ソフトウェア工学 58－14
（１９８８．２．５）

Ａｄａタスキングプログラムのデバッギングシステム

吉野　真澄[+]，　天満　隆夫[+]，　坪谷　英昭[+]，　田中　稔[++]，　市川　忠男[++]

[+]広島大学大学院　　[++]広島大学工学部

本稿では、Adaタスキングプログラムのためのデバッギングシステムについて述べる。システムは、1)プログラム実行中に発生したデッドロックを常に検出する、2)タスク間通信といった複数のタスクにまたがるイベントに関する情報を提供する、3)イベント記述によってプログラムの実行に制約を課すことができる、といった機能を提供している。プログラマは、これらの機能を用いて、Adaタスキングプログラムのデバッグを効率良く行なうことができる。

A Debugging System for Ada Tasking Programs

Masumi YOSHINO, Takao TENMA, Hideaki TSUBOTANI,
Minoru TANAKA, and Tadao ICHIKAWA

Faculty of Engineering, Hiroshima University
Shitami, Saijo-cho, Higashi-Hiroshima, 724 Japan

This paper presents a bedugging system for Ada tasking programs. The system provides facilities 1) to detect deadlock errors when it occurs during the execution of a program, 2) to offer information of events which affect more than one tasks such as inter-task communications, and 3) to impose constraints on the execution of programs by specifying description of events. By using the debugging system, the programmers can debug Ada tasking programs efficiently.

1．はじめに

最近のプログラミング言語、例えばAda[1]，Modula-2[2]などでは、並行プログラムを記述できる機能が導入されている。これらの言語を用いたプログラム開発を支援するためには、並行プログラムを開発するための環境が提供されなければならない。本研究では、プログラミング言語Adaをターゲットとして、そのタスキングプログラムの実行環境を構築し、さらにその環境上にデバッギングシステムを開発した。

Adaタスキングプログラムの実行は、逐次的プログラムの実行と比べて、

(1) あるタスクの実行が他のタスクの実行に影響する、
(2) タスク間通信の順序が非決定的であるため、デッドロックに陥る可能性を持つ、
(3) スケジューリングの方法やデバッガの介入などに影響される、

などといった特徴を持ち、実行には再現性がないといった問題がある。このため、一般にタスキングプログラムのデバッグは非常に困難である。タスキングプログラムのデバッグを支援するためには、デバッガは特に以下のような機能を提供すべきであると考えられる。

(1) デッドロックエラーの検出。
(2) タスク間通信のような複数のタスクにまたがるイベントに関する情報の提供。
(3) 期待するイベントの発生順序と異なるイベントの発生の検出。
(4) スケジューリング方法の指定、変更。

(1)については、本システムでは従来のシステムで検出できないデッドロックエラーの検出も行なえるようにした。(2)については、プログラムの状態に関する情報に加えて、ヒストリ情報の提供も行なっている。さらに、(3)については、タスク間通信といったイベントの発生する順序をプログラムの実行に対する制約として記述できる機能(イベント記述の機能)として実現している。(4)の機能については、現在サポートしていないが、今後研究を行なっていく予定である。

以下、2．でAdaにおけるタスキングメカニズムについて説明する。システムの概要を3．で述べた後、デッドロックの検出、モニタ情報の提供、イベント記述について、それぞれ4．5．6．で説明する。最後に7．でシステムのまとめを述べる。

2．Adaにおけるタスキングメカニズム

Adaでは、タスクと呼ばれるプログラム単位を用いて並行プログラムを記述する。本章では、Adaのタスキングプログラムについてその概略を述べる。

2．1　タスク

Adaにおいて、タスクとは並行に実行可能なプログラム単位のことである。タスクは型を持ち、タスクオブジェクト宣言あるいは割り当て子によって動的に生成される。

タスク間には生成の関係によって親子関係が定義される。例えば、タスクTpがタスクTcを生成した場合、タスクTpはTcの親タスク、タスクTcはTpの子タスクと呼ばれる。親タスクは、すべての子タスクが終了するまで終了することができない。

