Ada の並列機能のプログラム変換†

石畑 清††

プログラミング言語 Ada の諸並列機能に対するプログラム変換法を示す．変換の方向は，ある機能をそれより基本的と考えられるもので置き換えることである．変換の主な目的は，Ada の並列処理機能の本質を追及することである．また，これらの変換は，公平な非決定性などの原始プログラム上での実現やプログラムの効率向上にも有効である．

1. はじめに

プログラミング言語 Ada[9] は，並列処理を記述するための機能として，メッセージ交換方式の一種であるランデブー方式[6] を採用している．この方式は，それ以前に提案されたメッセージ交換方式の機能である CSP[5] や DP[1] と共通する点が多いが，いくつかの違った点も含んでいる．このため，実際的なプログラムの記述能力の面でかなりの違いがある．

従来，Ada とそれ以前に提案された言語の並列プログラムの記述能力の差を調べる研究がいくつか行われてきた[11]．その方法としては，共通の例題を各言語で記述して，その際に生ずる問題点を数え上げるものが中心である．このほか，Ada の並列処理機能を実現する立場に立って，その効率を論じた研究も多い[4),8]．

しかし，現在でも Ada と他の言語の並列処理機能の得失が十分理解できる状態になったとは言い難い．Ada では記述しにくい問題があることが指摘され[7),10]，言語仕様を拡張してこれに対処しようという提案もいくつかある[2),3)] が，個別の提案にとどまり，並列処理機能の本質をついたものとは言えない．言語仕様を拡張するのは簡単だが，本質についての理解なしに性急に拡張して，混沌とした言語を得ることの愚は言語の設計においてしばしば見るところである．

ここでは，プログラム変換を使って Ada の並列処理機能の本質を追及する．Ada の並列処理はランデブーによる値の受渡しが中心になるが，その周辺に各種の機能が付随している．これらの機能を使ったプログラムをそれらを使わないものに書き換えられることを示すのがこの論文の主眼である．具体的には，select 文を取り除く，1タスクあたり1エントリに限る，などの5種類の変換を示す．

これによって，各機能の存在意義の評価に新しい視点が得られると考える．また，データと制御の使い分けや単一タスクへの集中と複数タスクへの分散の対比など，並列プログラムにおける各種のプログラミングスタイルの比較の場としても有効である．変換によって言語仕様を論ずる手法は，1970 年代初期に構造化制御文の記述能力に関して行われた研究と同様である．

さらに，変換の意義として，Ada では記述の難しい問題の一般的な記述法を提供することと，効率向上の手段を提供することが挙げられる．Ada では，複数のエントリに対する呼出しを公平に受け付けること，逆に各エントリに優先度を設定してそれに従って受け付けること，また資源管理の問題などを記述することが難しい．これらの問題は，いずれもここに示した変換を使うことによって解くことができる．また，処理系によっては，特定の機能の効率が他のものと比べて悪いことがありうる．この場合，ここで示す変換によってその機能を別の機能に置き換えられれば，効率の向上が期待できる．

2. Ada の並列処理機能

Ada では並列実行の単位をタスクと呼ぶ．各タスクは，いくつかのエントリを持つことができる．エントリは，他のタスクから呼び出されてサービスを提供する窓口のようなものである．タスクは，他のタスクのエントリを呼び出すことができる．一方，エントリを持つタスクは，プログラム中の任意の場所でそのエントリに対する accept 文を実行することができる．accept 文を実行した時点でそのエントリがすでに呼び出されていれば，その呼出しの処理を行う．処理の内容の記述法は手続きの本体と同様である．そのエントリに対する呼出しがなければ，呼び出されるまで待

† Program Transformations of Ada Parallel Programs by KIYOSHI ISHIHATA (Department of Information Science, Faculty of Science, University of Tokyo).

†† 東京大学理学部情報科学科

機状態に入る．

逆にエントリを呼び出す側から見ると，そのタスクが accept 文で待機状態にある時に呼び出せば直ちに処理が始まるが，そうでなければ accept 文が実行されるまで待機状態におかれる．このように互いに相手を待って，双方がそろった時点で初めて処理が始まる機構をランデブー方式と言う．

各エントリには待ち行列が付属している．一つのエントリを呼び出したタスクは，順次その待ち行列に入れられて，先頭から処理される．

select 文は，いくつかのエントリに対する accept 文を並べて，そのうちのどれが呼ばれても対応できるようにしたものである．どれも呼ばれていない時は待機状態に入り，呼ばれたエントリがあればそれに対する処理を行う．複数のエントリが呼ばれている時はどれを選ぶか定義されていない．

なお，Ada ではタスク間の共有変数を置くことができるが，これはタスクの動作を記述するための手段として勧められるものではない．この論文でもこれについては考慮しない．

