ソフトウェア基礎論19－6
プログラミング言語 9－6
（1986. 12. 12）

Ada®タスクに一意的識別子を付ける方法

Assigning Unique Identifiers to Ada ® Tasks

程 京徳　　牛島 和夫
Jingde CHENG and Kazuo USHIJIMA

九州大学 工学部 情報工学科
Department of Computer Science and Communication Engineering, Kyushu University

あらまし　Ada並列プログラムのタスキング振る舞いをモニタする際に、被モニタプログラムの各タスクを一意的に識別するために、各タスクに一意的識別子を付けなければならない。更に、タスクとそのマスタとの間の依存関係を把握するために、各タスクの一意的識別子をそのマスタとの間の依存関係と共に実行モニタが管理するタスク名前表に登録しなければならない。本論文では、我々がAda並列プログラムの事象駆動型実行モニタを設計し開発する際に考案し実現した、プログラム変換によってAdaタスクに一意的識別子を付ける方法を述べる。本方法の主な特徴は、タスクとそのマスタとの間の依存関係を考慮したこと、被モニタプログラムの各タスクのセマンティクスをそのまま保存することなどである。

Abstract　In order to identify uniquely Ada tasks during monitoring the tasking behavior of Ada programs, it is necessary to assign internally a unique identifier to every task. This task identifier must be saved with the dependence relation between the task and its masters in the task name table managed by an execution monitor. In this paper, we present a program transformation approach for assigning unique identifiers to Ada tasks. It is implemented in an event-driven execution monitor which we are developing. The features of our approach are as follows : 1) The dependence relation between the task and its masters is considered; 2) The original semantics of all tasks of the monitored program is preserved.

1. まえがき

我々は、Ada[1]プログラミング支援環境[2),3)]を構築する一環として、Ada並列プログラムの事象駆動型実行モニタを設計し開発中である[4),5)]。これは、Ada並列プログラムのテスト、デバッグの手段を利用者に提供するソフトウェア・ツールの一つであり、Adaプログラミング支援環境の要求[2)]に応じてAda処理系の上で実現するものである。その機能には、実行中の被モニタプログラムについて、そのタスキング振る舞い（例えば、各タスクの状態とその状態変化や、各エントリの待ち行列の状態とその状態変化や、タスク間の依存関係や、タスク間におけるランデブーによる通信など）をモニタすること、タスク間でランデブーにより通信する際に起こるデッドロックを自動的に検出すること、タスキング振る舞いの履歴を保存すること、（被モニタプログラムの実行中または実行後）ユーザの求めに応じてタスキング振る舞いを報告することなどがある[4),5)]。

Ada並列プログラムのタスキング振る舞いを事象駆動によりモニタするとは、被モニタプログラムの実行中に、そのタスキングに関する事象の生起に応じて、タスキングに関する情報を収集することである。このように情報を収集するものを事象駆動型実行モニタ（以下、実行モニタと略する）と呼ぶ。

® Adaは、アメリカ政府 Ada Joint Program Office の登録商標である。

Ada並列プログラムのタスキング振る舞いをモニタするのであるから、実行モニタは、被モニタプログラムの各タスクを一意的に識別できなければならない。Ada並列プログラムのソース・テキスト中のタスク名前はプログラムの実行中に複数のタスクに対応できるので、タスク名前により各タスクを一意的に識別することはできない。従って、別の手段で各タスクに一意的識別子を付けなければならない。

本論文では、我々がAda並列プログラムの事象駆動型実行モニタを設計し開発する際に考案し実現した、プログラム変換によってAdaタスクに一意的識別子を付ける方法を述べる。本方法の主な特徴は、タスクとそのマスタとの間の依存関係を考慮したこと、被モニタプログラムの各タスクのセマンティクスをそのまま保存することなどである。

以下、2節では、Adaの並列処理機構タスキングについて簡単に説明する。なお、Adaのシンタクスとセマンティクスとの詳細については、参考文献1)を参照されたい。3節では、Ada並列プログラムのタスキング振る舞いをモニタする際に、Adaタスクの一意的識別子に対する要求を考察する。4節では、タスク間の依存関係について述べる。5節では、Adaタスクに一意的識別子を付ける方法を述べる。6節で、他の方法との比較などを考察する。

