

B8 オンライン・シミュレータ SIMBÖL

山本欣子，松本青樹，立花英里子，小沢友則
(日本情報処理開発センター)

目 次

まえがき

§ 1. SIMBÖLの概略

- 1.1 モデルビルディング
- 1.2 モデルの実行モード
- 1.3 ランタイムインタラクション
- 1.4 インタープリティブモードとコンパイルモード

§ 2. SIMBÖLの言語体系

- 2.1 オンラインシミュレータのworld view
- 2.2 モデル記述の概略
- 2.3 プロセスのスケジューリング
- 2.4 SIMBÖLのステートメント
- 2.5 SIMBÖLのコマンド

§ 3. SIMBÖLのインプリメンテーション

- 3.1 エンバイロメント
- 3.2 SIMBÖLシステムモード
- 3.3 SIMBÖLシステムの構造
- 3.4 モデルの構造
- 3.5 モデルプログラムの実行
- 3.6 プロセスコントロールブロック
- 3.7 スケジューリング処理
- 3.8 プロセスのステータス

まえがき

シミュレーションのオンライン利用の効果が云々されてすでに久しい。モデルの作成や変更、部分的な、あるいは試行錯誤的な実行の繰返し、これらを会話的にまたはインタラクティブにおこなえることの効果は確かにあるであろう。しかし一方シミュレーションの仕事そのものは、相当なCPUタイムと豊富なメモリスペースを必要とする代表的なものの一つでもある。

TSSやオンラインシステム的环境下で、これらの要求を十分に満足させようとするれば、必然

的にかなり大型のシステムを対象とせざるを得ないであろう。

MITのCTSSのもとで使用されていたというOPS-3、その後MULTICSのもとで使用される予定といわれたOPS-4等は、数少ないオンライン・シミュレーション・システムの例であろう。

我々が計画したシステムは、会話型的処理とバッチ型処理の両方の機能を持ち、使用者の希望により任意の切りかえが可能なものとした。シミュレーションという仕事の性質から、会話型処理のみではスピードの点で、実用性がやゝ薄いのではないかと判断したからである。またTSS端末としてCRTディスプレイを主体に考えたのは、簡単なグラフ表示、モデルのステータスのダイナミックなモニタリング等に新しい効果を出させようと試みたためである。

この実験的な開発が、何等かの参考になれば幸せである。特にシミュレーション分野の方々の御批判をいただきたく、今後も実用システムとしての評価、改善を続けてゆきたい。

§ 1 SIMBOLの概要

SIMBOLは離散系モデルのシミュレーションをTSSのもとでオンライン・インタラクティブで行う為に設計された会話型言語で次の様な特徴を持っている。

1. モデルの作成がインタラクティブに出来、随時修正や追加が出来る。
2. モデル全体が完成していなくても部分的にテストしながらモデルを構成して行くことが出来る。
3. プロセスタイプの言語であり、従ってモデル表現のし易さと記述性の豊さを備えている。
4. モデルの実行に関するオペレーション機能が豊富である。
5. トレース機能が豊かでそのオペレーションも自由である。
6. デバックのモデルの一部或は全部をコンパイルしてオブジェクトプログラムを作成し、モデルの実行スピードを上げる事が出来る。
7. モデル実行時に途中でステータスをセーブし、後にそこから再実行する事が出来る。
8. ランタイムにQuitをかけモデルのステータスを参照したり変更したりが自由に出来る。
9. キャラクターディスプレイを端末とし、モデルのステータス等をバーチャートで表示する事もできる。必要に応じてハードコピーの作成も可能である。

以下にこれらの特徴のうち、モデルビルディング、モデルの実行モード、ランタイムインタラクション、およびインタプリタモードとコンパイルモードにつき、簡単に説明する。

1.1 モデルビルディング

バッチ処理用のシミュレーション言語ではほんの一寸でもモデルを修正しようとするときコンパイルをしなくてはならず大変不便である。これは特にシミュレーション言語に限った事ではないが、インタラクティブシステムの最大の利点の1つは臨気応返に手軽にプログラムの修正ができ、ついでにそのテストランが可能だと言う事である。SIMBOLは端末としてキャラクターディスプレイを用いている為、タイプライター端末に比べ、エディティング機能はより

優れている。

1.2 モデルの実行モード

複雑に関連付けられた要素によって構成されるシミュレーションプログラムに於ては、全体を関連させてテストを行う以前に個々の要素で十分なテストをしておいた方が効率が良い。その為にモデル全体が完成していなくても、出来た部分からテストをする手段が用意されるべきであろう。この為SIMBOLではモデルの実行に2つのモードを用意している。1つは個々のプログラム毎に、特にシミュレーションクロックの進行を行わず、他の1つは、一応モデル全体が出そろってからシミュレーションクロックを実際に進めながら実行するモードである。

1.3 ランタイムインタラクション

シミュレーをオンライン・インタラクティブに行う事は、モデル作成時のテスト、修正と言ったインタラクションも重要であるが、それと同時にモデル実行時のインタラクションがより重要である。モデルの実行方法に関する自由度とか、ランタイムのインタラクションというような事は従来のバッチ処理用のシミュレーション言語に於ては十分満足の行くものであったとは言いがたい。オンラインシミュレーションの一つのねらいはこれらの機能を充実する事である。トレース機能、チェックポイントリスタートあるいはモニタリング等の機能はその例である。

1.3.1 トレース機能

一般にバッチ処理用言語ではひとたびトレース指定をすると、途中ではやその必要がないと気付いても、一区切りつくまでは無駄なトレースが続行され、途中で中断させるわけにはいかない。

実行中のプログラムとランタイムに自由にインタラクションがとれば、この問題は解決する。

1.3.2 チェックポイントリスタート

シミュレーションプログラムの実行は一般に時間のかかるものが多い。なんらかの理由で実行最中に一時仕事を中断し、再び別の機会にこれを続けたいという要求が起る。SIMBOLでは仕事を中断した時点でモデルのステータスをセーブして、次の時にはそのステータスを再びロードし実行を再開する、いわゆるチェックポイントリスタート機能を持っている。

一般にシミュレーションでは実行を開始してからしばらくして、系が安定するまでは役に立たないのが普通である。それから種々のパラメータを変えながらシミュレーションを繰り返す事が多いので、毎回系が安定するまでの経過を繰り返すのは無駄であるから、系が安定した事を判定出来たならばこの時のステータスをセーブしておき以後はそこからリスタートを繰り返す利用方法も可能である。この為にステータスを再びロードした時点で一度pauseしユーザがパラメータの値を変えたりする措置が取れる様になっている。

1.3.3 モニタリング

実行中のモデルのスケジュールテーブルの様子や、各種変数の内容などのモデルステータ

スをいつでも参照する事が可能である。これには2通りの方法があり、1つは実行途中でQuitをかけモデルの実行を一度中断し、ステータスを参照する方法と、他の1つは実行を中断せず、随時特定変数の内容などを、バーチャートによってキャラクターディスプレイに表示しその様子を知る方法である。前者はモデルの実行が止っている時にモニターするという意味でスタティックなモニタリングと言う。一方後者はこれに対してダイナミックなモニタリングと言う。これらのモニタリングを利用することにより素早くシステムの状態を把握する事ができる。例えば特定変数の変化を見る事によってその系が安定したか否かを判別する事も出来よう。

