データベースシステム　100－8
（１９９４．１０．１４）

動的制約分析による OODB 設計法

鬼塚 真,　山室 雅司

NTT 情報通信研究所

ＣＡＤやＣＡＳＥ等がサポートする設計業務では、設計データの進捗度に応じて実行する処理が変化する。このような処理は設計データの状態に依存するため、データと共にＤＢに格納されることが望ましい。従来のそのようなＤＢ設計法は一般論すぎることと、分析と実装のギャップが大きいという問題がある。

本稿では、データの状態に依存する処理等の制約を分析することにより、ＤＢを設計する手法を提案する。ＤＢを実装するために分析すべき制約を明らかにして、それらの制約を分析する方法と、分析結果をC++ ベースのＯＯＤＢＭＳにより実装する設計法とを提案する。そして、市販のＯＯＤＢＭＳを用いて本手法を適用した事例を報告する。

A Object-Oriented Database Design Dealing with Dynamic Constraints

Makoto Onizuka,　Masashi Yamamuro
email:{onizuka,masashi}@ciladb.ntt.jp

NTT Information and Communication Systems Laboratories

This paper presents a methodology to design object-oriented databases(OODB) to deal with constraints to be satisfied before and after some processes on the data depending on the states of data. It handles instance evolution as well. The methodology comprises analysis of constraints to be considered and implementation of the constraints using C++ base OODBMS. Since it takes into account both analysis and implementation, the gap which is usually seen in OODB design between analysis and implementation proves to be fairly filled by desiging a CASE tool dictionary as an OODB example.

1 はじめに

OODB の利用が適するとされている分野の特徴として、

1. 複雑なデータ構造を持つこと
2. 複雑な制約を持つこと

が挙げられる。1. の複雑なデータ構造に関しては、オブジェクト指向設計法 [1][2][3] を用いることにより、分析・設計が可能である。一方、2. の複雑な制約に関しては、ECA ルールなどを用いる方法 [4][5][6]、ペトリネットを用いる方法 [7][8]、論理を用いる方法 [9] 等によって複雑な制約の表現が可能であるが、分析が容易ではないという問題がある。これらの方法より分析が容易な方法としては、[1][2][3] があるが、DB を設計するには一般論すぎることと、分析と実装のギャップが大きいという問題がある。

本稿では、オブジェクトの構造に関する表現（オブジェクトモデル）と状態に関する表現（ライフサイクル図）を用いて DB の制約を分析する方法、及び OODBMS を用いて DB を実装する方法を提案し、上記の問題を解決する。まず、実装するために必要な制約を整理し、それらの制約を表現するため、OMT[1] のモデルを拡張・変更した。次に、そのモデルを用いて DB の制約分析を行なう方法、及び C++ ベースの OODBMS によって DB を実装する方法を提案する。

本稿は以下のように構成されている。第 2 章では、制約に関する関連研究について述べる。第 3 章では、制約分析による OODB 設計法を示す。第 4 章では、本手法をリレーショナルデータベース設計支援ツールのディクショナリ設計に適用した例を示し、本手法の有効性を確認したことを報告する。

2 関連研究

本章では、制約の分析と表現に関する関連研究とその問題点を整理する。

まず、制約を表現する方法として ECA ルールなどを用いる方法 [4][5][6]、ペトリネットを用いる方法 [7][8]、論理を用いる方法 [9] がある。ペトリネットを用いる方法 [7][8] はペトリネット理論に基づき、論理を用いる方法 [9] は時間論理に基づいているため仕様の検証が容易であるが、記法が複雑であるため分析することが容易でないという問題がある。また、ECA ルールなどを用いる方法 [4][5][6] も制約の表現に関して述べているだけで、その制約の分析法については述べていない。