Ada の並列処理記述方式の特徴をまとめると次のようになる．

1) 通信が同期式に行われる．送り手と受け手の双方がそろって初めてデータの受渡しが行われる．メールボックス方式のように受取りを確認しないまま送り手が先に進むということはない．これは CSP と同様である．

2) ランデブーの単位が大きい．CSP と違って1回の通信で一方向にデータを渡すだけでなく，手続き呼出しのようにパラメタを通して双方向のデータの受渡しや渡されたデータの加工ができる．これはプログラミングに便利であるが記述能力には大した影響を与えない．なお，データの送り手と受け手のいずれを呼び手，いずれを受付け手にするかを選ぶことができる点は重要である．

3) 通信が非対称である．呼ぶ側はエントリを持っているタスクの名前を指定するが，受付けの側では相手のタスクの名前を指定できない．これは CSP と違う点である．これを利用して，不特定の相手に対してサービスを提供するタスクを書くことができる．しかし，複雑なプロトコルの通信を行う時には，このことが障害となって困難を生じることがある．

4) 受付けの制御を制御の流れによって行う．どのエントリが受付け可能で，どのエントリが受付け不可能かは，そのエントリに対する accept 文を実行するかしないかで決まる．言い換えれば，エントリの受付けの順序は，そのタスクの実行する文の順序によって決まる．これは CSP と同様であり，共有変数方式や多くの並列オブジェクト指向言語と決定的に違う点である．これらの言語ではどれを受け付けてどれを受け付けないかは，それぞれに与えられた条件式の真偽によって決まる．つまり，そのタスクの状態を表す変数の値によって決まる．

一般にものごとを制御するのに，プログラムの制御の流れによる方法とデータの値による方法がある．これらは互いに交換可能であるが，プログラムの記述性の面では大きな違いとなって現れる．

なお，エントリを受け付けるか否かを決めるにあたって，そのエントリ呼出しのパラメタを調べてから決めることは許されないという点に注意すべきである．

3. 変 換 法

Ada の並列処理機能に対する5種類の変換法を順次示す．変換の一般形を示すのは煩雑なので，例を使って説明することにする．一般の場合の変換法を類推することは容易である．

変換にあたっては，プログラムの働きを保存しなければならない．しかし，並列プログラムの場合はプログラムの可能な動作順序が複数ありうるので，プログラムの等価性の定義をあらかじめ与えておく必要がある．ここでは，元のプログラムで可能な実行系列の集合と変換後のプログラムで可能な実行系列の集合が等しいことという定義を採用する．それぞれの実行系列が原理的に可能でありさえすればよいという比較的緩い条件である．実際の処理系で走らせた時に特定の実行順序をとる確率が変わり，その結果として違った動作をするように観測されることがあっても，この定義には違反しない．実行系列には同期操作に関係する文を含むが，変換の対象としている文を例外にするのはもちろんである．

変換の対象としては，正しいプログラムだけを考える．正しくないプログラム，たとえばデッドロックするプログラムについては変換前と後が等価でなくてもかまわないものとする．等価性を示すには，例外，パラメタの受渡しの方法，タスクの異常終了などについても配慮しなければならないが，これらについては必要に応じて個別に説明する．

なお，変換にあたっては，Ada の規則を尊重するが，不自然と思われる制限事項については無視することがある．

3.1 when 条件の除去

select 文の when 条件は，エントリを受け付けるかどうかを論理式の値で制御する．条件を簡潔に書く効果があるが，記述能力の点で本質的ではない．ここでは，条件の値の組み合わせごとに一つずつ select 文を用意することによって，when 条件を取り除く変換を示す．

select 文は，最初に when 条件をすべて評価して，真になった選択肢だけを選択の対象にする．つまり，タスク間通信の観点からは，条件が真となる選択肢だけを含んだ select 文を実行するのと等価である．変換は，まず条件式の真偽値のすべての組み合わせに対して，選択の対象にする選択肢だけを含んだ select 文を用意する．そして，when 条件をすべて評価してから，if 文によってその結果に対応する select 文を選んで実行すればよい．

元のプログラムの select 文が次のとおりだとする．

```
select
  when P => accept A do...;
or
  when Q => accept B do...;
end select;
```

この時，変換結果は次のとおりである．

```
Pval := P;
Qval := Q;
if Pval and Qval then
  select
    accept A do...;
  or
    accept B do...;
  end select;
elsif Pval then
  accept A do...;
elsif Qval then
  accept B do...;
else
  raise PROGRAM_ERROR;
end if;
```

