情報システムにおけるリアリティのモデル化

田村 浩一郎　koichiro Tamura
中京大学情報理工学部情報システム工学科 教授

情報システムにおけるリアリティのモデル化

中京大学情報理工学部　　田村浩一郎

１．はじめに

ここでいう情報システムとは，情報のサービスを提供するシステムのことである．情報のサービスとは，情報へのなんらかの要求を受け，それに答えるべく適切な処理を施し，得られた情報を応答として提供することである．２１世紀に入ってから確実にいえることは，このような情報システムが全世界の社会基盤を形成し，その善し悪しが文明の健全性を決定的に左右するほどに至っていることである．さらに具体的に言えば，その情報システムの大半がインターネットを基盤とするウェブアプリケーションシステムになってきているということである．すなわち，情報サービスを求めるクライアントと情報サービスを提供するサーバとがウェブ（World Wide Web)のプロトコル体系の上に載っている，ということであり，この状況は今後数十年は変わらないのではないかと考えられる．

この現実を踏まえた上で，あらためて情報システムを構築する技術を見直すと，実は問題が山積している．セキュリティなどは，目に見えやすく，誰にとってもわかりやすい問題の一つであるが，ずっと深刻な問題が目に見えにくい深層に潜んでいる．実世界すなわちリアリティのモデル化の問題である．

人類がウェブ上に巨大な情報世界を急速に築きつつある現在，もっとも恐れなければならない問題は，その情報世界と実世界（リアリティ）との乖離である．実世界の正しい描写，写像が出来てこその情報世界であり，それが信頼出来ないものになるならば，この巨大な情報世界は存在基盤を失い、単なる記号の集積と化す．一種の情報世界である経済が，その情報への信頼を失うことによって深刻な危機を招いているが，同様の事態がインターネット上の情報世界全般に広がりかねないのである．

では，われわれは，実世界を忠実に情報化する，すなわち，何らかの方法で記述する，言い換えれば，モデル化する技術を確かなものにしているのだろうか．プログラミング，データベース，あるいは，人工知能で言う知識表現など，いずれもリアリティのモデル化の手法である．過去数十年にわたって情報科学の研究はこの一点を中心にして展開されてきた，といっても過言ではないだろう．ところが，それらをあらためて基盤から見直すとき，実は多くの怪しげな点が未だに残されていることに気がつく．

本稿では，以上の意味での情報システムにおけるリアリティのモデル化について論じ，２０世紀に発達した認識論，記号論を踏まえた上で，数学の圏論 category theory をベースにしたモデル化の理論を構築し，それに基づきあらたに開発した実践的手法について述べる．

２．情報システムアーキテクチャ

現在の情報システム，特にウェブアプリケーションシステムの構築においては，MVC アーキテクチャが標準的に採用されつつあると言ってよいであろう．もちろんさまざまなアーキテクチャが考えられるし，なかにはアーキテクチャアなど意に介さずに（あるいは知らずに）作られるシステムも多いが，情報システム構築の長年の経験の中で，このアーキテクチャがもっとも自然で，分業による構築や構築後の改良，拡張などの進化の段階においてもその有効性が認められてきていると考えられる．

MVC アーキテクチャとは，Model (M)， View (V)， Controller (C) の三部構成のことをいう（**図 1**）．クライアントからの要求がシステムに届くと，まず C の部分で解釈され，必要に応じて M の機能を呼び出して，得られたデータを V に渡す．V は C から渡されたデータをもとに応答情報を作成しクライアントに送る．

ここで重要な役割を果たすのが M である．モデル部 M はまさに，情報システムが対象とする世界をモデル化するものだからである．

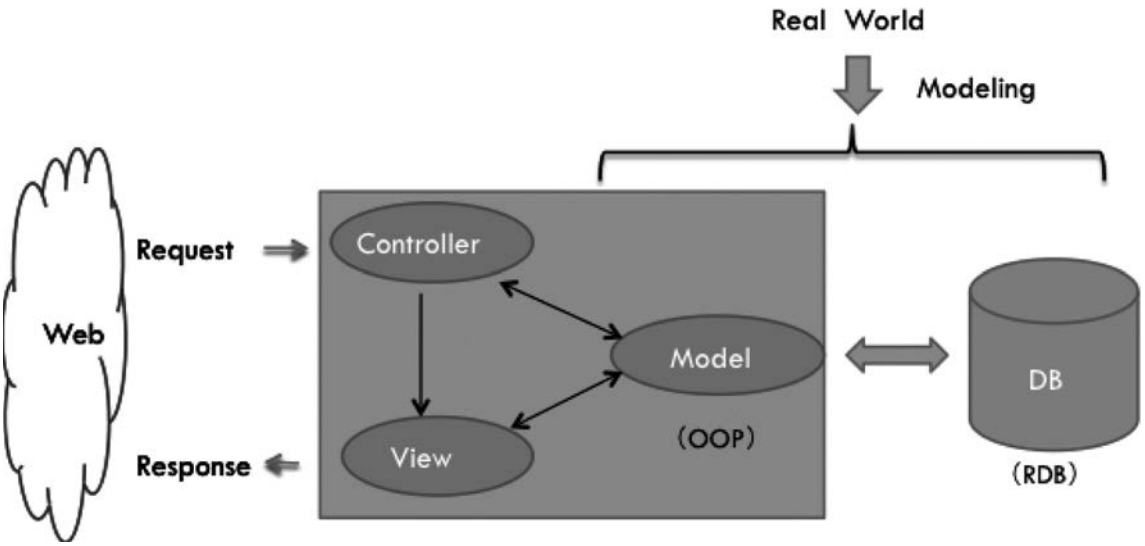

図 1　MVC アーキテクチャ

ほとんどのシステムで，モデル部にはデータベースシステム(DB)が付けられ，実世界のデータを構造化し格納するのに利用される．ここに実世界の事実とされるデータが収められる．情報システムの歴史的発展の経緯からDBは独立したシステムとして作られ，本体のモデル部とはデータベース言語によって情報のやりとりが行われる．本体のモデル部の構築にはしばしばオブジェクト指向言語(OOP)が用いられ，データベースには関係データベース(RDB)が使用され，その言語はSQLである．

本来，情報システムは一パラダイム，一言語で一貫して構築されるべきであるが，残念ながら現在においては，歴史的経緯を引きずったまま，複数言語，複数パラダイムの混成として作られる．その状況の中にあって，モデル部のOOPとRDBは，使用言語が異なるものの，そのパラダイムとしては，実は，相性がよく，しばしば両者を一体化させるべく様々な補助概念とそれを技術的に裏付ける補助手段が考えられている．本稿で想定するActive Recordの考え方もその一つである．

以上をまとめると，本稿が前提とする情報システムの構築法は次のようになる．

（1）MVCアーキテクチャを採る．

（2）そのうちのモデル部が最も重要で，実世界を写像し，記述するものである．

（3）モデル部はOOPによるプログラミングの部分とその管理下にあるRDBにより構成され，OOP部で情報が処理され，RDB部で実世界の事実を示すデータを格納する．

（4）OOPによるモデル部とRDBとの結合はActive Recordの概念による．

3．これまでの実世界のモデル化と基本的考察

モデル化とは対象世界の構造とデータを記述することである．従来，情報処理の様々な分野で実世界のモデル化の手法が開発されてきた．例えば，

（1）人工知能：　意味ネットワーク，プロダクションシステム，フレーム，述語論理，オントロジー

（2）データベース：　関係スキーマ，E-R図

（3）ソフトウエア工学：　抽象データ型，UML

（4）プログラミング言語：　手続き型，関数型，宣言型，論理型，オブジェクト指向型(OOP)

（5）一般論として：　形式的概念分析[1]，概念グラフ

などなど．

このように，実世界のモデル化は情報処理技術，あるいは情報科学の中心的課題でもあっ

[1] [DS05]参照のこと．

た．また，用語や見かけは異なるものの，その意味がほとんど同じであるものも多い．たとえば，オブジェクト指向パラダイム (OOP)，人工知能で言うフレーム，それにソフトウエア工学の抽象データ型などは，個別の何かを想定し，それについての記述をモジュールとする点ではほとんど同一の概念構造と見ることが出来る．また，述語論理が論理型プログラミングの基盤になっていることは当然として，RDB のスキーマが関係概念を軸とする一種の論理的言明になっており，本質的には共通した概念を基礎においていると言ってよい．事実，多くの具体例において，相互の書き換えさえ可能である．

モデル化の基本は，世界の認識の仕方である．世界をどのように認識するか，その視座が定まって始めてその記述が可能になる．記述をしばしば「表現」という．かつて人工知能では「知識表現」が重要であると言われ続けてきたが，実はその表現方法，つまり記述法よりも，認識の視座が重要であり，実際，表現方法という見かけでの相違はあってもしばしば本質が同一になるのは，認識の視座が同根だからである．

心理学や認知科学，あるいは近年の脳科学では人間の認識過程がどのようなものかについて論じられるが，人間も世界の一部であるから，人間の認識過程を分析，解明するという行為そのものがすでに世界認識の視座を前提としている．「どのように行っているのか」ではなく，「どのような」立場，観点で世界認識を行うことが有効なのか，それが問われるのである．簡明で数少ない部品概念を矛盾無く組み立てることによって一貫性のある世界の写像，描写を得られるかどうか，これが，認識の視座が有効かどうか，の判断基準となる．つまり，認識の視座は解明するものではなく，考え出すものである．このようにして作り出されたものが，上述した情報科学におけるさまざまなモデル化手法である．幸いにもコンピュータプログラミングとして実装することにより，それらの手法が実践的に適用できるため，さまざまな分野においてその有効性を検証し，必要があれば修正，改良することが出来る．これまでの半世紀以上にわたる情報処理技術の経験と進化は，つまりはこのような弁証論的発展の過程であったと考えることが出来る．

