ＩＤＬ：ネットワークソフトウェアの設計記述言語

高橋　薫　　白鳥　則郎　　野口　正一
（東北大学　電気通信研究所）

１．はじめに

近年、計算機ネットワークに代表される分散処理システムの発展、普及と共に、通信ソフトウェアの開発も増加の一途をたどっている。一般にこれらのソフトウェアは予め取り決められた通信プロトコルを実現するように意図され、同じ機能を持ったソフトウェアが複数の計算機システム上に実現される。この場合、各システムのソフトウェア設計者、プログラマーは各々に独自な手段でソフトウェアを設計し、インプリメントしてきているのが現状であり、ソフトウェアの生産性の面からも、保守性の面からも非効率である。これに対する一つの解は、実現システムに依存しない携帯性又は移植性に秀れた高水準プログラミング言語（例えば、CやＡｄａなど）を用いてソフトウェアを書くことである。

しかしながら、これらの言語は汎用性がある反面、適用するソフトウェアの規模や複雑さが増大するにつれ、読解性、保守性の点で問題がある。

そのため、むしろ通信ソフトウェアの特徴点に着目し、ソフトウェア・ライフサイクル上の初期段階（設計段階）で適用でき、そのまま実行できる言語を開発することが望ましい。このような言語を採用することによって、(1) ソフトウェア・ライフサイクルの短縮（開発の容易さ）、(2) 携帯性（移植性）、(3) 理解の容易性（保守の容易性）、(4) 自動プログラミングへの可能性などが図れることになる。

通信プロトコルは一般に、事象を入力し、その時の状態に依存して事象を出力するようなものとして記述されることが多く、そのため、有限状態機械を用いてプロトコルを仕様化し、記述することが広く受入れられている[1]。

このことから、通信ソフトウェアを有限状態機械でモデル化し、設計記述しようとするのは自然であり、本論文はこの観点に立脚している。

このような考え方に基づいて、特に交換ソフトウェアの設計用言語としてＳＬ１[2] が提案されている。しかしながら、交換システムにおいては、扱われる事象が単純な信号（例えば、電話のオフフック、オンフックなど）のみに限定されており、このことはネットワークなどで扱われる構造を持った事象（例えば、ヘッダー部とデータ部からなるパケットなど）とは異なっている。

本論文では、このような構造を持った事象を扱えるだけでなく、有限状態機械の内部で使用されるデータ構造の宣言、操作を可能にし、また、事象の同時入力機構、有限状態機械の階層化を可能とする言語ＩＤＬ(Implementation and Design Language)を提案する。これらは、ＳＬ１にはない特徴であり、有限状態機械によるソフトウェア設計記述の強力なツールとして機能する。

関連する記述言語として、PC／N[12]が提案されており、これは6入力6出力の有限状態機械モデルに基づいて、状態推移関数を入出力に対応して網羅的に与えることによってプロトコル仕様を記述し、コンパイラによってプログラムの自動生成を図っている。また、ＩＢＭのＳＮＡ(Systems Network Architecture)ノード全体の実現仕様を記述する目的でＦＡＰＬ[3] が開発されており、この言語はＰＬ／Ｉをベースとして、補足的に状態推移表による有限状態機械の記述機構を備えている。

本論文の構成は次のとうりである。最初にＩＤＬの基本概念を述べ、その概念に基づいたＩＤＬの記述形式（言語の構文と意味）を与える。次にＩＤＬで記述された通信ソフトウェアを実行可能にするためのコンパイラについて述べ、最後に実際上の通信プロトコル（HDLC[4][5][6] ）への適用について述べる。

２．ＩＤＬの基本概念

2.1. 有限状態機械による通信ソフトウェアのモデル化

ＩＤＬは特に、計算機ネットワークなどで用いられる通信ソフトウェアの設計記述を対象としている。

一般に、ネットワークアーキテクチャーは階層化の概念を採用[7][8]しており、ソフトウェア設計を考えるときには、各レイヤは互いに独立であるとするのが一般的である。この時、ネットワークの特徴によって、ソフトウェアは通常、ある状態において環境からの事象に従って動作を実行し、一連の結果を環境へ出力する。よって、ソフトウェアは有限状態機械（以下、単にＦＳＭという）としてモデル化することができ、ここでは通常の有限オートマトンを用いてモデル化する。

つまり、各レイヤのＦＳＭを次のように定義する。

$FSM=<K, \Sigma, \Delta, \delta, \omega, q0>$

K：状態の有限集合
Σ：入力の有限集合
Δ：出力の有限集合
δ：状態推移関数　$(K\times\Sigma\rightarrow K)$
ω：出力関数　$(K\times\Sigma\rightarrow\Delta)$
q0：初期状態

2.2. ＦＳＭの拡張

一般に、ＦＳＭに対する入力／出力はｎ入力／ｍ出力と考えられる。よって、各入力をそれぞれ $I_1, I_2, \cdots, I_n$ とし、各出力をそれぞれ $0_1, 0_2, \cdots, 0_m$ とすると、$\Sigma \subseteq I_1 \times\cdots\times I_n$、$\Delta \subseteq 0_1 \times\cdots\times 0_m$ と表わすことができる（図１）。

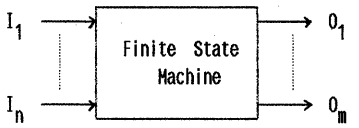


図１．　多入出力ＦＳＭ

また、単純なプロトコルを実現するＦＳＭは別として、通常環境からの入力に対して取られる状態推移及び出力は、入力とＦＳＭのその時点での状態に依存するだけでなく、ＦＳＭ内部状態（ＦＳＭの入力待ち状態とは異なる）にも依存する。

