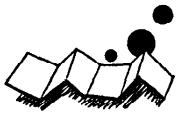
Vol. 27 No. 3　　情　報　処　理　　Mar. 1986

解　説

Ada の適用例(その 3)



仮想計算機システムの再設計†

細　川　　馨††

1. ソフトウェアの部品化

ソフトウェアの生産性を向上するために，数々の手法が提案されている[6),7)]．その一つとしてソフトウェアの部品化がある．部品化の手法そのものは，古くから採用されている．たとえば，自動車の製造は，特注の場合を除いて，すべて部品の組み合わせによって行われている．そこでは，部品の設計工程を省略することと，同じ部品を再利用することによって生産性の向上を図っている．

しかし，ソフトウェアの部品化の場合には，いくつかの問題点が考えられる．一つは，どのようなソフトウェア部品が有効に再利用できるかが明確になっていないことであり，もう一つは，再利用の定義自体が曖昧であることである．再利用という用語の誤った使用例としては，あるソフトウェアを移植した例を取り上げて再利用が可能であると言ったり，スタックをソフトウェアの中で何回も使用したので再利用したなどと言ったりする場合があげられる．

1.1 背　景

アセンブラ・プログラムの保守は，非常に困難であり，費用もかかる．特に，古いプログラムは，いくつもの更新が行われており，解読し難くなっている．マニュアルの更新が遅れがちのために，十分な情報をマニュアルから得ることもできない．このような環境でソフトウェアを保守するためには，まず古いソフトウェアを高級言語で書き直すことが必要である．しかし，アセンブラを単に書き直すことは難しく，ソフトウェアの構造を再設計しなくてはならない．

ソフトウェアの開発方法の一つに，抽象データ型[4)]を用いて構造化する方法がある．この方法の利点として，ソフトウェアの保守容易さなどもあげられるが，ソフトウェアの部品化が素直に行えることが最も重要である．つまり，ソフトウェアの設計に使われた抽象データ型がそのまま，部品としてなりえるのである．

1.2 研究目的

ソフトウェアの部品化の成功例として，Raytheon[5)]の仕事があげられる．そこでの成功の理由は，適用分野を業務用ソフトウェアに限ったことであった．この研究でも，適用分野を，OS のソフトウェアに限った．実際には，仮想計算機システムのスケジューラをもとにして，スケジューラの部品をつくった．部品の記述には，Ada[3)]の汎用パッケージを使用した．スケジューラの部品化は，二段階に分けて行われた．まず，スケジューラの機能を整理するために，Ada でその機能を記述し，次にこの仕様から，再利用を考えながら部品化を行った．

本稿では，アセンブラで書かれた古い OS のソフトウェアを仕様として，OS の再設計を行い，OS 部品の製作の可能性を検討した．これらの部品は，再利用ができるように工夫されている．また，抽象データ型を記述する際の，Ada の有効性についても確かめる．

2. 仮想計算機システムのスケジューラ

一般にスケジューラとは，OS のプロセスを管理するソフトウェアのことである．実計算機上での実行を終えたプロセスの状態管理や実行可能なプロセスの検出などがその主な機能である．仮想計算機システム[2)]のスケジューラも同様な機能を持っていて仮想計算機の管理を行っている．

スケジューラの実行は，ディスパッチャというプログラムと深く係わっている．ディスパッチャの働きは，実計算機に仮想計算機を割り当てることである．これによって，仮想計算機の，実行は行われる．機能レベルで考えると，ディスパッチャは，スケジューラから仮想計算機を受け取り，それを実計算機に割り当てる．仮想計算機が実行を終えるとディスパッチャは，その仮想計算機を実計算機から取りはずし，スケジューラに戻す．

このようなディスパッチャとの関係から分かるよう

† A Redesign of the Virtual Machine Operating System by Kaoru HOSOKAWA (IBM Japan Science Institute).

†† 日本アイ・ビー・エム(株)サイエンス・インスティチュート

にスケジューラの働きは，仮想計算機の受け渡しである．スケジューラの働きを，もう少し詳しく説明すると，ディスパッチャに仮想計算機を渡す際，任意の仮想計算機を選択して渡すのではなく，実行に最適な仮想計算機を選んでいる．最適な仮想計算機の選択は，スケジューリング・アルゴリズム[1]によって定義されている．

仮想計算機システムのスケジューラは，フィードバック方式のスケジューリング・アルゴリズムを採用している．このフィードバック・アルゴリズムは，過去の仮想計算機の状態を考慮してスケジューリングを行っている．過去の仮想計算機の状態とは，以前この仮想計算機がスケジューラに管理されていたときの状態(たとえば，実行可能な計算機であったかなど)を示している．

