A Library of Anticipatory Random Number Generators

Vladimir Wojcik
Technical Report # CS-02-14
May 2002
A LIBRARY OF ANTICIPATORY RANDOM NUMBER GENERATORS

Vladimir Wojcik
Department of Computer Science
Brock University
St.Catharines, Ontario L2S 3A1
e-mail: vwojcik@sandcastle.cosc.brocku.ca

KEY WORDS
Ada generic package, exception, parallel software engineering, task.

ABSTRACT
The goal of this paper is to offer a quick parallel software engineering tutorial while creating an anticipatory random number generating tool. In this approach several difficulties were overcome: the generating task, coexisting with the user program, had to be made aware a priori of the required distributions. Also, this task must be capable of handling possible error conditions. This poses some difficulties, since the task is unaware of the nature of the simulation problem. This difficulty is compounded if the task were to be capable of accepting user-defined distributions: in such situation it is impossible to predict all error conditions that might arise. The task commits suicide, once it is not needed by the user program.

This paper focuses on the parallel software engineering aspects of the problem, assuming that the numerical principles of random number generation are well-known.

DESIGN GOALS
It is a well known fact that in most industrial settings the cost of software outpaces the cost of hardware by a factor of five to seven or more, while the costs of military software are approx. nine times greater than those of hardware. The bulk of these costs stem from software maintenance problems. Proper software engineering practices are intended to reduce the costs of development and software maintenance.

It is also a fact of life that software engineering has grown to be a discipline on its own. Industrial and military software practitioners face a dilemma when deciding how much time should be spent on learning the "arcane ways" of good software engineering, and how much time should they allocate to the actual development and maintenance of software products. Advances in parallel processing tend to exacerbate this dilemma. The knowledge and use of good software engineering concepts like modularity, abstraction, information hiding, encapsulation, inheritance, genericity, etc. becomes essential.

Indeed, the three programming languages most widely used in industry today are FORTRAN, COBOL and C. Their design predates the creation of modern software engineering techniques. In particular, their parallel processing extensions (if any) are non-standard, thus generating a significant part of software portability and maintenance costs.

This paper is intended to be a quick tutorial on modern parallel software engineering techniques. The author has chosen ADA as the language of the tutorial, since the ADA language standard supports parallel processing and most of the other software engineering techniques, while being a relatively popular language. The author assumes here that the reader knows the syntax and semantics of ADA, but may not be entirely familiar with the issues of parallel programming and software engineering in ADA.

The issues discussed in the paper transcend ADA, however. Given that the tutorial is offered in a simulation context, a familiar subject of random number generation has been chosen as the underlying application. A generic package of anticipatory random generators constitutes the library.
The value of the paper is twofold. First, it is an advanced tutorial on modern software engineering practices. The reader is led through the process of design decisions made when creating the library. Each decision is described and illustrated with the emerging code skeleton. The text of the paper amounts to a description of the modern software engineering design process.

Second, the library itself, in a machine-readable form, could be made available to the readers, together with another tutorial of more introductory nature: "Parallel Software Engineering: A Tutorial for the State of Mind". Please contact the author.

Apart from the random number generation, the library could be used as a software engineering template for development of arbitrary library packages, with the module responsible for the random number generation appropriately replaced.

The library, as it stands, is a useful product on its own. It offers ten independent random number streams. It supports most of the distributions used in simulation, namely beta, binomial, Erlang, exponential, gamma, log-normal, normal, Poisson, integer-valued uniform distribution (called RANDI after Simscript 5.II) real-valued uniform distribution (called UNIFORM, again after Simscript 5.II), and Weibull distribution. Additionally, two functions are provided as generic parameters. These functions, called CUSTINT and CUSTOM, allow the user to define an infinite number of integer-valued and real-valued generators.

Each generator can be used in the anticipatory mode (i.e. implemented as a separate parallel processing task) and in the usual way (i.e. implemented as a traditional procedure or function). Each generator is provided with two simple application programs, aimed as an illustration of its use in both ways.

