# Ada 並列プログラムの事象駆動型実行モニタ EDEN の開発と応用†

程 京徳†† 荒木啓二郎†† 牛島和夫††

並列プログラムのテスト・デバッグにおける道具と方法は，並列・分散処理ソフトウェアを開発する際に不可欠なものである．本論文では，著者らが開発した Ada 並列プログラムの事象駆動型実行モニタ EDEN の設計思想，基本機能，実現技法および応用について述べる．EDEN は，並列処理単位間の相互作用のみに着目し並列プログラムをテスト・デバッグする手段を提供することを目指し，かつ，Ada プログラミング支援環境の要求仕様に応じて開発したものである．その主な機能は，Ada 並列プログラムの実行を監視しそのタスキング挙動に関する情報を収集，保存してから利用者の要求に応じて報告することである．また，収集した情報に基づいて被監視プログラムの実行中に起こる通信デッドロックを自動的に検出しその原因を報告する．EDEN は，主に Ada 並列プログラムのタスク間の相互作用におけるエラーを検出しその発生場所を特定するテスト・デバッグ支援ツールとして使うことができる．

## 1. はじめに

逐次プログラムのテスト・デバッグについては，道具から方法まで有効なものが既に数多く提案され実用に供されている．これに対して，並列プログラムのテスト・デバッグの分野は，依然として未熟の状態である．

並列プログラムは，その挙動から見れば，幾つかの並列処理単位からなっており（各並列処理単位は更に幾つかの並列処理単位からなるかもしれない），各並列処理単位が内部のデータ処理をしながら相互に協調し相手の挙動に影響を与え，全体として「計算」をしているものと見なすことができる．この観点に基づいて，並列プログラム中のエラーを，並列処理単位間の相互作用におけるものと並列処理単位内部のデータ処理におけるものという2種類に分けることができる．並列処理単位内部のデータ処理におけるエラーは，その結果として並列処理単位間の相互作用におけるエラーを導いてしまう場合もあり得よう．

我々は，上記の観点に基づいて，並列プログラムのテスト・デバッグを二つのレベルに分けて考える．一つは，並列処理単位間の相互作用のみに着目しそれにおけるエラーをテスト・デバッグすることであり，もう一つは，各並列処理単位内部の計算のみに着目しそれにおけるエラーをテスト・デバッグすることである．このように二つのレベルで並列プログラムをテスト・デバッグする方式の主な利点としては，並列・分散処理ソフトウェアのテスト・デバッグの複雑さを減少させることや，段階的詳細化方式で並列・分散処理ソフトウェアを開発する際に枠組だけの段階からプログラムをテスト・デバッグできることなどが挙げられよう．

並列プログラムのテスト・デバッグの以上のような方法論を確立するために，まず，並列プログラムの挙動を並列処理単位間の相互作用と並列処理単位内部の計算とにどの程度はっきり分けることができるか，並列処理単位間の相互作用のみに着目し並列プログラムをテスト・デバッグするにはどんなテスト・デバッグ手段が必要でありかつ有効であるかという二つの問題に答えなければならない．

Ada は，大規模実時間ソフトウェアを記述する共通言語として設計されたプログラミング言語である[1]．Ada では，並列処理単位（タスクと呼ぶ）を明示的に記述し，タスク間にランデブーと呼ぶ同期型通信機構を導入している．また，実時間システムにとって重要な概念である遅延や，時間切れや，非決定的処理や，例外処理を明示的に記述する手段を提供している．その結果，Ada は強力かつ複雑な言語になっており，Ada 並列プログラムのテスト・デバッグが相当難しくなった．したがって，Ada 並列プログラムのテスト・デバッグの手段は，Ada プログラミング支援環境[2]（APSE と略す）が提供しなければならない機能の一つになる[3),4)]．

我々は，並列処理単位間の相互作用のみに着目し並

† Development and Practical Applications of EDEN—An Event-Driven Execution Monitor for Concurrent Ada Programs by JINGDE CHENG, KEIJIRO ARAKI and KAZUO USHIJIMA (Department of Computer Science and Communication Engineering, Faculty of Engineering, Kyushu University).

†† 九州大学工学部情報工学科

列プログラムをテスト・デバッグする手段を提供することを目指し，かつ，Ada プログラミング支援環境の要求仕様 STONEMAN[2] に応じて，Ada 並列プログラムの事象駆動型実行モニタ EDEN を開発した[5]．

EDEN の主な機能は，Ada 並列プログラムの実行を監視しそのタスキング挙動に関する情報を収集，保存してから利用者の要求に応じて報告することである．そのほか，EDEN は，収集した情報に基づいて被監視プログラムの実行中に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（タスキングにおける通信デッドロックという）を自動的に検出しその原因を報告する．EDEN は，主に Ada 並列プログラムのタスク間相互作用におけるエラーを検出しその発生場所を特定するテスト・デバッグ支援ツールとして使うことができる．

本論文では，EDEN の設計思想，基本機能，実現技法および応用について述べる．以下，2章では，EDEN の概要を述べる．3章では，Ada 並列プログラムを監視するために解決しなければならない問題を提示し，それらに対する我々の解決法を述べる．4章では，EDEN の実現を簡単に報告する．5章では，EDEN の応用について述べる．6章では，EDEN が通信デッドロックを自動的に検出する例を示す．

## 2. EDEN システムの概要

EDEN システムは，Ada で記述しており，Data General 社の ECLIPSE MV/10000 上の Ada 処理系 ADE[6] の上で実現した．EDEN は，前処理部と実行時モニタという二つの部分からなる．

### 2.1 処理の概要

EDEN のモニタリングの原理は，次のとおりである．前処理部によって，被監視 Ada 並列プログラム P を Ada プログラム P' に変換する．P' は，その実行中に，P のタスキングに関する事象が生起すると，その事象に関する情報をランデブーによる通信によって実行時モニタへ伝達する．実行時モニタは，収集した情報を保存してから利用者の要求に応じて報告する．

Ada 並列プログラムの実行を EDEN を用いて監視する過程を図1に示す．利用者が EDEN を使用するには，被監視プログラムを格納するテキスト・ファイル名を EDEN に知らせればよい．EDEN は，まず被監視プログラムのソース・テキストを前処理してテキスト中でタスキング事象の生起に対応する場所に実行時モニタと通信するためのエントリ呼び出し文を挿入する．次に，変換後のプログラムを翻訳し，既に分離翻訳してある実行時モニタや他のライブラリ単位と共にリンクし実行可能形式のプログラムを生成する．利用者は，プログラムが実行を開始した後，被監視プログラムのタスキング挙動について EDEN の実行時モニタに質問することができる．

### 2.2 テスト・デバッグ機能

EDEN は，被監視 Ada 並列プログラムのテスト・デバッグをする利用者に次のような機能を提供する．

（1） 只今あるいは特定の時点におけるタスキング状態のスナップショットを報告する．利用者は，特定のタスクか，特定のタスクに依存するすべてのタスクか，プログラム中のすべてのタスクかを指定することができる．タスキング状態のスナップショットには，指定したタスクあるいは確立された (elaborated) すべてのタスクにおける，ソース・テキスト上の名前，一意的内部名（3.2 節で詳述），状態（3.1 節で詳述）が含まれる．もしタスクが通信状態にあれば，通信の準備ができたエントリの名前，あるいは，通信に用いているエントリの名前と相手タスクの名前も含まれる．