1.4 インタープリットモードとコンパイルモード

SIMBOLのモデルはメインプログラム、アクティビティプログラム、プロセデューアと言ったプログラム要素で構成されていて、インタープリティブに実行されるのが普通である。メインプログラムやアクティビティプログラムなどは特にシステムのダイナミックな部分を記述するもので(これらについては後述)お互いに他の要素と複雑な関係を持っているものであるが、プロセデューアはFORTRANで言うサブルーチンやファンクションと同じようにこれだけで独立したスタティックなものである。したがってプロセデューアはデバックが済んでしまえば特にインタープリティブに実行されずともコンパイルしてオブジェクトを作成してしまった方が処理のスピードが向上する。SIMBOLに於てはプロセデューア単位にコンパイルしてオブジェクトを作成しこれをインタープリティブなメインやアクティビティと組合せて実行する事が可能である。さらにモデル全体も完全にデバックが済みあとはプロダクションのみ行えば良い時にはメインとアクティビティ群をまとめてコンパイルし、すでにコンパイルされているプロセデューアとリンクして、まったくSIMBOLプロセッサが介入せずにバッチ的にランさせる事も可能である。

§2 SIMBOLの言語体系

このセクションではSIMBOLのシミュレーション言語としての基本概念について述べ、そのあとに一通り言語機能の説明を行なう。

2.1 オンラインシミュレータのworld-view

シミュレーション言語におけるモデル記述の思想はよくworld-viewと言われる。汎用と言われるシミュレーション言語のworld-viewは、ほぼ幾つかのグループに類別されてしまう。

トランザクション タイプ …… GPSS
イベント タイプ …… SIMSCRIPT, GASP
プロモス タイプ …… SOL, SIMULA

注) GPSSをトランザクションタイプと呼ばず、プロセスタイプの言語とする人もいる。

これらの区別はスケジューリングの単位が何であるかによって分類されることが多い。逆に見れば、スケジューリングの考え方が、すなわち、言語の **world-view** であると言うことができるかもしれない。オンラインシミュレーション言語の **world-view** に、どのようなものが適しているかを簡単に結論づけるのは、むずかしい。トランザクションタイプの言語は、モデルの表現がしやすいし、もともとインタープリティブに実行されるものであるから、オンライン向きであるかもしれない。反面、イベントタイプの言語は使いなれない人にとっては、モデル化するのがむずかしい点、又、イベント単位にプログラムが記述されるので、コントロールの流れがGPSSなどと比べると見通しにくい点などは、むしろオンライン向きでないとも言える。一方、プロセスタイプの言語はどちらかと言うと、2つの中間を行くようなものである。プロセスタイプの言語であるSIMULAを例にとると、GPSSの記述し易さと、SIMSCRIPTにおける表現力の両方を備えていると言えよう。ある意味ではオンライン向きと言えるかもしれない。

SIMBOLの **world-view** は分類するなら、プロセスタイプである。言語仕様はプロセスタイプという意味でSIMULAと類似している点も多いし、プロセスというものに対する解釈は、SIMULAの場合とほぼ同じである。しかし、スケジューリングに関する機能とか、プロセスの扱い方、特に名前付けの問題と、他のプロセスのアトリビュートを参照する仕方についてはSIMULAと違った方法を採用している。

2.2 モデル記述の概略

SIMBOLによるモデルの記述はプロセスを基本に行なわれる。プロセスは、一連の過程の表現で複数イベントをまとめて記述したものである。このプロセスはアクティビティ・ルーチンによって定義される。アクティビティは一種のプログラム単位で、SIMBOLのモデルは他に、メインプログラムと必要ならプロセデューアによって記述される。同種のパターンを持つプロセスは、1つのアクティビティによって定義することができる。従ってアクティビティは同一のパターンを持つプロセスを作る集合の代表と見ることができる。

プロセスは、アトリビュートを表わすために、自分にローカルなデータを持つことができる。プロセスには、ローカルデータによって作られるデータブロックが対応する。同一アクティビティに定義される異なるプロセスには、それぞれにこのデータブロックが割り当てられる。従ってプロセスには常に、ローカルデータを表わすデータブロックとアクティビティプログラムが対応づけられる。この為、プロセスは2つの性格を持つと言われ、前者をデータストラクチャーとしての性格、後者を「動作様式」と言う。

データ・ブロックとしての性格のみしか持たない要素を導入し、これをブロックという。これを特に新しい要素とせず、アクティビティプログラムで変数の宣言のみで実行ステートメントの無いものを作り、これを代用することもできる。事実SIMULAでは、この方法を採用しているが、あくまでも、これはプロセスであるから、その為に余計なコントロール情報が必要になり、メモリーが無駄になる。そこで、はっきりプロセスとは区別したブロックの導入が必

要となる。これは丁度SIMSCRIPTのテンポラリー・エンティティと同じものである。

データ・ストラクチャーとしてのプロセスとデータ・ブロックは全く同じように扱うことができる。そこでこれらを総称してエレメントと呼ぶ。エレメントはテンポラリーな要素であり、ユーザの責任において創成、消去されるものである。従ってその手段が用意されている。

プロセスが自分にローカルなデータを参照することは簡単である。これはGPSSのトランザクションが持つパラメータを自分で参照するのに相当する。しかし、GPSSにおいて、他のトランザクションのパラメータを直接参照する事は簡単ではない。(グループ・エンティティが導入されてから、この扱いにおいて可能となっている。)SIMBOLでは、他のプロセスやブロックのローカル・データを参照する為に **external reference** という手段が用意されている。

プロセスやブロックの集合はセットによって表わされる。また、待ち行列はキューで表わす。すなわち、プロセスやブロックはセットやキューのメンバーとなる。セットとキューの違いはキューの場合、平均待ち時間、平均長さなどの統計を取る点のみである。SIMBOLの場合、セットやキューは、それ自体テンポラリーな要素と考えるので、創成・消去を行わなければならないが、創成はシステム側が自動的に行なっている。

セットやキューに対するオペレーションとしては、次のような事が可能である。

1. メンバーを登録すること。
2. メンバーを取り去ること。
3. メンバーの位置を移動すること。
4. 2つのメンバーの位置を入れ換えること。

メンバーを登録する時には、先頭や最後尾の指定だけでなく、特定メンバーの直前や直後を指定することができる。また、メンバーを取り去る場合にも、先頭や最後尾からだけでなく、メンバー名を指定して取り出すことができる。

