オブジェクトの振舞いに関するコントラクトの設計について†

酒井 博敬††

オブジェクトの操作とその制約の設計，すなわち振舞いの設計はオブジェクト指向設計の基本課題である．本論文は，オブジェクト指向によるシステムの概念設計の基礎となる，オブジェクトの振舞いのモデルとこれにもとづく振舞いの設計法について論じている．オブジェクトの振舞いは，オブジェクトの発生，状態遷移，消滅の過程によって表現される．その形式表現のために，振舞いモデルの基本要素としてライフサイクルスキーマを定義し，さらに振舞いの構造化の手段として，ライフサイクルスキーマ洗練の概念を定義した．オブジェクトの状態遷移は一貫性制約にしたがわねばならない．ここでは複数のオブジェクトの振舞いに関する制約を，状態間制約および遷移制約として形式化し，体系化するとともに，これらの制約を制約関連図として表現した．一般にある業務環境において，与えられた機能を果たすために必要なオブジェクトの集まりをサブシステムという．サブシステムにおける複数のオブジェクトの協調的な振舞いを記述するために，コントラクトの概念を定義した．コントラクトは，オブジェクトの振舞いに関する制約と機能ごとのスクリプトの記述から構成され，サブシステムにおけるオブジェクトの振舞いを設計するための仕様となるものである．

1. はじめに

オブジェクト指向設計は情報工学の多くの分野に導入され，急速に発展しつつある．オブジェクトは実世界において認識される実体の構造，操作，および制約に関する諸性質の凝集概念である．そしてオブジェクト指向概念は，オブジェクトの再利用性と拡張性を目標として，オブジェクトアイデンティティ，構造，操作，制約の諸性質をカプセル化したクラスまたは型，またこれら諸性質の継承，および操作起動のためのオブジェクト間のメッセージ送受などの諸概念によって形成される[1),2)]．

オブジェクトの構造とその制約に関する設計は，とくに意味データモデルにおいて多くの研究がなされ，類型化，集約化，汎化・特化などの抽象化概念として発展し，整理されてきた[3),6),8),11)]．

一方オブジェクトの操作とその制約の設計，いわゆる振舞いの設計はオブジェクト指向設計の一つの中心課題である．これまでも意味データモデルにおいて，実体の操作に関する性質を，実体と一体化して設計するための技法とモデル化が研究，開発されてきた[4),9),10),15)]．

振舞いはオブジェクトの構造と密接な関係がある．この観点から，集約化，結合，汎化・特化などのオブジェクトの構造の抽象化に対比させて，オブジェクトの操作に関するシーケンス，繰返し，選択などの制御の抽象化を行い，個別トランザクションごとにこれをオブジェクトの基本操作に分解し，トランザクションの仕様を導く技法が開発されている[4),15)]．さらに最近，ライフサイクル概念を用いてオブジェクト固有の操作を組織的に抽出する方法，あるいはそのための設計ツールに関する研究，開発が進められている[7),13),16)~19)]．しかしこれまでのところ，振舞いの基本的なモデルの確立，およびオブジェクト固有の振舞いに関する一貫性制約の組織的な抽出の方法に関する議論は十分なされていない．

オブジェクト指向による情報システムの設計は，概念設計とインプリメンテーションからなる．概念設計は，対象とするアプリケーションを構成するオブジェクトの諸概念を抽出し，仕様化する段階である．すなわち，いかなるオブジェクトが存在し，それらがいかなる構造と振舞いをもち，そしていかなる制約にしたがうかを分析し，仕様化する段階である．仕様化されたオブジェクトの諸概念は，特定のオブジェクト指向言語あるいはオブジェクト指向データベースシステムを用いてインプリメントされる．本論文では，とくにオブジェクトの振舞いと振舞いに関する一貫性制約の抽出に着目した概念設計の技法について論じる．

ある業務環境を構成するオブジェクトの集まりをサブシステムとよぶ．いかなるサブシステムにおいても，業務機能を実現するために複数のオブジェクトが協調的に振舞わなければならない．サブシステムにおけるオブジェクトの振舞いに関する一貫性を維持するためには，個々のオブジェクトに固有な振舞いとそれにともなう制約だけではなく，協調するオブジェクト

† Designing Contracts for Behavior of Objects by HIROTAKA SAKAI (Department of Industrial and Systems Engineering, Faculty of Science and Engineering, Chuo University).

†† 中央大学理工学部管理工学科

間の振舞いの関連とその間で維持されるべき制約を組織的に抽出し，それにもとづいて振舞いの協調に関する仕様を定めなければならない．オブジェクトの協調的な振舞いの設計に関しては，スクリプト (script)[3] あるいはコントラクト (contract)[5),14)] の概念が提案されている．しかしこれらの概念の形式化および一貫性制約を考慮した設計法については議論がなされていない．

この論文では次の事項を目標として，オブジェクトの更新操作の体系を表現する振舞いをモデル化し，サブシステムにおけるオブジェクトの協調的な振舞いを設計する方法について論じる．

(1) オブジェクト固有の振舞いの表現：オブジェクト固有の振舞いを明示的に表現する手段として，オブジェクトの発生，状態遷移，消滅の過程をライフサイクルスキーマとして定義する．さらに振舞いの表現を詳細化する手段として，ライフサイクルスキーマ洗練の概念を定義する．

(2) 振舞いにおける一貫性制約の表現：オブジェクトの振舞いは一貫性制約にしたがわねばならない．ここでは複数のオブジェクトの振舞いに関する制約を状態間制約および遷移制約として定義し，その体系を制約関連図として表現する．

(3) 振舞いの協調 (collaboration) の設計：サブシステムを構成するオブジェクトをそのサブシステムのパートナとよぶ．サブシステムにおけるパートナの協調的な振舞いを記述するために，コントラクトの概念を定義する．コントラクトは，オブジェクトの振舞いに関する制約と機能ごとのスクリプトの記述から構成される．スクリプトは，機能を実現するためにパートナがいかに協調して振舞うべきかについて記述したもので，振舞いの制約にしたがって設計される．コントラクトはサブシステムにおけるオブジェクトの振舞いを設計するための仕様を与えるものである．

論文の構成は以下のとおりである．まず2章においてオブジェクトの構造の抽象化概念についてまとめる．3章では振舞いモデルの基本要素として，ライフサイクルスキーマおよびライフサイクルスキーマ洗練の概念について定義する．これらの概念にもとづいて，4章では振舞いに関する制約の定義を行う．以上の諸概念を用いて，5章ではサブシステムにおけるコントラクトの設計について論じる．