最初に，論理式 P と Q を評価した結果を変数 Pval と Qval に代入する．両方が真なら最初の select 文を実行する．すでに条件が真であることがわかっているので，この select 文に when 条件は不要である．一方だけが真ならそれぞれの accept 文を実行し，両方が偽なら例外を発生させる．これは，選択肢の条件がすべて偽なら例外が発生するという Ada の規則に合わせたものである．

select 文の他の形式の選択肢，すなわち delay 選択肢，terminate 選択肢，else 部がある場合も変換の方法は同じである．

when 条件は，Ada のタスク間通信では例外的にデータの値で通信を制御する機能である．これを制御の流れで制御するように変形したのが上に示した変換にあたる．

3.2 accept 文の統合

Ada は，制御の流れで通信を制御する方針をとっている関係で，accept 文や select 文がタスクの中に分散して現れる．場合によっては，同じエントリに対する accept 文が複数個使われる場合もある．これらをすべて1箇所にまとめて，select 文の形にするのが次に示す変換である．ただし，後で述べるように accept 文が入れ子になっている場合は除く．

この変換では，タスクの現在実行中の場所を表す変数，すなわちロケーションカウンタを導入する．

元のプログラムが次のとおりだとする．

```
task body X is
  ...
begin
  ...
  accept A do...end A;
  ...
  accept B do...end B;
  ...
  select
    when P => accept A do...end A;
  or
    when Q => accept B do...end B;
  end select;
  ...
end X;
```

エントリ A にも B にも二つの accept 文があることに注意されたい（一つは単独，一つは select 文の中）．

最初に，プログラムを基本ブロックに分割して，それぞれに番号をふる．基本ブロックへの分割の方法は，データフロー解析やプログラム最適化で使われる手法を流用すればよい．ただし，accept 文や select

文のようにタスク間通信に関係する文は，それ単独で基本ブロックをなすと定める．これは，次にどのエントリを受付け可能な状態にあるかを，どの基本ブロックに属するかで表現するからである．

変数 LC を用意して，実行中の基本ブロックの番号を保持するようにした結果は次のようになる．

```
task body X is
  ...
begin
  LC :=1;
  ...
  LC :=2;
  accept A do...end A;
  LC :=3;
  ...
  LC :=4;
  accept B do...end B;
  LC :=5;
  ...
  LC :=6;
  select
    when P => accept A do...end A;
  or
    when Q => accept B do...end B;
  end select;
  LC :=7;
  ...
end X;
```

次に accept 文や select 文を一つの select 文にまとめる．変換後の select 文には，元のプログラムのどの位置でどのエントリを受付け可能かを忠実に反映するように when 条件をつける．すなわち，元のプログラムの accept 文または select 文の位置に対応するかどうかを変数 LC の値を使って調べる．この例の場合の変換結果は次のようになる．

```
task body X is
  ...
begin
  ...
  loop
    select
      when LC=2 or (LC=6 and P) =>
        accept A do
          if LC=2 then
            ...
          else
            ...
          end if;
        end A;
    or
      when LC=4 or (LC=6 and Q) =>
        accept B do...end B;
    end select;
    ...
  end loop;
end X;
```

変換結果は，select 文とその他の文を交互に実行する loop 文となる．元のプログラムで accept 文ないし select 文を実行する状態の時に，この loop 文の先頭に制御を移す．loop 文の残りの部分は，次の accept 文までに実行する文である．

繰返しや条件分岐などの制御構造が accept 文や select 文を含んでいる場合，制御文の中身を分解しなければならないが，変数 LC の値が元のプログラムの位置を反映するようにすれば問題はない．文の入れ子の関係が変わるので，宣言の有効範囲に問題が生じるが，適当に宣言を変えて等価なプログラムを得ることは容易である．

変換の結果，同じエントリに対する accept 文が1箇所に統合される．したがって，各 accept 文では，現在実行しているのが元のプログラムのどの accept 文にあたるかを変数 LC の値から判断して，それに対応する文を実行する．

元のプログラムに select 文があって，その中に delay 選択肢，terminate 選択肢，else 部のいずれかがある場合には，少し複雑になる．delay 選択肢または else 部がある場合は，変換結果にも delay 選択肢を設ければよい．遅延時間や when 条件は，元のプログラムの実行経路に合わせて計算する．else 部の場合の遅延時間は0である．元のプログラムに terminate 選択肢がある場合は，変換結果にも terminate 選択肢を入れておかなければならない．この場合，元のプログラムの単独の accept 文や terminate 選択肢を含まない select 文にも，この terminate 選択肢が適用されることになるが，デッドロックに陥るプログラムをそうでないプログラムに変換する恐れがあるだけで，正しいプログラムに問題を生じることはない．

ここで示した変換は，accept 文の中に別の accept

文が入れ子状になっているものまで統合することはできない．入れ子の accept 文の働きを考えれば，これは当然である．入れ子になった accept 文は，3者以上のランデブーを実現するもので，普通の accept 文と同列には扱えない．一つの accept 文の中にある accept 文同士を上の変換でまとめることはできるので，変換の目的は十分達成できる．

