データベースシステム 99－2
（1994．7．20）

オブジェクト指向データベース操作言語の構成管理機能について

木村 裕　　鶴岡 邦敏
NEC C&C 研究所

オブジェクト指向データベース・システム上の構成管理モデルとそれを操作する C++ ベースの言語を提案する。本構成管理モデルは、構成を異なるインスタンス集合からなるビューとして定義する。構成ビューは、コンパイラのパーサ生成言語 - YACC に似たルールにより構成を記述する。構成ビューは、対応するクラスのエクステントを得ることによって具現化するか、または連係オブジェクトを使って関連付けた後、最初の要素参照で具現化され、この時点でデータベースからチェックアウトされる。このルールによる構成の記述により、スキーマ進化に対するプログラム独立性が向上する点を述べる。さらに、構成の要素となるオブジェクトを通常のオブジェクトと同じ式で参照できる実現方式について述べる。

An Object-Oriented Database Programming Language for a Configuration Based on a View Concept

Yutaka Kimura and Kunitoshi Tsuruoka
C&C Research Laboratories
NEC Corporation
4-1-1 Miyazaki, Miyamae-ku, Kawasaki 216, Japan

This paper proposes a configuration model on object-oriented database systems, and its manipulation language, based on C++. In the configuration model, a configuration is defined as a view, which consists of a set of instances for classes. A definition for a configuration view is described in rules. Rule notation is somewhat like that employed in YACC, which is a parser generator. A configuration instantiation is carried out by getting an extent for a configuration, or by the first reference its component object after associating the configuration to an association object. The component objects are checked out at the instantiation time. By using the rules, program independence from schema evolution is accomplished. Furthermore, the authors states how a reference to a component object in a configuration is implemented.

1 はじめに

ある製品のデータベースを設計・開発するとき、比較的その規模の大きい場合は、通常、複数の設計者によって分担して設計作業が進められる。この場合、共通のデータベースに製品全体の複合オブジェクトを構築していくことは困難である。このため、ワークスペース (または私的データベース) と構成管理が導入された [Katz90, Ohbo88]。各設計者に製品の要素部品 (component) を割り当て、これを1つの大きな構成として管理する。この構成は他の設計者が作った構成を含んでいるかもしれない。設計者は作業を開始または再開する際には、自分の構成をワークスペースに一括してチェックアウトする。構成自身の内容 (i.e. 実現) は、通常のオブジェクトによる表現と同じである。すなわち、異なる型のオブジェクトのある版により構成される複合オブジェクトである。このように、構成は設計者の意味のある設計あるいは作業単位であり、また他の設計者が担当している構成は、それを利用する設計者にとって抽象化された、その仕様のみ見える部品である。

オブジェクト指向データベースの研究が始まって以来、多くの構成管理に関連するモデルが提案されている。これらは大きく2つに分類できる。1つは、総称版インスタンス (generic version instance) を使う方法である [Ditt88, Kim89, Land86]。総称版は特定していない版であり、参照時に予め与えられた条件のオブジェクトが選択される。[Kim89] は複合オブジェクト内のオブジェクト間に、従属、非従属、排他、共有、普通の5種類の関連を提案し、この関連と生成 / 削除演算の意味について議論している。この提案も限定されてはいるがある種の構成と考えられる。[Ditt88] は、参照時の環境によって総称版を具現化する動的な方式を提案している。[Tale92] も総称版を提案しているが、さらに総称版クラスも導入している。これらの方式では、設計オブジェクトは単なる複合オブジェクトであり、構成のように1つの操作単位として扱うことができない。また複合オブジェクト (i.e. 構成) を一括してチェックアウト / インする概念がないので、複数設計者の同時利用の環境ではデータベースの一貫性が保たれない可能性がある。

もう1つの方式は、構成を1つの操作単位として扱えるものである [Katz87, Katz90, ODI91]。作業空間、構成単位のチェックアウト / インの概念を持つ。[Katz87, Katz90] では、さらに動的な構成の生成、コピーの伝搬についても議論している。[ODI91] は構成をプログラミング言語 C++ で扱うことができる本格的な商用システムである。構成オブジェクトが要素オブジェクトを管理しており、ロックや版管理の1つの単位とすることができる。しかし、これらの方式は構成が他のクラスのオブジェクトと独立した概念になっており融合されていない。たとえば、ある型の変数にその特定のオブジェクトだけでなく構成オブジェクトも値として持てるようになっていない。