４．Peirce の三つ組

世界をどのように見るか．この命題は，古代ギリシャ，インド，中国から始まり現代に至るまで，重要な哲学的命題である．この命題自体が自覚されるとき，それはすでに言語化され，さらにその思考過程や回答も言語化される．すなわち，問いから答えに至るまで記述される，つまりモデル化される．この自己参照によってある種の堂々巡りが始まり，しばしば思考の迷路にはまり込むこととなる．この堂々巡りは決して乗り越えられないが，かろうじて明晰さを保つ強靱な思考体系が時空を越えて生き残り伝えられる．そのなかで，ここでは Peirce の三つ

組 Triad[1]を取り上げることにする．三つ組は，世界を分割し，分類する(カテゴリーに分ける)観点をメタな立場から分析するもので，（１）１者 (1stness)（２）２者 (2ndness)（３）３者（3rdness）から構成される．１者は，２者あるいは３者への参照を必要としない自立したもの，２者は，１者への参照を考慮してはじめて意味あるもの，３者は，２者の１者への参照の仕方を与えるものであるとする．１者１に対し，２者は多（０を含む）である．この三つ組の関係の組み合わせによって世界を認識し，解釈する．その通底にはおそらくキリスト教の 三位一体 Trinity があるのであろう．

三角形はそれを組み合わせることによって面を覆いながら，無限に広がることが出来る．三角形を上に接続するならば，その２者が下の三角形の１者となる．また，３者は他の三角形の１者とも２者ともなり得る．

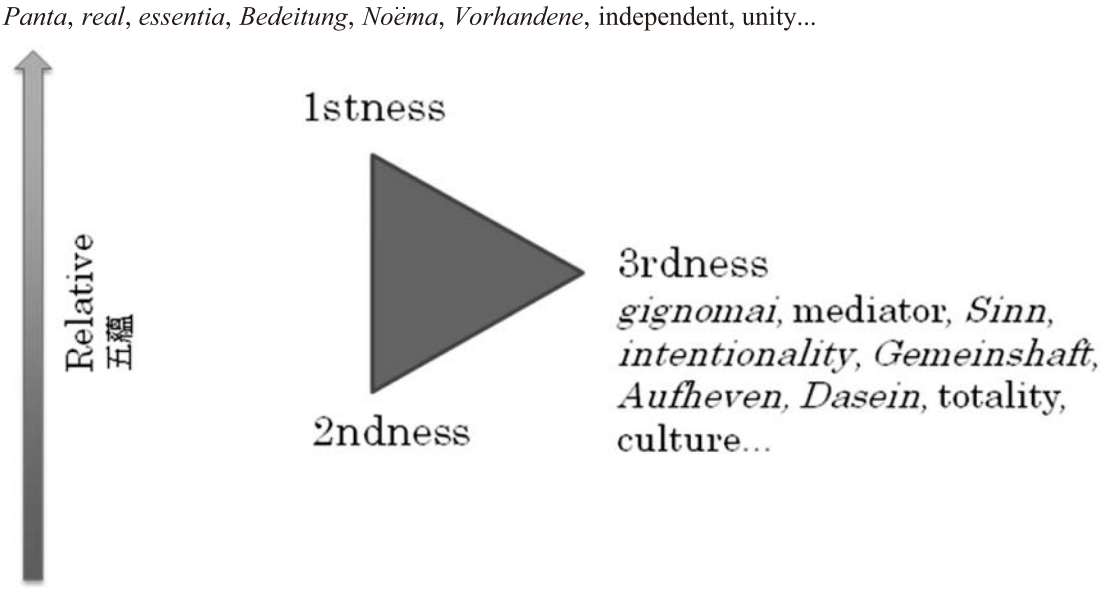

図 2　Peirce の三つ組と様々な基本概念[2]

三つ組を抽象と具象，イデアールとレアルの狭間に置くと，１，２，３者の関係の相対性を見ることが出来る（**図 2**）．この相対空間の中に古代ギリシャ哲学，仏教哲学，フレーゲ，ヘーゲル，フッサール，ハイデッガーなどの認識論，存在論におけるキーワードを，きわめておお

[1] 主として[So00]を参考にした．

[2] [So00]および[熊野 06] [熊野 07]を参考にした．

ざっぱなものにしろ，図式に配置するとこのようになるだろう．また，廣松[1]のいう「所与，所識」の対は所与を１者，所識を２者とする３者であると考えることが出来よう．

図 2において，２者から１者への依存性（参照）を矢印で示すと，主観が客観を目指す図式になる．われわれは認識者として主観的に実世界を見るが，たえず客観をめざし，実世界を正しく把握しようとする．しかし，その実世界には決して到達しえない．そのことは，カント他ほとんどすべての哲学者が指摘しているとおりであるが，「語れぬ事」を「語ろうとする」ことこそが知の地平を目指すことであるから，そこに迫る必死の努力が「哲学」であるのかも知れない．後に述べるように，この方向性（矢印）が類（概念）の属性であり，なにか（世界の「部分」）の「記述」あるいは「述語」である．つまり，なにかを語ること自体が，到達し得ないリアリティに接近する行為である．このことを承知の上で，では，より具体的にどのように語るべきか，それがモデル化の課題である．

MVC アーキテクチャも三つ組の好例である（**図 3**）．Model は１者としてただ一つの存在であるが，View は２者として一つの Model に複数個あり得る．これは REST（Representational State Transfer）タイプのアーキテクチャでいうところの representation に相当する．すなわち，（HTML，XML，その他）複数の表示形式があり得る，ということである．両者をつなぐのが Controller である．

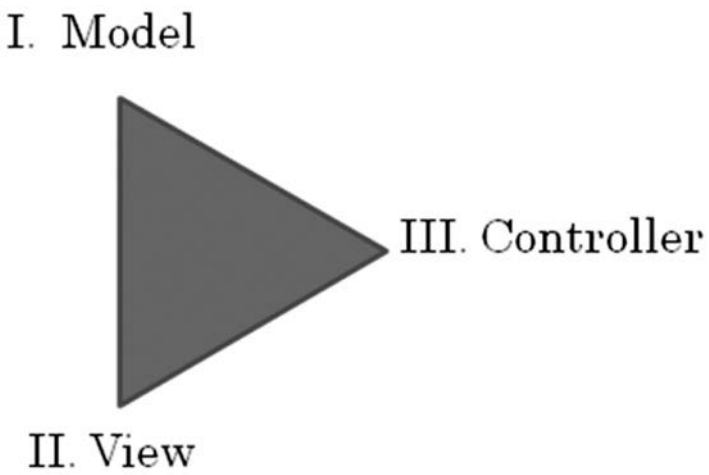

図 3　三つ組としての MVC アーキテクチャ

[1] [廣松 82]

５．世界をどのように見るか　ー　素朴な世界観

さて，このようにわれわれは推論を交えながら世界認識をするが，その認識の視座について，素朴な観点から見てみよう．

（１）**分割**：　ひとは，五感によって得る情報，あるいは思考がもたらす情報をモノ，コトとして**分割（分節）segmentation**する[1]．分割された個々のものを**実体　entity**と呼ぶ．実体は，唯一かつ不変である．また，実体は互いに**関わり　reference**を持つ[2]．

（２）**分類**：　互いに似ていると認識される実体をまとめ，**分類　classification**する．分類したものを**類　concept**と呼ぶことにする．どの実体もその類はただ一つ指定され，実体をその類の**実例　instance**と呼ぶ．

（３）**属性**：　類Xの実例xが類Yの実例yに直接の関わりaが認められるとき，aを類Xの類Yへの**属性　attribute**という．このとき，yを，xの属性aの値（**属性値　attribute value**）であるという．

（４）**状況**：　属性は，**状況　situation**によって変動する．

（５）**社会性**：　人々は共通の認識構造により互いに同類の認識を行うものとする[3]．そして，このような共通認識が得られる（はずの）認識対象を実世界（リアリティ）とする．

類と属性の例を**図 4**に示す．

６．類と属性

ここで，類 concept は，類似の実体をまとめるものと言う意味での便宜的な語で，ほかに様々な呼び名がある．たとえば，“idea”，“*eidos*”，“*genos*″，“category”，“paradigm”，“collection”，“set”，“pattern”，“kind”，“name”，“label”，“class”，“type”，“sort”などである．さらに，**記述対象世界　universe of discourse**とも呼ばれる．まさに「なんと呼ばれようと，バラはバラ．その甘い香りに変わりない」（シェークスピア）．通常はそれを表示する（意味する，あるいは代替する）語ないし記号が与えられるが，必ずしもそうとは限らず，適切な語ないし記号が欠ける場合もある．語ないし記号はあくまでも人々の間で概念，

1　分割以前の世界（認識以前のリアリティ）は，空，混沌，あるいはカオス Chaos とよばれる．これが究極の１者であろう．キリスト教の Trinity における God も当然１者であり，宇宙に遍在しかつ唯一者であることから，これらの語とおなじ認識を持たれているのではないだろうか．実際，A. N. Whitehead はこう述べている（[So00]p.63による）"God is an actual entity, and so is a most trivial puff of existence in far-off, empty space."