実際にスケジューラのプログラムを解読すると，このフィードバック・アルゴリズムは，三つのキューによって実現されている．Q1, Q2, Q3 と呼ばれるこの三つのキューは，仮想計算機使用者の種類によって分けられている．Q1には，interactive と言われる端末との入出力操作が多い，計算機時間の必要性が少ない使用者が保管されている．逆に Q3 には，端末との入出力操作が少ない，計算機時間の必要性が多い，heavy と言われる使用者が保管されている．そして，Q2 には中間的な，compute bound と言われる使用者が保管されている．この三つのキューは，プライオリティ・キューであり，キューの要素である仮想計算機の優先順位に従って並んでいる．たとえば，実行を終えてディスパッチャから受け渡された仮想計算機でも，その優先順位によっては，キューの先頭に入る可能性もある．

スケジューラの働きをここで整理してみると，まず仮想計算機をディスパッチャなどから受け取った場合，その仮想計算機の過去の状態などを考慮して，状態の更新を行い，三つのキュー内のどのキューに保管するかを決定し，そのキューに仮想計算機を入れる．逆に，仮想計算機をディスパッチャに渡すときは，実行可能な計算機をQ1, Q2, Q3という順番で見つかるまで探し，その仮想計算機を取り出し，ディスパッチャに渡す．図-1にスケジューラとその環境を示す．ディスパッチャは，Put と Get を使ってスケジューラと仮想計算機のやりとりをする．スケジューラは，個々のキューにたいして Put と Get を使って仮想計算機のやり取りを行う．

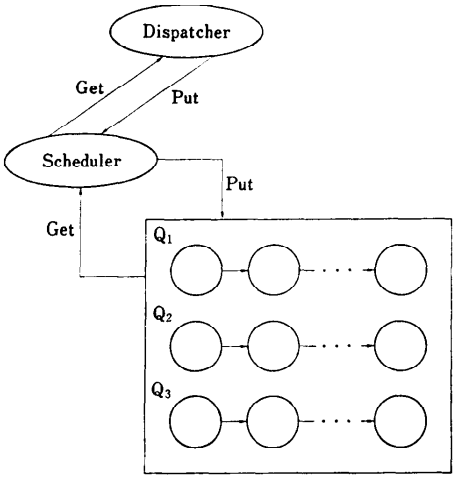


図-1 スケジューラとその環境

スケジューラは，上記のような個々の仮想計算機の管理のほかに，三つのキューの整理も行っている．そのことについて以下に説明する．仮想計算機のキューでの順番を決めるもう一つの要因として，メモリの大きさがある．実メモリの使用可能な領域は，動的に変化しているので，メモリの大きさによって以前は実行が不可能であった仮想計算機が今回は，実行可能になることがある．このような動的なメモリ変化に対応してキューも整理されていく．たとえば，メモリ不足のために実行不可能な Q1 の仮想計算機を実行可能にすることによって，Q1 での順番を更新することができる．

今までのスケジューラの説明は，若干抽象的である．現実には，効率の問題を重視するのでスケジューラの構造は，もう少し複雑である．たとえば，ディスパッチャとスケジューラ間の仮想計算機のやり取りは，実際には行われず，ディスパッチャは，スケジューラを呼ばずに直接キューをアクセスして仮想計算機を取り出している．また，実際には，実行可能な仮想計算機を保管するキューがあり，そこから直接実行する．

3. スケジューラのパッケージ

パッケージの設計で重要なことは，できるだけ基本的な機能だけを残して抽象化を行うことである．スケジューラの場合，パッケージが扱う対象としては，仮想計算機を考え，パッケージの操作としては，仮想計算機を取り出す操作と収める操作を考えた．また，

```
package vm_information is

  max_user : constant POSITIVE := 200 ;                    ……vm 使用者数

  type group is ( interactive, compute_bound, heavy ) ;    ……vm 使用者種類

  type vm is                                               ……vm
  record
    priority         : NATURAL ;
    user             : group ;
    candidate        : BOOLEAN ;
    forced           : BOOLEAN ;
    is_runnable      : BOOLEAN ;
    was_runnable     : BOOLEAN ;
    end_of_slice     : BOOLEAN ;
    end_of_candidate : BOOLEAN ;
    good_page_usage  : BOOLEAN ;
    assured          : BOOLEAN ;
  end record ;

end vm_information ;
```

```
with vm_information ;

use vm_information ;

package vm_queue_manager is

   type vm_queue is private ;                              ……vm_queue の宣言

   function empty( q : vm_queue ) return BOOLEAN ;         ……空か？
   function full( q : vm_queue ) return BOOLEAN ;          ……いっぱいか？

   procedure get( q : in out vm_queue ; a_vm : out vm ) ;  ……vm を取り出す.
   procedure put( q : in out vm_queue ; a_vm : vm ) ;      ……vm を入れる.

private

   type vector is array ( 1 .. max_user ) of vm ;

   type vm_queue is record
                      size  : NATURAL := 0 ;
                      space : vector ;
                    end record ;

end vm_queue_manager ;
```