2. Adaの並列処理機構タスキング

Adaの並列処理機構をタスキングと呼び、その並列処理単位をタスク単位と呼ぶ[1)]。主プログラムとして使用する副プログラムは、ある環境タスク(主タスクとも呼ぶ)によって呼び出されたかのように動作する[1)]。主プログラムが主タスクの仮想的な本体の中で宣言されると考えればよい。

各タスク型(あるいは単一タスク)は、タスク仕様とタスク本体とからなるタスク単位によって定義される。タスク仕様は、タスク型とそのエントリ(もしあれば)を宣言し、そのタスク型のタスクと、それと同じまたは異なるタスク型の他のタスクとの間のインタフェースを定義する。タスク本体は、そのタスク型のタスク演算対象によって指されるタスクの実行を定義する。

タスクのエントリは、他のタスクから呼び出すことができる。タスクは、エントリに対応するaccept文を実行することによって、そのエントリの呼び出しを受付ける。同期は、エントリを呼び出したタスクとその呼び出しを受付けるタスクとの間の待ち合せ(ランデブーという)によって達成される。エントリにはパラメタを持つものがある。そのようなエントリに対するエントリ呼び出しとaccept文とがタスク間で値をやり取りする主要な手段である。

タスク本体の実行の最初の部分は、タスク演算対象の起動(activation)と呼ばれ、タスク本体の宣言部(もしあれば)の確立(elaboration)からなる。

各タスクは、少なくとも一つのマスタに依存する。マスタとは、タスク、現在実行中のブロック文又は副プログラム、もしくはライブラリ・パッケージの何れかである。割り当て子(allocator)の評価によって作られる演算対象又はそのような演算対象の下位要素であるタスク演算対象によって指されるタスクは、対応するアクセス型定義を確立したマスタに直接に依存する。その他のタスク演算対象によって指されるタスクは、実行によってそのタスク演算対象を作り出したマスタに直接に依存する。

以下では、タスクとは、あるタスク演算対象を指し、タスク仕様とタスク本体とは、あるタスク型の仕様と本体とをそれぞれ指す。

3. Adaタスクの一意的識別子

タスクの一意的識別子とは、Ada並列プログラムのタスキング振る舞いをモニタする際に、各タスクの起動から終了までの生存期にそのタスクを一意的に識別することができる「名前」である。

被モニタプログラムのタスクに一意的識別子を付けるために、適当なプログラム変換を行って、タスクの宣言に応じて一意的識別子を設定する変数を宣言し、その値をタスクの一意的識別子として用いる方法を取る。管理上の便利を図るために、一意的識別子を設定する変数をNATURAL型の演算対象とする。

以下、一意的識別子変数で一意的識別子を設定する変数を指し、一意的識別子で一意的識別子を設定する変数の値を指す。

Ada並列プログラムではタスクを動的に生成し消滅させることができる。また、各タスクの起動が並列に行われるので、各タスクに識別子を一意的に付けることは、各タスクの生成に応じて同期的に行わなければならない。

Ada並列プログラムのタスキング振る舞いをモニタするために、次の二つの場合にタスクを識別しなければならない。

1) タスクが他のタスクによって呼び出される場合、

2) タスク自身がその本体によって定義される実行をする場合。

1)の場合では、呼び出したタスクが呼び出されたタスクの一意的識別子を実行モニタに知らせなければならない。このような一意的識別子をタスクの外部識別子という。

2)の場合では、タスクがその本体を実行する際に自身の一意的識別子を実行モニタに知らせなければな

らない。このような一意的識別子をタスクの内部識別子という。

タスクは、その宣言が可視であれば、エントリ呼び出し文によって指名されうる。従って、ソース・テキスト上(即ち、静的)では、タスクの宣言が可視である場所においてタスクの外部識別子変数も可視でなければならない。

タスクは、その宣言が確立すると(まだ本体は起動されていないかもしれない)、他のタスクによって呼び出されうる[6]。従って、実行中(即ち、動的)では、タスクが他のタスクによって呼び出されうる時においてその外部識別子の値が設定されていなければならない。

タスクの外部識別子に対する要求を満足するために、タスク宣言の直後に外部識別子を設定する変数を宣言し、初期値設定の形式で外部識別子を設定すればよい(図1参照)。タスクが他のタスクによって呼び出される際に、呼び出したタスクが呼び出されたタスクの外部識別子を実行モニタに対するエントリ呼び出しのパラメタとして実行モニタに渡すことができる(図1参照)。

```
procedure P is
  :
  task type TT is
    entry E;
  end TT;
  T : TT;
  T__ID : NATURAL := GET__EXTERNAL__TASK__ID;
  :
begin
  :
  Monitor . ENTRY__CALL ( …, T__ID, "E" );
  T . E;
  :
end P;
```