2 仏教でいう「縁」である．

3 もしそうでなければ，言語や記号は成立しない．

考えを共有し理解するための手がかりを与えるものであって，それ以上のものではない．しかし，ひとびとはしばしば語ないし記号自体が自立的に自存するかのような錯覚にとらわれ，判断を誤らせる．

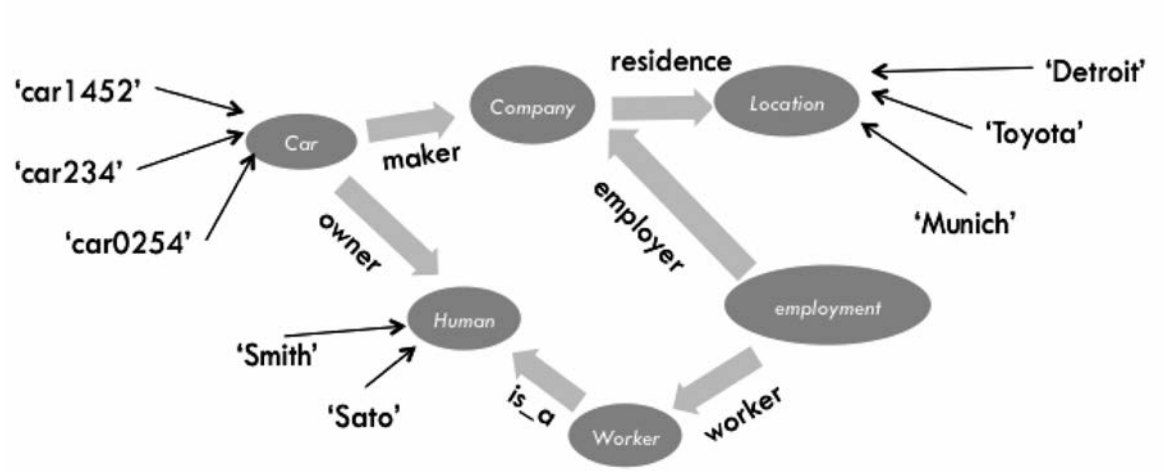

図 4　類，属性，実例の例

特殊な類として，**空類**　Nil と**単独類**　singleton がある．空類は実例を持たない類，単独類は実例をただ一個のみ持つ類であり，いずれも概念的世界構成において重要な役割を果たす不可欠な概念である．実例を持たない類というのは奇妙な感じを抱かせるかもしれないが，ちょうど数のゼロ，あるいは数学の空集合のようなものである．一方，単独類の例は多く，たとえば，「太陽」「次期米国大統領」のように，英語での表現では“the”が付けられるものである．

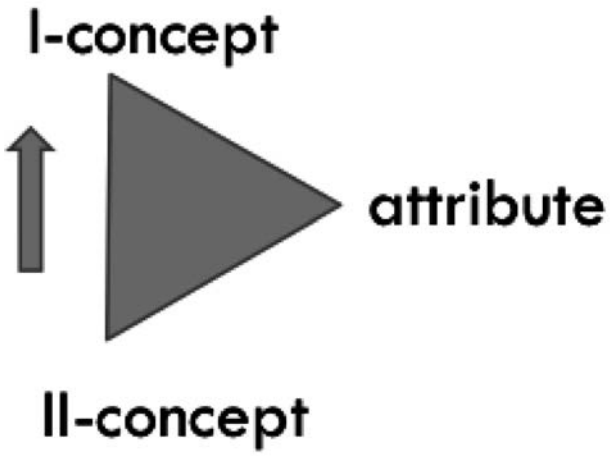

図 5　類 concept と属性 attribute

「属性　attribute」にも様々な呼び名がある．たとえば，“feature”，“property”，“aspect”，“coordinate”，“factor”などがある．属性は類の実例を別の（あるいは自分の）類の実例に結びつける（参照する）仕方を規定するものであるが，これにもいろいろな言い方がある．たとえば，“associated with”，“correspond to”，“point to”，“linked to”，“related to”，“relevant to”，“belong to”あるいは“depend on”，さらには，“transfer to”や“aspect of”などがある．いずれも，これらを三つ組に位置づけてみれば，同じ意味を表していることになり，基本概念は同一であると見ることが出来る（**図 5**）．

属性とは，類Ａから類Ｂへの「関わり方」，すなわち，関わる類の対を指定しているだけであるから，類Ａの実例と類Ｂの実例の組み合わせだけの数の具体的関わりがある．この具体的関わりの総称を属性と呼んでいることに注意が必要である．特に，状況によって「属性が変わる」，という場合，この具体的組み合わせが変ることを意味する．後に圏論によって正確な定義を与える．

すべての実例について自身への関わりを与えるものも一種の属性と見なし，**自己属性　self attribute**とよぶ．自己属性以外の属性が与えられない類を**素類　prime concept**[1]と呼ぶ．たとえば，「色」という類の実例として，「赤」「白」などがあるとし，それぞれの色についてそれ以上の（スペクトルなどの）分析をしないとき，類「色」は素類である．

類のある属性について，属性値が与えられない実例を持つ場合がある．そのような属性は類の部分にたいしてのみ定まるため，「**部分属性　partial attribute**」と呼ぶ．値が与えられない場合，コンピュータ科学では，その「値」が「未定義」，「未知」，“Don't care”，“N/A（not available)”，あるいは，“garbage”などといい，しばしばnil，またはnullまたは ＊ で表示する．ただし，これは「無　nothing」という値ではないことに注意．たとえば，長さ0の文字列は一種の「無」であるが，そういう値が与えられているのであって，ここでいうnilではない．この問題をどう扱うか，認識論，ないし，それに関する理論体系の基礎をなすものであり，厳密な議論を必要とする．

７．状況によって変動する実体

実体　entityは，唯一不変のものである．そして，状況によって変動して見えても同一物であると認識されるのが実体である．たとえば，ひとはだれも年を取り，いつかは死ぬが，しかし，「変わりなく」誰それである，と継続して認識されるのが，ある特定の「ひと」という実体

[1] Leibnizは同様の概念をprimary conceptと呼んでいる（[So00]による）．

である．この変動は，時間に沿った変化であるが，見方によっても同一のものが違って見えるという変動がある．まず，物理的に視角が異なれば違って見えるし，時には全く見えなくなって再び見えるようになることがあっても，同一人物であるというように同一視する．見る人が異なればやはり同一人物が異なって見えるが，それでも互いに同一人物を見ていることを無意識に，あるいは意識的に了解している．あるモノを二人が見ているとき，その同一のモノの見え方はそれぞれ全く異なるにもかかわらず，互いに同じ同一のモノを見ているはずであると無意識に考え，ときに確認し合い，認識を共有しようとする（いわゆる３項関係）．昨日の太陽と今日の太陽は異なるはずであるし，また，自分の見る太陽と他人の見る太陽と，全く別に見えるはずであるにもかかわらず，太陽（という実体）は世界に一つしかなく，不変であるとみなす．すなわち唯一性，普遍性を持つモノ，コトとして継続的に認識される（推論される）何かしらを実体と見なしている．

この変動を与える要因を「状況」と呼ぶ．状況は，時間，場所，観察者，など様々な因子を持つ．観察者が変われば同一物の属性が変わる例として興味深いのが Hilary Putnam の「双子の地球星人」のたとえ話[1]である．外観など観察だけによるとしたとき，地球原住民は水を構成する分子（属性である）を H_20 とするのに対し，この異星人は X_2Y とするであろう．この例は極端であるにしても，実体の評価など，観察者に依存する例は日常にあふれている．

この不変と変動をどのような構造のなかで考えるか．我々の立場は，ある状況と実体との対をその状況における実体の**状態　state** と考え，この状態の「変動」が実体の属性の値を変化させると考える．たとえば，ある時間 t（状況）におけるある物体 e（実体）の位置（属性値）はその物体の状態（対$\langle t, e\rangle$）によって変動する，と考えるのである．

情報システムとは実世界についての事実の情報処理とその記録であるから，その状況も同時に記録する必要があるのはこのような事情があるからであり，過去において，文書などによるきちんとした記録では当然のこととしてなされてきているにもかかわらず，現代の多くの情報システムでは，一貫した体系としてこの仕組みを構築することが必ずしもなされておらず，場当たり的に対処されるため，しばしば深刻なトラブルをもたらすことになる[2]．

実体の状態の時間的変動だけを見ても，その生成，更新，消滅がある．さらに複雑な変動として，実体の合併と分離がある．物品管理システムなどではこれらの変動を扱うことは必須である．組織についても同様であり，これらの変動は日常的に溢れている．

---

[1] なにもかも地球にそっくりの星から瞬間的に地球に移された人間（にそっくりなその異星人）がいる．その星ではただし，水の分子だけが地球のものと異なり，X_2Y で出来ている．その異星人は，分子以外は外見など全く同じ地球の水をみてどう考えるか．

[2] 社会保険庁はその極端な例であるが，多くの情報システムでこの問題が潜在していると考えられる．

８．各種術語の対応

ここで，われわれの用語と他の用語（術語）とのおおざっぱな対応付けを行う．

- *typed λ-calculus* ： concept → type,　attribute → N/A,　instance → variable or literal,　state→ N/A
- *Semantic Network* ： concept → concept,　attribute → link,　instance → N/A,　state → N/A.
- *Many sorted logic*： concept → sort,　attribute → N/A,　instance → variable or literal,　state → N/A
- *Unified Modeling Language*（*UML*）： concept → class,　attribute → attribute or link,　instance → instance,　state → N/A
- *E-R diagram* ： concept → entity,　attribute → attribute and link,　instance → N/A,　state → N/A
- *Relational database*： concept → domain or table,　attribute → table column,　instance → table row,　state → N/A.
- *Object oriented paradigm* ： concept → class,　instance → instance,　attribute → attribute,　entity → object,　state → N/A.