2. オブジェクトの構造

オブジェクト指向設計の基本はオブジェクトの構造設計にある．本章では，類型化，集約化，および汎化・特化の各抽象化概念に代表されるオブジェクトの構造的性質について定義する．

[1] 類型化 (Classification)

オブジェクトの構造に関する性質を特性 (property) という．オブジェクトの特性は，そのオブジェクトとあるオブジェクト（またはオブジェクトの集合）との関連である．共通の特性をもつオブジェクトの集合を考えることを類型化，またこの集合をクラスという．クラスに属するオブジェクトの共通の特性を単にクラスの特性ともいう．

クラスEのオブジェクトeがある特性pによって，Eと同一または異なるクラスFのオブジェクトf（あるいはFのオブジェクトの集合 $f^*=\{f_1,\cdots,f_k\}$）と関連をもつとき，f（あるいは f^*）をeに対する特性pの値といい，p(e) で表す．クラスEの任意のオブジェクトeに対して，特性pの値f（p(e) が集合 f^* のときはその各要素）は同一クラスFのオブジェクトでなければならない．Fをpの領域という．オブジェクトに対する特性の値は時間とともに変わり得る．

Eの任意のオブジェクトに対して特性pのとり得る値の集合Vは，一般にFあるいはFのべき集合 F^*，あるいはそれらの部分集合であるが，これをEに対する特性pの値域という．値域Vをもつ特性pを，$p:V$ と表記することがある．

さらに特性のとる値に関する制約条件は特性制約として定義される．特性制約は，クラスを類 (sort) の解釈とする多類1階述語論理の閉論理式によって表現される．

[2] 集約化 (aggregation)

クラスEがそれぞれクラス $E_1,\cdots,E_n$ を領域とする特性 $p_1,\cdots,p_n$ をもつとし，Eのオブジェクトeに対する特性 $p_1,\cdots,p_n$ の値をそれぞれ $e_1,\cdots,e_n$ とする．各 e_i はクラス E_i のオブジェクト，または E_i のオブジェクトの集合である．このときオブジェクトeは $e_1,\cdots,e_n$ の集約化によって定義されるといい，eを集約オブジェクト，各 e_i（e_i が集合であるときはその各要素）をeの成分オブジェクトという．また各 E_i をEの成分クラス，Eを $E_1,\cdots,E_n$ を成分クラスとする集約クラスという．集約と成分の関連は相対的な意味をもつ．たとえば，集約クラスEの成分クラス E_i

が，さらに成分クラス $F_1, \cdots, F_k$ をもつ集約クラスとなることがある．

［3］ 汎化・特化 (generalization, specialization)

二つのクラスE，Fにおいて，Fのオブジェクトはまた Eのオブジェクトでもあるとき，FはEの特化クラスまたはサブクラス，またEはFの汎化クラスまたはスーパクラスであるといい，is_a (F, E) で表す．スーパクラス，サブクラスの関係は階層関係をなすものとする．

FがEのサブクラスであるとき，Eの特性はFの特性でもある．このことを，FはEの特性を継承するという．Fは継承した特性のほかに付加的な特性をもつ．FがEから継承した特性の領域は，Eにおけるその特性の領域のサブクラスに再定義されることがある．さらにFがEから継承した特性の値域は，Fにおいて再定義されることもある．

クラス	特性：値域	注
Emp	e_name : string salary : integer works_in : Proj	Proj は Emp の成分クラス
Proj	p_name : string p_status :{'active','suspended'} leader : Emp members : set of Emp task : Prod	Emp, Prod はProj の成分クラス
Prod	d_name : string d_status : {'planned', 'designed', ' approved'}	
S_Prod	is_a(S_Prod, Prod) d_status : {'planned', 'documented', 'designed', 'approved'} version# : integer	S_Prod は Prod のサブクラス
H_Prod	is_a(H_Prod, Prod) d_status : {'planned', 'drawn', 'designed', 'approved'} version# : integer	H_prod は Prod のサブクラス

図 1 プロジェクト管理におけるクラスの構造的性質の定義
Fig. 1 Definitions of structural characteristics of classes in project management.

［例1］ 図1はプロジェクト管理における，それぞれ社員，プロジェクト，製品，ソフトウェア製品，およびハードウェア製品を表すクラス Emp, Proj, Prod, S_Prod, および H_Prod の構造的な性質に関する定義である．Emp は Proj を成分クラスとしてもち，また Proj は Emp および Prod を成分クラスとしてもつ．S_Prod と H_Prod は Prod のサブクラスで，Prod の特性を継承するとともに付加的な特性 version# をもつ．さらに継承した特性 d_status の値域はそれぞれ S_Prod, H_Prod において再定義されている．

さらにクラス Emp の特性に関しては，次の特性制約を仮定する．

valid_salary :
  for all e : Emp (salary (e)≥minimum_value);
valid_project_salary :
  for all e : Emp (works_in (e)≠null⇒
    salary (e)≤salary (leader (works_in (e))));

3. 振舞いモデル

3.1 ライフサイクルスキーマ

振舞いはオブジェクトの発生，状態遷移，消滅の過程を表すライフサイクルによって表現することができる．ライフサイクルは，オブジェクトの状態 (state) および状態遷移をもたらす事象 (event) の集合によって表現される．状態はオブジェクトのライフサイクルにおける到達点ないし段階を，また事象は状態遷移を与える操作を意味する．同一クラスのオブジェクトは定められたライフサイクルに沿ってのみ状態遷移するという考えに立ち，以下のようにクラスのライフサイクルスキーマを定義する．

$\mathcal{U}, \mathcal{T}$ を互いに素な記号の有限集合とし，それぞれ状態集合，事象集合とよぶ．また $\mathcal{U}, \mathcal{T}$ の要素をそれぞれ状態，事象とよぶ．$\mathcal{U}, \mathcal{T}$，および有向辺集合 $\mathcal{A} \subset (\mathcal{U}\times\mathcal{T}) \cup (\mathcal{T}\times\mathcal{U})$ からなる2部グラフを $\mathcal{B} = (\mathcal{U}, \mathcal{T}, \mathcal{A})$ で表す．すなわち $\mathcal{B}$ は，状態と事象を節点とし，状態から事象への有向辺，および事象から状態への有向辺によって構成されるグラフである．