図1　タスクの外部識別子

ソース・テキスト上では、タスク本体の内部(宣言部、例外処理も含む)においてタスクの内部識別子変数が可視でなければならない。

タスクは、その起動中に、関数副プログラムの呼び出しにより演算対象の初期値を設定する際に他のタスクと通信することができる。従って、実行中では、タスクが起動すると、その内部識別子の値が設定されていなければならない。

タスクの内部識別子に対する要求を満足するために、タスク本体の宣言部の始めに内部識別子を設定する変数を宣言し、初期値設定の形式で内部識別子を設定すればよい。タスクがタスキング振る舞いに関する動作をする際に、自身の内部識別子を実行モニタに対するエントリ呼び出しのパラメタとして実行モニタに渡すことができる(図2参照)。

タスクは、その本体の中で副プログラムを呼び出すことがある。もし呼び出された副プログラムがその本体の宣言部で宣言されていれば、特に問題はない。タスク本体の宣言部の始めに宣言している内部識別子変数は、Adaの可視性規則によって、そのような副プログラムの本体の中から見えるからである(図3参照)。

```
task body T is
  TASK__ID : NATURAL
             := GET__INTERNAL__TASK__ID;
  :
begin
  :
  Monitor . ACCEPTABLE ( TASK__ID, … );
  accept E;
  :
end T;
```

図2　タスクの内部識別子

```
task body T is
  TASK__ID : NATURAL
             := GET__INTERNAL__TASK__ID;
  :
  procedure P is
    :
  begin
    :
    Monitor . WILL__BE__DELAYED ( TASK__ID, … );
    delay E;
    :
  end P;
  :
begin  …  P;  …  end T;
```

図3　タスクの内部識別子が見える副プログラム呼び出し

もし呼び出された副プログラムがライブラリ単位であるか、あるいはライブラリ・パッケージの可視部で宣言されていれば、その副プログラムの本体がタスク内部識別子を知ることができるように、何らかの方法で(例えば、副プログラムのパラメタとして)内部識別子を副プログラムの本体へ渡さなければならない(図4参照)。

```
package P is
  procedure PP (TASK__ID : in NATURAL; … );
  :
end P;
package body P is
  procedure PP (TASK__ID : in NATURAL; … ) is
    :
  begin
    :
    Monitor . WILL__BE__DELAYED ( TASK__ID, … );
    delay E;
    :
  end PP;
  :
end P;

with P;
:
task body T is
  :
begin  …  P . PP;  …  end T;
```

図4　タスクの内部識別子が見えない副プログラム呼び出し

原理的に言えば、同一のタスクの外部識別子と内部識別子とに必ずしも異なる値を持たせなければならないというわけではない。むしろタスクの外部識別子と内部識別子との対応に関する管理を省略するために、両者に同じ値を持たせることができれば便利である。

しかし、このためには、何らかの方法でタスクの内部識別子を先に設定してから、それを用いてそのタスクの外部識別子を設定しなければならない(逆でもよい)。外部識別子と内部識別子とに対する要求を両方とも満足させようとすると、デッドロックが起こる恐れがある。

なお、主プログラムは主タスクによって呼び出されるので、主プログラムの本体は、主タスク内部識別子を知らなければならない。

4. タスク間の依存関係

Ada並列プログラムのある種のタスキング振る舞い(例えば、マスタの終了、terminate選択肢によるタスクの終了、abort文の実行によるタスクの強制終了など)は、タスクとそのマスタとの間の依存関係に関連する。実行モニタは、このようなタスキング振る舞いとそれに関連するデッドロックとにも対処しなければならない[5]。このために、各タスクの一意的識別子をそのマスタとの間の依存関係と共に実行モニタが管理するタスク名前表に登録しなければならない。

タスクとそのマスタとの間の依存関係に基づいてタスク間の親子関係を次のように定義する。

Ada並列プログラムの実行中にタスクAとタスクBとに対して次の何れかが成立すれば、AをBの親タスクといい、BをAの子タスクという:

1) Bが直接に依存しているマスタはAである。

2) Bが間接に依存しているマスタは、Aであり、かつ、Aに依存している全てのタスクは、Bのマスタではない。

3) Bが直接に依存しているマスタは、ライブラリ・パッケージであり、Aは、主タスクである。

4) Bが間接に依存しているマスタは、ライブラリ・パッケージであり、かつ、このライブラリ・パッケージに依存している全てのタスクは、Bのマスタではない。Aは、主タスクである。