これらの対応は，われわれの用語のイメージを示すための便宜的なものであって，厳密なものではない．そもそもわれわれ自身の用語を含め，いずれも厳密な定義をしていないからである．

９．類の部分　ー　白馬は馬にあらず？

実体にはただ一つの類のみ指定されるとした．実体は唯一性を規定されているから，別な類の実例は実体として異なることになる．このように，本稿では，類は，universe of discourse とさえ呼ばれることを許容するくらい，互いに峻別されるものとして考えている．では，ある中国の論者が唱えたという「白馬は馬にあらず」という議論はどうだろうか．白馬を馬とは別の類とすればその通りになる．しかし，素朴な感覚では，白馬は馬という類の「部分」をなすものであるから，この議論は受け入れがたい．では，白馬は馬であるとして，白馬が何かしらの理由で栗毛になったらどうなるか．栗毛はもはや白馬ではないから，すると，馬でもなくな

るのだろうか．同様の議論が，「教師は人間である」という命題を巡って行われる．たしかに教師はみな人間である．では，ある教師が教師を辞めたとき，人間も止めるのであろうか．人工知能の素朴な意味ネットワークでは，is-a，あるいは part-ofというリンクが使用されたが，いずれも曖昧さを持ち，この種の議論に耐えることが出来ない．従って，その曖昧さを残したままで作られた情報システムでの情報処理，つまり「推論」は怪しいものになる．

この混乱のもとは，集合論の多くの書物で，「部分集合のどの要素も集合の要素である」と説明されることに起因する．これでは，部分集合の要素で無くなったときにはもとの集合の要素でもなくなってしまう，と解釈されかねない．集合論の説明によく使用されるベン図もこの解釈を助長している．これまでの多くの集合論では，このような基本的な「推論」さえ危うくなるのである．残念ながら現在のオントロジー[1]にも同様の混乱が残されているのは，かなり深刻な問題である．より基礎的で明確な論理と表現手法が必要になるゆえんである．

この「白馬」「教師」の問題についてのわれわれの立場は，次の通りである．「馬」，「人間」はそれぞれ「類」である．それに対し，「白馬」「教師」は，「馬」「教師」の実例にある条件が付けられた実例を要素とする類である．したがって，それらの実例は互いに異なっている（実体として別物である）が，相互に「ある条件」によって結ばれている．「部分」とは，このように異なる実体間の関わり方に関することである．より精密な議論をするならば，「部分」とは，後述する圏論でいう単射 monomorphism として表現できる属性の一種である．

１０．圏論による基盤構築

リアリティのモデリングの手法として，われわれは圏論　category theory を用いることにする．かつて圏論は理論的な厳密さに欠けるとか，あるいは応用性がないなどと批判的に見られていたが，現代において，圏論はすくなくとも数学の基礎理論を構築するためのもっとも優れた理論の一つではないだろうか．また，情報処理技術の理論に利用されることも多い．われわれの場合，圏の一種であるトポス（elementary topos）を利用する[2]．

圏は，**対象 object**[3]と**射 map**から構成される．射は二つの対象（**定義域 domain**と**値域 codomain**）を結び，方向性を持つ．三つ組で表せば**図 6**のようになる．

集合論では，対象である集合を要素の集まりと見なして，自他の集合の要素間の関係に注目するが，圏論では，対象の中身に立ち入らず，外側から相互関係を見る，すなわち，射どうし

---

[1] [SH05]を参考にした．

[2] 集合論を含め，圏論，トポスについては主として[G84] [LR03]を参考にした．

[3] OOP のオブジェクトとは用語上全く無関係である．

の関係のみに注目して対象世界の構造を分析する．圏論が重視するのは射であって，対象は射の結節点にしか過ぎない．たとえば，対象の「要素（一般化要素）」はその対象を値域とする射であり，「部分」も射（単射 homomorphism）として与えられる．また，直積も，（集合のような）対象の積ではなく射の積である．集合の中身とそれらの構造を自立的なものとして見る集合論的世界観と比較すると，図と地を反転させる違いがあり，それが圏論的世界観の根底になっている．

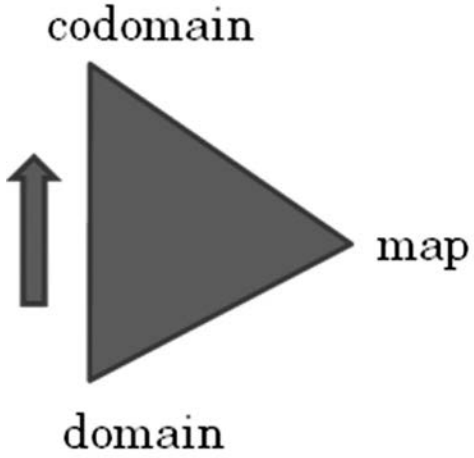

図 6　圏の三つ組．定義域 codomain から値域 domain への射 map

われわれの「素朴な世界観」を圏論の用語で定義するならば，次のようになる．

- 類は対象である．この対象を**類対象**と呼ぶ．
- 類対象 C の実例は C を値域とする射（**一般化要素 general element**[1]）である．
- 類対象 C の類対象 C' への射 $a : C \to C'$ を**属性射**といい，属性射の総称を**属性**という．**自己属性射**は自己射（1_C）である．どの属性射も自己属性射以外の複数の属性射が作る合成射ではない[2]．
- **状況** T は特別な対象である．
- 空類は，始対象 0 である．
- 単独類は，終対象 1 である．

---

[1] [LR03]による．可変要素 variable element ともいう．

[2] 類間の直接の関わり方を属性としたからである．

これらの対象と射を基礎として作る圏を *Concept* と呼ぶ．*Concept* はトポスを形成すると仮定することにより，（直）積 product，（直）和 sum，射対象（指数）map object (exponent)，部分対象 subobject，冪対象 power objectなど各種の合成対象，あるいは合成射を得ることが出来るものとする．

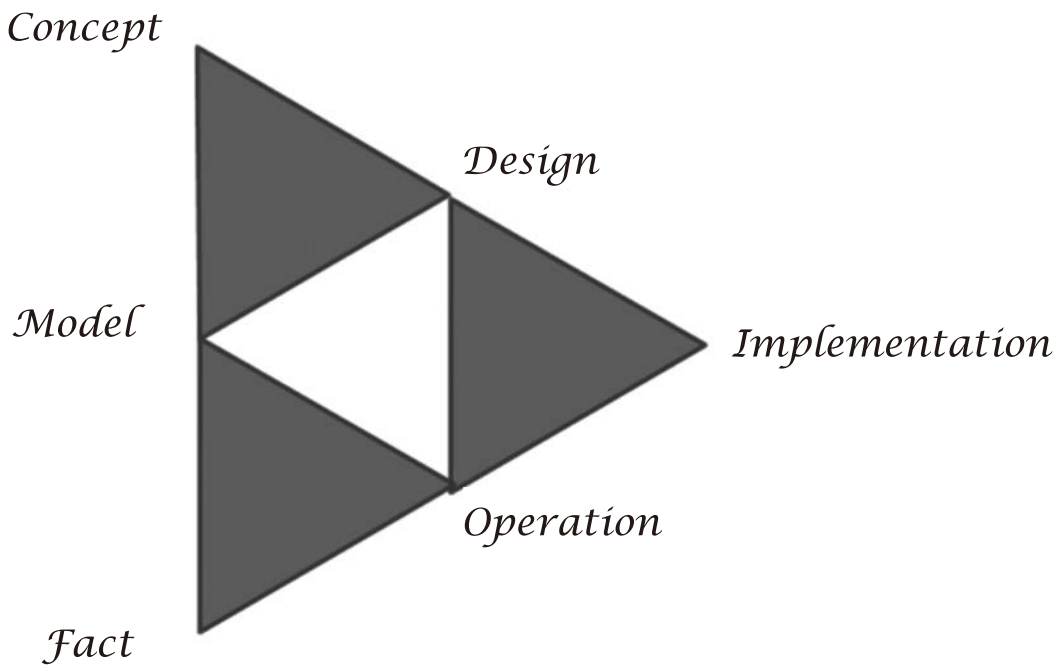

図 7　モデル化の操作過程

１１．類のモデルと事実

さらに具体化する作業によってモデル化を行う．その具体化作業の過程を三つ組で表せば，**図 7**のようになる．

まず，属性がどのような構成を取るか，詳しく見てみよう．類 C がある複数の類に（Peirce の三つ組の２者として）同時に関わりを持つと見なされるとき，圏論では，それを**属性射の積**によって表すことが出来る．たとえば，「社員」という類の属性として社員番号，名前，所属，地位などがその例になろう．属性射の積はその値域を類対象の積（$\prod_i C_i$ のように）で表す．一

方，複数の類に対し，（Peirce の三つ組の２者として）同時に関わることはなく，そのいずれかに関わると考えられる属性は，属性射の和で表すことが出来る．たとえば，「住所」は，「工場」，「顧客」，「社員」等のいずれに対しても「その住所」として共通に関わりを持つが，それぞれの「住所」はたがいに独立している．しかも，「住所」は関わり先（「顧客」など）の実例一つに対し，複数あり得る．このとき，「工場」，「顧客」，「社員」などをひとまとめにした類を想定し，それを値域とする属性射を「住所」の属性の一つとする．その値域は各属性射の値域の和（$\sum_i C_i$のように）で表す．

以上を合わせて，一般に，いずれの類 C にたいしても，その属性を束ねて，

$$C \xrightarrow{a} \prod_i \sum_j C_{ij}$$

の形で表すことが出来る．ここで，考える圏がトポスであるという仮定から，積和に関しての分配則が成り立つ．$\prod_i \sum_j C_{ij}$にあらわれる類の族 $\{C_{ij}\}_{i,j}$が C のすべての属性値域を尽くすとき，射 a を C の**属性束**と呼び，$\prod_i \sum_j C_{ij}$ を属性域と呼ぶ．こうして，C の属性射はいずれもある属性束の一つの成分 component となる．

類対象 C の属性域を A としたとき，射対象 A^C の単純要素 $1 \xrightarrow{\text{“}a\text{”}} A^C$ は射を指定し，この射は自然な形で属性束 $C \xrightarrow{a} A$ に対応する[1]．すなわち，A^C は C の属性すべてについての情報を持っていると考えられる．状況により属性が変動することは，次の射で表す．

$$T \longrightarrow A^C$$

この射は自然な形で次の射に対応する．

$$T \times C \xrightarrow{a} A$$

そこで，$S = T \times C$ とし，S を類 C の**状態**，射 a を**状態属性束**，射対象 A^S を類 C の**モデル model** と呼ぶ．その単純要素 $1 \xrightarrow{\text{“}a\text{”}} A^S$ は同様に自然な形で $S \xrightarrow{a} A$ に対応する．すなわち，モデルは，ある類について，状況により変動するその属性のすべてについての情報を持つものである．

モデルは**OOP のクラス class** によって表現できる．クラスのインスタンスは，状態 S の要素 $1 \xrightarrow{<t,e>} S = T \times C$ である．またモデルの単純要素を $a = <\ldots, a_i, \ldots>$ としたとき，その成分である属性射 a_i はクラスのインスタンス属性に対応する．