また、各タスクは、少なくとも1つのマスタに依存する。タスクのマスタとは、タスクを生成したブロック文、サブプログラム、ライブラリパッケージ、あるいはタスクである。マスタの実行は、そのマスタが生成したすべてのタスクが終了するまで、マスタを抜け出ることができない。例えば、あるブロック文Bの中でタスクAが生成された場合、ブロック文BがタスクAのマスタとなる。この場合、ブロック文Bを実行するタスクは、タスクAが終了するまでブロック文Bの外へ実行を移すことができず、ブロック文Bから抜け出る実行文(例えば、end文やexit文等)の前でタスクAの実行終了を待機する状態(ブロック文の完了待機状態(図1))となる。

2．2　タスク同期メカニズム

タスク間の同期・通信はランデヴと呼ばれるメカニズムによって行なわれる。ランデヴは、2つのタスクがそれぞれエントリコール文とアクセプト文を実行することによって行なわれる。エントリコール文は、タスクオブジェクト名、エントリ名及び実引数よりなり、ランデヴしたいタスクのエントリに対して行なわれる。ここで、エントリとは、他のタスクからのエントリコールを受ける受け口であり、各タスクは0個以上のエントリを持つことができる。また、複数のタスクが1つのエントリに対して、エントリコールを行なうことができる。アクセプト文はエントリ名及びランデヴの際に実行される文の列より成る。各エントリはキューを持ち、エントリにエントリコールを実行したタスクは、呼び出しているエントリのキューに入れられ、待機状態となる。呼び出されたエントリを持つタスクがそのエントリに対するアクセプト文を実行する時、キューの先頭からタスクが1つだけ取り出され、ランデヴが行なわれる。キューが空の時は、エントリコールが行なわれるまで、このタスクは待機状態となる。ランデヴが行なわれる間、エントリコールを実行中のタスクは、ランデヴの終了まで待たされる。ランデヴ終了後、2つのタスクは同時に実行を再開する。

2．3　タスク状態

本システムでは、タスクが生成されてその実行が終了するまでにとりうる状態を17に分類している(図1)。各タスクは、生成されてから終了するまでの任意の時点で、これらの状態のうち必ず1つの状態を持つ。

タスクが生成されると最初に起動状態となり、タスクごとに割り当てられる作業領域の確保と宣言されている変数の確立が行なわれる。この時、確立している変数中に別のタスクの宣言があれば、起動状態のタスクは起動待機状態となり、その子タスクの起動が開始される。子タスクの起

動が終了するまで、親タスクの起動は待たされる。起動を終えたタスクは、実行可状態となる。実行可状態のタスクしか実行状態に移ることができない。実行状態のタスクは、ディレイ文、エントリコール文、アクセプト文の実行などにより状態を変えていく。

本システムでは、これらの状態のうち、起動待機、エントリコールによる待機、アクセプト文による待機、ランデヴ待機、ブロック文の完了待機、サブプログラムの完了待機、完了、終了待機の状態をブロック状態と呼ぶ。ブロック状態のタスクは、他のタスクから何らかの作用がなければ、状態を変えることができない。

各状態への遷移は、タスキングスーパバイザへのコールによって行なわれる[3]。タスキングスーパバイザについては3章で述べる。

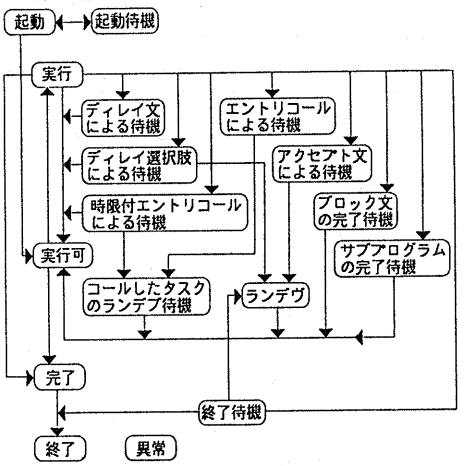


図1　タスクの状態遷移

3．システム概要

本システムの構成を図2に示す。本システムでは、Adaタスキングプログラムの実行は、著者らが開発した高水準言語マシンSOA[4]上で行なわれる。SOAは、Pascal、Adaといったクラスの言語のための高水準言語マシンであり、その命令インタプリタをC言語で作成している。タスク管理を行なうために、SOAのオペレーティングシステム内にタスキングスーパバイザを付加し、タスキングスーパバイザ上にデバッガの構築を行なった。ユーザは、デバッガを通してコマンドを入力したり、イベント記述を与えることができる。デバッガは、ユーザからの要求に従って、モニタ情報を提供する。以下、タスキングスーパバイザとデバッガについて説明する。