我々が提案する構成モデルは次のような特長を持つ。まず、構成を様々な種類のオブジェクトからなるビューと考える点である。ビューは生成規則をもち、ビューのエクステントを得ることで具現化される。これは関係データベースのビューが、生成条件式を持ち、ビューへの問合せがあったときに具現化されるのに類似している。このビューを構成ビューと呼び、クラスで実現する。第2に、構成ビューの生成規則、すなわち構成の構造をクラス定義時にルールで記述する。ルールは要素クラスの構造を表現している。構成ビューが具現化されるとき、ルールで記述された構造がデータベースに存在するかどうかを検査する。これにより構成の具現化時に適切な構成を構築することができ、後に述べるようにクラスの変更に伴うプログラムの変更を避けることができる。第3に、1つの構成ビューは1つの型を持つ点である。これにより、構成固有の操作を定義でき、(構成以外の) 通常のクラスやインスタンスとまったく同じ方法で、構成クラスやその要素オブジェクトを参照できる。

以下で述べる構成モデルは、NEC C&C 研究所で研究開発されたオブジェクト指向データベース管理システム - Odin (製品名 PERCIO(ペルシオ)) をベースとしている [Kimu92, Tsur92]。Odin の操作言語は C++ を拡張した構文を持ち、その仕様は ODMG の標準仕様 R1.0 [Catt93] の一部を取り込んでいる。Odin の大きな特長の1つは、オブジェクト指向モデル上にビューを実現している点である [Kimu91a, Kimu92b]。

本論文では、第2節で構成モデルの概要を述べ、第3節以降でその操作言語について述べる。第3節で構成定義、および構成の生成と参照、第4節でこれらの実現方式について述べ、第5節で本論文をまとめる。

2 構成管理モデルの概要

この節では、本論文のベースとなっている構成管理モデルを概観する。図1に、以下の節で使用する飛行機データベースの構造を示す。図で大文字で始まる名前がクラス名、小文字がインスタンス変数名、矢印は他のオブジェクトとの関連を表す論理ポインタ、黒丸が1: 多関係を表す。

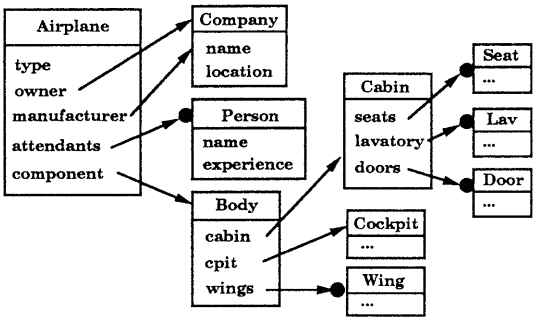

図 1: 飛行機データベース

このデータベースは1つの製品全体を表している。設計過程では、通常いくつかの要素部品に分けて複数の設計者により並行して作業が進められる。このように分割されたオブジェクトのグループを設計オブジェクトと呼ぶ。構成は、より基本的な要素部品オブジェクトの版からなる設計オブジェクトの版である。構成は設計者の責任の範囲であり作業単位である。しかし、実際の作業では、構成の全範囲を必要とせず、さらに部分構成に分割して作業しやすくする。言い替えると、構成の必要な部分だけを自分の作業空間にチェックアウトする。図2は、飛行機の機体 (Body) の一部を作業空間にチェックアウトしたときの図である。設計者は、 Body と Cabin を操作するが、このうち興味のあるのはインスタンス変数 cabin, cpit, seats だけである。このように、構成は設計者にある種のビューを与えていると考えることができる。

本モデルでは、構成をビューと考える。これを**構成ビュー**と呼ぶ。構成ビューのインスタンスは、構成を構成する各種の要素オブジェクトである。構成ビューはインスタンスを生成する条件式を持つ。通常のクラスのインスタンス集合がそのエクステントから得られるように、構成ビューのインスタンスもそのエクステントから得られる。すなわち、ビューのエクステント・インスタンスを生成したときに、ビュー・インスタンスの生成条件式が起動され、その結果が一括してチェックアウトされ構成ビューの

