B-010

MDA ： その数学的定式化

中村　正治
Masaharu NAKAMURA

１．まえがき

２０１４年度の FIT にて、筆者は MDA, MDD の全体像を与える試みの論文を提出した。実践的な内容を報告ことが目的であり、実際、現実のプロジェクトへの応用に基づいたものでもあった。

しかしながら、モデリングアプローチは常に二つの問題を内包する。一つは、現実世界のモデルへのマッピングであり、これはモデリングの目的に応じた、恣意的にものにならざるを得ない。マッピングルールを定めたとしても、絶対的なルールなどは定めようもなく、議論を呼ぶマッピングが常に起こりうる。二つ目は、モデル自体のあいまいさである。ほとんどの文献で、モデルの定義はあいまいで厳密さを欠いており、その無矛盾性などは全く議論されていない。多くの解説書は、モデル自体の議論と現実世界とのマッピングを行きつ戻りつの記述であり、モデル自体のモデルが定義されていないという、奇妙な状態である。

多くのモデリング現場でも、権威と経験が結論を与える、というような場面が多く、およそ論理的とは言い難い方法論が多くみられる。

２．本論文の概要

本論文では、まずは、それぞれのモデルとそこに現れる概念の定義を与えることを試みた。コンピューター・サイエンスで扱うモデルは、ほとんど常に、クラス/インスタンスというメタモデル構造を持っており、それがモデル自体の定義を大変複雑にする。

定義部分だけ見れば、数学の一般的分野よりもかなり込み入っているのがわかる。いくつかの簡単な命題・補題とその証明は与えたが、今回は低利の構成に基づいた理論体系を与えるには至っていない。

ただ、このような定義体系そのものがあまりないように考えられるので、まずは、出発点として、報告することとする。

3. プロセスモデル

3.1　タスク

T : タスククラスの集合。**C :** 条件クラスの集合。

任意の **t1, t2　∈　T,**　において **t1 ≠ t2**　のとき、**T** を **タスククラスの正規化集合**と呼ぶ。

t1≠t2, c∈C のときに、**s=[t1, t2, c]** の組を**順次クラス**と定義する。

この時、任意の **t1,t2,t3,t4 ∈ T**　で、**s1=[t1, t2, c1] , s2=[t3, t4, c2]** において、**t1=t3** ならば **c1=c2** であると定義する。

s1=[t1,t2,c1], s2=[t3,t4,c2],　・・・s1, s2 ∈ S, かつ　**(t1≠t3 ∨ t2≠t4 ∨ c1≠c2)**のとき、**S** を **順次クラスの正規化集合**と呼ぶ。

3.2　プロセス・クラス

sp=[t tp, cp] が存在して **sq=[tq, t, cq]** なる **tq, cq** が存在しない、ただ一つの空のタスクを　**φs**　とし **始タスク**と呼ぶ。

sr=[tr, t, cr] が存在して **ss=[t, ts, cs]** なる　**ts, cs** が存在しない、ただ一つの空のタスクを　**φe**　とし **終タスク**と呼ぶ。

o, d 成る函数を定義して、**s=(t1,t2,c)** としたとき、**o(s) = t1, d(s) = t2** と定義する。

順次クラス**{s1,s2,・・・sn}=p,**　として、∀**si ∈ p** で ∃**sj　| d(s1)=o(sj)** ただし、**si≠φe**　および　**sj≠φs** ならば、**p** を**プロセス・クラス**と呼ぶ。

sp=[tp t, cp] が存在して **sq=[t, tq, cq]** なる **tq, cq** が存在せず、しかも **t≠ φs** の場合、**t** を**始端が開のタスク**とよぶ。

sr=[t, tr, cr] が存在して **ss=[ts, t, cs]** なる　**ts, cs** が存在せず、しかも **t≠ φe** の場合、**t** を**終端が開のタスク**とよぶ。