3.3 select 文の除去

select 文は，実際的な並列処理を記述する上で欠かせない機能であるが，必須のものではない．単純な accept 文だけを使って同じ機能を持つプログラムに書き換えることが可能である．ここでは，select 文を取り除いて accept 文だけに置き換える変換を示す．

この変換では，エントリ呼出しをしたタスクとエントリの名前を管理するための表を用いる．また，NOTIFIER と呼ぶタスクと AGENT と呼ぶ仲介タスクを導入する．

この変換は，terminate 選択肢がある場合は適用できない．select 文以外に terminate 選択肢に対応する機能がないため，これはやむを得ない．

select 文は，複数のエントリに対する呼出しを処理するのが基本的な役割である．これを除くには，どのエントリに対する呼出しが来ても受け付けられるようにしておかなければならない．accept 文には，特定のエントリに対する呼出しを待つという機能しかないので，accept 文で実現するためには，どのエントリが呼ばれるかをあらかじめ知らせておかなければならない．そこで，呼びたいエントリの名前を伝えるためのランデブーを最初に行い，次に目的のエントリを呼び出すという2段階のプロトコルを採用することにする．つまり，1回の通信を2回のランデブーに分けて行うことになる．

元のプログラム（タスク SERVER と呼ぶ）に次の select 文があったとする．

```
select
  when P1 => accept E1(...) do...end E1;
or
  ...
or
  when Pn => accept En(...) do...end En;
end select;
```

これを変換した結果は次のとおりである．

```
NOTIFIER. WAKE_UP;
WAITING := TRUE;
while NO_ACCEPTABLE_CALLS or
      WAITING loop
  accept SIGN_IN (R : REQUEST_NAME) do
    if R=NOTIFY then
      WAITING := FALSE;
    else
      INSERT_TABLE (R);
    end if;
  end SIGN_IN;
end loop;
case ENTRY_TO_BE_ACCEPTED is
  when R1 => accept E1(...) do...end E1;
  ...
  when Rn => accept En(...) do...end En;
end case;
```

タスク NOTIFIER は，次のようにエントリ WAKE_UP の呼出しを受けると，直ちに SIGN_IN を呼び返すタスクである．

```
task body NOTIFIER is
begin
  loop
    accept WAKE_UP;
    SERVER. SIGN_IN (NOTIFY);
  end loop;
end NOTIFIER;
```

エントリ SIGN_IN のパラメタは，次に呼び出すエントリの名前を列挙型によってコード化したものである．タスク SERVER のエントリを呼び出す時は，まずエントリ SIGN_IN を呼び出して，対象のエントリの名前を伝えなければならない．

SERVER は，NOTIFIER にメッセージを送ってから，受付け可能なエントリに対する要求が現れるまで，エントリ SIGN_IN の受付けを繰り返す．SIGN_IN に対する accept 文では，エントリ呼出しの要求を管理するための表にこの名前を挿入する．while 文の繰返し条件では，この表を調べて when 条件の値が真であるような選択肢のエントリに対する要求があるかどうかを調べる．受付け可能なエントリ呼出しがある場合は繰返しから抜け出して，実際の受付けに入る．ただし，NOTIFIER から SIGN_IN に対する呼出しが到着するまでは必ず待つ．受け付けるべきエントリを選ぶアルゴリズム (ENTRY_TO_BE_ACCEPTED) は任意である．どのエントリを受け付けるかがこの時点で決まるので，select 文を使わずに case 文で場合

分けして accept 文を実行すればよい.

タスク NOTIFIER は，全エントリをひととおり調べる時間を確保する役割を持つ．このタスクがないと，受付け可能な呼出しを片端から受け付けるだけになって，エントリの選択に判断を加える余地がなくなる.

元の select 文に delay 選択肢が含まれている時は，所定の時間が経過した時にこのタスクにエントリ呼出しを行うタスクを用意しておく．このタスクは NOTIFIER に delay 文を加えた形である．このタスクからのエントリ呼出しが他のエントリ呼出しより先に行われれば，delay 選択肢の文を実行する．else 部が含まれていた場合も同様である.

SERVER の中に，他にも accept 文や select 文がある場合は，それらも同様の形に変形して，この表を調べるようにすることが必要である．なお，3.2 節の変換結果を使うのであれば，この配慮は不要である.

