形式的方法概観

谷 津 弘 一

1. はじめに

1968 年 Garmisch にて開かれた NATO 会議にて，開発されるソフトウェアの危機的状況（所謂ソフトウェア危機）が認識され，既存の工学で行なわれていたことと同様のことをソフトウェア開発の分野でも行ないたい，語弊があるのを承知でより平たく言うと，他の工業製品と同様にソフトウェアを作成したいという願望の下，「ソフトウェア工学」なる学問が提唱された．

形式的方法（formal methods）とは，記号論理学の力を借りて，作るべきソフトウェアの仕様，すなわち，ソフトウェアに要求されている性質あるいはソフトウェアに要求されている性質を満足するモデルを記述し，記述された仕様から実際のコードを導出していく方法である．ソフトウェア工学の成り立ちを考えると，形式的方法はソフトウェア工学の中核に据えられるテーマであると言えよう[*1]．

形式的方法に基づくソフトウェア開発において，仕様記述は必須の行為であり，仕様記述に用いられる形式仕様言語（formal specification languages）は形式的方法において最も基本的な道具である．そこで，本稿では，形式仕様言語に限定して形式的方法を概観することにする．形式仕様言語の分類には，記述する仕様の種類による次の基準がよく用いれられる．

性質指向（property-oriented）

ソフトウェアに要求されている**性質**を記述する言語で，

・OBJ[19]，Larch[20]，ASL[36]，ACT ONE[15]，CASL[11]等の代数仕様言語

・CSP[22]，CCS[28]，CCS の発展形である π-calculus[29,30]等の並行プロセス理論

等がこの範疇に入る．ACT ONE と CCS を組み合わせて作られた LOTOS[23]もここに分類される．

モデル指向（model-oriented）

ソフトウェアに要求されている性質を満足する**モデル**を記述する言語で，

・Z[35]，Object-Z[14]VDM_SL[24]B[9]等の集合論に基づく言語

等がこの範疇に入る．これらの言語では，記述対象とするソフトウェアを状態機械として記述するのが通常である．

VDM_SL を基にして作られた RSL[32]のように，性質指向とモデル指向の両方の性質を併せ持つ言語もある．RSL は，集合論に基づく記述もできれば，代数仕様や並行プロセスも記述できる，「all in one」の言語である．

本稿の構成は次のとおりである．上の分類基準の順序とは異なるが，モデル指向形式仕様言語の代表である Z をまず紹介し，その次に性質指向形式仕様言語の代表的である OBJ を紹介する．そして，形式的方法に対する産業界の対応や標準化の動向について触れ，これからのソフトウェア開発における形式的方法の必要性について議論する．

2. Z

Z は 1970 年代の終わりに Oxford 大学 Programming Research Group により開発された言語であり，先に述べた通り，集合論に基づき対象を状態機械として記述するスタイルをとる，モデル指向の形式仕様言語である．

状態機械は

・状態空間
・初期状態
・状態空間に作用する操作

の三つで表現されるが，Z ではこの三つをスキーマという Z 独特の記述単位を用いて記述する．

2.1 誕 生 日 帳

ここでは，簡単な例を記述しながら，Z の基本的な記法を説明する．記述するのは，次のような機能を持つ誕生日帳[35]である．

1. 誕生日帳に新たな知人の名前と誕生日を追加できる．
2. 誕生日帳から知人の名前と誕生日を削除できる．
3. 知人の名前を与えるとその人の誕生日がわかる．
4. 日付を与えるとその日が誕生日の知人の名前がわかる．

なお，説明の都合上，この誕生日帳の中では知人の名前は重複しないものとする．

基本型の宣言

さしあたってこの誕生日帳の記述に必要なのは，知人の名前と（誕生日を表す）日付である．実装の段階では，知人の名前や日付に何らかの構造が入るかも知れないし，日付の正当性のチェックもされるかも知れない．しかし，この段階の記述では，知人の名前や日付の構造，さらに，日付の正当性の定義は必要がない．そこで，知人の名前の集合（*Name* とする）と正当な日付の集合（*Date* とする）を使う，ということだけを明らかにしておく．

$$[Name, Date]$$

このようにして宣言される集合を，Z では**基本型（basic type）**と呼ぶ．

誕生日帳の状態空間の記述

上で宣言した基本型を用いて，誕生日帳の状態空間を記述する．ここでは，誕生日帳の状態空間が

・誕生日帳に誕生日が登録されている知人の名前の集合 *known*
・知人の名前とその誕生日の対の集合 *birthdaybook*

から成るものとして記述する[*2] 一人の人間には一つの誕生日しかないことと，*birthdaybook* は全ての知人の誕生日を網羅しているわけではないことに留意する．

BirthdayBook

$$known : \mathbb{P}\, Name$$
$$birthdaybook : Name \pfun Date$$
$$known = \mathrm{dom}\, birthdaybook$$

この記述形式が，**スキーマ（schema）**と呼ばれるものである．ボックスの中の短

い線の上部を宣言部，下部を述語部と呼ぶ．宣言部で，状態空間を構成する属性と属性の値の型を宣言し，述語部で，宣言した属性の値の間に成り立つ条件を記述する．

X と Y を型とし，f を X と Y の対の集合とする．このとき，f に現われる X の要素の集合を f の定義域と呼ぶ．また，f の中で，f 定義域にある X の要素 a に対して Y の要素が一意に定まるとき，f を部分関数と呼ぶ．f の定義域が必ずしも X と一致しないことに注意してほしい．

この記述では，

1. 属性 *known* が知人の名前の集合であることを宣言し，
2. 属性 *birthdaybook* が知人の名前から日付への部分関数である（(定義域にある）知人の名前からその誕生日が一意に定まる．ただし，全ての知人の誕生日を網羅しているわけではない）ことを宣言し，
3. *known* が，誕生日が *birthdaybook* にある知人の名前の集合になっていることを条件付けている．