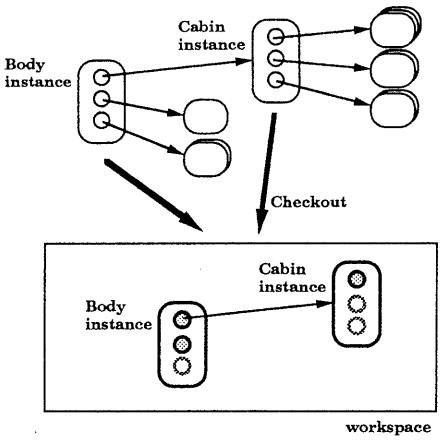

図 2: 構成のチェックアウト

エクステントの要素となる。

構成は、他の構成の一部になれる。これを可能にするためには、インスタンス変数は、構成の最上位の複合オブジェクトだけでなく、構成型自身もオブジェクトとして保持できなくてはならない。連係オブジェクト (association object) は、構成ビューと構成オブジェクトを参照するオブジェクトとの連係を取るために使用される。この連係クラスは、構成の最上位に位置する複合オブジェクトのクラスのサブクラスで実現されるので、参照するオブジェクトのインスタンス変数には、この連係オブジェクトを代入することが可能である。構成ビューが具現化したとき、最上位オブジェクトはその連係オブジェクトにマップされる。従って、構成ビューが具現化された後に構成内のオブジェクトを参照する場合、通常のオブジェクトと同じ構文で参照することが可能である。

3 構成操作言語機能

3.1 構成の定義

構成ビューの仕様は、クラス定義で規定される。構成ビュー・クラスは、ビューの生成条件式であるルールと、参照できるインスタンス変数とメソッドのリストと、構成ビュー固有の付加メソッド、からなる。図3に飛行機 B747 の機体設計用に作った構成ビュー・クラスの例を示す。スーパクラス Od_Pobject は永続性を与えるクラスであり、 ODMG R1.0 仕様 [Catt93] のクラス Pobject に相当する。“:od_spec” は、 Odin 固有の属性を与えるときのキーワードで、この後に各種の引数が指定できる。この場合の引数 “config<Body>” は、当該クラス

```
class B747 : Od_Pobject
           : od_spec config<Body>
{
 configuration:
   body -> cabin(Cabin) cpit(Cockpit)
           wings(Wing)
               { /* action */ }
   cabin -> seats(Seat) lavatory(Lav)
            doors(Door)
               { /* action */ }
 public:
   Body::cabin;  Body::cpit;
   Cabin::seats;
};
```

図 3: 構成ビュー・クラス定義

が構成ビュー・クラスであり、構成中の最上位クラスがクラス Body であることを示す。構成の構造は、"configuration:" の後にルールの形式で記述する。

ルールは、コンパイラの構文解析プログラムを自動生成する言語 **YACC** に類似した構文を持つ。図4に構文規則を示す。ルールの左辺は構成の要素となるクラスの名前、右辺はそのクラスの構造を定義する。クラスの構造は、インスタンス変数名、その型、導出条件、アクションの 4 つ組 (*quadruple*) で定義される。各ルールは右辺の型 (*type*) と同じ名前の左辺の型名によって関連付けられる。この関連が構成の構造を表現する。最初のルールが構成の最上位クラスを表す。このルール解釈プログラムは、最初に記述されているルールから解釈を初め、右辺を左から順に調べ関連をたどっていき、末端のルールに達したときそのアクションを実行して元のルールに戻る (還元処理)。つまり、(最左) 導出手続きに基づくトップ・ダウン解釈を行なう。ルールが導出されるとき、必ずそのルールに一致するクラスのスキーマがデータベース中に存在するかどうか検査され、もしなければエラー・ステータスをトランザクションに渡す。ルールに記述されているクラス名 (*class-name*) に対応するクラスがデータベース中に存在する場合、最新のスキーマの版から過去の版を探す。この機能により、スキーマの進化に依存しない構成の定義が可能となる。同じルール (左辺のクラス名が同じルール) が複数あるとき、ルールの定義順にスキーマ検査が行なわれる。このルールを選択するアルゴリズムをまとめるとアルゴリズム 1 のようになる。

config-rule-declaration:
    config-rule-declaration '\n' *rule*
    rule
rule:
    class-name → *rule-body*
    rule-name = *class-name* → *rule-body*
rule-body:
    rule-body quadruple
    quadruple
quadruple:
    variable (*type*) *d-condition action*
    variable (*type*) *action*
    variable (*type*)
d-condition:
    [*conditional-expression*]
action:
    { *statement-list* }

図 4: 構成規則の構文

[アルゴリズム 1]

