

# Dynamic tasks verification with QUASAR

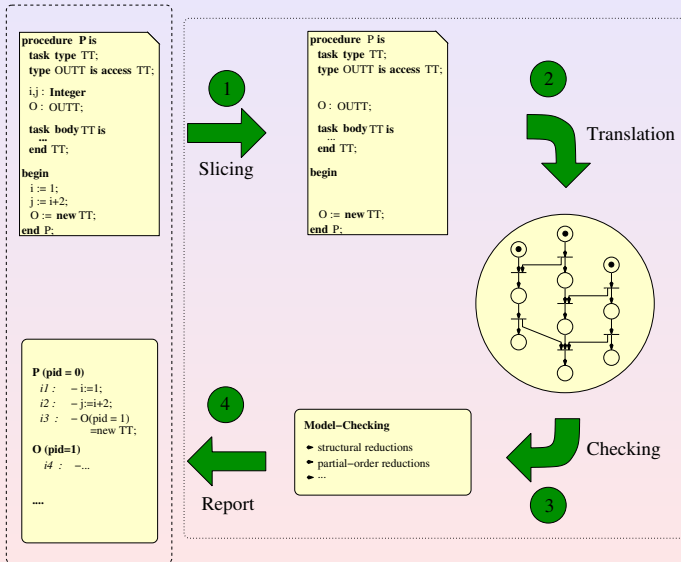
S. Evangelista, C. Kaiser, C. Pajault\*, J-F. Pradat-Peyre and P. Rousseau

 Cedric CNAM - Paris 

10th International Conference on Reliable Software Technologies Ada-Europe  
2005



# Presentation of QUASAR



# Dynamic task needs

## Motivation

A version of [QUASAR](#) that checks static programs already exists. The verification of static concurrent programs is easier than the verification of dynamic ones. But a large number of critical applications uses dynamic tasks. Thus, we decided to extend [QUASAR](#) to take dynamic task into account.

## Dynamic task handling problems

- 1 Combinatorial explosion
- 2 Modeling dependences between tasks



## Dynamic tasks identification



# Task identification

## The dynamic *id* problem

### Task identification in QUASAR

In **QUASAR**, each task is identified by a **unique** *id* value added in the task token.  
In the **static version** of **QUASAR**, *id* are statically computed as we only deal with static tasks.

To deal with **dynamic tasks** it is necessary to dynamically generate unique *ids* while limiting – as much as possible – the combinatorial explosion .



# Task identification

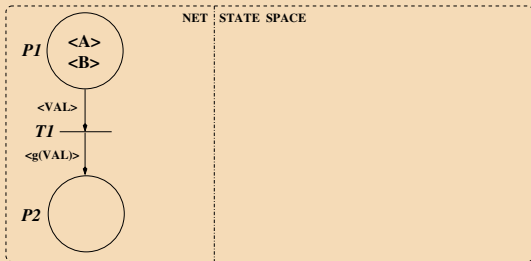
## The dynamic *id* problem

### Task identification in QUASAR

In **QUASAR**, each task is identified by a **unique** *id* value added in the task token.  
In the **static version** of **QUASAR**, *id* are statically computed as we only deal with static tasks.

To deal with **dynamic tasks** it is necessary to dynamically generate unique *ids* while limiting – as much as possible – the combinatorial explosion .

### Task naming problem



# Task identification

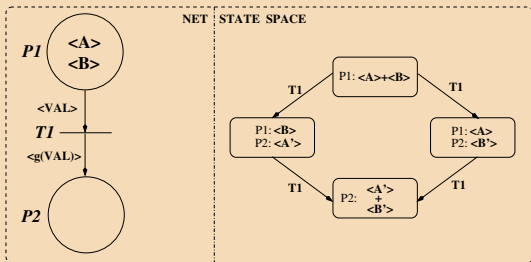
## The dynamic *id* problem

### Task identification in QUASAR

In **QUASAR**, each task is identified by a **unique** *id* value added in the task token.  
In the **static version** of **QUASAR**, *id* are statically computed as we only deal with static tasks.

To deal with **dynamic tasks** it is necessary to dynamically generate unique *ids* while limiting – as much as possible – the combinatorial explosion .