状態空間を表すスキーマを**状態スキーマ**と呼ぶ．状態スキーマに記された条件は，初期状態はもちろん，どの操作を施された後の状態でも成り立っていなければならない．状態スキーマに記された条件を特に，**不変条件（invariant）**と呼ぶ．

誕生日帳の初期状態の記述

誕生日帳の初期状態については何も言及していなかったが，誕生日帳にはどの知人の誕生日も書き加えられていない，ということを記述しておけばよいだろう．

Init

$$BirthdayBook$$

$$known = \emptyset$$

初期状態を表すスキーマを**初期状態スキーマ**と呼ぶ．

宣言部で誕生日帳の状態スキーマの名前 *BirthDay* があるが，これにより誕生日帳の初期状態スキーマ *Init* の宣言部と述語部にそれぞれ，*Birthday* の宣言部と述語部が含まれる．

Init

$$known : \mathbb{P}\, Name$$
$$birthdaybook : Name \pfun Date$$

$$known = \mathrm{dom}\, birthdaybook$$
$$known = \emptyset$$

Init では *birthdaybook* の値に関する記述がないが，

$$\mathrm{dom}\ birthdaybook = known = \emptyset$$

であることから，*birthdaybook* は空集合であることがわかる．

誕生日帳に施される操作の記述

まず，新しい知人の名前と誕生日を誕生日帳に追加する操作を記述しよう．X から Y への部分関数は，X と Y の対の集合であるから，誕生日帳への追加は集合の和で表現できる．誕生日帳の中では知人の名前の重複は許されていないから，与えられ

た名前が誕生日帳の中に無いことをこの操作の前提としたい.

AddBirthday

$$\Delta BirthdayBook$$
$$name? : Name$$
$$date? : Date$$

$$name? \notin known$$
$$birthdaybook' = birthdaybook \cup \{name? \mapsto date?\}$$

操作を表すスキーマを**操作スキーマ**と呼ぶ.

この記述の中で, $\Delta BirthdayBook$ は次のスキーマを表している.

$\Delta BirthdayBook$

$$BirthdayBook$$
$$BirthdayBook'$$

Z では, スキーマ名に「′」を修飾して参照すると, そのスキーマで宣言されている全て属性の名前に「′」が修飾される.

Birthday′

$$known' : \mathbb{P}\, Name$$
$$birthdaybook' : Name \pfun Date$$

$$known' = \mathrm{dom}\, birthdaybook'$$

結局, $\Delta BirthdayBook$ は次のようになる.

$\Delta Birthday$

$$known : \mathbb{P}\, Name$$
$$known' : \mathbb{P}\, Name$$
$$birthdaybook : Name \pfun Date$$
$$birthdaybook' : Name \pfun Date$$

$$known = \mathrm{dom}\, birthdaybook$$
$$known' = \mathrm{dom}\, birthdaybook'$$

Z では慣習的に, 操作を施される前の状態空間の属性と操作を施された後の状態空間の属性を,「′」を付けることで区別する. ここでは, *known* と *birthdaybook* が操作を施される前の状態空間の属性であり, *known* ′と *birthdaybook* ′が操作を施された後の状態空間の属性である. また, これもやはり慣習であるが, 名前に「?」が付いた属性で入力を表し, 名前に「!」が付いた属性で出力を表す.「′」や「!」が付いた属性が出現しない条件を**事前条件**と呼び, 名前に「′」や「!」が付いた属性が出現する条件を**事後条件**と呼ぶ.

操作 *AddBirthday* は, 入力された名前が *birthdaybook* の中にない場合に, *birthdaybook* に入力された名前と誕生日の対 *name*?↦*date*? を追加する操作である.

なお, 初期状態の記述では *birthdaybook* の値に関する条件が陽に書かれなかったが, ここでは *known* に関する条件が陽に書かれていない. この場合も, 次のように

して $known'$ の値がどうなるかを導きだすことが出来る．

$$\begin{aligned} known' &= \mathrm{dom}\, birthdaybook' && (BirthdayBook' \text{の不変式}) \\ &= \mathrm{dom}(birthdaybook \cup \{ name? \mapsto date? \}) && (AddBirthday \text{の事後条件}) \\ &= known \cup \{ name? \} \end{aligned}$$

次に，誕生日帳から知人の名前と誕生日を削除する操作を記述しよう．この場合，入力は削除したい知人の名前だけで十分である．操作の事前条件は，入力である知人の名前が誕生日帳の中にあること，である．

Delete

$$\Delta BirthdayBook$$
$$name? : Name$$
$$name? \in known$$
$$birthdaybook' = \{name?\} \lhd birthdaybook$$

ここで，

$$\{ name? \} \lhd birthdaybook = \{ name : Name;\ date : Date \mid name \neq name? \land name \mapsto date \in birthdaybook \}$$

を表す．これは，操作を施される前の $birthdaybook$ から入力された知人の名前 $name?$ とその誕生日を削除し，名前が $name?$ と異なる知人の誕生日はそのまま残されることを表している．

最後に，与えられた知人の名前の誕生日を検索する操作と与えられた日付が誕生日の知人の名前を検索する操作を記述する．

FindBirthday

$$\Xi BirthdayBook$$
$$name? : Name$$
$$date! : Date$$
$$name? \mapsto date! \in birthdaybook$$

RemindBirthday

$$\Xi BirthdayBook$$
$$date? : Date$$
$$reminder! : \mathbb{P}\, Name$$
$$reminder! = \{ name : Name \mid name \mapsto date? \in birthdaybook \}$$