この変換を行うには，ユーザタスクと SERVER の間にエントリ呼出しの仲介をするタスクを置くことが必要である．仲介タスクは，タスクのエントリごとに用意する．エントリ E*i* に対する仲介タスク AGENT*i* の内容は以下のとおりである.

```
task body AGENTi is
begin
  loop
    accept Ei (...) do
      SERVER. SIGN_IN (Ei);
      SERVER. Ei (...);
    end Ei;
  end loop;
end AGENTi;
```

ユーザタスクがこのタスクを通さずに直接サーバタスクのエントリを呼ぶことは許さない.

仲介タスクを使わないと，SIGN_IN を呼び出したタスクが本体のエントリを呼ぶ前に abort 文で強制終了させられた場合に，サーバタスクが accept 文から先に進めなくなるという問題がある．仲介タスクを間にはさめば，ユーザタスクが abort 文などによって終了しても，タスク SERVER にも仲介タスクにも悪影響は及ばない．このほか，仲介タスクを設けることには，呼出しタスクの管理を容易にする，複雑なプロトコルをユーザタスクから隠すなどの利点がある.

仲介タスクを設けると，時限エントリ呼出しや条件付きエントリ呼出しで呼び出した場合の動作が正しくなくなる．仲介タスクに受け付けられただけで，タイムアウトの機構が働かなくなるからである．これについては 3.4 節の変換で時限エントリ呼出しを取り除くことによって対処する．同じ議論は 3.5 節に示す変換にも適用される.

この変換では，どのエントリが呼ばれているかに関する情報を Ada の実行時システムに代わってプログラム自身で管理するように改めた．これによって，どのエントリの呼出しを先に受け付けるかの選択もプログラムの中で行うことができるようになる．一般に select 文がどのエントリ呼出しを先に受け付けるかは言語規約の上では規定されていないので，実用的なプログラムで細かな制御を必要とする場合に不都合を感ずることがあった．この変換は，この問題に対する解答にもなっている.

この変換によって，どのエントリを先に受け付けるかの選択が非決定的なものから決定的なものに変わるが，3 章の冒頭に述べた定義に従う限り，これは問題にならない．タスクのスケジューリングに不確定性があるため，可能な実行系列の集合は変化しない.

3.4 時限エントリ呼出しの除去

時限エントリ呼出しは，エントリを呼び出すが，一定時間内に受け付けられなければ呼出しを取り消すというものである．Ada のタスク間通信は，相手のタスクの状態によって動作を変える権利が受付け側にだけあって呼出し側にないのが原則だが，これと条件付きエントリ呼出しだけは例外である．条件付きエントリ呼出しは，遅れが 0 の時限エントリ呼出しに帰着できる．ここでは，時限エントリ呼出しを取り除いて，単なるエントリ呼出しだけに置き換える変換を示す．この際，新たなタスク三つが必要になる.

この変換のためには，呼出しを受け付ける側のタスクに 3.3 節に示した変換をあらかじめ加えておくことが必要である．時限エントリ呼出しをするタスクには，特に制約はない.

時限エントリ呼出しを除くために新たに三つの仲介タスクを用意する．これらを AGENT1, AGENT2, AGENT3 と呼ぶことにする．AGENT1 はユーザタスクから依頼を受けて，三つの仲介タスクの初期化を行い，経過時間を監視する．AGENT2 が中心となるタスクで，SERVER が呼出しを受け付けるのと AGENT1 からの時間切れの報告の両方を select 文で待つ．AGENT3 は，補助的なタスクである.

元のプログラムのユーザタスクに次の時限エントリ

呼出しがあったとする.

```
select
  SERVER. REQUEST (...);
or
  delay T;
end select;
```

これを変換した結果は次に示すとおりである（図1参照）.

```
ユーザタスク:
AGENT1. REQUEST (...);
AGENT2. RESULT (...);

AGENT1:
accept REQUEST (...) do...end REQUEST;
AGENT2. START (...);
delay T;
AGENT2. STOP;

AGENT2:
accept START (...) do...end START;
AGENT3. START;
select
  accept SERVICE (...) do...end SERVICE;
or
  accept STOP do...end STOP;
end select;
accept RESULT (...) do...end RESULT;

AGENT3:
accept START;
SERVER. SIGN_IN (..., AGENT2);
```

AGENT1 は，AGENT2 を起動してから，delay 文で休止する．起動の際には，ユーザタスクから受け取ったパラメタを渡す．delay 文で所定の時間が経過したら，AGENT2 に停止命令を送る．これが時限エントリ呼出しのタイムアウトにあたる．AGENT2 は SERVER からの呼出し（後述）と AGENT1 からのタイムアウトの知らせを select 文で待つ形をとる（前節の変換でこの select 文を取り除くこともできる）．どちらか先に来た方を受け付ければよい．いずれにせよ，実行が終わったら，ユーザタスクに結果を返すための accept 文を実行する．

AGENT3 はエントリを呼び出したいということを SERVER に伝える役目を果たす．AGENT2 が直接これを行うと所定の時間が過ぎても SERVER と交信できない恐れがある．これを避けるための仲介のタスクである．