始端が開のタスクも、終端が開のタスクも、どちらのタスクも存在しないプロセス・クラスを、**完備なプロセス・クラス**と呼ぶ。

任意の **p1, p2 ∈ P** について、**p1≠p2** ならば、**P** をプロセス・クラスの正規化集合と呼ぶ。

3.3　ネストと規約タスク

単射 **n : ∃p → t** かつ **n(-1)**が定義できて 、**n(-1) : t → p** が単射の時、**n** をネスティング、**t** を **p** のネストプロセスと呼ぶ。

ある **t** において ネスティング**(**とその逆**)**が存在しない時、**t** を既約タスクという。

3.4　タスクと条件のインスタンス

T = {t1,t2,・・・}として∀**t∈T**∃**t∈T** に単射 **m : t → t** を定義できるとき、t は **t** のインスタンスと呼ぶ。

C = {c1,c2,・・・}として∀**c∈C**∃**c∈C** に単射 **n : c → c** を定義できるとき、c は **c** のインスタンスと呼ぶ。

3.5　フロークラスとフロー

p ={s1,s2,s3,・・・, sn}　なる **p** を**プロセス・クラスとする。**

順次クラス **x1,x2,・・・xm ∈p, 1≦m≦n** として、**f={x1,x2,・・・xm }**する。

このとき、∀**xi ∈ f**　で ∃**xj　| d(x1)=o(xj)** ただし、**xi≠φe**　および　**xj≠φs**　なる　**f ⊂ p** をフローと呼ぶ。

f1, f2,・・・fn ⊂ p なる　**F={f1, f2,・・・fn }** を**フロークラス**と呼ぶ。

f ={x1, x2, x3,・・・, xn}　なるフロークラス **f** において、順次クラス **x1,x2,・・・xm ∈p, 1≦m≦n** として、　**f={x1,x2,・・・xm }**する。

このとき、∀**xi** ∈ **f** で ∃**xj** | **d(x1)=o(xj)** ただし、**xi≠φe** および **xj≠φs** なる **xj** がただ一つしか存在しない場合、**f** を **STP** フローと呼ぶ。

STP は Straight Through Process の略である。

命題

f が STP の場合、同時並行タスクは存在しない。

∵ 同時並行タスクとは **xp=(t1,t2,c)**, **xq=(t1,t3,c)** のことであり、**o(Xp)=o(xq)** となり、STP の定義に反する。 ■

命題

f が STP 場合、ループは存在しない。

∵ **x1=(t1,t2,c1)**, **x2=(t2,t3,c2)**, ・・・ **xi=(ti,ti+1,ci)**, ・・・ **xn(tn, ti,cn)**, **xn+1(ti,ti+1)**・・・ となり **o(xn+1)=ti**, **o(xi)=ti** となり、一つしかないとする定義に反する。 ■

