマ変更に対するデータ独立性を提供する。オブジェクト指向データベースシステム [21], [22], [23], [24] では、唯一のクラス階層が存在し、かつクラスがオブジェクトの入れ物であるため、ビューは問合せが作る仮想クラスとして実現される。クラス階層は、オブジェクト指向データベースにおける属性/メソッドの継承を決める。従って、問合せが定義する仮想クラスを既存のクラス階層の適切な場所に配置することが、重要な問題となる。この問題は、新しいビューを作成する度に起きる。この点において、このオブジェクトビューの実現法は問題である。

本論文では、これに代わるアプローチを提示する。INADA では、クラスはオブジェクトの入れ物ではなく、集合オブジェクトがオブジェクトの入れ物となる。そして、ビューは仮想集合として実現される。集合オブジェクトはクラス階層とは独立しており、クラス階層の再構成は必要ない。ビュー作成には、集合オブジェクトにビューに対する型を追加すればよい。また、ビューが必要なくなれば、集合オブジェクトからその型を削除すればよい。

INADA は、ビュー定義に次の形式を提供する。

```
view ViewSet on BaseSet {
    ai1 for aj1
    ai2 for aj2
    ...
    aik for ajk
    where condition
};
```

ここで、 a_{il} ($l=1, 2, \dots, k$) はビューの属性/メソッドを、 a_{jl} ($l=1, 2, \dots, k$) は基底オブジェクトの属性/メソッドを表わす。キーワード **view** は、ビュー定義を表わす。 a_{il} for a_{jl} ($l=1, 2, \dots, k$) はビューが属性/メソッド a_{il} を持ち、それが基底オブジェクトの属性/メソッド a_{jl} に対応することを表わす。尚、 a_{jl} は public でなければならない。

関係データベースにおける射影や選択により定義されるビューの実装は、以下の処理による。尚、以降では、この2種類のビューを各々、射影ビューと選択ビューと呼ぶ。

- (1) 射影ビューとして、 $a_{i1}, a_{i2}, \dots, a_{ik}$ を属性/メソッドとして持つC++クラス View を定義する。このクラスのインスタンス・オブジェクトは、(2)で定義される ViewSet クラスのインスタンス・オブジェクトの要素であり、仮想的に存在する。ViewSet のインスタンス・オブジェクトは、既に定義されている BaseSet クラスのインスタンス・オブジェクトのビューを構成する。
- (2) Viewset クラスが、INADA で定義される。このインスタンス・オブジェクトの要素の型は、射影ビューでは View クラス、選択ビューでは Base クラスとなる。Base クラスは、既に定義されている。ViewSet クラスのメソッドは、"OID as Type" という型の参照を除いて、BaseSet クラスのメソッドにより評価される。選択ビューとしての ViewSet のインタフェースは、condition 部に現れる選択述語を含む。
- (3) ViewSet クラスのオブジェクトは、基底オブジェクトの別クラス（あるいは型）のオブジェクトとして作成される。言い換えれば、マルチタイプオブジェクトは BaseSet 型のオブジェクトに ViewSet 型を追加することにより作成される。このマルチタイプオブジェクトは、基底集合オブジェクトのビューである。
- (4) ビューをアクセスする文は、基底集合オブジェクトをアクセスする文に翻訳される。BaseSet と ViewSet は共に INADA の集合クラスと同じインタフェースを持つため、この変換処理は自動的に行われる。

ビューに関して行うことは、集合オブジェクトを基底集合のビューとして作成すること、必要なくなった時点でそれを削除することである。ビュー集合の要素クラスは (1)で定義されるが、このクラスに対するオブジェクトは実際には作成されない。従って、これに伴う負荷はない。

会社の従業員を例とする。各従業員は名前、年齢、部門番号、給与額といった属性を持つ Employee クラスによりモデル化される。そして、集合オブジェクトが Employee クラスのオブジェクトを保持することとする。図5は、この例に対するクラス定義であり、Set はテンプレート・クラスとして実装される。

```
class Employee{
public:
    char name[40];
    int age;
    int deptno;
private:
    int pay;
public:
```

```

Employee(char* n="No-Name", int a=0, int d=-1, int p=-1)
{strcpy(name, n);
 age=a;
 deptno=d;
 pay=p; } // Constructor
~Employee( ); // Destructor
char* Name( ){return name; }
int Age( ){return age; }
int DeptNo( ){return deptno; }
int Pay( ){return pay; }
Change_Dept(int d) {deptno=d; } // Update
};

template <class T>
class Set{
public:
    Set( ); // Constructor
    T* GetElement(iterator<T>* i);
    iterator<T>* Next(iterator<T>* i);
    iterator<T>* OpenScan(iterator<T>* I);
    void CloseScan(iterator<T>* I);
};