(a)

```
with vm_information, vm_queue_manager ;

use vm_information, vm_queue_manager ;

package vm_scheduler_manager is

  type vm_scheduler is private ;                                  ……vm_queue の宣言

  function empty( s : vm_scheduler ) return BOOLEAN ;             ……空か?
  function full( s : vm_scheduler ) return BOOLEAN ;              ……いっぱいか?

  procedure get( s : in out vm_scheduler ; a_vm : out vm ) ;      ……vm を取り出す.
  procedure put( s : in out vm_scheduler ; a_vm : vm ) ;          ……vm を入れる.

private

  type vm_scheduler is record
                         size        : NATURAL := 0 ;
                         runnable_q  : vm_queue ;
                         eligible_q1 : vm_queue ;
                         eligible_q2 : vm_queue ;
                         eligible_q3 : vm_queue ;
                       end record ;

end vm_scheduler_manager ;
```

(b)

プログラム 1: vm_scheduler_manager パッケージ仕様

補助操作として，仮想計算機の収納区域が空かいっぱいか判断する関数も考えた．パッケージの仕様部分は，プログラム1のようになる．パッケージ名は，vm scheduler manager で，scheduler という型，empty, full という関数と get, put と言う手続きが宣言されている．図-2にスケジューラの操作の型を示した．scheduler は密閉型なので，その内部構造は，

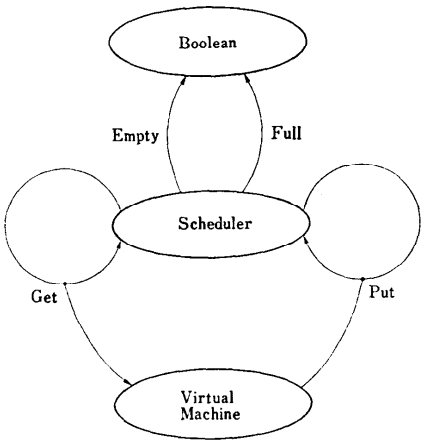


図-2 スケジューラの操作の型

使用者からは直接アクセスすることができない．パッケージによって宣言された関数と手続きによってのみアクセスできる．このパッケージの使用例としてディスパッチャの一部分を記述したのがプログラム2である．

プログラム2は，check_and_get という手続きを定義している．with 文によって，vm_scheduler_manager を使うことを宣言している．パッケージの対象及び操作を使用する場合，そのパッケージ名を付け足して修飾を行う．しかし，パッケージ名が長い場合などは，use 節を使ってパッケージを定義すると，修飾を省略することができる．たとえば vm_scheduler_manager.s_cheduler と書かずに単に，scheduler と書ける．check_nd_get は，s という scheduler に対して空かどうかをチェックし，空であればなにもしないが，空でなければ仮想計算機を s から一つ取り出す．

vm_scheduler_manager の実現の一部を記述したのがプログラム3である．効率を考慮して，runnable_q と eligible_q をもとにして，仮想計算機を管理している．実行可能な仮想計算機は，runnable_q に置かれ，その他は eligible_q に置かれる．

このパッケージは，再利用するのが難しい．たとえ

```
with vm_information, vm_scheduler_manager ;

use vm_information, vm_scheduler_manager ;

procedure check_and_get( s : in out vm_scheduler ; a_vm : out vm ) is
begin
  if not empty( s ) then                                    ……s が空でなければ
    get( s, a_vm ) ;                                        ……vm を取り出す.
  end if ;
end check_and_get ;
```

