データベース・システム　79－2
プログラミング言語　26－2
（1990．　9．　7）

ＣＯＢにおける持続性オブジェクトの設計と実現

三ッ井欽一

日本アイビーエム株式会社　東京基礎研究所

多くのアプリケーションプログラムでは、何らかの形でデータを保存する必要がある。例えば、設計支援環境の開発などには、複雑なデータ構造が必要になり、従来のファイルやデータベース機能では不十分である。我々はこのような要求に答えるためにオブジェクト指向プログラミング言語COBをベースにした持続性プログラミング言語PCOBを設計し、実験システムを作成している。持続性プログラミング言語では、通常のプログラミング言語の機能に加えて、一貫性制御、問合せ機能、ビュー機能などが解決すべき問題となる。本報告では、これらの問題に対する解決策を含めて現在のPCOBの仕様と実現上の工夫について述べている。

Design and Implementation of Persistent Objects in COB

Kin-ichi Mitsui
Tokyo Research Laboratory, IBM Japan

There are many application programs which require complex data structures to be stored in a secondary storage. Design support environments, for example, belong to this category. However, conventional file systems or database management systems are insufficient to deal with complex data. To meet this requirement, we are designing and implementing the persistent programming language PCOB, which is based on the object-oriented programming language COB. In designing persistent languages, issues such as integrity control, query processing and view functions arise. This paper reports current status of PCOB design and implementation, and our approaches to these issues.

1. はじめに

従来、データを保存し、複数のアプリケーションプログラムでそれらを共有する必要がある場合には、データをファイルとして保存するか、データベース管理システムとリンクするか、どちらかであった。前者の場合、データ構造が非常に単純なものに限られるので、プログラム中で複雑なデータ構造を扱う場合に両者の対応をアプリケーションの責任で管理する必要がある。また、共有管理や並行制御は一般には不十分である。後者の場合も、データベースがサポートしているデータ構造とプログラム言語がサポートしているデータ構造との間にギャップがあるために同様の問題を生じる。これはアプリケーションプログラムの生産性の低下を招く。この問題は、インピーダンススマッチの問題としてよく指摘されている。この問題に対して、プログラミング言語と持続性機能の親和性をよくしよう、あるいは、完全に融合させてしまおうという試みが、最近特に活発に議論されるようになってきた。

多くのアプリケーションプログラムでは、何らか形でデータを保存する必要がある。アプリケーションプログラムがそれほど複雑なデータ構造を必要としないような問題領域では、上の問題はそれほど深刻ではないが、例えば、プログラミング環境、ＣＡＳＥ環境、設計支援環境などでは、複雑なデータ構造が必要になり、アプリケーションプログラム開発を容易にするためのなんらかの方策が必要である。また、このような問題領域では、従来のデータベースが得意とした、大量で均一なデータを効率的に扱う必要性よりも、複雑なデータをモデル化でき、それを非常に高速に扱う必要性が重要視されるといった、従来のデータベースとは異なった要求が生じる。

我々は、このような要求に答えるために、汎用なオブジェクト指向プログラミング言語COBのクラス定義機能をベースにした持続性プログラミング言語PCOBを設計している。また、実験システムを実現しつつあり、プログラミング環境の構築に応用を試みている。

現在までに、持続性プログラミング言語に関して幾つかの実験システムが発表されている。インピーダンスミスマッチの問題も、それぞれの言語においてある程度解決されているように思われる。しかし、持続性機能が本質的に問題となるのは、大量のデータを扱い得ること、オブジェクトの寿命が長く、多くのアプリケーションや多くのプログラマによって共有され得ることである。このために従来にプログラミング言語では、あまり考えられていない問題点が生じる。例えば、問合せ表現、ビュー機能、スキーマ進化・バージョン管理の問題などである。我々は、COBという言語をベースにして、このような問題を解決すべく持続性機能の検討を続けている。

本報告では、現在までに実験的に作成されているPCOBの持続性機能の説明、更に検討段階ではあるが、上記の問題点に対する幾つかの提案を行なっている。以降、２章では既に報告されている関連する研究について、３章では、PCOBの言語仕様について、４章ではその実現に関して、５章ではPCOBの現状について述べる。最後に６章でまとめる。

2　関連する研究・アプローチ

オブジェクト指向概念とデータベース機能とを融合させようという試みは、大きく分けて二つの流れがあるように見える。一つは、プログラミング言語から出発するもので、SmalltalkをベースにしたGemStone[3]のOPAL言語、C++をベースにしたO++[1]、E[13]、CLOSをベースにしたPCLOS[12]、CLUに影響を受けているTrellis/Owl[10]などがある。一方、データベースとして出発しているものとしては、ORION[7]、O2[6]、Vbase[2]、Iris[5]などがあげられる。もちろん、二つの流れは非常に近いものであり、それぞれのシステムの分類はそれほど明確なものではない。