$\mathcal{B}$ において $u \in \mathcal{U} \cup \mathcal{T}$ とするとき，pre (u), post (u), sources $(\mathcal{B})$, および sinks $(\mathcal{B})$ を次式で定義する．

$$\mathrm{pre}(u) = \{v \mid v \in \mathcal{U} \cup \mathcal{T},\ (v, u) \in \mathcal{A}\}$$
$$\mathrm{post}(u) = \{v \mid v \in \mathcal{U} \cup \mathcal{T},\ (u, v) \in \mathcal{A}\}$$
$$\mathrm{sources}(\mathcal{B}) = \{u \mid u \in \mathcal{U} \cup \mathcal{T},\ \mathrm{pre}(u) = \phi\}$$
$$\mathrm{sinks}(\mathcal{B}) = \{u \mid u \in \mathcal{U} \cup \mathcal{T},\ \mathrm{post}(u) = \phi\}$$

sources $(\mathcal{B})$, sinks $(\mathcal{B})$ をそれぞれ $\mathcal{B}$ の始点集合，終点集合，またその要素をそれぞれ $\mathcal{B}$ の始点，終点という．

始点および終点以外の $t \in \mathcal{T}$ に対して，$|\mathrm{pre}(t)| = 1$, $|\mathrm{post}(t)| = 1$ ($|X|$ は集合 X の要素の個数を表す) であるような $\mathcal{B}$ を振舞いグラフという．

クラスEのライフサイクルスキーマを，どの節点も始点から到達可能な連結な振舞いグラフ $\mathcal{B}=(\mathcal{U}, \mathcal{T}, \mathcal{A})$ によって定義する．ただし $\mathcal{B}$ の始点および終点は $\mathcal{T}$ の要素で，始点はただ一つである．終点は複数個あってもよい．

クラスEのライフサイクルスキーマをペトリネットグラフ[12]に図示したものをライフサイクル図という．ライフサイクル図では，状態集合 $\mathcal{U}$ の要素は円で，事象集合 $\mathcal{T}$ の要素はたて形の長方形で表される．

ライフサイクル図の意味解釈は次のとおりである．事象 $t\in\mathcal{T}$ に対し，pre(t)，post(t) の各要素（いずれもただ一つ）をそれぞれ t の事前状態 (prestate)，事後状態 (poststate) とよぶ．E のオブジェクト e の状態が pre(t) であるとき t は起動可能となり，t が起動されると e の状態は post(t) に移る．これを e の状態遷移という．とくにグラフ中に閉路があれば，同一事象の起動による同一状態への遷移が繰り返し起こり得る．また始点および終点はそれぞれオブジェクトの生成および消去を行う事象である．

オブジェクトがある状態に遷移するとき，オブジェクトのある特性の値が更新される．更新対象となる特性およびその特性の値に関する指定を，状態の特性仕様という．

オブジェクトの特性の値に関して常に成り立つべき制約条件は特性制約として定義されたが，その破壊は特性制約に含まれる特性の値を更新する状態への遷移において起こり得る．したがってそのような状態への遷移に際して，特性制約が破壊されないことを確かめるよう指定する．これを状態に対する特性制約モニタ指定という．

クラスの特性，特性制約，およびライフサイクルスキーマはクラス記述として表現される．ライフサイクルスキーマは状態と事象に分けて記述し，状態に関してはその特性仕様と特性制約モニタ指定を，また事象に関しては事前状態と事後状態を書く．

[例2] 図2(a)はクラス Prod のライフサイクル図である．また図2(b)は Prod のクラス記述である．イタリックで書かれた部分が状態名で，*not_exist* はオブジェクトが存在していない状態を表す．

オブジェクトのインプリメンテーションを考えるとき，状態はオブジェクトの記憶，すなわちオブジェクトの特性値の集合を指し，事象はその記憶を更新（生成，変化，消滅）するメソドとして，また事象の起動はオブジェクトへのメッセージ送信として実現される．

3.2 ライフサイクルスキーマの洗練

ライフサイクルスキーマにおける状態および事象を洗練することによって，オブジェクトの振舞いの表現を構造的に詳細化することが可能になる．

$\mathcal{B}$ を振舞いグラフとする．$\mathcal{B}$ の状態または事象 u を，より細分された状態および事象をもつ振舞いグラフに分解することを u の洗練という．$\mathcal{B}$ のいくつかの状態または事象を洗練して得られる振舞いグラフ $\mathcal{B}^*$ が次の条件 [R_1]，[R_2]，および [R_3] をみたすとき，$\mathcal{B}^*$ を $\mathcal{B}$ の洗練振舞いグラフという．

[R_1] $\mathcal{B}$ の状態または事象 u を振舞いグラフ B(u) に分解したとき，B(u) の始点および終点は u が状態ならばすべて状態であり，u が事象ならばすべて事象である．

ただし B(u) が閉路を連結成分としてもつときは，

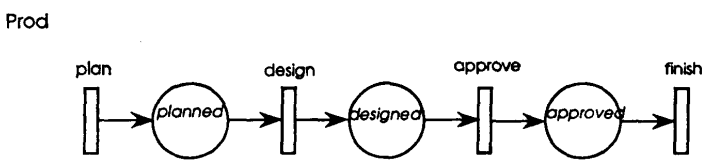

図 2(a)　クラス Prod のライフサイクル図
Fig. 2(a)　The life cycle diagram of class Prod.

```
class Prod;

properies
 d_name : string;
 d_status : {'planned', 'designed', 'approved'};

states
  planned : d_name ≠ null; d_staus = 'planned';
  designed : d_status = 'designed';
  approved  : d_status = 'approved';

events
  plan (dName: string)
    prestate :  not_exist;
    poststate : planned;
  design
    prestate : planned;
    poststate : designed;
  approve
    prestate : designed;
    poststate : approved;
  finish
    prestate : approved;
    poststate : not_exist;

end class Prod;
```

図 2(b)　クラス Prod の記述
Fig. 2(b)　The description of class Prod.

u におけるオブジェクトのライフサイクルの意味解釈から，閉路中の適切な状態または事象を閉路の始点あるいは終点と解釈するものとする．

[R_2] $\mathcal{B}$ の状態 u が B(u) に洗練されたとき，次が成り立つ．

(1) B(u) の任意の始点 v に対して，pre(v) の要素は pre(u) の要素であるか，または pre(u) のある要素 r の洗練 B(r) の終点である．

(2) B(u) の任意の終点 w に対して，post(w) の要素は post(u) の要素であるか，または post(u) のある要素 s の洗練 B(s) の始点である．