### Task naming problem



# Task identification

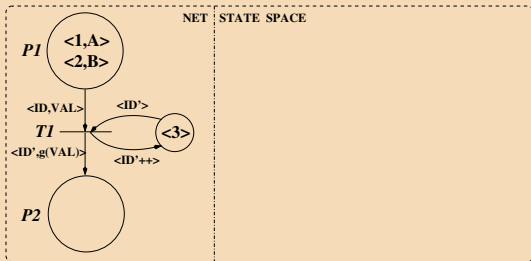
## The dynamic *id* problem

### Task identification in QUASAR

In **QUASAR**, each task is identified by a **unique** *id* value added in the task token.  
In the **static version** of **QUASAR**, *id* are statically computed as we only deal with static tasks.

To deal with **dynamic tasks** it is necessary to dynamically generate unique *ids* while limiting – as much as possible – the combinatorial explosion .

### Task naming problem



# Task identification

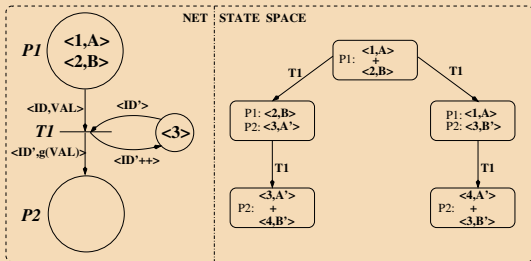
## The dynamic *id* problem

### Task identification in QUASAR

In **QUASAR**, each task is identified by a **unique** *id* value added in the task token.  
In the **static version** of **QUASAR**, *id* are statically computed as we only deal with static tasks.

To deal with **dynamic tasks** it is necessary to dynamically generate unique *ids* while limiting – as much as possible – the combinatorial explosion .

### Task naming problem



# How to handle dynamic naming (1/2)

## Computing a one-to-one function

### Idea

The combinatory induced by task naming is the result of the possibility to give two different *ids* to the same task while considering two different possible executions.

☛ in order to limit combinatory, when a task is spawned the best case would be to always give the same *id* to that task for every possible execution.

### One-to-one function

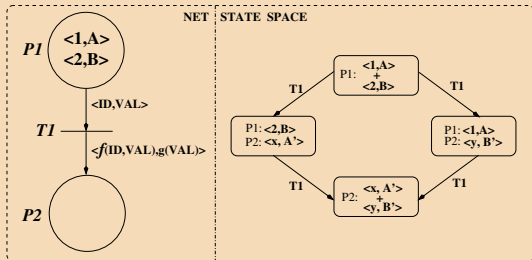
**Idea** : The naming of a task is given by the result of a **one-to-one function**.  
When spawning a task, the new *id* is the result of a unique one-to-one function  $f$  calculated when firing the spawning transition.



# How to handle dynamic naming (2/2)

## Computing a one-to-one function

### Evaluation on a simple example



### Conclusion

- 😊 efficient (no useless combinatory).
- 😞 in the general case, some more information are needed to compute the one-to-one function (for example : a value greater or equal to the maximum number of tasks that may be spawned in the program).



# How to handle dynamic naming (2/2)

## Computing a one-to-one function

### Evaluation on a simple example

Evaluation on a simple example with only 3 tasks.

<i>id</i> allocation mechanisms	global <i>id</i> server	local <i>id</i> server	one-to-one function
Number of generated states	493	205	125
Number of different name sequences	6	2	1

### Conclusion

- 😊 efficient (no useless combinatory).
- 😞 in the general case, some more information are needed to compute the one-to-one function (for example : a value greater or equal to the maximum number of tasks that may be spawned in the program).



## Modeling dependences and synchronizations between blocks



# Task dependences

## Definition

The **master** of a task is defined by the two following rules :

- The father of an **elaborated** task instance is also the **direct master** of the **elaborated** task instance.
- The instance that declares the **access type** of an **allocator expression** is the **direct master** of all **allocated tasks** instances that are created by evaluating the expression.

## Life cycle and synchronizations

A task (or block) life cycle consists in four step :

- **activation** ➡ father must wait for activation of the elaborated tasks it has spawned.
- **execution**
- **completion** ➡ to begin termination, each block has to wait for all the tasks it is the master to terminate.
- **termination**



# New patterns

## Our work

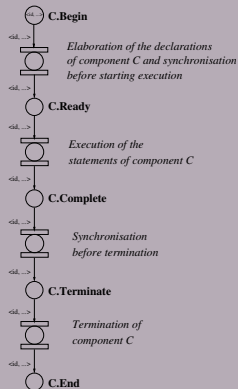
- We had to enrich our block model in order to take block life cycle into account.
- We had then to define new patterns to consider hierarchy between tasks and deal with synchronizations.