プログラム 2: check_and_get

```
with vm_information ;

use vm_information ;

package memory_information is

  function memory_available( a_vm : vm ) return BOOLEAN ;      ……充分メモリがあるか?

end memory_information ;

with vm_information, vm_queue_manager, memory_information ;

use vm_information, vm_queue_manager, memory_information ;

package body vm_scheduler_manager is

  function empty( s : vm_scheduler ) return BOOLEAN is
  begin
    return empty( s.runnable_q ) ;
  end empty ;

  function full( s : vm_scheduler ) return BOOLEAN is
  begin
    return s.size = max_user ;
  end full ;

  procedure get( s : in out vm_scheduler ; a_vm : out vm ) is
  begin
    get( s.runnable_q, a_vm ) ;                             ……s の runnable-q から
    s.size := s.size - 1 ;                                  ……vm を取り出す.
  end get ;

 procedure update( a_vm : in out vm ) is
 begin
   if a_vm.candidate and a_vm.user = interactive and a_vm.forced and  ……vm の現在の
      a_vm.is_runnable and a_vm.was_runnable then                     ……状態によって
     if a_vm.end_of_slice and not a_vm.end_of_candidate and           ……更新する.
        a_vm.good_page_usage then
       a_vm.user := compute_bound ;
     elsif not a_vm.assured then
       a_vm.candidate := FALSE ;
     end if ;
   else
     null ; -- other situations
   end if ;
 end update ;
```

(a)

```
procedure shuffle( s : in out vm_scheduler ) is

  procedure try_promote( q1, q2 : in out vm_queue ) is
    a_vm : vm ;
  begin
    loop
      get( q1, a_vm ) ;                                  ……vm を取り出し
      if memory_available( a_vm ) then                   ……充分メモリがあれば
        a_vm.candidate := TRUE ;
        put( q2, a_vm ) ;                                ……runnable-q に入れ,
      else                                               ……そうでなければ, vm を
        put( q1, a_vm ) ;                                ……もとの場所に戻す.
        exit ;
      end if ;
    end loop ;
  end try_promote ;

begin
  try_promote( s.eligible_q1, s.runnable_q ) ;           ……各 eligible-q から実行候補を
  try_promote( s.eligible_q2, s.runnable_q ) ;           ……探し出し, runnable-q に移す.
  try_promote( s.eligible_q3, s.runnable_q ) ;
end shuffle ;


  procedure put( s : in out vm_scheduler ; a_vm : vm ) is
    temp_vm : vm := a_vm ;
  begin
    update( temp_vm ) ;                                  ……vm の更新
    if temp_vm.candidate then                            ……vm が実行候補であればsの
      put( s.runnable_q, temp_vm ) ;                     ……runnable-q に入れる.
    else                                                 ……そうでなければ
      case temp_vm.user is
        when interactive   => put( s.eligible_q1, temp_vm ) ;   ……s の eligible-q のどれか
        when compute_bound => put( s.eligible_q2, temp_vm ) ;   ……に入れる.
        when heavy         => put( s.eligible_q3, temp_vm ) ;
      end case ;
    end if ;
    s.size := s.size + 1 ;
    shuffle( s ) ;                                       ……s の整理
  end put ;

end vm_scheduler_manager ;
```

(b)

プログラム 3: vm_scheduler_manager パッケージ実現の一部

ば, 仮想計算機の管理アルゴリズムがパッケージの内部で記述されているため, 新たなアルゴリズムを導入する場合, パッケージを書き直さなくてはならない. また, このパッケージが扱っている対象は, 仮想計算機に固定されているため, たとえばプロセスなどの対象を扱うパッケージを記述する場合にも, 書き直さなくてはならない. 対象を書き直すときは, 管理アルゴリズムも関係しているので同時に改良しなくてはならない.

再利用が可能なスケジューラは, パッケージの内部を書き直さずに使えなければならない.

4. スケジューラの汎用パッケージ

Ada の汎用パッケージを使うことによって, パッ

```
generic

  max_size : POSITIVE ;                                          ……サイズ

  type element is private ;                                      ……要素

  with function "<" ( e1, e2 : element ) return BOOLEAN ;        ……順位関数

package pool_manager is

  type pool is private ;

  function empty( p : pool ) return BOOLEAN ;                    ……空か?
  function full( p : pool ) return BOOLEAN ;                     ……いっぱいか?

  procedure get( p : in out pool ; e : out element ) ;           ……要素を取り出す.
  procedure put( p : in out pool ; e : out element ) ;           ……要素を入れる.

private

  type vector is array ( 1 .. max_size ) of element ;

  type pool is record
                 size  : NATURAL := 0 ;
                 space : vector ;
               end record ;

end pool_manager ;
```

プログラム 4: pool-manager パッケージの仕様

```
with vm_manager, pool_manager ;

use vm_manager ;

package vm_pool_manager is new pool_manager( max_user, vm, "<" ) ;
      ……max-user のサイズを持ち, vm を要素とし, "<" によってその
      ……要素を並べる新しい pool-manager (vm-pool-manager) の生成.
```

プログラム 5: vm-pool-manager の生成

ケージのパラメタ化が可能になる. この機能は, 手続きなどのパラメタ化と同じで, パラメタの値によって新しいパッケージを作ることを可能にしている. たとえば, スタックの汎用パッケージを定義して, そのパッケージの扱う対象をパラメタ化する. この汎用パッケージに整数をパラメタの値として与えると, 整数スタックのパッケージを生成することができる.

