計算機システム記述用言語に関する一考察*

田中穂積** 張　素華***

Abstract

To design or to develop a computer system, we had to do a duplicate work, simulation and actual system programming. Each of them was a separate work and was dispatched to each working group. To eliminate this inefficient duplication, the author proposes a computer system description language named CSDL that enables us to describe both a simulation model and actual system programs. Programs described by CSDL are used once for simulation and once for actual system programs.

Many problems are expounded in the paper to realise such a kind of languages as CSDL. As a first step, many fundamental notions or principles are extracted from existant computer systems. Then these are abstructed into correspondant basic functions of CSDL. For instance, new concept, I-ID relation, is introduced to represent interruptions.

CSDL is a higher level language based on SIMULA and is designed to have many convenient functions to describe operating systems.

1. はじめに

計算機システムに対する需要の拡大は，大規模化した計算機システムの出現をうながした．システムの大規模化とともに，システムに組み込まれるソフトウェア・システムが膨脹複雑化し，柔軟性がうすれ，その開発は，量的質的にきわめて困難な情況にある．こうした傾向は，ソフトウェア・システムの中核ともいうべきオペレーティング・システム（以後 OS と略記）に特に顕著で，OS の能率的開発手法の確立がさけばれている．

OS 開発の困難さの第1は，OS 記述用言語として，低水準・低生産性で論理の見通しのきかないアセンブラ語が依然として広く使用されていることである．OS 記述用のすぐれた(高級)言語の開発が急務とされているのは，このためである．しかし，こうした OS 記述用言語は，アセンブラ言語の単なる延長上のものとしてとらえるべきではない．既存の OS の構造，ダイナミズムを精確に分析し，これに含まれる基本概念の抽出と整理の過程を経て後に生まれる，いわゆる問題向き言語の性格を強く帯びたものでなくてはならない．本稿の第1の目的は，こうした立場からの OS 記述用言語の1つのあり方を示すことにある．これをかりに，CSDL とよぶことにする．OS の開発を困難にする第2の問題点は，完成後の性能予測をなしえないままに OS の開発が進行してしまうことである．それゆえ，OS の一応の完成後に，モニタを組み込むなどして，実物ランの計測結果にもとずき実物の手直しを何度か繰り返している．こうした現状は，それが困難な作業であるだけに解消したい．それには，早い機会に精密な性能予測を行なうことで，シミュレーションはそのための有力な武器である．しかし，OS の動作をシミュレートするためには，OS そのものの記述とハードウェアの記述の双方を含む，いわば計算機システム全体のシミュレーション・モデルを作成せねばならず，こうしたモデルの精密な記述自体，かなりの労力を要する作業となる．現実の OS 作成にも大量の労力をつぎ込まなければならないことから，最終的に現実の OS 作成作業が，シミュレーションに優先進行し，所望のシミュレーション結果を得られないままに，その作業が中断してしまうケースが多い．これはシミュレーションと現実の OS 作成グループが並列的に存在し，作業が2元化していることから生ずる．しかし，こうした作業の2つの流れは，記述言語の共通化により一元化できる．既存の離散事象形シミュレーション言語が，

* An Consideration of a Computer System description Language, by Hozumi Tanaka (Pattern Processing Section, Pattern Information Division Electrotechnical Laboratory) and 張 素華 (Chang Sou Wah) (Waseda University, Graduate Student)

** 電子技術総合研究所パターン情報部図形処理研究室
*** 早稲田大学大学院

OS の構造，ダイナミズムを把握するに有効な概念を含んでいるとの指摘とともに，この種の言語による作業の一元化の可能性が認識されはじめている[15),21)]．しかし，既存の離散事象形シミュレーション言語は，必ずしも OS を含む計算機システムの記述に便利な機能を含んではいない[15),18),21)]．特に，計算機システムに典形的な，割込み機構の記述に対する便利な概念，機能を欠いていることは問題である．詳細は，本文中で考察するが，OS は割込み処理を核に構造化されているだけに，割込み機構の正確簡明な記述が必要とされる．本稿で提案する CSDL は，割込み機構の記述に便利ないくつかの新概念を導入するなどして，シミュレーションと現実の OS 作成作業の一体化を可能にする1つの高級言語の具体化を目標に設計される．