制約の分析法と表現法の両方に関する研究としては、いわゆるオブジェクト指向設計法 [1][2][3] がある。オブジェクト指向設計 [3] では、ライフサイクル図等によって制約を表現しており、表現法自体の完成度は高いと考えられる。OMT [1] では、制約に関して十分論じていない。OOA/OOD [2] では、状態モデルを用いて制約の分析を行なっているものの、プロセスも状態として扱っているため、実装時に無駄な状態を生じる結果になってしまう。また、カスケードデリートなどのデータ構造に関する制約を状態モデルで表現してしまっている等の問題がある。

さらに、上記のオブジェクト指向設計法 [1][2][3] に共通する重大な問題は、OODB を設計するのには一般論過ぎることと、分析と実装のギャップが大きいということである。例えば、OODB ではオブジェクトの追加・検索・削除に関する制約分析・実装が必須であるが、それらをどのように行なうかの議論は明確でないし、また CAD や CASE 等がサポートする設計業務に現れるオブジェクトでは、そのオブジェクトの進化に応じた動的制約の分析・実装が重要であるが、そのような動的制約をどのように分析し実装するかははっきり論じていない。

3 制約とその記法

制約の分析法を述べる前に、本章では DB を実装するために分析しなければならない制約を整理し、その整理した制約を表現する記法について説明する。なお、本稿で用いているオブジェクト、クラス、インスタンスを以下のように定義する。

オブジェクトの定義 属性値群によって表現される状態と、状態に関する操作であるメソッド群から構成されるもの。オブジェクトはアイデンティティを持つ。

クラスの定義 クラスはオブジェクトの型であり、同時にその型のオブジェクト全てに対し大域的な属性 (クラス属性と呼ぶ) とメソッド (クラスメソッドと呼ぶ) を持つ。

インスタンスの定義 あるクラスに属するオブジェクトを、そのクラスのインスタンスという。インスタンスの属性をインスタンス属性、インスタンスのメソッドをインスタンスメソッドと呼ぶ。

3.1 制約の整理

静的制約 DB において、常に成り立たねばならない制約を静的制約と呼ぶことにする。文献 [12][13] に基づき整理した。

1. あるインスタンスまたはクラスの属性値が、特定のインスタンス群またはクラス群の属性値のドメインを制限するドメイン制約
   (a) 特定の インスタンス または クラス の属性値のドメインを制限する制約
   (b) あるクラスに属するインスタンス群のある属性に関して、重複する値が存在しないという主キー制約
   (c) あるインスタンスまたはクラスの属性値が、特定のインスタンスまたはクラスの属性値を計算によって導き出す制約
   (d) あるインスタンスまたはクラスの存在が、特定のインスタンスまたはクラスの存在を決定する存在制約
   (e) あるインスタンスまたはクラスの属性値が、特定のインスタンスまたはクラスの属性値のドメインを制限する制約
2. ある インスタンス の存在によって、特定の インスタンス の存在数が制限される基数制約 (1 対 1、1 対多、多対多)
   (a) 基数制約の多側について、順序づけが必要ない制約 (1 対多)
   (b) 基数制約の多側について、順序づけが必要な制約 (順序つき 1 対多)
3. 複数の基数制約間の包含関係に関する制約

動的制約 文献 [13] での概念である事前・事後条件を一般化し、動的制約を次のように定義する。

> 特定の条件が満たされ、且つ特定の事象の発生した場合、オブジェクトまたはクラスは新たな状態へ推移する。この特定の条件 (いくつかのオブジェクトの状態、いくつかのクラスの状態、入力、に対する条件) をその事象の事前条件とする。また、上記の事象が終了する時に保証しなければならない条件 (着目しているオブジェクトの状態、着目しているクラスの状態、出力、に対する条件) をその事象の事後条件とする。これら 2 つの条件を合わせて、推移が起こる対象の動的制約と呼ぶ。

静的制約は常に成り立つものであるため、データ構造を表現するオブジェクトモデルで表現する。一方、動的制約はオブジェクトの状態変化に依存するため、オブジェクトの時間推移を表現するライフサイクル図を用いて表現する。オブジェクトモデル及びライフサイクル図について 3.2 章で述べる。