（2） 只今あるいは特定の時点におけるエントリ待ち行列状態のスナップショットを報告する．利用者は，特定のエントリか，特定のタスクのすべてのエントリか，プログラム中のすべてのエントリかを指定することができる．エントリ待ち行列状態のスナップショットには，指定したあるいはすべてのエントリ待ち行列中で待機しているタスクとそれらの順序とが含まれる．

（3） タスキング状態あるいはエントリ待ち行列状態に関する条件を利用者が指定しその条件が成立すれば上記の(1)あるいは(2)を報告する．

（4） タスクとそのマスタとの依存関係を報告する．利用者は，特定のタスクか，プログラム中のすべてのタスクかを指定することができる．

（5） 被監視プログラムの実行を中断する．

（6） タスクの状態変化の履歴を報告する．利用者は，タスクの確立から只今までか，特定の時期までかを指定することができる．また，特定のタスクか，特定のタスクに依存するすべてのタスクか，プログラム中のすべてのタスクかも指定することができる．

（7） エントリ待ち行列の状態変化の履歴を報告する．利用者は，エントリの確立から只今までか，特定

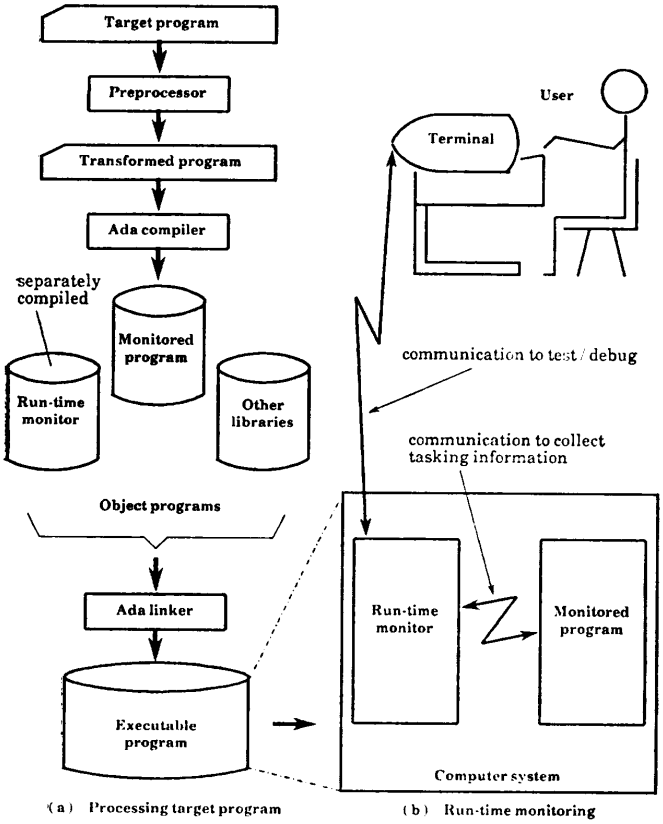


**図 1** EDEN を用いたモニタリング過程
Fig. 1 Monitoring process with EDEN.

の時期までかを指定することができる．また，特定のエントリか，特定のタスクのすべてのエントリか，プログラム中のすべてのエントリかも指定することができる．

（8） タスキング挙動の履歴を外部ファイルに保存する．

（9） 被監視プログラムの実行中に起こる通信デッドロックを自動的に検出し，通信デッドロックに陥ったタスクについて，そのソース・テキスト上の名前，一意的内部名，状態，通信中に用いているエントリ，マスタとの依存関係およびデッドロックの原因を報告する．

これらの機能の大部分は，Ada プログラミング支援環境の要求仕様 STONEMAN[2] 中の動的解析ツールに関する要求のうちで，タスキングに関連するものに相当する．利用者は，EDEN を用いて対象プログラムの実際のタスキング挙動を観察し，それと期待する挙動とを比較することができる．これによって，利用者は，プログラム中のエラーを発見したり除去したりすることができる．

## 3. モニタリング機構

Ada 並列プログラムのタスキング挙動を監視するためには，次の三つの問題を解決しなければならない．

（1） タスキング挙動を厳密にどう定義するか．そのうちで何を監視すべきか，かつ，ソース・テキストのレベルで何を監視することができるか．

（2） タスキング挙動に関する情報を収集する際にタスクをどのように特定するか．

（3） 監視する際に実行モニタの動作から被監視プログラムのタスキング挙動への影響を最小限に止めるにはどうすればよいか．

以下，これらがなぜ問題になるのか，我々がこれらの問題をどのように解決したかについて述べる．

### 3.1 監視の対象

Ada 並列プログラムのタスキング挙動を監視するためには，まず，監視の対象を明確にしなければならない．利用者はプログラムのタスキング挙動に関して十分な情報を得たいであろう．また，実行モニタが提供する情報は，ソース・テキスト上の宣言文と実行文とに簡単に対応できることが望ましい．我々は，この認識に基づいて，Ada 並列プログラムの実行中に生起しかつソース・テキスト上の宣言文と実行文とに対応できるタスキング挙動における原子的動作（タスキング事象と呼ぶ）を，監視の基本対象とすることにした．

Ada 基準文法書[1]のタスクに関する記述を吟味した結果，我々は，タスキング挙動に関する原子的動作として 33 通りを認め，そのそれぞれに以下に列挙する名前（タスキング事象名と呼ぶ）を付けた．「確立開始」，「確立完了」，「起動開始」，「依存タスク宣言の確立開始」，「依存タスク宣言の確立完了」，「起動完了」，「実行開始」，「割り当て子の評価により依存タスクの生成」，「割り当て子の評価により依存タスクの生成完了」，「単純なエントリ呼び出し」，「条件付きエントリ呼び出し」，「時限エントリ呼び出し」，「エントリ受付」，「単純な選択」，「条件付き選択」，「時限選択」，「終了選択」，「エントリ呼び出しの取消し」，「選択の取消し」，「ランデブー開始」，「再開」，「失敗させる」，「失敗させられた」，「ブロック起動開始」，「ブロック起動完了」，「ブロック実行開始」，「ブロック実行完

了」,「ブロック終了した」,「起動中例外発生」,「実行中例外発生」,「通信中例外発生」,「完了」,「終了」. これらの意味については,紙面の都合で説明を省略するので Ada 基準文法書[1]または参考文献 7) を参照されたい.

Ada 並列プログラムの実行中に生起しうるタスキング事象を 8 項組 (T, N, To, E, Me, B, Ma, t) で抽象的に定義する. ここで, Tはこの事象を生起するタスクの一意的名前を表す. Nはタスキング事象名を表す. To は, タスク T が呼び出すタスクの一意的名前か, タスク T が失敗させるタスクの一意的名前か, あるいは⊥ (無定義) かを表す. E は, 通信のためのエントリの一意的名前の集合かあるいは⊥かを表す. Me は, 通信する際に伝達するメッセージかあるいは⊥かを表す. B は, タスクが実行するブロックの一意的名前か, タスクが呼び出す副プログラムの一意的名前か, あるいは⊥かを表す. Ma は, タスクが直接に依存するマスタかあるいは⊥かを表す. t は, この事象が生起する時点を表す.