プログラミング言語におけるデータオブジェクトに持続性を持たせたものを持続性プログラミング言語（persistent programming language)とよぶ。PS-Algol[4]は、オブジェクト指向言語とは直接関係ないが、言語と持続性機能の融合の重要性を指摘し、特に言語の型システムと持続性機能の直交性を指摘して注目された。その後、オブジェクト指向言語の進歩に伴い、数々の言語に対して持続性機能を融合する試みが行なわれている。また、最近では、Ｃをベースにしたもの、あるいは強い型付けを行なうオブジェクト指向言語が、高い効率、実用性の意味から注目されている。C++やCOBはこの範疇に属する。

Eは、C++をベースにして応用に応じて拡張されるデータベース管理システムを容易に作成するための持続性言語として設計された。このため一般の応用プログラムの開発を容易にするような問合せの機能などの高度な機能よりも、効率のよい持続性オブジェクトの管理に重点が置かれている。

O++は、C++をベースにしてデータベース機能を必要とする応用プログラムを容易に開発するために設計された持続性言語である。集合指向の問合せ記述、バージョン管理、制約記述、トリガ機能などが盛り込まれている。

我々の考えでは、CやC++では、システム記述に向くようにポインタ演算などにより裸の機械が見えるようになっており、従って、持続性機能を考える上でも、オブジェクトの共有管理、並行制御、ガーベージコレクション

などを難しくしている。また、言語における安全性も持続性オブジェクトについてはより重視されるべきである。汎用的な持続性プログラミング言語のベースとしては、やや制限のつよい高級言語が望ましいと考えられる。COBはこのような要件を満たしている。

これらの流れとは別に、データベースの観点からオブジェクト指向概念を取り入れる試みがある。これらは従来のデータベースが扱える比較的単純なモデル化機能をより高度なものにし、またデータに手続き的な機能を付加させようとするものである。こちらのアプローチでは、言語との融合性よりも、従来データベース分野で議論されてきた、一貫性制御の記述、意味記述を含めた高いモデル化能力、問合せ言語、ビュー機能、スキーマ進化などが注目される。

ORIONは、Common Lisp上に作成されたオブジェクト指向データベースである。この研究の中で、クラス階層や複合オブジェクト(composite object)に基づいたトランザクション管理やスキーマ進化に対する議論が展開されている。

O2では、問合せの議論の中でオブジェクトと値とを明確に区別している。これは、問合せの意味を考える上で注目に値する。また、O2は多言語環境になっている。

IRISも、多言語環境を考えている。特に簡単な対話的言語を通してデータベースにアクセスする機能は、計算能力に制限が加えられたとしても、例えばシステムのコンフィギュレーションやユーザのカスタマイズ機能を実現する上で今後重要になると考えられる。

このように持続性機能といっても実に様々な観点から活発に研究されている。これらのシステムには既に商品化されたものもあり、その意味では実用化に一歩近づいたと言えるが、例えば、問合せ表現と情報隠ぺいとの親和性、ビューの問題、スキーマ進化の問題などはまだまだ議論の余地を残しているし、そのような機能を統一的に議論できるような意味論的な整備はこれからの課題といえる。このような観点から今後も研究がされていくと思われる。

3. PCOBの設計

我々は既に[9]においてPCOBの初期の設計について報告している。その後検討を続けており、幾つか変更を加えた点や新しく議論をしている点がある。ここでは、それらを含めた議論を行なう。ここで議論する仕様の幾つかは実際に実現され実験的に用いられている。また、幾つかの仕様については、未だ実現されず検討の段階であるが、それらも持続性言語の設計を議論する上で意味のあることであると考えられるので本報告でも取り上げる。

PCOBのベースであるオブジェクト指向プログラム言語COB[11]は、Cの上位互換でオブジェクト指向機能を付加した言語で、強い型付けを行なう、インタフェースと内部実現を完全に区別する、GCを行なう、などの特徴をもつ。詳しい仕様については[11]を参照されたい。

3. 1 クラス

COBのプログラミングの基本となるのはクラス定義である。クラス定義に基づきそのインスタンス（オブジェクト）が生成され実際の計算をおこなう。PCOBでは、このインスタンスに持続性を持たせ、プログラム終了後もその状態を保存することができる。

持続性を管理するためには実行時のオーバーヘッドをともなう。PCOBでは、効率上の配慮から、このようなオーバーヘッドを持たない従来のCOBのクラスと、持続性を持たせることができるクラスとを共存させることにした。後者を区別するために持続性クラスと呼ぶことにする。但し、持続性クラスの全てのインスタンスが持続性を持つわけではない。従来のCOBのクラスを一過性クラスとよぶ。

