ソフトウェア工学　56－2
（１９８７．９．９）

ＪＳＬ（ジャクソンシステム開発法言語）インタプリタの開発

山住　巌、　加藤　潤三
日本ユニバック（株）

システム開発技法の普及にはそれを支援する開発環境が必要である。ＪＳＥ（Jackson System development Environment）の開発はＪＳＤ（Jackson System Development）の適用を支援する開発環境の提供を目的としている。
ＪＳＥはリスプ専用機（KS-301）上で Common Lisp を使い実現されている。リスプ専用機をシステム開発現場で利用できないことが多いので、ＪＳＥの一部であるＪＳＬインタプリタのサブセットをＰＣ（DS7）にＣ言語を使い移植した。
本稿ではＪＳＬインタプリタの概要と、ＰＣへの移植について報告する。

ＪＳＬ（Jackson System development Language）ｉｎｔｅｒｐｒｅｔｅｒ

Iwao Yamazumi, Junzo Kato
Nippon UNIVAC Kaisha, LTD.
17-51, Akasaka 2-chome, Minatoku, Tokyo 107, Japan

JSE(Jackson System development Environment) is a software development environment for JSD(Jackson System Development) method, and is implemented on the Explorer(Lisp machine) using Common Lisp. On the actual spot of the system development, it is hard to use the lisp machine for their development. So, we have implemented the subset of JSL(Jackson System development Language) interpreter, which was a part of JSE, on our Personal Computer using language 'C'.

This paper describes an overview of the JSL interpreter,and reports on how to convert the subset of JSL interpreter into our Personal Computer.

１．　はじめに

システム開発技法の普及にはそれを支援する開発環境が必要である。ＪＳＤ（Jackson System Development）[1][2]の普及を考えたときＪＳＤに適した開発環境を開発者が要求することが予想できる。

ＪＳＤの仕様を直接実行する処理系[3]にいくつかの改良を加え、ＪＳＥ（Jackson System development Environment）[4]として統合した。このＪＳＥはＪＳＤの適用を支援する開発環境である。ＪＳＥの最初の処理系はリスプ専用機（KS-301）で Common Lisp を使い実現した[5]。

システム開発環境としてはリスプ専用機は高価であるため現場で利用できないことが多い。このため直接実行系をリスプ専用機から比較的廉価なＰＣ（DS7）へ言語Ｃを使い移植した。ＪＳＥはリスプ専用機とＰＣ（DS7）で稼働している。

本稿は、ＪＳＤとその仕様について概略説明し、ＪＳＥの簡単な説明の後、直接実行系の実現とＰＣへの移植を報告する。

２．　ＪＳＤとその仕様

２．１　ＪＳＤの特徴

ＪＳＤは「手法」とそれを使う「手順」からなる。ＪＳＤの「手順」は、仕様作成段階と仕様の実現段階にわかれる。仕様作成段階では実際の実行環境や実現のための資源に関する決定をしない。実現段階で実際の実行環境や実現のための資源に関するすべてを決定する。

ＪＳＤの「手順」には、事実または事実に近い決定は安定しているが事実から遠い決定ほど不安定であるので事実から遠い決定を後にするということと、やさしい決定を先にし、むずかしい決定ほど後にするという簡単な原理がある。この原理にもとづき各々の段階では事実に近い順に決定する。仕様作成段階では、事実の記述から始め実世界のモデルを作る。つぎに機能要求を仕様化するためにモデルを拡張し機能要求をすべて仕様化する。実現段階では仕様を変換し実現環境にあった最終コードにする。事実というよりも開発者が考案する必要があるスケジューリング・アルゴリズムは、実行環境に関する決定を含むため実現段階で追加する。

ＪＳＤの「手法」には、階層構造がない「並行処理」概念と「やさしいわかりやすいプログラム」を「要求される速さと効率を満たすプログラム」に変換する「プログラム変換」手法とが背景にある。始めに仕様として並列に動くプログラムを記述する。次ぎに「プログラム変換」を使い効率よいシステムに仕上げるのがＪＳＤである。「手法」と「手順」とはＪＳＤ固有の関連があるので「手順」と抱合せでＪＳＤの概略を整理する。