2.3 プロセスのスケジューリング

SIMBOLにおけるスケジュールの単位はプロセスである。プロセスの考え方は一見、GPSSのトランザクションと非常に似通っている。事実、GPSSをプロセスタイプの言語として分類することもある。しかし、スケジュールについては、全く異なっている。つまり、トランザクションの考え方では、直接他のトランザクションに対してスケジュールを行なう機能を持っていない。他のトランザクションをコントロールする時には、全て間接的に、例えばロジックスイッチなどを用いて行なっている。プロセスタイプの言語では、これができる点ではっきり違いが出てくる。この事は、モデルの概念にも大きな影響を持っていて、GPSSでは、自分を中心に(つまり、どう動くか)、記述するのに対して、プロセスタイプの言語では、1つのプロセス内では、自分を中心に記述すると同時に、他のプロセスに対しても何か働きかける3人称的な表現が可能となる。この意味で表現形式がより自由になる点に特徴がある。

さて、SIMBOLのスケジューリング・メカニズムは、スケジュールテーブルとスケジューラーによって構成

されている。スケジューラはプロセスをスケジュールテーブルに登録し、プロセスの実行時刻を指定する。スケジューリング機能とは、スケジュールテーブルをモディファイする手段である。

SIMBOLでは、以下のスケジューリング機能を持っている。

1. プロセスをスケジュールする。
2. すでにスケジュールされているプロセスをスケジュールし直す。
3. すでにスケジュールされているプロセスをキャンセルする。
4. プロセスをターミネイトさせる。
5. 実行中のプロセスを待ち状態にする。
6. 待ち状態にあるプロセスをスケジュールする。
7. スケジュールされているプロセスをインタラプト状態にする。
8. インタラプト状態にあるプロセスを開放しスケジュールする。

2.4 SIMBOLのステートメント

SIMBOL言語は、ステートメントとコマンドに大別され、ステートメントは通常モデルを記述する為に使用し、コマンドは種々のオペレーションをコントロールする場合に使用する。コマンドについては、次節にゆずり、ここではステートメントの構文とその意味につき説明する。紙面の都合上、ここでは**SIMBOL**で特長的なものに重点をおき、他はごく簡単にふれる。尚、記述は原則として、**BNF**によるが次の記号も併用する。

[] カッコの中を省略してよいことを表わしている。

例えば $\langle A \rangle ::= \langle B \rangle [\langle C \rangle]$ は

$\langle A \rangle ::= \langle B \rangle \mid \langle B \rangle \langle C \rangle$ を表わす。

2.4.1 プログラム構成

SIMBOLへのプログラム・インプット単位は、ラインと呼ばれる。

ラインの形式は、

$$\langle \text{ライン} \rangle ::= \langle \text{ラインナンバー} \rangle [\langle \text{ラベル} \rangle] \langle \text{ステートメント} \rangle ; \mid$$
$$\% \langle \text{コマンド} \rangle ;$$

である。前述のとおり、**SIMBOL**のモデルプログラムは、メインプログラム、アクティビティプログラム、プロセデュアによって構成される。このうちメインプログラムは必ず必要であるが、他は必要に応じて組込めばよい。また、プロセデュアは、いわゆるサブルーチンに当るものと、ファンクションに当るものがある。ファンクションには**PROCEDURE**の前に、**REAL**とか**INTEGER**とかいったタイプを付けて区別する。これらのプログラムは全てラインによって構成される。

各ステートメントは、必要に応じてラベルを付けることが可能であり、ステートメント間の参照はこのラベルを通じて行なわれる。

ラインナンバーは直接ステートメントから参照することはできず、エディットをする際のみ使われる。

2.4.2 変数

SIMBOLで扱える変数のタイプは4つある。

Real ……値として実数をとる。

Integer ……値として整数をとる。

Logic ……値として**true, false**の論理値をとる。

Element ……値として**Element**, つまり, プロセス, ブロックをとる。

このうち, **Element** 変数はプロセスやブロック, キュー, セットの名前として使われる。(セット, キューの場合は, その**head**を示すブロックの名前と考える。)

これらの変数は, 単純変数または添字付変数として使うことができる。又, 変数は, もしメイン・プログラムで宣言されたものならばグローバル変数, アクティビティプログラムやプロセデュアで宣言されたものならばローカル変数と呼ばれる。グローバル変数はどのプロセスからでも, 直接その名前を書くことにより値を参照できるものであるが, ローカル変数は, 他のプロセスやプロセデュアからは参照不可能である。しかし, プロセス間で外部変数参照の方法(2.4.3)を用いれば, 参照可能である。

2.4.3 外部変数の参照

自分以外のプロセスにローカルな変数の値を参照するには, プロセスの名前の後に, そのプロセスが属するクラスの名前(後述)と変数名を並べて書くことにより行なう。例えばプロセス**P**はアクティビティ**A**に属しており, その中のローカル変数**X**を参照するには

P : A . X

と書く。

一般には後述する単純エレメント式を用いて

<単純エレメント式> : <クラス名> · <変数>

の形をとる。

ここに

<クラス名> ::= <アクティビティ名> | <ブロック名>

でクラス名はプロセスを定義するアクティビティプログラムの名前か, ブロックを定義するブロック名のいずれかである。(ブロックに対しても, そこで定義されている変数も, この外部変数の参照の形式で参照する。)

2.4.4 式

SIMBOLでは, 3つのタイプの式を扱う。以下にBNFで示す。

<式> ::= <算術式> | <論理式> | <エレメント式>

<算術式> ::= <項> | <加減作用素> <項> | <算術式> <加減作用素> <項>

<項> ::= <因子> | <項> <乗除作用素> <因子>

<因子> ::= <1次子> | <1次子> * * <1次子>

<1次子> ::= <符号のない数> | <算術変数> | <算術関数呼出し> | (<算術式>)

<加減作用素> ::= + | -
 <乗除作用素> ::= * | /
 <論理式> ::= <論理項> | <論理式> ! <論理項>
 <論理項> ::= <論理因子> | <論理項> & <論理因子>
 <論理因子> ::= <論理1次子> | <論理1次子>
 <論理1次子> ::= <論理値> | <論理変数> | <関係> | <論理関数呼出し> |
 (<論理式>)
 <関係> ::= <算術式> <関係作用素> <算術式>
 <関係作用素> ::= < | = < | = | > = | > | / =
 <論理値> ::= TRUE | FALSE
 <エレメント式> ::= <単純エレメント式> | <生成式>
 <単純エレメント式> ::= NONE | <エレメント変数> | <エレメント関数呼出し> |
 <単純エレメント式> : <クラス名> . <エレメント変数>
 <生成式> ::= NEW <クラス名> [(<実パラメタ部>)]

意味 算術式、論理式については省略する。エレメント式は、プロセス、ブロック、セット、キューを参照する為に使われ、これらを値とする。生成式の値は新しく発生したプロセスやブロックである。

2.4.5 エレメントの発生、消去に関するステートメント

NAME ステートメント

構文

<NAME ステートメント> ::=
 NAME <エレメント変数> : = <エレメント式>

意味 エレメント変数にエレメント式の値を代入する。

DESTROY ステートメント

構文

<DESTROY ステートメント> ::= DESTROY <単純エレメント式>

意味 ブロックまたはプロセス（単純エレメント式の値）を消去する。

2.4.6 セットやキューと関連したステートメント

INSERT ステートメント

構文

<INSERT ステートメント> ::=
 INSERT <エレメント式> -> <単純エレメント式> [<位置指定>] |
 INSERT <エレメント変数> : = <生成式> -> <単純エレメント式>
 [<位置指定>]