[R_3] $\mathcal{B}$ の事象 t が B(t) に洗練されたとき，次が成り立つ．

(1) $|pre(t)|=1$ (すなわち t がBの始点でない) ならば，B(t) の任意の始点 r に対して，pre(r) の要素は pre(t) の要素であるか，または pre(t) の要素 u の洗練 B(u) の終点である．

(2) $|post(t)|=1$ (すなわち t がBの終点でない) ならば，B(t) の任意の終点 s に対して，post(s) の要素は post(t) の要素であるか，または post(t) の要素 v の洗練 B(v) の始点である．

クラスEのライフサイクルスキーマ $\mathcal{B}$ が振舞いグラフ $\mathcal{B}^*$ に洗練されるとき，Eのライフサイクルスキーマは $\mathcal{B}$ から $\mathcal{B}^*$ に洗練されるという．また $\mathcal{B}^*$ を $\mathcal{B}$ の洗練ライフサイクルスキーマという．もはやそれ以上洗練されないライフサイクルスキーマを原子ライフサイクルスキーマといい，原子ライフサイクルスキーマにおける状態および事象を，それぞれ原子状態および原子事象という．原子事象は，それ以上細分することのできないオブジェクト固有の基本操作とみなすことができる．

$\mathcal{B}$ の洗練ライフサイクルスキーマ $\mathcal{B}^*$ のライフサイクル図は，便宜上 $\mathcal{B}$ と $\mathcal{B}^*$ の要素を重ねた形で表す．

[例3] 図3(a)は Prod のサブクラス S_Prod, H_Prod のライフサイクル図，図3(b)は S_Prod のクラス記述である．図2(a)，図3(a)とも原子ライフサイクルスキーマを表しているが，後者は前者を洗練したものになっている．

[例3] で示されたように，サブクラスはスーパクラスのライフサイクルスキーマまたはその洗練ライフサイクルスキーマをライフサイクルスキーマとしてもつ．このことをサブクラスはスーパクラスのライフサ

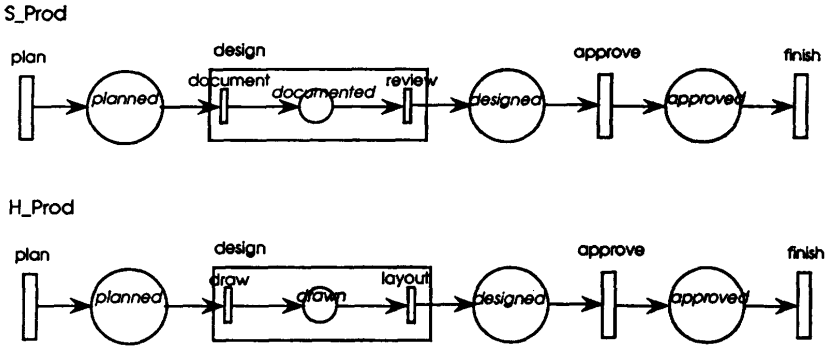

図3(a) クラス S_Prod, H_Prod のライフサイクル図
Fig. 3(a) The life cycle diagrams of classes S_Prod and H_Prod.

```
class S_Prod is_a Prod;

properties
 d_status : {'planned', 'documented', 'designed', 'approved'}
 version# : integer;

states
  documented : d_status = 'documented';

events
  design
    document
      prestate : planned;
      poststate : documented;
    review
      prestate : documented;
      poststate : designed;

end class S_Prod;
```

図3(b) クラス S_Prod の記述
Fig. 3(b) The description of class S_Prod.

イクルスキーマを継承するという．サブクラスが継承したライフサイクルスキーマにおいて，ある状態または事象に対する洗練は，対応するスーパクラスの状態または事象の再定義を意味する．たとえば事象のメソドとしてのインプリメンテーションを考えるとき，事象の洗練はスーパクラスにおいてテンプレートとして定義されたメソドをサブクラスにおいて具象化することを意味する．

[例4] 図4はEmpおよびProjの原子ライフサイクルスキーマのライフサイクル図，また図5，図6はそれぞれのクラスのクラス記述である．

exist はオブジェクトが存在していることを表すもっとも粗い状態で，それ以外の状態はすべて原子状態である．状態 *exist* はいくつかの原子状態に洗練されるが，それらの原子状態はオブジェクトのライフサイクルにおいて変化するため，クラス記述ではオブジェクト生成時の初期の原子状態を，記号 initially のあとの括弧の中に指定している．

[クラス Emp について]

実世界における社員の給与歴および職務歴に対応して，原子状態 *has_salary*（昇級または減給された状態），*job_assigned*（プロジェクトを担当した状態），および *job_released*（プロジェクトを解任された状態）をもつ．とくに事象 hire の事後状態および fire の事前状態はいずれも *exist* であるが，原子状態としては *has_salary* と *job_released* になければならないことを表している．

さらに状態 *has_salary* および *job_assigned* の記述では，特性制約 valid_salary および valid_project_salary のモニタ指定がなされている．

[クラス Proj について]

実世界におけるプロジェクトの活動状態に対応して，原子状態 *active*（"プロジェクトが活動している状態"），*suspended*（"プロジェクトが休止している状態"），*has_leader*（"プロジェクトのリーダが任命された状態"），*has_members*（"プロジェクトのメンバが任命された状態"），*no_leader*（"プロジェクトのリーダが解任された状態"），および *no_members*（"プロジェクトのメンバが解任された状態"）をもつ．とくに事象 create の事後状態および terminate の事前状態はいずれも *exist* であるが，原子状態としては前者は *active*, *has_leader*, *no_members*，後者は *suspended*, *no_leader*, *no_members* になければならないことを表している．

4. 振舞いの制約

オブジェクトの状態および状態遷移に関する一貫性制約を振舞いの制約とよぶ．

単一のオブジェクトが事象の起動によってある状態に遷移するには，ライフサイクルスキーマによって定義された事前状態になければならない．この意味でライフサイクルスキーマは，オブジェクト単独の振舞いに関する一つの制約を与えるものである．

複数のオブジェクトが存在する環境においては，複数のオブジェクトの状態および状態遷移の間に何らかの関連が維持されることが要求される．この間連はライフサイクル制約のみでは表現されない．複数のオブ