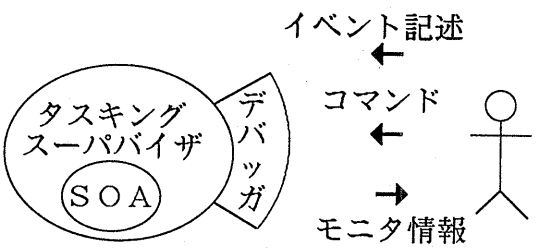


図2　システム構成

3．1　タスキングスーパバイザ

タスキングスーパバイザは、タスキングプログラムの実行を支援するためのオペレーティングシステム機能の一部であり、タスクの実行管理を行なう。プログラムの実行がタスク宣言、ディレイ文、エントリコール文、アクセプト文、セレクト文、ブロック文の完了及びタスク本体の完了に達した時、タスキングスーパバイザが呼び出される。

タスキングスーパバイザは、以下のことを行なう。

(1) 動的に生成されるタスクに一意的な識別子を与え、各タスクの状態の更新を行なう。

(2) タスクの親子関係及びブロック依存の関係を保持する。ここで、ブロック依存の関係とは、ブロック状態のタスクがどのタスクによってブロックされているかを示す関係のことである。

(3) レディキュー、ディレイキュー、エントリキューを管理する。レディキューには実行可状態のタスクが入れられ、ディレイキューには遅延による待機状態のタスクが入れられる。エントリキューは各エントリに対して割り当てられ、エントリコールによる待機状態のタスクが入れられる。

(4) 実行中のタスクがブロック状態に陥った時、デッドロックが発生していないかどうかチェックを行なう。

(5) デバッグに必要な情報をデバッガに提供する。

3．2　デバッガ

デバッガは、ユーザからの要求によって、デバッグのために必要な情報を提供したり、プログラムの実行がイベント記述によって課せられた制約を満たしているかどうかチェックを行なう。提供可能な情報については5章で、イベント記述については6章で詳述する。

4．デッドロックの検出

デッドロックとは、タスキングプログラムの実行中、あるブロック状態のタスクが永久にその実行を継続できなくなった状態をいう。タスキングプログラムの実行を支援する環境では、このようなデッドロックは必ず検出されなければならない。

並行プログラムのデッドロック検出には、プログラム解析による静的な方法と実行時モニタリングによる動的な方法[5]がある。静的な方法は単純なエラーを検出するには適しているが、多くのエラーを検出するには、実行されるプログラムの可能な状態を全て調べなければならないため、非常にコストがかかるという欠点を持つ。また、Adaタスキングプログラムにおいては、タスクは動的に生成されるため、静的な方法の適用には限界がある。本システムでは、タスクの実行をタスキングスーパバイザによって動的にモニタするで、デッドロックの検出を行なっている。以下、本システムにおいて検出可能なデッドロックの分類とその検出方法について述べる。

４．１　デッドロックの分類

従来のシステムにおいて、プログラムの実行をモニタすることによって検出できるデッドロックには、

(1) エントリコールが循環になった結果発生する循環デッドロック、

(2) すべてのタスクがブロック状態になった結果発生する大域デッドロック、

の２種類ある。これら２種類のデッドロックは、プログラムの状態情報のみから検出することができる。

デッドロックは可能な限り早期に検出されることが望ましいが、従来のシステムではアクセプト待機のタスクを含むデッドロックを早期に検出することはできない。アクセプト待機のタスクを含むデッドロックとは、タスクが永久にエントリコールが行なわれないエントリでアクセプト待機状態になっているデッドロックのことである。プログラムの状態情報だけからこのデッドロックを検出することはできない。このため、従来のシステムではこの種のデッドロックの検出は、結果として起こる大域デッドロックの発生まで遅らされる。そのためプログラマは、大域デッドロックが発生した時点でのプログラムの状態からデッドロック発生の原因を究明しなければならない。これに対して、本システムでは、従来のシステムで検出可能な２種類のデッドロックに加えて、プログラムのコンパイル情報を用い、アクセプト待機におけるデッドロックの早期検出を可能にしている。