```

図 5 Employee クラスと Set クラスの定義

年齢 50 歳以上の従業員を返すビューは、INADA では次のように記述される。

```

view Combview on Set<Employee> {
    char* Name( ) for Name( );
    where Age( ) >=50 ;
}

```

この定義は、それ自身の属性、選択条件を指定する where 節を持つ。これは、射影と選択を組合せたビューであり、図 6 のように翻訳される。Combview_Employee クラスは、ビュー定義より public メソッド Name() のみを持つ。条件 Age() >=50 は、クラス Combview の標準インタフェースに構築される。

```

class Combiview_Employee{
private:
    char name[40];
    char age;
    char deptno;
    int pay;
public:
    char* Name( ){return ((Employee*)this)->Name( );}
}

```

```
protected:
    int Age( ) {return ((Employee*)->Age( ));}
    friend class Combiview;
};
class Combiview{
public:
    Combiview_Employee* GetElement(iterator<Combiview_Employee>* i){
        return (Combiview_Employee*)
            (this as Set<Employee>->GetElement((iterator<Employee>*) i));}
    iterator<Combiview_Employee>* Next(iterator<Combiview_Employee>* i){
        iterator<Employee>* tmp
            =this as Set<Employee>->Next((iterator<Employee>*) i);
        if (! tmp) {
            CloseScan((iterator<Combiview_Employee>*)tmp);
            return 0;}
        if ((GetElement((iterator<Combiview_Employee>*)tmp)->Age( ))>=50)
            return (iterator<Combiview_Employee>*) tmp ;
        else tmp= (iterator<Employee>*) Next((iterator<Combiview_Employee>*) tmp) ;
        return (iterator<Combiview_Employee>*) tmp;}
    iterator<Combiview_Employee>* OpenScan(iterator<Combiview_Employee>* i){
        iterator<Employee>* tmp
            =this as Set<Employee>->OpenScan((iterator<Employee>*) i);
        if (! tmp) {
            CloseScan((iterator<Combiview_Employee>*)i);
            return 0;}
        if ((GetElement((iterator<Combiview_Employee>*)tmp)->Age( ))>=50)
            return (iterator<Combiview_Employee>*) tmp;
        tmp= (iterator<Employee>*) Next((iterator<Combiview_Employee>*)tmp);
        return (iterator<Combiview_Employee>*) tmp;}
    void CloseScan(iterator<Combiview_Employee>* i){
        this.as(Set<Employee>->CloseScan(iterator<Employee>*)i);}
};
```

図6 射影と選択の組合せビュー

4. 2 集合オブジェクトの多重実装

ハッシュ集合、索引付け集合、B 氏木や B+氏木による集合など [25], [26], [27], 検索と更新を含む処理性能改善のため、オブジェクト集合の構造に関する研究が行われてきた。これらの研究は、優れたデータ構造を提案し、記憶コストの解析やコスト性能を示している。しかし、複数の応用間でオブジェクト集合が共有される場合、各応用のニーズは互いに異なるため、全ての応用にとって最も良い集合の実装は選択できない。集合が多重実装を持ち、個々の応用の性能改善がされれば、最も良い実装の選択は必要ない。

多重実装機構は、マルチタイプオブジェクト機能を集合に適用することで実現可能である。集合の多重実装のアイデアは、集合の実装を型として定義し、集合の実装が必要（不必要）

になった時点で型を獲得（喪失）させることである。この機構は、集合に複数アクセスパスを持たせ、集合を共有する全ての応用の性能改善を図る。しかし、時間とともに、集合操作も変わりうる。そのため、多重実装による集合の処理性能は、それがない場合より悪くなることもある。それでも、この機構は有用であり、新たな操作が必要になれば集合に新しい機能を作成できる。

集合オブジェクトのインタフェースに加えて、ODMG 標準の `d_List` [14], [15], [16], [17] に類似の可変集合オブジェクトの演算子が提供される。この集合と `d_List` との主な差異は、このインタフェースが索引に対するキー値を扱うことにある。以下で、`K` はキーの型を示し、`H` はキーから値を計算する関数の名前を表わす。

- `d_IndexedIterator<T, K, H> create_iterator() const ;`

この演算子は、集合の先頭要素を指す `d_IndexedIterator` オブジェクトを返す。集合が要素を持たないなら、`NULL` 歴訪子を返す。

- `d_IndexedIterator<T, K, H> create_iteratorOP(K k) const ;`

`OP` は、`EQ` (equal to), `GT` (greater than), `GE` (greater than or equal to), `LT` (less than), `LE` (less than or equal to) である。この演算子は、`OP` を満たす集合の先頭要素を指す `d_IndexedIterator` オブジェクトを返す。集合が要素を持たないなら、`NULL` 歴訪子を返す。`k` は、要素のキー値を示す。