CSDL による OS 開発手順は，**Fig. 1** のようである．CSDL により，計算機システムのハードウェアとソフトウェアは明確に2分して記述される．これら総体は，シミュレータの入力となり，システムの性能評価予測に使用される一方，ソフトウェア（OS 記述が主体）記述部は，トランスレータの処理を介して現実の“もの”としてそのまま組み込まれて動作する*1．

2. 計算機システムと CSDL

本章は，計算機システムに含まれているさまざまな基本概念の抽出と整理の作業と，これらがいかに抽象化され CSDL の機能として吸収されているかを，シミュレーション言語の立場から，informal に説明する*2．

2.1 プロセス

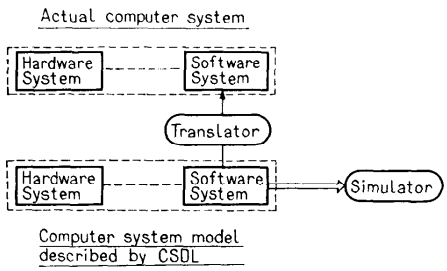

Fig. 1 The development process of computer system by CSDL.

*1 CSDL のベースとなる離散事象形シミュレーション言語は SIMULA である．これと，データ構造操作用として，Hoare らの提案した言語仕様をとり入れているが，これらの詳細は，文献3), 22)を参照してほしい．なお，SIMULA は，ALGOL 60 のスーパ・セットとして設計された言語であることを注意しておく．

*2 CSDL の詳細は，電子技術総合研究所彙報に報告する予定である．

計算機システムは，多数のプロセスの発生消滅と，プロセス相互の複雑な干渉作用により全体の動作が規定されている．プロセスとは，計算機システムを構成する相互に並列動作を行ないうる独立した実体をいう．SIMULA で定義されているものと同義である．

OS の制御プログラムが管理するプロセスは，ソフトウェアでシミュレートされたもので，チャネルなどは金物として具体化されているプロセスとみることができる．プロセスの導入により，ハードとソフトの双方を，同一レベルの“もの”として扱うことが可能になる．シミュレータの側からみるとシミュレーションを進める場合の制御の基本単位がプロセスということになる．

プロセスを，計算機システムの基本要素とした理由はつぎのとおり．よく知られているように，並列動作を行なう実体を多数含む系の記述法に2とおりある。1つはイベント中心の記述法であり，SIMSCRIPT はその代表的言語である[12),14)]．これは，システムをグローバルな立場から見通して，システム全体をイベント単位にばらばらに分解し，各イベントごとにイベントの起こるタイミングをも含めて記述するものであるが，この作業はそう簡単なものではない．これにたいし，プロセス中心の記述法は，そのプロセスに関係するイベントの発生を逐一追って記述する形式であり，個々のプロセスの記述が他プロセスの存在を比較的意識せずに独立して記述できる利点がある．こうした記述は人間の自然な思考法によく適合する．このことは記述したもののわかりやすさにも通ずる．また MULTICS をはじめ数多くのシステムで，プロセス中心の OS 構成法が整理[4),16),17)]され具体化されていることを考慮し，CSDL の基本的動作要素としてプロセスを採用した．

2.2 プロセスの生成と消滅

プロセスの生成は SIMULA とほぼ同様の型式を採用する．プロセスの動作の種類（クラス）を **activity** で規定する．プロセスの生成は **new** 文で行なう．

① 〈process designator〉: =**new** 〈activity identifier〉〈actual parameter〉

プロセスの制御は〈process designator〉を指定して行なう．プロセスの消滅は **delete** 文を使う．

② **delete** 〈process designator〉

SIMULA 同様，**new** 文により1つのプロセスが生成されると，このプロセスを制御するためのレコード（データ・ブロック）が2つ作られる．第1のレコード

は，すべてのプロセスに共通のデータのわく組みをもつもので，Hoare のレコード定義の仕方に従うとつぎのようになる[22]．

record class process (**reference** (process) pred, suc, **integer** priority*3, **reference** (CSDL) process aspect)