$\Xi BirthdayBook$ は次のスキーマを表す。

ΞBirthdayBook

$$BirthdayBook$$
$$BirthdayBook'$$
$$known = known'$$
$$birthdaybook = birthdaybook'$$

これは，操作によって状態空間の属性の値が変わらないことを表している．

2.2 Z の限界

対象を状態機械として記述するという，Z を始めとするモデル指向形式仕様言語に共通の記述スタイルは，記述対象を単一の状態空間を持つ状態機械として記述することを強いる．これは，逐次処理型のソフトウェアの仕様を記述するのであればどうということはないのだが，分散協調型のソフトウェアの仕様を記述するとなると致命的な問題を引き起こす．

試しに，CORBA に提案されている Mobile Agent Facility[13]の（MAF）ファインダ（MAFFinder）を例にとり，ちょっとした記述実験をしてみよう．まず，議論を簡単にするため区域（region）や場（place）を捨象し，ネットワークにはエージェントシステムとファインダが混在し，エージェントシステムには複数のエージェントが存在するものとする．また，エージェントシステムでないシステムや，エージェントでないオブジェクトなども一切考慮にいれないことにする．

$[AgentId, AgentSystemId, MAFFinderId]$

Agent

$id : AgentId$

$\ldots$

AgentSystem

$id : AgentSystemId$

$agents : \mathbb{P}\, Agent$

$\ldots$

MAFFinder

$id : MAFFinderId$

$\ldots$

Network

$systems : \mathbb{P}\, AgentSystem$

$finders : \mathbb{P}\, MAFFinder$

ここで，「与えられた識別子を持つエージェントが存在するエージェントシステムの識別子を返す」というファインダのエージェント探索機能を記述してみよう．ファインダの探索範囲は自身を含むネットワーク全域であるから，この機能を記述するのに必要な状態空間はネットワークである．エージェント探索の要求を出すクライアントをエージェントに限定すると，エージェント探索機能は次のように記述されるだろう．

$$
\begin{array}{|l}
\hline
\textit{FindAgent} \\
\hline
\Xi \textit{Network} \\
\textit{client?} : \textit{AgentId} \\
\textit{server?} : \textit{MAFFinderId} \\
\textit{target?} : \textit{AgentId} \\
\textit{address!} : \textit{AgentSystemId} \\
\hline
\exists\, c : \textit{Agent};\ s : \textit{AgentSystem} \bullet \\
\quad s \in \textit{systems} \land c \in s.\textit{agents} \land c.\textit{id} = \textit{client?} \\
\exists\, f : \textit{MAFFinder} \bullet \\
\quad f \in \textit{finders} \land f.\textit{id} = \textit{server?} \\
\exists\, t : \textit{Agent};\ s : \textit{AgentSystem} \bullet \\
\quad s \in \textit{systems} \land s.\textit{id} = \textit{address!} \land t \in s.\textit{agents} \land t.\textit{id} = \textit{target?} \\
\hline
\end{array}
$$

しかし，よく考えてみると，この記述はおかしい．例えば，エージェント探索の前後でネットワークの状態が変化しないと宣言しているが，実際にはネットワークのそこかしこで構成要素（この例の場合にはエージェント，エージェントシステム，ファインダしかないが）同士の相互作用があり，エージェント探索操作が施されている間であってもネットワークの状態は変化する．それでは $\Delta \textit{Network}$ と宣言すればよいかというと，そうでもない．というのも，事後条件の記述が極めて難しいからである．

ファインダにエージェントの位置に関する情報をエージェントの識別子からエージェントシステムの識別子への部分関数のような形で持たせれば，エージェント探索機能の記述は非常に容易になるが，そうするとエージェント移動の様子を記述する時に同種の問題にぶつかる．何よりも，モデルが記述言語により制限されるというのは本末転倒以外の何物でもない．

Z のようなモデル指向形式仕様言語にとって，分散協調型ソフトウェアの仕様記述は極めて難しい問題であると言えよう．

ここ数年，モデル指向形式仕様言語とプロセス代数を組み合わせた仕様記述方法が盛んに研究されている[26,34]．この仕様記述方法を用いれば，個々の構成要素の逐次処理の部分をモデル指向形式仕様言語で，構成要素同士の相互作用をプロセス代数でそれぞれ記述することができるので，モデル指向形式仕様言語だけでは不可能と思われる分散協調システムの仕様記述が可能になる．

この分野の研究は始まったばかりであり，構成要素の相互作用を記述する仕組みを工夫する必要がある（プロセス代数は並行計算の理論でありソフトウェアの仕様の記述に直接用いるには基本的すぎるので，実際のソフトウェアを記述するとその記述量が莫大なものになる）等解決すべき問題はまだまだ多いが，その行く末に期待したい．

3. OBJ

OBJ は代数仕様言語の名前ではなく，ある特定の代数仕様言語族の名前である．OBJ 0 から始まって，OBJT，OBJ 1，OBJ 2，OBJ 3 と様々な代数仕様言語が開発されてきた．これらの言語族が OBJ 一族というわけである．

本節では，まず代数仕様言語についての簡単な説明をした後，簡単な例を基にして，

OBJ 族の一つである OBJ 3[19]の紹介を行なう．そして，OBJ 3 自身には反映されていないが，仕様を記述するのに便利な道具立てである隠蔽ソート代数（hidden algebra）[18]と書換え規則（rewriting rule）[27]を紹介し，最後に OBJ の限界について触れる．