持続性クラス定義の構文は、先頭に"persistent"と書くこと、メンバーに対する修飾子が追加されていること、を除けばCOBのそれと全く同じである。PCOBの持続性クラス定義の例を挙げておく。

```
persistent class Emp {  /* interface */
  ...
  int getAge(void);
};
persistent class impl Emp { /* implementation */
  char name[20];
  int age;
  ...
```

```
definition:
  ...
  int getAge(void) { return(age); }
};
```

3．1．1　メンバー

クラスは、メンバーの集まりとして定義される。メンバーにはインスタンスメンバーとコモンメンバーとがある。更に、それぞれについてメンバー変数とメンバー関数（メソッド）とがある。更に、この４種類のメンバーに対して、パブリックなものとプライベートなものとがある。これらの意味については[11]を参照してもらいたい。

持続性クラスにおいては、各オブジェクトに対して（パブリック、プライベートを問わず）インスタンス変数の値をデータベース中に保存する。但し、アドレスを値として持つようなポインタ変数についてはその値は保存しない。（COBのクラス型変数は、オブジェクトへの参照を表しているが、ポインタ変数とは考えない。）また、共用型についても値は保存しない。共用型を使用するよりは、同じことをクラスで表現できるので、そのほうが望ましい。また、一過性持続性クラス型の変数についても値を保存しない。

また、修飾子として"transient"を指定されたメンバー変数については値を保存しない。オブジェクトがデータベースから再生された時点での"transient"なメンバー変数の値は、変数宣言に初期値が与えられている場合にはその初期値になる。また、持続性クラスのメソッド"when_restored"を定義しておくと、これがオブジェクトの再生時に自動的に呼び出され、変数の初期化などの操作を記述することができる。

3．1．2　継承

持続性クラスの上位クラスは、持続性クラスであってもよいし、従来のCOBのクラスであってもよい。後者の場合、上位クラスの部分の内部状態は保存されない。上位クラスが持続性クラスである場合は、その下位クラスは持続性クラスでなければならない。持続性という性質が継承されると考える。

3．2　名前空間

通常COBでは、トップレベルの大域変数を用いて異なるコンパイル単位モジュール間で変数を共有する。持続性オブジェクトに関しては、より大きなスコープを持つDB変数を導入して、異なるプログラム間で変数を共有する。DB変数の宣言の前には"persistent"と指定する。DB変数はオブジェクトへの参照を表す（オブジェクトそのものではない）。constと指定されたDB変数へは、代入ができない代わりに、別々のプロセスが同時にその変数を用いることができる。DB変数の宣言の例を挙げておく。

```
persistent Employee e1;
const persistent Employee e2;
```

3．3　インスタンス

持続性クラスのインスタンスは、持続性をもちデータベース中に管理されるものと一過性でプログラムの終了と共に消えてしまうものとがある。どちらもコモンメソッドnewを用いて生成される。持続性クラスのインスタンスが実際に持続性を持つかどうかは、あるデータベース中で宣言された全てのDB変数をルートとしインスタンス変数の参照をたどったとき、そのオブジェクトまでの参照パスが存在するかどうかで決まる。当然、持続性は動的に決まり、持続性をもったオブジェクトが、別のプログラムの実行後に一過性オブジェクトに変化する場合もある。

ある持続性オブジェクトが一過性オブジェクトに変化した場合、そのオブジェクトの２次記憶上の記憶域は自動的に回収される。すなわち、ガーベージコレクションが行なわれる。

幾つかの持続性言語では、明示的なdeleteメッセージによりオブジェクトを削除している。しかし、明示的な削除はバグを誘発する可能性が高い。持続性言語では、オブジェクトが複数のアプリケーションプログラムで共有されるため、デバッグのために考慮する必要のある範囲が広くなり、いっそうデバッグを困難にするであろう。持続性言語では、特に、明示的な削除は避けるべきであると考える。

3．4　トランザクション

従来データベースの分野では、データベース中のデータの一貫性を保つために、幾つかのデータ操作を一まとめにしたトランザクションという単位を考えた。これは不可分な原子的操作（atomic operation)であり、障害があったとしてもトランザクションの途中の状態が正しい状態として参照されることがないことが保証される。また、トランザクションの並列性を考える上では、データベースの最終状態がトランザクションを順次実行した場合と同じになるという条件（serializability)がよく用いられている。

PCOBにおいてもトランザクションを用いる。トランザクションは一貫した手続きの集まりであるので、手続きを構成する言語上の構文に合わせたほうが良いと考えた。トランザクションを指定するためには、関数定義またはクラス定義中のメソッド宣言の前に修飾子"atomic"を付ける。

atomicな関数またはメソッドが実行される直前にトランザクションの初期化が自動的に行なわれる。この関数またはメソッドが正常に終了した場合は、トランザクション中で更新された持続性オブジェクトがデータベース中に反映される。トランザクションの実行中にrollback関数を呼び出すとトランザクションはアボートし、制御は関数またはメソッドを呼び出した時点に戻る。