- `void insert_element(const T &elem, const K k) ;`

この演算子は、引数が指定する要素を集合に挿入する。

- `void remove_element(const T &elem) ;`

この演算子は、引数が指定する要素を集合から取り除く。

- `int next(T &objRef) ;`

この演算子は歴訪子の終了をチェックし、歴訪子を進めて、最新要素を返す。歴訪子が `create_iteratorOP(K)` で作成されたなら、歴訪子は `OP` を満たす要素にのみ働く。

オブジェクトの集合に対して、基底集合型と付加集合型の2種類の型が導入される。基底集合型は、オブジェクトのコレクションを管理する集合オブジェクトを作成する。ODMG 標準で定義されるコレクション・クラスは、基底集合型である。付加集合型は、集合オブジェクトに対して、型の1つとして追加されるアクセスパスを作成する。従って、集合オブジェクトは唯一の基底集合型と複数の付加集合型を保持する。図7は、`Person` オブジェクトの集合に対する例である。

図7では、付加集合の要素は隣接する要素の参照、キー値、基底集合が持つノードの参照を保持する。ノードは要素を保持する抽象物であるが、オブジェクトのコレクションを実装する際に具体物となる。ノードは、付加集合型を実装するために必要である。ODMG 標準の歴訪子により、ノードの置換えが可能であるが、歴訪子はデータベースに保存不可能である。

これが、ノードを導入した理由である。標準化仕様はコレクション・クラスの実装方法は問わず、ODMG 標準はこのノードを定義していない。

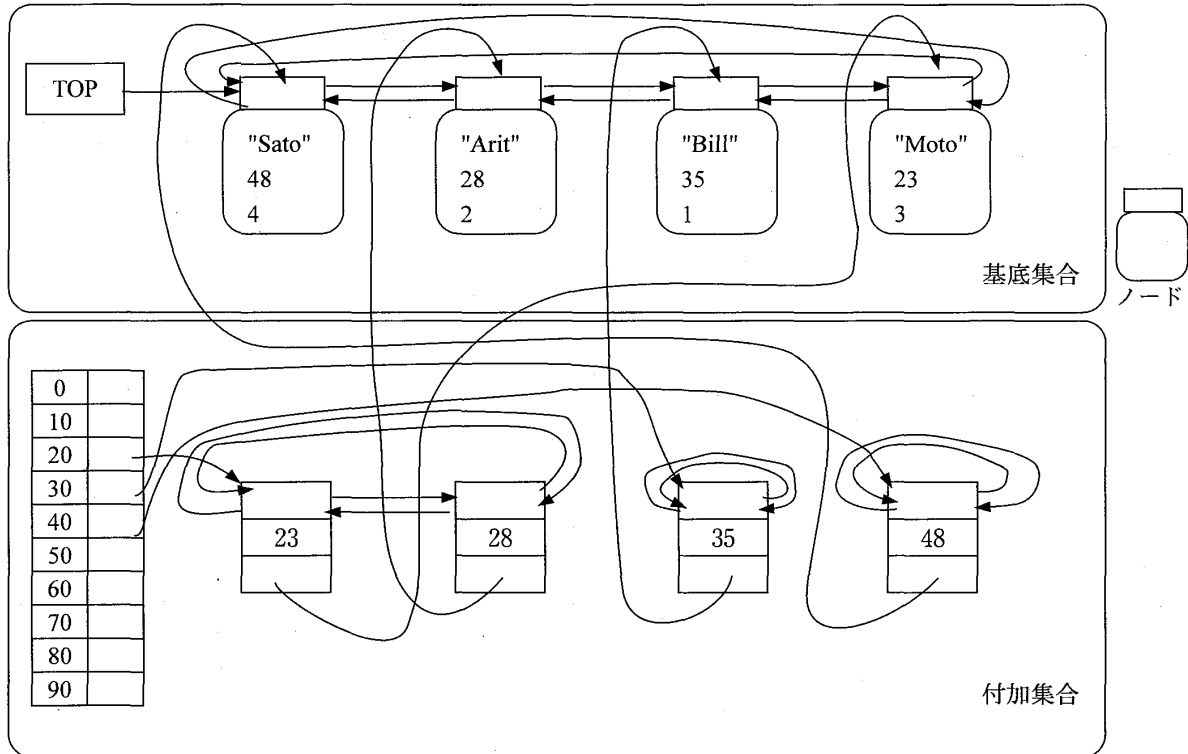


図7 基底集合と付加集合

age に索引付けられた付加集合型 IndexedbyAge を持つ Person オブジェクトの集合 pset を想定する。以下において、集合の要素 Person の多くが条件を満たさない場合、S1 の実行性能は S2 の実行性能よりよい。

```

(S1)
d_IndexedIterator<Person,AGE,Age> iter=pset as IndexedbyAge->create_iteratorGT (30) ;
Person p;
while (iter.next(p)){
    cout<<p.Name() <<endl;
}

(S2)
d_Iterator<Person> iter=pset->create_iterator( );
Person p;
while (iter.next(p)){
    if (p.Age()>30) {
        cout<<p.Name() <<endl;
    }
}
    
```