タスキング事象という抽象概念に基づいて, タスクの状態を二つの相続いて起こるタスキング事象で定義することができる. 例えば, タスクは「起動完了」という事象の生起から「実行開始」という事象の生起までの間「実行待機」という状態になると定義することができる. このようなタスク状態は 20 通りあり, それぞれに以下に列挙する名前 (タスク状態名と呼ぶ) を付けている.「確立中」,「確立した」,「起動中」,「依存タスク宣言の確立中」,「実行待機」,「割り当て子の評価により依存タスクの生成中」,「単純な呼び出しによる待機」,「時限呼び出しによる待機」,「受付による待機」,「単純な選択待機」,「時限選択待機」,「終了選択待機」,「ランデブーによる待機」,「異常」,「ブロック起動中」,「ブロック実行待機」,「ブロック完了した」,「完了した」,「終了した」,「内部処理」[1),7)]. これらの意味の説明は紙面の都合で省略する.

このように, Ada 並列プログラム中のタスクの挙動を, タスキング事象のようなタスク間の相互作用に関するものと「内部処理」状態のようなタスク内部の計算に関するものとに分けて考えることができる. Ada 並列プログラム全体のタスキング挙動は, 各タスクの実行中に生起したすべてのタスキング事象の集合とみなすことができる. この集合に基づいて, プログラム中のどのタスクがいつ何をしたか, かつ, その時に他のタスクとどのように関係しているかを調べることができる.

EDEN は, 被監視プログラムの実行中に生起するタスキング事象を監視の基本対象とする. タスキング事象における情報を収集する実行時モニタ中のタスク (情報収集部と呼ぶ) は, 各種のタスキング事象に対応して一つのエントリを持つ. 実行時モニタは, それらのエントリを通じて被監視プログラムと通信したり, 生起したタスキング事象に関する情報を解析したりすることによって, プログラムのタスキング挙動を把握する.

### 3.2 タスクの特定方法

以上述べたように, EDEN は, 被監視プログラムの各タスクにその実行中に生起したタスキング事象に関する情報を EDEN の実行時モニタに報告させることによって, そのプログラムのタスキング挙動を把握する. したがって, タスキング事象 (すなわち, 8 項組 (T, N, To, E, Me, B, Ma, t)) に関する報告には, どのタスク (すなわち, T) にこの事象が生起したかという情報を含まなければならない. また, 報告するタスクがこの事象によってどのタスク (すなわち, To) と関係するかとか, 報告するタスクのマスタ (すなわち, Ma) がどれであるかとかいった情報を含まなければならない場合もある. したがって, 生起したタスキング事象を報告するタスクは, 自分自身や, 自分のマスタや, 関係する他のタスクを特定できなければならない. 以下, それぞれの場合に分けて述べる.

まず, 被監視プログラム中の各タスクは, EDEN の実行時モニタに自分が「どれ」であるかを報告するために, 自分自身を実行時に特定することが必要である. Ada では, タスクを型 (タスク型と呼ぶ) として静的に宣言し, 一つのタスク型に対して実行時に多数のタスクを生成することができる. しかし, タスクが自分自身を実行時に特定できる手段は何も提供されていない. したがって, EDEN は何らかの手段でタスクにそれ自身が参照できる一意的名前を実行時に付けなければならない. タスクの一意的内部名とは, タスクがその起動から終了までの生涯に自分の本体の内部で参照できかつ他のタスクと区別できる「名前」である[5),8)].

ここでは, 我々が EDEN に用いた方法の概略を示す. Ada 並列プログラム中のタスクに一意的名前を提供するタスク名前サーバをパッケージとしてプログラム・ライブラリに導入する. そのパッケージ TASK_NAME_SERVER を図 2 に示す. 対象プロ

```
package TASK_NAME_SERVER is
  MAXIMAL_NUMBER_OF_TASKS : constant POSITIVE := 100;
  type INTERNAL_ID_OF_TASK is private;
  function GET_INTERNAL_TASK_ID return INTERNAL_ID_OF_TASK;
    :
private
  type INTERNAL_ID_OF_TASK is range 0..MAXIMAL_NUMBER_OF_TASKS;
    :
end TASK_NAME_SERVER;

package body TASK_NAME_SERVER is
  task NAME_SERVER is
    entry GET_A_NEW_TASK_ID (ID : out INTERNAL_ID_OF_TASK);
  end NAME_SERVER;
  function GET_INTERNAL_TASK_ID return INTERNAL_ID_OF_TASK is
    ID : INTERNAL_ID_OF_TASK;
  begin
    NAME_SERVER.GET_A_NEW_TASK_ID (ID);
    return ID;
  end GET_INTERNAL_TASK_ID;
    :
  task body NAME_SERVER is
    NUMBER_OF_TASKS_OVERFLOW : exception;
    TASK_ID_COUNTER : INTERNAL_ID_OF_TASK := 0;
  begin
    loop
      select
        accept GET_A_NEW_TASK_ID (ID : out INTERNAL_ID_OF_TASK) do
          if TASK_ID_COUNTER = MAXIMAL_NUMBER_OF_TASKS then
            raise NUMBER_OF_TASKS_OVERFLOW;
          end if;
          TASK_ID_COUNTER := TASK_ID_COUNTER + 1;
          ID := TASK_ID_COUNTER;
        end GET_A_NEW_TASK_ID;
      or
        terminate;
      end select;
    end loop;
  end NAME_SERVER;
    :
end TASK_NAME_SERVER;
```

**図 2** パッケージ TASK_NAME_SERVER
Fig. 2 Package TASK_NAME_SERVER.

グラムは，with 節によってタスク名前サーバを参照し，各タスク型の本体の宣言部の先頭にタスク名を格納する変数 TASK_ID を宣言しその変数の初期値設定として一意的内部名を与える関数副プログラム GET_INTERNAL_TASK_ID を呼び出す．これによって，各タスクは実行時に起動すると TASK_ID の確立によって自分の名前を持つ．それからタスクの終了までに，タスク自身が TASK_ID に格納されている内部名を参照することができる．タスク内部名の一意性は，一意的内部名を与える関数副プログラム GET_INTERNAL_TASK_ID がタスク名前サーバ中の名前管理タスクの同一のエントリを呼び出してタスク名をもらうことによって保証される．

EDEN の前処理部は，対象プログラムのソース・テキスト中の各タスク型の本体の宣言部の先頭に，タスクの一意的内部名を格納する変数 TASK_ID の宣言とその初期値設定とを挿入する．そのためのプログラム変換規則を**図 3** に示す．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

```
Pattern:
task body <id> is
  [ <declarative part> ]
begin
```

---

```
Replacement:
task body <id> is
  TASK_ID : INTERNAL_ID_OF_TASK :=
    GET_INTERNAL_TASK_ID;
  [ <declarative part> ]
begin
```

**図 3** タスクに一意的内部名を付けるためのプログラム変換規則
Fig. 3 Program transformation rule for naming tasks.

```
Pattern:
begin
  <statements-1>
  accept <entry simple name>[ <formal part> ][do
    <statements-2>
    [ <text piece-1>
    return <expression>; <text piece-2> ]
    <statements-3>
  end[ <entry simple name> ]];
```