更に、推移の過程でそれらの内部状態を更新することが必要となる。従って、そのようなＦＳＭ内部状態を表わす内部状態ベクトルの概念を導入し、$v=(s_1, s_2, \cdots, s_r)$ で表わすことにする。但し、各 s_i はＦＳＭのｉ番目の内部状態要素の状態（値）とする。

このようにＦＳＭを拡張して考えることにより、window制御、buffer状態によるflow制御などが容易にモデル化できることになる。

以上の考察から、ＦＳＭの形式的定義を次のように拡張する。

$FSM=<K, \Omega, \Sigma, \Delta, \delta, \theta, \omega, q0>$

K	（入力待ち状態の有限集合）
$\Omega \subseteq S_1 \times \cdots \times S_r$	（S_i ：内部状態要素ｉのdomain）
$\Sigma \subseteq I_1 \times \cdots \times I_n$	（I_i ：入力ｉの有限集合）
$\Delta \subseteq O_1 \times \cdots \times O_m$	（O_i ：出力ｉの有限集合）
$\delta : K \times \Sigma \times \Omega \rightarrow K$	（状態推移関数）
$\theta : K \times \Sigma \times \Omega \rightarrow \Omega$	（内部状態更新関数）
$\omega : K \times \Sigma \times \Omega \rightarrow \Delta$	（出力関数）
q0	（初期状態）

2.3. ＦＳＭの分解と階層化

ネットワークアーキテクチャ上の各階層のソフトウェア設計は、その階層に対して適用されるプロトコルが複雑になるのに比例して困難さが増大する。 このことは、ＦＳＭ設計の立場からは、"state explosion problem"[1]を引きおこす。 従ってＦＳＭの分解（モジュール化）が解決策の１つとして考えられる。 分解の例を図２に示す。

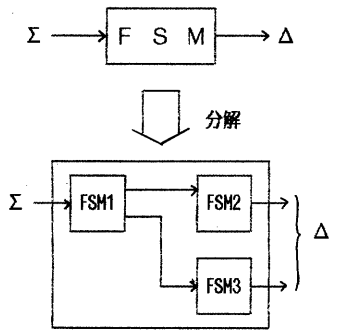


図２． ＦＳＭの分解（例）

また、分解された各ＦＳＭに対して、階層的な設計を可能とするために、sub-FSM の概念を導入する。

この概念は、ＦＳＭのある状態を抽象化してとらえようとするものであり、抽象的な状態を設定し、その状態の詳細はsub-FSMとして展開することによって、設計の容易性、理解の容易性を図ることができる。 図３に階層化の例を示す。 この例では、階層０の状態S_i が下位のsub-FSM （階層１）によって具体化されている。

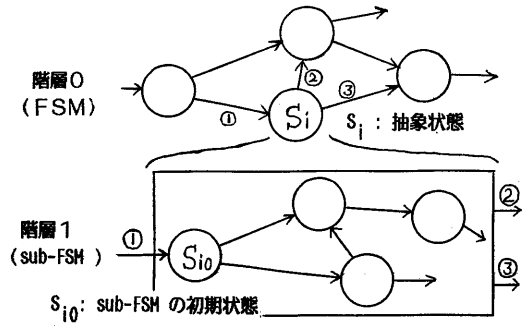


図３． sub-FSM の概念（例）

2.4. ＦＳＭインターフェースのモデル化（実行環境のモデル化）

ＦＳＭに対する入力事象は、ネットワークアーキテクチャ上の最下層レベルでは、ハードウェアから与えられ、他の階層では、他のＦＳＭの出力事象として与えられる。 特別な場合として、時間に関する事象（タイムアウト等）はＯＳ経由で与えられることがある。 また、出力事象は他のＦＳＭに対する入力事象となるか、ＯＳに対する制御命令（例えば、タイマー起動）あるいは、ハードウェア・アクセス命令と考えられる。

ＩＤＬではＦＳＭに対するタイマーを除いた入出力事象は全てキュー（FIFO）を介することとし、キューに対する事象の入出力は特別なシステム・インターフェースが介在することを仮定する（例えば、プロセス間通信サポート機能）。 このことによって、キューに対する操作上の問題や排他制御問題をＦＳＭ設計から独立させることができる。

またスケジューラの存在を仮定し、各ＦＳＭの実行スケジューリングや事象入力のタイミングは、このスケジューラによって支配管理されるものとする。

タイマーの起動や取消しについては、キューを介さず特別なシステム・インターフェースによって制御されるものとする。 タイムアウトについては、システムがそれを検出後、タイマー起動を行なったＦＳＭに対して、そのことをスケジューラを介して通知するものとする。

３．ＩＤＬ記述形式

本節ではＩＤＬの記述形式、即ち、言語の構文と意味について述べ、最後に簡単なＩＤＬ記述例を示す。

3.1. 記述の枠組みと構文記法

記述の単位は１ＦＳＭとし、記述の前半でそのＦＳＭで使用するデータの宣言を行ない、後半でＦＳＭの動作記述を行なう。

記述の枠組みは図４に示すとうりであり、最初にＦＳＭ名を宣言し、次にデータの宣言、動作の記述を行なう。 もし、sub-FSM が定義されれば、その記述を続けて行なう。

```
FSM-FSM 名
  FSM-DATA-DECLARATION
      データの宣言
  END-OF-DECLARATION
  FSM-BODY
      FSM 動作の記述
  END-OF-BODY
  SUBFSM-sub-FSM名                    ┐
  FSM-BODY                            │
      sub-FSM の動作記述               ├ 0個以上複数
  END-OF-BODY                         ┘
```