汎用パッケージでソフトウェア部品を記述することによって, ソフトウェア部品の使いやすさが改良された. パッケージで記述された部品がそのままで使用できない場合は, そのパッケージの内部を書き直さなければならないが, 汎用パッケージで記述することによって書き直しの必要性が縮小された.

汎用パッケージで記述された部品の利用度は, そのパッケージのパラメタの種類によって左右される. パラメタの種類を決定する適当な規則はない. これは, パッケージを設計するときに, その対象と操作を決める規則がないことと類似している. 現時点では, パッケージ設計者の経験に頼るしかない.

スケジューラ・パッケージを生成するための汎用

パッケージとしてプールというパッケージを考えた．プールの仕様部分はプログラム4に定義してある．プールの機能としては，任意の対象を収めたり，取り出したりすることと，プール自体の収納状況を判定することである．プールのパラメタとして，対象の最大収納量，対象自体とさらに収納されている対象の取り出しの際の順番が定義されている．対象の最大収納量のパラメタ化によって，プールの大きさを自由自在に管理することが可能である．たとえば，対象を10個収納できるプールも1000個収納できるプールも簡単に生成できる．対象自体をパラメタ化することによって，任意の対象を取り扱うプールを生成することができる．したがって，仮想計算機用のプールやプロセス用のプールを簡単に生成することができる．最後のパラメタは，二つの対象に順番を定める述語である．この述語のパラメタ化によって，収納されている対象の順番を任意に決めることができる．プールの内部では，この順番に対象が保管されていて，対象を取り出す操作は，その順番における一番最初の対象を取り出す．したがって，この順番の指定によってどの対象を取り出すかが決定される．

プールの汎用パッケージから仮想計算機のスケジューラを生成するためには，プールの大きさを指定し，対象自体を仮想計算機にして，対象の順番を決定する述語にスケジューラの管理アルゴリズムを指定する．Adaでは，プログラム5のように記述して，スケジューラのもとになるvm_pool_managerを生成する．任意の二つの仮想計算機の順番を定める関数として，<を定義した．順番は，仮想計算機の使用者の種類，実行の可能性，そして，優先順位で判定される．vm_poo_managerは，pool_managerにプールの大きさとしてmax_user，対象をvm，そして順番判定の関数として<を指定して，生成された．

パッケージのパラメタ化によって，再利用が簡単になったと同時に，スケジューリング・アルゴリズムの局所化も行われた．<は，順番の判定だけを行い，そのほかにはなにも行わない．局所化によって，保守性も向上した．

```
package vm_manager is

  max_user : constant POSITIVE := 200 ;                                  ……vm 使用者数

  type vm is private ;

  function "<" ( vm1, vm2 : vm ) return BOOLEAN ;                        ……vm 順位関数

  procedure update( a_vm : in out vm ) ;                                 ……vm 更新
  procedure try_promote( a_vm : in out vm ; done : out BOOLEAN ) ;       ……実行可能な vm の進級

private

  type group is ( interactive, compute_bound, heavy ) ;                  ……vm 使用者種類

  type vm is                                                             ……vm
  record
    priority         : NATURAL ;
    user             : group ;
    candidate        : BOOLEAN ;
    forced           : BOOLEAN ;
    is_runnable      : BOOLEAN ;
    was_runnable     : BOOLEAN ;
    end_of_slice     : BOOLEAN ;
    end_of_candidate : BOOLEAN ;
    good_page_usage  : BOOLEAN ;
    assured          : BOOLEAN ;
  end record ;

end vm_manager ;
```