ＪＳＤは並列に稼働しているプロセスを明示的に記述する仕様作成段階と、実際の実行環境に適したプログラムにこの仕様を変換する実現段階にわかれる。並列に動くプロセスを記述する利点は、動的な側面を忠実に仕様に反映できることと、複雑な階層構造の導入が避けられることである。実現段階で動的側面の考察をスケジューラの導入で解決し、主プログラムとサブルーチンからなる階層構造にする。プログラムの複雑さを手法によって制御・回避している。

原理的にはＪＳＤの仕様は直接実行可能であり、ＪＳＤを適用したラピッド・プロトタイピングが採用できる。さらに仕様を直接実行し確認した後、一連の変換を経由して実際の実行環境に適したプログラムに変換することが可能である。これは操作的アプローチ（Operatioanl Approarch）[6]によるシステム開発も可能であることを意味する。ＪＳＤを操作的アプローチの一つに挙げるのもこれによる。

２．２　ＪＳＤの仕様

ＪＳＤの仕様は主としてつぎの文書からなる[2]。

(1) 逐次プロセス（sequential processes）からなる分散通信網（distributed network）。この通信網を SSD（System Specification Diagram）という。これについての表記法が定まっている。

(2) それぞれの逐次プロセスは図（またはテキスト）表現のジャクソン木（Jackson Tree）で定義されている。

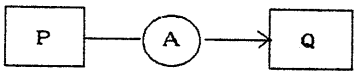
(3) 行動（action）の定義

(3)の行動（action）は外界での事象（event）を示す。プロセスは事象によって起動され、つぎの事象の発生を待つ所で停止する。またシステムの外界と接しているプロセスの他にシステム内部にあるプロセスも同様に事象によって起動する。したがって以降行動も含めて事象ということにする。事象はその存在の他に、それに付随する属性を持つ場合がある。事象の属性によってプロセスがもつデータが定義される。プロセス間通信の事象を通

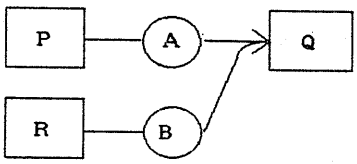
信文と解釈でき、事象毎に対応する通信文の型がある。

プロセス間の通信方式には２通りの基本方式がある。①データストリーム結合による通信②状態ベクトル結合による通信である。いずれも非同期通信である。通信文が通過する通信路は無限長の通信文のバッファーであり、仕様において静的に定義される。図１の例を使いプロセス間通信について述べる。

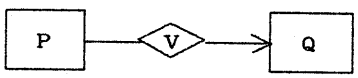
図１(a),(b)はデータストリーム結合による通信である。これの解釈を図２を使い述べる。図２(a)は図１(a)と対応しプロセス間の通信路が一つである結合を示している。通信路Ａで無制限に通信文を待たすことができるのでプロセスＰは通信文を送るために停止することはない。図１(b)はラフマージといい複数のプロセスからの通信文が一つの通信路にまとめられ一つのプロセスに送られる結合を示す。図２(b)はこれの解釈であり、通信路Ａ，Ｂのいずれの通信文を優先的にプロセスＱへ送るかは非決定的であり、先に到着した通信文を優先的にプロセスＱへ送るという原則があるにすぎない。図１(c)は状態ベクトル結合による通信である。これは相手のプロセスの内部変数の値を参照する通信で、例えばプロセスＱがプロセスＰの内部変数の値をプロセスＰの内部状態を変えることなく送ることを要求する通信である。図２(c)はその一つの解釈を示している。