## Idea

We have chosen to implement differently allocated and elaborated tasks (and more generally, any kind of block).

- A **master** knows all its **elaborated** tasks since it has directly spawned them.
- A **master** does not know neither which **allocated** tasks it is the master, nor the number of such tasks. **But** these allocated tasks know the task type of their master  $\Rightarrow$  they just have to catch the right master of that type and notify it of their creation and termination.

## Block model



## Our solution

- one model for each block body ,
- one more model for each allocated task type declared in the program
- ➡ one sub-net for each access type and for each body.

## Example

```
procedure ALLOC is
  task type TT;
  type OUTT is access TT;

  procedure P is
    type INTT is access TT;
    O : OUTT;
    I : INTT;
  begin
    O := new TT;
    I := new TT;
    ...
  end P;

  task body TT is
    ...
  end TT;

  T : TT;
begin
  P;
  ...
end ALLOC;
```



## Our solution

- one model for each block body ,
- one more model for each allocated task type declared in the program
- ➡ one sub-net for each access type and for each body.

## Example

```

procedure ALLOC is
  task type TT;
  type OUTT is access TT;

  procedure P is
    type INTT is access TT;
    O : OUTT;
    I : INTT;
  begin
    O := new TT;
    I := new TT;
    ...
  end P;

  task body TT is
    ...
  end TT;

  T : TT;
begin
  P;
  ...
end ALLOC;
    
```



## Our solution

- one model for each block body ,
- one more model for each allocated task type declared in the program
- ⇒ one sub-net for each access type and for each body.

## Example

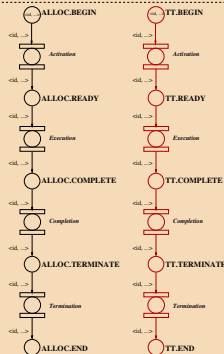
```

procedure ALLOC is
  task type TT;
  type OUTT is access TT;

  procedure P is
    type INTT is access TT;
    O : OUTT;
    I : INTT;
  begin
    O := new TT;
    I := new TT;
    ...
  end P;

  task body TT is
    ...
  end TT;

  T : TT;
begin
  P;
  ...
end ALLOC;
    
```



## Our solution

- one model for each block body ,
- one more model for each allocated task type declared in the program
- ➡ one sub-net for each access type and for each body.

## Example

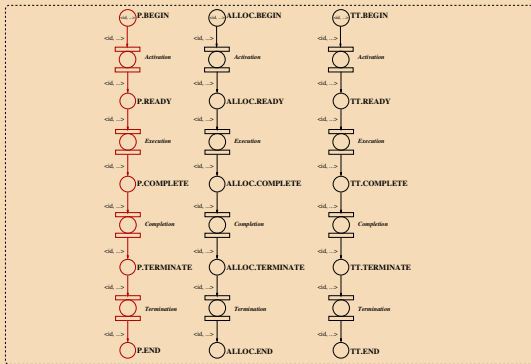
```

procedure ALLOC is
  task type TT;
  type OUTT is access TT;

  procedure P is
    type INTT is access TT;
    O : OUTT;
    I : INTT;
  begin
    O := new TT;
    I := new TT;
    ...
  end P;

  task body TT is
    ...
  end TT;

  T : TT;
begin
  P;
  ...
end ALLOC;
    
```



## Our solution

- one model for each block body ,
- one more model for each allocated task type declared in the program
- ➡ one sub-net for each access type and for each body.

## Example

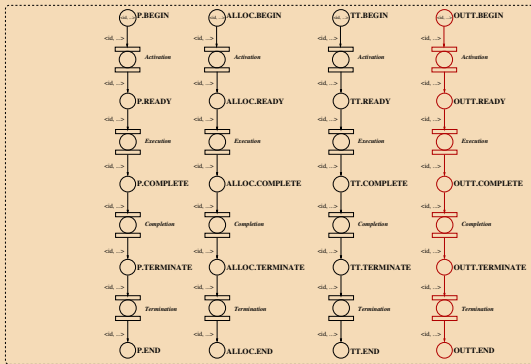
```

procedure ALLOC is
  task type TT;
  type OUTT is access TT;

  procedure P is
    type INTT is access TT;
    O : OUTT;
    I : INTT;
  begin
    O := new TT;
    I := new TT;
    ...
  end P;

  task body TT is
    ...
  end TT;

  T : TT;
begin
  P;
  ...
end ALLOC;
    
```



## Our solution

- one model for each block body ,
- one more model for each allocated task type declared in the program
- ➡ one sub-net for each access type and for each body.