(a)

```
package body vm_manager is

  function "<" ( vm1, vm2 : vm ) return BOOLEAN is
  begin
    if    group'POS( vm1.user ) < group'POS( vm2.user ) then  ……vm の個々の
      return TRUE ;                                          ……情報によって
    elsif group'POS( vm1.user ) > group'POS( vm2.user ) then  ……順位を決める.
      return FALSE ;
    elsif vm1.candidate and not vm2.candidate then
      return TRUE ;
    elsif not vm1.candidate and vm2.candidate then
      return FALSE ;
    elsif vm1.priority < vm2.priority then
      return TRUE ;
    else
      return FALSE ;
    end if ;
  end "<" ;

  procedure update( a_vm : in out vm ) is

  begin
    if a_vm.candidate and a_vm.user = interactive and a_vm.forced and  ……vm の個々の
       a_vm.is_runnable and a_vm.was_runnable then                     ……情報によって
      if a_vm.end_of_slice and not a_vm.end_of_candidate and           ……更新する.
         a_vm.good_page_usage then
        a_vm.user := compute_bound ;
      elsif not a_vm.assured then
        a_vm.candidate := FALSE ;
      end if ;
    else
      null ; -- other situations
    end if ;
  end update ;

  function memory_available( a_vm : vm ) return BOOLEAN is separate ;  ……vm が実行
                                                                       ……に必要なメモリがあるか?
  procedure try_promote( a_vm : in out vm ; done : out BOOLEAN ) is
  begin
    if a_vm.candidate then
      done := TRUE ;
    else
      if memory_available( a_vm ) then                                 ……実行可能な vm の
        a_vm.candidate := TRUE ;                                       ……候補をメモリ状況に
        done := TRUE ;                                                 ……よって決める.
      else
        done := FALSE ;
      end if ;
    end if ;
  end try_promote ;

end vm_manager ;
```

(b)

プログラム 6: vm_manager パッケージ

vm_manager は，vm を隠蔽するパッケージである．それは，プログラム6に記述されている．vm を密閉型と定義してあるのでこのパッケージの使用者からは，その中身は見えない．操作は，すべて定義されているものであり，これによって，誤まった操作を行うことを防いでいる．

vm_pool_manager だけでは，まだ仮想計算機のスケジューラとして不十分である．仮想計算機を受け取った際に，その状態を更新することと，仮想計算機が置かれている環境の変化への対応を行わなければならない．これらが記述されているのが，プログラム7である．put が実行されたとき，仮想計算機の状態は，update により更新される．そして，環境の変化への対応については，get を行う際に，try_promote が対処する．この手続きは，実行が不可能な仮想計算機を新しい環境で再検討し，実行可能にできれば，する．たとえば，入出力待ちで実行ができなかった仮想計算機を入出力状況のチェックを行って実行可能にする．

汎用パッケージから生成されたスケジューラは，内部データ構造としてキューを一つ扱っている．これに対して普通のパッケージとして記述されたスケジューラは，内部データ構造にキューを三つ扱っている．このように，機能の局所化だけではなく，データ構造の簡単化も汎用パッケージを使うことによって行われた．

5. 汎用パッケージの問題点

パッケージのパラメタ化により記述できる汎用パッケージの有効性を述べてきたが，その記述性に関していくつかの問題点がある．一つは，パラメタ数が固定されていることである．もう一つは，パッケージ型がないことで，それによってパッケージの部分型化が不可能になっている．これらの問題点を順を追って説明する．最後に効率と差し替えについて述べる．

5.1 汎用パッケージのパラメタ数

汎用パッケージのパラメタ数が，固定されているために，記述し難い場合がある．ここでは，仮想計算機のスケジューラを記述したときに直面した問題を例に取って説明する．

仮想計算機のスケジューラの基本になるアルゴリズムは，任意の仮想計算機に対してその状態を確認し，情報を更新する．状態の確認をプログラムで記述すると，if 文の数の多さとそのネストの深さによって，非常に読み難いものとなってしまう．（プログラム6のくなどは，その例である．）このため，汎用パッケージを使ってスケジューラを記述しても，順番を指定する述語の内部は，相変らず読み難い．この読み難さを解消するために，ディシジョン・テーブルを使うことにした．ディシジョン・テーブルとは，if 文を二次元的な表で表したものと考えることができる．ディシ

```
with vm_manager, vm_pool_manager ;

use vm_manager, vm_pool_manager ;

package vm_scheduler_manager is

  type vm_scheduler is private ;

  function empty( s : vm_scheduler ) return BOOLEAN ;        ……空か？
  function full( s : vm_scheduler )  return BOOLEAN ;        ……いっぱいか？

  procedure get( s : in out vm_scheduler ; a_vm : out vm ) ; ……vm を取り出す．
  procedure put( s : in out vm_scheduler ; a_vm :     vm ) ; ……vm を入れる．

  no_candidate : exception ;                                 ……実行可能な vm がない場合の例外

private

  type vm_scheduler is record                                ……vm_scheduler を
                         p : pool ;                          ……pool によって定義する．
                       end record ;

end vm_scheduler_manager ;
```