本システムで検出可能なデッドロックをまとめると以下の３種類となる。

(1) 大域デッドロック

(2) 循環デッドロック

(3) アクセプト待機におけるデッドロック

４．２　デッドロックの検出方法

デッドロックの検出は、タスキングスーパバイザによって行なわれる。以下、デッドロックの検出方法について、デッドロックの種類ごとに説明する。

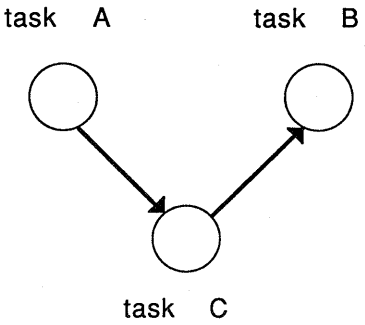
(1) 大域デッドロック

大域デッドロックとは、プログラム中の終了していないタスクのすべてがブロック状態に陥ったデッドロックである。これはタスクのスケジューリングの際に、レディキュー及びディレイキューをチェックすることにより検出することができる。レディキューにはすべての実行可状態のタスク、ディレイキューにはディレイ文による待機、ディレイ選択肢による待機、及び時限付きエントリコールによる待機状態のタスクが入れられている。ディレイキュー中のタスクは、時間の経過によって実行可状態になることができる。タスクのスケジューリングは、レディキューの先頭からタスクを取り出すことによって行なう。スケジューリングを行なう際、レディキュー、ディレイキューの２つのキューが空であれば、大域デッドロックが発生している。

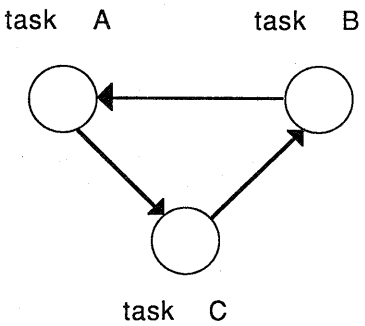
(2) 循環デッドロック

循環デッドロックは、タスクのエントリコールの関係が循環に陥った時に発生する。タスキングスーパバイザは、タスクがエントリコールによる待機状態になった際、呼び出しているタスクと呼び出されているタスクをリンクで結合し、２つのタスク間でランデヴが行なわれた後、リンクを取り除くといった作業を行なっている。そのため、このデッドロックは、あるタスクがエントリコールによる待機状態になった時、生成したリンクをたどることによって検出可能である。すなわち、リンクが循環になっていれば循環デッドロックが発生している。

循環デッドロックの例を図３に示す。図３(a)において、タスクＡはタスクＣに、タスクＣはタスクＢにそれぞれエントリコールを行なっている。この状態において、タスクＢがタスクＡのエントリに対してエントリコールを行なった時(図３(b))、循環デッドロックが発生する。この時、タスクＡ、Ｂ、Ｃはそれぞれデッドロック状態である。



(a)　循環デッドロック発生前



(b)　循環デッドロック発生

図３　循環デッドロック

(3) アクセプト待機におけるデッドロック

アクセプト待機におけるデッドロックは、タスクが永久にコールされることができなくなったエントリでアクセプト待機状態になった時発生する。具体的には、あるエントリでアクセプト待機のタスクに対し、

a) エントリコール文を持つタスクがない場合、

b) エントリコール文を持つタスクがすべて完了あるいは

終了している場合、

c) エントリコール文を持つすべてのタスクがアクセプト待機のタスクに対し、直接あるいは間接的にブロック依存している場合、

にアクセプト待機におけるデッドロックが発生する。

このデッドロックはプログラムの状態情報だけでは検出できない。本システムでは、タスクのエントリコールに関する情報をタスキングスーパバイザに与えることによってデッドロック検出を可能にしている。この情報は、各タスクがどのエントリに対するエントリコール文を持っているかを示す表(エントリコール表)の形で記述される。これはプログラムの静的な解析によって得ることができる。図4にエントリコール表の例を示す。表の横の欄はタスク、縦の欄はエントリコール文を示している。例えば、この表の左から2列目は、ソースプログラム中にタスクAのエントリYへのエントリコール文を持っているのはタスクC及びタスクDであることを示している。