3.6 プロセス・インスタンスとプロセス・ステート

p をプロセス・クラス、**t1, t2, t3**・・・**ti, tj,**・・・を **p** を構成するタスククラスとする。

ある、**ti,tj** について、**o(**・・・**o(o(tj)**・・・**)=ti** のとき、**ti<<tj** と記述する。

命題

この時 **d(**・・・**d(ti)**・・・**)=tj** が成立する。これを **tj>>ti** と記述する。逆も真なので、

即ち、**ti<<tj** と **tj>>ti** は同等である。

ti<<tj も **ti>>tj** が成立しない場合、 **ti<>tj** と記述する。

証明

o(・・・**o(o(tj)**・・・**)=ti** の両辺に繰り返し **d** を作用させる。■

t をタスククラス、**t** のインスタンスを｛t1,t2,・・・ti・・・｝とする。

Tt : t → t**j** ∈ ｛t1,t2,・・・ti・・・｝ をインスタンス化と呼ぶ。

プロセス・クラス **p** を構成するタスククラスを、**t1, t2, t3,**・・・**tn** とする。

命名規則として、**i<j** のとき、**ti<<tj** ないしは **ti<>tj** とする。

1≦j≦n のある **j** にたいして、**Ttj : tj**→ tjp なるインスタンス化があった時に、

ti<<tj を満たすすべての **ti** には **Tti : ti** → tiq なるインスタンス化が存在する場合、

t1,t2,t3,・・・**tj** を含むフローを **f1,f2,** ・・・**fx** とすると、

［**f1,** t1,t2,t3, ・・tj］, ［**f2,** t1,t2,t3, ・・tj］,・・・［**fx,** t1,t2,t3, ・・tj］ をプロセス・クラス **p** プロセス・インスタンスと呼ぶ。

この時、 ［t1,t2,t3, ・・tj］ をプロセス・インスタンスのプロセス・ステートと呼ぶ。

クラス **p** のあるフロークラス **fy** の全てのタスクにインスタンスが定義されていれば、「**p** の フローは **fy** で決定された」と表現する。

4. データモデル

4.1 エンティティ

A : アトリビュート・クラスの集合。

E : エンティティ・クラスの集合。

任意の **a1, a2** ∈ **A,** において **a1 ≠ a2** のとき、**A** をアトリビュート・クラスの正規化集合と呼ぶ。

A = {a1,a2,・・・}, この時、∀**a**∈**A**∃**a**∈**A** に単射 **m :** a → **a** を定義できるとき、a は **a** のインスタンスと呼ぶ。

任意の **e1, e2** ∈ **E,** において **e1 ≠ e2** のとき、**E** をエンティティ・クラスの正規化集合と呼ぶ。

E = {e1,e2,・・・}, この時、∀e∈E∃**e**∈**E** に単射 **n :** e → **e** を定義できるとき、e は **e** のインスタンスと呼ぶ。

A1 をあるアトリビュート・クラスの正規化集合、 **e1** ∈ **E** を あるエンティティとするとき、**[e1 , A1]** を**アトリビュートが定義されたエンティティ**と呼ぶ。

E={e1,e2, ・・・**en}** をエンティティ・クラスの正規化集合、文字列を **s1,s2,**・・・**sn** として、

i≠j ならば、**si≠sj** で、**(ei,si)** の組が定義できるならば、**si** を **ei** の「意味づけられた表現」 と呼ぶ。

A={a1,a2, ・・・**an}** をアトリビュート・ラスの正規化集合、文字列を **z1,z2,** ・・・**zn** として、**i≠j** ならば、**zi≠zj** で、**(ai,zi)** の組が定義できるならば、**zi** を **ai** の「意味づけられた表現」 と呼ぶ。

4.2 正規化エンティティ

エンティティ・クラス **e** = { e1, e2,・・・en } のとき、定義により、可附番なので順番に自然数を与えることができる。

この時、∀i,j 1 ≦ i,j ≦ n で ei ≠ ej ならば、**e** は正規化エンティティである、という。

この正規化エンティティ **e** で自然数 i : 1< i ≦ n と自然数 k を与え、全単射 X : i → k が定義されるとき、この k をエンティティ・インスタンス ei の**インデックス**と呼ぶ。

定義から明らかなように、en≠em ならば n≠m。 逆も正しい。

A をアトリビュート・クラスの集合で、**A = {a1, a2, a3,** ・・・**}**

[e, A]が定義できる時、これを正規化エンティティ **e** のアトリビュートリストと呼ぶ。

エンティティ・インスタンス ep ∈ **e** が存在して、**A|e** で、∃a1x ∈ **a1 ,** ∃a2y ∈ **a2,** ・・・のとき、

これらを、ep の値と呼び、[ep | a1x, a2y, ・・・]と表記し、ep **の値**と呼ぶ。

エンティティとアトリビュートに関して、次の公理を与える。

公理 :非区別性公理

∃e1, e2 ∈ **e** で e1 ≠ e2 のとき、 **[e, A]**において、∃a1x ∈ **a1 ,** ∃a2y ∈ **a2,** ・・・を適切に選んで、