---

```
Replacement:
begin
  <statements-1>
  Tasking_Information_Collector.ACCEPTANCE(
    TASK_ID, CLOCK,
    GET_SIMPLE_NAME("<entry simple name>"));
  accept <entry simple name> [ <formal part> ] do
    Tasking_Information_Collector.RENDEZVOUS_START(
      TASK_ID, CLOCK,
    GET_SIMPLE_NAME("<entry simple name>"));
    [ <statements-2> ]
    [ <text piece-1>
    Tasking_Information_Collector.CONTINUATION(
      TASK_ID, CLOCK,
    GET_SIMPLE_NAME("<entry simple name>"));
    return <expression>; <text piece-2> ]
    [ <statements-3> ]
    Tasking_Information_Collector.CONTINUATION(
      TASK_ID, CLOCK,
    GET_SIMPLE_NAME("<entry simple name>"));
  end[ <entry simple name> ];
```

**図 4** accept 文の実行に関する情報を収集するためのプログラム変換規則
Fig. 4 Program transformation rule for collecting information about accept statement.

文字列でもよい）に対応する．パターン変数の照合には制約がある．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む．

タスクがその一意的内部名を用いて自分自身を特定する例として，accept 文の実行に関する情報を収集するためのプログラム変換規則を**図 4** に示す．そこでは，Tasking_Information_Collector が実行時モニタの情報収集部タスクであり，ACCEPTANCE, RENDEZVOUS_START, CONTINUATION などのエントリを持つ．これらは，それぞれ 2.1 節に述べた「エントリ受付」，「ランデブー開始」，「再開」などのタスキング事象名に対応する．図 4 の変換規則は，

accept 文の先頭と直後，return 文の先頭，および accept 文本体の end の先頭で，実行時モニタの情報収集部の適当なエントリを呼び出す文を挿入することを記述する．これらのエントリのいずれも，図3のプログラム変換規則によって導入されるタスク内部名を格納する変数 TASK_ID をパラメータの一つとする．このように，EDEN の実行時モニタは，被監視 Ada 並列プログラムとの通信によってどのタスクがいつ何をしたかを把握することができる．

次に，タスクのマスタを特定する方法について述べる．被監視プログラム中の各タスタは，EDEN の実行時モニタに自分のマスタが「どれ」であるかを報告するために，自分のマスタを特定できなければならない．そのためには，タスクは自分のマスタの一意的名前を参照できればよい．したがって，EDEN は被監視 Ada 並列プログラム中の各タスクのマスタにその本体の内部で参照できかつ他のマスタと区別できる「名前」を実行時に付けなければならない．これは，実際にタスクに一意的内部名を付けることと類似の問題なので，上述と同様の方法を用いて対象プログラム中のタスクのマスタに一意的名前を実行時に付けることにした．

最後に，タスクが他のタスクを特定する方法について述べる．被監視プログラム中の各タスクは，その実行中に生起したタスキング事象が他のタスクと関係する際に，EDEN の実行時モニタに相手タスクが「どれ」であるかを報告しなければならない．そのためには，相手タスクを特定することが必要である．この場合，相手タスクに付けられた前述の一意的内部名は使えない．なぜなら，一意的内部名というのは，その内部名を付けられたタスク自身の本体の内部でしか参照できないからである．また，相手タスクのソース・テキスト中の識別子も使えない．なぜなら，その識別子は，複数のタスクを表すことがある（例えば，同時に他の複数のタスクによって呼び出された副プログラム中のタスクの場合）ので，EDEN がそれにより該当タスクを一意的に特定することができないからである．したがって，EDEN は，被監視 Ada 並列プログラム中の各タスクにその識別子と同じ有効範囲を持つ「名前」（一意的外部名と呼ぶ）を実行時に付けなければならない．タスクは，他のタスクと関係しようとする際にそのタスクの一意的外部名を EDEN の実行時モニタに報告すればよい．EDEN の前処理部は，対象プログラムのソース・テキスト中の各タスクの宣言に対応して，そのタスクの一意的外部名を格納する変数の宣言とその初期値設定とを挿入する．タスクの一意的内部名を与える方法と同じ方法を用いて，タスクの一意的外部名を与えることができる．

### 3.3 タスキング挙動への影響の削減

並列プログラムの挙動は，その構成要素である並列処理単位間の相対実行速度によって異なることがある．EDEN の実行時モニタの動作は，被監視 Ada 並列プログラムのタスクの実行を遅延させうるので，被監視プログラムのタスキング挙動に影響を与えうる．この影響を完全に除くことは不可能である．なぜなら，実行時モニタが被監視プログラムのタスキングに関する事象を監視するためにその実行を遅延させることはやむをえないからである．したがって，何らかの基準を設定して，それに基づいて上記の影響を最小限に抑えるように努める必要がある．我々は，タスク間相互作用を抽象化した代数構造に基づいてこの問題を考察した．

Ada 並列プログラムの正当性において最も本質的なのは，タスク間相互作用の観点から見れば，タスキングに関する任意の二つの事象の間に成り立つ半順序関係と，タスク間で通信する際に伝達されるメッセージとであろう．プログラマはこれらに基づいてプログラムの正当な挙動を想定していると言える．したがっ

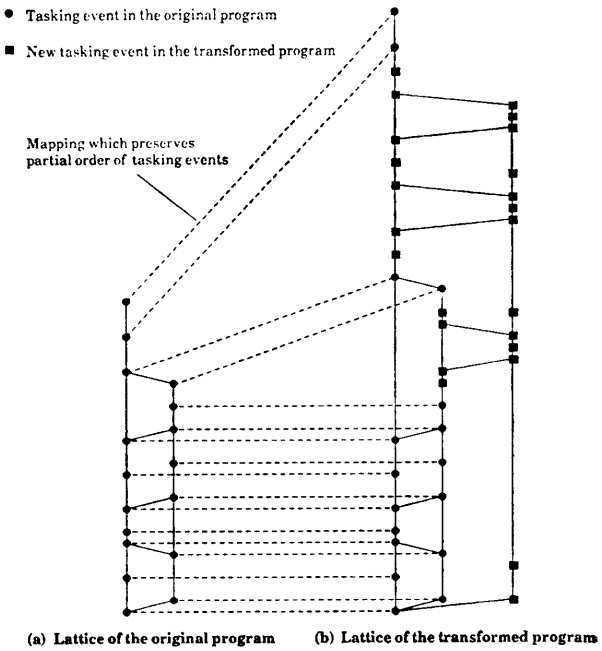


図 5 タスキング事象間の半順序関係を保存する同型写像
Fig. 5 Mapping to preserve the partial order of tasking events.