<位置指定> ::= FIRST | LAST | AFTER <単純エレメント式> |

BEFORE<単純エレメント式>

意味 第1の形式はエレメント(エレメント式の値)をセット(単純エレメント式の値)に挿入する。この際、位置指定がFIRSTならセットの先頭に、LASTなら最後に、エレメントを挿入する。又、BEFORE、AFTERの指定があったならエレメント(単純エレメント式の値)の直前、直後に挿入され、位置指定が省略された場合には、FIRSTと指定されたものとみなす。

第2の形式では生成したエレメントを、第1の形式と同じ要領で挿入し、そのエレメントに名前(エレメント変数)を付ける。この名前はメンバー名となる。

REMOVEステートメント

構文

```
<REMOVEステートメント> ::=  
    REMOVE [ <単純エレメント式> ] <-<単純エレメント式> |  
    REMOVE <エレメント変数> := <エレメント式> <-<単純エレメント式>
```

意味 セット(単純エレメント式の値)から指定されたエレメント(単純エレメント式の値)を取り出す。取り出すエレメントの指定が省略された場合には、セットの先頭にあるエレメントが指定されたものとして処理する。また、後半の形式は、上記と同じ要領でエレメントを取り出した後、それに名前(エレメント変数)を付ける。

REMOVEQステートメント

構文

```
<REMOVEQステートメント> ::=  
    REMOVEQ [ <単純エレメント式> ] <-<単純エレメント式> |  
    REMOVEQ <エレメント変数> := <エレメント式> <-<単純エレメント式>
```

意味 セットをキューで置き換えれば、REMOVEステートメントと全く同じである。

2.4.7 スケジューリング・ステートメント

SCHEDULEステートメント

構文

```
<SCHEDULEステートメント> ::=  
    SCHEDULE <エレメント式> <スケジュール句> |  
    SCHEDULE <エレメント変数> := <生成式> <スケジュール句>  
<スケジュール句> ::=  
    <スケジュール指定> [ PRIOR ]  
<スケジュール指定> ::= A T <時刻> |  
    DELAY <時間> | AFTER <単純エレメント式> |  
    BEFORE <単純エレメント式>  
<時刻> ::= <算術式>
```

<時間> ::= <算術式>

意味 第1の形式では、プロセス(エレメント式の値)をスケジュール句の様式に従ってスケジュールする。

第2の形式では、プロセスを発生し、それに名前(エレメント変数)を付けた後、形式1と同じくスケジュールする。

スケジュール指定の意味はAT<時刻>なら、その時刻に実行されるように、DELAY<時間>なら、現時刻から指定時間後に実行されるように、BEFORE<単純エレメント式>およびAFTER<単純エレメント式>ならば、単純エレメント式が示すプロセスの直前、直後に実行されるようにスケジュールすることを表す。もしPRIORの指定があると、同時にスケジュールされているプロセスのうち、一番始めに実行されるようにスケジュールされる。この指定は、BEFORE、AFTERを使う場合には意味を持たない。

RESCHEDULEステートメント

構文

<RESCHEDULEステートメント> ::=

RESCHEDULE<単純エレメント式><スケジュール句>

意味 すでにスケジュールされているプロセス(単純エレメント式の値)をスケジュール句の指定に従ってスケジュールし直す。

CANCELステートメント

構文

<CANCELステートメント> ::=

CANCEL[<単純エレメント式>]

意味 スケジュールされているプロセス(単純エレメント式の値)のスケジュールをキャンセルする。そのプロセスはパッシブな状態となる。プロセスの指定がない場合には、このステートメントを実行しているプロセスを対象とする。

TERMINATEステートメント

構文

<TERMINATEステートメント> ::=

TERMINATE[<単純エレメント式>]

意味 プロセス(単純エレメント式の値)をターミネイト状態にする。プロセス指定がない場合には、このステートメントを実行しているプロセスを対象とする。

DELAYステートメント

構文

<DELAYステートメント> ::= DELAY<時間>

意味 現在このステートメントを実行しているプロセスを指定時間後に再び実行されるようにスケジュールする。従ってこのプロセスはアクティブな状態から、パッシブな状態に移

する。

WAITステートメント

構文

<WAITステートメント> ::=

WAIT [<エレメント式>] - > <単純エレメント式> [<位置指定>] |

WAIT <エレメント変数> : = <生成式> - > <単純エレメント式> [<位置指定>]

意味 第1の形式では、プロセス（エレメント式の値）をキュー（単純エレメント式の値）に位置指定により挿入し、パッシブな状態とする。

第2の形式はプロセスを新たに発生し、名前（キューにおける名前）を付けて挿入する点のみ異なる。

WAKEステートメント

構文

<WAKEステートメント> ::=

WAKE [<単純エレメント式>] <- <単純エレメント式>

[<スケジュール句>] |

WAKE <エレメント変数> : = <単純エレメント式> <- <単純エレメント式>

[<スケジュール句>]

意味 キュー（単純エレメント式の値）からプロセスを取り出し、スケジュール句の指定に従ってスケジュールする。もし、スケジュール指定がなければ、DELAY 0 とされたものとみなす。また、プロセスの指定がない場合には、キューの先頭にあるプロセスを対象とする。第2の形式では、キューから取り出したプロセスに名前（エレメント変数）をつける点のみ異なる。

INTERRUPTステートメント

構文

<INTERRUPTステートメント> ::=

INTERRUPT <エレメント変数> - > <単純エレメント式>

意味 スケジュールされているプロセス（エレメント変数の値）をインタラプト状態にし、セット（単純エレメント式）に登録する。この際、エレメント変数の値は、このステートメントを実行したプロセスになる。

RESUMEステートメント

構文

<RESUMEステートメント> ::=

RESUME <エレメント変数> <- <単純エレメント式>

意味 セット（単純エレメント式）の先頭にあるプロセス（インタラプト状態にある）を現時刻にスケジュールし、エレメント変数の値をそのプロセスにする。

2.4.8 繰り返しステートメント

DOステートメント

構文

```
<DOステートメント> ::= DO<終端ラベル>, <繰り返し節> [WITH<理論式>]
<繰り返し節> ::= <数値による繰り返し節> | <エレメントによる繰り返し節>
<数値による繰り返し節> ::=
    FOR<コントロール変数>=<初期値>, <終値>[, <増分> ]
<初期値> ::= <算術式>
<終値> ::= <算術式>
<増分> ::= <算術式>
<コントロール変数> ::= <算術変数>
<エレメントによる繰り返し節> ::=
    FOR<コントロールエレメント変数> :=<開始エレメント>,
    <終了エレメント>, <次のエレメント>
<コントロールエレメント変数> ::= <エレメント変数>
<開始エレメント> ::= <単純エレメント式>
<終了エレメント> ::= <単純エレメント式>
<次のエレメント> ::= <エレメント関数呼び出し>
<終端ラベル> ::= <ラベル>
```