[e1 | a1x, a2y, ・・・]、 [e2 | a1x, a2y, ・・・] を定義できる。

補題 :区別性定理

∃e1, e2 ∈ **e** で e1 ≠ e2 のとき、 **[e, Ap]**において、∃a1x ∈ **a1 ,** ∃a2y ∈ **a2,** ・・・を適切に選んで、

[e1 | a1x, a2y, ・・・]、 [e2 | a1x, a2y, ・・・] を定義したとき、

Ap ⊂ **Aq** なるアトリビュート・クラス集合で **aq** ∈ **Aq** ∧ **aq** ∉ **Ap** で ∃ aq1, aq2 ∈ **aq** ∧ aq1≠,aq2 を適切に選んで **[e, Aq]** を定義し、

[e1 | a1x, a2y, ・・・,aq1]、 [e2 | a1x, a2y, ・・,aq2] を定義できる。

証明

N ⊂ 自然数 なる **N** を選べば、定義により、n1,n2 ∈ **N** ∧ n1 ≠ n2 をえらんで、aq1=n1, aq2=n2 とすることが出来る。 ■

これらを合わせて、存在整合性とよぶ。

4.3 エンティティとインスタンスの表現

存在整合性を利用すると、エンティティとインスタンスを、視覚的に表現する根拠が得られる。

インスタンスを「一行」で表現し、その行に一から始まる連続した整数 **(**したがって、ユニークな数**)** である「行番号」を割り当てれば、この番号をインスタンスの識別子として利用できる。一から始まることや、連続していることは要件ではないので、行数から離れて、ユニークな任意の数を用いてもよい。

これらの数は、この複数行によるインスタンスの存在の整合性を担保するものと解釈できる。本論文では、必要があればこの数を**インデックス**と呼ぶ。

全てのインスタンスには、アトリビュート・クラスで定義されたアトリビュート・インスタンスが定義されている。

アトリビュート・ラスを「列」とすれば、インスタンスの集合は、個々のインスタンスを表現する列と、アトリビュート・クラスとそのインスタンスを表現する列のよって、「表」として表現することもできる。

この表現の正当性は自明であるので、特に証明は付さない。

4.4 正規化データモデル

エンティティ・クラス **ex, ey** ∈ **E** において∀ey ∈ **ey** **|** ∃単射 r : ey → ex, ex ∈ **ex** のとき、**ey** は **ex** に従属的という。

任意の **ey, ex** ∈ **E** で **E** が正規化集合の場合、**ey** が **ex** に従属的で、かつ **ex** は **ey** に従属的でなければ、**E** は第三正規化集合であるという。

現実世界の実体と対応させて、アトリビュート a1,a2,・・・, 第三正規化エンティティ e1,e2, ・・・,が定義できて、これらが、クラス、インスタンスの定義と矛盾しない場合、この対応を**正規化データモデル**と呼ぶ。

エンティティ・クラス **ex, ey, ez** ∈ **E** において

(∀ey ∈ **ey** **|** ∃単射 r : ey → ex, ex ∈ **ex** **)** ∧ **(**∀ez ∈ **ez** **|** ∃単射 p : ez → ex, ex ∈ **ex** **)**のとき、**ey, ez** は **ex** に従属的という。

この場合も、定義により、**E** は第三正規化集合である。

エンティティ・クラス **eq, ex, ey** ∈ **E** において

(∀ey ∈ **ey** **|** ∃単射 r : ey → ex, ex ∈ **ex** **)** ∧ **(**∀ey ∈ **ey** **|** ∃単射 p : ey → eq, eq ∈ **eq** **)**のとき、**ey** は **ex, eq** に従属的という。