3.2 制約記法

3.2.1 オブジェクトモデル

DB の静的制約を表現するため、表現能力の高い OMT[1] のオブジェクトモデルを用いることにするが、3.1 章の静的制約を表現するには適さない部分があるため、以下のような拡張・制限を行なう。

拡張 1 3.1 章の静的制約 3. である、複数の基数制約間の包含関係に関する制約を表現するため、図 1 の左のように拡張する。多対多の関連の場合は検索方向を指定して (図 1 の左では、クラス A からクラス C 方向) 関連間の包含関係を記述する。

意味制限 1 3.1 章の静的制約 1.(d) である存在制約を表現するため、集約関連は存在制約を表現することに制限する。

3.2.2 ライフサイクル図

3.1 章の DB の動的制約を表現するため、OMT[1] の状態図を拡張 1、拡張 2 のように拡張し (図 1 の右)、また状態図の状態間の関係を簡略化するため、変更 1、変更 2 を行なった。その結果をライフサイクル図と呼ぶことにする。

拡張 1 事象の入力パラメータを表現できるように、事象名に続く () 内にパラメータのクラスを記述するよう拡張する。なお、入力パラメータが複数ある場合は区切り文字"," を用いることとする。

拡張 2 事象の事前・事後条件を表現するため、これらを状態と共に記述するよう拡張する。

変更 1 複数の状態を満たし、新たな状態へ推移する AND 推移を表現するよう変更する。

変更 2 事象発生後の状態が前の状態の制約を継承する場合は通常の矢印を用い、継承しない場合は破線の矢印を用いて、推移を表現するよう変更する。

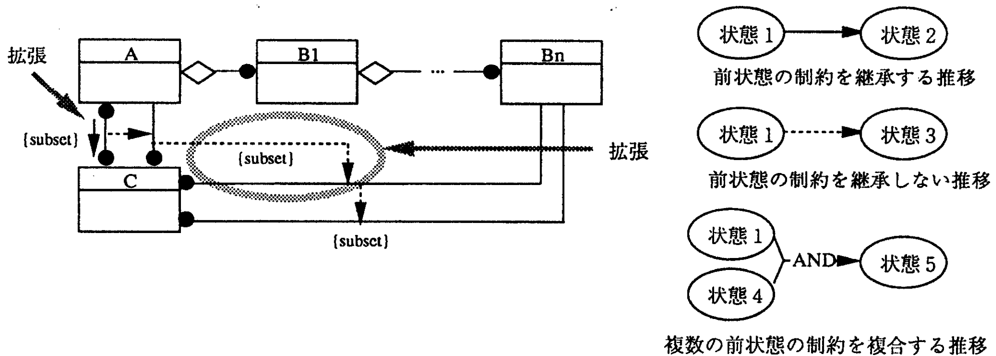

図 1: 記法の拡張

4 制約分析による OODB 設計法

3 章で説明した記法を用いて DB の制約を分析し、その結果を C++ をベースとする OODBMS を用いて実装する設計手順を以下に示す。なお、C++ に関する用語は文献 [14] から引用した。

4.1 分析

- オブジェクトのデータ構造分析

  データ項目の抽出 永続化したいデータを元にデータ項目を抽出する。この際、データ項目の集合に関するデータ項目 (クラス属性に相当する) は抽出しない。

  正規化 項目間の関数従属性を分析し、第三正規化を行なう [12][13]。さらに、多値従属性を分析し、第四正規化を行なう [13]。この結果から得られた項目の集まりをクラスとし、データ項目はそのクラスのインスタンスの属性とする。この時、静的制約 1.(b) の主キー制約を追加する。

  関連の生成 静的制約 2. である基数制約を表現するため、クラス間の関連を生成する。これは参照関係を元に生成する。この際、関連によって表現される外部キーに相当するの属性群は消去する。弱実体の場合は静的制約 1.(d) の存在制約を表現する必要があるため、集約関連として関連を生成する。関連を生成する際は、関連の多重度・関連名・順序つき関連か否かを決定する。