The issues of parallel processing and exception (i.e. error) handling are discussed in detail in this paper. For each generator a set of conditions generating exceptions is defined, as appropriate. Exceptions could be caused by the misuse of the generators, or could be detected if generated by the hardware of the underlying machine. The issue of propagation and detection of exception conditions between various threads of execution in a parallel program is analyzed.

A final word on performance. Although the library was primarily intended for multi-CPU machines, with groups of generators being allocated separate CPUs, the adopted approach might enhance processing speed even in a uniprocessor environment, should the overall application frequently wait for data during its execution. With the priority of the generators adjusted appropriately, they could execute in the anticipatory fashion while the application waits for data.

**TOOL DESIGN**

The random number generating tool is presented in the form of a generic package, offering the user two functions returning random numbers of INTEGER and FLOAT types, to handle discrete and continuous distributions accordingly. These two functions have the form:

```plaintext
function COUNT (RATE : DISTRIBUTION) return INTEGER;
function VALUE (RATE : DISTRIBUTION) return FLOAT;
```

where the type DISTRIBUTION is a private type. The idea behind this approach lies in the observation that most of the errors when generating random numbers occur due to improper description of the desired distribution. Therefore the user is requested to introduce the parameters of the desired distributions *a priori* to the generating task running asynchronously "behind the scenes." During this introduction most errors are trapped by the functions provided to the user to permit him to define typical distributions.

All generators have the form of functions:

```plaintext
function BETA (K1, K2 : FLOAT; S : STREAM_ID) return DISTRIBUTION;
function BINOMIAL (N : NATURAL; P : FLOAT; S : STREAM_ID) return DISTRIBUTION;
function CUSTINT (CID : POSITIVE; S : STREAM_ID) return DISTRIBUTION;
function CUSTOM (CID : POSITIVE; S: STREAM_ID) return DISTRIBUTION;
function ERLANG (MU : FLOAT; K :POSITIVE; S : STREAM_ID)
```


function EXPONENTIAL (MU : FLOAT; S : STREAM_ID) return DISTRIBUTION;
function GAMMA (MU, K : FLOAT; S : STREAM_ID) return DISTRIBUTION;
function LOG_NORMAL (MU, SIGMA : FLOAT; S : STREAM_ID) return DISTRIBUTION;
function NORMAL (MU, SIGMA : FLOAT; S : STREAM_ID) return DISTRIBUTION;
function POISSON (MU : FLOAT; S : STREAM_ID) return DISTRIBUTION;
function RANDI (I, J : NATURAL; S : STREAM_ID) return DISTRIBUTION;
function UNIFORM (A, B : FLOAT; S : STREAM_ID) return DISTRIBUTION;
function WEIBULL (SHAPE, SCALE : FLOAT; S : STREAM_ID) return DISTRIBUTION;

The description of the use of the functions CUSTINT and CUSTOM is a little more complex and therefore is described a bit later in the paper.

The role of the functions returning the DISTRIBUTION values is twofold: the checks are applied to ensure that the distribution parameters are formed correctly, and, should these checks pass, then the resulting distribution values are introduced to the generating task.

An exception DISTRIBUTION_ERROR is raised by these functions, should they encounter mathematically improper values of the parameters. Specifically, this exception is raised in the foreground thread of control in the following situations:

<table>
<thead>
<tr>
<th>DISTRIBUTION</th>
<th>ERROR CONDITION</th>
<th>DISTRIBUTION</th>
<th>ERROR CONDITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beta</td>
<td>K1•0 and/or K2•0</td>
<td>Log-Normal</td>
<td>MU•0 and/or SIGMA•0</td>
</tr>
<tr>
<td>Binomial</td>
<td>P&lt;0 and/or P&gt;1</td>
<td>Normal</td>
<td>SIGMA•0</td>
</tr>
<tr>
<td>Erlang</td>
<td>MU•0</td>
<td>Poisson</td>
<td>MU•0</td>
</tr>
<tr>
<td>Exponential</td>
<td>MU•0</td>
<td>Randi</td>
<td>J&lt;I</td>
</tr>
<tr>
<td>Gamma</td>
<td>MU•0 and/or K•0</td>
<td>Uniform</td>
<td>B&lt;A</td>
</tr>
<tr>
<td>Weibull</td>
<td>SHAPE•0 and/or SCALE•0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The tool is written in the form of a generic package, in order to:

1. establish the desired number of random number streams,
2. provide the possibility of using customized seeds,
3. enable the introduction of user-defined distributions.