以上で定義したタスク間の親子関係によれば、親タスクの実行によって新しい子タスクを作り出した際にその親子関係を実行モニタに知らせることができる。

5. Adaタスクに一意的識別子を付ける方法

5.1 基本構想

一意的識別子を意識せずに書かれた被モニタプログラムを、実行モニタの前処理部(preprocessor)がプログラム変換を行って、図1～図4のようなプログラムを生成する。これを実行させて、実行モニタが各タスクに一意的識別子をタスクの生成に応じて同期的に付ける。そのために、以下のような道具立てを行う。

被モニタプログラムの主タスクと並列的に実行し、被モニタプログラムの各タスクの名前、外部識別子と内部識別子、及びタスク間の依存関係を管理するタスクTIMを導入する。タスクTIMに三つのエントリSTARTING_ACTIVATION、CORRELATE とGET_EXTERNAL_ID,及び被モニタプログラムの起動されるタスクを計数するカウンタ(タスクの内部識別子のカウンタでもある)INTERNAL_TASK_IDを持たせる。

被モニタプログラムの全てのタスク本体の宣言部の初めで、タスクの内部識別子変数TASK_IDを宣言し、予め用意した関数GET_INTERNAL_TASK_IDの呼び出しにより内部識別子を設定する(図2参照)。タスクTIMのエントリSTARTING_ACTIVATIONが関数GET_INTERNAL_TASK_IDの本体の中で呼び出される。

被モニタプログラムの各タスク型ごとに、新しいエントリOUTPUT_TASK_IDを導入する。タスクは、起動した後にエントリOUTPUT_TASK_IDのoutパラメタによって自分の内部識別子を本体の外へ知らせる。

被モニタプログラムのタスク(「***」でその名前を表す)の宣言の直後で、タスクの外部識別子変数***_IDを宣言し、予め用意した関数GET_EXTERNAL_TASK_IDの呼び出しにより外部識別子を設定する(図1参照)。タスクTIMのエントリGET_EXTERNAL_IDが関数GET_EXTERNAL_TASK_IDの本体の中で呼び出される。

図5に、一意的識別子を付ける過程を示す。

被モニタプログラムの各タスクが起動すると、関数GET_INTERNAL_TASK_IDの呼び出しによりタスクTIMとのエントリSTARTING_ACTIVATIONによる通信によって、タスクの起動をタスクTIMに知らせ、INTERNAL_TASK_IDの値をエントリSTARTING_ACTIVATIONのパラメタとして返しタスクの内部識別子変数TASK_IDに代入する(図5(1))。

各タスクの宣言が確立した直後に、そのマスタは、関数GET_EXTERNAL_TASK_IDの呼び出し

によりタスクTIMとのエントリGET__EXTERNAL__TASK__IDによる通信によって、タスクの外部識別子をもらって、外部識別子変数***__IDに代入する(図5(2))。

各タスクは、起動した直後にそのマスタとのエントリOUTPUT__TASK__IDによる通信によって、自分の内部識別子をマスタに知らせる(図5(3))。

各タスクのマスタとタスクTIMとのエントリCORRELATEによる通信は、タスクの親タスクの識別子、タスクの名前、タスクの外部識別子と内部識別子をエントリCORRELATEのパラメタとしてタスクTIMに知らせる(図5(4))。タスクTIMは、これらの情報をタスク名前表に登録する。

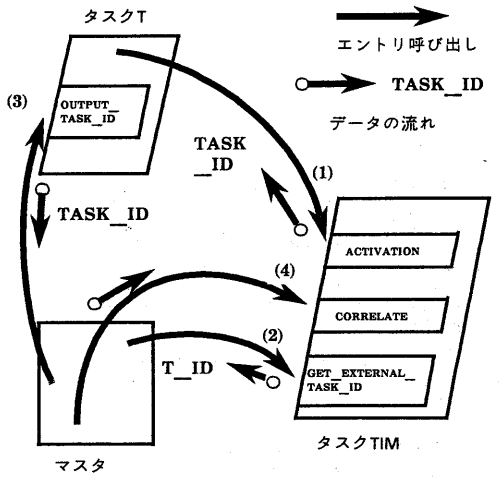

図5　Adaタスクに一意的識別子を付ける方法

従って、被モニタプログラムの各タスクごとに以上で述べた四回のランデブーによる通信によって、各タスクに一意的識別子をタスクの生成に応じて同期的に付け、タスク間の依存関係を動的に把握することができる。