エントリコール

タスク	A.X	A.Y	B.Z	C.M
A				1
B	1			
C		1	1	
D		1		

図4　エントリコール表

エントリコール表を用いることによって、アクセプト待機状態のタスクにおけるデッドロックは、以下のようにして検出される。

任意のタスクがブロック状態になった時、そのタスクが直接的あるいは間接的にブロック依存しているタスクの状態についてタスキングスーパバイザが調べていく。調べたタスク中にアクセプト文による待機状態のタスクがあれば、エントリコール表(図4)を用いて、そのタスクが他のタスクからコール可能かどうかを調べる。これは、アクセプト待機状態のタスクが待っているエントリコール文を持っているすべてのタスクの状態をチェックすることにより行なう。このチェックによって、他のタスクからのコールが不可能であれば、アクセプト待機におけるデッドロックが発生している。アクセプト待機におけるデッドロックの例を図5に示す。図において、タスクAはエントリYでアクセプト待機状態であり、エントリコール表(図4)は、タスクAのエントリYに対し、タスクCとタスクDがエントリコール文を持っていることを示している。タスクDは終了状態であり、タスクCはタスクAに対し間接的にブロック依存の関係であるから、図の3つのタスクA，B，Cは、デッドロック状態に陥っていることがわかる。

エントリコール表において、任意のエントリコール文を持つタスクが完了あるいは終了した時、その欄は消去される。この際、このタスクのエントリコールを待っているアクセプト待機状態のタスクをチェックし、デッドロックが発生していないかどうかのチェックを行なう。このときのアクセプト待機におけるデッドロックの検出は、アクセプト待機状態のタスクが待っているエントリコールを持つタスクの完了あるいは終了まで遅れるが、大域デッドロックの発生まで遅れることはない。

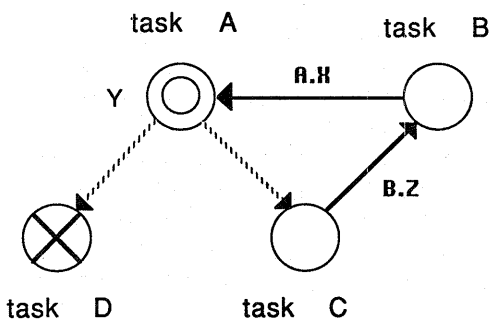


図5　アクセプト待機におけるデッドロック

4．3　デッドロック検出の問題点

Adaでは、タスクを強制終了させる実行文(abort文)が規定されている。すなわち、abort文によって待機状態のタスクを強制終了させることでデッドロックを解消することができる。abort文を考慮すると、あるタスクが強制終了させられる可能性がなくなるまでデッドロック状態に陥ったと断定することはできない。しかし、ある待機状態にあるタスクが強制終了させられるかどうかを判別することは非常に難しい。本システムでは、この状況に対処するため、エントリコール表と同様にabort文に関する情報もタスキングスーパバイザに渡している。デッドロック検出はabort文を考慮せずに行なわれる。このデッドロックの検出後、システムはデッドロック状態のタスクが強制終了される可能性があるかどうかをチェックし、その情報をプログラマに提供する。デッドロック発生後、プログラマはその情報に基づいてプログラムの実行を終了あるいは継続させることができる。

5．モニタ情報の提供

本システムはプログラマに並行プログラムの実行状態に関する情報(モニタ情報)を提供する[6]。プログラマがデバッガにモニタ情報の提供を要求した時、タスキングスーパバイザはモニタ情報をデバッガに渡し、デバッガはこの情報をプログラマに表示する。モニタ情報は、以下の時に提供可能である。

