人工知能と対話技法 22－2
（1981. 9. 18）

オブジェクト・オリエンティド言語に基づく ハードウェア記述システム

竹内彰一
三菱電機株式会社　中央研究所

1. はじめに.

VLSIに代表されるような近年のデバイスの進歩により、計算機のハードウェアな、より大きく、より複雑化してきており、その結果それらの設計過程もますます複雑化して来ている。人間の処理できる複雑度には限界があり、計算機ハードウェアの設計もまた、ソフトウェアやVLSIの設計のように[1]、この限界に近づきつつあるようである。このいわゆるComplexity Barrierを乗り越えるためには、記述言語だけでなく、設計過程の全般にわたって計算機が設計者を支援するような設計環境を構築する必要があると考えられる。

本報告ではOODE（Object Oriented Description Environmentの略）と呼ぶ記述システムについて報告する。OODEは関数的仕様記述とかドキュメンテーション等よりは、設計者がハードウェアのモデルを容易に構成、シミュレート、評価でき、モデルを徐々に発展させていくことが可能になるようなインターラクティブな設計環境を提供することを目的としている。

OODEの特徴は次の3つである。

1) モジュール構造

OODEはオブジェクト・オリエンティド言語という非常に高いモジュール構造をもった言語をベースに設計されている。モジュールは記述の単位でありCONLAN[2)3)4)]で採用されている抽象データ型に類似している。モジュールを定義するには2通りの方法があり、一つは、それを階層的に他のモジュールから構成する方法であり、他はそれを別のある抽象的なモジュールの一つのインスタンスとして定義する方法である。モジュールとそれらの間の階層関係は、記述全体を実際のハードウェアとして実現する際に、実際の構成要素へ自然な形でマッピングされる。

2) プロセス・ベーズドな動作記述

クロックを基準にした動作記述のかわりに、OODEではメッセージ・パッシングを動作記述の基礎として用いている。それにより、設計者は対象の細かい部分にとらわれずにシミュレート・評価ができる。

3) サポート・システム

設計者の知識、経験を蓄積し、必要なときに容易にとり出せるようにモジュール・データベースが提供されている。

次章では、OODEの一般的概念について述べ、3章ではシンタックスについて述べる。4章では、ディスクリプターについて、5章では記述から実際のハードウェアへのマッピングについて述べ、最後に、サポート・システムのレイアウトについて、6章で述べる。

2. OODEの一般概念

OODEはオブジェクト・オリエンティド言語に基づいている。オブジェクトオリエンティド言語としてはSIMULA[5)]やSMALLTALK[6)]等のプログラミング言語があり、また、Hewittら[7)]やWard[8)]らはこの言語の基本的性質であるメッ

セージ・パッシング機構について理論的に調べている。本章では OODE の基本概念について説明する。

データ型としてはオブジェクトだけが提供されている。オブジェクトは、記述の基礎として用いられ、実際のハードウェア上のチップやボードに対応付けることもできる。オブジェクトは、その内部状態に対応するデータ構造と、それを操作する手続きをもつことができる。オブジェクトの内部状態を参照したり、修正したりする手段はこの手続きしかなく、従ってデータ・アブストラクションが達成されている。オブジェクトに何かをさせるには、その行為の名前を含むメッセージをオブジェクトに送信することによってなされる。オブジェクトはメッセージを受けとると、それを解読し、それに示されている行為を実行する；この行為の間にオブジェクトが別の複数のオブジェクトにメッセージを送信することもある。

OODE は一般的な時間軸といったものを提供しない。そのため、OODE における計算は純粋に event-driven な形で行われる。あるオブジェクトをある信号で起動するということはメッセージを送るという形で表現され、次のように書かれる。

(<オブジェクト名> <メッセージ>)

例えば

(register set 0)

では、"register" がオブジェクト名、"set 0" がメッセージである。このメッセージの送信のことをイベントと呼ぶことにする(より厳密な定義は3章で行う)。イベントは、オブジェクト間の通信(行うべき行為名とデータがメッセージの中に書かれている)を表わしていると考えることができる。