インスタンスメソッド h_i は，その入力を類 S_i とすれば，$h_i : S \to C_i^{S_i}$ で表される．インスタン

[1] 「自然な形で対応する」の意味については[LR03]p.98 参照のこと．表現は異なるものの，論理的に同等であるということ．

ス属性はインスタンスメソッドで入力がない（$S_i = 1$）特殊な場合であると見なすことが出来る．したがって，このような射対象も一つの類対象であるならば，インスタンスメソッドはモデルの要素の一つの成分に対応する．

なお，クラスメソッドは，状態Sの部分を要素とする冪対象を定義域とする射として定義できる（その例について後述する）．

上記で与えたモデルの要素である属性束 $S \xrightarrow{a} A$ は，ある状況における実例の他の（あるいは自身の）実例への関わりの一切を与えるものである．Fを状態 Sの部分として， Fのすべての要素についてその属性値（*を含める）が得られているとする．そのとき，グラフ

$$F \longrightarrow F \times A$$

をCについて得られている**事実 fact**と言う．これは，RDBのテーブルとして表記することが出来る．各行は各状態要素sに対応するグラフの要素を表し，列として，s（自身の識別子，すなわちテーブルの行識別子，つまり，主キー），状況の要素t，実例値e（実体識別子），そして，状態要素sの各属性値 $a_i s$（a_iは事実属性束 aの属性成分）が指定される．なお，属性値はリテラル（値域が素類のとき）あるいは実体識別子であることに注意．後者の場合，その属性射は通常，**外部キー foreign key**と呼ばれるものに相当するが，われわれの場合，その値はテーブルの主キー（これは状態識別子である）ではなく，実体識別子である．

属性域 Aを積の因子に分解したとき，その因子が唯一のとき，すなわち複数の類の和になっていないとき，通常のRDBのテーブル表記のように，その属性値を置けばよいが，複数の類の和 $\sum_j C_{ij}$ になっているとき，値をどのように表示するべきか．通常のデータベースの理論や実装技術ではこの問題を体系的には扱ってこなかったように考えられる．しかし，素類を値域としない場合は，*Ruby on Rails*[1]では属性の和が考慮されており，**多相 Polymorphic 属性**とし，類とその実例の対を与える．素類の和の場合には，それらの値をリテラルとして列記し，SQLのIN比較演算子を用いて対応の成立を見ることになる．

このテーブル形式に実データを入力することによって実世界の記録を作ることが出来る．これがデータベースの役割である．

なお，こうして作られるテーブルは基本的に第1，2，3正規形 normal formの条件[2]を満たすことになる．ただし，第一正規形 1NFの，各項目には値が一つであるという条件は，素類の和の属性では満たさないが，そもそも正規形の考え方において属性の和の概念が欠けていることが問題であろう．

---

[1] ウェブアプリケーションを構築するためのフレームワークシステム．[Rails]，[TH06] 他を参照のこと．

[2] たとえば，[EN07] を参考にした．

１２．実体への作用と事実の検出

実体は，生成 create，更新 update，合併 merge，分離 split，消滅 deleteの変動を受ける．これらの事象をモデル化して情報システムに組み込む必要がある．また，情報システムは，ある条件を満たす属性を持つ実体を見出す機能 read が不可欠である．これらのうち４つは，しばしばCRUD操作と呼ばれ，情報システムに不可欠の基本操作と考えられている．しかし，ここでいうC, U, Dの作用は実世界での出来事をモデル化するものであるが，情報システムにおけるいわゆるC, U, Dは情報としての記録データの生成，更新，削除である．たとえば，通常のUpdate と呼ばれる操作はデータの更新であるが，その場合，実世界での実体の状態変化を表すものなのか，あるいは実世界の状態変化に関係なく，単に記録済みデータを修正，変更するものなのか，区別されない．ここでいう Update は，当然，前者の意味に用いる．後者には状態を書き直す correct という別の操作を用意する．このように，記録と実世界の作用との混同を起こさないようにシステムを組まなければならないが，このことへの言及はこれまでほとんどされてこなかった．これもまたしばしばモデル化段階，つまり仕様設計段階でのバグのもとになる点である．

これらの変動も実体自身は不変であるが，それらについての事実を表す状態属性束 a が作用 α を受けて変わるとみなす（**図 8**）．$c,\ u,\ d$ を，それぞれ操作 Create, Update, Delete に対応する状態属性束への作用とする．ある類において，その実例を e，状態属性束を a，状態属性値を s とする．状況は時間のみ（要素を t とする）を考慮することにして，作用時を t_oとすれば，$t \geq t_0$ における状況において，

- $c = Create(e, s)$：　$a<t', e>=*$（すべての t' において）である a に対し、$ca<t, e>=s$ とする作用
- $d = Delete(e)$：　a に対し、$ca<t, e>=*$ とする作用
- $u=Update(e, s)$：　a に対し、$ua<t, e>=s$ とする作用
- $Merge(e, s, e_1, e_2) = Delete(e_1)\ Delete(e_2)\ Create(e, s)$
- $Split(e, e_1, s_1, e_2, s_2) = Delete(e)\ Create(e_1, s_1)\ Create(e_2, s_2)$

ここで，＊ は，すべての属性域で属性値が ＊（undefined）であるものを表すとする．

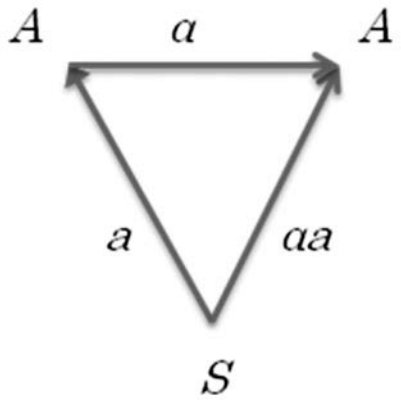

図８　属性束 a への作用 α

一方，状態の事実の検出である *Read*(*find*) はこれらの作用と異なり，OOP のクラスメソッドとして表現される．すなわち，事実 X に対し，属性について与えられた条件を満たす状態の集まりとして X の部分 Y を見出す操作である．

属性束への作用である C, U, D などをデータの記録に反映させることが出来る．この場合，*Read*（*find*) と同様のクラスメソッドとして，つまり事実に対する操作（メソッド）として，通常の Create（実体とその状態の記録を新規に追加する），Update（実体の現在の状態の期間を限定し，新規の状態を追加する），Delete（実体の記録を削除する）などの記録のためのメソッドを使用する．ただし，実体の Delete 作用に対して，記録の削除をする必要はなく，実体の消滅という出来事をなんらかの形で記録すればよい．後述するように，一般に記録の削除はしないほうがよい．

１３．継承

OOP の重要な機能のひとつとして，クラスの**継承** heritage がある．クラス A がクラス B のサブクラスであるとき，A は B のメソッド（インスタンスメソッド，クラスメソッドともに）を「継承する」，すなわち，A でも B のそれらを適用できる，とするものである．われわれの圏論での表現はどうなるであろうか．類 X，類 Y に対し，射 $h : X \to Y$ を X から Y への継承と呼ぶ．このとき，Y の属性射を a_Y とすると，合成射 $a_Y h$ も X の属性射とみなすならば，X のモデル（クラス）は Y のモデル（クラス）を「継承」することになる．特に，$h = 1_X$（X の自己射）とすれば，この事情はより明解になる．すなわちクラスの継承とは，その類が親クラスの類にいわばすっぽりと入ることから生じる．同様にこの関係からクラスメソッドの継承も生まれる．モデルの要素（のグラフ）である事実も同じ継承が適用されるから，RDB でのテーブル間の継承も

同様に表現できる．しばしば，継承はOOPにあってRDBにないもの（OR Mappingのインピーダンスミスマッチの一つ）と言われるが，われわれの形式によるActive Recordの体系ではそうはならない[1]．

１４．状況の因子としての時間

状況の因子として最も重要なのが時間である．以下では，状況Tは時間のみであるとする．

先述したように，すべての類Cについて，$T \to A^C$が成り立つ（AはCの属性域とする）としている．このことは，どの類の実体の変動（実は属性の変動）も時間を共通に持つことを意味している．実体間の共時性synchronismである．一方，$T \times C$は$C \times T$と同型isomorphismであることから，自然な形での対応により，$C \to A^T$が成り立つ．これは，Cの各実体の軌跡（歴史）を示す射である．これは，実体の持つ通時性diachronismを示す．このように見るならば，実体の状態は両者の交差するところで与えられる．

データベースに事実を記録するために，時間を量子化し，さらに，状況の因子としての時間を時間間隔で表現する．すなわち，時間因子を範囲[since, till]で表す．これにより，ある時間tでの実体の状態を得るには，tがこの範囲に入る状態の各属性値を見ればよい．

この状況を示す時間は，時変動データベースtemporal database[2]では妥当時間valid timeと呼ばれる．その時間で成立する（妥当する）事実を示す，と見なすからである．同様の状況の時間因子として，記録時間record timeがある．これは状態sが記録されてから「消去」（仮想消去）されるまでの時間間隔を示すものである．状態の記録の属性であるから，記録についての記録というメタ属性であることに注意する．したがって，状態の生成と消滅とも異なることにも注意する．時変動データベースではこの時間を取扱時間transaction timeと呼んでいる．記録された事実をすべて残す，すなわち，記録の消去は消去時間を記録することで示し，記録自体は残すようにすることによって，実世界の時間経過の記録を作ることが出来る．これは，追記型データベースappend(add) only databaseと呼ばれる．現在の記録メディアの低価格化により，このようなデータベースの有効性が高まっている．そのとき，記録時間をメタ属性として組み込むことはきわめて重要である．データベースのデータとして，記録時間もやはり量子化され，範囲[created at, deleted at]によって与えられる．ただし，この命名から，実体のCreateあるいはDelete作用の時間と混同してはならない．先述したように，実体の生成，消滅，あるいは変更と，記録の生成，消滅，更新とは別物だからである．実装ではしばしばこの項目を属