3.1 代数仕様記述法

代数仕様記述法は，1970 年代の半ばに等式論理（equational logic）という，等式のみを述語に持つ，表現力は乏しいが極めて性質のよい論理に基づいて，抽象データ型を厳密に表現する方法として提案された．抽象データ型は，データ型からデータの表現形式やデータ型上の演算の実現手段を車掌したもので，演算の振る舞いは演算から生成される項の間の関係によって定められる．データ型は，データの集合（台集合）とその上の演算を一体化させたものであり，台集合，台集合上の演算，演算から生成される項の間の等式の三つで表現される．台集合とその上の演算を併せたものを**指標（signature）**と呼ぶ．データ型は数学的には代数として捉えられる．これが代数仕様記述法と呼ばれる所以である．

代数仕様記述法に基づき記述された仕様を**代数仕様**と呼び，代数仕様を記述するための言語を**代数仕様言語**と呼ぶ．代数仕様はデータ型の「集まり（数学的には，集合ではなく圏として捉えられる）」を表すが，この集まりには，次の二つの条件

1. 台集合の要素（項）は，宣言された演算だけから生成される．
2. 二つの項の値はその間の等式が定義された等式群から導出されるときに限り等しく，かつその場合には必ず二つの項は同じ要素になる．

を満たすデータ型が，名前替えによる指標の違いを除いて一意に定まるという性質がある．このデータ型は**始代数（initial algebra）**と呼ばれる．代数仕様の意味（これが代数仕様が表す抽象データ型）を始代数で与える意味論を始代数意味論と呼ぶ. OBJ 3 は始代数意味論に基づく．

代数仕様で宣言されている項の間の等式を，左辺から右辺への項の簡約規則と見做すことにより，項の値の計算ができる．どの簡約規則を用いてもそれ以上簡約できない項を**正規形（normal form）**と呼ぶ．代数仕様における項の値とは，項の正規形のことであり，代数仕様の実行とは仕様に記述された等式を簡約規則と見做して項の正規形を求めることである．記述された等式に基づき仕様の実行を行なう系を，**項書換え系（rewriting system）**と呼ぶ.

3.2 OBJ 3

代数仕様は抽象データ型を表すが, データ型の集まりをも表すと前節で述べた. OBJ 3 の記述単位には，抽象データ型を表す記述単位とデータ型の集まりを表す記述単位がある．前者は**オブジェクト（object）**[*3] と呼ばれ，後者は**セオリ（theory）**と呼ばれる．

まず最初に，簡単な例を基にしてオブジェクトの記法の説明をしよう．ここで紹介するのは，スタックのオブジェクトである[*4]

```
object STACK is
  sorts Elt NeStack Stack .
  subsorts NeStack < Stack .
  op empty : -> Stack .
  op push : Elt Stack -> NeStack .
  op pop : NeStack -> Stack .
  op top : NeStack -> Elt .
  var E : Elt .
  var S : Stack .
  op pop(push(E,S)) = S .
  op top(push(E,S)) = E .
endo
```

このオブジェクトの行毎の説明を下に挙げる．

1 行目　「STACK」と名付けられたオブジェクトの宣言の開始．

2 行目　スタックの記述に必要な台集合（これをソートと呼ぶ）の宣言．ここで宣言されているソートは，スタック(**Stack**)，空でないスタック(**NeStack**)，スタックに納める要素（**Elt**）の三つである．

3 行目　部分ソートの宣言．この宣言により，NeStack の項は Stack の項でもあると見做される．

4 行目　空のスタックを表す定数 **empty** の宣言．矢印（−>）の左側にソートがない演算は定数と見做される．

5 行目　要素をスタックに押し込む演算 **push** の宣言．**push** は **Elt** と **Stack** の項から **NeStack** の項を構成する演算である．

6 行目　空でないスタックからスタックへの演算 **pop** の宣言．

7 行目　空でないスタックからスタックに納める要素への演算 **top** の宣言．

8 行目　等式の記述に使われる変数の宣言．

9 行目　（同上）

10 行目　演算 **pop** の振る舞いの定義．この等式により，**pop** が与えられた空でないスタックから最後に押し込まれた要素を抜き取る演算であることがわかる．

11 行目　演算 **top** の振る舞いの定義．この等式により，**top** が与えられた空でないスタックから最後に押し込まれた要素を取り出す演算であることがわかる．

12 行目　オブジェクトの宣言の終了．

記述の開始と終了の行を除いて，オブジェクトとセオリの構文上の差はない．上のオブジェクトの記述の最初と最後の行を次のように変えれば，スタックのセオリを宣言できる．

```
theory STACK is
  （オブジェクト　STACK　の 2 行目　〜　11 行目）
endth
```

最初の行が「STACK」と名付けられたセオリの宣言の開始を表し，最後の行がセオリの宣言の終了を表す．

OBJ 3 には記述された仕様から項書換え系を作り出す処理系がある．スタックのオ

ブジェクトから作り出された項書換え系を使うと，次のような具合に項の値の計算ができる．

```
    top(pop(push(e2,push(e1,empty))))
→   top(push(e1,empty))
→   e1
```

3.3 隠蔽ソート代数

データ型の場合，二つの要素の等価性をその値で判断すればよいが，二つのシステムの等価性をデータ型の場合と同じようにして考えるわけにはいかない．たとえ異なる内部状態を持っていても，あらゆる刺激に対して同じ振る舞いをする二つのシステムは等価であると判断したい．

OBJ 言語族の研究開発グループのリーダーである Goguen 教授は，そのようなシステムの等価性を判断するため，振る舞い等価性に基づく隠蔽ソート代数を提唱した．

隠蔽ソート代数では，ソートは通常のデータ型を表す**可視ソート（visible sort）**と（オブジェクト指向におけるオブジェクトのように）内部状態を隠し持つ対象の集まりを表す**隠蔽ソート（hidden sort）**に分かれる．同じ可視ソートの二つの項の等価性は値の等価性に帰着され，同じ隠蔽ソートの二つの項の等価性は振る舞いの等価性に帰着される．