第2のレコードは，個々のプロセスに固有の情報を蓄える部分で，**new** 文の指定する **activity** の body で定義する．第2のレコードは，第1レコードの process aspect フィールドを仲介にアクセスされる．pred, suc は，**record class** process をポイントする変数が pred, suc ということで，プロセスの set, membership の制御に使う[6]．第1の **record class** processは，システムに定義が組み込まれているので，使用者が定義しなおす必要のないものである．

例を示す．

```
reference (process) son;
son := new initialize;
priority (son, 1)
. . . .
  activity initialize;
 {activity の具体的記述} activity の body
  end initialize;
```

son の指すプロセスの動作は initialize という名のアクティビティに属す．組み込み手順 priority (son, 1)は，son の指すプロセスに優先度1を与える．上述の **record class** process の priority フィールドの値に1がセットされる．**new** 文により生成直後のプロセスの状態は suspend である (**2.3.2** 参照)．

2.3 プロセスの状態

計算機システムを構成するプロセスは，さまざまな状態を推移し動作する．状態推移に忠実に模擬することが，シミュレータの基本的な役割である．CSDL はプロセスの状態としてつぎのものを考える．

(i) Autonomous……execute, time delayed

(ii) Heteronomous……suspend, conditional wait

2.3.1 Autonomous 状態

現在，なんらかの動作を行ないつつあるプロセスの状態を autonomous という．シミュレータ内部では，この状態はさらに execute と time-delayed にわけられ制御される．Autonomous プロセスは (外部からこれを変える強制力が作用しない限り)，時間の経過につれて自動的に execute 状態となり，具体的なシミュレーションが実施される．Time-delayed 状態のプロセスは，将来 execute となるべき時間が決まっており，それまでの間一時的に，そのプロセスのシミュレーションが保留されている．

Time-delayed 状態は，シミュレータ内部に仮に設けられた人工的状態であることを注意しておく．現実の世界では，time-delayed 状態のプロセスはやはり動作しているのであり，execute 状態のプロセスとなんら相違はない．離散事象形シミュレーションの世界で時間を消費する人工的動作を行なっているものが time-delayed 状態のプロセスであると理解すればよかろう．execute 状態のプロセスは，**wait** 文を実行すると time-delayed 状態となる．

③ **wait** ⟨expression⟩;

⟨expression⟩ は消費時間を指定する．この時間は，現実世界ではプロセスの走行時間として吸収される．

2.3.2 Heteronomous 状態

Heteronomous 状態のプロセスは，conditional wait か suspend 状態で，事象の発生ないしは割込みの発生により autonomous 状態になる．suspend は，他プロセスから直接制御の対象となる．他プロセスが実行再開を命ずるまで待ち行列で待つプロセスは，シミュレータ内部では suspend 状態となる．conditional wait は，指定の事象が生起するまで待つ状態をいう．**wait until** 文を実行したプロセスは conditional wait 状態になる．事象の生起は，**activate** 文で行なう．

④ **wait until** ⟨compound event⟩

⑤ **activate** ⟨event designator⟩ ⟨event status⟩

suspend は，CSDL の **enter, suspend, start** およ

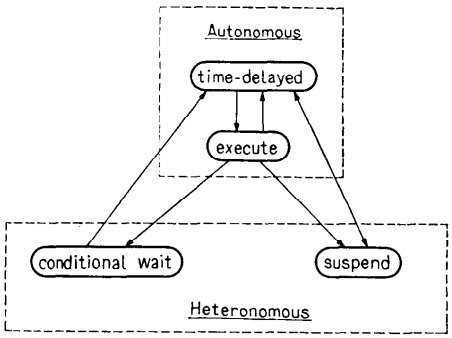

Fig. 2 Process states and state transition diagram

*3 第1レコードは，SIMULA の **element** に **integer** priority が加わったもので，これを加えた理由は，プロセスに優先順位が与えられ，これらを優先順に配列する必要がしばしばあることを考慮したからである．

び **signal** 文が関係する．**enter** 文と **signal** 文については，**2.6** で詳述する．

⑥ **suspend** 〈process designator〉

suspend 文は，autonomous プロセスを suspend にする．**start** 文は，suspend プロセスを autonomous にする．