Consequently, the specifications of generic parameters are as follows:

type STREAM_ID is (<>);
with function SEED(N : NATURAL) return NATURAL is <>;
with function GEN_CUSTINT (X : FLOAT; CID : POSITIVE) return INTEGER is <>;
with function GEN_CUSTOM (X : FLOAT; CID : POSITIVE) return FLOAT is <>;

The user obtains a customized random number generation tool by instantiating the generic package while substituting the formal generic parameters with actual ones. An exact method of doing so is described later in this paper. At this moment it is sufficient to state that the generic type STREAM_ID can be replaced with any actual enumerated or integer type.

The function SEED is used to "fertilize" the generator so that it can produce different but reproducible random number sequences. It is recommended that one value of its argument (perhaps zero?) is selected to generate irreproducible random value sequences. This can be achieved by having its implementation peruse the system clock while computing seed values.
Finally, the generic functions \texttt{GEN\_CUSTINT} and \texttt{GEN\_CUSTOM} are provided to enable the user to introduce any number of custom discrete and continuous random number distributions correspondingly. The particular distributions are distinguished by assigning to them specific values of the \texttt{CID} (Custom IDentifier) parameter. The convention adopted here is that the generating task, when calling these functions will pass an uniformly distributed random value \(x\) to the formal parameter \(X\). The value \(x\) will satisfy the inequality \(0 \leq x \leq 1\).

With all these explanations, the complete package specification is as follows:

```plaintext
generic

type STREAM\_ID is (<>);
with function SEED(N : NATURAL) return NATURAL is <>;
with function GEN\_CUSTINT (X : FLOAT; CID : POSITIVE) return INTEGER is <>;
with function GEN\_CUSTOM (X : FLOAT; CID : POSITIVE) return FLOAT is <>;
package RANDOM is

type DISTRIBUTION is private;
-- two functions below are random generators:
function COUNT (RATE : DISTRIBUTION) return INTEGER;
function VALUE (RATE : DISTRIBUTION) return FLOAT;

DISTRIBUTION\_ERROR : exception;
function BETA (K1, K2 : FLOAT; S : STREAM\_ID) return DISTRIBUTION;
function BINOMIAL (N : NATURAL; P : FLOAT; S : STREAM\_ID) return DISTRIBUTION;
function CUSTINT (CID : POSITIVE; S : STREAM\_ID) return DISTRIBUTION;
function CUSTOM (CID : POSITIVE; S : STREAM\_ID) return DISTRIBUTION;
function ERLANG (MU : FLOAT; K :POSITIVE; S : STREAM\_ID) return DISTRIBUTION;
function EXPONENTIAL (MU : FLOAT; S : STREAM\_ID) return DISTRIBUTION;
function GAMMA (MU, K : FLOAT; S : STREAM\_ID) return DISTRIBUTION;
function LOG\_NORMAL (MU, SIGMA : FLOAT; S : STREAM\_ID) return DISTRIBUTION;
function NORMAL (MU, SIGMA : FLOAT; S : STREAM\_ID) return DISTRIBUTION;
function POISSON (MU : FLOAT; S : STREAM\_ID) return DISTRIBUTION;
function RANDI (I, J : NATURAL; S : STREAM\_ID) return DISTRIBUTION;
function UNIFORM (A, B : FLOAT; S : STREAM\_ID) return DISTRIBUTION;
function WEIBULL (SHAPE, SCALE : FLOAT; S : STREAM\_ID) return DISTRIBUTION;
private

type DISTRIBUTION is ... -- implementation dependent details
end RANDOM;
```
As said before, the generic tool must be instantiated before use. Two methods of specifying actual values are provided. A simple method requires the direct visibility of identically named actual values - in that situation the substitution of generic parameters will be automatic during compilation.