4. 3 SGML 文書データベースの問合せ

近年, SGML [28], [29] のような構造化文書が, ソフトウェア工学, デジタル・ライブラリ, WWW などに広汎に使われている。SGML は, 独自構造を持つ文書情報交換用の国際標準である。SGML による文書情報の表現は, 再利用と配布を促進する。さらに, SGML 文書データベースは SGML 標準の長所を促進し, その研究開発が広く進められている。SGML 文書応用の研究課題は, (1) SGML 文法のデータベーススキーマへの効率的な変換, (2) SGML 情報検索のための問合せ言語, (3) SGML 問合せをサポートする効率的な索引構造である[30]。

以降では, 上記の課題 (1), (2) に焦点をあてる。そして, 内容検索と構造検索をサポートする SGML 文書管理システムの実装法を提案する。SGML 文書は単なるテキストであると同時に, テキスト内に埋め込まれたタグの対<tag>, </tag>により構造化される。タグ<tag>と</tag>は, 各々, 文書の構造要素の開始と終了を表わす。構造要素は入れ子構造となるため, SGML は精緻な文書構造を正確に記述可能である。SGML 文書の保存・検索のため, それはマルチタイプオブジェクトとして表現され, 特定の文書を定義する型が単なるテキストを保存する型に追加される。そして, SGML 文書オブジェクトの問合せ OQL 文が, 文書検索に固有なメソッドを使って作成される。

図 8 は, SGML 文書管理の実装例に関する OMT 図 [31] である。図 8 で, クラスはシステム定義類とユーザ定義類に分けられる。前者は SGML 文書管理システムのシステム設計者により, 後者は SGML 文書応用システムのユーザにより定義される。

SGMLText クラスは, 出世魚データベース管理システムのラージオブジェクト [32] として, SGML 文書ファイルを登録する。ラージオブジェクト自体は永続分散共有仮想記憶 [33] 上にマップされるファイルであり, ネットワーク上でユニークな識別子, すなわち永続オブジェクト・ポインタを与えられる。SGMLText クラスは SGML 文書を単なるテキストとして扱い, 文書構造に関連するメソッドは持たない。StructuredSGMLText クラスは, SGML 文書を文書構造を有する複合オブジェクトとして扱う。このクラスは, オブジェクトに追加される型として, 文書構造に対応する一種のオブジェクトビューを定義する。クラス SGMLElement と SGMLElementList は, 各々, SGML 文書から動的に抽出される構造要素と構造要素リストをオブジェクトとして扱う。

SGML 文書構造は, 文書型定義 (Document Type Definition ; DTD) により定義される。文書型 Paper は, Title, Abstract, Authors, Sections から成るとする。前記のシステム定義クラスが与えられると, 文書型 Paper に対するユーザ定義クラスが定義される。Paper クラスは SGMLText の下位クラスであり, 単なるテキストとして定義される。StructureText クラスは, StructuredSGMLText の下位クラスであり, 固有な文書構造を有する複合オブジェクトとして定義される。同様に, 各構造要素は SGMLElement あるいは SGMLElementList の下位クラスとして定義される。これらのクラス定義により, 単なるテキストとしての SGML 文書から動的に各文書構造が抽出される。Paper オブジェクトは単なるテキストとして, あるい