ここで，振る舞いの等価性がどういうものか，説明しておかなければなるまい．演算の宣言において，矢印（->）の左側にあるソートの列をアリティ，右側にあるソートをコアリティと呼ぶことにする．そして，コアリティが隠蔽ソートで，アリティにもその隠蔽ソートが存在する演算を**手続き（method）**と呼び，コアリティが可視ソートでアリティに隠蔽ソートが存在する演算を**属性（attribute）**と呼ぶ．手続きは隠蔽ソートの項が隠し持つ状態を遷移させるような演算であり，属性は隠蔽ソートの項が隠し持つ状態についての情報を提供する演算である．

手続きの作用による状態の遷移の履歴を仮に「境遇」と呼ぶことにしよう．隠蔽ソートの二つの項が，どの「境遇」を選んでも，その「境遇」の下であらゆる属性の値が等しければ，その二つの項は振る舞い等価であると言う．

次に示すのは，Z の説明に用いた誕生日帳を隠蔽ソート代数に基づいて OBJ 3 の記法に基づいて記述したものである．

```
obj BIRTHDAYBOOK is
  hsort BirthdayBook .
  sorts Person Persons .
  sorts Error Date EDate.
  subsorts Person < Persons .
  subsorts Error < Date EDate .
  op empty : -> Persons .
  op new : -> BirthdayBook .
  op error : -> Error .
  op _,_ : Persons Persons -> Persons [assoc comm idem id: empty] .
  bop add : Person Date BirthdayBook -> BirthdayBook .
  bop delete : Person BirthdayBook -> BirthdayBook .
```

```
  bop find : Person BirthdayBook -> EDate .
  bop remind : Date BirthdayBook -> Persons .
  vars P P' : Person .
  vars D D' : Date .
  var BB : BirthdayBook .
  bceq add(P,D,BB) = BB if find(P,BB) =/= error .
  beq  delete(P,new) = new .
  bceq delete(P,add(P',D',BB)) = BB if P == P' .
  bceq delete(P,add(P',D',BB)) = add(P',D',delete(P,BB)) if P =/= P' .
  eq  find(P,new) = error .
  ceq find(P,add(P',D,BB)) = D' if P == P' .
  ceq find(P,add(P',D,BB)) = find(P,BB) if P =/= P' .
  eq  remind(D,new) = empty .
  ceq remind(D,add(P,D',BB)) = P,remind(D,BB) if D == D' .
  ceq remind(D,add(P,D',BB)) = remind(D,BB) if D =/= D' .
endo
```

ここで,

- **hsort** で宣言された **BirthdayBook** が隠蔽ソートである.
- **bop** で宣言された演算のうち，**add** と **delete** は手続きであり，**find** と **remind** は属性である.
- **beq** は振る舞い等価性を表す等式であり，**bceq** は条件付き振る舞い等価性を表す等式である.

この記述は **BirthdayBook** の項を状態機械として記述している．しかし，Z を用いた記述例と異なり，ソート **BirthdayBook** の項の状態が陽に記述されていないことに注意してほしい．同じ状態機械を記述しても，モデル指向と性質指向の違いがここに現われている.

3.4 書換え規則

やはり OBJ 言語族の研究開発グループの指導的立場にいる José　Meseguer 博士は，等式とは異なる左辺から右辺への書換えを表す述語（書換え規則）を等式論理に導入した書換え論理を提唱し，書換え論理に基づく言語 Maude を開発した．書換え規則を用いると，非決定的な値の遷移を容易に記述することができる.

次に示す例は，集合から要素を一つ取り出す演算を記述したものである．記述を簡単にするため，集合の上の操作として，和を取る演算と所属関係（membership relation）を表す述語しか記述していない.

```
obj SET is
  sorts Elt Set .
  subsort Elt < Set .
  op empty : -> Set .
  op _,_ : Set Set -> Set [assoc comm idem id: empty] .
  op _in_ : Elt Set -> Bool .
  vars E E' : Elt .
```

```
  vars S S' : Set .
  eq E in empty = false .
  eq E in E' = (E == E') .
  eq E in (S,S') = (E in S) or (E in S') .
endo

obj CHOICE is
  including SET .
  sorts Error Result .
  subsorts Error Elt < Result .
  op error : -> Error .
  op choice : Set -> Result .
  var E : Elt .
  var S : Set .
  eq   choice(empty) = error .
  rule choice(E,S) => E .
endo
```

ここで，**including** で外で定義されたオブジェクト **SET** を参照することを表している．**rule** で始まる文が等式とは異なる値の変化を表す書換え規則である．

この変化の関係は等式では表現できない．

もし等式で表現すると，

$$a = choice(a,(b,c)) = choice(b,(a,c)) = b$$

となり，**Elt** の全ての要素が等しくなってしまう．

書換え規則は，局所的な部分の値の変化を記述できるという点で，Z などのモデル指向形式仕様言語では不可能と思われる分散強調システムの仕様を記述するにあたって有用な道具立てになるだろうと思われる．

3.5 OBJ 3 の限界

実行可能であることから，OBJ 3 には数々の利点がある．例えば，検証の容易さなどはその一つであろう．仕様の実行は等式論理に基づく推論と見做すことができる．等式論理が良い性質を備えていることに加えて項書換え系という推論エンジンを持つことから，集合論に基づくモデル指向の仕様に比べて，仕様の性質の検証を非常に容易に行なえるのである．OBJ 3 を使ってプログラムの抽象解釈を行なう，すなわち，OBJ 3 で記述した抽象プログラムを項書換え系で抽象実行してプログラムの性質を解析する，というのはなかなか興味深い方法であると思われる．