## Example

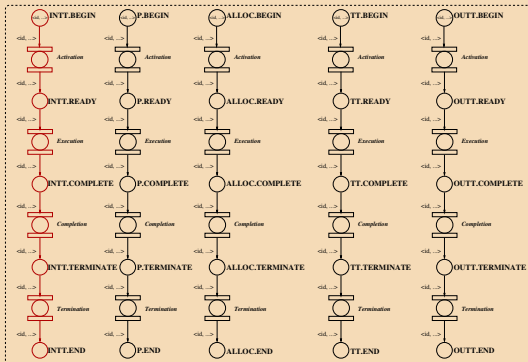
```

procedure ALLOC is
  task type TT;
  type OUTT is access TT;

  procedure P is
    type INTT is access TT;
    O : OUTT;
    I : INTT;
  begin
    O := new TT;
    I := new TT;
    ...
  end P;

  task body TT is
    ...
  end TT;

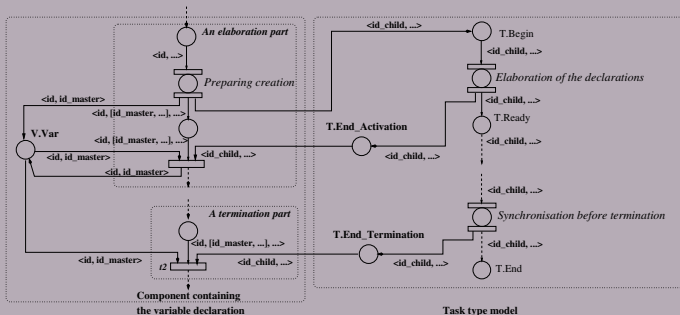
  T : TT;
begin
  P;
  ...
end ALLOC;
    
```



# Patterns (1/3) : Elaborated tasks

## Synchronizations with the father

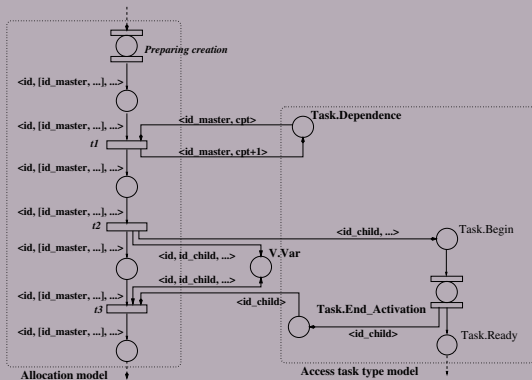
Synchronizations of an elaborated task with its father after activation and termination



# Patterns (2/3) : Allocated tasks

Evaluation of a task allocator

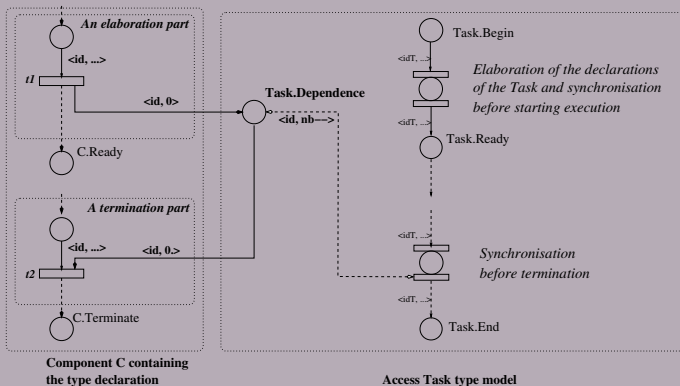
## Synchronization of an allocated task with its father at allocation



# Patterns (3/3) : Allocated tasks

## Synchronization with the master after termination

### Synchronization of an allocated task with its master after termination



## Evaluation on 2 examples



# Sieve of Erathostene

Number of states generated to prove deadlock freeness

<i>N</i>	<i>Running tasks</i>	<i>States without reductions</i>	<i>States with reductions</i>	<i>Reduction factor</i>
3	3	1 556	294	5
6	4	19 160	1 784	11
10	5	224 102	10 047	22
12	6	2 810 870	65 645	43