SERVER は，3.3 節の変換の結果なので，各エントリの accept 文は，SIGN_IN を受け付けてから実行される．このことを利用して，ここではエントリ名に加えて呼び出したタスクを識別するパラメタを SIGN_IN に渡すことにする．つまり，accept 文を実行する時は，どのタスクから呼ばれているかがわかっている．この状態では，accept 文をエントリ呼出しに変えることができる．タスク名が見えるか見えないかを除けば，ランデブーによる待ち合わせの機構そのものは対称だからである．accept 文の中で参照できる大域変数の環境が異なるが，必要な変数をすべてパラメタで渡すことにすれば，問題はない．そこで，相手が時限エントリ呼出しである時に限って，逆に SERVER の側から呼出しをすることにする．AGENT3 からの SIGN_IN に対応する accept 文を実行することに決まったら，AGENT2 のエントリ SERVICE を呼び出す．

ユーザタスクは，AGENT1 にパラメタを与えて起動したら，AGENT2 から結果を受け取るまで待っていればよい．この結果には，エントリの出力パラメタのほかに，エントリを呼び出すことができたか，タイムアウトによって打ち切られたかを示す情報が含まれている．

三つの仲介タスクは，時限エントリ呼出しのたびに新たに作って，ランデブー終了後は消去する．タイム

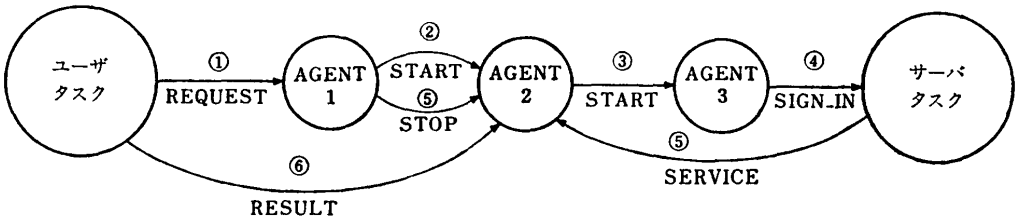

図 1 時限エントリ呼出しの除去
Fig. 1 Removal of timed entry calls.

アウトになった後に，SERVER が AGENT2 を呼び出すことがありうるが，この場合 SERVER には例外が発生するので，これを処理する機構を準備しておくことが必要である．

この変換によって待ち時間がどう変化するかを調べる．この論文では等価性の判定に実行の順序だけを考慮し，時間は無視している．一般にプログラム変換で時間まで保存することは不可能である．しかし，時限エントリ呼出しの場合は時間を無視できないので，ここではこの方針から1歩踏み出して，時間に関しても考察することにする．AGENT1 の REQUEST や AGENT 2 の START が受け付けられるまでに，これらのタスクをスケジュールするための時間がかかる．これによって実質的な待ち時間が増える．しかし，これは少なくとも理論的には問題とするに足りない．時限エントリ呼出しの実行にあたっては，待ち時間を計算してから，相手に呼出しをかけるという手順を踏む．時分割方式であれば，この手順の途中のどこでも，実行が中断されて他のタスクの実行に移る可能性がある．つまり，元の時限エントリ呼出しでも，文の実行開始からの時間という点では，正確な値を与えるという保証はない．変換後のプログラムと事情は同じである．

3.5 エントリの統合

タスクからタスクにエントリを通じて渡す情報には，エントリの名前とエントリのパラメタの二つがある．前者は accept 文の機構によって，受け付けるかどうかを判断する材料とすることができるが，後者は受け付けてからでないと参照できない．この違いを除けば，個々の情報をどちらを使って表現するかには自由度がある．そこで，パラメタだけで必要な情報を表現することを考える．ここでは，一つのタスクの持つ複数のエントリを一つにまとめる変換を示す．言い換えれば，一つのタスクあたり一つのエントリに限っても，プログラムを書けることを示す．

変換の対象とするのは，3.3 節の変換の結果である．したがって，次に呼び出すエントリの名前は，エントリ SIGN_IN によってすでにデータとして扱える形になっている．ここでは，便宜的に 3.2 節の変換によって accept 文を一つにまとめた形を扱うが，これは本質的ではない．

元のプログラム（以下，3.3 節の変換結果を元のプログラムという）には，SIGN_IN を除いて n 個のエントリがあるとする．最初に，これらをすべて統合したエントリを作る．このエントリは，元のエントリそれぞれが持っていたパラメタすべてを持つことになる．