は StructuredPaper 型を通せば固有な構造を持つ文書として操作可能である。

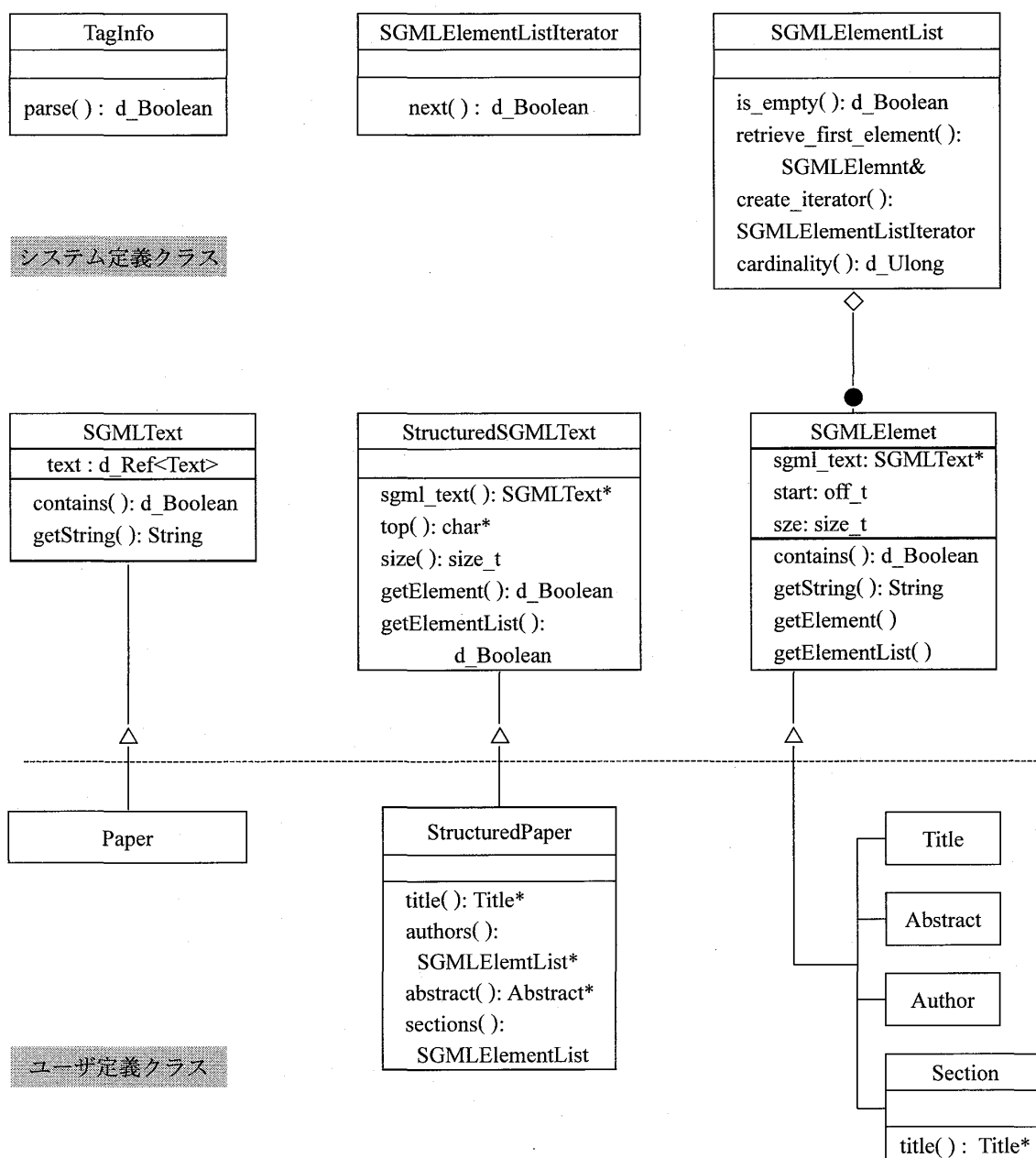


図8 システム定義クラスとユーザ定義クラス

表2は、SGML 文書オブジェクトに対する問合せコマンドを示す。Q5は、OQLによる内容検索例である。Q5のPapersは、Paper型のエクステンツである。メソッドcontains("database")はキーワード検索に固有であり、システム定義クラスSGMLTextで実装される。Q6は構造検索例であり、Paperオブジェクトは複合オブジェクトとして扱われる。この問合せでは、マルチタイプオブジェクト機能のメソッドas("StructuredPaper")がStructuredPaper

ビューを通して Paper をアクセスするために使われる。Q7 は、属性を使った検索例である。メソッド `attribute("date")` は SGML 文書検索に固有であり、システム定義クラス `StructuredSGMLText` で実装される。Q8 は文書ランキング検索 [34] の例であり、検索結果は関連性の順にソートされる。メソッド `contains_with_rank("database")` は文書ランキングに固有であり、引数として与えられるキーワードを有する文書を探し、文書の関連度を計算する。計算された関連度はオブジェクトの private データメンバに保存され、ランキングのために `order by` 節に現れるメソッド `score()` により参照される。これらのメソッドは、システム定義クラス `StructuredSGMLText` で実装される。