同様、この場合も、定義により、**E** は第三正規化集合である。

5. オブジェクトモデル

5.1 オブジェクト・クラスとその継承

P を任意のアトリビュート・クラスの集合、**M** をプログラムモジュールの集合とする。

m ∈ **M** が **p1, p2, p3** ・・・ **pm, q1, q2, q3** ・・・ **qn** ∈ **P** | m **: (p1, p2, p3** ・・・ **pm)** → **(q1, q2, q3** ・・・ **qn)** のとき、m をメソッド と呼ぶ。

m をメソッドの集合とする。

p1, p2, p3 ・・・ **pm** ∈ **P**, m1, m2, m3 ・・・ mn ∈ **m** のとき、

｛｛ **p1, p2, p3** ・・・ **pm** **},** {m1, m2, m3 ・・・ mn}} なる集合を、**オブジェクト・クラス**と呼ぶ。

あるオブジェクト・クラスを構成するアトリビュートを、このオプジェクト・クラスの**プロパティ**と呼ぶ。

オブジェクト・クラス **o1,o2** を構成するプロパティとメソッドがすべて等しいとき、

o1 = o2 と記述する。

o1, o2 をオブジェクト・クラス、**P1, P2** をアトリビュート・クラスの集合, **m1,m2** をメソッドの集合としたとき、

o1 ≠ o2 ∧ **P1** ⊆ **P2** ∧ **m1** ⊆ **m2,**

このとき、**o2** を **o1** の**継承 (Inheritance)** と呼ぶ。

o1 を **o2** の**上位クラス (Superclass),** 同様に、**o2** を **o1** の**下位クラス (Subclass)** と呼ぶ。

o1={{ p1, p2,・・・**pi },** { m1, m2,・・・mj }},

o2={{ p1, p2,・・・**pi+1, pi+2,**・・・**px },** **{** m1, m2・・・mj+1, mj+2,・・・my }} の時、

o2={ [o1], { pi+1, pi+2,・・・**px },** { mj+1, mj+2,・・・my }} と記す。

この記法の類推で、

ox={ [o1,o2,o3・・・**],{ p** ・・・**}**, { **m** ・・・}と記した時、**ox** を **o1,o2,o3**・・・ の**多重継承 (Multi-Inheritance)** と呼ぶ。

5.2 オブジェクトのインスタンス

p1, p2, p3 ・・・ **pm** ∈ **P,** m1, m2, m3 ・・・ mn ∈ **m** として、｛｛ **p1, p2, p3** ・・・ **pm** **},** {m1, m2, m3 ・・・ mn }} なるオブジェクト・クラス **o** において、

p1, p2, p3・・・,pm をそれぞれ **p1, p2, p3** ・・・ **pm** のそれぞれのインスタンスとする。

このとき、 o = ｛｛ (**p1**, p1), (**p2**, p2), (**p3**, p3) ・・・ (**pm**, pm) }, {m1, m2, m3 ・・・ mn }} と書いて、

o の (**p1**, p1), (**p2**, p2), (**p3**, p3) ・・・ (**pm**, pm) の値を取る**インスタンス**と呼ぶ。

∃(**pi,** pi), (**pj,** pj) ・・・|1≦i, j≦m で

IO(-1) (**(pi**, pi), (**pj**, pj), ・・・) :

o = ｛｛ (**p1**, p1), (**p2**, p2), (**p3**, p3) ・・・ (**pm**, pm) } → **IO** **((pi,** pi), (**pj,** pj), ・・・)

が単射の時、 **IO** を**オブジェクト・クラスのインスタンス化**と呼び、 (**pi,** pi), (**pj,** pj) ・・・を o の初期状態とよぶ。

当然ながら、

∃(**pi,** pxi), (**pj,** pyj) ・・・|1≦i, j≦m で

IO **((pi,** px), (**pj,** py), ・・・) **: o={{ p1, p2,**・・・**pm },** **{ m1, m2,**・・・**mn** }} → ϕ