► Program



# A client/server program

Number of states generated to prove deadlock freeness

$N$	<i>Running tasks</i>	<i>States without reductions</i>	<i>States with reductions</i>	<i>Slicing &amp; reductions</i>	<i>Reduction factor</i>
1	4	2 549	247	221	12
2	6	438 913	9 499	5 939	74
3	8	–	735 767	239 723	–
4	10	–	–	12 847 017	–

► Program



# Conclusion

## Conclusion

- It is possible to translate dynamic aspects of programming languages with Petri nets and so to verify such applications with [QUASAR](#) .
- However, highly dynamic and parallel applications (like the 2 previous examples) can lead [QUASAR](#) to combinatorial explosion.
- But our existing patterns and the new ones allow efficient reductions. . .
- . . . then, applying some reductions (slicing, static reductions, PO-reductions, . . . ), can lead to a high reduction of the combinatory and then to an efficient verification process of [QUASAR](#) .



# Appendix (1) : the client/server program (1/2)

## The program

```
procedure Server is
  N : Integer := 3;  -- The number of clients

  protected Data is  -- The accessed data
    procedure Get_Value (Value : out Integer);
  private
    Data_Value : Integer := 0;
  end Data;

  protected body Data is
    procedure Get_Value (Value : out Integer) is
      begin
        Data_Value := Data_Value + 1;
        Value := Data_Value;
      end Get_Value ;
    end Data ;

  task type Thread is
  begin
    accept Get_Value (Param : out Integer);
  end Thread;

  type Access_Thread is access Thread ;

  task body Thread is
  begin
    accept Get_Value (Param : out Integer) do
      Data.Get_Value (Param);
    end Get_Value ;
  end Thread;

  task type Task_Server is  -- Creates a thread for each client's request
  entry Get_Thread (Id : out Access_Thread);
  end Task_Server ;
```

```
task body Task_Server is
begin
  for I in 1..N loop
    accept Get_Thread (Id : out Access_Thread) do
      Id := new Thread;
    end Get_Thread;
  end loop;
end Task_Server ;

The_Task_Server : Task_Server ;  -- The task server

task type Client  -- Client of the task server
type Access_Client is access Client;

task body Client is
  Id : Access_Thread;
  Value : Integer;
begin
  The_Task_Server.Get_Thread (Id);  -- Get the thread
  Id.Get_Value (Value);  -- Get the value of the data
end Client;

A_Client : Access_Client ;

begin
  for I in 1..N loop
    -- main loop, creates the clients
    A_Client := new Client;
  end loop;
end Server;
```