(1) プログラム終了時
(2) デッドロック発生時
(3) イベント記述による制約に反するイベントの発生時
(4) イベント記述によるブレイク時

なおイベント記述については次章で述べる。

プログラマに提供されるモニタ情報は、デバッガの持つ

シンボル情報によってソースプログラム中の名前で表示される。モニタ情報には、プログラムの状態情報とヒストリ情報の２種類がある。以下、これらの情報について説明する。

５．１　プログラムの状態情報

プログラムの状態情報とは、任意の時点におけるプログラムの状態に関する情報のことであり、タスクの状態情報、タスクの依存関係の情報、キューの状態情報がある。

(1) タスクの状態情報

起動されたすべてのタスク、あるいはユーザが指定したタスクの状態に関する情報である。図６(a)にタスクの状態情報の例を示す。図において、タスク名mainのタスクは実行可状態、タスク名managerのタスクはエントリコールによる待機状態、タスク名xのタスクは起動状態であることが示されている。

```
TASK TYPE    TASK NAME          STATE
  main         main             ready
   t1         manager       suspended by entry call
operater        x               activating
   ⋮            ⋮                 ⋮
```

(a) タスクの状態情報

```
TASK TYPE    TASK NAME       STATE
   t1          manager    suspended by entry call
                             ENTRY NAME access
                             --> TASK NAME z1
```

(b) タスクの依存関係の情報

```
ENTRY QUEUE   TASK NAME                    TASK NAME
ENTRY NAME e1   1 t1       ENTRY NAME ex   1 ts1
                2 t2                          ⋮
                  ⋮
READY QUEUE     TASK NAME
                1 main
                  ⋮
DELAY QUEUE is empty !!
```

(c) キューの状態情報

図６　プログラムの状態情報の例

(2) タスクの依存関係の情報

起動されたすべてのタスク、あるいはユーザが指定したタスクの依存関係の情報である。タスクの依存関係とは、タスクの親子関係、及びタスクのブロック依存の関係のことである。注目するタスクの親タスク、子タスク、兄弟タスクのタスク名と状態が示される。また、ブロック状態のタスクにおいて、それが依存しているタスクに関する情報が提供される。例えば、エントリコールによる待機状態のタスクについては、このタスクが呼び出しているタスクの名前とエントリの名前が示され、完了状態のタスクであれば、そのすべての子タスクの名前と状態が示される。図６(b)にタスクの依存関係の情報の例を示す。図において、タスク名managerのタスクがエントリコール待機状態であり、呼び出しているタスクの名前はz1、エントリの名前はaccessであることを示している。

(3) キューの状態情報

キューに入れられているタスクに関する情報である。本システムでは、レディキュー、ディレイキュー、エントリキューの３種類のキューを用いているため、これら３つのキューに入れられたすべてのタスク名が示される。図６(c)にキューの状態情報を示す。図において、エントリ名e1のエントリキューの先頭からタスクt1、t2、エントリ名exのエントリキューの先頭からタスクtxが入れられていることが示されている。また、レディキューの先頭にはタスクmainが入れられており、ディレイキューは空であることが示されている。

５．２　ヒストリ情報

ヒストリ情報とは、プログラムの実行がどのように行なわれたかに関する情報である。ヒストリ情報には、タスクの状態遷移情報、ランデヴ情報がある。

(1) タスクの状態遷移情報

タスクの状態遷移に関する情報である。図７(a)にタスクの状態遷移情報の例を示す。図はタスクx、yの状態遷移を示している。すべてのタスクの状態遷移情報を知ることによって、プログラマはプログラムの実行がどのように行なわれたかを知ることができる。また、任意のタスクを指定することによって、注目するタスクのみの状態遷移を知ることができる。

```
 TASK TYPE  TASK NAME      STATE
⋮   ⋮         ⋮            ⋮
7   T2         y           suspended by entry call
8   T1         x           executing
9   T2         y           suspended by rendezvous
10  T1         x           rendezvous
11  T2         y           ready
    ⋮         ⋮             ⋮
```