被モニタプログラムの主プログラムの宣言部の初めで、主タスクの内部識別子変数TASK__IDを宣言しその内部識別子を設定する。

被モニタプログラムの全てのライブラリ・パッケージ仕様の可視部の初めで主タスクの内部識別子変数TASK__IDを宣言しその内部識別子を設定する。

タスクは、ライブラリ単位である副プログラムか、ライブラリ・パッケージ仕様の可視部で宣言されている副プログラムかを呼び出す際に、自分の内部識別子を実パラメタとしてその副プログラムに知らせる。

以上で述べたように被モニタプログラムの各タスクに一意的識別子を付ければ、各タスク本体のすぐ内側で直接可視のTASK__IDがタスク自身の内部識別子変数だけであるということは、Adaの可視性規則によって保証することができる。タスクの外部識別子変数はタスク宣言の直後に宣言するので、その可視範囲がタスク宣言の可視範囲と同じであるということも、Adaの可視性規則によって保証することができる。

以上で述べたAdaタスクに一意的識別子を付ける方法は、被モニタプログラムの各タスクのセマンティクスをそのまま保存することができる[4]。

なお、タスクTIM、関数副プログラムGET__INTERNAL__TASK__ID、関数副プログラムGET__EXTERNAL__TASK__IDをまとめて一つのライブラリ・パッケージTASK__ID__MANAGERとして、Adaプログラム・ライブラリに導入する。パッケージTASK__ID__MANAGERの仕様は付録に示す。

5.2　主プログラムとライブラリ・パッケージとに関する変換

実行モニタの前処理部におけるプログラム変換規則を次のような形式で記述する。

```
パターン
適用条件
―――――――――――――――――――――――――― [Rule i]
置き換え
```

変換規則の中では、角括弧で囲んでいるもの(例えば、<id>)をパターン変数という。変換規則の直線の上にあるパターン変数と変換規則の直線の下にあるパターン変数とは、被変換Adaプログラムのソース・テキストの同一部分(空文字列でもよい)に対応する。パターン変数の照合は制限されている。例えば、<id>は識別子としか照合することができない。また、大括弧で省略できる項目を囲み、中括弧で繰り返し項目を囲み、縦線で代替項目を分離する。

変換規則は、その適用条件が成立する時かつその時に限り適用すれば正しい。

主プログラムの文脈節でwith節を用いてライブラリ・パッケージTASK__ID__MANAGERを導入し、宣言部の初めで主タスクの内部識別子変数TASK__IDを宣言しその内部識別子を設定する(**Rule 1**)。

```
procedure <id> is   <[declarative part]>
begin
  <statements>
<Enabling conditions>
    1.  procedure <id> is the main program;
―――――――――――――――――――――――――― [Rule 1]
with TASK__ID__MANAGER;use TASK__ID__MANAGER;
procedure <id> is
  TASK__ID : NATURAL := 1;
  <[declarative part]>
begin
  IC.CORRELATE(1,
    (GET__SIMPLE__NAME("<id>"),0),1,1);
  <statements>
```

ライブラリ・パッケージ仕様の可視部の初めで主タスクの内部識別子変数TASK__IDを宣言しその内部識別子を設定する(**Rule 2**)。

```
package <id> is
<Enabling conditions>
   1.  package <id> is a library package;
```
[Rule 2]
```
package <id> is
  TASK__ID : NATURAL := 1;
```

5.3 タスク型のタスク演算対象に関する変換

タスク型に対して、エントリOUTPUT__TASK__IDを宣言し、本体の宣言部の初めでタスクの内部識別子変数TASK__IDを宣言しその内部識別子を設定し、本体のbeginの直後でエントリOUTPUT__TASK__IDに対応するaccept文を挿入する(**Rule 3** と **Rule 4**)。

```
task body <id> is   <[declarative part]>
begin
  <statements>
```
[Rule 3]
```
task body [id] is
  TASK__ID : NATURAL
          := GET__INTERNAL__TASK__ID;
  <[declarative part]>
begin
  accept OUTPUT__TASK__ID (ID : out NATURAL ) do
      ID := TASK__ID;
  end OUTPUT__TASK__ID ;
  <statements>
```

```
task type <id> [is]
  <[entry declaration]> <[representation clause]>
[end] <[id]>;
```
[Rule 4]
```
task type <id> is
  entry OUTPUT__TASK__ID (ID : out NATURAL );
  <[entry declaration]> <[representation clause]>
end <[id]>;
```