意味 このDOステートメントから終端ラベルの指すLOOPステートメントまでを、繰り返し節の指定に従って繰り返し実行する。もし、WITH以下の指定があった場合には、繰り返しの各回において、論理式が真の時にだけ実行し、疑であればその回の実行をせず、次の回に進む。

数値による繰り返し節では、まず1回目にコントロール変数の値を初期値とし、以下は毎回増分を加えて行き、コントロール変数の値が終値以下である間、実行を繰り返す。

また、エレメントによる繰り返し節では、コントロールエレメント変数の値を開始エレメントとし、次の回はエレメント関数の値をコントロールエレメント変数に代入し、以下この操作を繰り返し、終了エレメントと等しくなるまで実行して終る。

2.4.9 その他のステートメント

宣言ステートメント

REALステートメント ……Real変数の宣言をする。

INTEGERステートメント ……Integer変数の宣言をする。

LOGICステートメント ……Logic変数の宣言をする。

ELEMENTステートメント ……Element変数の宣言をする。

TABLEステートメント ……統計をとるためのテーブルを定義する。

代入ステートメント

LET ステートメント ……算術式の値を算術式に、論理式の値を論理変数に代入する。

実行制御ステートメント

GO TO ステートメント ……指定されたステートメントへコントロールを移す。

IF ステートメント ……論理式の値により、実行順序をコントロールする。

入出力ステートメント

READ ステートメント ……ファイルからデータを読む。

WRITE ステートメント ……ファイルにデータを書く。

ACCEPT ステートメント ……キャラクタ・ディスプレイからデータを読む。

DI SPLAY ステートメント ……キャラクタ・ディスプレイにデータを表示する。

OPEN ステートメント ……ファイルをオープンする。

CLOSE ステートメント ……ファイルをクローズする。

FILE ステートメント ……ファイルの定義をする。

FORM ステートメント ……入出力形式を指定する。

統計処理ステートメント

SAMPLE ステートメント ……ヒストグラムを作るために、データをテーブルに登録する。

2.5 SIMBOLのコマンド

コマンドは、直接モデルに組込まれることなく、モデルの扱いに関連した種々の操作を行なうためにある。ステートメントは指示があるまで実行されないのに対して、コマンドは、ただちに実行される点が異っている。本来ステートメントとして使われるものの幾つかはコマンドとしても使い事ができる。

ここでは、コマンドとしてだけ、使われるものの機能を簡単に説明する。

SAVE ……現在作っているモデルの一部または全部をソースプログラム・ファイルに登録する。

LOAD ……ソースプログラム・ファイルに登録されているプログラムや、相対形式プログラム・ファイルに登録されているプログラムをロードする。

COMPILE ……ソースプログラム・ファイルに登録されているプログラムをコンパイルし、相対形式プログラムを作成、相対形式プログラム・ファイルに登録する。

DELETE ……ソースプログラム・ファイルや相対形式プログラム・ファイルに登録されているプログラムを削除する。

EDIT ……現在作成中のモデルプログラム（コアメモリー上にある）に対して、ラインの削除、追加、変更などのエディットを行なう。

EXECUTE ……モデルの一部を実行させる。プロセデューアを単独でテストする時などに用いる。

START……モデルのステータスを初期設定してシミュレーションの実行を始める。この場合には、実際にシミュレーションクロックを進めながら実行する。

GO ……実行途中でquitをかけ、実行を中断した後、に再会したい場合に用いる。

CHECK……モデルの実行をquitにより中断し、このコマンドを使えば、その時のステータスがファイルにセーブされる。

RESTART ……CHECKコマンドでセーブしたステータスを再ロードする。ロードが完了すると、pause状態となるので、必要があればパラメタの値の変更などを行ない、実行を再開するときにはGOコマンドを用いる。

CLEAR……モデルのステータスをイニシャライズする。

TRACE……トレース指定を行なう。

なお、次のステートメントは、コマンドとしても用いることができる。

スケジューリング・ステートメント

LETステートメント

NAMEステートメント

DESTROYステートメント

INSERTステートメント

REMOVEステートメント

REMOVEQステートメント

入出力ステートメント

§ 3 SIMBÖLのインプリメンテーション

この章ではSIMBÖLのプロセッサの概略を述べる。

3.1 エンパイロメント

SIMBÖLはFACÖM230/60にインプリメントされてTSSモニターM-Vのもとで稼動する。プロセッサの大きさは約30kwであって現在のメモリー容量ではオーバレイ構造を取らざるを得ない為、現在のところリエントラントにはしていない。

◦端 末

SIMBÖLの端末は前述の如くキャラクターディスプレイを用いている。しかし現在接続されているディスプレイはTSS端末としての資格が無くここからシステムの稼動が出来ない。その為、タイプライター端末と組合せて使用し、ここからLOG-IN、LOG-OUTを行ったり、ディスプレイのハードコピー用として使っている。しかしプログラムの入力、エディット結果の出力を始め一切の会話は原則としてキャラクターディスプレイを通して行い事になっている。(ディスプレイはメーカーの開発を待つて出来るだけ早い機会にTSS 端末として使用出来るものに入れ換える予定である。)

○オペレーティングシステムとの関連

SIMBÖLはM-Vのもとでデマンド用のユーザプログラムの1つとして動くようになっている。またエディット機能はSIMBÖLのエディット機能の他にファイルに登録されているプログラムに対してはM-Vに備えられているデマンド用エディター・LINED やバッチ用ライブラリーエディター・LIBEの使用も可能である。従ってファイルフォーマットはこれらのユティリティと互換性を持たせてある。又SIMBÖL が介入せずにモデルをランさせる為にSIMBÖL がコンパイルして出すオブジェクトプログラムはリンクエディター・LIED が扱えるフォーマットに合せ実行形式プログラムが作成出来る様にしてある。この場合もプログラムはデマンドジョブとして実行するので端末から簡単なインタラクションを取る事が出来る。

3.2 SIMBÖLシステムモード

ユーザがSIMBÖL を利用している場合、それがモデル作成中であるかとか、あるいはモデルの作成中であるかなどという状態をSIMBÖL では4つのモードに分けて考える。

1. プリパレーションモード(Preparation mode)

登録されているプログラムをデリート(delete)したり、内容をリストアップするなど、直接モデルの作成や実行に関係ない作業を行うモードである。

2. モデリングモード(Modeling mode)

これは、言わゆるモデルビルディングを行う為のモードで、モデルプログラムのステートメントをインプットしたり、修正したり、又、プログラムの一部を実行しテストを行うモードである。

3. パーシャルテストモード(Partial test mode)

これはモデリングモードでプログラムを作成している過程で、テスト的にその一部を実行し、途中でQuitをかけ、変数の内容やその他の状態を知るためにインタラクションを取った場合のモードである。

4. オバーオールテストモード(Over-all test mode)