⑦ **start** 〈process designator〉

たとえば，**start new** controlprocess;
　　wait 1000;
　　stop;

は，control process なる名称の **activity** を実行するプロセスを1つ生成しこれを autonomous に，その後1000単位時間だけ消費し，**stop** 文でシミュレーションを止める．

2.4 OS を構成するプロセスの類別

OS 内部にはソフトウェアでシミュレートされたプロセスが多数存在するが，これらを制御，被制御の立場から2つに分類し，それぞれ CP (Control process), CDP (Controlled process) とよぶことにする．CP は制御プログラムを **activity** としてもち，Traffic Controller, Process Controller などともよばれているもので，CDP を制御する立場のプロセスである．Saltzer は Traffic Controller を1つのプロセスに埋めこまれたモジュールとみなしたが[16]，本稿ではこれを1つの独立したプロセスとみなす．同様の類別は，Iliffe が行なっており，CP を system process とよんでいる[10),20)]．筆者らもこれと同様の考え方で GPSS/360 を用いて ETSS の記述を試みた経験があるが，この考え方の自然さを確認できた[7]．

CP の動作する契機は，割込みの発生である．CP の主要な仕事は，割込み処理と CDP 制御，すなわち，次に実行すべき CDP の決定と，状態の設定（再起動番地やレジスタ類の設定）や，起動を行なったり，場合によっては，CDP の実行中止などを行なう．CP の個数は，計算機システムに含まれる割込み機構数，いいかえれば中央処理装置数に一致する．

2.5 事　象

事象とは，システム内部に生起する出来事をいう．システムのダイナミズムを表現する基本要素の1つである．広義に考えると“命令の実行”も，計算機システム内部に生起する事象の1つであるが，以下ではこの種のミクロな出来事でなく，さらに大きな単位の出来事のことを事象とよぶ．すなわち，プロセス相互の間で共通の関心をもたれ，プロセス相互の間に情報の交換をともなう出来事を事象とよぶことにする．

事象に関連するプロセス数は，基本的には2個である．事象の発生を知り，その対応処置をとるプロセスと，事象の発生を知らせるプロセスである．

activate 文は，事象の発生を他プロセスに知らせるためのものである．事象は，たとえば，つぎのわく組みをもつレコードで定義される．

record class event (**boolean** flag, reference (process) owner, **integer** event status)

事象の発生を記憶する flag，この事象の発生を待つプロセス owner，**activate** 文で送られてくる情報を受け取る event status フィールドがある．event の生成はたとえば，

reference (event) complete;
complete : =(**false**, **current**[*4], 0);

一方 event の発生は，つぎのようにする．

activate complete (5);

2.6 割込みの抽象化

I-ID (Invoke-Invoked) 関係は，本稿で導入される新概念で，CP と CDP 間の基本的関係である．これを中心に割込みの基本構造と，OS の基本動作を明らかにする．

2.6.1 割込みと I-ID 関係

I-ID 関係は，特定の状態の対をもつ CP と CDP の間の関係をいう．この状態の対は，autonomous と suspend である．CP が suspend なら，これと I-ID 関係にある CDP は必ず autonomous であり，逆に CP が autonomous なら対応する CDP の状態は必ず suspend である（**Fig. 3** 参照）．

I-ID 関係の生成は，**enter** 文で行なう．CP が，特定の CDP を指定して **enter** CDP; を実行すると，この CP と CDP の間に I-ID 関係が生成される．このとき，**enter** 文を実行したプロセス（今の場合は CP）は suspend 状態となり，逆に **enter** 文で指定し

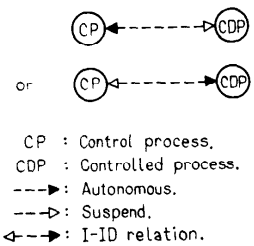

Fig. 3 I-ID (Invoke-Invoked relation)