これだけでは，どのエントリを受け付けられる状態にあるかを表現できないので，元のプログラムのエントリごとに仲介タスク AGENTi を一つ用意して，不要なエントリ呼出しをさえぎる役割を持たせる．SERVER は，特定のエントリを受け付けることが決まった時に，AGENT にそのことを知らせて，実際のエントリ呼出しを行わせる．他のエントリに対応する AGENT はエントリ呼出しを行わずに待っているので，受け付けられないエントリが呼び出されることはない．このほか，ユーザタスクと AGENT の間にもう一つの仲介タスク BUFFER を設ける．

以上で，SIGN_IN 以外のエントリを統合することができた．SIGN_IN とその他のエントリを統合するためには，SIGN_IN の呼出しであるか，その他のエントリの呼出しであるかを区別するパラメタを追加する．SIGN_IN の呼出しはいつでも受け付けてよいし，その他のエントリの呼出しは許可を与えた1個の呼出ししか存在しえないので，エントリ名による受付けの選択は不要である．

以上の考え方によって変換した結果を示す（図2参照）．

```
type UNIFY_TYPE is (SIGN_IN, EXECUTE);

task body SERVER is
  -- declaration of TABLE
 egin
  READY := 0;
  loop
     if READY = 0 then
     -- search acceptable entry
     -- set its number to READY
     case READY is
       when 0 =>
         null;
       when 1 =>
         AGENT_1. REQUEST_1 (...);
       ...
       when n =>
         AGENT_n. REQUEST_n (...);
     end case;
   end if;
```

```
    accept U (TYP: UNIFY_
        TYPE; N: ...; ...) do
      if TYP=SIGN_IN then
        INSERT (N, TABLE);
      else
        case N is
          when 1 =>...
          ...
          when n => ...
        end case;
        RETRIEVE (N, TABLE);
        READY := 0;
      end if;
    end U;
  end loop;
end SERVER;

task body AGENTi is
begin
  loop
    accept REQUESTi (...) do
      SERVER. U (SIGN-IN, i, ...);
      accept REQUESTi (...);
      SERVER. U (EXECUTE, i, ...);
    end REQUESTi;
  end loop;
end AGENTi;

task body BUFFERi is
begin
  loop
    accept REQUESTi (...) do
      AGENTi. REQUESTi (...);
    end REQUESTi;
  end loop;
end BUFFERi;
```

ユーザタスクの要求は，BUFFER，AGENT，サーバタスクの順に伝えられる．

サーバタスクは，3.3 節と同様にどのエントリに対する呼出し要求があったかを表を使って管理する．この中に受付け可能な呼出しがあれば，それを管理するAGENT タスクにそのことを伝える．このエントリに対する呼出しを受け付けたならば，この要求を表から取り除く．

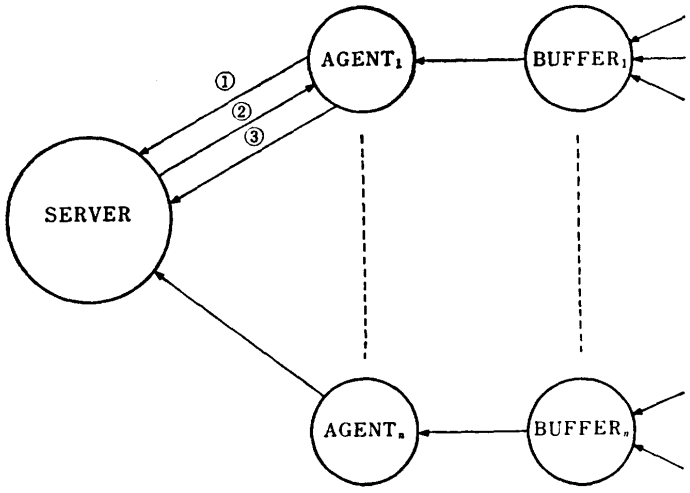

図 2 エントリの統合
Fig. 2 Unifying entries.

AGENT タスクは，ユーザタスクからの呼出し要求を受け取ったら，まずサーバタスクに SIGN_IN を要求する．そして，サーバタスクから呼出しの許可があるまで accept 文で待つ．許可が出てからエントリの本体を呼び出す．

BUFFER タスクは，AGENT タスクにユーザタスクからの要求を一つずつ伝える働きを持つ．このタスクがないと，AGENTi の内側の accept 文でサーバタスクからの許可と別のユーザタスクからの新しい要求とを区別できない．BUFFERi を導入すれば，一つのタスクの要求が満たされるまで，他のタスクの要求はこのタスクの入り口で待たされるので，この問題点は解消する．これは，特にプロトコルを設けずに，タスク間の位置関係を工夫するだけでも同期を制御できる場合があることを示す例になっている．

AGENT タスクの中には，同じエントリ REQUESTi に対する accept 文が入れ子になっている箇所がある．Ada の規則によれば，同じエントリに対する accept 文を入れ子にすることは許されない．しかし，この規定には明確な理由がないと思われるので，ここでは無視する．