---

[1] 処理効率を考え，このようにテーブルを分離するか，あるいはSTI(Simple Table Inheritance)と呼ぶ一つのテーブルにまとめる方法をとるか，選択する余地がある．

[2] [Jensen]を参考にした．

性とならべてテーブルの列として表記するが，この項目は属性などと同列に位置するものではなく，実体の状態記録についてのメタな**注釈 annotation** である．実体の合併や分離についてのデータも同様である．

われわれのシステムでは，属性値は（類の要素である）実体であって，その状態ではない．すなわち，事実間で互いに参照するのは実体（の識別子）であり，その実体の状態は状況を指定して始めて定まる．このことは時変動する世界をモデル化する際に決定的に重要である．たとえば，どんな組織でも組織変更があり，またそのメンバーのステータスは変動する．このとき，そのような変動によってシステムの一貫性が乱されてはならない．たとえば，スケジュール表において，３月３１日の財務部の予定を示す記事は，４月１日に人事異動で財務部に異動した社員のスケジュール表には現れない（ように作られなければならない）．単なるスケジュール記事ならばたいした問題ではないが，もしそれがインサイダー取引に関わるような機密事項であるならば，不正確なシステム処理は深刻な問題を引き起こす．また，会社の組織をはじめ，実世界の変動はどんな場合もつきもので，そのためのデータの更新作業はしばしば複雑で誤りを犯しやすい[1]．われわれのシステムのように，相互参照を状態ではなく実体とすることによって，この問題に対する一貫した処理が可能となる．

さて，ここまで状況の因子としての時間を扱ってきたが，状況の時間系列は，類の属性（たとえば，過去，現在，未来などを値とする時制）を規定するための時間系列と区別しなければならない．状況は認識（観察）の起点でありその対象ではならないからである．もしこの区別をしないと，有名な McTaggart の時間の非実在性論[2]に陥ってしまう．たとえば，A 系列（A 属性と言うべきであろう）と呼ぶ時間系列において，事象は過去，現在，未来の属性のいずれも持つがそれらは互いに排反するので矛盾である，という論法である．われわれのモデリングでは，状況の時間系列（事象を見る時間の系列）と対象実体としての事象を位置づける時間系列との相対関係で A 系列の属性値（過去，現在，未来）が与えられるとすれば，この矛盾は生じない．しかし，McTaggart は，この論法は，時系列（の実在を）説明をするのに時系列（の実在）を使っているため，無限後退の矛盾を招くということになる．時間系列は「変化」と不可分であるが，認識者の時間から対象事象をみて，それが変化すると考えるのではなく，対象自体が時間とともに「変化」していくというのがリアルな「変化」であるはずである（そのため，A 系列が時間系列の本質を与えると考える）．したがって，このような，認識者の時間系列と認識対象の置かれる時間系列の概念分離は出来ないのだ，という．

「時間の非実在性」論法を逆からとらえるならば，両者の分離のようななんらかの設定なし

---

1 ここでもまた典型的な悪例として社保庁のシステムを引き合いに出したくなる．しかし，このような例は，実は我々の身の回りの多くのシステムに潜在していると考えられる．

2 [Mc198]参照.

には，「時間」という概念を矛盾なく記述することは出来ない，ということになる．実際，両者の分離設定は，モデリングの概念装置として極めて有効である．たとえば，この装置なしに物理学は成立しないであろう．しかし，残念ながら，現在の多くの情報システムでは，この装置が欠落し，システム構築の視野にあるのは「現在」という実時間のみ（つまりは，A系列のみ）で，「過去」も「未来」も考慮されず（もし考慮すれば，自己矛盾を起こす），あってもアドホックにしか対処されないのが実情であろう．

ところで，各「時間」そのものは認識対象としての実体になり得るだろうか．もちろんなり得て，それに名称を付けることが出来る．そして，その変動（状態の違いがもたらす属性値の違い）もあり得る．この場合，状況をやはり時間Tのみであるとすれば，その状態Sは，$T\times T$である（McTaggartにいわせれば，この「時間」と呼ぶ実体は，時間そのものではなく，その化石標本に過ぎないと言うであろう）．

いずれにせよ，先述したように，リアリティとは，われわれの認識以前のあるがままの世界であるとすれば，何らかの精神作用によって得た（作られた）「時間」という概念は，そもそもリアリティではない．すなわち，この「時間」は非実在unrealということになる．このように考えるならば，この種の議論は，あらゆる概念について成り立ち，どのような「概念」も実は非実在であり，実際，実在性を主張すると論理に破綻が生じることは，仏教哲学のみならず，多くの哲学で言ってきたことである．しかし，このことは，もちろん我々の精神作用が無意味だと言っているのではない．リアリティそのものを精神作用によって得られないことは承知の上で，それに迫る工夫をしてきたのが，モデリング（何らかの記述）の進化の道であった．その過程の最先端にたてるかどうか，それが情報システム構築のモデリング手法に問われていることである．

１５．モデル間の関連

対象世界を一つの類，あるいはそのモデルのみで表せることは少なく，特に，対象世界の中のある条件の付いた実体を見出す（Find）操作では，相互に関連するモデルの複合による対象世界の表現が必要である（**図 4**参照）．RDBではこのことが早くから意識され，テーブルの接合(JOIN)操作が導入されている．ここではモデル間の関連についてわれわれの考えを述べる．

関連には２種類ある．ひとつは類の属性による類から類への参照によるものである．もうひとつは，二つの類の各実例の属性値を比較し，その真偽値を与えるものである．

（１）**参照関連**．　類C，C'がいずれも非素類で，類Cから類C'への属性射があるとき，その属性射をCのモデルからC'のモデルへの参照関連という．モデルの事実をテーブル化したRDBのテーブルでは，C'に対応するモデルの事実のテーブルを参照する一種の外部キーとなり，そ

の値はC'の実例の識別子である（テーブルの通例のIDは状態の識別子になる）．*Ruby on Rails*の″belongs_to″に対応する．逆は″has_many″あるいは″has_one″（属性射が単射の場合）である．

（２）**比較関連**．比較演算子をθ，類Cの属性値をa，類C'の属性値をa'としたとき，$\theta\,(a, a')$を比較関連という．ここで，比較が出来るためには，a，a'は同じ類の要素でなければならない．比較演算子としては，大小の比較やand, orなどの論理演算子がある．属性の値域が素類のとき属性値はリテラルで与えられることになる．実装では，SQLの比較演算子（ただし，二項演算子のみ）を用いる．SQLの比較演算子にはないが重要な比較演算子として，*contains*（反対は*contained-by*）がある．これは，比較の場の類が半順序集合partial ordered setの場合，二つの要素の順序を比較するものである．組織構造など階層構造の表現においてこの比較が不可欠となる．たとえば，社内で宛先を人事部全員とするメッセージを送る場合，人事部以下のすべての部署に所属する社員をその受信者としなければならないとき，この比較演算子は不可欠であり，またその処理を誤ればきわめて処理効率の悪いものになる．RDBでの実装においては，階層構造を反映するように要素の識別子（コード）を工夫し，LIKE演算子を用いることにより，この処理を高速化できる．このほか，特殊比較処理として幾何図形の比較などがある．

参照関連は，外部キーの値とそれに対応する実体識別子とが等しいということであるから，このように見るならば，参照関連も一種の比較関連と見なすことが出来る．結局，一般に関連は同じ類の二つの実例の属性値どうしの比較によって与えられ，その比較が真になるとき，実例間で**関連が成立する**，という．

１６．部分属性と部分要素

属性射が部分射の場合がある．すなわち，実例のすべてにその属性値が与えられない部分属性（６．参照）の場合である．部分射をどう扱うか，われわれはScottとFourmanのアイデア[1]にしたがい，その場合の属性値を部分要素であるとして，記号 ＊ で示す．すなわち，属性値が定まらない場合，その値を*とする．プログラミング言語では，nil, undefinedなどと呼ばれ，データベースではNULLと表記されるものに相当する．ここで，＊は属性値としてのみ意味を持ち，実例（実体）ではないことに注意[2]．したがって，状況tにおいて，ある類の実例をx，その類の属性をaとしたとき，xに対しその属性値が定まらないとき，$a\langle t, x\rangle = *$となる．

属性値どうしの比較である関連においては，そのうちのどちらかが*の場合は，関連は成立し

---

[1] [G84] p.267以下を参照のこと．

[2] この記号の導入は，従来の情報処理理論で用いられたundefinedやgarbageなどの意味の*と異なる．その場合は，対象とするどの類（集合）も*を持つものとするため，圏$1/S$（Sは集合を対象とする圏）あるいは圏S*と呼ばれる圏となり，これはトポスにならない．

ない，と見なす.

１７．モデルの結合とテーブルの結合

ある条件の実例を見出す（検索する）とき，対象の類の属性値がその条件に合う実例を見出すことになるが，その条件が複数の非素類の属性に関わる場合がある．たとえば，ある部署のある従業員の給与明細を知りたい，といった場合，会社の組織，従業員，給与体系などの類が関わる．すなわち，検索対象（範囲 scope）が複数の非素類により構成される必要が生じる．複数の非素類の組み合わせを**類の結合**と呼び，それはモデルの結合，さらには，事実をしめすテーブルの**結合**につながる．RDB ではそのためにテーブルの結合(join)操作が用意されている．結合されたモデルを**複合モデル**と呼ぶ.

検索条件は，複合モデルを表す複合テーブルに対する条件となる．類，あるいはモデルの結合にはそれらの関連を使用する．圏論では，参照関連の結合は，グラフ（関係の一種）の関係合成 relational composite[1]であり，これは RDB で多用される外部キーと主キーの対応による内部結合（inner join）に対応する．しかし，モデルの結合はそれに止まるものではないので，比較関連を合わせた関連全般を用いることにする.

類 X，その属性 a，類 Y，その属性 b としたとき，それらのモデル間の関連を次のように表記する．型理論 type theory の表記法にならい，類（型）X の実例 x を $x:X$ のように表す．また，指定された状況をその要素 t とする.