(a) ﾃﾞｰﾀｽﾄﾘｰﾑ結合



(b)ﾗﾌﾏｰｼﾞのある結合



(c)状態ﾍﾞｸﾄﾙ結合

図１ システム仕様図（ＳＳＤ）の例

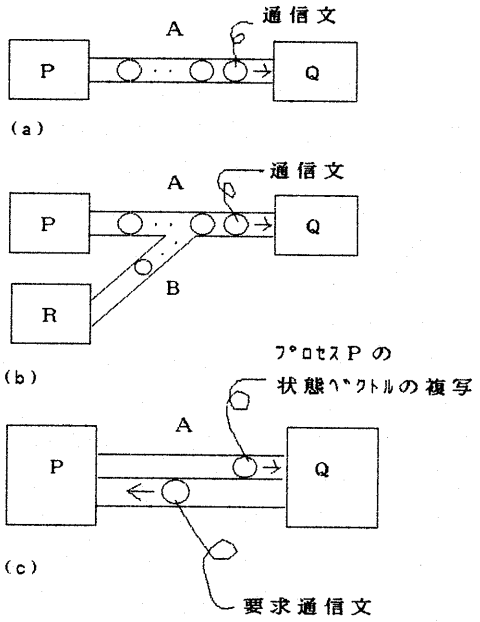


図２ 図１の通信路による表現

３．ＪＳＥの概要

３．１ ＪＳＥの構成

ＪＳＤの仕様作成段階で仕様を作成・編集するためのエディタ、仕様の確認のための直接実行系、仕様を実際の実行環境・資源のもとで効率よく稼働する変換系からＪＳＥは構成されている。ＪＳＥの構成を図３に示す。

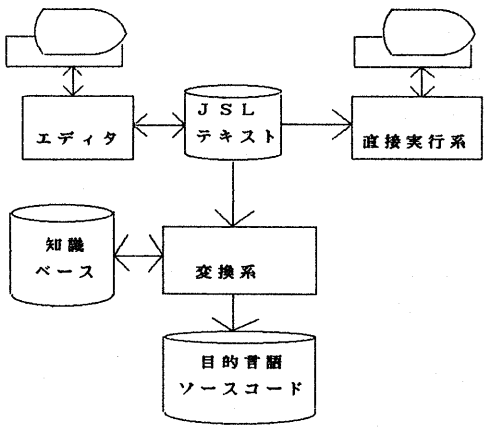


図３ ＪＳＥの構成

３．２　ＪＳＬ

ＪＳＬ（Jackson System development Language）はＪＳＥの重要な言語であり、直接実行系と変換系の両方に共通な言語である。またＪＳＬはＪＳＤの仕様のテキスト表現でもある。ＪＳＬで記述した仕様はＪＳＥのつぎの機能の入出力になる。

(1)直接実行系の入力

(2)変換系の入力

(3)エディタの入／出力

ＪＳＤの仕様はプロセス間通信とプロセスの制御構文で処理の抽象化には成功している一方データオブジェクトの抽象化については不十分[7]という評価がある。これは仕様で使うすべてのデータ型を予め明らかにできないということと、実現への不必要な制約を回避するという精神によると解釈できる。しかし、仕様作成段階を完了した時は出力の書式がドラフトであれ決っているので、完成した仕様では属性のデータ型は決っている。ＪＳＬではデータオブジェクトを陽に記述するようにした。例えば、事象宣言、データストリーム宣言、状態ベクトル宣言の一部がそれである。他に、整数型、実数型、文字型、文字列型という組み込みのデータ型の他に、部分型とか配列・レコード型のような構造型のデータ型宣言ができ、プロセス宣言の中には変数の宣言で使える。

ＪＳＬは以下の宣言からなる。

(1)システム宣言：システムの名前を宣言する。

(2)データ型宣言：データ型、特に構造型や部分型を宣言する。

(3)事象（Event）宣言：プロセス間通信の通信文の型を宣言する。

(4)データストリーム（Data Stream）宣言：通信文の通路と通路を通過できる通信文の型および通信文を待ち行列に入れるバッファーを宣言する。

(5)状態ベクトル（State-vector）宣言：別のプロセスの内部状態のコピーを要求する通信文の通路で通過できる変数とアクセスの順序、通過できる条件を宣言する。