並行制御は、オブジェクトを施錠の単位とした２相施錠方式を用いている。（但し、施錠の単位は実際には効率を考慮してもう少し大きな単位で行なっている。）施錠および解錠は基本的には暗黙に行なわれる。トランザクション中で初めてオブジェクトが内部状態を読み込む必要が生じるまえに施錠を試みる。また、トランザクションがコミットした時点で解錠する。デッドロックが起きた場合は、後から施錠要求を出したトランザクションをアボートする。

３．５　オブジェクトのグループ化

オブジェクト指向によるデータのモデリングにおいて複合（composite)オブジェクトのようにオブジェクトをグループ化する機能が議論されている[8]。PCOBでも次の二つの意味からこの概念を言語に導入している。一つは、効率的に施錠を管理するための階層的施錠を行なうためである。もう一つは、２次記憶中の物理的配置を決めるヒントを与えるためである。グループ化されたオブジェクト全体をグループオブジェクトとよぶ。グループオブジェクトは、一つの主オブジェクトをもつ。各オブジェクトは唯一の主オブジェクトを親として持つことができる。主オブジェクトも更に親オブジェクトをもつ。この結果、グループオブジェクトは木構造になる。この木構造に対して階層的施錠ができる。すなわち、ある主オブジェクトに施錠した場合、その子オブジェクト全体が施錠されていると考える。また、グループオブジェクトに含まれるオブジェクトは、できるかぎり互いに物理的に近い場所に保存される。

３．６　一貫性保持、トリガ

O++のみならず、幾つかのデータベース言語では、データの一貫性保持のために制約条件の記述を行なうことができる。また、トランザクションの結果に応じて、新しく別のトランザクションを自動的に起動するといったトリガ機能などがある。PCOBでは、いかにこれらの機能が記述できるかを述べる。

まず、一貫性保持について述べる。一般には、いろいろなタイミングで制約条件を記述する必要がある。例えば、インスタンス変数がアクセスされたとき、メソッドが終了したとき、等である。これらは、対応するメソッド中にその検査を行なうコードを埋め込んでおけば良い。しかし、特に、トランザクションがコミットする直前（すなわち、データベースが更新される直前）に制約条件を検査する意義は大きいと考えられるが、このタイミングで全ての更新されたオブジェクトをアプリケーション側でたどる負担は大きい。PCOBでは、更新されたオブジェクトに対してトランザクションがコミットする直前に一回だけwhen_savedメッセージが自動的に送られる。対応するメソッド中にそのオブジェクトに関する制約条件の検査をするコードを記述しておけばよい。条件を満たさない場合は、この場所からトランザクションをアボートすることができる。

但し、若干の注意が必要である。when_savedメソッド中には任意の手続きを書くことができる。従って、既にwhen_savedメッセージが送られて制約条件の検査を行なったオブジェクトに別のメッセージを送ってその内部状態を変えてしまうかもしれない。このようなことが起きないように、一度when_savedメソッドを実行したオブジェクトは、読み込み専用オブジェクトに変化して以降トランザクションがコミットするまでその内部状態が変化しないことを保証する。読み込み専用オブジェクトに更新操作を行なおうとした場合はトランザクションがアボートする。

次にトリガについて述べる。トリガについて問題となるのは、オブジェクトの内部状態に応じて別のトランザクションを起動させるとき、これがトリガ条件を検査した時点よりもずっと後であることである。すなわち、トリガ条件が発火したとしてもトランザクションが結局アボートしたなら対応するアクションは実行すべきではな

い。ここでは、このトリガの仕組みが特別の機構を用いなくても記述できることを示す。

下の例では、ある供給者(Supplier)がトランザクションの最後で在庫がある値以下になっていた場合に、トリガ条件が発火し、これを補充する作業を定義したオブジェクト(CallDealer)を生成し、これを集合オブジェクト(TriggerActions)に加えている。実際のトリガアクションの実行は、後のトランザクション中でTriggerActionsから要素を取り出してexecメッセージを送る。

```
persistent class TriggerAction {
  void exec(void) deferred;
};
persistent class CallDealer < TriggerAction {
  void exec(void); /* trigger action */
};
persistent class PSet TriggerActions; /* set of trigger action */
persistent class impl Supplier { /* implementation of supplier */
  ...
  void when_saved(void) {
    if (self->stock_qty() < LOWER_LIMIT) { /* trigger condition */
       TriggerActions->add(new@CallDealer());
    }
  }};
```