図４． 記述の全体の枠組み

以下では、言語定義の全般を通して使用される構文記法を要約する。 中括弧（［］）は構文の選択があることを示し、かつ、必ず選択されねばならないことを表わす。 大括弧（｛｝）は構文の選択が省略可能であることを除いて、中括弧と同じ意味を表

わす。 構文の繰返し表現は頭部に、記号∞をつけて表わす。 文字列定数、ビット列定数はそれぞれ、C´…文字列…´、B´…2進数列…´で表わす。 整数定数は0－9の整数列で表わす。

3.2. データ型と変数宣言

ＩＤＬでは使用できるデータ型として、「予約型」、「基本データ型」、「構造型」、「イベント型」の４種類を用意している。

予約型変数は単にメモリー空間を予約する為にだけ使用される。基本データ型変数は、ビット型、文字型、整数型のいずれかであり、アレー表現も許している。 構造型変数は、データ内容を構造的に表現する為に使用し、ビット型と文字型の混在形式として表わされ、アレー表現も許される。 イベント型は基本的に構造型と同じであるが、ＦＳＭの外部事象（ＦＳＭの入出力データ）を明示的に表わすために使用する。

図５に、上述の各データ型を用いた変数の宣言例を示す。

同図から分かるように、構造型及びイベント型変数は、他のデータ型とは違い、変数名の前に、文字列'STRUCT'及び'EVENT' がプレフィックスされる。 また構造内容を表わすのにレベル番号０２がプレフィックスされている。 ここでの構造化データの表現形式はCOBOL-likeであることを注意しておく。 同様の構造化は他の言語でも可能であるが、データ構造の一見した理解の容易性の観点からは、この表現がより適切であると考えられる。

変数に対する参照は、PASCALとほぼ同様の形式を採用している。

```
FILLER  BIT       /* 1bit の空間予約 */
A       BIT(3)  VALUE B'010'  /* 3bitsのビット変数*/
B       INT       /*  整数変数 B */
C       C         /*  文字変数 C */
STRUCT  MESSAGE   /*  構造型変数 MESSAGE */
        02 HEADER1 BIT(3)
        02 HEADER2 BIT(5)
        02 TEXT    C(64)
EVENT   EVENT     /*  イベント型変数 EVENT */
        02 FILLER  BIT(14)
        02 ACK     BIT(2)   VALUE B'01'
```

図５． データ宣言例

3.3. ＦＳＭ動作記述

ＩＤＬにおけるＦＳＭ動作記述は、前節で述べたＦＳＭによるソフトウェアのモデル化を言語の構造及び意味に反映させることである。 つまり；

『ＦＳＭの言語記述中に状態の概念を導入し、各状態における事象入力（１つとは限らない）、さらにはその入力のもとでの内部状態（変数値）に依存した事象の出力系列、内部状態の更新系列、そして後続状態への推移を明示的に表現する。』

ここで、ある状態における入力事象が１つとは限らない理由は、複数事象の入力を１つのまとまった単位と見なすような設計を可能とするためのものである。 同様に、出力事象に関しても、その出力順序は特に意味がないという状況が十分に考えられ、そのために、順序を規定しないということを明示的に表わす構文を導入している。

モデル化の上での他の特徴（例えば、sub-FSM の概念やＦＳＭインターフェース）も言語構造上に反映される。

ＦＳＭ動作記述の枠組みは図６に示すとうりである。 すなわち、ＦＳＭ動作の記述単位を各状態とその状態から次の状態までの推移過程（これを、ＦＳＭ動作単位と呼ぶ）とし、その様な独立した動作単位の記述の全体によって、ＦＳＭ全体の動作記述を行なう。

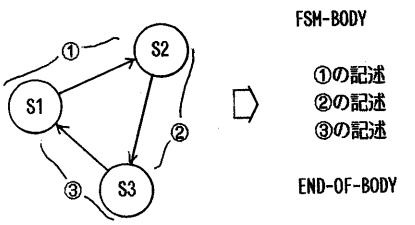


図６． ＦＳＭの動作記述の枠組み例

ＦＳＭ動作単位は次の構文を用いて記述される（sub-FSM 動作単位も同様）。

```
STATE  [ INITIAL ]
       [ 状態名  ]

  EVENT   イベント識別式   }  １個以上の
  ACTION  動作系列         }  繰返し
```

イベント識別式には次の４つの選択がある。

```
qid*erf=[ce] { ∞ AND qid*erf=[ce] }  …  type 1
        [be]                 [be]

qid*evn  { ∞ AND qid*evn }           …  type 2

TIMEOUT(タイマー名)                  …  type 3

NULL                                 …  type 4
```

ここで予約語INITIAL は初期状態を表わす。 qid はＦＳＭ間インターフェースで用いるキューの識別名であり、スケジューラに知られたものである。 erf はイベント中の必要なフィールドに対する参照形式である。 ce、beはそれぞれ文字列定数、ビット列定数である。 evn はイベント型変数名である。 タイマー名は以前に起動したタイマーの識別名である。 動作系列は後述する。

イベント識別式は次のような意味（働き）を持つ。

1)EVENT 句が上から順に１つづつ参照される（回帰的に）。

2)イベント識別式の調査。

type 1の時、3)へ。 type 2の時、4)へ。

type 3の時、5)へ。 type 4の時、6)へ。

3)qid で示されたキューの全てになんらかのイベントが存在し、かつ、キューの内容をerf 及びce、beで調べあげ、合致したならば、それらのイベントをイベント変数に代入し、後続の動作系列を実行する。 そうでなければ、1)へ。

4)qid で示されたキューの全てになんらかのイベントが存在しているならば、キューの内容をevn で示されたイベント変数に代入し後続の動作系列を実行する。 そうでなければ1)へ。