(a) タスクの状態遷移情報

```
⋮                    ⋮
2  rendezvous   t1  on  e1  from  t2
3  rendezvous   tx  on  ea  from  ty
⋮                    ⋮
```

(b) ランデヴ情報

図７ ヒストリ情報の提供の例

タスクの状態を指定することによって、指定された状態になったタスクに関する情報を要求することもできる。これによって、タスクの生成や終了がどのように行なわれた

か、あるいはタスクがどのようにスケジューリングされたかといった情報を得ることができる。

(2) ランデヴ情報

ランデヴがどのように行なわれたかに関する情報である。呼び出したタスク名、呼び出されたタスク名とエントリ名が提示される。図7(b)にランデヴ情報の例を示す。図において、タスクt1がエントリe1においてタスクt2とランデヴを行ない、次にタスクtxがエントリeaにおいてタスクtyとランデヴを行なったことを示している。

6．イベント記述

通常プログラマは、タスクの同期や通信などのイベントの発生順序に関する制約を考えながらプログラムを作成する。このようなプログラマの考え反してプログラムが実行された時、デッドロックなどのエラーが発生する場合が多い。このようなエラーの検出を容易にするため、本システムではイベント記述の機能を導入した[7] [8]。この機能を用いて、プログラマは意図するイベントの順序情報(イベント記述)をシステムに与えることができる。システムは与えられた順序情報に反する実行が起こった時にプログラムの実行を中断し、プログラマにその発生を知らせる。

6．1 イベント記述処理の概要

イベント記述で書かれたイベントの発生順序に関する制約はトランスレータによって内部記述に変換され、デバッガに渡される。プログラムの実行中、タスキングスーパバイザはデバッガに各イベント(イベントについては次節で述べる)の発生を知らせる。デバッガは、イベントの発生順序がイベント記述による制約を満たしているかどうかチェックする。制約に反していれば、プログラムの実行を中断し、ユーザからの要求に従ってモニタ情報の提供や実行の継続などを行なう。

6．2 イベント記述言語

著者らはイベント記述を記述するための言語を開発した。イベント記述言語の設計目標は、

(a) デバッギングに有用なイベント記述が行なえること、
(b) ユーザがイベント発生の制約を記述しやすく、記述されたイベント記述が理解しやすいこと、
(c) イベント記述制約のチェックをシステムが行ないやすいこと、

である。イベント記述は、各タスクに対して、あるいはプログラム全体に対して行なうことができる。以下、イベントの定義とイベント記述によって表現できる制約について説明する。

6．2．1 イベント

本システムでは、Adaタスキングプログラムの実行におけるイベントとして以下の6種類が定義されている。

(1) タスクの起動
(2) タスクの終了
(3) エントリコール
(4) アクセプト
(5) ランデヴ
(6) ディレイ

プログラマは、これら6種類のイベントの発生の順序に対して制約を課すことができる。上の6種類のイベントをイベント記述言語で記述すると次のようになる。

(1) (CREATE name1)
(2) (TERMINATE name1)
(3) (ENTRY CALL name1 ON name2 TO name3)
(4) (ACCEPT name1 ON name2)
(5) (RENDEZVOUS name1 ON name2 FROM name3)
(6) (DELAY name1)

ここでname1はタスク名、name2はエントリ名、name3はタスク名でなければならない。下線は予約語を示している。また、nameはANYで置き換えることができる。例えば、次の記述は任意のタスクによるタスクt1のエントリe1へのエントリコールというイベントを示している。

(ENTRY CALL ANY ON e1 TO t1)

6．2．2 イベント記述による制約

現在、イベント記述言語によって記述できる制約は次の3種類である。

(1) 最初に起こるべきイベントの指定。
(2) あるイベントの次に起こるべきイベントの指定。
(3) あるイベントの発生に対するブレイクの指定。

これら3つのイベント発生の制約をイベント記述言語で記述すると以下のようになる。

(1) FIRST (イベント列)
(2) NEXT (イベント列)
(3) BREAK (イベント列)

イベント列には、6．2．1で示した6つのイベントをFIRSTでは1つ、NEXT、BREAKでは複数個記述するのを許している。また、FIRSTとNEXTのイベント列中では、ORオペレータを用いてイベントを結合することができる。例えば、

```
FIRST [ OR [ (CREATE t1) (CREATE t2) ] ]
```