表 2 SGML 文書オブジェクトに対する問合せコマンド

<p>Q5: 用語"database"を含む論文を求めよ。</p> <pre>select p from Papers p where p.contains("database")</pre>
<p>Q6: 用語"database"を含む論文の節を求めよ。</p> <pre>select s from Papers p, ((StructuredPaper)p.as("StructuredPaper")).sections s where s.contains("database")</pre>
<p>Q7: <date>属性が"1997-9-21"である論文を求めよ。</p> <pre>select ((StructuredPaper)p.as("StructuredPaper")) from Papers p where ((Date) ((StructuredPaper)p.as("StructuredPaper")).attribute("date"))= '1997-9-21'</pre>
<p>Q8: 用語"database"を含む論文の節を関連度の順にランキングせよ。</p> <pre>select s from Papers p,((StructuredPaper) p.as("StructuredPaper")).sections s where s. contains_with_rank("database") order by s.score() desc</pre>

5 関連研究

Iris [1] は、オブジェクトが型の動的な獲得・喪失を可能とした最初のシステムである。しかし、オブジェクトが同一の属性名/メソッド名を持つ型に属する場合、名前の競合が起きる。Object specialization [2] は、型単位でなく、オブジェクト単位での継承機構を提供する。オブジェクトは親からの委譲により振舞いを継承する。型のないプロトタイプに基づく言語は自由度が高すぎるため、大規模システムのオブジェクト管理は困難となる。

Clover [3] は、多重の役割 (ロール) を持つオブジェクトをモデル化する。型 A のオブジェ

クトはその下位型 B のインスタンスとなることができ、型 B に固有な属性／メソッドを獲得する。型階層を上昇・下降するための演算子は提供されるが、明示的に指定された型を通してオブジェクトをアクセスすることはできない。

Aspect 機構 [4] は、オブジェクトの型が実装とは分離している独自のデータモデルに基づき、オブジェクトの多重の役割（ロール）をサポートする。型はオブジェクトをアクセスするインタフェースのみを提供し、aspect は型の実装として定義される。例えば、型が B であるとして作成されたオブジェクト b のアスペクトとして、オブジェクト a は型 A に対して作成される。この場合、a と b とは同一オブジェクトの異なる aspect を表わす。オブジェクト識別子とオブジェクト参照との2つの概念が、オブジェクトと aspect とを識別するために導入されている。

Fashion [5] は、オブジェクトの全体を一部分として、逆にオブジェクトの一部分を全体として扱う手段を提供する。しかし、この機能は、すでに定義された型の属性／メソッドにのみ適用可能となっている。

Fibonacci [6] は強い型付けのオブジェクト指向データベース・プログラミング言語であり、役割（ロール）を持つオブジェクトをモデル化する能力を持つ。この言語は、独自のデータモデルに基づいている。また、aspect 機構にはないが、Fibonacci は継承をサポートしている。オブジェクト代理モデル [7] は、オブジェクト・ビュー、役割（ロール）を持つオブジェクト、オブジェクトマイグレーション、多重継承を実現するために提案された。このため、このモデルは代理オブジェクトを原オブジェクトに関連付けている。しかし、結果として、実世界における同一の実体に対して異なるオブジェクト識別子の割当てが行われる。

文献 [8] は、オブジェクト進化を扱うために役割（ロール）概念の必要性を唱えた。その実装は、Gemotone の OPAL といった Smalltalk [10] に基づくオブジェクト指向データベース・プログラミング言語に容易に移植可能である。Aspect 機構 [4] と同様に、実体識別性と役割識別性とが導入され、2つのインスタンスが同じ実世界の実体に対応しているか否かを調べる実体等価性の概念が存在する。

他の研究と比較して強調したい点は、本研究は世界に広く普及することが期待されている ODMG 標準に準拠していることにある。他の研究はすべて各々に独自のデータ／オブジェクトモデルに基づいており、結果としてそれらを使うためには個々のモデルの学習が必要となる。本論文では、他の研究に類似したマルチタイプオブジェクト機能を提案し、これを INADA と OQL に組込んだ。INADA は拡張 C++ 永続プログラミング言語であり、ODMG C++ Binding の具体化である。従って、INADA のデータモデルは、世界に普及しているプログラミング言語 C++ のモデルに基づいている。また、マルチタイプオブジェクト機能に必要な実装は永続ポインタ `d_Ref<T>` 型のメソッド `as(TypeName)` に閉じることができたため、ODMG OQL へのマルチタイプオブジェクト機能の組込みにはその言語仕様の拡張は必要ない。著者が知る限り、本研究は、OQL のような高水準問合せ言語にマルチタイプオブジェクト機能に