5)以前に起動したタイマーがタイムアウトになったならば、後続の動作系列を実行する。 そうでなければ、1)へ。

6)無条件に動作系列を実行する。

イベント識別式の形式において特に、type 2は設計者が予期しないイベントが入ってくるような場合、あるいは無効なイベント

が入ってくるような場合に、効果的に使用できる。

またtype 4は、無条件に実行される手続きなどを記述する時に用いることが出来る。

動作系列の記述には、通常の「代入文」の他に、表1に示すような構文を用意している。

送信	TRANSMIT(qid* イベント型変数名{(i)})
次状態指定	NEXT-STATE [状態名 / sub-FSM 名 / EXIT]
任意順序指定	ANYORDER : 送信文 : END
選択	IF B_1 : S_1 | : B_n : S_n | END
繰返し的選択	DO B_1 : S_1 | : B_n : S_n | END
タイマー起動	TIMER(timer 名, 時間)
タイマー取消し	CANCEL (timer 名1, …, timer 名n)

表1.　FSM動作系列記述で使用する構文

「送信」文は指定されたキューに指定された変数内容を送信することを意味する。「次状態指定」文は指定された状態への推移あるいは、指定された下位のsub-FSM の呼出し、あるいは上位FSMへの復帰を指定する。「任意順序指定」文は、囲まれた送信文を任意の順序で実行して良いことを示し、イベント出力が順序によらないような場合に用いられる。「選択」文は条件式 B_1 ～B_n の内、真であるものを任意に1つ取りだして、対応する文系列S_i を実行して終了する。真であるものが無い時にはスキップされる。「繰返し的選択」文は「選択」文の繰返しを表わす。但し、B_1 からB_n の内、真であるものが1つも無い時にはスキップされる。なお、「選択」及び「繰返し的選択」の構文、意味はB.Hansenの'Distributed Process'[9]から借りた。「タイマー起動」文は指定されたタイマーを指定されたタイムアウト時間と共に起動する。「タイマー取消し」文は指定されたタイマー群をキャンセルする。

3.4. 単純なプロトコルへの適用例

通信ソフトウェアがどのようにIDLを用いて記述されるかを示すため、以下のような簡単なプロトコルを実現するソフトウェアを考察する。

『メッセージの送信規約』

1)上位階層のソフトウェアからメッセージを入力し、相手ソフトウェア(local or remote) へ送信する。

2)応答を受信する。

3)肯定応答(ack) であれば、メッセージが正しく送信完了したとして'success' を上位へ通知する。無応答(3 sec) であれば、前のメッセージを再送する。再送を3回行なっても回復しない時には、メッセージ転送が失敗したとして'failure' を上位へ通知する。

4) 1)-3)を上位のメッセージが発生する毎に繰返す。

ここで以下の事を仮定する。

＊上位階層のソフトウェアからのメッセージが格納されるキュー　：　q1

＊上位階層のソフトウェアへの応答メッセージを格納するキュー　：　q2

＊相手ソフトウェアへのメッセージを格納するキュー：　q3

＊相手ソフトウェアからの応答メッセージが格納されるキュー　：　q4

＊上位ソフトウェア及び相手ソフトウェアへのメッセージ形式　：　128バイト固定のビット列

＊相手ソフトウェアからの応答メッセージの形式　：

ack　-　1 byte固定　B'xxxxxx00'

＊上位ソフトウェアへの応答メッセージの形式　：

success　-　1 byte固定　B'xxxxxx00'

failure　-　1 byte固定　B'xxxxxx10'

対象とするソフトウェア（FSM）を設計記述するには、まず上記のプロトコルを分析し、対応するソフトウェアの設計を状態推移図で書き表わす。そしてその状態推移図に対応して、IDLを用いて記述すれば良い。図7、図8にそれぞれ、設計したソフトウェアの状態推移図、対応するIDL記述を示す。

この記述例から明らかなように、IDL記述と状態推移図設計がほぼ1対1に対応しており、FSM設計段階における記述ツールとしても、保守用としてのドキュメンテーション・ツールとしても有効に機能していることが分かる。

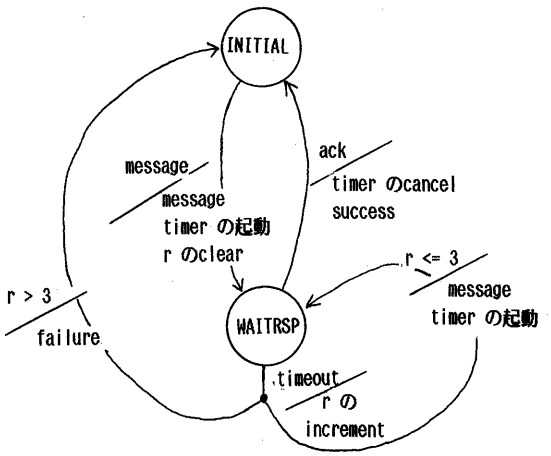


図7.　状態推移図

```
FSM-TRANSMITTER
FSM-DATA-DECLARATION
  EVENT  MESSAGE
         02  TEXT     C(128)
  EVENT  RESPONSE                          ┐
         02  FILLER   BIT(6)               │
         02  ACK      BIT(2)               │ データの宣言
  EVENT  REPLY                             │
         02  FILLER   BIT(6)               │
         02  SF       BIT(2)               ┘
  R      INT          VALUE   0
```

```
END-OF-DECLARATION
FSM-BODY
  STATE   INITIAL
    EVENT   Q1*MESSAGE
    ACTION  TRANSMIT(Q3*MESSAGE)
            TIMER(TIMER,3000)
            R:=0
            NEXT-STATE  WAITRSP
  STATE   WAITRSP
    EVENT   Q4*RESPONSE.ACK=B'00'
    ACTION  CANCEL(TIMER)
            REPLY.SF:=B'00'
            TRANSMIT(Q2*REPLY)
            NEXT-STATE  INITIAL
    EVENT   TIMEOUT(TIMER)
    ACTION  R:=R+1
            IF R > 3: REPLY.SF:=B'10'
                      TRANSMIT(Q2*REPLY)
                      NEXT-STATE INITIAL   |
               R <= 3: TRANSMIT(Q3*MESSAGE)
                      TIMER(TIMER,3000)
                      NEXT-STATE WAITRSP   |
            END
END-OF-BODY
```