[*4] **current** は組み込み関数で，自身の process designator を値としてかえす．

たプロセス（いまの場合は CDP）は autonomous 状態となる．換言すると，上記した例では，CP は実行中止に，CDP は実行再開されることになる．

I-ID 関係が新たに結ばれると，以前に他プロセスの間に結ばれていた I-ID 関係は解消される．

I-ID 関係の導入により，割込みの基本構造をつぎのように考えることができる．

割込みが発生して制御が特定の番地に移行し，ここから割込み処理プログラムの実行がはじまることは，割込み処理を行なう CP が，割込み発生と同時に suspend から autonomous になり動作を再開する一方，これと I-ID 関係にある CDP が autonomous から suspend になり実行が中断されることである．これは，I-ID 関係の反転として抽象化される．

I-ID 関係の反転とは，この関係にある2つのプロセスの間で，状態の交換が行なわれることをいう．すなわち，suspent プロセスは autonomous に，逆に autonomous プロセスは suspend になる（**Fig. 4** 参照）．このことから，割込みの発生とは，I-ID 関係が反転することで，割込み信号は，I-ID 関係の反転信号のことと理解できる．

割込みの発生は，**2.6.3** で説明する **signal** 文で行なう．

2.6.2 割込みと事象の発生

計算機システムに特徴的な割込みを，事象を用いて記述しがたい理由を考察する（本節は，外部割込みを対象に考察を進めるが，内部割込み（割出し）についての特殊性は **2.6.4** でふれる．）．

2.5 に述べたが，事象に関係するプロセスは，事象を知らせるものと事象の発生を知りその対応処置をとるプロセスの合計2個であり，これが基本であった．しかし割込みは，割込み信号を送るプロセス，割込み信号を受けて走り出すプロセス，それに割込み信号により実行が中断される（割込まれる）プロセスの合計3個が同時に関係する．後2者は I-ID 関係を結んでいるプロセスである．

既存の離散事象形シミュレーション言語は，計算機

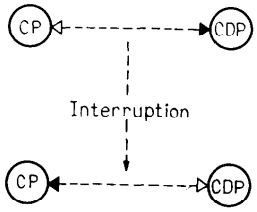

Fig. 4 The inversion of I-D relation

システムに特徴的な割込みを意識して設計されていない．すなわち，事象と割込みに同時に関係するプロセス数の相違が意識されていない．それゆえ，ハードウェアとソフトウェアの接点で両者のインターフェイスの役割をもつ割込み機構を，事象を頼りにして記述せざるを得ず，これが計算機システムの記述を著しく困難にしていた．CSDL では，事象を割込みの相違を明らかにする過程で I-ID 関係を導入し，計算機システムに典形的な割込み機構の記述を透明化した．

割込みの基本構造は以上のようであるが，現実の割込み機構はさらに複雑である．そのため次節 **2.6.3** で，さらに I-ID 関係の拡張を行なう．

2.6.3 条件付き I-ID 関係

割込み機構を正確に記述するためには，その複雑性を正当に反映した概念が必要である．現実の割込み機構のもつ複雑さとして，以下の3点をあげることができる．第1に，割込み要因が多数あり，これらのいくつかが同時に生起する可能性のあること，第2に各割込み要因ごとに優先順位が決められ，多数の割込み要因が同時に発生した場合，優先順に1つが選択され他は一時的にペンディングとなること，第3に，プログラムで特定の割込み要因を指定して割込みの抑止が可能，すなわち割込みマスクの制御が可能のことである．この複雑な機構は，I-ID 関係の反転を条件付きにすることで容易に抽象化される．

ここに，条件付き I-ID 関係とは，指定の条件が満たされてはじめて反転が生ずる I-ID 関係のことをいう．反転が生じた直後の I-ID 関係の再反転は無条件に禁止されるが，これは，割込み発生直後に割込み禁止モードになることに対応している．

条件付き I-ID 関係の生成は，**enter until** 文で行なう．**until** 以下は，割込み許容要因の指定である．たとえば，割込み要因 power error, channel #1 などを許容したいときは

enter CDP **until** (power error, channel #1,…);

と書く．

割込み要因の定義は組み込まれており，つぎのようである．

record class int (**boolean** flag, mask, lock bit, **reference** (process) owner, **lavel** interrupt handler, **message reference** (interrupt status) pointer);

message タイプは，文献 19) に導入されているもので，1つの割込み要因に対し，複数箇所から同時に

割込み信号と情報が送られてきたとき，これらを自動的にスタックしておくことを示す．割込み処理ルーチンの所在は interrupt handler で指示される．

割込み要因の生成はつぎのようにする．たとえば

```
reference (int) array itable〔1 : n〕;
    itable〔1〕 defined power error
        : =int (false, false, false, current,
            power error handler, empty);
```