(a)

```
with vm_manager, vm_pool_manager ;

use vm_manager, vm_pool_manager ;

package body vm_scheduler_manager is

  function empty( s : vm_scheduler ) return BOOLEAN is
  begin
    return empty( s.p ) ;                          ……(pool に対して) 空か?
  end empty ;

  function full( s : vm_scheduler ) return BOOLEAN is
  begin
    return full( s.p ) ;                           ……(pool に対して) いっぱいか?
  end full ;

  procedure get( s : in out vm_scheduler ; a_vm : out vm ) is
    temp_vm : vm ;
    done : BOOLEAN ;
  begin
    get( s.p, temp_vm ) ;                          ……pool から vm を取り出し,
    try_promote( temp_vm, done ) ;                 ……実行可能な vm であれば
    if done then
      a_vm := temp_vm ;                            ……それを渡し, そうでなければ
    else
      put( s.p, temp_vm ) ;
      raise no_candidate ;                         ……例外
    end if ;
  end get ;

  procedure put( s : in out vm_scheduler ; a_vm : vm ) is
    temp_vm : vm := a_vm ;
  begin
    update( temp_vm ) ;                            ……vm の情報の更新
    put( s.p, temp_vm ) ;                          ……pool にそれを入れる.
  end put ;

end vm_scheduler_manager ;
```

(b)

プログラム 7: vm_scheduler_manager パッケージの実現

ジョン・テーブルには, どの条件が満たされていればどの行動を実行するかが記述されている. つまり, 条件の集合, 実行する行動の集合及びその二つの集合の関係が明確に記述されている.

このディシジョン・テーブルの汎用パッケージを作った. プログラム 8 がその例である. パラメタとしては, 条件の集合, 実行する行動の集合及び, 条件と行動が取り扱う対象が用意されている. したがって, 仮想計算機の基本アルゴリズムを記述するためには, 対象を仮想計算機にし, 個々の条件と行動を指定する. この汎用パッケージを使って状態確認のプログラムの一部分を記述すると, プログラムの読みやすさが向上するのは明らかであろう.

しかし, この汎用パッケージにも, 欠点がある. それは, 条件の数が固定されていることで, 可変にするようには, 記述できない. プログラム 8 の場合, 条件数が 4 つと行動数が 5 つに固定されている. つまり, 条件が 4 つあるディシジョン・テーブルの汎用パッケージから, 条件が 10 指定できるディシジョン・テーブルが生成できないのである. 必要であれば, そのつど汎用パッケージを定義しなくてはならない. この問題の解決策の一案として, 50 の条件を指定できる汎用パ

ッケージを定義し，その汎用パッケージのパラメタとして条件数の指定を可能にすることが考えられる．このようにすれば，この汎用パッケージから最高50までの条件を判断するディシジョン・テーブル・パッケージが生成できることになる．しかし，50の条件を必要としない場合には，コード領域の無駄になる．また，51以上の条件を必要とするディシジョン・テーブル・パッケージを生成することはできない．

5.2 パッケージの型化

抽象データ型をAdaのパッケージで記述することが可能である．しかし，Adaではパッケージを高階の対象（たとえば，関数の結果）として扱えない．このことは，プログラムを設計する場合に影響を及ぼす．パッケージは，すべてグローバルに定義されてしまうので，プログラムの構造が本来の自然な構造からはずれてしまう可能性がある．

さらに，パッケージの配列とかパッケージを要素とした複雑なデータ構造をつくることが不可能である．このことの不便さは，明らかであろう．たとえば，バッファの配列を定義したい場合，バッファ名をすべて明示的に，記述せざるをえない．

パッケージのパッケージなども定義できない．たとえば，プールの汎用パッケージから，バッファ・プールのパッケージを生成することができない．

スケジューラを記述するにあたって，パッケージの対象として常に，それを型として定義した．パッケージの操作として，関数や手続きを定義する場合は必ず，この型をパラメタとする．これによって，抽象データ型のインスタンスをいくつも生成することが可能である．(型として定義しない場合は，パッケージにつきひとつのインスタンスしか生成できない．) また，関数の結果として扱うことも可能である．このように，型宣言することによって，いくつかの問題は，解決されたが，部分型化については，まだ問題が残されている．

5.3 パッケージの部分型化

型の部分型化が，Adaなどの言語では可能である．たとえば，整数の型の部分型として自然数を定義することができる．パッケージによって抽象データ型が記述可能である．部分型の概念をパッケージに適用することによって，記述能力が向上するのは明らかであろう．たとえば，人という型の部分型として男または女が定義できるようになる．部分型によってインヘリタンスを組み込むことが可能になる．しかし，Adaには，パッケージの部分型化を行う機能がないので，拡張機能についての提案がいくつかある[8]．