モデル全体がある程度完成した後に実際にシミュレーションクロックを進めて実行している最中にQuitをかける事によりこのモードに移る。パーシャルテストモードとの違いは、前者はモデル作成中であるのに対し、このモードは完成した(プログラム全部がそろっているという意味で)モデルの実行中である点である。次に示す図1はこれらのモードの相互関係および各種コマンドによるモードの切り変り等のステータス遷移の関係を示したものである。

3.3 SIMBÖLシステムの構造

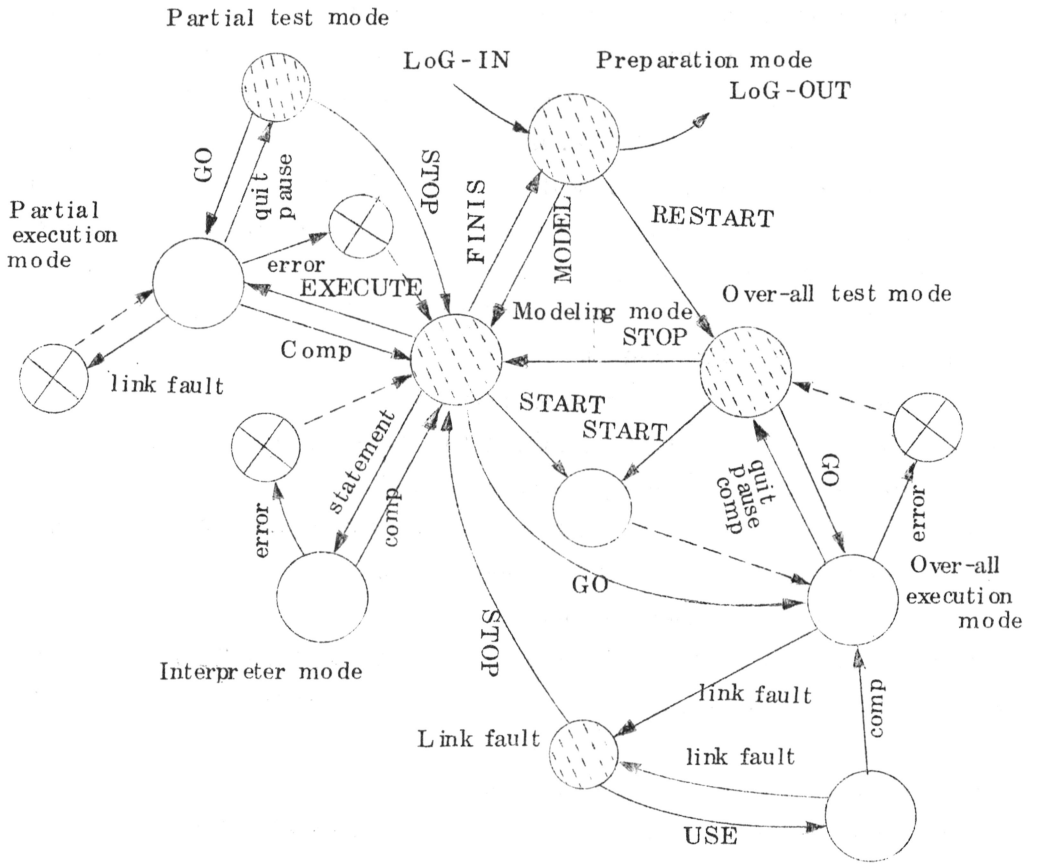
SIMBÖL を構成するモジュールを簡単に紹介すると以下のごとくである。

○コマンドコントローラ(Command Controller)




入ってきたコマンドに従って各処理モジュールにコントロールを渡す。

○エディター(Editor)

モデル作成中にコアメモリーにあるプログラムに対してエディティングを行う。



記号の説明

-  エラー表示を示す
-  コマンド受付可能状態
-  処理中

----> 自動的に遷移する事を示す

大文字 コマンドによって遷移する事を示す。

小文字 アクションなどにより遷移する事を示す。

comp 処理が完了した事を示す。

図1 SIMBOLシステムモードの遷移図

- コンパイラ(Compiler)

プロセデュー単位、又はメインプログラムとアクティビティプログラムの組をコンパイルし相対形式プログラムを作成する。

- インタープリター(Interpreter)

インタープリティブに実行するオブジェクトを作成する。

- ローダー(Loader)

相対形式プログラムやライブラリルーチンのローディングを行う。

- リンカー(Linker)

SIMBÖL はプログラム間のリンクを実行時に行っているため、この処理を行う。

- スケジューラ(Scheduler)

モデルの実行時にプロセス間の実行順序をコントロールする。

- イクセキューター(Executer)

インタプリティブな実行をする時、ローカルシーケンスに従ってプログラムの実行を管理する。又、トレースに関してもこのモジュールが扱う。

3.4 モデルの構造

前述のとおりSIMBÖL で扱うモデルプログラムの構造は2つの形式があり、1つはインタープリティブに実行されるプログラムで、他はコンパイルされてオブジェクトプログラムが作成されているものである。

これらの2つの形式のプログラムが混在しているモデルも扱う事ができるが次のルールに従っていなくてはならない。

1. SIMBÖLが介入して実行する場合

メインプログラムとアクティビティプログラムは全てインタープリティブモードのプログラムで、プロセデューはどちらの形式であってもよい。

2. SIMBÖLが介入せず実行する場合

この方法はM-Vのリンクエディターにより実行形式プログラムが統合された1つのプログラムとしてSIMBÖL から独立して実行される場合である。そのためにはメインプログラム、アクティビティ、プロセデューは全てコンパイルされている事を必要とする。

ついでにコンパイル単位について述べておくと、メインプログラム、アクティビティはまとめて1つのエレメント(M-Vの相対形式プログラムの単位)としてコンパイルし、プロセデューは1個ずつ単独なエレメントとしてコンパイルされる。(アクティビティを単独でコンパイルする事はできない)。又、コンパイルされたオブジェクトプログラムはM-Vの相対形式プログラムの形式と同格である。

インタープリティブに実行されるプログラムもインプットされたソースプログラムがそのままの形で保存されているわけではなく、ラインごとインクリメンタルコンパイルをおこなうことができるだけオブジェクトを作成し、実行スピードを上げるように工夫している。例えば、

```
100 LET A=B+C;
```

というラインに対しては、

```
FL    B
FA    C
FST   A
```

の様なオブジェクトコードを作成し、他に種々の情報(ライン番号、次や前のラインへのポインタなど)を付けて1つのラインに対するオブジェクトとしている。これをオブジェクトラインと呼ぶ。

インタプリティブに実行されるプログラムはこの様なオブジェクトラインがチェイニングされて構成されている。それとは別に、ソースプログラム自身もEDITコマンドの要求に応じられるようにチェイニングされ保存されている。

3.5 モデルプログラムの実行

インタプリティブモードにおける実行は、おもにEXECUTERとSCHEDULERがつかさどる。(コンパイルモードで実行する場合にはSCHEDULERがrun time systemとしてモデルに組込まれる)