この配列を利用して，割込み要因の優先順にもとづく選別と，指定の割込み処理ルーチンへの飛び越しは組み込み関数の **drive** (itable) を用いる．**drive** は，itable を順にスキャンし，割込み要因ごとに mask と flag の論理積をとり，最初にこれが **true** となる要因 (itable の配列順序が優先順位に対応) に書き込まれている割込み処理ルーチンのアドレスを関数の値としてかえす．割込み要因の選別と割込み処理ルーチンへの飛び越しは，**go to drive** (itable) と書くだけでよい．

enter 文をまとめると

⑧ **enter** 〈process designator〉 **until** 〈interrupt designator〉;

割込み信号および情報の送信は，**signal** 文で行なう．**signal** 文は，I-ID 関係の反転に用いる．

⑨ **signal** 〈interrupt designator〉 〈interrupt status〉

なお，割込みに関する **enter until** 文，および **signal** 文は，それぞれ，事象に関する **wait until** 文および **activate** 文が対応していることを注意しておく (**Fig. 5** 参照)．

2.6.4 外部割込みと内部割込み

計算機システムに発生する多数の割込みは，外部割込みと内部割込み（割出しともいう）に分類される．

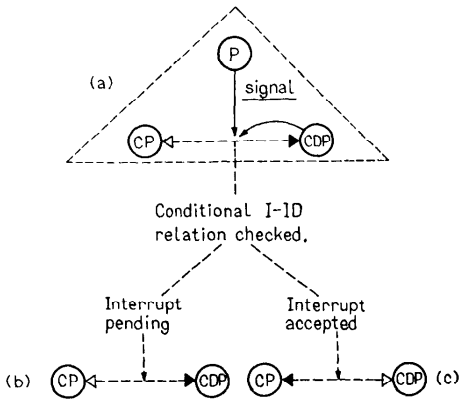

Fig. 5　Conditional I-ID relation.

割込まれたプロセスと，割込み要因の論理的結合度の強さから両者の相違を検討する立場もある[20]．本節では，これとやや異なる視点から，外部割込みと内部割込みの相違を検討してみる．

外部割込みの例として，チャネルからの入出力終了割込み，他の中央処理装置からの信号，計時装置の時間超過による割込みなどをあげることができる．これらはいずれも，割込み信号を発生するプロセスが金物として別個に存在しており，このプロセスと，I-ID 関係を結んでいる2つのプロセスと合わせて，割込みに3個のプロセスが関係する (**2.6.2**, Fig 5 (a) 参照)．したがって，外部割込みに関しては，**2.6.2** での割込みの抽象化の議論がそのまま適用できる．これに対し，内部割込みは若干性質を異にする．

内部割込みの例として，スーパバイザコール (SVC)，不当命令コード，特権命令の実行や，演算結果のオーバフロー，不当アドレス実行による割込みがある．これらを詳細に観察すると **2.6.2** に説明した割込みに同時に関係するプロセスの3者関係の縮退したものであることが知れる．すなわち，内部割込みは，割込み信号を出すプロセスと，割込みによって実行の中断されるプロセスが同一の場合である．内部割込みは，自身と I-ID 関係にあるプロセスと割込み要因を指定して **signal** 文を実行した場合に相当する．したがって，この場合，内部割込みに関係するプロセス数は2個で，事象のそれと同数となるが，両者の相違はつぎの点にある．自身と I-ID 関係にあるプロセスと割込み要因を指定して **signal** 文を実行したプロセスは，割込みマスクとの照合に合格すると，直ちに割込まれ suspend 状態になり実行が中断されてしまうのに，**activate** 文で事象を知らせるプロセスはこうした中断がなく，以後の動作が必ず保証される．I-ID 関係の反転はこうした現象を一括表現したものである．