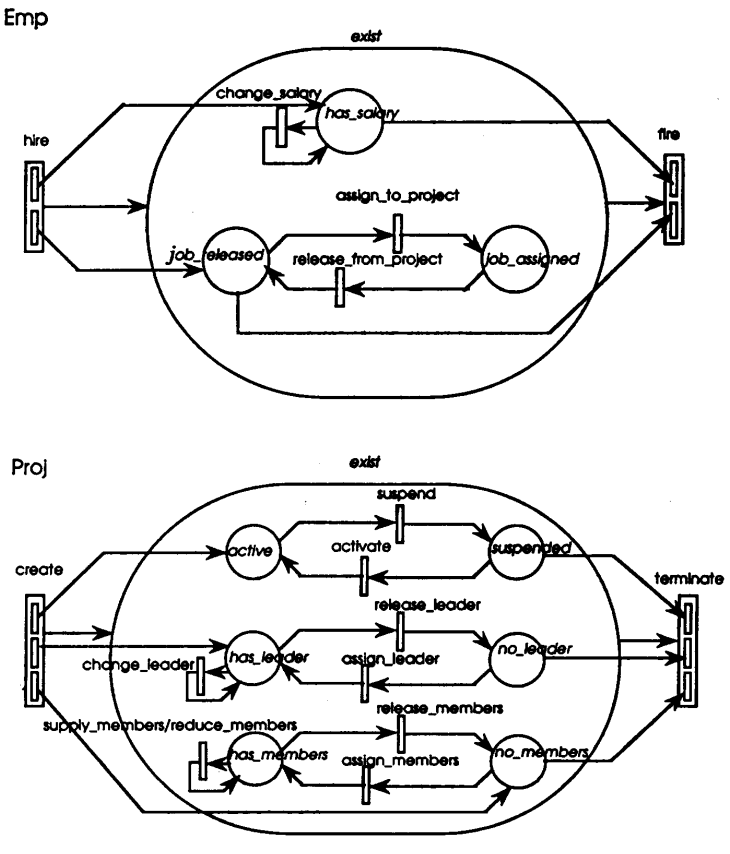

図4　クラス Emp, Prod の原子ライフサイクル図
Fig. 4　The primitive life cycle diagrams of classes Emp and Prod.

ジェクトの状態および状態遷移の間で維持されるべき一貫性制約を，状態間制約および遷移制約として定義する．

[状態間制約]

クラスEのオブジェクトeが状態uにあることを state (e : E, u) で表し，これを状態述語という．「あるクラスEのオブジェクトeが状態uにあるとき，同一または異なるクラスFのオブジェクトfが状態vになければならない」という制約を状態間制約といい，次のように表す．

state(e : E, u) requires state (f : F, v)

状態間制約は，オブジェクトeが状態uにあるかぎり，オブジェクトfは状態vを持続しなければならないことを要求する．

記号 requires の右には複数個の状態述語が現れてもよい．状態間制約はオブジェクトの状態遷移の波及の可能性を意味する．

[遷移制約]

あるクラスEのオブジェクトeに対する事象tの起動を [e : E, t] で表す（ただしあるオブジェクトを生成する場合は e : E をクラスE自身に置き換える）．

「事象起動 [e : E, t] を実行するためには，t の事前状態および事前状態に関する状態間制約に加えて，[e : E, t] の実行時にEと同一または異なるクラスFのオブジェクトfが状態vになければならない」という制約があるとき，これを遷移制約といい，次のように表す．

[e : E, t] requires state(f : F, v)

遷移制約は，オブジェクトeに対する事象tの起動の時点でオブジェクトfは状態vになければならないことを要求するが，eが事後状態に遷移した後もfが状態vを持続する必要はない．

記号 requires の右には複数個の状態述語が現れてもよい．

[例5] クラス Emp, Proj, Prod のオブジェクトに関する状態間制約と遷移制約を次のように仮定する．

[状態間制約]

state(p : Proj, *active*) requires state (p : Proj, *has_leader*)

state(p : Proj, *has_leader*) requires state(p : Proj, *active*), state(e : Emp, *job_assigned*) (ただし e=leader(p))

state(p : Proj, *has_members*) requires state(p : Proj, *active*), state(e : Emp, *job_assigned*) (ただし e は members(p) の任意の要素)

state(p : Proj, *suspended*) requires state(p : Proj, *no_leader*), state(p : Proj, *no_members*)

state(p : Proj, *no_leader*) requires state(p : Proj, *suspended*)

[遷移制約]

[Proj, create] requires state(d : Prod, *planned*) (ただし d は生成される Proj のオブジェクトの属性 p の値)

[p : Proj, release_leader] requires state(e : Emp, *job_released*) (ただし e=leader(p))

[p : Proj, release_members] requires state(e : Emp, *job_released*) (ただし e は members(p) の任意の値)

[d : Prod, design] requires state (p : Proj, *has_members*)

```
Class Emp;

properties
 e_name : string;
 salary : integer;
 works_in : Proj;

property constraints

 valid_salary : for all e:Emp (salary(e) ≥ minimum_salary);

 valid_project_salary : for all e:Emp (works_in(e) ≠ null ⇒
                                  salary(e) ≤ salary(leader(works_in(e))));

states
 exist : initially (has_salary, job_released); e_name ≠ null;
 has_salary  : salary ≠ null;  monitor valid_salary, valid_project_salary;
 job_assigned  : works_in ≠ null;  monitor valid_project_salary;
 job_released  : works_in = null;

events
 hire(eName:string, aSalary:integer);
  prestate : not_exist;
  poststate : exist(has_salary, job_released);
 fire
  prestate : exist(has_salary, job_released);
  poststate : not_exist;
 change_salary(aSalary:integer);
  prestate : has_salary;
  poststate : has_salary;
 assign_to_project(aProject:Proj);
  prestate : job_released;
  poststate : job_assigned;
 release_from_project;
  prestate : job_assigned;
  poststate : job_released;

end Class Emp;
```

図5 クラス Emp 記述
Fig. 5 The description of class Emp.

```
Class Proj;

properties
 p_name : string;
 p_status : ('active', 'suspended');
 leader : Emp;
 members : set of Emp;
 task : Prod;

states
 exist : initially (active, has_leader, no_members); p_name ≠ null; task ≠ null;
 active : status = 'active';
 suspended : status = 'suspended';
 has_leader : leader ≠ null;
 has_members : members ≠ null;
 no_leader : leader = null;
 no_members : members = null;

events
 create(pName:string, new_leader:Emp, d:Prod);
   prestate : not_exist;
   poststate : exist (active, has_leader, no_members);
 terminate;
   prestate : exist(suspended, no_leader, no_members);
   poststate : not_exist;
 suspend;
   prestate : active;
   poststate : suspended;
 activate;
   prestate : suspended;
   poststate : active;
 assign_leader(newLeader:Emp);
   prestate : no_leader;
   poststate : has_leader;
 release_leader;
   prestate : has_leader;
   poststate : no_leader;
 change_leader(newLeader:Emp);
   prestate : has_leader;
   poststate : has_leader;
 assign_members(newMembers : set of Emp);
   prestate : no_members;
   poststate : has_members;
 release_members;
   prestate : has_members;
   poststate : no_members;
 supply_members(employees:set of Emp);
   prestate : has_members;
   poststate : has_members;
 reduce_members(employees:set of Emp);
   prestate : has_members;
   poststate : has_members;

end Class Proj;
```

図 6 クラス Proj の記述
Fig. 6 The description of class Proj.