しかし，OBJ 3 には土台にしている等式論理の表現力の低さから来る致命的な欠点がある．例として，述語 $p(a, b)$ を満たす要素 a と b がそれぞれ与えられた二つの集合 S と T に存在すれば真，そうでなければ偽となる述語を考えてみよう．

Z を使えば，次のように直接的に書くことができる．

$$
\begin{array}{|l}
p : Elt \times Elt \to Bool \\
q : \mathbb{P}\, Elt \times \mathbb{P}\, Elt \to Bool \\
\hline
\forall S, T : \mathbb{P}\, Elt \bullet \\
\quad q(S, T) \Leftrightarrow (\exists a, b : Elt \bullet a \in S \land b \in T \land p(a, b))
\end{array}
$$

しかし，OBJ 3 ではこのように直接的には書けないのである[*5]．OBJ 3 で書くと，次のような具合になる．

```
obj EXAMPLE is
  including SET .
  op p : Elt Elt -> Bool .
  op q : Set Set -> Bool .
  vars X Y : Elt .
  vars S T U : Set .
  eq q(empty,S) = false .
  eq q(S,empty) = false .
  eq q(X,Y) = p(X,Y) .
  eq q(X,(S,T)) = q(X,S) or q(X,T) .
  eq q((S,T),U) = q(S,U) or q(T,U) .
endo
```

これでは，$q(S, T)$ の真偽を判定するアルゴリズムを記述するのとさほど変わりがない．

OBJ 3 に限らず，代数仕様言語を試した人達は異口同音に「関数型言語でプログラミングするのと何処が違うのかよくわからない」と言う．これは，土台となる等式論理の表現力があまりにも低いので，代数仕様言語では記述対象に求められている性質を直接的に記述することができない，すなわち，記述対象に求められている性質を導き出せるような等式群を工夫して記述しなければならないからであろう．関数型言語によるプログラミングと差がなくて当然なのである．

このように考えると，OBJ を始めとする代数仕様言語はプログラムの検証のための言語であり，仕様記述のための言語ではない，ということを強く感じずにはいられない．

4. 産業界の対応

現在，形式的方法は極めて微妙な状況に置かれている．それは，必要性は認められているけれども使われていない，というものだ（使われていないけれども必要性は認められている，と言う方が形式的方法の未来に希望が持てるかもしれない）．York 大学の John MacDermid 教授はこの状況を「over sold, under used」[25]と評したが，なるほど，言い得て妙である．

IBM 社の CICS（Customer Information Control System）やヒースロー空港の離発着管理システムなど，成功例が少なくない割に，形式的方法は産業界では使われていないらしい[*6]．

しかし，捨てる神あれば拾う神あり，形式的方法に注目する企業も存在する．不思議に思われるかも知れないが，Daimler-Benz 社（今や Chrysler 社と合併し Daimler-

Chrysler 社とその名前を変えたが，本稿ではこの古い名前で呼ぶことにする）は世界で最も形式的方法に力を注いでいる企業の一つである．

Daimler-Benz 社は，自分達が必要とする構文要素を Z に追加した自前の形式仕様言語をとその「処理系」（残念ながら，どのような処理（型検査，検証支援，コード導出，テストケース生成，等々）をするものなのかは筆者は知らない）を持っている．Daimler-Benz 社はそれらをソフトウェア開発に活用しているそうで，Daimler-Benz 社による形式的方法を用いたソフトウェアの開発事例がいくつか報告されている[2]．また，1998 年の後半，欧州の大学や研究所から多数の形式的方法の研究者が Daimler-Benz 社に移ったとも聞いている．その他にも，1998 年に Berlin で開かれた Z User Meeting（略称 ZUM 98：形式的方法に関する国際会議で最も大きなものの一つである）のスポンサーになったり，Z の ISO 標準化活動に積極的に参加をしているなど，Daimler-Benz 社は形式的方法に対して様々な貢献をしている．

5. 形式仕様言語の標準化活動

前節にて Daimler-Benz 社が Z の ISO 標準化活動に参加していると述べたが，現在，いくつかの形式仕様言語の ISO 標準ができており，いくつかの形式仕様言語の ISO 標準化活動が行われている．

既に ISO 標準ができているものの例として，LOTOS，VDM_SL（構造化機構を除く部分）等がある．また，現在 Z，VDM_SL（構造化機構に関する部分），E-LOTOS（LOTOS に実時間の概念を導入したもの）等の ISO 標準案が作成されつつあるところである．

6. お わ り に

朝日新聞のさるコラム[37]にて，ボーイング 777 の設計にまつわる話が紹介されていた．それによると，ボーイング 777 の設計において，航空機，部品，素材メーカーの技術者や航空会社の乗務員，整備員といった多種多数の人間が共通の図面の下で議論がなされ，その結果，ボーイング 777 という新規就航機としては極めて稀な，初期不具合が少ない「名機」が誕生したという．

web 上で航空機の図面が公開され議論がなされたというので，朝日新聞のコラム子は「web の積極的利用が人類の知恵の質的な向上を促進する」と評していたが，筆者の意見はこれとは異なる．この事例で最も本質的なところは，図面という仕様の推敲が十分なされた事だと筆者は思う（この場合の仕様は，要求されている性質を満足するモデルである）．航空機を作成する前の図面の段階で，相当数の不具合の修正ができたからこそ，ボーイング 777 という「名機」が誕生したのではなかろうか．

翻って，ソフトウェア開発を眺めるとどうだろうか．ボーイング 777 の場合には図面という的確に記述された仕様があったが，現実のソフトウェア開発で図面ほど的確に記述された仕様が作られるだろうか．また，ボーイング 777 の開発で行われたような，仕様を納得の行くまで煮詰めるという作業がなされるだろうか．ソフトウェア開発で発生するトラブルの多くが的確に仕様を記述し，コーディングに入る前に十分に仕様を推敲しておくことで回避できるのではないだろうか．