(6)プロセス宣言：プロセスの構造テキスト表現、プロセスは定数、変数宣言部、構造部、操作部、条件部からなる。

図４にSimple Bank[1]というＪＳＬの簡単な記述例を示す。顧客１人につき1口座しか持てない銀行の預金取引で口座の残高をいつでも参照したいという要求を満たすシステムの仕様の抜粋である。

```
SYSTEM : simplebank
 EVENT
   invest customer_name (customer_name amount)
      .
      .
 EVENT END
 DATASTREAM
   c (invest pay_in withdraw terminate)
     : # > _ > customer_1;
      .
      .
 DATASTREAM END
 STATEVECTOR
   cv   (balance) : customer_1 >> _  > enquiry_fn;
 STATEVECTOR END
 PROCESS
   customer_1
    STRUCTURE
     customer_1 SEQ
       .
       .
     customer_1 END
    OPERATIONS
     1: balance := 0
       .
       .
    CONDITIONS
     c1: (withdraw or pay_in)
       .
       .
    customer_1 END
       .
       .
 PROCESS END
SYSTEM END
```

図４　ＪＳＬの例

４．　直接実行系とその実現

直接実行系はＪＳＬを解釈し実行する処理系である。基本構成はリスプ専用機とＰＣとは同じなので、実現した処理系とは独立に説明する。

４．１　直接実行系の構成

直接実行系はＪＳＬパーサとＪＳＬ実行系からなる。その構成を図５に示す。

４．１．１　ＪＳＬパーサ

ＪＳＬパーサはＪＳＬテキストの構文解析と静的な意味解析をする。

ＪＳＬパーサは以下を生成する。

事象テーブル（event table）：事象宣言を表現しており、通信文パーサが通信文を解釈し宛先のプロセスを決定し実行系内部の通信文を作るのに必要な情報をもつ。

プロセス間結合のテーブル(connector table)：データストリーム宣言、状態ベクトル宣言を表現しており、通信文パーサがどの通信路に通信文を送るかを決定するための情報をもつ。

補助テーブル(auxilialy table)：状態ベクトルのアクセス法、データストリームのラフマージについての情報をもつ。

プロセス宣言テーブル（process class table）：プロセス宣言の表現で、局所変数、構造部の解析木、命令や条件式の解析木をもつ。

４．１．２　ＪＳＬ実行系

図５で示すようにＪＳＬ実行系は通信文パーサ、プロセス解釈系、スケジューラ、ユーザインターフェイス系、ユーティリティからなり、リスプ専用機ではそれぞれオブジェクト指向言語であるフレーバ（Flavor）のクラスで実現している。通信文パーサとプロセス解釈系はＪＳＬパーサによって生成された諸テーブルを参照する。またリスプ専用機ではプロセスとバッファーはフレーバのクラスで表現し、通信をオブジェクト間のメッセージ通信で実行することによって、ＪＳＬ実行系は構成要素間のデータの交換をすべてフレーバ間のメッセージ通信に統一した。ＰＣではＪＳＬ実行系の実現方法が異なり、これを”５．ＰＣへの移植”で述べる。プロセスとバッファーの表現については”４．２ＪＳＬ実行系の内部表現”で詳しく述べる。

以下に、ＪＳＬ実行系の構成要素の概要を述べる。

■通信文パーサ：事象をＪＳＬ実行系内部の通信文に変換し、これを宛先のプロセスのバッファーに送る。宛先のプロセスの状態によって次の２通りの処理をする。①宛先のプロセスが存在すれば、そのプロセスに結合されているバッファーの待ち行列に通信文を加える。②宛先のプロセスがまだ生成されていないとき、そのプロセスとプロセスに付随するバッファーを生成する。生成され