は、最初にタスクt1の起動あるいはタスクt2の起動が起こらないといけないことを示している。また、NEXTのイベント列中には、繰り返しを示す※文の使用が許されている。例えば、

```
NEXT [ * [ (ENTRY CALL t1 ON e1 TO t2)
           (RENDEZVOUS t1 ON e1 FROM t2) ]
       (TERMINATE t1) ]
```

は、t1からt2のe1に対するエントリコールとタスクt1がt2とエントリe1でランデヴを繰り返した後、タスクt1が終了しなければならないことを示している。

また、任意のイベント(ANY)、及び無のイベント(NULL)を記述することもできる。例えば、イベントA，B，Cの発生をA->B->CあるいはA->Cの順番で制約を課したい場合、

NEXT [(A) OR [(B) (NULL)] (C)]

と記述すればよい。また、イベントBの発生前にイベントAが発生していなくてはいけないという記述は、

NEXT [(A) * [(ANY)] (B)]

となる。すべてのランデヴの発生に対し、プログラムの実行を中断してほしい場合は、

BREAK [(RENDEZVOUS ANY ON ANY FROM ANY)]

と記述すればよい。

7．おわりに

本稿では、Adaタスキングプログラムのためのデバッギングシステムについて述べた。本システムが提供する機能には、

(1) デッドロックエラーの早期検出、
(2) タスク間通信のような複数のタスクにまたがるイベントに関するモニタ情報の提供、
(3) イベント記述によるプログラムの実行に対する制約の付加、

がある。これらの機能を用いてユーザは、効率的にタスキングプログラムのデバッグを行なうことができる。

システムはSUN-3のUNIX上にC言語を用いて構築されている。現在、テストプログラムによる性能評価を行なっている。

今後の課題として、以下のことが挙げられる。

(1) スケジューリングの変更機能の導入。
(2) ヒストリ情報で与えられた状態遷移に基づくプログラム実行の再現する機能。
(3) イベント記述仕様の改善。

タスクプログラムのエラーには、デッドロック状態の他にタスクの飢餓状態がある。このような飢餓状態を検出できるようなイベント記述を行なえることを現在検討中である。

謝辞　本研究を進めるにあたり、有益な御意見を頂いた広島大学工学部第二類平川正人博士、システムの実現に際して御協力頂いた情報システム研究室の山本昭二氏、永良裕氏、佐藤康臣氏、ならびに日頃御討論頂く同研究室の諸氏に感謝する。

文献

[1] Reference Manual for the Ada Programming Language, U.S. Dep. Defense, Rep. ANSI/MIL-STD-1815A, Jan. 1983.

[2] N.Wirth," Programming in Modula-2 2nd Edition," Springer-Verlag, 1983.

[3] T.P.Baker and G.A.Riccardi,"Ada Tasking: From Semantics to Efficient Implementation," IEEE Software, Vol.2, No.2, pp.34-46, Mar. 1985.

[4] H.Tsubotani, N.Monden, M.Tanaka, and T.Ichikawa, "A High Level Language-Based Computing Environment to Support Production and Execution of Reliable Programs," IEEE Transactions on Software Engineering, Vol. SE-12, No.1, pp.134-146, Jan. 1986.

[5] S.M.German,"Monitoring for Deadlock and Blocking in Ada Tasking," IEEE Transactions on Software Engineering, Vol. SE-10, No.6, pp.767-777, Nov. 1984.

[6] D.Helmbold and D.Luckham,"Debugging Ada Tasking Programs," IEEE Software, Vol.2, No.2, pp.47-57, 1985.

[7] F.Baiardi, N.D.Francesco, and G.Vaglini, "Development of a Debugger for a Concurent Language," IEEE Transactions on Software Engineering, Vol. SE-12, No.4, pp.547-553, Apr. 1986.

[8] D.Helmbold and D.Luckham,"TSL: TASK SEQUENCING LANGUAGE," Proceedings of the Ada International Conference, pp.255-274, May 1985.