1. 最初のルールを解析する。
2. データベース中の当該ルールの *class-name* に対応するクラスの版木を最新の版から過去にさかのぼって検索する。
3. 該当する版があれば、見つかったことを示す値を返す。
4. 該当する版がない場合、同じ *class-name* の次のルールを探す。そのようなルールが存在しない場合、見つからなかったことを示す値を返す。
5. 次のルールが存在した場合、(2) ～ (5) を繰り返す。

トランスレータは、上記のルールを検査するメソッドをそのクラスのクラス・メソッド (i.e. 静的メンバ関数) として出力する。このクラス・メソッドは次の節で述べるように構成ビューを具現化するときに起動される。また、ルールには名前 (*rule-name*) が付けられる。ユーザはこのルール名を使って特定の版のスキーマの構成を具現化できる。

アクションには、任意の式が書ける。ここには、データベース中に対応するスキーマが見つかった場合に実行する式を書く。通常、構成の要素となる適切な版オブジェクトを取り出す式やこれらの要素間の妥当性の検査等を記述する。

```
body ->
  cabin(Cabin) cpit(Cockpit) wings(Wing)
        { verify ($1, $2, $3); }
```

ここで、$\$i$は右辺の$i$番目の4つ組が還元されるときに返却される値へのポインタを保持する変数である。また、$$はこのルール自身が還元されるときに返す値へのポインタを保持する変数である。

本解釈方式の詳細、および構文における導出条件 (*d-condition*) については誌面の制約のためここでは述べない。

3.2 構成の生成と参照

構成ビュー・クラスのインスタンスは、通常のクラスと同様、そのエクステントで管理される。構成ビュー・インスタンスの生成、すなわち構成ビューの具現化は、このエクステント・オブジェクトを生成したときに行なわれる。通常のクラスおよび (構成ビュー以外の) ビューは、すべて同種の構造を持つインスタンスの集まりであるのに対して、構成ビューのインスタンスは、一般的に異種の構造を持つインスタンスの集まりで、1つの構成の実体を表している (複数の構成実体を表していない)。下記の式は、変数 ext1, ext2 にエクステント・オブジェクトを割り当てる例である。

```
Od_Extent<B747*>  ext1;
Body* top;
Od_Extent<B747*>  ext2 (top);
```

エクステントのコンストラクタは、要求されているインスタンスのクラスが構成ビュー・クラスの場合、そのクラスのルール解釈メソッドを呼び出し、このメソッドの実行の結果得られたインスタンス集合が構成ビューのエクステントとなる。この返却値は、必ず構成の最上位オブジェクトへのポインタである。上記の例で引数付きのコンストラクタは、その引数 top にルール解釈メソッドの実行結果である集合オブジェクトを設定する。従って、構成を参照するインスタンス変数には、この値をそのまま代入することができる。

```
Airplane* aPlane = new (db) Airplane;
aPlane.component = top;
```

右辺が同じクラス名のルールが複数ある場合、特定のルールに対応する構成、すなわち特定の版の構成をプログラム中で扱いたい場合は、構成のエクステントを生成するときに、そのルール名 *rule-name* (図4) を指定する。

```
Body* top;
Od_Extent<B747*>  ext2 (top, "Body_V1");
```

この式では、ルール名 "Body_V1" のルールが具現化され top に構成へのポインタが設定される。

構成の生存期間は、エクステントの生存期間に一致する。すなわち、構成のエクステントが割り当てられる領域が自動変数であればブロック内、大域変数であればプログラム内の式を実行している間存在し、演算子 new(db) であれば明示して削除しない限り永続的に存在する。構成ビューのインスタンスは、エクステントが消滅するときに、データベースへチェックインされる。

構成を生成し他のオブジェクトと関連付ける方法は、連係オブジェクトによっても可能である。この連係オブジェクトを利用して構成を参照する可能性のあるインスタンス変数は、図5に示すように構成参照属性 config_ref を宣言する。図では、イン

```
class Airplane : Od_Pobject {
 public:
   char     type[10];
   Company* owner;
   Company* manufacturer;
   Person*  attendants;
   Body*    component  config_ref;
};
```

図 5: 構成の参照宣言

スタンス変数 component は構成参照属性を持つ。この属性を持つ変数は、宣言されている型 (Body*) だけでなく、連係クラス Assoc<T> と呼ぶ総称クラス (parameterized class) のインスタンスへのポインタも保持することができる。ここで、T は構成ビュー・クラス定義で指定されたクラス属性 config⟨T⟩ の T、すなわち構成を形成する最上位オブジェクトの型である。