OODE は、分散化したプロセスを時間に沿ってではなく、上のイベントの半順序関係付の集合として記述することができる。メッセージの発信は直列にも並列にも行われるので、OODE は複雑なシステムのからみ合った動作を、複数の基本的には独立並列に動作するオブジェクトがメッセージの送受信により同期をとったり、データの交換をしたりして協調的に行っている行為の全体として記述する。このような記述の仕方は、計算機ハードウェアのようなモジュール性が強く、処理の分散化が進んでいるようなシステムの記述に向いており、また、記述からそれに対応する実際のハードウェアを作ることも容易である。

上に述べたイベントの形式は、オブジェクト名、メッセージをそれぞれ、関数あるいはプロセージャ名、引数と考えれば、functional application や procedure invocation の形に類似しているが、コントロール・リターンがないという点でイベントは他の2つと大きく異なった意味をもつ。一般にイベントは、あるオブジェクトに何らかの命令とともに起動をかけるという意味しかなく、メッセージを送った側は、それによって何の影響(例えば、起動をかけられたオブジェクトの行為が終るまで待っているといったようなこと)も受けない。

3. シンタックス

本章では、OODE のシンタックスについて考える。これらは Ward らの理論[87]に基づいている。

3.1 オブジェクト

定義　オブジェクトは次のいずれかである。

i　定数
ii　基本オペレータ
iii　変数
iv　モジュール（構造をもったオブジェクト）

モジュールは "indefinite", "definite" という二つのカテゴリーに分類される（この二つのカテゴリーは、SIMULA や SMALLTALK の "class", "instance" の概念にそれぞれ等しい）。definite モジュールは、それが定義された環境の中でユニークな名前をもち、同時に2つ以上のオブジェクトからアクセスされることはできない。これに対し、定数や基本オペレータは常に任意の数のオブジェクトから同時アクセス可能である。Indefinite モジュールは、define モジュールや indefinite モジュールを新たに定義する際にその原型として用いられる。

モジュールの定義は、

(module <name> <category>
　(<descriptor> ··· <descriptor>))

である。ただし、<category> は indefinite か definite かのどちらかである。<descriptor> は モジュールをいくつかの観点から記述するのに使われ、現在、6つの descriptor が（states、script、is、submodules、shared、property）が用意されている。しかし、すべてを使う必要はない。本章では、この中の2つ states と script descriptor について説明し、残りは次章で説明する。

states　descriptor はモジュールの内部状態を宣言する。

(states (var-1 ··· var-n) (val-1 ··· val-n)).

上の形では、var-1 から var-n までの n 個の内部状態を表わす n 個の変数が宣言され、それぞれに val-1 から val-n までの値が初期値として与えられる。

script　descriptor は、メッセージの解読並びにそれに対応したアクションをイベントの形で記述する。

(script <eventset> ··· <eventset>)
　<eventset> ::= (<key> <event> ··· <event>).

モジュールはメッセージを受けとると、その第一番目の要素をとり出し、それとマッチする <key> をもつ <eventset> に含まれる <event> の集合を同時に起動する。

3.2　イベント

イベントの完全な形を下に示す。

(<object> <object> ··· <object> {<continuation>})
　<continuation> ::= (next <event> ··· <event>)。

（本報告の中では、"X があってもなくても良い" ことを示すために {X} という記法を用いる。）イベントの中の第一番目のオブジェクトをレシーバ、残りをメッセージと呼ぶ。continuation はこのイベントで起動されるアクションが終了した後で行われるべきアクションを記述する。

あるイベントを起動すると、そのレシーバオブジェクトは、自身の script に従いまた別のイベント（複数個もあり得る）を起動することがある。このイベント起動の因果関係により、プロセスの概念を次のように定義する。イベント E によ

り起動されたプロセスとは、イベントEで始まり起動の因果関係で結合されたイベントのチェインのことである。また、プロセスの終了は、プロセスに含まれるすべてのイベントが起動され終えたときのことであるとする。以上のような定義した言葉を使うとcontinuationについて次のように言うことができる。イベントEに含まれるcontinuation中のイベントが起動されるのは、イベントEで起動されたプロセスが終了したときである。

一般に起動をかけられたオブジェクトは、そのscriptに含まれる複数のイベントを並列に起動できるので、プロセスのforkは容易に記述できる。また逐次動作もcontinuationにより容易に書ける。これらに加えて、プロセスの同期を実現するためにFork-Join形が提供されている。その形式は、