2.7 再起動点の制御

2.7.1 再起動点制御の分類

再起動点の制御は，あるプロセスの実行再開番地を制御することをいう．割込みなどで実行を中断された仕事を継続するため，以前に動作の中断された番地のつぎから実行を再開する場合が，一般的なプロセスの再起動の例である．これに対し，PL/I の ON コンディションの指定するルーチンを実行する場合などは，割込み発生後に，実行再開番地がこのルーチンに移行してしまう場合である．

現実の OS の制御プログラムは，こうした複雑な再

起動点制御を行なわなければならず，この制御を容易に行なえる機能，概念が必要である．

再起動点制御の種類をつぎの3つに大別できる．

① 動作中断点のつぎから実行再開．

② 動作中断時に実行していた処理手続きの範囲内で実行再開番地を決定．

③ 実行再開番地が他の処理手続きに移行．

①は，**activate** 文および **from** なしの **start** および **enter** 文で起動する場合であり，②は **from** 付きの **start, enter** 文を用いる．③は **2.7.2** に説明する **give call** と **give return** 文を用いる．

2.7.2 give call と give return

実行再開番地が他の処理手続きに移る再起動点制御は，**give call** 文，**give return** 文で行なう．

プロセス A, B を考えたとき，プロセス A がプロセス B に **give call** することは，A が B に **call** 文をパスすること，すなわち，プロセス B に対してパスした **call** 文を実行させることを意味する．プロセス B が **execute** 状態になるとパスされた **call** 文の指定した処理手続きに制御がうつる．**give call** の対象となった B のコール・スタックの深さは1増す．

give return は **give call** の逆，すなわち **return** をパスされたプロセスは，あたかも自分が **return** 文を実行したがごとき状態になる．コール・スタックの深さは1減る．

⑩ **give call** ⟨process designator⟩ ⟨procedure id⟩ ⟨actual parameter⟩

⑪ **give return** ⟨process designator⟩

プロセス B に overflow という名の処理手続きを実行させる場合には，つぎのようにする．

```
give call B overflow (condition);
enter B until (……);
```

give call の概念は，MIT の MULTICS の interprocess communication の初期の構想に使われていたが，その後，別の形で具体化されたようである[17),19)]．**give return** は見当らない．

3. その他の CSDL 文

CSDL は SIMULA, SOL を参考にし，ALGOL 60 のスーパセットとして設計されている．2章にふれなかった CSDL 文のおもなものを概略説明する．

Ⓐ facility 制御 **lock**, **unlock**．これはプロセス相互間での racing を防止するために必要なもので，Dijkstra の P, V operation[5)] もあるが，GPSS の **seize, release** と同じ **lock, unlock** を採用する[4)]．

Ⓑ 統計データの収集とシミュレーションの進行制御 SIMULA の仕様をとりいれる．

Ⓒ Remote accessing **consider when**, Remote accessing は他プロセスの保有しているデータ (local data) にアクセスすることである．**consider** 文で **record class** のタイプを調べてから，**when** 文以下で，そのレコードのフィールドにアクセスする．文献9)参照．なお，データだけからなる **activity** の body が **record class** の定義と考えることもできる．

Ⓓ その他の ALGOL 60 に定義された文．

4. おわりに

CSDL は，計算機システムに含まれている種々の概念を整理抽象化し新概念を積極的に導入した．しかし，データ構造，ストリング操作用として，さらに便利な言語仕様が必要である．前者については，APL[6)] 程度の set, membership の制御がほしい (ASP ほどの柔軟性はさしあたり必要ないであろう)．

本稿は，CSDL とかりに名付けた言語の仕様を設定するための基本的な考え方を示した．OS をはじめ，各種制御プログラムの記述の透明化が1つの目標であった．これらは，今後，具体的な素材，システムを記述する試みの中で洗練すべきものである．

高級言語による OS 記述の利点は，文献2), 8) に述べられている．ここでは，高級言語により記述されたものが，将来手直しの作業を行なううえで意外な威力を発揮することを指摘したい．たとえコンパイラのない場合でも高級言語で最初に一度記述を行なっておいてから，人がコンパイルした場合には，はじめからアセンブリ言語を用いて記述を進めた場合に比べ，プログラムの生産性とオブジェクト・コードの効率が上まわるとの報告もある[13)]．