  汎化クラスの生成 任意の 2 つのクラスにおいて、それぞれの主キーに相当する属性の間に包含関係がなく、且つそれぞれの部分集合間に包含関係がある場合、包含関係が成り立つ最大の属性の集合を決定し、それらの属性を持つクラスを、元の 2 つのクラスの汎化クラスとして生成する。そして、元のクラスから汎化クラスの属性を消去し、汎化クラスとの間に継承関係を生成する。また第三正規化の際に、継承関係を持つクラスの属性は不適切に消去されるため、消去された属性のいくつかを汎化クラスまたは特化クラスへ追加しなければならない。

クラス属性の抽出 データ項目抽出の段階で、抽出しなかったデータ項目の集合に関するデータ項目をクラス属性とする。

静的制約 1. の分析 静的制約 1.(c)、(e) がないか決定する。

関連間の関連生成 静的制約 3. がないか決定する。

属性のクラスの決定 属性のクラスを決める。この際、既存のライブラリのクラスでは不十分である場合、新たなクラスを定義する (内部クラスと呼ぶことにする)。

- オブジェクトの制約の分析

ライフサイクルの分析 個々のクラス・インスタンスのライフサイクルを分析して、状態及び状態推移を起こす事象を洗い出し、そのクラス・インスタンスのメソッドとする。事象は、悲観的トランザクションか、または楽観的トランザクションの事後条件であるものとする。また、前状態の制約を継承する状態遷移であるか否かを区別する。集約関連を持つ場合は、部品クラスの状態は全て集約クラスへ伝搬され集約クラスの状態となるが、表現が冗長になるため、1 つの部品クラスの状態だけを参照して生成される集約クラスの状態は表現しないものとする。

事象の分析 動的制約を分析するため、事象の入出力及び事前・事後条件を決定する。再利用性の高いと思われる事前・事後条件はメソッドとして定義する。

- オブジェクトの再整理

汎化クラスの生成 複数の異なるクラスのメソッドまたは複数の異なるクラスのインスタンスメソッドが共通である場合、それら共通のメソッドを持つ汎化クラスを生成する。例えば、DB に関する基本操作 (追加、更新、削除、出力) を持つクラスを用意すべきである。

4.2 実装 (C++ ベースの OODBMS の場合)

クラスのインタフェースの実装 分析より得られたクラスを C++ のクラスとして実装する。次に、インスタンス属性及びインスタンスメソッドを、それぞれ C++ のデータメンバ、メンバ関数として定義するため、それらのスコープを決定する。各インスタンス属性・インスタンスメソッドが、自分のクラス以外から呼び出される否か、またそれらがサブクラスで継承されて利用されるか否かを分析して、スコープ (public, protected, private, virtual, friend) を決定する。例えば、インスタンスを持たない純仮想クラスなどのインスタンス属性・インスタンスメソッドなどは継承されることが前提となっているため、継承による属性・メソッドのスコープに関する考慮が重要である。また、動的結合を用いる場合はスコープが実行時に変化するため、どのクラスのメソッドが実際に起動されるかを注意する必要がある。また、モジュールの 5 つの原則 ([13]2.2 章) をできるだけクラスが満たすようスコープを設定する。

一方、分析より得られたクラス属性及びクラスメソッドを、それぞれ static を用いて C++ のデータメンバ、メンバ関数として定義する。もし、それらをインスタンスの属性やメソッドのように詳細にスコープを定義したい場合は、集合クラスを明示的に構成しクラスメソッドをインスタンスメソッドに変更しなければならない。

次に、分析で事象から得られたメソッドを悲観的トランザクションとして実装し、またその事前・事後条件を実装する。

汎化クラスのメソッドの入出力のクラスが、継承先のクラスに依存する場合、テンプレートクラスを用いて汎化クラスを実装する。DB に関する基本操作 (追加、更新、削除、出力) を持つクラスはその典型である。