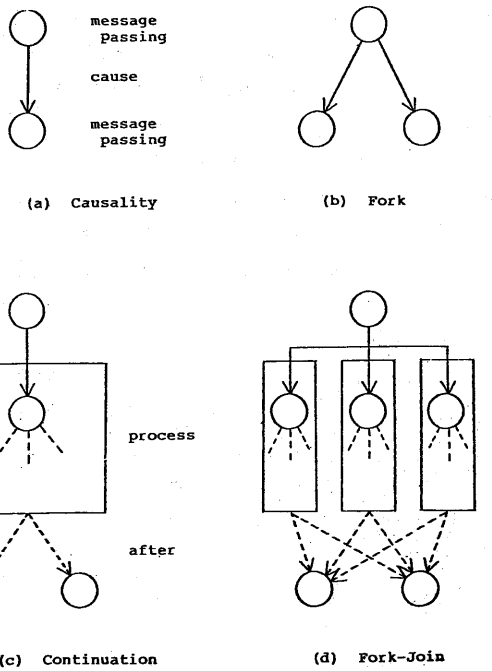
$(f_j \ \langle event_1 \rangle \ \cdots \ \langle event_n \rangle \ \langle continuation \rangle)$

である。上の形式が評価されると、まず$\langle event_1 \rangle$から$\langle event_n \rangle$までが同時に起動され(Fork)、起動された各プロセスがすべて終了したときに、$\langle continuation \rangle$に含まれるイベントが起動される。この$f_j$オペレータにより複数プロセスの間の同期をとり、次のプロセスを開始することができる。

以上のイベント・シンタックスによりオブジェクトの動作をevent-drivenな形で記述することができる。図1にイベント間の関係を示す。図において、円はメッセージ・パッシング、矢印は因果関係、方形はプロセスを表わしている。また方形より出ている点線はプロセスとイベントの間の前後関係を表わしている。

次に掲げるものはイベントの例である。

(s set 5)、(+ 3 5)、(counter up (next (register clear)
(50 set 10))).



(a) Causality (b) Fork

(c) Continuation (d) Fork-Join

Figure 1. Relation among message passings

3.3 値を返すイベント

いくつかのオブジェクトは適当なメッセージを受けとったときに値を返す。例えば定数や変数は空メッセージを受けとったとき、それ自身の値を返す。

(3) ⟶ 3　　(S) ⟶ Sの値.

また、基本オペレータも演算結果を値として返す。モジュールもそのことを明示することにより値を返すことができる。

メッセージ・パッシング・メカニズムの中にはコントロール・リターンがなくまたメッセージは評価されないので、今のままでは、返された値がどこに行くか書けない。これを解決するために、2つの特別なオブジェクト、Throw、Catchを導入する。Throwは返されてきた値をつかまえ、それをCatchに渡す。Catchは、その値を別のイベントの中のメッセージ中に埋め込む。これらの形式は、

(throw <event> as <name> {<continuation>})
(catch <list> in <event>).

ただし、throwに含まれるイベントは値を返すイベントでなくてはならない。また<name>は値につける名前であり、<list>は、それら名前のリストである。Throwは値に名前を付けて投げる。Catchはリストに登録されている名前であればそれを受けとり、その値をイベントの中に埋め込む。例:

```
(throw (+ 2 3) as x
        (next (catch (x) in (register set x))))
(fj (throw (-9 2) as x)
    (throw (+2 1) as y)
       (next (catch (x y) in (throw (+ x y) as x
                                    (next (catch (x) in (s set x))))))).
```

このような扱いは、前述のようにメッセージ中に、あるイベントの返す値を置きたいときに生じる。実用上、毎回上のように書くのは大変なので、上の形を基本として、構文を拡張し、メッセージ中にイベント(値を返すもの)を置けるようにする。ただし、それらは下のルールに従って、処理系が基本形に展開する。

```
(obj x1 x2 --- xn <continuation>)
  ⟹  (fj (throw (y1) as z1)
          (throw (y2) as z2)
                  ⋮
          (throw (yn) as zn)
              (next (catch (z1 z2 ··· zn)
                           in (obj z1 z2 --- zn <continuation>)))).
```