(ただし d=task(p))

[d : Prod, approve] requires state (p : Proj, *active*)

(ただし d=task(p))

状態間制約および遷移制約は図7のような制約関連図として表現される．図7において状態は円，事象は正方形で描かれている．さらに便宜上特性制約を菱形で表し，特性制約のモニタ指定を円から菱形への矢印で表している．

5. コントラクトの設計

5.1 コントラクトの概念

一般にオブジェクトは与えられた業務環境において単独に振舞うのではなく，他のいくつかのオブジェクトと協調して振舞う．業務環境において，与えられた機能を果たすために必要なオブジェクトの集まりをサブシステム，各オブジェクトをそのパートナ，オブジェクトの属するクラスをパートナクラスとよぶ．

サブシステムでは，個々のパートナの振舞いは固有のライフサイクルスキーマにしたがうだけではなく，複雑な協調関係に関する一貫性を維持する制約にしたがわなければならない．振舞いの協調に関する仕様を表現する手段としてサブシステムにおけるコントラクトを定義する．コントラクトはサブシステムが外部に対して請け負う責任を仕様として記述したものである．ただしこれは単にサブシステムのもつ機能を列挙したものではなく，それらの機能を実現するために，パートナがどのような制約にしたがって協調的に振舞うべきかを記述したものである．

コントラクトは振舞いの制約に関する記述と，個々の機能に対するスクリプトの記述から構成される．前者はパートナ間で維持されるべき状態間制約および遷移制約の記述で，4章で定義されたものである．後者は各機能を実現するために，どのパートナがどのような事前状態と事後状態の間を遷移すべきかを記述したものである．この意味で，コントラクトは振舞いの制約とパートナクラスの記述という基盤の上に設計される．

5.2 スクリプト

サブシステムの各機能ごとに，それを実現するためにパートナがどのように協調的に振舞うべきかを記述したものをスクリプトという．スクリプトはインタフェース部と本体からなる．

[インタフェース部]

機能名 f を機能実現に必要なオブジェクト $e_i : V_i$ ($i=1,\cdots,k$, V_i は値域) とともに $f(e_1 : V_1, \cdots, e_k : V_k)$ のように表す．たとえば社員の採用は hire_emp (eName : string, aSalary : integer) で表す．

[本体]

スクリプトの本体は，機能の実現に参画するすべてのパートナについて，その振舞いを記述したものである．パートナの振舞いは状態間制約と遷移制約にしたがうものでなければならない．本体の表現には状態表現とアクション表現がある．

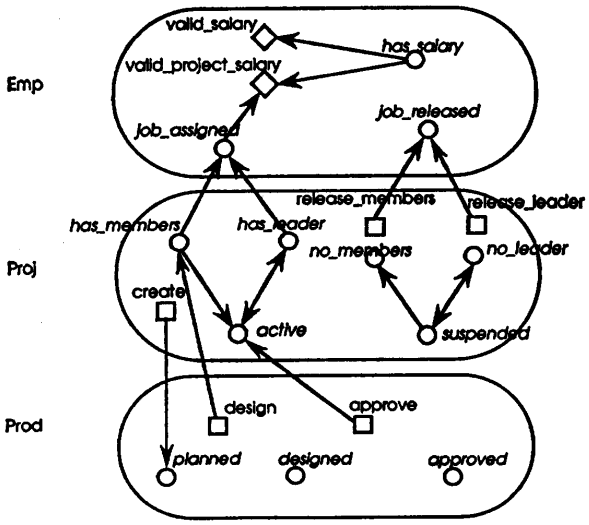

図 7 プロジェクト管理サブシステムにおける制約関連図
Fig. 7 The constraints diagram in project management.

(1) 状態表現

状態表現はすべてのパートナについて，その事前状態と事後状態を指定した宣言的な記述である．参画するパートナを $e_1 : E_1, \cdots, e_n : E_n$，各パートナについて，パートナクラス E_i のライフサイクルスキーマにおいて定義された事前状態を u_i，事後状態を v_i とするとき，状態表現を次のように記述する．

prestate : state ($e_1 : E_1$, u_1), ⋯, state($e_n : E_n$, u_n);

poststate : state($e_1 : E_1$, v_1), ⋯, state($e_n : E_n$, v_n);

(2) アクション表現

アクション表現はどのパートナのどの事象が，どのような順に起動されるべきかを指定した手続き的な記述である．参画するパートナの事象起動の列を形式 $action_1 \rightarrow, \cdots, \rightarrow action_n$ で表す．各 $action_i$ はあるパートナ e : E に対する事象起動 [e : E, t] である．ただし t はパートナクラスEのライフサイクルスキーマにおいて定義された事象である．

[例6] プロジェクト管理サブシステムにおけるスクリプトの例をあげる．

(1) プロジェクト p : Proj にリーダ newLeader : Emp を任命し，p を状態 *active* にするスクリプトは次のように記述される．

インタフェース部：

activate_project(p : Proj, newLeader : Emp);

状態表現：パートナの事前状態，事後状態が状態間制約にしたがうよう次のように記述される．

prestate : state (p : Proj, *exist* (*suspended*, *no_leader*));
state (newLeader : Emp, *job_released*);

poststate : state (p : Proj, *exist* (*active*, *has_leader*));
state (newLeader : Emp, *job_assigned*);

アクション表現：状態表現から容易に次の記述が得られる．

[p : Proj, activate]→[p : Proj, assign_leader (newLeader : Emp)]
→[newLeader : Emp, assign_to_project (p : Proj)]

(2) プロジェクト p : Proj の請け負う製品 d : Prod (ただし d=task(p)) を *designed* の状態にするスクリプトは次のように記述される．