状態の実装 状態が事前・事後条件から頻繁に参照されるたり、参照するための処理に時間を要する場合、状態フラグなどで状態を実装し、状態の参照処理を高速化する。

インスタンス進化への対応 インスタンスが進化に応じて詳細化され、1つのクラスだけではインスタンスを実装できない場合がある。この問題に対処するには以下のいずれかの方法が考えられる。

1. 詳細化されたインスタンスを表現できるように、初期状態のインスタンスの属するクラスのサブクラスを新たに C++ のクラスとして生成し、元のクラスを実装した C++ のクラスと、新たに生成したクラス間の移動を用いてインスタンスの状態の進化を実装する。
2. 詳細化されたインスタンスを表現するためのデータメンバを、初期状態のインスタンスの属するクラスを実装した C++ のクラスに追加し、状態フラグを用いて状態の進化を実装する。

関連の実装 通常 OODBMS 側で参照整合性を保つ関連クラスを用意しているため、それらを用いるか、それらを拡張して C++ のクラスのデータメンバとして追加する。OODBMS 側で参照整合性を保つ関連クラスを用いない場合は、メソッドを利用する業務側で参照整合性を考慮する必要がある。

メソッドの内部処理の実装 メソッドの内部処理の実装を行なう。メソッドの部品化を図るため、他のメソッドと共通となる部分を括り出し、新たに C++ のメソッドとする。

永続データ操作メソッド 注意すべき点を以下に挙げる。

- C++ へのインタフェースを持つ OODBMS では、通常 JOIN 相当の操作ができないため、関連をたどることによって検索を行なわねばならない。
- OODBMS が提供する検索結果を格納するクラスと、検索自体の機能は不十分であるため、検索結果クラス自体を拡張して利用するか、または検索結果を操作する必要があることが多い。

多重継承の問題の回避 汎化クラスを用いると多重継承による曖昧性の問題 [14] pp.584 が起こる場合があるが、仮想基本クラスを用いたり、メソッドの再定義をすることによって曖昧性を排除するか、もしくは汎化クラスがない状態へ戻ることにより曖昧性を排除する。

5 RDB 設計支援ツールのディクショナリ設計への適用

本手法をリレーショナルデータベース (以後 RDB と略す) 設計支援ツールのディクショナリ設計に適用した例を示す。RDB 設計支援ツールを選択した理由は、RDB に理論的な裏付けがあるため RDB の設計工程での制約を分析しやすく、本手法の適用が適していると考えられるからである。本章では、まず RDB 設計についての概要を説明し、次に本手法を適用した事例を示す。

5.1 RDB 設計手順

RDB の設計の手順は、テーブルとカラムの洗い出し、カラム間の関数従属性の洗い出し、テーブル作成、第二正規化、第三正規化、アプリケーションの DB アクセス情報に基づく DB の再構成、インデックスの付与という手順で行なわれる。

5.2 本手法による分析

5.2.1 オブジェクトモデル

図2 にオブジェクトモデルを示した。より見やすくするため、継承関係と包含関係を独立に記述し、属性やメソッドのほとんどを省略した。

包含関係の図において静的制約が表現されている。例えば、*Table* と *Index* の間の存在制約や、*Table* に属する任意の *Index* を構成する *Column* 群は、*Table* を構成する *Column* 群の subset であるという関連間制約が表現されている。

継承関係の図の方も説明すると、クラス *Basic* は永続性を持つクラス *PObject* をスーパークラスとすることにより自分自身のインスタンスを永続化している[1]。また *Basic* は、永続性を持つインスタンスを操作する基本機能 (追加、更新、削除、出力、ソート) を有するクラスでもある。クラス *Table_Column_Common* はクラス *Table* と *Column* の共通部分を汎化したものである。