ただし、

(yi) = (xi)　　if　xi = object.
(yi) = xi　　if　xi = event

以上の拡張により、上の2例は、次のように書ける。

```
(register set (+ 2 3))
(s set (+ (-9 2) (+ 2 1))).
```

3.4 モジュールにおけるメッセージ処理

モジュールが受けとったメッセージは、一旦、FIFO キューに入れられる。Script descriptor は、必要なときにそのキューより一語ずつ読み出して処理を進める。キューへアクセスするイベントとして次の2つがある。(:), (::). (:)はキューの最初のオブジェクトを値として返し、同じにキューからそれを除く。(::)もやはりキューの最初のオブジェクトを値として返すが、それを除くことはしない。例えば、今、受けとったメッセージが "shift left with 2" であったとする。メッセージの最初のオブジェクトは、イベントセットを選ぶために script により自動的にとられるので、最初の(:)は "left" を返す。このとき、キューは "with 2" に変化する。ここで、(::)が起動されると "with" を返すが、キューは前のままである。

いくつかの制御構造が、プログラミングを容易にするために導入されている。それらのシンタックスと意味を下に掲げる。

If: (if <event> then <event> ··· <event> {<continuation>})

最初のイベントの返す値が true のとき、残りのイベントと continuation を Fork-Join 形で並列に起動する。

Select: (select <event>
(<object> <event> ··· <event>)
⋮
(<object> <event> ··· <event>
{<continuation>})

最初のイベントの返す値と等しい <object> をもつ組のイベントと continuation を Fork-Join 形で起動.

Do: (do <number> <event> ··· <event> {<continuation>})

指定された回数だけ、イベントを並列に起動し、すべて終了の後、continuation.

While: (while <event> do <event> ··· <event> {<continuation>})

最初のイベントが true を返す間、残りのイベントを繰り返し起動する。

Return: (return <event>)

値をモジュールの中から返すときに用いる。

図2に、モジュール定義の例を示す。

```
(module  COUNTER  definite
   ((states (s) (0))
    (script (clear  (s set 0))
            (up     (s set (+ s 1)))
            (down   (s set (- s 1)))
            (out    (return s)))))
```

(a) Counter

```
(module STAGE1 definite
        ((states (s1) (0))
         (script (fire ( ..........
                         (next (STAGE2 fire data))))
                 (sleep ( ......... )))))

(module STAGE2 definite
        ((states (s2) (0))
         (script (fire ( ..........
                         (next (STAGE3 fire data))))
                 (sleep ( ......... )))))

                 .
                 .
                 .

(module STAGEn definite
        ((states (sn) (0))
         (script (fire ( ..........
                         (next (OUT fire data))))
                 (sleep ( ......... )))))
```

(b) Pipeline

Figure 2. Sample description

図3に、2次元パイプラインの定義の例を示す。この例ではクロックを用いている。

```
(module P[i] definite
   ((states (input1 input2) (0 0))
    (script
     (fire      (throw (and input1 input2) as x
                    (next (catch (x) in
                                 (P[i+1] setleft x)
                                 (P[i-1] setright x)
                                 (Q[i] up x)))))
     (setleft   (input1 set (:)))
     (setright  (input2 set (:))))))

(module Q[i] definite
   ((states (input1 input2) (0 0))
    (script
     (fire      (throw (and input1 input2) as x
                    (next (catch (x) in
                                 (Q[i+1] setleft x)
                                 (R[i] up x)))))
     (setleft   (input1 set (:)))
     (up        (input2 set (:))))))

(module R[i] definite
   ((states (input1 input2) (0 0))
    (script
     (fire      (throw (and input1 input2) as x
                    (next (catch (x) in
                                 (R[i-1] setright x)))))
     (setright  (input1 set (:)))
     (up        (input2 set (:))))))
```

```
(module CLOCK definite
   ((script
     (start     (fj (P[1] fire) ... (P[n] fire)
                    (Q[1] fire) ... (Q[n] fire)
                    (R[1] fire) ... (R[n] fire)
                    (next (CLOCK start)))))))
```

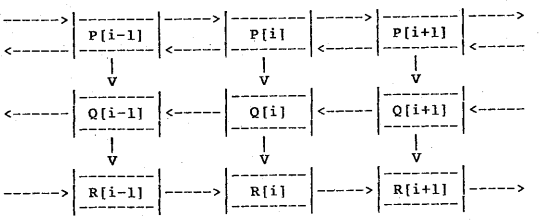


Figure 3. Two dimensional pipeline

4. ディスクリプター

モジュールをいくつかの観点から記述するために、6種類のディスクリプター（states、script、property、is、submodules、shared）が提供されている。

4.1 Property ディスクリプター

Property ディスクリプターはモジュールの設計パラメータをそのデフォール値とともに記述する。形式：

(property <prop-elem> ··· <prop-elem>)、

<prop-elem> ::= (<propname> <propval>) or <switch>.