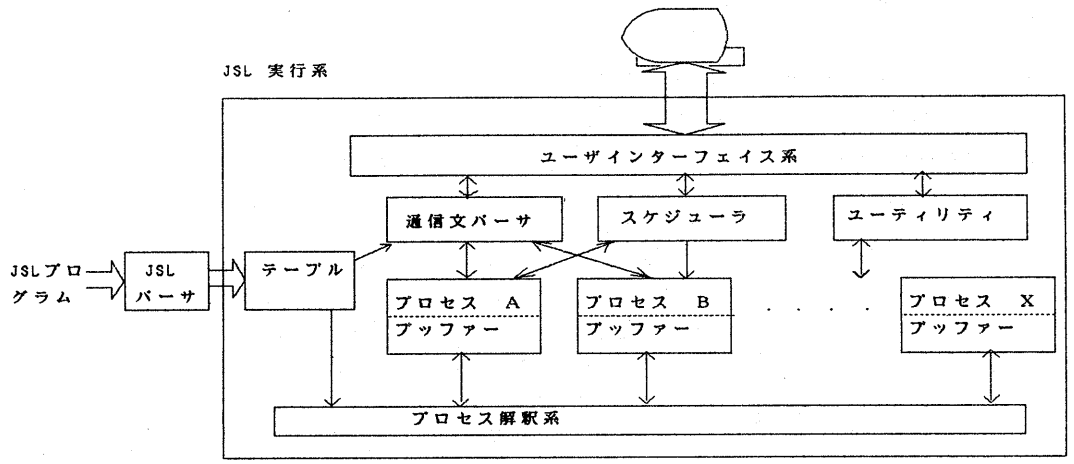


図５　直接実行系の概要

たプロセスの内部状態は初期化される。逆に、プロセスから外部へ通信文を送るとき、プロセスの出力する実行系内部の通信文をユーザインターフェイスの書式に変換してユーザインターフェイスに送る機能を持つ。

■プロセス解釈系：プロセスの初期状態にするとか、プロセスの実行・停止をプロセス宣言テーブルにある解析木（構造、命令、条件式）を参照して実行し内部状態を更新する。

■スケジューラ：ランダムにプロセスを走査し実行可能なプロセスを起動する。すべてのプロセスが通信文待ちで停止するまでスケジューラは起動可能なプロセスを起動し続ける。起動可能なプロセスがなくなったときスケジューラは制御をユーザインターフェイス系に戻す。ただし、途中でスケジューラからユーザインタフェイス系に制御を戻すことができるように常にユーザインターフェイス系を監視し、ユーザインターフェイス系が実行の停止指令を受け取ったとき制御をユーザインターフェイス系に戻す。

■ユーザインターフェイス系：主な機能を以下に列挙する。①システムの外で発生している事象を受け取り通信文パーサへ送る。②システムの外にあるプロセスの状態ベクトルの値を要求する通信文を通信文パーサから受け取ると、システムの外にあるプロセスの状態ベクトルの値を送る手続きを呼び出す。この手続き呼び出しの結果を回答として通信文に編集し通信文パーサへ送り返す。③スケジューラの停止指令などの利用者からの指令を解析し実行する。

■ユーティリティ：プロセスの状態ベクトルの追跡やプロセスが受け取った通信文の追跡などの追跡機能や、現在の状態ベクトルの値を表示する。

４．２　ＪＳＬ直接実行系の内部表現

ＪＳＬにおいてはプロセスとバッファーが基本の構成要素である。プロセスはプロセス宣言で定義され、バッファーはデータストリーム宣言と状態ベクトル宣言で定義される。プロセス間の通信は必ずバッファーを経由する。通信文はその中の宛先（プロセスの識別子とキューされるバッファーの識別子）へ送られる。通信文の宛先がないときその宛先は生成される。プロセス生成の順序は　①プロセス・インデックスに宛先のプロセスの登録の有無を調べ、②なければ内部識別子およびプロセスを作りプロセス・インデックスに登録する。このとき同時に必要なバッファーを生成しバッファー・インデックスに登録する。プロセスとバッファーの生成が終了した後、プロセスを初期化する。である。