一貫性保持やトリガ機能をO++のように特別な言語機能とする場合、記述量が少ない、読み易い、最適化の可能性、などの利点が考えられるが、逆に制限を多く付ける必要があったり、様々な使われ方を考えるとその意味付けが複雑になったりして結果的に言語全体を複雑でわかりにくいものにしてしまうおそれがある。PCOBでは、ほとんど特別な言語機能を導入しなくても一貫性保持やトリガ機能をそれほどきたなくなく記述できることを示した。

3．7　その他の機能

ここでは、問合せの記述とビューの機能について述べる。これらの機能については、その設計は十分結論を出す段階ではなく、また実験システムにおいてもまだ実現されていないが、持続性プログラミング言語を設計する上で非常に重要な観点であり、我々も検討を続けているのでここで議論し、提案をしてみたいと思う。

3．7．1　問合せ記述

幾つかの持続性言語では、集合指向の問合せを行なうことができる。２次記憶に保存されたたくさんの持続性オブジェクトから目的のものを取り出すことを考えると、やはり特別な言語構造や最適化の支援がほしくなる。従来のデータベースシステムの問合せ技術に比べて、オブジェクト指向システムの問合せの決定的な違いは情報隠ぺいにどのように対処すべきかということであろう。いくつかのシステムでは情報隠ぺいは破られている。

問合せ記述について我々は、[9]で一方法を提案した。これは、情報隠ぺいを問合せの記述においても厳密に守るために、問合せを予め検索対象となるオブジェクトのメソッドとして用意しておくものであった。この方法に対する主な批判は、予め与えられたメソッドの簡単な組合せでしか検索条件が記述できず、問合せ表現の柔軟性が低下するということであった。

しかし、情報隠ぺいはオブジェクト指向システムの保守性・拡張性を考える上で非常に大切な概念であるので、問合せ表現においても守るべきであると考えている。ここでは[9]とは別の方式を提案する。

一般に問合せは、検索対象となるオブジェクトから検索結果のオブジェクトへのマッピングとして統一的に扱うことができる。ナビゲーショナルな検索は、あるオブジェクトにメッセージを送ってそのオブジェクトが知っている別のオブジェクトを得ることに相当する。集合指向(set-oriented)あるいは連想的(associative)な検索は、オブジェクトの集合オブジェクトに検索条件をメッセージとして送って部分集合オブジェクトを得ることに相当する。例えば、下の例ではEmpクラスの集合オブジェクトにメッセージselectを送ることによりその部分集合を求めている。ここで、[]内は検索条件をデータとして扱うために導入した構文である。束縛変数の宣言の部分と検索条件の部分とからなる。検索条件データの型は、パラメタ付きクラスdbconditionで、下の例では、dbconditi

on[Emp]型になる。オブジェクトの集合を表すEmpsの型は、パラメタ付きクラスdbcollection型で、実際にはdbcollection[Emp]である。

```
YoungEmps=Emps->select([(Emp)x|x->age() > 20 && x->age() < 30]);
```

検索条件中においても情報隠ぺいのルールが適用され、オブジェクトへのアクセスには公開されているインタフェースを用いなければならない。この場合に問題になるのは、任意のメソッドを検索条件に用いてもよいかどうかである。

例えば、メソッドが副作用をもち、メソッドの実行ごとに異なる値を返す可能性のあるものは問合せには不適当である。また、メソッドの実行では同じ値を返すが副作用がある場合も、実際にメソッドを実行して検索する場合とインデックスを使って検索する場合の意味が異なってしまうのでやはり適当でない。検索に使えるメソッドは関数的で副作用のないものでなければならない。

ここでいう副作用とはメソッドを実行するオブジェクトの内部状態のみならずメソッドの実行に関係する全てのオブジェクトの内部状態に対してである。あるメソッドがそれを実行しているオブジェクトに対して副作用を持つかどうかは、メソッド中に内部状態を更新する操作があるかを調べればよいが、計算に用いる別のオブジェクトの副作用の有無はインタフェースをながめただけではわからない。

そこで、静的にこの検査を行えるようにインタフェース部に、対応するメソッドがそのオブジェクト内で副作用を持たないことを表示する修飾子constをつけることにする。あるメソッドの計算に関連する全てのオブジェクトのインタフェースに対して副作用がないことがわかればそのメソッドはデータベースに対して副作用を持たないことがわかる。このconstは、サブクラスでメソッドを再定義する場合にも制約条件として継承される。すなわち、const指定されたメソッドはサブクラスにおける再定義も含めて副作用を起こす操作を行なってはいけないし、constでない別のメソッドを呼び出してもいけない。このような制限を記述できるようにすれば、それらのメソッドが返す値に対してインデックスを作成し効率的な検索も行える。