連係オブジェクトと構成ビュー・クラスは、連係クラスのコンストラクタで関連をとる。すなわち、連係オブジェクトを生成したときに構成ビュー・クラスとの関連がとられる。一方、構成を参照する側 (図5のクラス "Airplane") には、単に連係オブジェクトへのポインタを代入するだけでよい。この代入が可能なのは、次節で述べるように連係クラス Assoc<T> が T のサブクラスとして実現されているためである。図6は、構成の生成と参照を含むプログラムである。行 (4)(5) は新しい連係オブジェ

```
(1) Airplane* anAirplane;
(2) Od_Type* b747;
(3) db.find_class (b747, "B747");
(4) anAirplane->component =
(5)     new (db) Assoc<Body>(b747);
(6) Cabin* aCabin =
(7)     anAirplane->component->cabin;
(8) Set<Seat*>* seat_set =
(9)     anAirplane->component->cabin
(10)        ->seats;
```

図 6: 構成の代入と参照

クトを生成し、飛行機オブジェクトを構成ビュー・クラス "B747" と関連付けている。(コンストラクタの引数には行 (3) で得た構成ビュー・クラスへのポインタを渡す。) この時点で連係オブジェクトは未定義状態であり、構成ビューはまだ具現化されていない (図 7(a))。行 (6)(7) は飛行機の機体の客室

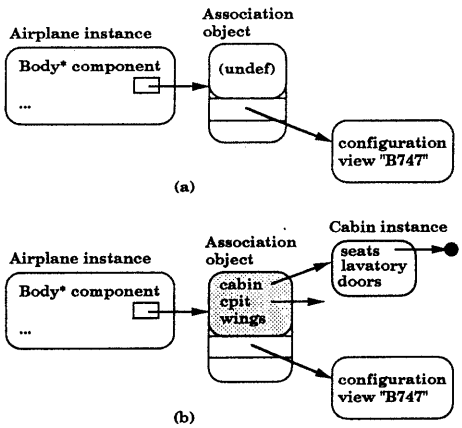

図 7: 連係オブジェクトと構成ビュー

オブジェクトを参照している。この客室は構成の一部であるので、このオブジェクトを取り出すためには構成を具現化する必要がある。このように構成の具現化は、構成の連係がとられた後、最初の構成要素の参照時に行われる。この後、インスタンス変数 "compoment" は実際の値の入った要素部品を指すことになる (図 7(b))。従って、この後の行 (8) ～ (10) では、具現化は行われず直接に客室の座席オブジェクトを取り出すことができる。行 (6) ～ (10) のように、構成の要素オブジェクトは、通常のオブジェクトとまったく同じ方法でポインタ参照することができる。これは、連係オブジェクトの型をそれを保持 (参照) するインスタンス変数の型の部分型とすることによって実現される。

4 実現

構成は、構成ビュー・クラスと連係オブジェクトによって実現されることを 3節で述べた。構成ビューに定義されているルールは、トランスレータによってルール解釈 (クラス) メソッドに翻訳される。ルール解釈メソッドは、構成ビューを具現化するときに起動される。構成ビューの具現化する方法には、2 つの方法がある。1 つは構成ビューのエクステントを得ることによる方法であり、もう 1 つは構成を連係オブジェクトによって関連付け、構成要素の最初の参照時に具現化する方法である。本節では、後者の方法で使われる連係クラスの構造と役割と、トンラスレータが構成要素を参照する式を展開する方法について述べる。

連係クラスは、引数として構成の最上位要素の型をとる総称クラス (parameterized class) である。図 8に連係クラスの定義の一部を示す。連係クラスは、引数 T のサブクラスとして実現される。従って、型 T のポインタ変数によって、T 型のインスタンスと Assoc<T> のインスタンスの両方を参照可能となる。連係クラスは、スーパクラス T のイン

```
template <class T> class Assoc
    : Od_Pobject,
      public  T  // T is a superclass.
{
 public:
    persistent char*  name;
    Od_Type*   p_view;
    int        mode;
    T* freeze();
    Assoc<T>* make_config();
    Assoc (Od_Type*, int=OD_STATIC);
};
```