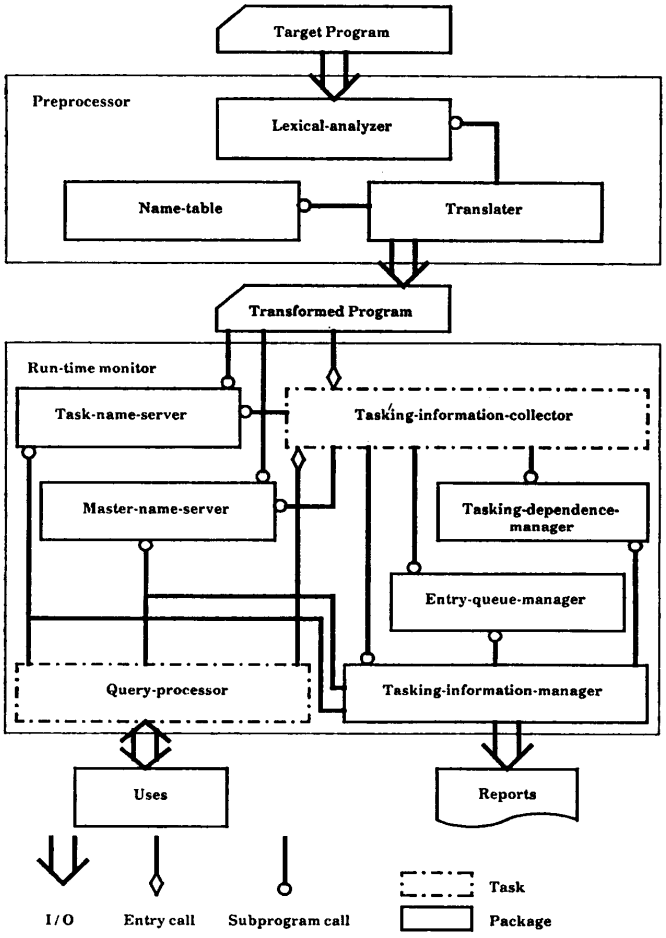


**図 6** EDEN システムの構成
Fig. 6 Overall structure of EDEN system.

て，被監視 Ada 並列プログラムを監視する際に，もし，上記の半順序関係とメッセージとをそのまま観察できれば，プログラマの観点から見れば，実行時モニタの動作から被監視プログラムのタスキング挙動への影響が，プログラマが想定しているプログラムの正当な挙動には及ばないと言える．

我々は，束論を用いて Ada 並列プログラムのタスキング挙動をモデル化した[9]．このモデルによれば，任意の Ada 並列プログラムのタスキング挙動の履歴は，そのプログラムの実行中に起こったタスキング事象からなる一つの束となる．このモデルに基づいて，抽象的な代数構造の観点から，元の被監視プログラムの履歴と監視するために変換したプログラムの履歴との関係を調べることができる．

もし，被監視プログラムの履歴束と監視するために変換したプログラムの履歴束の部分束との間に，タスキング事象間の半順序関係とタスク間で通信する際に伝達されるメッセージとをそのまま保存する同型写像が存在すれば，変換したプログラムのタスキング挙動は，被監視プログラムのタスキング挙動を完全に含み，実行時モニタの動作から被監視プログラムのタスキング挙動への影響が，プログラマが想定しているプログラムの正当な挙動には及ばないと言える．**図 5** に上記の同型写像の例を示す．ここでは，被監視プログラムのタスキング事象を黒い丸で表す．変換したプログラムのタスキング事象の中で，被監視プログラムのタスキング事象を対応するものを同様に黒い丸で表し，監視のために新たに生起する事象を黒い四角形で表す．タスキング事象間の半順序関係を実線で表し，同型写像を破線で表す．

我々は，上記の同型写像が存在するかどうかを，監視のために施すプログラム変換の正しさを保証するための基準とした．EDEN は，被監視プログラムを上記の同型写像が存在するように変換し，利用者に提供する情報の正確性を保証する[5),9)]．

## 4. EDEN システムの実現

EDEN の前処理部は，字句解析部，演算対象名前表管理部，変換部という三つのパッケージからなっており，被監視 Ada 並列プログラムを構文解析しながら一定のプログラム変換規則（約 40 個）に応じて変換する（**図 6** 参照）．

EDEN の実行時モニタは，タスク名前サーバ，マスタ名前サーバ，タスキング依存関係管理部，エントリ待ち行列管理部，情報収集部，情報管理部，質問処理部という七つの主要部分からなる．その中で，情報収集部と質問処理部とは，タスクとして実現し，それ以外は，パッケージとして実現する（図 6 参照）．

タスク名前サーバとマスタ名前サーバとは，それぞれ，タスク名前表とマスタ名前表とを管理する．

タスキング依存関係管理部は，各タスクとそのマスタとの間の依存関係をタスキング依存関係木というデータ構造に記録し管理する．各タスクとマスタとが確立されると，それぞれに対応する節点が関係木の中に生成される．その実行が終わると，それに対応する節点が除去される．

エントリ待ち行列管理部は，各タスクの各エントリに対して一つの行列を生成し，その中に呼び出し側タ

スクの一意的名前を呼び出しの順番に記録し管理する．エントリ呼び出しが取り消されると，対応する呼び出し側タスクの一意的名前は除去される．呼び出し側タスクが異常状態になると，その一意的名前は除去される．

情報収集部は，実行時モニタの中枢部分であり，実行時モニタと被監視 Ada 並列プログラムとの間のインタフェースでもある．被監視プログラムのタスキング挙動に関する情報の収集は，すべて，情報収集部のエントリを通してメッセージ伝達の形式で行う．

情報管理部は，情報収集部が収集した情報を記録し管理する．情報管理部に保存される被監視プログラムのタスキング挙動のデータに対する読み書きの相互排除のために，情報管理部が保存している情報に対する読み書き操作は，情報収集部によってしか行うことができない．

質問処理部は，実行時モニタと利用者との間のメニュー駆動型インタフェースである．

## 5. EDEN システムの応用

EDEN は，主に，Ada 並列プログラムのタスク間の相互作用におけるエラーを検出しその発生場所を特定するテスト・デバッグ支援ツールとして使うことができる．また，EDEN が収集し保存している被監視 Ada 並列プログラムのタスキング挙動に関する情報は，他のテスト・デバッグ支援ツールが利用するともできる．

### 5.1 EDEN による Ada 並列プログラムのテスト・デバッグ方式

まず，EDEN による Ada 並列プログラムのテスト・デバッグ方式を説明することによって，EDEN という支援ツールが Ada 並列プログラムのテスト・デバッグの分野でどのように位置づけられるかを明確にしたい．

与えられた設計仕様が許さない，または，設計仕様が規定していない状況がプログラムの挙動に現れれば，そのプログラムの中にエラーが存在するということにする．

Ada 並列プログラム中に存在しうるエラーを次のような3種類に分類することができる．

（1） 同期エラー：タスク間に，仕様の許さないまたは仕様の規定していない順序でタスキング事象が生起する．

（2） 通信エラー：タスクがメッセージを伝達したり受け取ったりする際，あるいは，タスクが大域変数を更新したり読んだりする際には，仕様の許さないまたは仕様の規定していない状況が存在する．

（3） 計算エラー：タスクの内部で，仕様の許さないまたは仕様の規定していない順序で実行文が実行される．あるいは，タスクがその内部の局所変数を更新したり読んだりする際には，仕様の許さないまたは仕様の規定していない状況が存在する．