A more general and therefore more complex method requires the programmer to explicitly specify the actual parameters to substitute the generics. Regardless of the method followed, the instantiation will produce another random number generating package, and the actual testing will be done using the new package rather than the generic original.

The recommended practice would be to establish a package that would encapsulate all the required parameters needed for instantiation. In our case it is the package LUCK, with the following specification:

```pascal
package LUCK is
  type STREAM_NO is range 1..10;
  function SEED(N : NATURAL) return NATURAL;
  function DRAW(X : FLOAT; CID : POSITIVE) return FLOAT;
  function COIN(X : FLOAT; CID : POSITIVE) return INTEGER;
end LUCK;
```

The above implies that we use ten streams for random number generation, and provide some function SEED. Let us assume that we need only two very simple custom distributions, one continuous and one discrete. The continuous distribution function, called DRAW, whenever called, will return a uniformly distributed random number within the interval [0, 1]. The discrete distribution will mimic a flip of a balanced coin, returning the values 0 and 1 with equal probability. A very simple (bordering on trivial!) implementation of the package LUCK could look like this:

```pascal
package body LUCK is
  function SEED(N : NATURAL) return NATURAL is
  begin
    return N;
  end SEED;

  function DRAW(X : FLOAT; CID : POSITIVE) return FLOAT is
  begin
    return X; -- more complex generation methods, viz. inversion of
    -- the cummulative distribution function, elimination
    -- method, etc. could be used here
  end DRAW;

  function COIN(X : FLOAT; CID : POSITIVE) return INTEGER is
  begin
    if X<=0.5 then return 0;
    else return 1;
  end if;
  end COIN;
end LUCK;
```

Since we need only one discrete and only one continuous distribution, it was possible that the implementation of the functions DRAW and COIN merely ignores the value of the parameter CID. Had we wanted to implement more custom-made distributions, then the implementation would have to sense the CID values and to proceed accordingly.

Please also note that the adopted convention requiring the parameter X of functions COIN and DRAW to be a realization of an uniformly distributed random variable within [0, 1] permits us merely to return the value of X when calling DRAW.

Having established our LUCK, we can start thinking of writing the testing program, say, RANTEST. Within it we have to instantiate the package RANDOM more or less like this:
As said before, it is the package TEST that will be the subject of testing.

Suppose the test procedure is to produce and print the value of a single coin toss. In its simplest it could take this form:

```pascal
with TEXT_IO, RANDOM, LUCK;
procedure RANTEST is
  package TEST is new RANDOM (STREAM_ID   => LUCK.STREAM_NO,
                               SEED        => LUCK.SEED,
                               GEN_CUSTINT => LUCK.COIN,
                               GEN_CUSTOM  => LUCK.DRAW);
  use TEST;

  package INT_IO is new TEXT_IO.INTEGER_IO(INTEGER);
  TOSS : DISTRIBUTION;

  begin
    TOSS := CUSTINT (1, 1);
    INT_IO.PUT(COUNT(TOSS));
    TEXT_IO.PUT_LINE;
  end RANTEST;
```

The declarations of this procedure involve the creation of the object TOSS. All the information regarding the parameters of the user-defined distribution COIN is stored in the variable TOSS by the first executable statement. The second statement involves the call of the function COUNT of the package TEST. When called, this information is perused. Since it implies that the function COIN is to be called (COIN being an actual parameter replacing GEN_CUSTINT), the proper call is made and the value printed.

**IMPLEMENTATION CONSIDERATIONS**

The concurrency is achieved by creating within the package body RANDOM an asynchronously running generator task. Its specifications are:

```pascal
  task GENERATOR is
    entry REGISTER (DIS : DISTRIBUTION);
    entry COUNT    (DIS : DISTRIBUTION; I : out INTEGER);
    entry VALUE    (DIS : DISTRIBUTION; X : out FLOAT);
  end GENERATOR;
```