CSDL で提案したシステム設計方式は，OS 記述と，シミュレーションによる効率測定を同時に行なえるので，いわゆるソフトウェアの危機を克服できる可能性もある．

最後に，CSDL で整理，抽象化された概念のうち，I-ID 関係は，多重割込みを積極的に使用したシステムの記述には不向きかもしれない．2プロセス間の関係として I-ID 関係をとらえるのでなく，この場合は，プロセスの集合の間の関係 (1：多プロセス) に拡張する必要があろうが，今後の検討が必要である．本稿で整理，抽象化された種々の概念が厳密に定義できれば，

OS を含む計算機システムの形式化の可能性もある．システム形式化の第1歩は，CSDL のごときシステム記述用言語の形式的定義をすることだと筆者らは考えている．この線に沿った検討を開始中である．

謝辞 研究の機会を与えられた森 英夫所長と野田克彦電子計算機部長，森 俊二図形処理研究室長，日ごろご指導いただいている相磯秀夫慶応大学教授に深謝する．特に渕一博主任研究官からは有益な示唆と助言をいただいた．ここに深く感謝する次第です．

参考文献

1) 相磯秀夫，渕一博他：ETSS 特集号，電試彙報，**32** (1968)．
2) Corbató, F. J.: PL/I as a tool for system programming, DATAMATION (May, 1969) pp. 68～76.
3) Dahl, O-J and Nygaard, K: SIMULA-an ALGOL based simulation language, CACM, **9**, 9 (1966), pp. 671～678.
4) Dennis, J. B. and Van Horn, E. C.: Programming semantics for multiprogrammed computations, CACM, 9, 3(1966), pp. 143～155.
5) Dijkstra, E. W.: Co-operating sequential processes, Programming Language, Academic Press (1968), pp. 43～112.
6) Dodd, G. G.: APL-A language for associative data handling in PL/I, FJCC (1966), pp. 110～117.
7) 渕一博，田中穂積，真子ゆり子，弓場敏嗣，古川康一：ETSS の解析その2，電試彙報，**34**, 2 (1970), pp. 151～168.
8) 浜田穂積，中田育男，霜田忠孝，小林正和：PL/IW によるシステムの開発，情報処理学会プログラミング・シンポジウム (1970), pp. D-1～8.
9) Hoare, C. A. R: Record handling, Programming Language, Academic Press (1968), pp. 291～347.
10) Iliffe, J. K.: Basic Machine Principles, Macdonald (1968).
11) Knuth, D. E. and Mcneley, J. L.: Sol-a symbolic language for general-purpose system simulation, IEEE, (Aug. 1964) pp. 401～408.
12) Markowitz, H. M. Hausner, B, and Karr, H. W: SIMSCRIPT —A simulation programming language, Rand Corp, (1963).
13) Neuman, P. G.: The role of motherhood in the popart of system programming, ACM 2nd Sympo. On OS Principles (Oct. 1969) pp. 13～18.
14) 野口健一郎：オペレーティング・システムの記述と設計法，情報処理学会 OS シンポジウム報告集，(Sept. 1970), pp. 84～102.
15) Parnas, D. L.: More on simulation languages and design methodology for computer systems, SJCC (1969), pp. 734～739.
16) Saltzer, J. H.: Traffic control in a multiplexed computer system, MAC-TR-30, M. I. T (June. 1966).
17) Salxen, J. H.: MULTICS SYSTEM, 電子協 (1967).
18) Scherr. A. L.: Analysis of time sharing system, MAC-TR-18, MIT (1965).
19) Spier, M. J. and Organick, E. I.: The multics interprocess communication facility. ACM 2nd. Symposium on OS principles (Oct. 1969), pp. 83～91.
20) 高橋秀俊，亀田壽夫：オペレーティング・システムの一構成法，情報処理，**11**, 1 (1970), pp. 20～31.
21) 田中穂積：計算機制御システムの記述，情報処理学会 OS シンポジウム報告集 (Sept. 1970), pp. 280～286.
22) Wirth, N. and Hoare, C. A. R.: A contribution to the development of ALGOL, CACM (June. 1966), pp. 413～431.