3.3 節以降，エントリ呼出しを途中にいくつものタスクを置いて中継して行くという方式を多く使った．ランデブーの基本的な働きについては等価性を確かめてきた．ところが，パラメタの渡し方に問題が残っている．もし，パラメタを参照渡しで渡していると，結果が変わる恐れがある．

Ada では，単純型のパラメタはコピー渡しとする

ことが決まっているが，構造型のパラメタは，コピー渡しでも，参照渡しでもよいことになっている．どちらか一方の方式を仮定したプログラムは正しいプログラムとはみなされない．仲介タスクを置く方式はコピー渡しなら正しく動くので，問題はない．

4. まとめ

Adaの並列処理機能のいくつかを取り除いたサブセットにおいても，フルセットの言語仕様で書いたプログラムと等価なプログラムを書けることを示した．ここで示した変換は次のとおりである．

- select 文の when 条件を取り除くこと
- accept 文や select 文を一つの select 文にまとめること
- select 文を取り除くこと
- 時限エントリ呼出しを取り除くこと
- 1タスクあたり一つのエントリに限ること

これらの機能を取り除くにあたって，エントリで待っているタスクの状態をプログラム自身で管理すること，余分のタスクを用意することなどの代償を払ったが，待ち合わせの関係は完全に再現できている．

ここで示した五つの変換は必ずしも同時に適用できるものではない．これらの変換の間の関係を簡単に示しておく．

一つのプログラムに3.1節の変換と他の四つの変換の両方を適用することには意味がない．3.3節に示した方法でselect文を除けば，3.1節の変換は不要である．3.2節以降の変換は，同時に適用することができる．一つのエントリに対する複数のaccept文，select文，時限エントリ呼出し，一つのタスクの持つ複数のエントリを順次取り除いていって，最終的には各タスクが一つのエントリだけを持っている形に書き換えることが可能である．なお，3.4，3.5節の変換には，3.3節の変換が前提条件となる．

ここでは，いくつかの機能がなくても並列プログラムを記述できることを示したが，それが直ちに言語仕様として不要であるという結論に結び付かないことはもちろんである．記述性などの面にも配慮して，議論を進めることが今後の課題である．

参考文献

1) Brinch Hansen, P.: Distributed Processes: a Concurrent Programming Concept, *Comm. ACM*, Vol. 21, No. 11, pp. 934-941 (1978).
2) Frances, N. and Yemini, S. A.: Symmetric Intertask Communication, *ACM Trans. Prog. Lang. Syst.*, Vol. 7, pp. 622-636 (1985).
3) Gehani, N. H. and Cargill, T. A.: Concurrent Programming in the Ada Language: the Polling Bias, *Softw. Pract. Exper.*, Vol. 14, pp. 413-427 (1984).
4) Habermann, A. N. and Nassi, I. R.: Efficient Implementation of Ada Tasks, Carnegie-Mellon University, CMU-CS-80-103, p. 21 (1980).
5) Hoare, C. A. R.: Communicating Sequential Processes, *Comm. ACM*, Vol. 21, No. 8, pp. 666-677 (1978).
6) Ichbiah, J. D. et al.: Rationale for the Design of the Ada Programming Language, *SIGPLAN Notices*, Vol. 14, No. 6, Part B (1979).
7) 石畑 清，筧 捷彦：Adaの待ち合わせの標準形について，京都大学数理解析研究所講究録470, pp. 14-28 (1982).
8) Roberts, E.S., Evans, A. Jr. and Morgan, C.R.: Task Management in Ada—a Critical Evaluation for Real-time Multiprocessors, *Softw. Pract. Exper.*, Vol. 11, pp. 1019-1051 (1981).
9) U. S. Department of Defense: Reference Manual for the Ada Programming Language (1983).
10) Wellings, A. J., Keeffe, D. and Tomlinson, G. M.: A Problem with Ada and Resource Allocation, *Ada Letters*, Vol. 3, No. 4, pp. 112-124 (1984).
11) Welsh, J. and Lister, A.: A Comparative Study of Task Communication in Ada, *Softw. Pract. Exper.*, Vol. 11, pp. 257-290 (1981).

(昭和63年1月11日受付)
(昭和63年6月24日採録)

石畑 清（正会員）

昭和27年生．昭和49年東京大学理学部物理学科卒業．昭和52年同大学院博士課程中退．同年東京大学理学部情報科学科助手．プログラミング言語とその処理系，プログラミング環境，アルゴリズムなどに興味をもつ．著書「Adaプログラミング」（共著，岩波書店），「Pascalの標準化」（共著，共立出版）など．日本ソフトウェア科学会，ACM各会員．