即ち、インスタンス化が不可能なアトリビュート・インスタンスの組を定義することができる。

5.3 オブジェクト・インスタンスの発展

∃(**pi,** pi), (**pj,** pj) ・・・|1≦i, j≦m で

IO ((pi, pi), **(pj,** pj), ・・・) **:** o**={{ p1, p2,**・・・**pm }, { m1, m2,**・・・**mn** }} → o = ｛ **(p1,** p1), **(p2,** p2), **(p3,** p3) ・・・ **(pm,** pm) }

を なる初期状態を持つオブジェクト・インスタンスとにおいて、

∃ pix, piy で(**pi,** pix), (**pi,** piy) ・・・|1≦i≦m のとき、

D : o → D(o) なる写像が存在し、

o={ (**p1,** p1), ・・・ (**pi,** pix) ・・・ (**pm,** pm) }

D(o)={ (**p1,** p1), ・・・, (**pi,** piy) ・・・ (**pm,** pm) } のとき、定義から明らかなように、

IO(-1) (D(o)) = ◐

つまり、D(o)も、オブジェクト・クラス ◐ のオブジェクト・インスタンスの一つである。

この D をアトリビュート **pi** についての**単独の発展(Development)**と呼ぶ

D<-1> : D(o) → o は、自明で定義できる。

D<0> : o → o

を定義した上で、

Dd : o → Dd(o) なる写像 Dd(o)={ (**p1,** p1), ・・・, (**pi,** pid) ・・・ (**pm,** pm) } 全体の集合に D<-1>、D<0> を追加したものを ◧ とし

Dp, Dq ∈ ◧ のとき 演算 * を Dp*Dq(o) = Dp(Dq(o)) と定義すれば、

Dp, Dq, Dr ∈ ◧ のとき Dp(Dq(Dr(o))) = Dp(Dq*Dr(o))=Dp*Dq(Dr(o)) なので、結合則が成り立って、

演算 * によって、 ◐ 上に群が定義される。

しかも定義から明らかなように、Dp*Dq=Dq*Dp なので可換群である。

さらに一般化して、

D : o → D(o) なる写像が存在し、

o={ (**p1,** p1) , ・・・ (**pi,** pix) , ・・・ (**pj,** pjx) ・・・ (pm, pm) }

D(o)={ (**p1,** p1), ・・・, (**pi,** piy) ・・・, (**pj,** pjy) ・・・ (**pm,** pm) } のとき、

同じく

D<-1> : D(o) → o

D<0> : o → o

を定義した上で、

Dd : o → Dd(o) なる写像 Dd(o)={ (**p1,** p1), ・・・, (**pi,** pid) ・・・, (**pj,** pjd) ・・・ (**pm,** pm) } 全体の集合に D<-1>、D<0> を追加したものを ◧ とし

Dp, Dq ∈ ◧ のとき 演算 * を Dp*Dq(o) = Dp(Dq(o)) と定義すれば、同様に、結合則は成立し、演算 * によって、 ◐ 上に群が定義される。

しかしながら、定義上、Dp*Dq=Dq*Dp を保証するものはなく、非可換群と考えるのが一般的である。

発展が３個以上のアトリビュート・クラスについてのものである場合に一般化すれば、演算の順序が重要な意味を持つことがわかる。

一般的に、

IO(-1) **:** o1 → ◐

D: o1→o2

o1={ (**p1,** p11) , ・・・ (**pi,** pi1) , ・・・ (**pj,** pj1) ・・・ (**pm,** pm1) }

IO(-1) **:** o2 → ◐

D: o1→o2

o2={ (**p1,** p12) , ・・・ (**pi,** pi2) , ・・・ (**pj,** pj2) ・・・ (**pm,** pm2) }

とると、 ◐ は定義の中に ◧ を含んでいない。

すなわち、インスタンス化は、写像に与える変数のみで決定されるので、オブジェクト・クラスはステートレスである、と表現する。