なお，計算エラーは，その結果として同期エラーあるいは通信エラーを導いてしまうことがあり得る．

同期エラーと通信エラーとをタスク間の相互作用のレベルにおけるエラーと呼び，計算エラーをタスクの内部のレベルにおけるエラーと呼ぶことにする．このような分類に基づいて，Ada 並列プログラムのテスト・デバッグをタスク間の相互作用のレベルとタスク内部の計算のレベルという二つのレベルに分けて行うこと（2レベル・テスト・デバッグ方式と呼ぶ）を考える．タスク間の相互作用というレベルでは，同期エラーと通信エラーとをテストしデバッグする．タスク内部の計算というレベルでは，計算エラーをテストしデバッグする．

このような2レベル・テスト・デバッグ方式の主な利点として，次の二つを挙げる．

（1） 各テスト・デバッグのレベルで異なる種々のエラーの検出と修正とに専念し，テスト・デバッグの複雑さを減少させることができる．例えば，タスク間の相互作用というレベルでは，同期エラーと通信エラーとを導かない計算エラーは考慮しなくてもよい．

（2） 段階的詳細化方式で並列・分散処理ソフトウェアを開発する際に，各並列処理単位の枠組を先に作成しそれらの間の相互作用を全体の完成に先立ってテスト・デバッグすることができる．

EDEN を用いたテスト・デバッグは，タスク間の相互作用というレベルでのテスト・デバッグである．

### 5.2 通信デッドロックの検出

通信デッドロックとは，Ada 並列プログラムの実行中においてタスク間でランデブーにより通信する際に，一部分あるいはすべてのタスクが通信の待ち合わせによって永遠に待機し封鎖され絶対に動作できない状況をいう．Ada 並列プログラムの実行中において，一部分だけのタスクが通信デッドロックに陥った状況を局所的な通信デッドロックと呼び，すべてのタスクが通信デッドロックに陥った状況を全域的な通信デッドロックと呼ぶ[10),11)]．通信デッドロックは，最も典

型的な同期エラーの一種である．

通信デッドロックにはその発生の原因に応じて起動ブロッキング，循環エントリ呼び出し，依存性ブロッキング，終了ブロッキング，受付ブロッキングなどを挙げることができる[10),11)]．

（1） 起動ブロッキング：タスクAの宣言部に宣言されるタスクBは自分の起動中にAのエントリを呼び出した．Adaの定義[1)]によれば，この場合には，Aは，Bの起動が終わるまで待機しており，本体の実行に入れない．一方，Bは，Aとのランデブーが終わるまで待機している．したがって，Aは，Bが呼び出しているエントリに対応するaccept文を実行できず，Bと共に永遠に待機し封鎖され絶対に動作できない状況になる．

（2） 循環エントリ呼び出し：タスク間のエントリ呼び出し関係が循環になった．すなわち，タスク $T_1$ がタスク $T_2$ のエントリを呼び出して，タスク $T_2$ がタスク $T_3$ のエントリを呼び出して，…，タスク $T_{n-1}$ がタスク $T_n$ のエントリを呼び出して，タスク $T_n$ がタスク $T_1$ のエントリを呼び出した．この場合には，循環の中に含まれるタスクは，どれもランデブーによる通信の待ち合わせによって永遠に待機し封鎖され絶対に動作できない状況になる．循環エントリ呼び出しの極端な例は，自己ブロッキング，すなわち，タスクが自分のエントリを呼び出したことである[1)]．

（3） 依存性ブロッキング：タスクAが実行しているブロック文の中であるいはAが呼び出している副プログラムの中で直接にあるいは間接に生成されたタスクBが，Aのエントリを呼び出した．Adaの定義[1)]によれば，この場合には，Bは，Aとのランデブーが終わるまで待機している．一方，Aは，自分が実行しているブロック文あるいは呼び出している副プログラムが終了しなければ実行を続けることができない．しかし，そのブロック文あるいは副プログラムは，自分が直接にあるいは間接に生成したBが終了してはじめて自分の実行を終了することができる．したがって，Aは，Bが呼び出しているエントリに対応するaccept文を実行できず，Bと共に永遠に待機し封鎖され絶対に動作できない状況になる．

（4） 終了ブロッキング：タスクAは，選択待機文の実行によって待機している．その選択待機文の終了選択肢 (terminate alternative) が開いており (open)，かつ，他のすべての選択肢が閉じている (closed)．しかし，あるタスクBは，Aのエントリに対する単純呼び出しを発行しそのエントリの待ち行列の中で待機している．Adaの定義[1)]によれば，この場合には，Aは，終了選択肢を選択することができないし他のどの選択肢も選択することができないので，Bと共に永遠に待機し封鎖され絶対に動作できない状況になる．

（5） 受付ブロッキング：タスクAは，エントリEに対する呼び出しを受け付けるために待機している．しかし，プログラム中にエントリEを呼び出す文がないか，あるいはAが待機し始めた時点以降プログラム中にエントリEを呼び出す文を実行することが有り得ない．この場合には，Aは，通信の待ち合わせによって永遠に待機し封鎖され絶対に動作できない状況になる．

以上の通信デッドロックのうち受付ブロッキング以外はいずれも，タスクの起動中にも実行中にも発生しうる．

我々は，標識付き有向グラフを用いてタスクとそのマスタとの依存関係およびタスク間のエントリ呼び出し関係をモデル化し，タスキング状態グラフと呼ぶモデルを導入した[5),11)]．そのモデルに基づく検討から，Ada並列プログラムの実行中に起動ブロッキング，自己ブロッキング，循環エントリ呼び出し，依存性ブロッキングが発生するための必要十分条件はそのタスキング状態グラフに有向閉路が存在することであることを示した．

EDENは，収集した情報に基づいて，被監視Ada並列プログラムのタスキング状態グラフを簡単に構築することができる．それに基づいて実行中に発生する起動ブロッキング，自己ブロッキング，循環エントリ呼び出し，依存性ブロッキング，終了ブロッキングを直接に検出することができる．また，EDENは，収集した情報に基づいて，被監視Ada並列プログラム中の確立されたタスクの数と通信待機のため封鎖されたタスクの数とを数えて，全域的な通信デッドロックを直接に検出することができる．受付ブロッキングは，全域的な通信デッドロックの検出によって間接に検出することができる．

### 5.3 他の応用

EDENが収集し保存している被監視Ada並列プログラムのタスキング挙動の履歴に関する情報は，他のテスト・デバッグ支援ツールも利用することができる．例えば以下のような利用を考えることができる．

EDENが提供する情報に基づいて，タスク間の通信のボトルネックを調べることができる．

Ada 並列プログラムをテストする際にそのプログラム中のタスク間の同期序列（ここで，プログラムの実行によって生起するタスキング事象の序列を同期序列と呼ぶ[12]）を必要とする場合がある[13),14)]．EDEN が提供する情報に基づいて，そのような同期序列を生成することができる．

データベースあるいは知識ベースに基づくデバッギング・システムでは，入力の一つとしてプログラムの実行履歴が必要である[15),16)]．APSE の中でそのようなシステムを開発し使用する際に，EDEN が提供する被監視プログラムの実行履歴を入力として利用することができる．

## 6. 例

図 7 に，タスク本体の実行中に依存性ブロッキングが発生する Ada 並列プログラムの例を示す．

このプログラムを実行する際に，タスク T1 が呼び出している副プログラム P1 の中で直接に生成されたタスク T2 が，T1 のエントリ E1 を呼び出すと，T2 は，T1 とのランデブーが終わるまで待機している．一方，T1 は，自分が呼び出している P1 が終了しなければ実行を続けることができない．しかし，P1 は，自分が直接に生成した T2 が終了してはじめて自分の実行を終了することができる．したがって，T1 は，T2 が呼び出しているエントリ E1 に対応する accept 文の実行に入れない．結局，依存性ブロッキングに陥る．

```
procedure DB is
  task T1 is entry E1; end T1;
  procedure P1 is
    task T2;
    task body T2 is
    begin  T1.E1;  end T2;
  begin  null;  end P1;
  task body T1 is
  begin
    P1;  accept E1;
  end T1;
begin  null;  end DB;
```

図 7 依存性ブロッキング通信デッドロックの例
Fig. 7 An example of dependence blocking communication deadlock.