動作の記述

図8.　IDLによる記述

4. IDLコンパイラとオブジェクトの実装

本節では、IDLコンパイラの概要と、コンパイラによって生成されたオブジェクトの実装について述べる。

IDLのコンパイラとして、次の2つを想定している。

(1) IDL → 既存の高級言語

(2) IDL → マシン依存言語

(1) はオブジェクトに対して汎用性を持たせるためのコンパイラであり、オブジェクトの移植性を考えた場合に適切である。 (2) はオブジェクトの実行効率を考慮したコンパイラであり、筆者らが計算機ネットワーク用に使用しているFACOM U-200 のアセンブリ言語への変換を行なう。

本論文では、(2) のコンパイラの特徴について述べる。 (1) のコンパイラ（検討中）の概要については、文献[10]を参照されたい。

前節で述べたように、IDLではFSMの動作記述が各状態毎に独立に記述される。 また状態における各入力事象は、それぞれ独立にEVENT 句によって記述され、それに伴った動作系列がACTION句によって記述される。 そこで、IDLコンパイラは、状態間推移の制御、及び現在状態における入力事象の判定とそれによる動作系列の実行をオブジェクトの制御構造として生成する必要がある。

状態間推移の制御については、まず最初に初期状態にオブジェクト・エントリを設定し、以降はソースプログラム中に現れるNEXT-STATE文に応じて、対応する目的先状態へのgoto命令を生成している。

入力事象の判定と動作系列の実行については、図9に示すように、イベント識別式が成立するかどうかの判定系列と成立したイベント識別式に対応した動作系列を実行させるような構造を生成する。

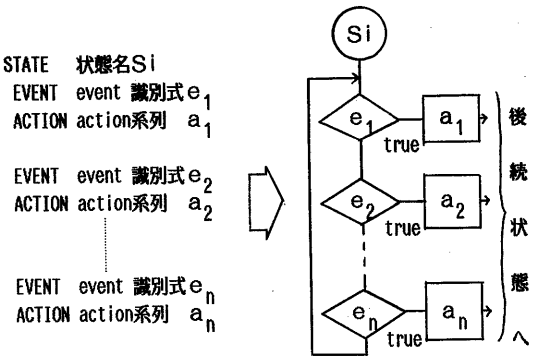


図9.　入力事象の判定とaction系列の実行に対するオブジェクト構造

IDLコンパイラが果すべき、もう1つの重要なことは、FSM環境とのインターフェースである。 つまり、キューやタイマーとのアクセスをオブジェクト中に反映させる必要がある。 この方策としては、オブジェクトの実行時ライブラリとして、それらとのアクセスプログラムを用意し、IDLコンパイラではオブジェクト中にライブラリへの呼出し命令を生成する。 ライブラリでは、キューやタイマーとのアクセスを既存のOSのサービス機能を利用して行なう。 要するにライブラリは、FSMとシステムとのインターフェース機能として働く。

一般的には、FSMがどのようなOSの基で実行されるかによって、システムインターフェース等が大きく異なってくるため、ライブラリの内容も異なってくると考えられる。

筆者らが開発したIDLコンパイラのオブジェクトは、既存のリアルタイムOSの配下で、並行プロセスの1つとして実装される。 そしてそのOSのスケジューリングの基で、キューやタイマーを介しながら、他の並行プロセスと非同期的に実行を行なっている。 キューやキューにエントリーされる内容は、全てメモリの常駐部に配置し、ライブラリはこれらへの排他的アクセスを保証している。

なお、開発したIDLコンパイラはアセンブリー言語を使用し、約 12000ステップ（約70Kbytes）程度の容量となっている（既存のシステムサブルーチン等を除く）。

5. HDLCへの適用

IDLを用いて、HDLCを実現するソフトウェアを記述する。

これはネットワークアーキテクチャでいうと、データリンク階層のソフトウェアの記述に対応し、ネットワークソフトウェアの典型例と考えられる。

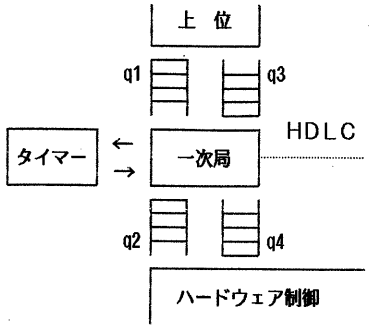
ここでは、特にHDLCの“基本手順クラス － 不平衡型正規応答モード”のプロトコルを実現するソフトウェアを記述する。詳細については、文献[4][5][6] を参照されたい。 記述としては、一次局を対象とする。

ここで、図10に示すようなFSM環境を仮定する。 また、フレームの送受信は下位プロセス（ハードウェア制御）によって実行されるとし、その際FCSエラーのあるフレームは廃棄するものとする。