図 8: 連係クラスの定義

スタンス変数やメソッドを持ち、かつ下記のような自分自身の性質を持つ。連係クラスは 3 つのインスタンス変数 name, p_view, mode を持つ。name は構成の名前である。この型 persistent name* は、永続文字列へのポインタを意味する。P_view は構成ビュー・クラス・オブジェクトへのポインタ

を保持する。クラス Od_Type はメタクラスであり、インスタンスはクラス・オブジェクトである (ODMG の Type に相当する)。mode は構成の具現化方法がが静的モードか動的モードかを示す。メソッドは3つ存在する。メソッド Assoc() は連係クラスのコンストラクタである。コンストラクタ Assoc() は、連係インスタンスが生成されたときに呼び出され、第1引数で p_view を、第2引数で mode を初期化する (既定値は静的モード)。メソッド make_config() は構成を具現化し、構成の最上位オブジェクトを連係オブジェクトにマップする。 このメソッドの利用方法については、構成の参照で展開されるコードの説明で述べる。メソッド freeze() は連係オブジェクトをそのスーパクラスのオブジェクトへ変換する。この操作により、構成は通常の複合オブジェクトの一部として組み込まれ、構成としての役割がなくなる。この逆操作 unfreeze() は、通常の複合オブジェクトを構成としての意味を持たせるメソッドであり、引数として構成ビュー・クラスを渡す。このメソッドは Od_Pobject に定義されている。

連係オブジェクトに関連づけられている構成を具現化し、その要素を参照できるようにするために、トランスレータは config_ref 属性付きインスタンス変数宣言、およびその変数を参照している式において、特別なコードを展開する。 config_ref 属性付きインスタンス変数に対しては、トランスレータは新しいインスタンス変数を展開する。

```
class Airplane : Od_Pobject {
 public:
    // ...
    Body*  component;
    int    component_flag;
};
```

このインスタンス変数は、構成の4つの状態 - 非具現化状態、具現化状態、動的構成状態、特定化状態を管理する。非具現化状態 (OD_NOT_INSTANTIATED) は、構成と連係している連係オブジェクトが未定義であることを示す。具現化状態 (OD_INSTANTIATED) は構成がすでに具現化され、連係オブジェクトに構成の最上位オブジェクトがマップされていることを示す。 動的構成状態 (OD_DYNAMIC) は、構成の要素参照の度に構成が具現化されることを意味する。特定化状態 (OD_SPECIFIC) は、構成オブジェクトではなく特定の元のインスタンスを保持していること示す。 freeze() を適用することによって、構成は特定化状態となる。

一方、参照の展開形は連係オブジェクトが未定義の場合、構成を具現化するコードが必要となる。図9は、図6の式 (6)(7) の展開形である。構成がすでに具現化されているか特定化されているとき、参照した変数値をそのまま返す (行 (6))。非具現化状態または動的構成状態の場合は、 make_config() で構成を具現化する (行 (11)(14))。ただし、静的構成の場合 (行 (7) ～ (11)) は、この後、具現化状態になる。

```
(1) aCabin =
(2)  (component_flag
(3)          == OD_INSTANTIATED ||
(4)   component_flag
(5)          == OD_SPECIFIC) ?
(6)      anAirplane->component->cabin
(7)  :(component_flag
(8)          == OD_NOT_INSTANTIATED) ?
(9)    (component_flag = OD_INSTANTIATED,
(10)    anAirplane->component->
(11)              make_config()->cabin)
(12) :(component_flag == OD_DYNAMIC) ?
(13)    anAirplane->component->
(14)              make_config()->cabin
(13) :/* エラー処理 */ ;
```

図 9: 構成の参照の展開形

5 おわりに

本論文では、オブジェクト指向データベース・システムに構成管理を統合するための1つの方式を提案した。提案方式の特徴は、構成をビューとして実現している点である。構成ビューの生成条件式は、そのクラス定義にルールの形式で記述する。構成ビューの具現化は、構成ビューのエクステントを得るか、連係オブジェクトに関連づけた後の最初の参照により行なわれる。この具現化時に、構成の要素オブジェクトは作業空間にチェックアウトされる。ルールは、具現化時にデータベース中に存在するクラスのスキーマと照合され、一致するスキーマの版が存在する場合、対応するアクションが実行される。この機能により、クラスのスキーマが進化した場合でも、それを参照しているプログラムへの影響を軽減することができる。これらは C++ の拡張した構文で記述する。