但し、若干注意が必要である。インデックスは、オブジェクトにメッセージを送った結果について作成する。メソッドが、その実行をするオブジェクトの内部状態にのみ依存して計算される場合には、そのオブジェクトが更新されたトランザクションの最後にインデックスを更新すればよい。しかし、メソッドの計算が別のオブジェクトに依存する場合は、別のオブジェクトの内部状態が変化した場合にもとのオブジェクトのインデックスが正しくなくなる可能性があるので変更があったことを基のオブジェクトに伝搬してインデックスを更新する必要がある。このためには、計算の依存関係の逆方向の関係を管理していなければならない。

問合せに関してもう一つ問題になるのは、関係モデルでいう結合操作である。特に、新しい構造を持つような関連をデータベースから導きたいような場合である。例えば、趣味が同じ社員のペアを検索するような場合である。O++では、問合せのためのfor-each構文により結合演算が記述できるようにしている。しかし、この方法だと検索結果をオブジェクトとして統一的に扱うことはできない。

我々の方法では次のようようになる。例を用いる。まず、社員のペアに対するクラスEmpPairを新しく定義する。このクラスのインスタンスは、二人の社員を初期値として与えることにより作成される。社員のペアの集合オブジェクト(Pairs)は、dbcollection[EmpPair]型である。この集合オブジェクトは、生成時に２つの社員の集合オブジェクトを与えられた場合には、内部状態としてその２つの集合の直積集合をもつ。このようなnewに関するインタフェースはEmpPairのinitメソッドから自動的に作成される。Pairsに検索条件を与えて求める集合SameHobbyPairsが得られる。この方法だと、検索結果はオブジェクトであり、更に別の検索対象として用いることもできる。

```
class EmpPair { /* relation of two employees */
  void init(class Emp e1, class Emp e2);
};
...
 Pairs=new@dbcollection[EmpPair](Emp1, Emp2);
 SameHobbyPairs=Pairs->select([(Emp)x,(Emp)y|x->hobby()==y->hobby()]);
```

３．７．２　ビューに関して

データベースプログラミング言語が通常のプログラム言語と異なる大きな点の一つは、対象とするデータの寿

命が長いということである。このため既に稼働しているデータオブジェクトが将来の様々な使われ方に対処できるような柔軟性を持つように要求される。従来、関係データベースでは、ビューを用いることにより既に存在しているデータを後の使われ方に応じて加工することができた。ビューにより、ユーザが不用なインタフェースを隠す、インタフェース名を変更する、導出される属性を新しくインタフェースに加えるといったことができる。

ビューの機能に関する議論は、オブジェクト指向言語においても同様に展開できる。COBのような強い型付けを行なうオブジェクト指向言語では、このようなインタフェースの拡張に関する柔軟性はサブクラスの機能により得られる。

ここでは、サブクラスを利用してビューを定義する方法を提案する。具体例を用いて述べる。下の例では、社員クラス(Emp)が既に持続性クラスとしてデータベースに登録されており、社員オブジェクトが既にデータベース中に存在しているとする。あるプログラム中でこの社員オブジェクトに対して評価値(eval)メソッドを新しく追加する要求があったとする。評価値は既に存在する社員オブジェクトを用いて計算できるものとする。例では、必要なビューをEmpMyViewとして定義している。このプログラムの中では、データベースから再生される社員オブジェクトは実際にはEmpMyViewクラスのオブジェクトの機能を持たせたい。このことを指示するためにビュークラス定義を用いている。

```
persistent class Emp { ... };
view class EmpMyView < Emp {
  int eval(void);
};
```

データベースから再生された社員クラスオブジェクトはEmpMyViewクラスのインスタンスになり、そのなかのEmpクラスに相当する部分のデータがデータベースから読み込まれる。Empクラスとして参照されている持続性オブジェクトは、EmpMyViewクラスに下方変換することができ、その結果、新しいインタフェースであるevalを呼び出すことができる。このように基になる持続性オブジェクトに個々のプログラムに特有なインタフェースをビューとして加えることができる。

逆にインタフェースを隠したい場合には、現在のCOBの仕様ではうまく制限する方法はない。クラスインタフェースに対してなんらかの方法でマスクをかけるような仕組みが必要であろう。但し、あるインタフェースについて再定義をしてそのメソッドを無効化することは可能である。この場合の無効化は強力で、間接的にそのメソッドが呼び出されるのも禁止することができる。