図 8(a) は，EDEN を用いて図 7 のプログラムを

```
*****  EDEN  V:1 L:1 1986-8-5     DATE: 1987- 9- 7    TIME: 21:16: 3  *****

  * At     2.0000000000 sec after start,
    states of all tasks are as follows :

  * Task name           => MAIN_TASK
  * ID of the task      =>  1
  * State of the task   => BLOCK_ACTIVATING
  * Called subprogram   => DB

*****  EDEN  V:1 L:1 1986-8-5     DATE: 1987- 9- 7    TIME: 21:16: 5  *****

  * At     4.4003906250 sec after start,
    states of all tasks are as follows :

  * Task name           => MAIN_TASK
  * ID of the task      =>  1
  * State of the task   => BLOCK_COMPLETED
  * Called subprogram   => DB

  * Task name           => T1
  * ID of the task      =>  2
  * State of the task   => BLOCK_EXECUTION_WAITING
  * Called subprogram   => P1

  * Task name           => T2
  * ID of the task      =>  3
  * State of the task   => ACTIVATING
```

図 8(a) 図 7 のプログラムのタスキング状態のスナップショット
Fig. 8(a) Snapshot of states of tasks in program given by Fig. 7.

```
*****  EDEN  V:1 L:1 1986-8-5     DATE: 1987- 9- 7    TIME: 21:16: 6  *****

  * A dependence blocking will arise in the program.

*****  EDEN  V:1 L:1 1986-8-5     DATE: 1987- 9- 7    TIME: 21:16: 6  *****

  * At     5.2988281250 sec after start,
    dependence relation of tasks in the program is as follows :

  * The task T1         has created, called or executed
    following tasks, subprograms or blocks :

  * Subprogram name => P1

  * The master P1         has created, called or executed
    following tasks, subprograms or blocks :

  * Task name => T2         * ID of the task =>  3


  * The task T2         has not created tasks.

*****  EDEN  V:1 L:1 1986-8-5     DATE: 1987- 9- 7    TIME: 21:16: 6  *****

  * At     5.2988281250 sec after start,
    states of all children of the task T1         are as follows :

  * Master name           => P1

  * Task name             => T2
  * ID of the task        =>  3
  * The parent of the task => P1
  * State of the task     => ENTRY_CALLING
  * Communicating entry   => T1          . E1

  * At     5.2988281250 sec after start,
    the task T2         has not created tasks.
```

図 8(b) 検出された図 7 のプログラム中の依存性ブロッキング
Fig. 8(b) Detected dependence blocking in program given by Fig. 7.

```
*****   EDEN   V:1 L:1 1986-8-5      DATE: 1987- 9- 7    TIME: 21:16: 6   *****

    * A global blocking will arise in the program.


*****   EDEN   V:1 L:1 1986-8-5      DATE: 1987- 9- 7    TIME: 21:16: 7   *****

    * At     5.8007812500 sec after start,
      states of all tasks are as follows :

    * Task name           => MAIN_TASK
    * ID of the task      =>  1
    * State of the task   => BLOCK_COMPLETED
    * Called subprogram   => DB

    * Task name           => T1
    * ID of the task      =>  2
    * State of the task   => BLOCK_COMPLETED
    * Called subprogram   => P1

    * Task name           => T2
    * ID of the task      =>  3
    * State of the task   => ENTRY_CALLING
    * Communicating entry => T1          . E1


*****   EDEN   V:1 L:1 1986-8-5      DATE: 1987- 9- 7    TIME: 21:16: 7   *****

    * At     5.8007812500 sec after start,
      states of all entry queues are as follows :

    * Entry name => T1          . E1
    *          1 => T2
```

**図 8(c)** 検出された図7のプログラム中の全域的な通信デッドロック
Fig. 8(c) Detected global communication deadlock in program given by Fig. 7.

監視した際に，EDEN が報告したタスキング状態のスナップショットを示す．この報告から，次のことが分かる．実行開始後2秒の時点で，副プログラム DB が確立中である．実行開始後4.4秒の時点で，副プログラム DB の実行が既に完了し，タスク T1 がその呼び出している副プログラム P1 の本体の実行開始を待機し，タスク T2 が起動中である．

**図 8(b)**は，依存性ブロッキングが発生する状況に関する EDEN の報告を示す．この報告から，依存性ブロッキングに陥るタスクとそれが呼び出している副プログラムとに関する情報が分かる．実行開始後 5.3 秒の時点で，タスク T1 が副プログラム P1 を呼び出しており，副プログラム P1 がタスク T2 を生成し，タスク T2 がタスク T1 のエントリ E1 を呼び出している．

この依存性ブロッキングが発生した結果として全域的な通信デッドロックが生じる．**図 8(c)**は，これに関する EDEN の報告を示す．ここでは，EDEN がプログラム中のすべてのタスクとすべてのエントリ待ち行列との状態を報告する．

## 7. おわりに

EDEN は，今まで提案され実現されている他の Ada プログラムのためのテスト・デバッグ支援ツール[17)~23)]と比較して，次の特徴を持つ．

（1） EDEN の開発では，Ada 並列プログラムのタスキング挙動をどのように形式的に定義するか，Ada 並列プログラムのタスキング挙動を監視する際にいかにして実行モニタの動作から被監視プログラムのタスキング挙動への影響を最小限に止めるかという，Ada 並列プログラムのためのテスト・デバッグ支援ツールの開発における二つの重要な問題を提示し取り扱った[5),9)]．

（2） EDEN には，Ada タスクに一意的名前を実行時に付ける新方法を採用した．この方法は，起動中のタスクも自分自身を特定することができることや，タスク間の起動開始あるいは実行文系列の実行開始の時間的順序にどんな制限も加えないという二つの特徴を持つ．また，タスクのマスタに一意的名前を実行時に付けることができる．これによって，EDEN は，他のツール[17)~23)]が支援していない Ada タスキング機能(例えば，タスクの起動と起動中の通信，タスクとそのマスタとの依存関係，タスクの失敗，例外処理など）も支援し，適用の対象となる被監視プログラムのクラスが他のツールよりも大きい．