- 参照関連：　$x:X \ .a \ >> \ y:Y$
  X の実例 x の Y への a 属性値 $a\langle t,x\rangle$ が Y の実例 y の実体識別子属性の値 y に等しいとき，実体 x，y の間で成立．属性射が単射の場合は，>>の代わりに，-> で表記する.
- 比較関連：　$x:X \ .a \ \theta \ .b \ y:Y$
  $\theta(a\langle t,x\rangle, b\langle t,y\rangle)$ が真のとき，実体 x, y の間で成立.

属性が多相の場合は，属性の和の値域を Z としたとき，それを *Z であらわし，関連の右または左に記す．属性値がリテラル（すなわち値域が素類）のばあいは，SQL の比較演算子 IN を用いる.

関連を成立させる相手がいない場合も，結合類の一つの実例としてその実例を残す場合，その関連を**部分関連**と呼び，関連演算子（>>と θ）の右，あるいは左に記号～を付けてあらわす．そして結合類の実例としては，その相手方の値を属性値の場合と同様に * とする．属性が全射

[1] [LR03] p.92 参照のこと.

epimorphism である必要が無い場合は，参照関連は，>>~ で表されることになる．また，逆に，属性が部分射である場合は，~ >>で表される．単射参照の場合や比較参照の場合についても同様である．部分関連でなく完全対応が要求される場合は，**完全関連**と呼ぶ（**図 9**参照）．例えば，*Car* という類と *Human* という類の関連を所有という関係で考えてみる．どの車も必ずひとり所有者がいるし，どの人も必ず車を所有しているという条件を想定する対象世界では，その結合は *Car* >> *Human* で現されるが，車を必ずしも持たない人がいるという想定ならば，*Car* >> ~ *Human* となる．後者は圏の一般的な射である．所有者が必ずしもいない，あるいは不明のこともある（盗難車の管理などの）場合は，*Car* ~ >> *Human* で表され，どちらも相手がかならずしも決められない，あるいは，互いに相手の存在を必要としない場合（たとえば，盗難車のように所有者が定まらない車があり，また，誰しもが車を持っているとは限らないような設定を対象とする場合）は，*Car* ~ >> ~ *Human* で表される．属性値域が素類のばあい，属性値はリテラルになるが，その属性が部分属性であるとする，つまり，値を必ずしも持たなくてよいとする場合は，RDB のテーブル設定において，NULL 値を可とする設定を行う．

検索において，部分関連を組み込む背景条件の指定は決定的に重要である．この設定を誤れば，バグとなる．それもしばしばバグであることさえがわかりにくい，たちの悪いバグとなる．

このように関連を定めた上で，モデル *X* とモデル *Y* の結合とは，関連が成立する実例の状態の組のすべての集まりを言う．これは，比較関連と部分関連を除くならば，関係合成 relational composite[1]である．われわれの場合は，比較関連と部分関連を導入しているので，一般化された関係合成であるといえる．

モデルの結合操作は事実の結合操作に対応し，これは RDB のテーブル操作に対応することとなる．SQL の JOIN 操作には，内部結合 inner join と外部結合 outer join の 2 種類がある[2]．われわれのいう完全結合が内部結合に，部分結合が外部結合に対応することは明らかであろう．両側の部分関連は SQL では FULL JOIN になるが，RDB によっては実装されていないものもある（たとえば MySQL5.1 にはない）．

複合モデルはこうしてモデル間の関連を指定して結合することでできあがるが，それはモデルを節 node，それらの関連をリンク link とする一種のネットワーク構造を形成する．これを**関連ネット association net** 略して **A-Net** と呼ぶ．このようにして，実体の検索の範囲を A-Net によって正確に指定できることが出来る．また，*Ruby on Rails* の has_many や belongs_to 宣言に変わるものとして，Create，Update および Delete 操作に伴う付随メソッドを生成するためのモデル関連の指定にも使用できる．

---

1 [LR03] p.91 以下を参照のこと．

2 この用語はわかりにくい．完全結合，部分結合と呼ぶべきであろう．

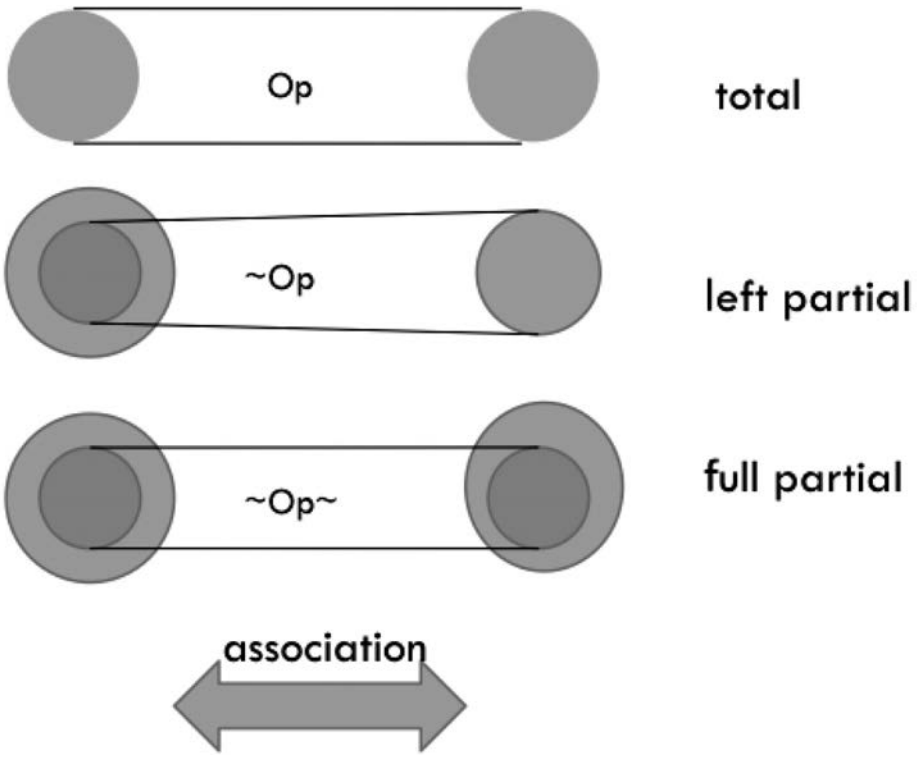

図 9　完全関連と部分関連（Op は関連演算子）

A-Net を直列化して文字列の集まりに変換することが出来る．これは対象世界の一種の記述を「言語」として与えることとなる．もとの表現がネット構造であるから，文字列の順の作り方，すなわち統語規則 syntax に任意性があり，語順を日本語風にすることも，英語風することも可能となる．A-Net はネットワークとして統語規則によらない一種の意味の正規表現になっているといってもよい（図 10．参照）．

検索範囲を与える A-Net は SQL の SELECT 文の FROM 部と WHERE 部に変換することによって，SQL による検索に直接利用できる．実際，すでにこの変換プログラムを筆者が作成している．この変換においては，状況を指定するパラメータとして妥当期間と記録期間の指定が伴い，そのための条件文が付けられる．現実的な対象を扱う場合，モデルの複合はかなり複雑になり，結合するテーブルの数は多くなる．しかもモデルの部分関連にはテーブルの部分結合を行わなければならない．実際，A-Net から変換した SQL 文の長さはしばしば A-Net 文の１０倍以上になってしまう．実世界の複雑な対象の範囲を指定するためには，SQL 文では往々にしてきわめてわかりにくいものになってしまうのに対し，A-Net 表記では，複合モデルそのものの必要にして十分な記述ですむため，書きやすく，読みやすく，また，修正しやすい．

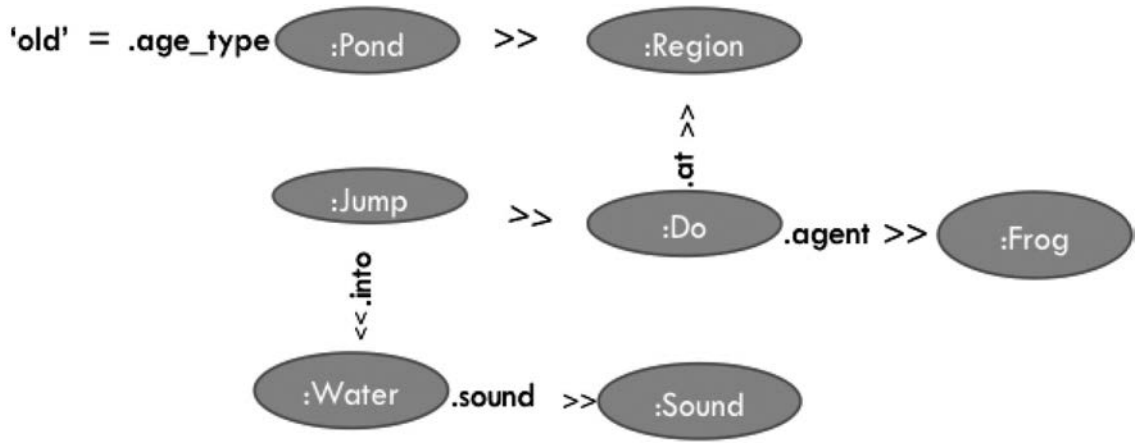

図 10　A-Net の例（「古池や蛙飛び込む水の音」）

１８. A-Net 表現と他の表現との比較

簡単な例を使用して A-Net の表現と他の「知識表現」とを比較してみる.
自然言語理解システムの先鞭を付けた Terry Winograd のシステム SHRDLU から例題を採る[1].
質問　"Which block supports pyramid ?"　をそれぞれの表現であらわしてみよう.

- **Microplanner** では,

  (goal (objects ?x1 block ?))(goal (objects ?x2 pyramid ?))(goal (supports ?x1 ?x2))

- **Prolog** では,

  objects(X1, block, *) & objects(X2, pyramid, *) & supports(X1, X2)

- **SQL** では,

  SELECT * FROM objects AS x1 JOIN objects AS x2 JOIN supports
  ON supports. supporter = x1. id AND supports. supportee = x2. id
  WHERE x1. shape = 'block' AND x2. shape = 'pyramid'

- ***Ruby on Rails***（われわれの拡張 find）では,

  Object. find : all,
  : scope => "x1:Object [(. shape = 'block') (<< .supporter)] : Support .
  supportee >> x2:Object . shape = 'pyramid' "

[1] [So00] p.157 を参照のこと.