仕様を明確に記述し，十分に推敲するには，自然言語でも不可能ではないが，自然言語よりも形式仕様言語を用いる方が容易である．形式仕様言語では，曖昧な表現は許されないので，開発するソフトウェアを明瞭に理解していなければその仕様を記述することはできない．逆に言えば，開発するソフトウェアの仕様を形式仕様言語で記述することで，それに対する明瞭な理解が得られる．この理解が仕様の推敲に大いに役立つ．

一方，自然言語では，曖昧な表現を残したまま記述が進みやすく，開発するソフトウェアを明瞭に理解していなくともその仕様を記述できてしまう．開発するソフトウェアの仕様を記述してもそれに対する理解が明瞭になるとは限らない．開発対象に対する明瞭な理解なしには仕様の推敲は難しい．

Z や OBJ がそうであるように，既存の形式仕様言語には何らかの限界がある．しかし，モデル指向形式仕様言語とプロセス代数を組み合わせる試みが示すように，いくつかの形式仕様言語を適宜組み合わせて使用することで，その限界を広げることができる．

海外には

・医療関係
・ミサイル制御などの軍事関係
・原子力制御関係

のソフトウェアを形式的方法で開発することを義務付けている国が多いと聞く．

これらのソフトウェアにおいては，ソフトウェアのほんの僅かな不具合が人命や社会の安全を大きく脅かす．テストは，ソフトウェアに不具合があることを示すことができるだけであり，ソフトウェアに不具合が起きないことを示すことはできない．このような安全性重視のソフトウェアの開発に，人命や社会の安全を脅かすような不具合が起きないことを示すことができるやり方，すなわち，形式的方法が義務付けられているというのは，当然であろう．

ソフトウェアが社会に深く入り込みつつある現在，ソフトウェアの不具合に対するソフトウェア開発者の責任は重い．不具合がなるべく出ないようなやり方でソフトウェアを開発するのはソフトウェア開発者の義務だと思うのだが，如何がだろうか．

＊1　現在，NATO 会議はソフトウェア工学国際会議（International Conference on Software Engineering：通称 ICSE）と名前を変え，ソフトウェア工学に関して最も権威ある国際会議となっているが，そのいくつかのセッションは必ず形式的方法がテーマとなっている．

＊2　付けられた名前から推察できるように，この誕生日帳の状態空間は *birthdaybook* のみで十分である．しかし，これもやはり Z を説明する都合上，*known* を状態空間に加えてある．ご承知おきいただきたい．ただ，*known* があると状態空間の上の操作の記述が（ほんの僅かではあるが）楽になることも事実である．

＊3　オブジェクト指向でいうところのオブジェクトとは異なる．混同しないように注意されたい．

＊4　代数仕様の例はスタックしかないのか，とよく言われる．コンパクトなサイズであるにもかかわらず OBJ 3 の構文の説明に必要十分な情報を備えているので，紹介するほうにとってはこんなに便利な例はないのだが，いつも同じものを紹介される読者はたまったものではなかろう．

＊5　スコーレム化（skolemization）という技を使うこともできるが，それで書けても，直接的に書けない，という点では変わりはない．

＊6　筆者は，形式的方法が使われていないのは日本だけで，外国ではそれなりに使われているの

ではないかと思っていたが，どうもそうでもないらしい．先日，形式的方法に関するさる国際会議にて NIST（National Institute of Standard and Technology）の人と少し話をしたところ，米国でも産業界では形式的方法はあまり使われておらず技術移転もままならないのだということであった．

参考文献

［1］ 形式的方法バーチャルライブラリホームページ：
http：//www.comlab.ox.ac.uk/archive/formal-methods/
世界中の形式的方法に関する情報がここから得られる．

［2］ Formal Methods Europe（Dutch hub）ホームページ
http：//www.fme-nl.org/
形式的方法の紹介だけでなく，ツール類や適用事例のデータベースがある．

［3］ Larch ホームページ：
http：//www.research.digital.com/SRC/Larch/Larch-home.html

［4］ Z ユーザグループホームページ：
http：//www.comlab.ox.ac.uk/archive/z/zug.html

［5］ B ユーザグループホームページ：
http://estas1.inrets.fr：8001/ESTAS/BUG/WWW/BUGhome/BUGhome.html

［6］ RAISE ホームページ：
http：//dream.dai.ed.ac.uk/raise/

［7］ TLA（Temporal Logic of Actions）ホームページ：
http://www.research.digital.com/SRC/personal/Leslie_Lamport/tla.html
本稿では触れなかったが，リアルタイムシステムや制御システムの仕様記述に，時制論理（temporal logic）に基づく言語が最近よく使われている．TLA は，時制論理に基づく代表的な形式仕様言語の一つである．

［8］ Extended ML ホームページ：
http：//www.dcs.ed.ac.uk/home/dts/eml/

［9］ Abrial, J.R.：The B Book-Assigning Programs to Meanings, Cambridge University Press, 1996.

［10］ Abrial, J.-R., Börger, E. and Langmaack, H.(eds.)：Formal Methods for Industrial Applications-Specifying and Programming the Steam Boiler Control, LNCS 1165, Springer-Verlag, 1996.
Abrial たちが出題したボイラー制御システムの仕様記述問題の解答集． 時制論理に基づく言語を含めて複数の言語を用いた記述例が多い．

［11］ CoFI：CASL The Common Algebraic Specification Language Summary ver 1.0.
available at **http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/**
CoFI（The Common Framework Initiative）とは，産業界で使用される代数仕様言語を提案するのを目標に，今から 2〜3 年前に結成された団体である．

［12］ Craigen, D., Gerhart, S. and Ralston, T.：An International Survey of Industrial Applications of Formal Methods, NIST GCR 93/626(vol. 1 and 2), U.S. National Institute of Standards and Technology, 1993.
available by
ftp：//ftp.ora.on ca/pub/doc/93-626-v1.ps.Z（vol.1）
ftp：//ftp.ora.on ca/pub/doc/93-626-v2.ps.Z（vol.2）
形式的方法の産業界の適用の様子を調べた超労作．

［13］ Crystaliz, Inc., General Magic, Inc., GMD FOKUS and International Business Machines Corporation：Joint Submission：Mobile Agent Facility Specification, 1997.