5.4 パッケージの実行効率と差し替え

プログラムの抽象度とそのプログラムの実行効率は，反比例の関係にある．Adaなどで書かれた抽象度の高いプログラムは，実行効率が悪く，アセンブラで書かれた抽象度の低いプログラムは，実行効率が良い．汎用スケジューラ・パッケージの場合，扱うキューをひとつにしたことによって，抽象化はされているが効率が悪くなっている．

```
--------------------------------------------------------------------------
--                                                                      --
--       Decision Table Manager                                         --
--            (4 Conditions and 5 Actions Version)                      --
--                                                                      --
--------------------------------------------------------------------------
--                                                                      --
--       Conditions |                                                   --
--       -----------+---+---+---+---+---+                               --
--       1          |   |   |   |   |   |                               --
--       -----------+---+---+---+---+---+                               --
--       2          |   |   |   |   |   |                               --
--       -----------+---+---+---+---+---+                               --
--       3          |   |   |   |   |   |                               --
--       -----------+---+---+---+---+---+                               --
--       4          |   |   |   |   |   |                               --
--       ==================================                             --
--       Actions    | 1 | 2 | 3 | 4 | 5 |                               --
--                                                                      --
--------------------------------------------------------------------------
```

(a)

```
generic

  type Element is private ;                                          ……対象

  with function Condition1 ( E : Element ) return Boolean ;          ……
  with function Condition2 ( E : Element ) return Boolean ;          ……
  with function Condition3 ( E : Element ) return Boolean ;          ……  条件の指定
  with function Condition4 ( E : Element ) return Boolean ;          ……

  with procedure Action1 ( E : in out Element ) ;                    ……
  with procedure Action2 ( E : in out Element ) ;                    ……
  with procedure Action3 ( E : in out Element ) ;                    ……  行動の指定
  with procedure Action4 ( E : in out Element ) ;                    ……
  with procedure Action5 ( E : in out Element ) ;                    ……

package DecisionTable4_5Manager is

  type Answer is ( No, Yes, DontCare ) ;

  type Condition is ( Condition1, Condition2, Condition3, Condition4 ) ;

  type Action is ( Act1, Act2, Act3, Act4,  Act5 ) ;

  type DecisionTable is private ;

  procedure Set ( DT : in out DecisionTable ;                        ……デシジョンテーブルのセット・アップ
                  Ans1, Ans2, Ans3, Ans4 : Answer ;
                  Act : Action ) ;

  procedure Go ( DT : DecisionTable ; E : in out Element ) ;         ……デシジョンテーブルの実行

private

  subtype FirmAnswer is Answer range No .. Yes ;

  type DecisionTable is array ( Condition, Action ) of Answer ;

end DecisionTable4_5Manager ;
```

(b)

プログラム 8: デシジョン・テーブルパッケージの仕様

プログラムをパッケージで記述することによって，そのインタフェースの整理ができた．しかし，このパッケージを既存のプログラムに組み込むとき，インタフェースを既存のプログラムに合うように書き替えなくてはならない．また，プログラムのコール文すべてをパッケージ用に書き替えなくてはならない．

現時点では，効率と差し替えの問題についての回答はない．ただし，パッケージをプログラムの仕様として考えて使うことによって，抽象化の利点である，理解しやすさを有効に使うことができる．たとえば，保守時に，アセンブラを解読するのではなく，パッケージを理解すればよい．また，開発でも，パッケージからそのプログラムの機能を読み取り，そこから効率を考えながら開発する．

謝辞 仮想計算機システムのスケジューラの解読にあたって，早稲田理工学部情報科学研究教育センターの『ソフトウェア開発における仕様記述あるいは検証の実用性に関する研究』の研究部会のメンバの協力を得たので，ここで感謝する．

参 考 文 献

1) Brinch Hansen, P.: Operating System Princi-

ples, Prentice-Hall (1973).
2) Creasy, R. J.: The Origin of the VL/370 Time-Sharing System, IBM J. Res. Dev., Vol 25, No. 5 (Sep. 1981).
3) Department of Defense, USA: Ada Programming Language, ANSI/MIL-STD-1815A-1983 (Jan. 1983).
4) Guttag, J. V.: Abstract Data Types and the Development of Data Structures, Comm. ACM 20, pp. 396-404 (Jun. 1977).
5) Lanergan, R. G. and Grasso, C. A.: Software Engineering with Reusable Designs and Code, IEEE Trans. Softw. Eng., Vol. SE-10, No. 5, pp. 498-501 (Sep. 1984).
6) 情報処理：プログラム設計技法（1），Vol. 25, No. 9 (Sep. 1984).
7) 情報処理：プログラム設計技法（2），Vol. 25, No. 11 (Nov. 1984).
8) 細川 馨：Ada Packageのサブタイピングについての一考察，プログラミング言語 2-3 (Sep. 1985).

(昭和60年11月14日受付)