タスク型演算対象Tの宣言の直後で、その外部識別子変数T__IDを宣言し外部識別子を設定する。Tを宣言しているブロック文あるいはプログラム単位の本体のbeginの直後で、TのエントリOUTPUT__TASK__IDを呼び出す文とタスクTIMのエントリCORRELATEを呼び出す文とを挿入する(**Rule 5**)。

単一タスクに関する変換規則は、以上の変換規則 **Rule 4 ~ Rule 5** の結合である(**Rule 6**)。

5.4 副プログラムに関する変換

ライブラリ単位である副プログラムか、ライブラリ・パッケージの可視部で宣言されている副プログラムかに対して、その仮パラメタ部に仮パラメタTASK__ID(この副プログラムを呼び出すタスクの内部識別子変数)を導入する(**Rule 7**)。

ライブラリ単位である副プログラムか、ライブラリ・パッケージの可視部で宣言されている副プログラムかに対する呼び出しは、実パラメタとしてこの副プログラムを呼び出したタスクの内部識別子変数TASK__IDを追加して行う(**Rule 8**)。

```
<is | declare>
  <{basic declarative item}>
  <id> : <task type>;
  <{declarative item}>
begin
  <statements>
```
[Rule 5]
```
<is | declare>
  <{basic declarative item}>
  <id> : <task type>;
  <id>__ID : NATURAL
          := GET__EXTERNAL__TASK__ID;
  ID : NATURAL;
  <{declarative item}>
begin
  <id> . OUTPUT__TASK__ID (ID);
  TIM.CORRELATE ( TASK__ID,
    ( GET__SIMPLE__NAME ("<id>"), 0 ),
     ID, <id>__ID );
  <statements>
```

```
<is | declare>
  <{basic declarative item}>
  task <id> [is]
    <[entry declaration]> <[representation clause]>
  [end] <[id]>;
  <{declarative item}>
begin
  <statements>
```
[Rule 6]
```
<is | declare>
  <{basic declarative item}>
  task <id> is
    entry OUTPUT__TASK__ID (ID : out NATURAL );
    <[entry declaration]> <[representation clause]>
  end <[id]>;
  <id>__ID : NATURAL
          := GET__EXTERNAL__TASK__ID;
  ID : NATURAL;
  <{declarative item}>
begin
  <id> . OUTPUT__TASK__ID (ID);
  TIM.CORRELATE ( TASK__ID,
    ( GET__SIMPLE__NAME ("<id>"), 0 ),
     ID, <id>__ID);
  <statements>
```

```
<procedure | function> <id>
  [ ( <parameter specification>
  {; <parameter specification>} ) ]
<Enabling conditions>
  1.  The subprogram  is a library unit or a declarative
item in a library package;
```
[Rule 7]
```
<procedure | function> <id>
  ( TASK__ID : in NATURAL;
  <parameter specification> {; <parameter specification>} )
```

```
<procedure name | function name >
  [ ( <parameter association>
  {; <parameter association>} ) ]
<Enabling conditions>
  1.  The subprogram  is a library unit or a declarative
item in a library package;
```
[Rule 8]
```
<procedure name | function name >
  ( TASK__ID, <parameter association>
  {; <parameter association>} )
```

5.5 アクセス型演算対象により指されるタスクに関する変換

アクセス型演算対象により指されるタスクに関する変換規則は、**Rule 9** と **Rule 10** との通りである。

```
<is|declare>
  <{basic declarative item}>
  <id> : <access type> := new <task type>;
  <{declarative item}>
begin
  <statements>
```
[Rule 9]

```
<is|declare>
  <{basic declarative item}>
  <id> : <access type> := new <task type>;
  <id>_all_ID : NATURAL
                  := GET_EXTERNAL_TASK_ID;
  ID : NATURAL;
  <{declarative item}>
begin
  <id> . all . OUTPUT_TASK_ID (ID);
  TIM.CORRELATE (TASK_ID,
    (GET_SIMPLE_NAME ("<id>"), 0),
    ID, <id>_all_ID);
  <statements>
```

```
<is|declare>
  <{basic declarative item}>
  <id> : <access type> ;
  <{declarative item}>
begin
  <statements>
  <id> := new <task type>;
<Enabling conditions>
  1.      <access type> is such an access type as
designates a task type;
```
[Rule 10]

```
<is|declare>
  <{basic declarative item}>
  <id> : <access type>;
  <id>_all_ID : NATURAL;
  ID : NATURAL;
  <{declarative item}>
begin
  <statements>
  <id> := new <task type>;
  <id>_all_ID := GET_EXTERNAL_TASK_ID;
  <id> . all . OUTPUT_TASK_ID (ID);
  TIM.CORRELATE (TASK_ID,
    (GET_SIMPLE_NAME ("<id>"), 0),
    ID, <id>_all_ID);
```