類する機能の組み込みを行なった最初のシステムである。本研究の成果は ODMG 標準に基づく他のオブジェクト指向データベースに適用可能である。

6 むすび

本論文では、マルチタイプオブジェクト機能を提案し、ODMG 標準に準拠する INADA と OQL への導入を議論した。実世界における実体を持つアスペクトの動的な変更は、マルチタイプオブジェクト機能によりモデル化できる。さらに、INADA へのマルチタイプオブジェクト機能の実装法についても述べた。また、マルチタイプオブジェクト機能の有効性がオブジェクト・ビュー、集合オブジェクトの多重実装法、SGML 文書データベースといった応用を使って示された。これらの応用はマルチタイプオブジェクト機能の高い潜在力を示すために選ばれたが、他の研究では見落とされている。著者が知る限り、本研究以外に、OQL のような高水準問合せ言語にマルチタイプオブジェクト機能に類する機能の組み込みを行なった研究はない。ODMG 標準に対する本拡張提案は、標準化の目的に矛盾するものではない。標準自体は、現実に将来の改定事項を含むものである。

最後に、マルチタイプオブジェクト機能に関して、今後の課題を指摘しておく。第1の課題として、提案された機能では、オブジェクトは同一の型のアスペクトを複数持つことができない。例えば、ある人は2ヶ所で働くことが可能であるが、本論文で提案されたマルチタイプオブジェクト機能は、例えば **Employee** 型の2つのアスペクトを持つオブジェクトを実現できない。しかし、著者は、この問題に対処する拡張データモデル [9] を別途提案している。第2の課題として、マルチタイプオブジェクト機能は型の間の一貫性制約を扱うべきである。例えば、オブジェクト **Person** に **Part** 型を追加することは、一般的な応用において意味をなさない。この場合、ユーザはオブジェクトの完全性を保つために一貫性制約の導入が必要となる。同様に、著者は、マルチタイプオブジェクト機能に関連する一貫性制約を管理のための枠組みを別途提案している [35], [36], [37], [38]。従って、これら2つの課題に関連する提案を INADA に具体化するための検討が急務となる。

注

- 1) 関数 `transform` と `as` は、`d_Ref<T>` 型のメソッドとしても提供される。
- 2) 本研究におけるアスペクト概念と区別するために、文献 [4] の類似概念を `aspect` と記す。

参考文献

- [1] D. H. Fishman et al., "Iris: An Object-Oriented Database Management System", *ACM Trans. Office Information Systems*, Vol. 5, No. 1, 1987, pp. 48-69
- [2] E. Sciore, "Object Specialization", *ACM Trans. Information Systems*, Vol. 7, No. 2, 1989, pp. 103-127

- [3] L. A. Stein and S. B. Zdonik, "Clovers : The Dynamic Behavior of Type and Instances", *Brown University Technical Report*, No. CS-89-42, 1989
- [4] J. Richardson and P. Schwarz, "Aspects : Extending Objects to Support Multiple, Independent Roles", *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1991, pp. 298-307
- [5] G. Moerkette and A. Zachmann, "Multiple Substitutability without Affecting the Taxonomy", in *Advances in Database Technology-EDMT92*, A. Pirotte, C. Delobel, and G. Gottob (ed.), Springer-Verlag, 1992, pp. 120-135
- [6] A. Albano, R. Bergamini, G. Gheli, and R. Orsini, "An Object Data Model with Roles", *Proceedings of International Conference on Very Large Data Base*, August 1993, pp. 39-51
- [7] Y. Kambayashi and Z. Peng, "Object Deputy Model and Its Applications", *Proceedings of International Conference on Database Systems for Advanced Applications*, April 1995, pp. 1-15
- [8] G. Gottlob, M. Schrefl, and B. Rock, "Extending Object-Oriented Systems with Roles", *ACM Trans. Information Systems*, Vol. 14, No. 3, pp. 268-296, 1996
- [9] 佐藤秀樹, 池田峰輝, 舟橋榮, 林達也, "多面的オブジェクト指向データモデル MAORI", *電子情報通信論文誌*, Vol. J79-D-I, No. 10, 1996年10月, pp. 781-790
- [10] A. Goldberg and D. Robinson, "*Smalltalk80: The Language and Its Implementation*", Addison-Wesley, 1983
- [11] B. Stroustrup, "*The C++ Programming Language*", Addison-Wesley, 1986
- [12] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks", *Communication of ACM*, Vol. 13, No. 2, February 1970, pp. 377-387
- [13] E. F. Codd, "Relational Completeness of Data Base Sublanguages", in *Data Base Systems*, Prentice-Hall, 1972, pp. 33-64
- [14] R. G. G. Cattell (ed.), "*The Object Database Standard: ODMG-93 Release 1.1*", Morgan Kaufmann, 1994
- [15] R. G. G. Cattell (ed.), "*The Object Database Standard: ODMG-93 Release 1.2*", Morgan Kaufmann, 1996
- [16] R. G. G. Cattell and D. K. Barry (ed.), "*The Object Database Standard: ODMG 2.0*", Morgan Kaufmann, 1997
- [17] R. G. G. Cattell and D. K. Burry (ed.), "*The Object Data Standard: ODMG 3.0*", Morgan Kaufmann, 2000
- [18] M. Stefik and D. G. Bobrow, "Object-Oriented Programming : Themes and Variations", *The AI Magazine*, Vol. 6, No. 4, 1986, pp. 182-204
- [19] G. Bai and A. Makinouchi, "WAKASHI/D : A Distributed Storage Server for New Generation Database Systems", *Proceedings of International Symposium on Advanced Database Technologies and Their Integration*, 1994, pp. 137-144
- [20] M. Aritsugi, K. Teramoto, G. Bai, and A. Makinouchi, "Several Implementations of Persistent Pointers in a Memory Mapped I/O Environment", *Proceedings of International Conference on Database and Expert Systems Applications, Lecture Notes in Computer Science 978*, Springer-Verlag, September 1995, pp. 490-501
- [21] S. Heiler and S. Zdonik, "Views, Data Abstraction, and Inheritance in the FUGUE Data Model", *Proceedings of International Workshop on Object-Oriented Database Systems, Lecture Notes in Computer Science 334*, Springer-Verlag, September 1988, pp. 225-241
- [22] E. A. Rundensteiner, "MultiView; A Methodology for Supporting Multiple Views in Object-Oriented Databases", *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1992, pp. 298-307
- [23] M. H. Scoll, C. Laasch, and M. Tresch, "Update Views in Object-Oriented Databases", *Proceedings of International Conference on Deductive and Object-Oriented Databases*, December 1991, pp. 189-207
- [24] K. Tanaka, M. Yoshikawa, and K. Ishihara, "Schema Virtualization in Object-Oriented Databases", *Proceedings of International Conference on Data Engineering*, February 1988, pp. 22-29
- [25] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indexes", *Acta Informatica*, Vol. 1, 1972, pp. 173-189
- [26] E. Bertino and W. Kim, "Indexing Techniques for Queries on Nested Objects", *IEEE Trans. Knowledge and Data Engineering*, Vol. 1, No. 2, 1989, pp. 196-189
- [27] A. Kemper and G. Moerkotte, "Access Support in Object Bases", *Proceedings of ACM SIGMOD International*