インタフェース部：

design-product (p : Proj)

状態表現：パートナが遷移制約にしたがうよう次のように記述される．

prestate : state (p : Proj, *has_members*);
state (d : Prod, *planned*); "d=task (p)"

poststate : state (d : Prod, *designed*);

アクション表現：状態表現から次の条件付き事象起動が得られる．

if state (p : Proj, *has_members*) then [d : Prod, design]; "d=task (p)"

図8はプロジェクト管理システムにおけるコントラクトで，状態間制約，遷移制約，およびスクリプトから構成されている．ここでスクリプトについては状態表現のみが記述されている．

コントラクトは，サブシステムにおけるパートナの振舞いの協調に関する制約と，サブシステムが請け負う機能ごとのスクリプトの記述であり，パートナの振舞いに関する仕様を提供するものであるが，そのインプリメンテーションは一意的ではない．

一般に，事象起動はオブジェクトへのメッセージ送受の形で実現される．メッセージを送るオブジェクトをクライアント，受けるオブジェクトをサーバとよぶ．コントラクトに記述されたスクリプトのインプリメンテーションは，スクリプトのアクション表現を構成する各事象起動について，クライアントとサーバの役割を適当なオブジェクトに委ねることによってなされる．これをスクリプトの委任 (delegation) とよぶ．

スクリプトの委任は一意的とはかぎらない．もっとも単純な委任の方法は，サブシステムにおいて特権的なオブジェクトを一つ設け，そのオブジェクトにすべ

てのスクリプトに関するクライアントの役割を委ね，他のすべてのパートナをサーバとすることである．また

```
Contract Project_Management with partner classes Emp, Proj, Prod ;
 inter state constraints
  state (p:Proj, active) requires state(p:Proj, has_leader);
  state(p:Proj, has_leader) requires state(p:Proj, active),
        state(e:Emp, job_assigned);   "e = leader(p)"
  state(p:Proj, has_members) requires state(p:Proj, active),
        state(e:Emp, job_assigned) "e is any element of members(p)"
  state(p:Proj, suspended) requires state(p:Proj, no_leader),
                                    state(p:Proj, no_members);
  state(p:Proj, no_leader) requires state(p:Proj, suspended)
 transition constraints
  (Proj create) requires state(d:Prod, planned);
                "d is the value of task for the created object of Proj"
  (p:Proj, release_leader) requires state(e:Emp, job_released);
                                                "e = leader(p)"
  (p:Proj, release_members) requires state(e:Emp, job_released);
                                "e is any element of members(p)"
  (d:Prod, design) requires state(p:Proj, has_members); "d = task(p)"
  (d:Prod, approve) requires state(p:Proj, active);  "d = task(p)"
 scripts
 hire_emp(eName:string, aSalary:integer);
  prestate  : state(e:Emp, not_exist);
  poststate : state(e:Emp, exist(has_salary, job_released));
 fire_emp(e:Emp);
  prestate  : state(e:Emp, exist(has_salary, job_released));
  poststate: state(e:Emp, not_exist);
 create_project(pName:string, newLeader:Emp, dName:string);
  prestate :  state(newLeader:Emp, job_released);
  poststate : state(p:Proj, exist(active, has_leader, no_members));
              state(newLeader:Emp, job_assigned);
              state(d:Prod, planned);      "d = task(p)"
 terminate_project(p:Proj);
  prestate :  state(p:Proj, exist(suspended, no_leader, no_members));
  poststate : state(p:Proj, not_exist);
 suspend_project(p:Proj);
   prestate : state(p:Proj,exist(active, has_leader));
   poststae : state(p:Proj, exist(suspended, no_leader, no_members);
              state(e:Emp, job_released);   "e = leader(p)"
              for all e' of members(p) (state(e':Emp, job_released));
 activate_project(p:Proj, newLeader:Emp);
   prestate :  state(p:Proj, exist(suspended, no_leader));
               state(newLeader:Emp, job_released);
   poststate : state(p:Proj, exist(active, has_leader);
               state(newLeader:Emp, job_assigned);
 change_project_leader(p:Proj, newLeder:Emp);
   prestate :  state(p:Proj, exist(active, has_leader));
               state(e:Emp, job_assigned);     "e = leader(p)"
               state(newLeader:Emp, job_released);
   poststate : state(p:Proj, exist(active, has_leader));
               state(e:Emp, job_released);
               state(newLeader:Emp, job_assigned);
 assign_project_members(p:proj, newMembers:set of Emp);
   prestate :  state(p:Proj, exist(active, no_members));
               for all e of newMembers (state(e:Emp, job_released));
   poststate : state(p:Proj, exist(active, has_members));
               for all e of newMembers (state(e:Emp, job_assigned));
 release_project_members(p:Proj);
   prestate :  state(p:Proj, has_members);
               for all e of members(p) (state(e:Emp, job_assigned));
   poststate : for all e of members(p) (state(e:Emp, job_released));
               state(p:Proj, no_members);
 design_product(p:Proj);
   prestate :  state(p:Proj, has_members); state(d:Prod, planned);  "d = task(p)"
   poststate : state(d:Prod, designed);
 approve_product(p:Proj);
   prestate :  state(p:Proj, active); state(d:Prod, designed);  "d = task(p)"
   poststate : state(d:Prod, approved);
end Contract Project_Management;
```

図 8 コントラクト Projedt_Management の記述
Fig. 8 The description of contract Project_Management.

たクライアントの役割を，いくつかのパートナに分担させる方法も考えられる．スクリプトの委任の問題は今後の検討課題である．

5.3 振舞いの制約とスクリプトの継承

クラス P_i ($i=1, \cdots, n$) をパートナクラスにもつサブシステムにおけるコントラクトを C ($P_1, \cdots, P_n$) で表す．Q_i を P_i のサブクラスとするとき，各 P_i の一部または全部を Q_i に置き換えて得られるコントラクト C ($R_1, \cdots, R_n$) (R_i は Q_i または P_i のいずれか) を C ($P_1, \cdots, P_n$) のサブコントラクト，C ($P_1, \cdots, P_n$) をそのスーパコントラクトという．

各サブクラス Q_i は，事象の洗練を除いてスーパクラス P_i と同じライフサイクルスキーマをもつと仮定する．このときサブコントラクトにおいては，各サブクラス Q_i の事象の洗練によって得られた新たな状態および事象に関する振舞いの制約が付加されることを除いて，スーパコントラクトと同じ振舞いの制約をもつ．このことを，サブコントラクトはスーパコントラクトの振舞いの制約を継承するという．さらにサブコントラクトのスクリプトについては，アクション表現における事象起動の洗練を除いてスーパコントラクトと同じスクリプトをもつ．このことを，サブコントラクトはスーパコントラクトのスクリプトを継承するという．