Ada並列プログラムの中で、タスクを指すアクセス型を定義し、そのアクセス値を用いて、タスクを格納したり交換したりすることができる[1]。従って、アクセス型演算対象により指されるタスクを同じアクセス型の別の演算対象に代入することができる。この時、タスクのマスタとタスクTIMとのエントリCORRELATEによる通信によって、タスクの親タスクの一意的識別子、タスクを指すアクセス型の演算対象の名前及び一意的識別子をタスクTIMに知らせタスク名前表に登録する(**Rule 11** と **Rule 12**)。

5.6 下位要素となるタスクに関する変換

Ada並列プログラムの中では、タスク型、アクセス型、配列型及びレコード型の組み合わせによって、さまざまなデータ構造を作り出すことができる。従って、Adaタスクに一意的識別子を付けるためには、これらのデータ構造に対応し、次のような演算対象に関するプログラム変換規則が必要である。

```
begin
  <statements>
  <id-1> := <id-2>;
<Enabling conditions>
  1.    <id-1> and <id-2> are the objects of such an
access type as designates a task type;
```
[Rule 11]

```
begin
  <statements>
  <id-1> := <id-2>;
  <id-1>_all_ID := <id-2>_all_ID;
  TIM.CORRELATE (TASK_ID,
    (GET_SIMPLE_NAME ("<id-1>"), 0),
    0, <id-1>_all_ID);
```

```
<is|declare>
  <{basic declarative item}>
  <id-1> : <access type> := <id-2> ;
  <{declarative item}>
begin
  <statements>
<Enabling conditions>
  1.      <id-1> and <id-2> are the objects of such an
access type as designates a task type;
```
[Rule 12]

```
  <is|declare>
  <{basic declarative item}>
  <id-1> : <access type> := <id-2>;
  <id-1>_all_ID : NATURAL := <id-2>_all_ID;
  <{declarative item}>
begin
  TIM.CORRELATE (TASK_ID,
    (GET_SIMPLE_NAME ("<id-1>"), 0),
    0, <id-1>_all_ID);
  <statements>
```

1) タスクを下位要素とする配列型演算対象、

2) タスクを下位要素とするレコード型演算対象、

3) タスク演算対象を下位要素とするレコード型演算対象を指すアクセス型演算対象、

4) タスク演算対象を下位要素とする配列型演算対象を指すアクセス型演算対象、

5) アクセス型演算対象により指されるタスク演算対象を下位要素とするレコード型演算対象、

6) アクセス型演算対象により指されるタスク演算対象を下位要素とする配列型演算対象。

これらの変換規則は、以上で説明した変換規則と本質的に変わらない。下位要素となるタスクの内部識別子変数の宣言と設定とは、変換規則 **Rule 3** と **Rule 4** の通りである。その外部識別子変数の宣言と設定とは、タスクの上位演算対象と同じデータ構造を持つ演算対象を追加すればよい。ここでは紙面の都合で省略する。

6. 考察

Adaタスクに一意的識別子を付けることは、Ada並列プログラムを対象とする動的解析ツール、ソー

ス・プログラム・デバッガなどのAdaプログラミング支援ツールをAda処理系の上で開発する際に不可欠である。

Adaプログラミング支援ツールをAda言語のレベルで開発することは、Adaプログラミング支援環境を構築する際に満足すべき要求の一つである[2),3)]。従って、本論文で提示した方法は、Adaプログラミング支援環境の構築にとって、普遍の意味がある。

従来、Adaタスクに一意的識別子を付ける方法については、殆ど報告されていない。Germanはプログラム変換によってタスク型演算対象に一意的識別子を付ける方法を与えている[7)]。これは、被モニタプログラムのタスク型をタスク演算対象要素とタスク識別子要素とを持つレコード型に変換し、更に、タスクの起動中にも一意的識別子を持たせるために元のタスク本体を変換後のタスク本体中のブロック文に変換するものである。