構成とそれを参照するオブジェクトとを関連付

ける方法として、連係オブジェクトを介する方法を提案した。連係オブジェクトは、その生成時に構成ビューと関連付けられ、具現化時に構成の最上位オブジェクトが連係オブジェクトにマップされる。連係オブジェクトは、通常のオブジェクト参照と同じ式で区別なく参照できる。これを可能にするために、連係クラスは構成の最上位クラスを引数とし、そのサブクラスとして実現される総称クラスとした。

本論文のルール記述により、構成を具現化することに関して、スキーマが進化することによるプログラムへの影響を減らすことができる。しかし、プログラム内で複数の構成の版を同時に扱う場合、どの版を今扱っているのかを実行時に調べる手続きを書く必要がある。通常の C++ の枠組では、任意に構造が変わり得る変数を定義し、それを参照する式を書くことができない。今後、これらの問題点を含めて、構成を扱うプログラムのスキーマ進化からの独立性をさらに向上させるための方法や、構成の要素オブジェクトに対して、メソッドを伝搬する方式を考察していきたい。また、提案方式を実際に Odin に実装することも今後の課題である。

参考文献

[Atki89] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D. and Zdonik, S. "The Object-Oriented Database System Manifesto," *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases*, 1989, pp.40-57.

[Aho86] Aho, A. V., Sethi, R. and Ullman, J. D. "Compilers - Principles, Techiniques, and Tools," Addison-Wesley Publishing Company, 1986.

[Catt93] Cattell, R. G. G. "The Object Database Standard: ODMG-93," Morgan Kaufmann, 1993.

[Ditt88] Dittrich, K. R. and Lorie, R. A. "Version Support for Engineering Database Systems," *IEEE Trans. on Soft. Engineering* Vol.14, No.4, Apr. 1988, pp.429-437.

[Elli90] Ellis, M. A. and Stroustrup,B. "The Annotated C++ Reference Manual," Addison-Wesley, 1990.

[Katz87] Katz, R. H., Bhateja, R. Chang, E. E-L. Gedye, D. and Trijanto, V. "Design Version Management," *IEEE Design & Test*, Vol.12, No.1, Feb. 1987, pp.375-408.

[Katz90] Katz, R. H. "Toward a Unified Framework for Version Modeling in Engineering Databases," *ACM Computing Surveys*, Vol.22, No.4, Dec. 1990, pp.375-408.

[Kim89] Kim, W., Bertino, E. and Garza, J. F. "Composite Objects Revisited," *Proc. 1989 ACM SIGMOD Int. Conf.on Management of Data*, May-June 1989, pp.337-347.

[Kimu91a] Kimura, Y. and Tsuruoka, K. "A View Class Mechanism for Object-Oriented Database Systems," *Proc. Int. Conf. on Database Systems for Advanced Applications (DASFAA'91)*, Information Processing Society of Japan, April 1991, pp.269-273.

[Kimu91b] Kimura, Y. and Tsuruoka, K. "A Language and View for Odin, an Object-Oriented Database System," *Proc. IEEE Pacific RIM Conf. on Communications, Computers and Signal Processing*, May 1991, pp.284-287.

[Kimu92] 木村裕、鶴岡邦敏、波内みさ "Odin データベースシステムのアーキテクチャと言語," アドバンスト・データベースシステム・シンポジウム, Dec. 1992, pp.63-72.

[Land86] Landis, G. S. "Design Evolution and History in an Object-Oriented CAD/CAM Database," *Proc. 13th COMPCON Conf.*, Mar. 1986, pp.297-303.

[ODI91] Object Design, Inc. "ObjectStore User Guide, Release 2.0" *Object Design, Inc.*, 1991.

[Ohbo88] 大保信夫 "CAD データベースの動向," アドバンストデータベースシステムシンポジウム, Dec. 1988, pp.73-82.

[Tale93] Talens, G., Oussalah, C. and Colinas, M. F. "Versions of Simple and Composite Objects," *Proc. 19th VLDB Conf.*, Aug. 1993, pp.62-72.

[Tsur92] Tsuruoka, K., Kimura, Y. and Namiuchi, M. "Architecture of an Object-Oriented Database Management System - Odin," *NEC Research & Development*, Vol.33, No.4, Oct. 1992, pp.669-678.

[Ullm88] Ullman, J. D. "Principles of Database and Knowledge-Base Systems Volume I," Computer Science Press, 1988.