図11にHDLC一次局のIDL記述を示す。

実際に、ここで記述されたIDLソースプログラムからコンパイラを用いてオブジェクトを生成し、実際のシステム（FACOM U-200 ）上で、実証実験を行なった。 この実験では、二次局とし

てPC8001を利用し、その上に簡単なテスト用プログラムを作成し、回線上のフレームの動きがプロトコル仕様に合致していることを検証確認した。



（データ形式）

①上位から一次局

＊情報　　：　128 bytes 固定の文字列

＊データリンク設定要求：　1 byte固定　xxxxxxx1

＊データリンク切断要求：　1 byte固定　1xxxxxxx

②一次局から上位

＊情報　　：　128 bytes 固定の文字列

＊データリンク設定終了(UA)　：　1 byte固定　xxxxxx00

＊データリンク設定拒否(DM)　：　1 byte固定　xxxxxx01

＊データリンク切断確認(UA)　：　1 byte固定　xxxxxx10

＊データリンク切断確認(DM)　：　1 byte固定　xxxxxx11

③一次局とタイマー

＊タイマー値　　：　全て1秒

図10.　FSM（一次局）の環境

```
FSM-Primary
FSM-DATA-DECLARATION
  EVENT inftu       /* information to the upper */
        02 text     C(128)
  EVENT rfc         /* control from the upper */
        02 cnttl    BIT
        02 FILLER   BIT(6)
        02 cntl     BIT
  EVENT inffu       /* information from the upper */
        02 text     C(128)
  EVENT reply       /* reply to the upper */
        02 FILLER   BIT(6)
        02 cntl     BIT(2)
  EVENT IU(8)       /* information frame buffers */
        02 FILLER   BIT(8)
        02 NR       BIT(3)
        02 PF       BIT
        02 NS       BIT(3)
        02 i        BIT     VALUE B'0'
        02 text     C(128)
  EVENT IL          /* information from the destination */
        02 FILLER   BIT(8)
        02 NR       BIT(3)
        02 PF       BIT
        02 NS       BIT(3)
        02 i        BIT     VALUE B'0'
        02 text     C(128)
  EVENT RR          /* receive ready */
        02 FILLER   BIT(8)
        02 NR       BIT(3)
        02 PF       BIT
        02 i        BIT(4)  VALUE B'0001'
  EVENT RNR         /* receive not ready */
        02 FILLER   BIT(8)
        02 NR       BIT(3)
        02 PF       BIT
        02 i        BIT(4)  VALUE B'0101'
  EVENT SNRM        /* set normal response mode */
        02 FILLER   BIT(8)
        02 i1       BIT(3)  VALUE B'100'
        02 P        BIT
        02 i2       BIT(4)  VALUE B'0011'
  EVENT DISC        /* disconnect */
        02 FILLER   BIT(8)
        02 i1       BIT(3)  VALUE B'010'
        02 P        BIT
        02 i2       BIT(4)  VALUE B'0011'
  EVENT UA          /* unnumbered ack */
        02 FILLER   BIT(8)
        02 i1       BIT(3)  VALUE B'011'
        02 F        BIT
        02 i2       BIT(4)  VALUE B'0011'
  EVENT DM          /* disconnected mode */
        02 FILLER   BIT(8)
        02 i1       BIT(3)  VALUE B'000'
        02 F        BIT
        02 i2       BIT(4)  VALUE B'1111'
  EVENT garbage
        02 FILLER   BIT(11)
        02 i        BIT
        02 FILLER   BIT(1588)
  VS    INT         /* send-variable */
  VR    INT         /* receive-variable */
  nt    INT         /* the # of frames transmitted */
  busy  INT         /* busy flag */
  lowest INT        /* left edge of the window */
  w0    INT
  w1    INT
  w2    INT
  window INT  VALUE 8   /* window size */
END-OF-DECLARATION
FSM-BODY
STATE   INITIAL                     /* initial state */
  EVENT  q3*rfc.cntl  = B'1'  /* request for connection */
  ACTION SNRM.P := B'1'
         TRANSMIT(q4*SNRM)
         TIMER(timer,1000)
         NEXT-STATE WAIT1
STATE   WAIT1       /* waiting state for SNRM response */
  EVENT  q2*UA.i2 = B'0011'           /* unnumbered ack */
  ACTION CANCEL(timer)
         reply.cntl := B'00'
         TRANSMIT(q1*reply)
         VS := 0
         VR := 0
         busy := 0
         nt := 0
         lowest := 0
         NEXT-STATE ESTABLISHED
  EVENT  q2*DM.i2 = B'1111'       /* disconnected mode */
  ACTION CANCEL(timer)
         reply.cntl := B'01'
         TRANSMIT(q1*reply)
         NEXT-STATE INITIAL
  EVENT  TIMEOUT(timer)                /* timeout */
  ACTION TRANSMIT(q4*SNRM)
         TIMER(timer,1000)
         NEXT-STATE WAIT1
STATE   ESTABLISHED ||          /* established state */
  EVENT  q3*rfc.cnttl=B'1'  /* request for disconnection*/
  ACTION RNR.NR := VR
         RNR.PF := B'1'
```

```
        TRANSMIT(q4*RNR)
        TIMER(timer,1000)
        NEXT-STATE WAIT5
  EVENT  NULL
  ACTION IF busy = 0 : NEXT-STATE WAIT3          |
         busy = 1 : RR.NR := VR
                    RR.PF := B'1'
                    TRANSMIT(q4*RR)
                    TIMER(timer,1000)
                    NEXT-STATE WAIT4           |
        END
STATE   WAIT5       /* waiting state for RNR response */
  EVENT  q2*RR.i = B'0001'          /* receive ready */
  ACTION CANCEL(timer)
        busy := 0
        w0 := RR.NR
        IF w0 => lowest : nt := nt - (w0 - lowest)     |
           w0 <  lowest : nt := nt - (8 - lowest + w0) |
        END
        lowest := w0
        IF RR.PF = B'0' : TIMER(timer,1000)
                          NEXT-STATE WAIT5         |
           RR.PF = B'1' : NEXT-STATE SUBF          |
        END
  EVENT  q2*RNR.i = B'0101'      /* receive not ready */
  ACTION CANCEL(timer)
        busy := 1
        w0 := RNR.NR
        IF w0 => lowest : nt := nt - (w0 - lowest)     |
           w0 <  lowest : nt := nt - (8 - lowest + w0) |
        END
        lowest := w0
        IF RNR.PF = B'0' : TIMER(timer,1000)
                           NEXT-STATE WAIT5        |
           RNR.PF = B'1' : NEXT-STATE SUBF         |
        END
  EVENT  q2*garbage                    /* garbages */
  ACTION CANCEL(timer)
        IF garbage.i = B'1' : TRANSMIT(q4*RNR)
                              TIMER(timer,1000)
                              NEXT-STATE WAIT5    |
           garbage.i = B'0' : TIMER(timer,1000)
                              NEXT-STATE WAIT5    |
        END
  EVENT  TIMEOUT(timer)                /* timeout */
  ACTION TRANSMIT(q4*RNR)
        TIMER(timer,1000)
        NEXT-STATE WAIT5
STATE   WAIT2      /* waiting state for DISC response */
  EVENT  q2*UA.i2 = B'0011'          /* unnumbered ack */
  ACTION CANCEL(timer)
        IF UA.F = B'1' : reply.cntl := B'10'
                         TRANSMIT(q1*reply)