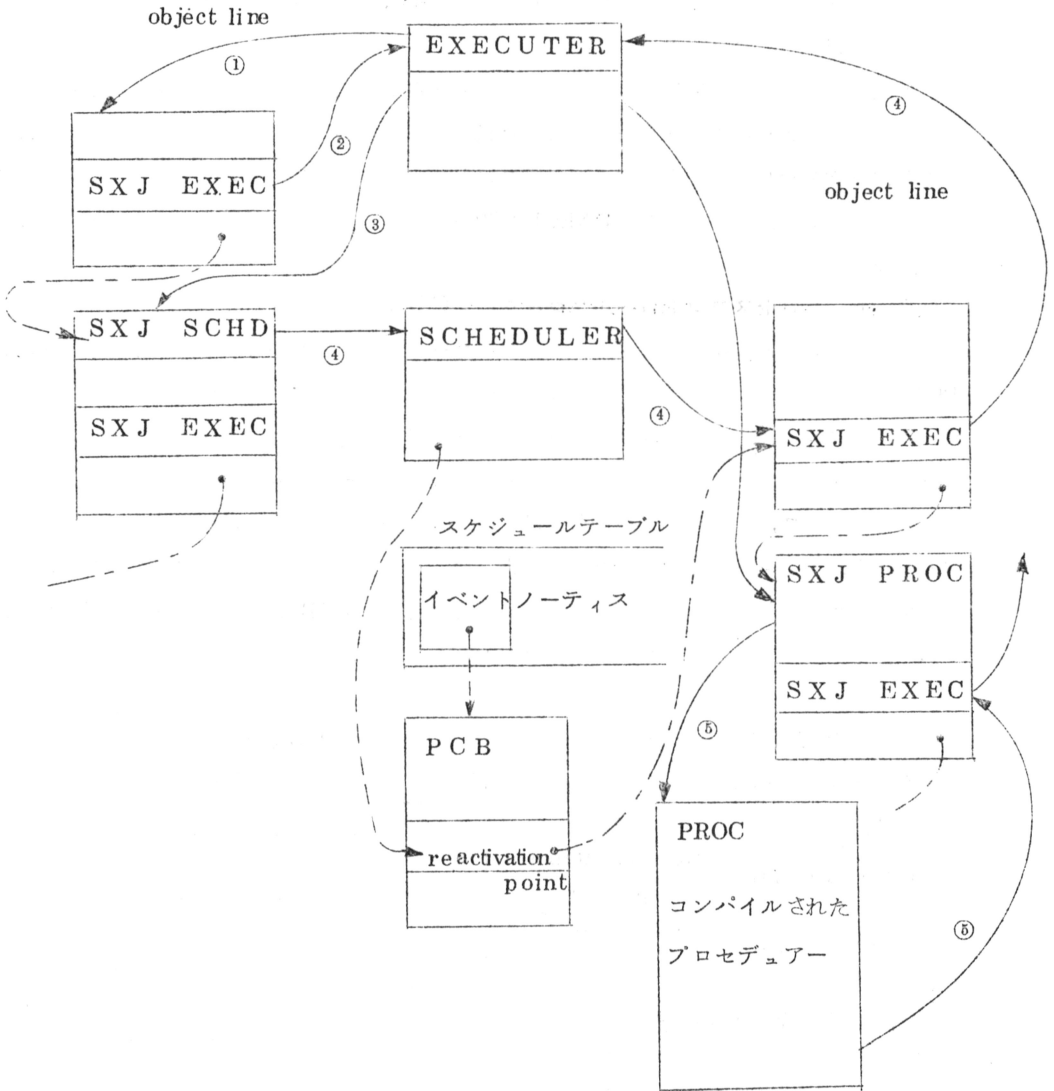
EXECUTERは1つのアクティビティやプロセデューア内でそのローカルシーケンスに従って次のラインを見つけて実行させたり、トレーススイッチの管理や処理などを受け持っている。一方、SCHEDULERはプロセス間の実行順序を決定し、EXECUTERに対するラインを教え実行を依頼する。この為SCHEDULERはスケジュールテーブルを管理している。

両者とも実行順序を決める機能を持っているが、EXECUTERがローカルシーケンスの決定をするのに対し、SCHEDULERは言わばモデル全体の実行順序の決定をする点が異なっている。

図2は両者の関係を示したものである。

これらの実行手順をさらに詳しく説明すると次の様になる。

1. EXECUTERからオブジェクトラインにはJump命令でコントロールを渡す。
2. オブジェクトラインを実行し終るとEXECUTERへSXJ命令(インデックスにこの命令の次のアドレスを入れてjumpする命令)で帰ってくる。
3. EXECUTERはインデックスのアドレスに従って次に実行すべきオブジェクトラインを見つかる事ができる。従って、またそこにjump命令でコントロールを渡し、以上の手順を繰り返す。
4. 途中のラインがスケジューリングステートメントに対するものであった場合は、そのラインにEXECUTERがコントロールを渡すとラインの中にSCHEDULERをサブルーチン形式で呼び出すオブジェクトができており、SCHEDULERにコントロールが渡る。SCHEDULERはその帰り番地に従ってreactivation point(後述)を書きかえステートメントに対応する処理をおこなう。そしてスケジュールテーブルの先頭にあるPCB(後述)



番号は次の説明の番号と関連している。又実線はコントロールの流れを，破線は参照関係，一点鎖線はチェーンを示す。

図2 モデルと EXECUTER, SCHEDULER の関係

に書かれている **reactivation point** にコントロールを戻す。戻った番地は常にオブジェクトラインの途中で、次に **EXECUTER** への **SXJ** 命令が入っている。従って、又 **EXECUTER** へのコントロールが渡る。

5. 他のプロセデューアを呼び出すステートメントのオブジェクトラインである場合には、そのプロセデューアにコントロールが渡った後、もしそれがコンパイルされたオブジェクトであれば全部実行した後にもとのラインにコントロールが戻るし、インタープリティブなオブジェクトであればラインごとに **EXECUTER** を介して実行され最後にもとのラインに戻ってくる。

3.6 プロセスコントロールブロック (PCB)

プロセスが新たに発生するとプロセスコントロールブロック (PCBと略記) が作成される。PCBは2つのデータブロックより成り、お互いにチェイニングされている。一方のデータブロックにはプロセスの実行をコントロールするために必要な情報が書き込まれており、サイズはどのプロセスについても一定になっている。他方はプロセスにローカルな変数のために割付けられるエリアで、ローカル変数の数だけ **area** が取られる。従って、一般にサイズはバリエーションである。もしローカル変数を1つも持たないプロセスの場合、後者は作成されない。又、アクティビティプログラムから見た場合、それと関係する PCB は1つだけでなくその時点でそのアクティビティに属すプロセスの数だけ存在している。