The entries COUNT and value correspond to the procedures COUNT and VALUE defined within the specification of the package RANDOM. Indeed, their implementation is very straightforward:

```pascal
  function COUNT (RATE : DISTRIBUTION) return INTEGER is
    INT : INTEGER;
  begin
    GENERATOR.COUNT(RATE, INT); return INT;
  end COUNT;

  function VALUE (RATE : DISTRIBUTION) return FLOAT is
    FLT : FLOAT;
  begin
    GENERATOR.VALUE(RATE, FLT); return FLT;
  end VALUE;
```
As discussed before, every distribution must be given specific parameter values to be registered with the GENERATOR for subsequent use. The registration (using the entry REGISTER) is done only if the particular parameter values have mathematical sense; failing that the exception DISTRIBUTION_ERROR is raised. This is the role of the functions BETA, BINOMIAL, ERLANG, etc., defined within the specification of the package RANDOM. The following example, being the implementation of the function BETA, is used as an illustration.

```
function BETA (K1, K2 : FLOAT; S : STREAM_ID) return DISTRIBUTION is
  DIS : DISTRIBUTION;
begin
  if (K1<=0.0) or (K2<=0.0) then
    raise DISTRIBUTION_ERROR;
  else
    DIS := ...;   -- implementation dependent
    -- storing of distribution parameters
    GENERATOR.REGISTER(DIS);
    return DIS;
  end if;
end BETA;
```

All standard mathematical distributions are handled in this manner.

The handling of user-defined distributions CUSTINT and CUSTOM deviates from that approach only because it is impossible to detect here possible nonsensical combinations of parameters. That must remain the responsibility of the user. The body of the function CUSTOM is listed below as an example:

```
function CUSTOM (CID : POSITIVE; S : STREAM_ID) return DISTRIBUTION is
  DIS : DISTRIBUTION;
begin
  DIS := ...;       -- implementation dependent
  -- storing of distribution parameters
  GENERATOR.REGISTER(DIS);
  return DIS;
end CUSTOM;
```

In that manner the majority of exceptions will be raised during the execution of the foreground thread of control. Other exception situations will be raised by the GENERATOR task running in the background and will have to be propagated to the foreground thread of execution. One such situation, not mentioned yet, will arise when calling the function COUNT or VALUE and passing to it a parameter of the type DISTRIBUTION, which has not been previously registered with the GENERATOR task.

In general, the generating task is able to handle and survive any exception situation. It will stop executing together with the user program. It stores the registered distributions in ITEMS of information organized into a two-way linked list. The typeITEM is as follows:

```
type ITEM;
type ITEM_REF is access ITEM;
type ITEM is
  record
    PAR      : PARAMS;
    VAL      : FLOAT;
    CNT      : INTEGER;
    NXT, FMR : ITEM_REF;
  end record;
```

In the above, the type PARAMS is used to store the distribution parameters (ellided by the three-dot notation when listing the functions BETA and CUSTOM). The fields VAL and CNT are used to store at the ready the pre-computed values of VALUE and COUNT. The procedure ALEA (Latin for 'dice'), described later in this paper, is used to that effect. In that way the handling of the corresponding entry calls in the task GENERATOR is as quick and straightforward as possible. The values NXT and FMR are needed to establish a two-way linked list: they are pointers to the next and former entries on the list.
The list has a header, declared as follows:

```plaintext
type LIST_HEADER is
  record
    THIS, FIRST, LAST : ITEM_REF;
  end record;

  HEADER : LIST_HEADER;
```

The header structure consists of three pointers THIS, FIRST and LAST, referencing the corresponding ITEMS of the list. (The current one is called THIS).

Additional objects declared in the GENERATOR body include:

```plaintext
DISCRETE, CONTINUOUS : BOOLEAN := FALSE;
REF : ITEM_REF;
```

The Boolean values DISCRETE and CONTINUOUS, initialized to FALSE, become TRUE only after at least one distribution of the discrete and continuous type correspondingly, has been registered with the generator. This unblocks the guards for its corresponding entries COUNT and VALUE. Finally, the variable REF is a temporary one.