これまでで述べたビューは、いわば特定のプログラム中のみで作成される一過的なものであった。ビューの考え方を更に進めると、ある持続性オブジェクトに対して、別の持続性の部分を拡張したいという要求も考えられる。例えば、公共なデータベース中のあるオブジェクトに対して、個人的な情報を拡張したいような場合である。この場合、公共な部分と個人的な部分は別々に管理されるべきである。このようなビューを持続性ビューと呼ぶことにする。上の例を持続性ビューに拡張すると次のようになる。

```
persistent class Emp { ... };
persistent view class EmpMyView in MyDatabase < Emp {
  int eval(void);
};
```

viewとして使われるサブクラスは持続性クラスとしてある個人用データベース(MyDatabase)に登録され、サブクラスに関するデータもその個人用データベースに保存される。後にこのビューを用いて参照される場合は、もとの持続性オブジェクトと個人データベース中のデータが結合されて一つのオブジェクトになる。

以上のようなビュー機能を用いることにより、データベースに保存するオブジェクトのクラスはできるだけ汎用に設計し、それを個々のプログラムにおいて自由に拡張して利用するといったことができるようになる。

4. PCOBの実現

4. 1 オブジェクトの記憶管理

いくつかの持続性プログラム言語やオブジェクト指向データベースでは、オブジェクトの記憶管理は、関係デ

ータベースシステムのように既に確立され実現されているシステムを用いている。(例えば、PCLOS、Trellis/Owl、IRISなど)このようなアプローチの最もおおきな利点は、データベース管理システムの実現という大変な作業の多くを省くことができるということである。

一方、同時にこれらの報告でよく取り上げられる問題点は、少なくとも現在利用可能なＲＤＢＭＳでは、十分な効率が得られないという点である。その理由の一つは、オブジェクト指向システムでは、オブジェクトの参照関係の追跡という関係データベースにとっては不利な操作が非常に多く行なわれるためである。また、関係データベースでは、タプル間の独立性が高く、逆に関連するタプルをまとめて物理的に近くに配置することは一般にはできない。これらが十分な効率が得られない理由であろう。別の問題点は、既存のシステムでは並行制御などの細かい制御ができないことである。

RDBMSの技術的な進歩に期待するという方向もありうるが、我々は実験システムの作成では、独自のオブジェクトの記憶管理を行なっている。

また、ガーベージコレクションに関しては方式を検討中であり現在までのところ実現はされていない。

４．２　持続性オブジェクトの再生と退避

持続性オブジェクトの内部状態は、トランザクション中でそれが必要になった時点で順次２次記憶から仮想記憶中に読み込まれる。このためにはオブジェクトのインスタンス変数がアクセスされる直前にオブジェクトが読み込まれていることを検査する必要がある。読み込まれていない場合は、施錠を行なった後に内部状態を読み込む。また、インスタンス変数の更新操作が行なわれる直前では、書き込み施錠がとられていることを検査する必要がある。書き込み施錠された持続性オブジェクトはトランザクションがコミットした時点でデータベースに書き込まれる。トランザクション終了時にはすべてオブジェクトを解錠する。

オブジェクトの検査のために実行時のオーバーヘッドが生じる。[9]においても議論したが、COBではメッセージとメソッドの動的束縛を実現するためにメソッドの呼び出しが間接的に行なわれる。あるオブジェクトが書き込み施錠されて仮想記憶中に読み込まれた後は、そのオブジェクトのインスタンス変数のアクセスに関しては検査を行なう必要はない。従って、このような検査を省略したメソッドを別に用意して動的にメソッドを付けかえれば実行のオーバーヘッドを減らすことが可能である。(但し、プログラムのサイズは大きくなる)実行効率が重要な状況ではこのような方式も考慮に値するだろう。

４．３　持続性オブジェクトのオブジェクト識別子

持続性オブジェクトは、一過性のオブジェクトと同様に仮想記憶中では(例えば３２ビットの)アドレス値を用いて参照されている。２次記憶中では、これとは別な表現になる。現在の実現では、オブジェクトが記憶されているファイル番号、ファイル内インデックスが参照情報として用いられている。また、型に関する情報が参照情報の一部として加えられている。更に分散環境に対応するためのサイト識別子やその他の情報が加えられる可能性もある。このように２次記憶上での参照情報は仮想記憶におけるアドレス値よりも長いビット長を要する。

仮想記憶中でオブジェクトの参照を解決する必要な生じたときには、２次記憶上での長い表現から仮想記憶中のアドレス値に変換をしなければならない。現在の実現ではこの変換を次のようなタイミングで行なう。今、あるオブジェクトが仮想記憶に再生されつつあるとする。このオブジェクトは、インスタンス変数として別のオブジェクトへの参照を持っているとする。この別のオブジェクトはまだ再生する必要はない。この時点で、ファイル番号などの情報を用いて参照先のオブジェクトが既に再生されているかを調べる。再生されていない場合は、参照情報の一部である型情報を用いて、ダミーのオブジェクトを仮想記憶上に作成し、インスタンス変数に代入するアドレス値はこのダミーを参照するようにする。ダミーオブジェクト中にファイル番号などの情報を書き込んでおく。後にダミーのオブジェクトを再生する必要が生じたときには、ダミーの中にオブジェクトの内容が読み込まれる。このようにすれば、仮想記憶中でオブジェクトを参照するインスタンス変数は通常の一過性のオブジェクトと同様にアドレス値のみを持てばよくアドレス値の付け替えを後の時点で行なう必要はない。