- Conference on Management of Data*, 1990, pp. 364-374
- [28] International Organization for Standardization, "Information Processing -Text and Office Systems- Standard Generalized Markup Language (SGML) ", ISO/IEC 8879, 1990
- [29] C. F. Goldfarb, "*The SGML Handbook*", Oxford University Press, 1990
- [30] R. Sacks-Davis, T. Arnold-Moore, and J. Zobel, "Database Systems for Structured Documents", *Proceedings of International Symposium on Advanced Database Technologies and Their Integration*, 1994
- [31] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991
- [32] K. Keneko and A. Makinouchi, "Data Storage and Query Processing for Structured Document Database", *Proceedings of International Conference on Database and Expert Systems Applications*, 1997, pp. 92-97
- [33] G. Yu, K. Kaneko, G. Bai, and A. Makinouchi, "Transaction Management for a Distributed Object Storage System WAKASHI-Design, Implementation and Performance", *Proceedings of International Conference on Data Engineering*, 1996, pp. 460-468
- [34] 小川泰嗣, "文字成分表を用いた効率的文書ランキング検索方式", 情報処理学会論文誌, Vol. 38, No. 11, 1997 年 11 月, pp. 2286-2297
- [35] 佐藤秀樹, 舟橋栄, 林達也, "多面的オブジェクトに対するオブジェクトマイグレーションの枠組み", 電子情報通信論文誌, Vol. J81-D-I, No. 3, 1998 年 3 月, pp. 271-282
- [36] H. Sato and T. Hayashi, "Object Migration Behavior Modeling with Petri-Nets", *Proceedings of IASTED International Conference, Artificial Intelligence and Soft Computing*, May 1998, pp. 250-253
- [37] 佐藤秀樹, 林達也, "カラーペトリネットによるオブジェクトマイグレーションの振舞いモデリング", 情報処理学会論文誌: データベース, Vol. 40, No. SIG8 (TOD4), 1999 年 11 月, pp. 13-28
- [38] H. Sato and A. Makinouchi, "Temporal Constraints for Object Migration and Behavior Modeling Using Colored Petri Nets", *Proceedings of International Conference on Conceptual Modeling*, October 2000

謝辞 本研究は、九州大学大学院システム情報科学研究院・教授牧之内顕文氏，群馬大学工学部情報工学科・助教授有次正義氏との共同研究に基づくものである。この事を記して，感謝の意を表わしたいと思います。