[1] この永続化に関する仕様は、ODMG-93[15] での標準仕様である

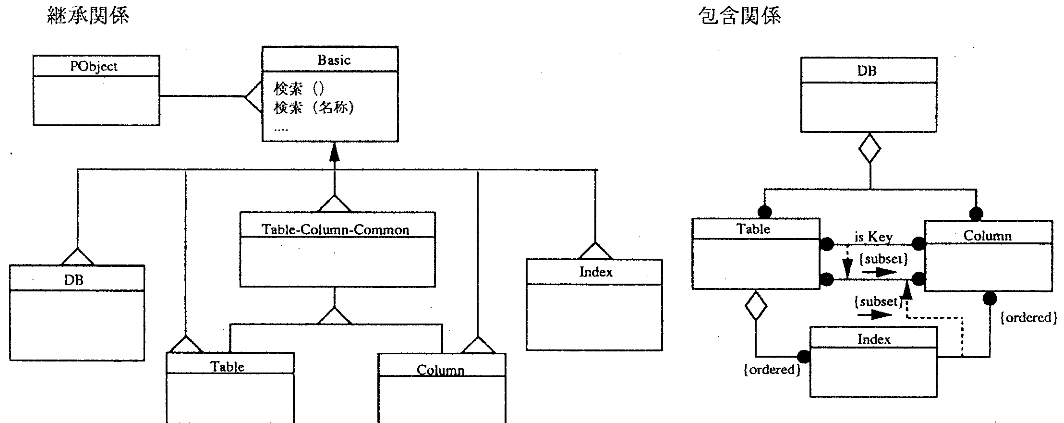

図 2: オブジェクトモデル

5.2.2 ライフサイクル図

図3 に、*DB* インスタンスの正常処理に関するライフサイクル図を示した。

ライフサイクル図においては動的制約が表現されている。例えば、自分自身の *DB* インスタンスに属する全ての *Table* は参照する *Column* が追加されており、且つ自分自身の *DB* インスタンスに属する全ての *Column* において従属性の矛盾がないならば、" テーブル&カラム間の参照矛盾チェック" を起動でき、それが正常終了した場合は、状態が" テーブル&カラム間参照無矛盾" になるといる制約が表現されている。また、テーブルを再構成すると第三正規化の結果は保証されないなども表現されている。