図６のシステム仕様図を直接実行したときのプロセスとバッファーのＪＳＬ実行系の状態を図７で示す。

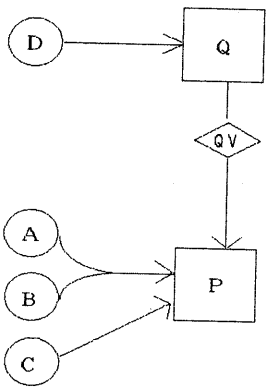


図６　簡単なシステムの例

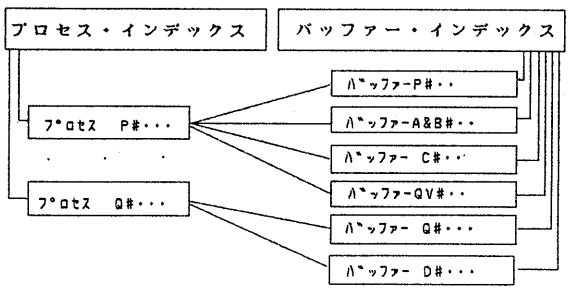


図７　ＪＳＬ実行系の状態

プロセスの表現（representaion）を図８にバッファーの表現を図９に示す。実行系のインタプリタは①停止状態から判断して起動したのがデータストリームか状態ベクトルのいずれかの通信文であったなら、対応するバッファーから通信文を取り出しこれを通信文のトレースに追加する。②前の局所変数とテキスト・ポインタを状態ベクトルのトレースに追加し、③その後テキスト・ポインタを進めながら命令を実行しプロセスにある局所変数を更新する。④プロセスを停止する命令（sread,getsV, wait命令）に達したときテキスト・ポインタの進行を止め、停止した命令の種類と次ぎ受け入れる通信文の型（A の通信文とか Bまたは C の通信文とか QV の回答の通信文）を停止状態に格納し制御を放棄する。という順序で処理する。

プロセスに付随するバッファーはすべて受信用である。システムの外部へ出力するデータストリームはリスプ専用機ではユーザインターフェイス系に付随し、ＰＣではＪＳＬ実行系が保持している特殊なプロセスに付随する。状態ベクトル参照の場合、状態ベクトル参照通信命令を解釈系が解読したとき状態ベクトル要求通信文を通信文パーサに送る。参照されるプロセスはその「状態ベクトル参照要求通信文バッファー」に通信文があればそれの送り先へ状態ベクトルのを回答として通信文パーサに送り返す。その回答は通信文パーサを経由して状態ベクトル参照用のバッファー（QV 通信文バッファー）に入る。

識別子
局所変数
テキスト・ポインタ
停止状態
状態ﾍﾞｸﾄﾙのﾄﾚｰｽ
受け入れた通信文のﾄﾚｰｽ
.ﾊﾞｯﾌｧｰへのﾎﾟｲﾝﾀ
状態ﾍﾞｸﾄﾙ参照要求通信文ﾊﾞｯﾌｧｰ
QV通信文ﾊﾞｯﾌｧｰ
A通信文ﾊﾞｯﾌｧｰ
B&C通信文ﾊﾞｯﾌｧｰ

図８　プロセスＰの表現

５．　ＰＣへの移植

ＰＣ上の基本構成もリスプ専用機の基本構成を保存するように移植するという方針でＰＣ上の処理系を設計した。移植の方針として、Common Lisp の構造はＣ言語の構造型に変換し、補助関数や Common Lisp の関数は同じ機能を持つ手続きに書き換えることにした。ここではフレーバで記述された部分の移植について説明する。

例えば、プロセスの主なメソッドとしては①プロセスの初期化（init-process）②データ・ストリームの通信文や状態ベクトル参照の回答による起動（activate）③状態ベクトル参照要求（send-me-copy-sv）④状態ベクトルのトレースを要求する（print-trace-sv）⑤処理済みの通信文のトレースの要求（print-event-seq）⑥起動可能かどうかの検査（can-activate?）がある。

リスプ専用機上の直接実行系ではフレーバの簡単な機構しか使っていないので、以下の方針でＣ言語へ移植した。

(1) フレーバの変数をＣの構造型で定義する。

(2) フレーバのメソッドは手続きにして、図９で示すリンクを持つ。

(3) オブジェクト間のメッセージ通信を手続き呼び出しの階層にする。

ＰＣ上の通信文パーサはフレーバのメッセージ通信機能も持っているように設計した。まず図１０で示す内部構成を作りフレーバとメソッドの関係を表現し、それをそのまま実現しようと考えていた。しかし、メモリサイズの不足が予想できたので、実際の実現ではいくつかの手続き呼び出しのオバーヘッドを避け、メモリを有効に使うため図１０の手続きのリンクをはずし以下のように手続きを直接呼び出している。