ここに事象起動の洗練とは，スーパコントラクトのスクリプトのアクション表現に事象起動 [p_i : P_i, t] が含まれ，かつ事象 t が Q_i において事象の列 $t_1, \cdots, t_k$ とそれぞれの事前状態，事後状態に洗練されるとき，サブコントラクトの対応するスクリプトにおいて [p_i : P_i, t] を事象起動の列 [p_i : P_i, t_1]→…→[p_i : P_i, t_k] に置き換えることをいう．

[例 7] コントラクト Project_Management，すなわち C (Emp, Proj, Prod) のサブコントラクト C (Emp, Proj, S_Prod) は，C(Emp, Proj, Prod) のスクリプトを継承する．このとき図 3 (b) から明らかなように，[例 6] にあげたスクリプト design_product のアクション表現における事象起動 [d : Prod, design] は，事象起動の列 [d : Prod, document]→[d : Prod, review] に洗練される．

6. おわりに

本論文では，オブジェクト指向設計におけるオブジェクトの振舞いに関する概念設計の基礎として，振舞いのモデルとコントラクトの設計法を提案した．これ

は次のように要約される．

(1) オブジェクトの振舞いを表現する手段として，ライフサイクルスキーマを定義した．さらにその構造化のためにライフサイクルスキーマ洗練の概念を導入した．

(2) サブシステムにおけるオブジェクトの協調的な振舞いに関する一貫性制約を，状態間制約および遷移制約として定義し，制約関連図として表現した．

(3) サブシステムにおけるパートナの振舞いの協調を記述する手段としてコントラクトの概念を定義し，コントラクトを振舞いの制約とスクリプトの記述によって構成する方法を提案した．

振舞いモデルはオブジェクト指向設計の中心課題として，さらに発展させる必要がある．とくにオブジェクトの多様な抽象化構造に対応する振舞いの抽象化構造の詳細な分析，コントラクトの抽象化構造概念の精密化，スクリプトのオブジェクトへの委任に関する設計法，および振舞いに関する制約の形式記述の高度化は今後に残された課題である．

参考文献

1) 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*, pp. 40-57 (1989).
2) Bloom, T. and Zdonik, B.: Issues in the Design of Object-Oriented Data Base Programming Languages, *Proc. Object-Oriented Programming Systems, Languages and Applications 87*, pp. 441-451 (1987).
3) Borgida, A., Mylopoulos, J. and Wong, H. K. T.: Generalization/Specialization as a Basisi for Software Specialization, *On Conceptual Modeling*, pp. 87-114, Springer-Verlag (1986).
4) Brodie, M. L. and Ridjanovic, D.: On the Design and Specification of Database Transactions, *On Conceptual Modeling*, pp. 277-306, Springer-Verlag (1986).
5) Helm, R., Holland, I. M. and Gangopadhyay, D.: Contracts: Specifying Behavioral Components in Object_Oriented Systems, *Proc. Object_Oriented Programming: Systems, Languages, and Applications '90*, pp. 169-180 (1990).
6) Hull, R. and King, R.: Semantic Database Modeling: Surveys, Applications, and Research Issues, *ACM Comput. Surv.*, Vol. 19, No. 3, pp. 201-260 (1987).
7) Kappel, G. and Schrefl, M.: Object/Behavior Diagrams, *Proc. 7th Int. Conf. on Data Engineering*, pp. 530-539 (1991).
8) Mattos, N. M.: Abstraction Concepts: The Basis for Data and Knowledge Modeling, *Proc. 7th Int. Conf. on Entity-Relationship Approach*, pp. 473-492 (1988).
9) Mees, M. and Put, F.: Extending a Dynamic Modeling Method Using Data Modeling Capabilities: The Case of JSD, *Proc. 5th Int. Conf. on Entity-Relationship Approach*, pp. 399-418 (1986).
10) Mylopoulos, J., Bernstein, P. A. and Wong, H. K. T.: A Language Facility for Designing Database-Intensive Applications, *ACM Trans. Database Syst.*, Vol. 5, No. 2, pp. 185-207 (1980).
11) Peckman, J. and Maryanski, F.: Semantic Data Models, *ACM Comput. Surv.*, Vol. 20, No. 3, pp. 153-189 (1988).
12) Peterson, J. L.: *Petri Net Theory and the Modeling of Systems*, North-Holland (1981).
13) Put, F.: The ER Approach Extended with the Action Concept as a Conceptual Modeling Tool, *Proc. 7th Int. Conf. on Entity-Relationship Approach*, pp. 423-440 (1988).
14) Rebecca, J., Wirfs-Brock, R. and Johnson, R.E.: Surveying Current Research in Object-Oriented Design, *Comm. ACM*, Vol. 33, No. 9, pp. 104-124 (1990).
15) Robinson, R. A.: An Entity/Event Data Modeling Method, *Comput. J.*, Vol. 22, No. 3, pp. 270-281 (1979).
16) Sakai, H.: A Method for Entity-Relationship Behavior Modeling, *Proc. 3rd Int. Conf. on Entity-Relationship Approach*, pp. 111-129 (1983).
17) Sakai, H. and Horiuchi, H.: A Method for Behavior Modeling in Data Oriented Approach to Systems Design, *Proc. 1st Int. Conf. on Data Engineering*, pp. 492-499 (1984).
18) Sakai, H.: An Object Behavior Modeling Method, *Proc. 1st Int. Conf. on Database and Expert Systems Applications*, pp. 42-48 (1990).
19) Schrefl, M.: Behavior Modeling by Stepwise Refining Behavior Diagrams, *Proc. 9th Int. Conf. on Entity-Relationship Approach*, pp. 119-134 (1990).

(平成3年9月19日受付)
(平成4年6月12日採録)

酒井 博敬（正会員）

昭和36年京都大学理学部理学研究科修士課程（数学専攻）修了．同年日立製作所，昭和58年日立ソフトウェアエンジニアリング，昭和59年京都産業大学教授，現在中央大学理工学部管理工学科教授．工学博士．研究テーマ：オブジェクト設計方法論．著書「情報資源管理の技法」，「オブジェクト指向入門」．電子情報通信学会，ACM，IEEE 各会員．