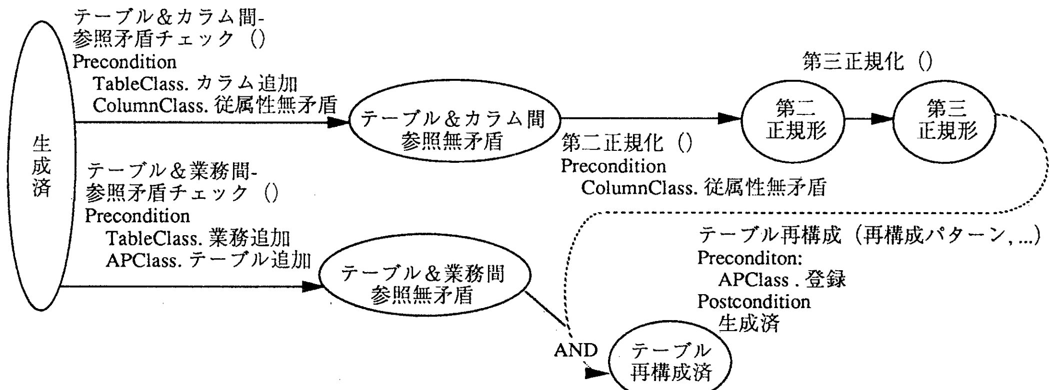

図 3: *DB* インスタンスのライフサイクル図

5.3 本手法による実装と評価

市販の C++ ベースの OODBMS を用いて分析モデルの実装を行なった。ここで、C++ ベースの OODBMS 特有の多重継承の曖昧性について説明しておく。クラス *Basic* を実装するには、メソッド (追加、更新、削除、出力、ソート) の入出力クラスが継承先のクラスに依存するため、テンプレートクラスを用いる必要がある。つまり、*Basic* ⟨*T*⟩ として定義し、それを継承する側では T にそれぞれの継承先のクラス名を代入するのである。この時、クラス *Table* を見てみると、そのスーパークラスは、*Basic* ⟨*Table*⟩ と *Basic* ⟨*Table_column_Common*⟩ であるため、多重継承による曖昧性が生じてしまっている。この問題を解消するには、*PObject* を *Basic* ⟨*T*⟩ の仮想基本クラスとし、*Basic* ⟨*T*⟩ で定義されているメソッドを、クラス *Table* において全て再定義しなければならない。

本手法は、実装に必要な制約が何かを明確にし、それをどのような方法で分析するかを明確にしたため、実装とのギャップが小さくなった。このことから、設計の手戻りが減少したと考えられる。また、個々の事象の動的制約を明確にしているため、事象を誤った条件で起動することが少なくなり、実装ミスが軽減できたと考えられる。

6 おわりに

本稿では、DB の制約の分析法と、その分析結果を C++ ベースの OODBMS を用いて実装する方法を提案した。そして、本手法を RDB 設計支援システムのディクショナリ設計に適用し、本手法の分析と実装のギャップが小さいため、設計の手戻りが減少したことと、事象の動的制約が明確であるため事象の誤った起動が減少したこと、を確認できたと考える。

今後は、オブジェクトリレーショナルシステムへ実装する方法を検討し、本手法をより洗練する予定である。

参考文献

[1] J.Rumbaugh, M.Blaha, W.Premerlani,F.Eddy and W.Lorensen "Object-Oriented Modeling and Design" Prentice-Hall,1991.

[2] S.Shlaer and S.J.Mellor "Object Lifecycles"Yourdon Press,1992.

[3] 酒井, 堀内"オブジェクト指向設計" オーム社,1993.

[4] U.Dayal, B.Blaustein, A.Buchmann, U.Chakravarthy, M.Hsu, R.Ledin, D.McCarthy, A.Rosenthal and S.Sarin "The HiPAC Project:Combining Active Databases and Timing Constraints" SIGMOD record, Vol.17, No.1, pp.51-70, March 1988.

[5] U.Dayal "Active Database Systems" Proc. of the 3rd Int. Conf. on Data and Knowledge Bases, Jerusalem, Isral, June 1988.

[6] H.Martin, M.Abida and B.Defude "Consistency Checking in Object Oriented Databases: a Behavioral Approach" Proc. of the 2nd Int. Conf. on Information and Knowledge Management, pp.53-68, 1992.

[7] G.Kappel and M.Schrefl "A Behavior Integrated Entity-Relationship Approach for the Design of Object-Oriented Databases" Proc. of the 7th Int. Conf. on Entity-Relationship Approach, Batini,C.(Ed), pp.175-192, 1988.

[8] G.Kappel and M.Schrefl "Object/Behavior Diagrams" Proc. of the 7th Int. Conf. on Database Engineering, pp.530-539, 1991.

[9] T.Hartmann, R.Jungclaus and G.Saake "Aggregation in a Behavior Oriented Object Model" Proc. of 1992 ECOOP Conf., O.L. Madison(Ed.), Springer-Verlag, 1992.

[12] 池田, 川下, 坂田, 関根, 中川"データベース概念設計" 阿部出版,1993.

[13] H.F.Korth, A.Silberschatz "Database System Concepts" McGraw-Hill,Inc.,1991.

[13] B.Meyer "オブジェクト指向入門" アスキー出版局,1991.

[14] S.B.Lippman "C++ プライマー第2版" トッパン,1993.

[15] R.G.G.Cattel "The Object Database Standard:ODMG-93(Release 1.1)" Morgan Kaufmann Publishers,1994.