ここでは，A-Net を，*Ruby on Rails* の *find* を拡張して設けた scope というオブションで使用している．このオブションによって与えられた A-Net に相当するテーブルの結合が SQL の SELECT 文の FROM 句と WHERE 句として作り出され，その複合テーブルに対して，SELECT 操作が実行される．

１９．局所制約としての A-Net と並列処理の可能性

A-Net は類（あるいは概念）間の関連に注目し，それらを可能な限り正確に記述しようとするものであるが，関連は，見方を変えれば，類間の関係を制約するものである．この制約ネットによって対象とする世界を記述し，その制約を満たす実体の状態の集まりを規定していると見ることが出来る．したがって，A-Net が規定する制約を充足する解を求めることが，求める世界の状態を得ることになる．問題の使用を制約充足問題として見ることは，制約緩和による並列処理をしやすくすることであるから，A-Net 表現は並列処理による問題解決への道を与えるものであるともいえる．

２０．A-Net による情景描写

A-Net の応用として，情景描写がある．これは自然言語で書かれた文章の意味を表示し，あるいは，画像処理で得られた情景を表現するのに利用される可能性がある．次の２例を挙げる．

- ‘東海’ = .name　t：Region [(contains)(‘small’ = .scale　:Island <<)]
  i：Region [(contains)(:Beach <<)]
  b：Region [(contains)(‘white’ = .color :Sands <<)]
  s：Region
  [(<< .at)(:Self << .agent)] :Do [(.modality = ‘present’)
  (<< Cry .mode = ‘wet’)

- ([(<<)(:Crab << .with)] :Play)]”
  ‘old’ = .age :Pond → :Region
  [(<< .at)(:Frog << .agent)] :Do [(.modality = ‘present’)
  (<< :Jump .into >> :Water .sound → :Sound)] ”

これらが何を表しているかは，あきらかであろう．なお，区別する必要のない場合は，実例変数名を省略している（モデル名をそのまま用いてもよいのでそれをデフォールトとする）．

この文字列はもともとネットワークである A-Netを文字列に変換したものであるから，語順に任意性がある．したがって，もとのネットワークを英語風の語順で表現しなおすことは容易である．このことは，翻訳における語順の正規化に役立つことを示唆していると考えられる．

２１．A-Net の応用

その他，A-Net の応用は多岐にわたると考えられる．いくつかの例を挙げれば次のようになろう．

☆　ソフトウエア工学における UML の代替.
☆　RDB の E-R 図の代替
☆　Web サービスにおける XML 表記の代替.
☆　Web 意味論におけるオントロジーの代替
☆　CAD における物体の表現
☆　CG における情景表現.
☆　高分子合成としての医薬品の設計
☆　自然言語理解と翻訳
☆　画像処理における情景分析
☆　ソフトコンピューティングにおける新パラダイム

２２．まとめ　－　何を得たか

情報システムの構築において，実世界（リアリティ）をいかに正確にモデル化し，記述するか，が最重要課題である．これまでの半世紀以上にわたる情報処理技術の研究開発も当然そのことに意を尽くしてきたが，改めて見直すとさまざまな欠陥があることがわかる．そして得たものはおおよそ次の通りである．

- 世界は実体とそれらの相互の関わり方（依存や参照）によって構成されるとする素朴な世界観を議論の出発点にした．
- 似ていると認識される実体をまとめ，それらの類とし，実体はその類の実例とした．また，実例の他の（あるいは自の）類の実例への関わりをその類の属性とした．

- この認識構造はPeirceの三つ組によって説明できること，また，三つ組を基礎にして圏論により数学的な明晰さを持ってそれを体系化出来ることを示した．
- 「不変」である実体が状況によって「変動する（ように見える）」仕組みを，その属性値の変化として分析した．ここで，状況とは，時間，場所，観点などを因子として持つ．
- 値が定まらないことがある属性を部分属性とし，それを定義した．
- 類の属性すべてをまとめ，それを「属性束」と呼んだ，属性束は，属性の和と積で表されることを示した．属性の積は関係データベースなどでいう「関係」であるが，これまであまり言及されなかった属性の和を導入し，多相属性であるとした．
- 自己属性のみを属性とする類を素類とした．素類の実例の値は自身に付けられた「名前」でリテラルであるとした．
- 状況と類の積をその類の状態とし，実体の状態変動を説明した．
- 状態の属性束すべてを射対象として示し，それを類の「モデル」とした．このモデルはOOPのクラスによって表せることを示した．
- モデルの要素である属性束をグラフとして表す方法を示した．これは状態とそれに対応する属性束の各属性値のリストである．
- 対応する属性束の各属性値が得られている状態の要素からなる状態の部分に対し，そのグラフを「事実」とした．事実はRDBのテーブルで表せることを示した．
- 状況の因子として時間をとりあげ，時変動実体を扱う体系的なモデル化手法を示した．これにより，時変動データベースを新たに見直すとともに，その実現方法を与えた．
- McTaggartのいう時間の非実在性について，われわれのモデリングの立場から論じた．
- 属性をもとにモデル間の「関連」を新たに定義した．この関連には完全関連と部分関連があることを示した．モデル間の関連により複数モデルの結合を行う方法を示した．それらに対応する事実の結合は，SQLのいうJOIN（結合）操作に対応し，完全関連は内部結合に，部分関連は外部結合に対応することを示した．
- 複数モデルの結合により対象世界の範囲（スコープ）を明確かつ簡潔に示す記法としてA-Netを新たに提案した．これはネットワーク構造を採るが，それを文字列の集まりとして表す方法を示した．
- A-Netは対象世界の構造の表現方法であり，この観点から他の述語論理などの表現方法との比較を行った．
- A-Netの多岐にわたる応用を論じた．

２３．残された課題

根本的なところから情報システムのモデル化について見直しを行い，実践に直結する水準にまでアイデアの具体化を行ったが，理論面ではデッサンの段階に止まっている．特に圏論の部分はより精緻な議論が必要であろう．一方，認識論や記号論など哲学的な課題について圏論の観点から全面的に見直すことも興味深い課題であろう．圏論では，対象は射で構成された網の単なる交叉点に過ぎず，その自立性を認めない，まさに徹底的な関係主義，相対主義にたっていて，仏教哲学[1]と相通じるものがある．それに対し，現象論のノエシスとノエマの議論などを見ると，対象の自立性を危ぶみながらそこまでの割り切りが出来ない（「括弧に入れて」?）故の議論の混乱があるように思われる．圏論の概念と用語により，これらの考え方を整理しなおすことが出来るならば，すくなくともその議論はずっとわかりやすいものになるのではないだろうか．

ソフトウエア工学におけるプログラムの意味論，プログラムの検証などにも重要な関わりを持ち，さらにモデル指向プログラミングとも当然関わっているが，それらとの関連の検討は今後の課題としたい．

情報処理一般にかかわる課題としては，A-Net の応用性について実践的に検証してみる必要がある．その対象分野は，知識表現，自然言語理解，画像処理の情景分析，パターン認識一般，ソフトコンピューティング一般に及ぶ．圏論の一般性は，代数的処理一般と位相数学的処理一般に共通するものであるから，基礎さえ確固たるものにすることが出来るならば，その応用性は一挙に広がるはずである．

謝辞

本稿は，2007年と2008年のＫフォーラムでの発表と討論をもとにして作成したものである．その機会および本論集への掲載の機会を作っていただいた福村晃夫先生（名古屋大学，中京大学両名誉教授）に謝意を捧げたい．また，Ｋフォーラムにおいて貴重なご意見をいただいた諸先生にこの場を借りて感謝したい．

[1] ［三枝90］を参考にした．

参考文献

[DS05] F. Dau, Marie-Laure Mugnier, G. Stumme (eds), "*Conceptual Structures: Common Semantics for Sharing Knowledge*, *13th International Conference on Conceptual Structures*, *ICCS 2005*", Springer-verlag, 2005

[EN07] R. Elmasri, S. B. Navathe : "*Fundamentals of Database Systems*, *5th edition*", Elmasri & Navathe, 2007

[G84] R. Goldbratt : "*Topoi. The categorical analysis of logic*", Dover Publications, Inc, 1984

[Jensen] C.S.Jensen: Temporal Database Management,http://www.cs.aau.dk/~csj/Thesis/

[LR03] F.W.Lawvere and R.Rosebrugh: *Sets for Mathematics*. Cambridge University Press, 2003

[LS91] F. W. Lawvere, S. H. Schanuel : "*Conceptual Mathematics*. *A first instruction to categories*", Cambridge University Press, 1991.

[Mc1908] J.E.McTaggart, "The Unreality of Time", *Mind: A Quarterly Review of Psychology and Philosophy* . 17 (1908): 456-473 (http://www.ditext.com/mctaggart/time.html)

[Rails] http://api.rubyonrails.com/

[SH05] M. P. Singh, M. N. Huhns :"*Service-Oriented Computing*. *Semantics*, *Processes*, *Agents*", John Wiley & Sons, Ltd, 2005

[So00] John F. Sowa: "*Knowledge Representation*. *Logical*, *Philosophical and Computational Foundations* ", Brooks/Cole, 2000

[TH06] D. Thomas, D. H. Hansson: "*Agile Web Development with Rails*, *second edition*", The Pragmatic Programmers LLC, 2006

［熊野 06］ 熊野純彦：「西洋哲学史．近代から現代へ」岩波新書，2006

［熊野 07］ 熊野純彦：「西洋哲学史．近代から現代へ」岩波新書，2007

［三枝 90］ 三枝充悳：「仏教入門」岩波新書、1990

［廣松 82］ 廣松渉：「存在と意味−事的世界観の定礎−」岩波書店．1982