                         NEXT-STATE INITIAL    |
           UA.F = B'0' : TIMER(timer,1000)
                         NEXT-STATE WAIT2      |
        END
  EVENT  q2*DM.i2 = B'1111'       /* disconnected mode */
  ACTION CANCEL(timer)
        IF DM.F = B'1' : reply.cntl := B'11'
                         TRANSMIT(q1*reply)
                         NEXT-STATE INITIAL    |
           DM.F = B'0' : TIMER(timer,1000)
                         NEXT-STATE WAIT2      |
        END
  EVENT  q2*garbage                    /* garbages */
  ACTION CANCEL(timer)
        IF garbage.i = B'1' : TRANSMIT(q4*DISC)
                              TIMER(timer,1000)
                              NEXT-STATE WAIT2  |
           garbage.i = B'0' : TIMER(timer,1000)
                              NEXT-STATE WAIT2  |
        END
  EVENT  TIMEOUT(timer)                /* timeout */
  ACTION TRANSMIT(q4*DISC)
        TIMER(timer,1000)
        NEXT-STATE WAIT2
STATE   SUBF
  EVENT  NULL
  ACTION IF lowest = VS : DISC.P := B'1'
                          TRANSMIT(q4*DISC)
                          TIMER(timer,1000)
                          NEXT-STATE WAIT2    |
        END
        IF busy = 1 : TRANSMIT(q4*RNR)
                      TIMER(timer,1000)
                      NEXT-STATE WAIT5        |
        END
        w0 := nt
        w1 := lowest
        DO w0 > 0 : w2 := w1 + 1
                    IU(w2).NR := VR
                    IU(w2).PF := B'0'
                    TRANSMIT(q4*IU(w2))
                    w0 := w0 - 1
                    w1 := (w1 + 1) % 8        |
        END
        TRANSMIT(q4*RNR)
        TIMER(timer,1000)
        NEXT-STATE WAIT5
STATE   WAIT3    /* waiting state for message from the */
                                              /* upper */
  EVENT  q3*inffu       /* input message from the upper */
  ACTION IF nt < window - 2 : w0 := VS + 1
                              IU(w0).text := inffu.text
                              IU(w0).PF := B'0'
                              IU(w0).NS := VS
                              IU(w0).NR := VR
                              TRANSMIT(q4*IU(w0))
                              VS := (VS + 1) % 8
                              nt := nt + 1
                              NEXT-STATE ESTABLISHED  |
            nt = window - 2 : w0 := VS + 1
                              IU(w0).text := inffu.text
                              IU(w0).PF := B'1'
                              IU(w0).NS := VS
                              IU(w0).NR := VR
                              TRANSMIT(q4*IU(w0))
                              VS := (VS + 1) % 8
                              nt := nt + 1
                              TIMER(timer,1000)
                              NEXT-STATE WAIT4        |
        END
  EVENT  NULL                  /* there is no message */
  ACTION RR.PF := B'1'
        RR.NR := VR
        TRANSMIT(q4*RR)
        TIMER(timer,1000)
        NEXT-STATE WAIT4
STATE   WAIT4      /* waiting state for response from */
                                              /* the secondary */
  EVENT  q2*RR.i = B'0001'          /* receive ready */
  ACTION CANCEL(timer)
        busy := 0
        w0 := RR.NR
        IF w0 => lowest : nt := nt - (w0 - lowest)     |
           w0 <  lowest : nt := nt - (8 - lowest + w0) |
        END
        lowest := w0
        IF RR.PF = B'0' : TIMER(timer,1000)
```

```
                        NEXT-STATE WAIT4              |
      RR.PF = B'1' : NEXT-STATE SUB                   |
    END
  EVENT  q2*UA.i2 = B'0011'          /* unnumbered ack */
  ACTION CANCEL(timer)
    busy := 0
    IF UA.F = B'0' : TIMER(timer,1000)
                     NEXT-STATE WAIT4               |
      UA.F = B'1' : NEXT-STATE ESTABLISHED          |
    END
  EVENT  q2*RNR.i = B'0101'       /* receive  not ready */
  ACTION CANCEL(timer)
    busy := 1
    w0 := RNR.NR
    IF w0 => lowest : nt := nt - (w0 - lowest)      |
       w0 <  lowest : nt := nt - (8 - lowest + w0)  |
    END
    lowest := w0
    IF RNR.PF = B'0' : TIMER(timer,1000)
                       NEXT-STATE WAIT4         |
       RNR.PF = B'1' : NEXT-STATE ESTABLISHED   |
    END
  EVENT  q2*IL.i = B'0'           /* information frame */
  ACTION CANCEL(timer)
    w0 := IL.NR
    w1 := IL.NS
    IF w0 => lowest : nt := nt - (w0 - lowest)      |
       w0 <  lowest : nt := nt - (8 - lowest + w0)  |
    END
    lowest := w0
  IF VR = w1 AND IL.PF = B'0' :  inftu.text := IL.text
                                 TRANSMIT(q1*inftu)
                                 VR := (VR + 1) % 8
                                 TIMER(timer,1000)
                                 NEXT-STATE WAIT4   |
     VR  ≠ w1 AND IL.PF = B'0' : TIMER(timer,1000)
                                 NEXT-STATE WAIT4   |
     VR = w1 AND IL.PF = B'1' :  busy := 0
                                 inftu.text := IL.text
                                 TRANSMIT(q1*inftu)
                                 VR := (VR + 1) % 8
                                 NEXT-STATE SUB     |
     VR  ≠ w1 AND IL.PF = B'1' : busy :=0
                                 NEXT-STATE SUB     |
    END
  EVENT  TIMEOUT(timer)                /* timeout */
  ACTION RR.PF := B'1'
    RR.NR := VR
    TRANSMIT(q4*RR)
    TIMER(timer,1000)
    NEXT-STATE WAIT4
STATE    SUB                /* Retransmission procedure */
  EVENT  NULL
  ACTION IF lowest = VS : NEXT-STATE ESTABLISHED    |
    END
    w0 := nt
    w1 := lowest
    DO w0 > 1 : w2 := w1 + 1
                IU(w2).NR := VR
                IU(w2).PF := B'0'
                TRANSMIT(q4*IU(w2))
                w0 := w0 - 1
                w1 := (w1 + 1) % 8                  |
    END
    w2 := w1 + 1
    IU(w2).NR := VR
    IU(w2).PF := B'1'
    TRANSMIT(q4*IU(w2))
    TIMER(timer,1000)
    NEXT-STATE WAIT4
END-OF-BODY
```