ただし、<propname> は、パラメータに与える名前、<propval> はそのデフォールト値である。<switch> は、通常、値として true が与えられ、script中で論理変数として扱われ、必要な動作を選択するのに使われる。このディスクリプターの主な目的は、モジュールの設計をパラメータ化しておくことにより、様々なパラメータ値をもつモジュールを、1つモジュールから容易に生成できるようにすることである。例：

(property (size 1024) (pagesize 512) (wordsize 36))

(property autoindex indirect).

図4に、switch を用いたモジュール定義の例を示す。Property ディスクリプターは次に掲げる Is descripton と密接な関係がある。

```
(module ADDRESSING indefinite
   ((property autoindex indirect)
    (script
        (create
           (MAR set (MB out)
           (next (if (and autoindex
                          (and (lessp (MAR out) 1777)
                               (greaterp (MAR out) 1000)))
                     then (MB set (MEM out)
                          (next (MB increment
                          (next (MEM set (MB out)
                          (next (MAR set (MB out))))))))
                          (next (if indirect
                                 then (MB set (MEM out)
                                  (next (MAR set (MB out)))
                                  ))))))))))

  MEM = memory,
  MB = memory buffer,
  MAR = memory address register
```

Figure 4. Sample description with switches

4.2 Is ディスクリプター

Is ディスクリプターは、プロタイプ・モジュールを参照するときに用いられる。形式：

(Is (a <modulename> <isa-elem> ... <isa-elem>)).

<isa-elem> ::= <with-elem> or <add-elem>

<with-elem> ::= (with <elem> ... <elem>)

<elem> ::= (<propname> <propval>) or <switch>

<add-elem> ::= (add <key> <event> ... <event>)

あるいは

(Is (like <modulename> <islike-elem> ... <islike-elem>))

<islike-elem> ::= (with <switch> ... <switch>) or <add-elem>.

ただし、<modulename> は参照するモジュールの名前であり、弟一形の場合は indefinite 弟二形では definite でなくてはならない。弟一形の意味は参照しているモジュールは、参照されている indefinite モジュールの1つの instance (ただし、新たにパラメータの値を設定しなおしたもの) であるということであり、弟二形の場合は、参照しているモジュールは参照されているモジュールの内部状態を含めたコピーであるということである。両方に場合に共通して、<add-elem> は、新たなメッセージ処理方法を追加するために使われる。また、<with-elem> は、参照されているモジュールの Property ディスクリプターと対応していなければならない。例:

(Is (a memory (with (size 2048) (pagesize 1024))))

(Is (like counter (add set (s set (:)))))

Is ディスクリプターを用いたモジュール定義例:

(module MEMO definite
   ((Is (a memory (with (size 2048))))).

4.3 Submodules ディスクリプター

このディスクリプターは、モジュールを定義する際、その中でのみ使われるローカルなモジュールを宣言するのに使われる。形式：

(Submodules (<modulename> <is descriptor>) ...
            (<modulename> <is descriptor>)).

宣言されたモジュールはすべて definite として扱われ、それらへのアクセスは、宣言しているモジュール内部に限られる。個々の submodules を記述するのは前述の Is ディスクリプターである。例:

```
(submodules  (mem  (is (a memory (with (size 2048) (wordsize 16))))
             (CO   (is (like counter ))))
```

4.4 Shared ディスクリプター

Submodules ディスクリプターは、モジュールの構造を階層的に記述するのに用いられるが、実際上は、1つのサブ・モジュールが、いく通りかの使われ方をしていて、完全に階層的には記述できないことが多い。Shared ディスクリプターは、このような、共有モジュールを宣言するのに用いられる。形式:

```
(shared ((<name> <subname>) ... (<name> <subname>))
                    ⋮
        ((<name> <subname>) ... (<name> <subname>))).
```