（3） EDEN は，今までに提案され実現されている検出方法または検出ツール[17)~19),24)]が検出できない通信デッドロック（例えば，起動ブロッキング，依存性ブロッキング，終了ブロッキングなど）を自動的に検出しその原因を報告することができる．特に，EDEN は，ほとんどの通信デッドロック（すなわち，受付ブロッキングを除いて）に対して，起こる直前にそれらを検出することができる．これについて，他のツール[17)~19)]は，ほとんど局所的な通信デッドロックに対して，その結果として大域的なデッドロックに導いてしまった時にしかそれらを検出することができない．しかし，局所的な通信デッドロックは必ずしも大域的なデッドロックにまで至らない．他のツール[17)~19)]はこのような局所的な通信デッドロックを検出することができない．

我々は，現在，Ada 並列プログラムのテスト・デバッグに対して，EDEN が提供している機能の必

要性，有効性を実際の適用を通じて確認中である．EDEN によって変換されたプログラムの実行時間は，元のプログラムの実行時間と比べてどれほど増加するか，また，それが大規模プログラムの実用上に問題になるかについても確認中である．更に，EDEN の機能を一層強化し実時間処理システム，分散処理システムのテスト・デバッグへの適用について調べている．

**謝辞** EDEN システムの前処理部の実現において御協力頂いた九州大学大学院情報工学専攻白木原敏雄氏に謝意を表する．

## 参 考 文 献

1) United States Department of Defense: Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815 A) (Jan. 1983).
2) United States Department of Defense: "STONEMAN", Requirements for Ada Programming Support Environment (Feb. 1980).
3) Fairley, R. E.: Ada Debugging and Testing Support Environments, *ACM SIGPLAN Notices*, Vol. 15, No. 11, pp. 16-25 (1980).
4) Taylor, R. N. and Standish, T. A.: Steps to an Advanced Ada Programming Environment, *IEEE Trans. Softw. Eng.*, Vol. SE-11, No. 3, pp. 302-310 (1985).
5) Cheng, J., Araki, K. and Ushijima, K.: Event-Driven Execution Monitor for Ada Tasking Programs, *IEEE Proc. of COMPSAC 87*, pp. 381-388 (1987).
6) Data General Corp.: Ada Development Environment (ADE) (AOS/VS) User's Manual (1984).
7) Burns, A., Lister, A. M. and Wellings, A. J.: A Review of Ada Tasking, *LNCS*, Vol. 262, Springer-Verlag, New York (1987).
8) 程 京徳，牛島和夫：Ada タスクに一意的名前を実行時に付ける方法，情報処理学会論文誌，Vol. 29, No. 12, pp. 1208-1212 (1988).
9) Cheng, J. and Ushijima, K.: Partial Order Transparency in Monitoring Tasking Behaviors of Concurrent Ada Programs, *Proc. of ICS '88*, to appear (1988).
10) Cheng, J., Araki, K. and Ushijima, K.: Tasking Communication Deadlocks in Concurrent Ada Programs, *ACM Ada Lett.*, Vol. 8, No. 5, pp. 61-70 (1988).
11) Cheng, J. and Ushijima, K.: Detecting Tasking Communication Deadlocks in Concurrent Ada Programs, *IEEE Proc. of ICSC '88*, to appear (1988).
12) Tai, K-C.: On Testing Concurrent Programs, *IEEE Proc. of COMPSAC 85*, pp. 310-317 (1985).
13) Tai, K-C. and Obaid, E. E.: Reproducible Testing of Ada Tasking Programs, *Proc. of the IEEE Second International Conference on Ada Applications and Environments*, pp. 69-79 (1986).
14) Tai, K-C., Carver, R. H. and Obaid, E. E.: A Methodology for Testing Concurrent Ada Programs, *Proc. of Joint Ada Conference*, pp. 459-464 (1987).
15) Powell, M. L. and Linton, M. A.: A Database Model of Debugging, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, *ACM Softw. Eng. Notes*, Vol. 8, No. 4, *SIGPLAN Notices*, Vol. 18, No. 8, pp. 67-70 (1983).
16) Seviora, R. E.: Knowledge-Based Program Debugging Systems, *IEEE Softw.*, Vol. 4, No. 3, pp. 20-32 (1987).
17) German, S. M.: Monitoring for Deadlock and Blocking in Ada Tasking, *IEEE Trans. Softw. Eng.*, Vol. SE-10, No. 6, pp. 764-777 (1984).
18) Helmbold, D. and Luckham, D. C.: Runtime Detection and Description of Deadness Errors in Ada Tasking, *ACM Ada Lett.*, Vol. 4, No. 6, pp. 60-72 (1984).
19) Helmbold, D. and Luckham, D.: Debugging Ada Tasking Programs, *IEEE Softw.*, Vol. 2, No. 2, pp. 47-57 (1985).
20) Data General Corp.: Ada Source Debugger (ADE) (AOS/VS) Reference Manual (1984).
21) LeDoux, C. H. and Parker, D. S. Jr.: Saving Traces for Ada Debugging, Ada in Use, pp. 97-108, *Proc. of the Ada International Conference*, Paris (May 1985).
22) Di Maio, A., Ceri, S. and Reghizzi, S. C.: Execution Monitoring and Debugging Tool for Ada Using Relational Algebra, Ada in Use, pp. 109-123, *Proc. of the Ada International Conference*, Paris (May 1985).
23) Mauger, C. and Pammett, K.: An Event-driven Debugger for Ada, Ada in Use, pp. 109-123, *Proc. of the Ada International Conference*, Paris (May 1985).
24) Zhou, B., Yeh, R. T. and Ng, P. A.: Principle of Deadlock Detection in Ada Programs, *IEEE Proc. of the 6th ICDCS*, pp. 572-579 (1986).

(昭和 63 年 5 月 25 日受付)
(昭和 63 年 11 月 14 日採録)

**程　京徳**（正会員）

1952年生．1982年中国清華大学計算機科学・技術系卒業．同年同大学助手．1986年九州大学大学院工学研究科修士課程情報工学専攻修了．現在同博士後期課程在学中．ソフトウェア仕様記述，プログラミング方法論，ソフトウェア開発支援環境などに興味を持つ．1986年情報処理学会学術奨励賞．日本ソフトウェア科学会会員．

**荒木啓二郎**（正会員）

昭和51年九州大学工学部情報工学科卒業．昭和53年同大大学院修士課程修了．同年九州大学工学部情報工学科助手．昭和59年同大助教授．昭和63年8月より西独パッサウ大学にて在外研究．プログラミング言語，形式的仕様記述，ソフトウェア開発方法論等に興味をもつ．工学博士．「プログラミング言語 Ada」(共訳，サイエンス社)．電子情報通信学会，ソフトウェア科学会，ACM各会員．

**牛島　正夫**（正会員）

1937年生．1961年東京大学工学部応用物理学科（数理工学）卒業．1963年同大学院修士課程修了．同年九州大学中央計数施設勤務．1977年九州大学工学部情報工学科教授（計算機ソフトウェア講座担当），現在に至る．1986年4月から九州大学情報処理教育センター長を兼務．工学博士．著書「Fortran プログラミングツール」(産業図書）ほか，日本ソフトウェア科学会，電子情報通信学会，ACM 各会員．