この方法は次のような場合に有利である。オブジェクトは、その内部状態が読み込まれていなくてもそのオブジェクトの参照に対して演算操作が行なわれることがある。例えば、別の変数への代入、等値性の判定、別の型の変数への代入(型変換)である。オブジェクトへの参照は必ず仮想記憶のアドレス値なっているので、これらの演算の度に持続性オブジェクトの参照の解決を行なう必要はない。

５．現在の状況

実験システムは、OS/2およびIBM AIXの環境上で稼働している。記憶管理は、それぞれのOSのファイルシステム

を用いて独自に行なっている。

OS/2拡張版では、SQLデータベース機能が利用できる。これも、OS/2のファイルシステムを用いており、動作環境はよく似ていると考えられるので、このSQLを用いた場合とPCOBを用いた場合とで性能の比較を行なった。但し、PCOBは実験システムの段階で、例えばアクセス制御などはしていない等データベース管理機能は不十分なので、単純な比較は注意が必要である。テストしたプログラムは、参照関係をたどるのが主な作業であるため、関係データベースには不利なものである。両者の反応時間を比べるとPCOBのほうが数十倍程度高速であった。使用した経験では、PCOBが十分高速であるというよりも、SQLを使った場合に性能がかなり問題になるであろう、という感じをもった。

6. まとめ

COBをベースにした持続性言語PCOBについて、現在までに検討された言語仕様について、また、実験システムを実現する際に考慮された主な工夫点について述べてきた。持続性言語は、通常のプログラミング言語としての面に加えていくつか解決すべき問題点がある。このうち、一貫性制御、問合せ機能、ビュー機能については本報告でも議論をしてきた。バージョン管理やスキーマ進化については、議論はしなかったが、同様に重要な問題であると考えている。また、このような持続性プログラミング言語システムでは、従来のプログラミング言語のようにプログラムテキストと処理システムとが独立ではだめで、すべてデータベースによって管理されるような統合的環境にする必要がある。今後そのような点についても考えていく必要がある。

参考文献

[1] Agrawal,R. and Gehani, N.H.: "ODE(Object Database and Environment): The Language and the Data Model", Proc. of the 1989 ACM SIGMOD International Conference, 1989.

[2] Andrews, T. and Harris, C.: "Combining Language and Database Advances in an Object-Oriented Development Environment", Proc. of OOPSLA'87, 1987.

[3] Bretl, R., Maier, D. and Otis, A.: "The GemStone Data Management System", in Kim, W. et. al.(eds.): Object-Oriented Concepts, Databases, and Applications, Addison Wesley, 1989.

[4] Cockshot, W.P., Atkinson, M.P. and Chisholm, K.J.: "Persistent Object Management System", Software-Practice and Experience, Vol. 14, 1984.

[5] Fishman, D.H. et. al:"Iris: An Object-Oriented Database Management System", ACM Transaction on Office Information Systems, Vol. 5, No. 1, 1987.

[6] Lecluse, C. and Richard, P.: "The O2 Database Programming Language", Proc. of VLDB'89, 1989.

[7] Kim, W., Ballow, N., Chou, H.T., Garza, J.F. and Woelk, D.: "Features of the ORION Object-Oriented Database", in Kim, W. et. al.(eds.):Object-Oriented Concepts, Databases, and Applications, Addison Wesley, 1989.

[8] Kim, W. et. al.: "Composite Object Support in an Object-Oriented Database System", Proc. of OOPSLA'87, 1987.

[9]三ッ井欽一、上村務: "COBにおける持続性機能の導入",WOOC'90,1990.

[10] O'Brien, P., Bullis, B. and Schaffert, C.:"Persistence and Shared Objects in Trellis/Owl", DEC TR 440, 1986.

[11] Onodera, T., Kuse, K. and Kamimura, T.:"Increasing Safety and Modularity of C Based Objects", IBM TRL TR RT0042, 1990.

[12] Paepcke, A.: "PCLOS: A Flexible Implementation of CLOS Persistence", In S. Gjessing and K. Nygard, eds., Proc of ECOOP, Lecture Notes in Computer Science, Springer, 1988.

[13] Richardson, J.E. and Carey, M.J:"Persistence in the E Language: Issues and Implementation", Software-Practice and Experience, Vol. 19, No. 12, 1989.