▶ Back



Cedric

## Appendix (1) : the client/server program (2/2)

How does it work

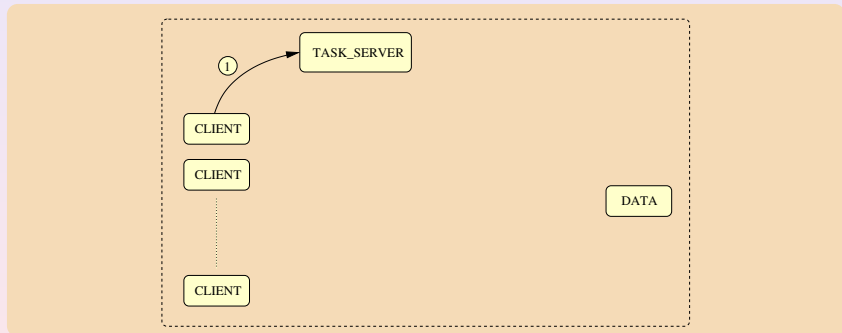


▶ Back



# Appendix (1) : the client/server program (2/2)

How does it work

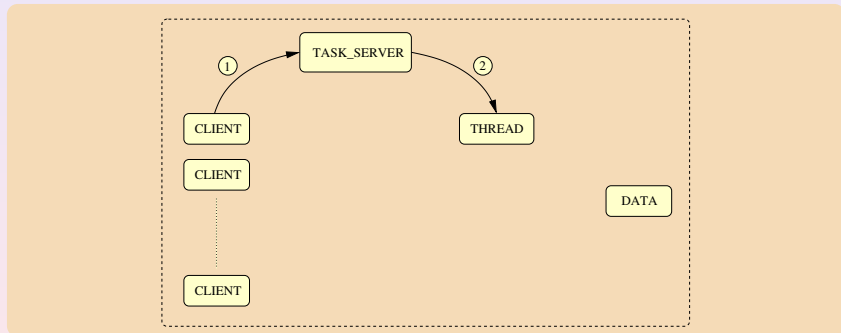


▶ Back



## Appendix (1) : the client/server program (2/2)

How does it works

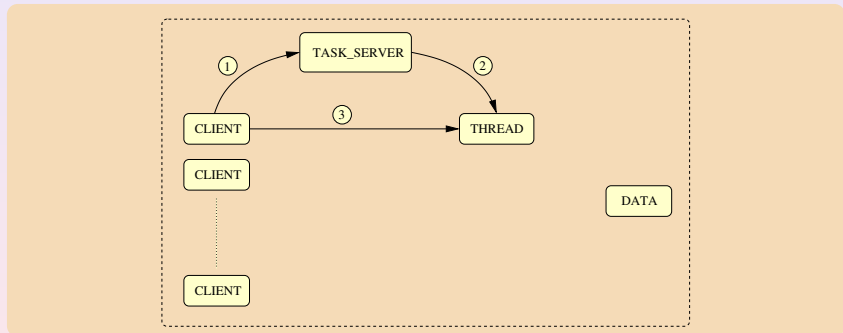


▶ Back



## Appendix (1) : the client/server program (2/2)

How does it works

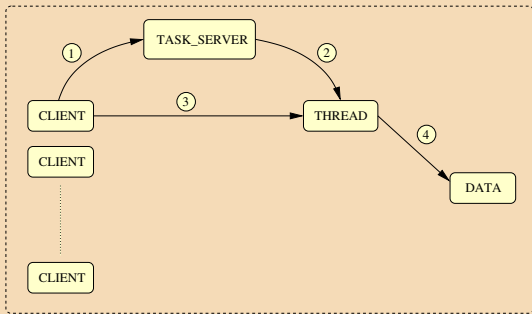


▶ Back



## Appendix (1) : the client/server program (2/2)

How does it works

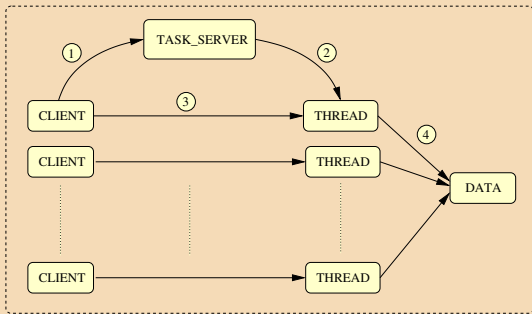


▶ Back



# Appendix (1) : the client/server program (2/2)

How does it works



▶ Back



# Appendix (2)

## The sieve of erathostene

```
procedure Eratho is
  task type Prime is
    entry Test_Primary (Number : in Integer);
  end Prime;

  type Access_Prime is access Prime;

  procedure Create_Prime (New_Prime : out Access_Prime) is
  begin
    New_Prime := new Prime;
  end Create_Prime;

  task body Prime is
    My_Number : Integer := 0;
    Temp      : Integer := 0;
    My_Next   : Access_Prime := null;
    Termination : Boolean := False;
  begin
    while not (Termination) loop
      accept Test_Primary(Number : in Integer) do
        Temp := Number;
      end Test_Primary ;
      if (Temp = 0) then
        -- Termination message
        Termination := True;
      if (My_Next /= null) then
        -- Propagate the termination message
        My_Next.Test_Primary(0);
      end if;
    end while;
  end Eratho;
```

```
else
  if (My_Number = 0) then
    -- Store the prime number
    My_Number := Temp;
  else
    -- Test the primary
    if ((Temp mod My_Number) /= 0) then
      if (My_Next = null) then
        -- If no neighbor, create one
        Create_Prime (My_Next);
      end if;
      -- Test the primary on the neighbor
      My_Next.Test_Primary(Temp);
    end if;
  end if;
end if;
end loop;
end Prime;
My_Prime : Prime;

begin
  for Number in 2..5 loop
    My_Prime.TestPrimarity (Number);
  end loop;
  -- Send the termination message
  My_Prime.TestPrimarity(0);
end Eratho;
```