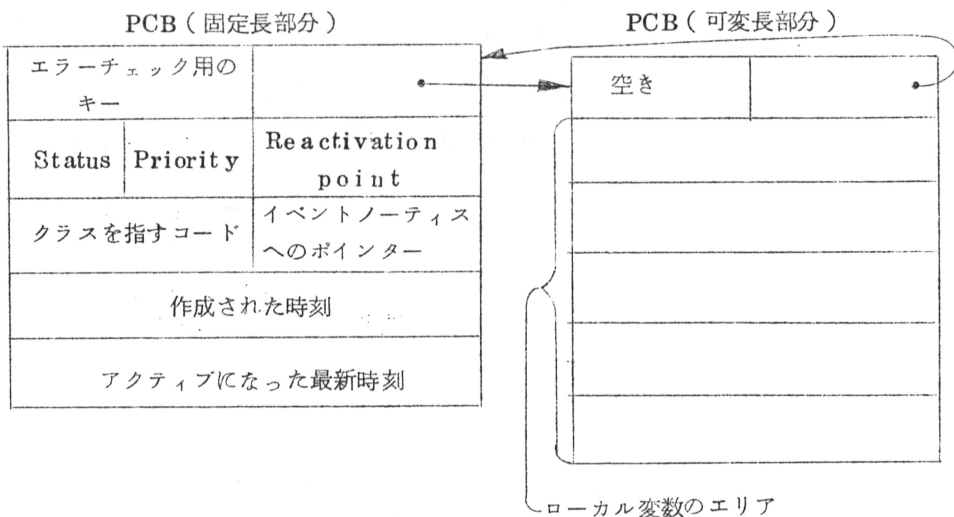


図3 PCBのフォーマット

プロセスに対応して定義される **reactivation point** とは、プロセスの実行が中断され (スケジューリングステートメントの実行により) 他のプロセスにコントロールが渡るような場合の再びもとのプロセスを実行する時の再開アドレスである。従って **reactivation point** はアクティビティプログラムが次に実行すべき番地として定義できる。この為に作成

されたばかりのプロセスの **reactivation point** としてはアクティビティの入口番地が記入されており、完全に実行を終わってしまったプロセスの **reactivation point** は、その事を示す特別なコードが入っている。

3.7 スケジューリング処理

イベントノティス

プロセスのスケジューリングに関連してイベントノティスが定義される。スケジューリングはスケジューラーがスケジュールテーブルを介して行いが、イベントノティスはプロセスに対応してスケジュールテーブルに登録されるレコードであり、3ワードからなっている。

	PCBのポインタ
先行イベントノティス	後続イベントノティス
スケジュールタイム	

図4 イベントノティスのフォーマット

このうちスケジュールタイムはこのプロセスが実行される時刻を表わしており、**SCHEDULE** ステートメントによって

SCHEDULE PROCESS / AT 1000

と **PROCESS** を時刻1000に実行すべく指定があった場合には、**PROCESS** に対応するイベントノティスのスケジュールタイムに1000が記入される。

イベントノティスはスケジュールタイムの順にチェーンされており、先行イベントノティス、後続イベントノティスのポインタはその順序にしたがったものである。

3.8 プロセスのステータス

プロセスにはスケジューリングに関して次に示す5つのステータスが定義される。

a. アクティブ (**active**)

現在実行中のプロセスのステータス

b. スケジュールド (**scheduled**)

スケジュールテーブルに登録されており原則として順番が来たら実行される状態

c. パッシブ (**passive**)

スケジュールテーブルに登録されておらず将来他のプロセスによってスケジュールされないかぎり実行される事のない状態

d. ターミネティド (**terminated**)

他のプロセスによってこの状態にされる事もあるが普通はプロセスの実行を完全に終了しており、もう2度と実行される事のない状態。この場合PCBはデータストラクチャーとしてシステムに残っている。(例えばセットのメンバーとして残っており後で参照される事がある)

e. インタラプティド (**interrupted**)

スケジュールテーブルに登録されていたプロセスが他のプロセスによってインタラプトを

かけられた状態。この状態においては将来他のプロセスによりこの状態を解除されるまで実行される事はない。

これらの状態とイベントノーティスや reactivation point の関係は次の様になる。

ステータス	イベントノーティス	reactivation point
アクティブ	有り	有り
スケジュールド	有り	有り
パッシブ	無し	有り
ターミネイティド	無し	無し
インタラプティド	有り	有り

注) インタラプティド状態のイベントノーティスはスケジュールテーブルでなくセットの中につながれている。

参 考 文 献

1. M. Markowitz 他 (1963), SIMSCRIPT A Simulation Language, RAND
2. P. J. Kiviat 他 (1968), The SIMSCRIPT II Programing Language, RAND
3. Ole-Johan Dahl and Kristen Nygaard (1967), SIMULA A language for programing and description of discrete event systems. Introduction and user's manual, Norwegian Computing Center
4. Malcolm M. Jones (1968), Incremental Simulation on a Time-Shared Computer, MAC-TR-48
5. Greenberger, Jones, Morris, Ness, On-Line Computation and Simulation : The OPS-3 System, The M. I. T. Press
6. Daniel Teichroew and John Francis Lubin (1966), Computer Simulation — Discussion of the Technique and Comparison of Languages C. ACM, Vol 9, Nov. 10

本 PDF ファイルは 1972 年発行の「第 13 回プログラミング・シンポジウム報告集」をスキャンし、項目ごとに整理して、情報処理学会電子図書館「情報学広場」に掲載するものです。

この出版物は情報処理学会への著作権譲渡がなされていませんが、情報処理学会公式 Web サイトの https://www.ipsj.or.jp/topics/Past_reports.html に下記「過去のプログラミング・シンポジウム報告集の利用許諾について」を掲載して、権利者の検索をおこないました。そのうえで同意をいただいたもの、お申し出のなかったものを掲載しています。

過去のプログラミング・シンポジウム報告集の利用許諾について

情報処理学会発行の出版物著作権は平成 12 年から情報処理学会著作権規程に従い、学会に帰属することになっています。

プログラミング・シンポジウムの報告集は、情報処理学会と設立の事情が異なるため、この改訂がシンポジウム内部で徹底しておらず、情報処理学会の他の出版物が情報学広場 (=情報処理学会電子図書館) で公開されているにも拘らず、古い報告集には公開されていないものが少からずありました。

プログラミング・シンポジウムは昭和 59 年に情報処理学会の一部門になりましたが、それ以前の報告集も含め、この度学会の他の出版物と同様の扱いにしたいと考えます。過去のすべての報告集の論文について、著作権者（論文を執筆された故人の相続人）を探し出して利用許諾に関する同意を頂くことは困難ですので、一定期間の権利者検索の努力をしたうえで、著作権者が見つからない場合も論文を情報学広場に掲載させていただきたいと思います。その後、著作権者が発見され、情報学広場への掲載の継続に同意が得られなかった場合には、当該論文については、掲載を停止致します。

この措置にご意見のある方は、プログラミング・シンポジウムの辻尚史運営委員長 (tsuji@math.s.chiba-u.ac.jp) までお申し出ください。

加えて、著作権者について情報をお持ちの方は事務局まで情報をお寄せくださいますようお願い申し上げます。

期間：2020 年 12 月 18 日～2021 年 3 月 19 日

掲載日：2020 年 12 月 18 日

プログラミング・シンポジウム委員会

情報処理学会著作権規程

<https://www.ipsj.or.jp/copyright/ronbun/copyright.html>