ただし、<name> は submodules ディスクリプターで宣言されたモジュールの名前であり、<subname> は、そのモジュールの中でさらにローカルなモジュールの名前でなければならない。shared ディスクリプターで宣言されたサブ・サブモジュールは全く同一のものとして扱われる。例:

```
(module MOD0 definite
      (submodules (M1 (is (a MOD1 (with ... ))))
                  (M2 (is (a MOD2 (with ... )))))
      (shared ((M1 MOD11) (M2 MOD21)))
                  ....                          ).
```

ただし、

```
(module MOD1 indefinite
      (submodules (MOD11 ... ) (MOD12 ... ))
                  ...                          ),
(module MOD2 indefinite
      (submodules (MOD21 ... ) (MOD22 ... ))
                  ...                          ).
```

5. 記述からハードウェアへのマッピングについて.

OODE の記述は抽象的なレベルで行われるので、実際のハードウェアを得るためには、記述をハードウェアにマッピングしなければならない。このマッピングにおいて、definite モジュールは、実際の回路あるいはマイクロ・プログラムへとマップされる。実際の回路へと写される場合には、states、submodules, shared Is 等のディスクリプターにより記述された構造はそのままの形でマップされ、それらの動作を記述した script は、制御回路へとマップされる。script 中に現われる Do や Select 等の制御オペレータは、何らかのあらかじめ定められた制御構造へとマップされる。例えば、Do はカウンターを備えた構造へ、Select はマルチプレクサーを備えた構造へとマップされる。現在、このようなマッピング・プログラムを開発中である。

6. サポート・システム

設計者を全体的に支援する目的で次のような2つのオブジェクト（DATABASE MANAGER および OPERATOR）が提供されている。DATABASE MANAGER は、モジュールが新たに定義されるたびに、そのモジュールの名前と構造を受けとり蓄える。構造に関する情報は、is、submodules、property 等のディスクリプターに基づいており、設計者は、定まった構造をもったものを検索したり、あるモジュールの定義を変更した結果生じるであろう影響をあらかじめ知ることができる。

OPERATOR は シミュレーションをモニターし、特定の状態が発生したら、何らかの行動を起したりする。例えば、いくつかのモジュールのアクセス回数を計算したり、特異な状態が発生したら警告を発したりである。

基本的にこれらのオブジェクトは、ユーザーが定義できるモジュールと同じ様にメッセージを受けとったり、発信したりできるので、ユーザーは、自分の目的により適合したサポートシステムを作り上げることができる。

7. 謝辞

日頃、御指導頂く当研究所システム第2グループ長岡GRと御討論をして頂いたグループ員の方々に深謝の意を表します。

8. 参考文献

1) Sussman, G. J., Holloway, J., Knight, Jr, T. F., Computer Aided Evolutionary Design for Digital Integrated Systems, MIT AI memo No. 526 (1979)
2) Piloty, R., Barbacci, M., Borrione, D., Dietmeyer, D., Hill, F., Skelly, P., CONLAN - A Formal Construction Method for Hardware Description Language: basic principles, Proc. NCC 80 (1980)
3) 同上, CONLAN - A Formal Construction Method for Hardware Description Language: Language derivation, Proc. NCC 80 (1980)
4) 同上, CONLAN - A Formal Construction Method for Hardware Description Language: Language application, Proc. NCC 80 (1980)
5) Ichbiah, J. D., Morse, S. P., General Concepts of the Simula 67 Programming Language, Annual Review in Automatic Programming Vol. 7, Part 1 (1972)
6) Ingalls, D., The Smalltalk 76 programming system design and implementation, Proc. ACM symposium on Principles of Prog. Lang. (1978)
7) Hewitt, C., Smith, B. C., Towards a Programming Apprentice, IEEE Trans. on Soft. Eng. Vol. 1, No. 1 (1975)
8) Ward, S. A., Halstead, R. H., A Syntactic Theory of Message Passing, JACM Vol. 27, No. 2 (1980)
9) Takeuchi, A., Object Oriented Description Environment for Computer Hardware, Proc. CHDL (1981).