［14］ Duke, R., Rose, G. and Smith, G.：Object-Z：A Specification Language Advocated for the Description of Standards, Technical Report 94-45, Software Verification Research Centre, School of Information Technology, The University of Queensland, 1994.
available at **http://svrc.it.uq.edu.au/Object-Z/pages/What_is_Object-Z.html**
本稿では紹介しなかったが，Object-Z とは，状態スキーマ，初期状態スキーマ，操作スキーマを一括りにして状態機械を陽に表現するクラス構文を Z に追加したものである．

［15］ Ehrig, H., Fey, W. and Hansen, H.：An Algebraic Specification Language with two levels of semantics, Bericht No. 83-03, Fachbereich 20-Informatik, Technische Univ. Berlin, 1983.

［16］ Ehrig, H and Mahr, B.：Fundamentals of Algebraic Specification 1：Equations and

Initial Semantics, EACTS Monographs on Theoretical Computer Science vol. 6, Springer-Verlag, 1985.
代数仕様研究者の必読本. 最後の章に, ACT ONE の構文と意味論が紹介されている.
[17] Ehrig, H and Mahr, B.: Fundamentals of Algebraic Specification 2: Module Specification and Constraints, EACTS Monographs on Theoretical Computer Science vol. 21, Springer-Verlag, 1990.
[18] Goguen, J.A. and Malcolm, G: A Hidden Agenda, UCSD Technical Report CS 97-538, Department of Computer Science and Engineering. University of Calfornia at San Diego, 1997. to appear in special issues of Theoretical Computer Science on Algebraic Engineering.
[19] Goguen, J.A., Winkler, T., Meseguer, J., Futatsugi, K. and Jouannaud, J.-P.: Introducing OBJ, Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, 1992.
available at **http**://**www-cse.ucsd.edu**/～**goguen**/**pubs**/
OBJ 3 のマニュアル.
[20] Guttag, J.V., Horning, J.J., Garland, S.J. and Jones, K.D.: Larch: Languages and Tools for Formal Specification, Texts and Monographs in Computer Science series, Springer-Verlag, 1993.
[21] Hayes, I.(ed.)：Specification Case Studies, Prentice-Hall, 1987.
形式的方法を用いた事例集の中で, 最も古典的なもの.
[22] Hoare, C.A.R.: Communicating Sequential Processes, Prentice-Hall, 1985.
[23] ISO / IEC: IS 8807: Information Processing Systems-Open System Interconnection-LOTOS-A Formal Description Technique based on the Temporal Ordering of Observational Behaviour, ISO/IEC, Geneva, Switzerland, 1988.
[24] Jones, C.B.: Systematic Software Development using VDM 2nd. edition, Prentice-Hall, 1990.
available by **ftp**://**ftp.cs.man.ac.uk**/**pub**/**cbj**/**ssdvdm.ps.gz**
[25] MacDermid, J.: Towards Industrially Applicable Formal Methods: Three Small Steps, and One Giant Leap, invited talk at 2nd International conference on Formal Engineering Methods (ICFEM'98), Brisbane, 10 Dec 1998.
[26] Mahony, B.P. and Dong, J.S.: Blending Object-Z and Timed CSP: An introduction to TCOZ, Proc. 20th International Conference on Software Engineering (ICSE'98). IEEE Computer Society Press, 1998.
[27] Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency, Theoretical Computer Science, Vol. 96, 1992.
[28] Milner, R.: Communication and Concurrency, Prentice-Hall, 1989.
[29] Milner, R., Parrow, J. and Walker, D.: A Calculus of Mobile Processes Part 1 LFCS technical report, ECS-LFCS-89-85, Edinburgh University, 1989.
[30] Milner, R., Parrow, J. and Walker, D.: A Calculus of Mobile Processes Part 2 LFCS technical report, ECS-LFCS-89-86, Edinburgh University, 1989.
[31] RAISE Method Group: The RAISE Method Manual, Prentice-Hall, 1995.
[32] RAISE Language Group: The RAISE Specification Language, Prentice-Hall, 1992.
[33] Sannella, D.T. and Tarlecki, A.: Formal program development in Extended ML for the working programmer, Proc. 3rd BCS/FACS Workshop on Refinment, Springer-Verlag, 1991.
[34] Smith, G.: A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems, Proc. FME'97, Industrial Applications and Strengthened Foundations of Formal Methods, LNCS 1313, Springer-Verlag, 1997.
[35] Spivey, J.M.: The Z Notation: A Reference Manual 2nd. ed., Prentice-Hall, 1992.
available at **http**://**spivey.oriel.ox.ac.uk**/～**mike**/**zrm**/**index.html**
[36] Wirsing, M.: Structured algebraic specifications: A kernel language, Theoretical Computer Science, Vol. 43, 1986.
[37] 朝日新聞夕刊コラム「経済気象台」, 98 年 11 月 9 日付,
http://**www.asahi.com**/**market**/**column**/**column.html**

執筆者紹介　谷　津　弘　一（Hirokazu Yatsu）
1985年東京大学理学部数学科卒業．同年日本ユニシス（株）入社．現在，情報技術部技術研究開発室所属．情報処理学会，日本ソフトウェア科学会各会員．