図11. HDLCの記述

6. むすび

本論文では、有限状態機械モデルに基づいた通信ソフトウェアの設計記述を行なう言語IDLを提案した。 そして、HDLCソフトウェア（の8割から9割程度）をIDLで記述し、実証実験を行ない、その適用性を示した。 実際、IDLソースプログラムからコンパイラによって得られたオブジェクトプログラム（アセンブリー言語）は、2500ステップ程度の容量となっており、ソースプログラムに対して約8倍になっている。 このことは、直接アセンブリー言語を用いて記述することに比較して、IDLによる記述がより簡潔さに寄与していることを示している。 従って、プログラム作成効率の面から見て、IDLの有用性が認められる。 またオブジェクトの実行環境とのインターフェースを前述した実行時ライブラリとして作成し、U-200 のOSのもとでは比較的小規模のソフトウェアとして構成できた。

ここでの実証実験の結果から、IDLの実行環境は、従来のリアルタイム型OSとOSに依存した実行時ライブラリを作成することによって、十分にサポートできると考えられる。

他の例として、Stenningのプロトコル[11]がIDLで記述できることが分かっている。

IDLの今後の課題としては、(1) 実行環境の明確なモデル化（NOSモデル）、(2) 有限状態機械設計の方法論の開発とIDL言語との関連、(3) IDL検証系の開発、等があげられる。

『謝辞』IDLコンパイラの開発に当たってくれた川合芳雄君（現在、富士通）に感謝致します。

［参考文献］

[1]A.S.Danthine:'Protocol Representation with Finite State Models',IEEE Trans. Commu.,Vol.COM-28,No.4(1980)

[2]M.Exel,B.T.Popovic and F.Prijatelj:'SL1 Language-A Specification and Design Tool for Switching Systems Software Development',IEEE Trans. Commu.,Vol.COM-30,No.6(1982)

[3]D.P.Pozefsky and F.D.Smith:'A Meta-Implementation for Systems Network Architecture',IEEE Trans. Commu.,Vol.COM-30,No.6(1982)

[4][5][6]'JIS C 6363, C 6364, C 6365',日本規格協会(1978)

[7]H.Zimmermann:'OSI Reference Model-The ISO Model of Architecture for Open Systems Interconnection',IEEE Trans. Commu.,Vol.COM-28,No.4(1980)

[8]A.S.Tanenbaum:'Network Protocols',ACM Computing Surveys,Vol.13,No.4(1981)

[9]B.Hansen:'Distributed Process:A Concurrent Programming Concept',CACM,Vol.21, No.11(1978)

[10]高橋、白鳥、野口:'ネットワークソフトウェアの設計法(6) - IDL とそのコンパイラ -',情処第28回全大2D-8(1984)

[11]N.V.Stenning:'A Data Transfer Protocol',Computer Networks,Vol.1(1976)

[12]松永、水野、井出口:'プロトコルの形式的記述によるプログラム自動生成システムの設計と作成',情処論文誌 ,Vol.22,No.6(1981)