バッファーの識別子
受け入れ可能な通信文の型
未処理通信文の長さ
未処理の通信文のリスト
処理済み通信文のリスト

図９　バッファーの表現

例えば、プロセスを起動するとき：

リスプ専用機での

```
(send p-id ':activate msg)
```

という手続き呼び出しをＰＣでは

```
send(p_id,ACTIVATE,msg)
```

と変換し、手続き send の中で

```
method = Method[MethodPtr(PROCESS)+ACTIVATE];
ret = (*method)(p_id,msg)
```

という命令をするようにしていた。これが図９における手続き呼び出しである。

実際の実現はMethod[MethodPtr(PROCESS)+ACTIVATE]が手続きresumeProcess をポイントしているので、これを resumeProcess(p-id,msg) にする。

つまり、

```
(send p-id ':activate msg)
```

を

```
resumeProcess(p-id,msg);
```

に変換した。

同様に他のメソッドも直接手続きを呼び出すように変換した。図１０をもとに変換したため、リスプ専用機での実現の基本構成は変更しなくてすんだ。

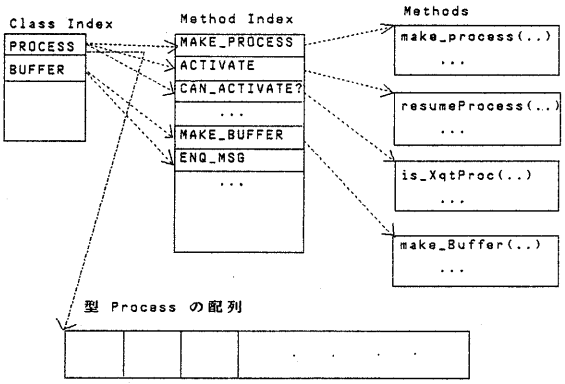


図１０　ＰＣ上での内部表現

６．　まとめ

ＰＣ上で実現したときの問題点は主としてメモリの大きさであった。簡単な例題ならばＰＣ上で実行可能である。しかし現場で使うには実行中の追跡（trace）を保存するための工夫とメモリの制約からくる問題を解消しておくことが必要である。また現在稼働しているＰＣ上の直接実行系はリスプ専用機の部分系であり今後リスプ専用機なみの機能をもたせるかどうかを検討する予定である。またＰＣ上でＪＳＬ直接実行系の実現が可能であることがわかったので、すでにＰＣ上で稼働しているエディタ[8]と統合しＪＳＥの部分系とすることも考えている。

謝辞：日頃、ご指導を賜っている大阪大学基礎工学部鳥居宏次教授に深謝する。また本報告のドラフトにたいして貴重なコメントを頂いた日本ユニバック（株）の森澤好臣、澤田晟司の両氏に感謝する。

参考文献

[1] M.A.Jackson: System Development,Prentice Hall, 1982

[2] J.R.Cameron: An overview of JSD, IEEE Trans. of SE Vol SE-12 No 2, Feb. 1986

[3] 加藤：JSD(Jackson Syatem Development)仕様の実行系の試作,「プロトタイピングと要求仕様」シンポジウム （1986年4月16日），情報処理学会

[4] 加藤、澤田：ＪＳＤ支援系その２，第35回情報処理学会全国大会（1987年9月），情報処理学会 (to appeare)

[5] J.Kato, Y.Morisawa: DIRECT EXECUTION OF A JSD SPECIFICATION, Proc.of COMPSAC87, Oct.1987 (to appeare)

[6] P.Zave : An operational Verus The Conventional Approach To Software Development, CACM, Vol.27 No.2,Feb. 1984

[7] C.Potts, A.Bartlett, B.Cherrie, R.Moclean : Discrete event simulation as a mean of validating JSD specifications, Proc. of 8th ICSE. Aug. 1985

[8] 澤田：ジャクソン法のための木構造エディタの試作，マイクロコンピュータ研究会報告，VOL. 87 NO. 45 (1987年6月30日），情報処理学会