この方法には、次の問題点がある。

1) 被モニタプログラムのエントリ呼び出しを全てレコードの選択要素であるタスク演算対象に対するエントリ呼び出しに変換しなければならない[7)]。それは、プログラムを変換する際の手間を増やすばかりではなく、プログラム変換規則の正当性の検証も難しくなる。

2) Adaではタスク本体に関する例外処理のセマンティクスとブロック文に関する例外処理のセマンティクスとが異なるので、変換前のプログラムと変換後のプログラムとは、例外処理に対して等価ではない[7)]。

3) プログラムの全てのタスクに一意的識別子を付けることができるわけではない。また、タスクとそのマスタとの間の依存関係を考慮していないので、Ada並列プログラムのタスキング振る舞いを完全にはモニタすることができない[7),8),9)]。

結局、この方法に基づいて開発したAdaプログラミング支援ツールも、タスク間の依存関係、例外処理などに関係するAdaタスキングの機能に対処することができなかった[7),8),9)]。

我々が提示した方法は、この方法の問題点を解決しているので、我々の方法に基づいて開発したAdaプログラミング支援ツールは、この方法に基づいて開発したAdaプログラミング支援ツールが対処できなかったAdaタスキングの機能にも対処することができる[5)]。

本論文で提案した方法を既にData General社が提供しているAda処理系ADE[10)]の上で実現した。

参考文献

1) United States Department of Defense : Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A), Jan. 1983.
2) United States Department of Defense : "STONEMAN", Requirements for Ada Programming Support Environment, 1980.
3) Taylor,R.N. and Standish,T.A. : Steps to an Advanced Ada Programming Envirronment, IEEE Trans. Softw. Eng., Vol. SE-11, No.3, pp.302-310, 1985.
4) Cheng,J., Araki,K. and Ushijima,K. : Monitoring Ada Tasking Programs Correctly,「ソフトウェア科学、工学における数理的方法」研究集会、京都大学数理解析研究所、Sep. 8-10 1986.
5) 程京徳、荒木啓二郎、牛島和夫：Adaタスキングにおけるデッドロックとライブロックの分類と検出、日本ソフトウェア科学会第3回大会論文集、D-2-3, pp.133-136, 1986.
6) Barnes,J.G.P. : Programming in Ada (second edition), Addison-Wesley, 1984.
7) German,S.M. : Monitoring for Deadlock and Blocking in Ada Tasking, IEEE Trans. Softw. Eng., Vol.SE-10, No.6, pp. 764-777, 1984.
8) Helmbold,D. and Luckham,D.C. : Debugging Ada Tasking Programs, IEEE Software, Vol.2, No.2, pp.47-57, 1985.
9) Helmbold,D. and Luckham,D.C. : Runtime Detection and Description of Deadness Errors in Ada Tasking, ACM Ada Letters, Vol.4, No.6, pp.60-72, 1985.
10) Data General Corp. : Ada Development Environment (ADE) (AOS/VS) User's Manual, 1984.

付録： パッケージTASK_ID_MANAGERの仕様

```
package TASK_ID_MANAGER is
  subtype SIMPLE_NAME is
    STRING ( 1..LENGTH_OF_IDENTIFIER );
  type NAME is
    record
      NAME : SIMPLE_NAME;
      INDEX : NATURAL;
    end record;
  -- Declare the name table of tasks
  function GET_SIMPLE_NAME ( NAME : STRING )
    return SIMPLE_NAME;
  -- Return the simple name of NAME
  function GET_INTERNAL_TASK_ID
    return NATURAL;
  -- Return an internal task identifier
  function GET_EXTERNAL_TASK_ID
    return NATURAL;
  -- Return an external task identifier
  task TIM is
    entry STARTING_ACTIVATION (
      TASK_ID : out NATURAL );
    -- Return an internal task identifier
    entry CORRELATE ( PARENT : in NATURAL;
      NAME_OF_TASK : in NAME;
      EXTERNAL_ID : in NATURAL;
      INTERNAL_ID : in NATURAL );
    -- Operate the name table of tasks
    entry GET_EXTERNAL_TASK_ID (
      TASK_ID : out NATURAL );
    -- Return an external task identifier
  end TIM;
end TASK_ID_MANAGER;
```