With that in mind, registration of a particular distribution with the GENERATOR amounts to storing of the corresponding ITEM of information in the list and updating its HEADER. The following procedure is intended to do just that:

```plaintext
procedure STORE (HDR : in out LIST_HEADER; REF : ITEM_REF) is
begin
  if HDR.LAST=null then
    HDR.LAST := REF;
    HDR.FIRST := REF;
    HDR.THIS := REF;
  else
    HDR.LAST.NXT := REF;   REF.FMR := HDR.LAST;
    HDR.LAST := REF;   HDR.THIS := REF;
  end if;
end STORE;
```

Calling the GENERATOR.COUNT and GENERATOR.VALUE entries in order to obtain random values is a two-step affair: firstly, we have to align THIS pointer of the list HEADER with the corresponding ITEM, and secondly, return the pre-computed VAL or CNT of that ITEM, as required. To speed up this search, we can start it with FIRST, LAST, or THIS pointer, in order to minimize the number of steps along the linked list. The procedure ALIGN is intended to achieve that:

```plaintext
procedure ALIGN (HDR : in out LIST_HEADER; DIS : DISTRIBUTION) is
  FROM_FIRST, FROM_THIS, FROM_LAST : INTEGER;
  -- the function STEPS performs the list traversal:
  function STEPS(N : INTEGER; FROM : ITEM_REF) return ITEM_REF
  is
    TMP : ITEM_REF; COUNT : INTEGER;
    begin
      TMP := FROM; COUNT := N;
      while COUNT > 0 loop
        TMP := TMP.NXT; COUNT := COUNT - 1;
      end loop;
      while COUNT < 0 loop
        TMP := TMP.FMR; COUNT := COUNT + 1;
      end loop;
      return TMP;
    end STEPS;
```

Calling the GENERATOR.COUNT and GENERATOR.VALUE entries in order to obtain random values is a two-step affair: firstly, we have to align THIS pointer of the list HEADER with the corresponding ITEM, and secondly, return the pre-computed VAL or CNT of that ITEM, as required. To speed up this search, we can start it with FIRST, LAST, or THIS pointer, in order to minimize the number of steps along the linked list. The procedure ALIGN is intended to achieve that:
begin
FROM_FIRST := ...; -- number of steps if traversal is to start here
FROM_THIS := ...; -- number of steps if traversal is to start here
FROM_LAST := ...; -- number of steps if traversal is to start here

-- we traverse starting from the best place:
if abs(FROM_FIRST)<abs(FROM_LAST) then -- FROM_FIRST
    if abs(FROM_THIS)<abs(FROM_FIRST) then -- FROM_THIS
        HDR.THIS := STEPS(FROM_THIS, HDR.THIS);
    else -- FROM_FIRST
        HDR.THIS := STEPS(FROM_FIRST, HDR.FIRST);
    end if;
else -- FROM_LAST
    if abs(FROM_THIS)<abs(FROM_LAST) then -- FROM_THIS
        HDR.THIS := STEPS(FROM_THIS, HDR.THIS);
    else -- FROM_LAST
        HDR.THIS := STEPS(FROM_LAST, HDR.LAST);
    end if;
end if;
end ALIGN;

With all these preparations in place we are now in position to sketch the outline of the body of the GENERATOR task. It takes the form of an indefinite loop:

loop
begin
    select
        accept REGISTER (DIS : DISTRIBUTION) do
            if DIS.RID=0 then -- this DIS hasn't been registered yet, so
                -- we mark it now as registered
                if HEADER.LAST=null then
                    DIS.RID := 1;
                else
                    DIS.RID := HEADER.LAST.PAR.RID + 1;
                end if;
            end if;
            REF := new ITEM; -- we create new ITEM and store the
            -- distribution parameters in its PAR
            REF.PAR := DIS.all;
        end REGISTER;
        if REF /= null then
            STORE(HEADER, REF);  -- storing ITEM in list
            ALEA(REF);           -- generating proper random value and
            -- opening a proper guard
            case REF.PAR.DID is
                when BINOMIAL | CUSTINT | POISSON | RANDI =>
                    DISCRETE := TRUE;
                when others =>
                    CONTINUOUS := TRUE;
            end case;
        end if;
    end if;
end loop

With all these preparations in place we are now in position to sketch the outline of the body of the GENERATOR task. It takes the form of an indefinite loop:
when BINOMIAL | CUSTINT | POISSON | RANDI =>
    ALIGN(HEADER, DIS);
    I := HEADER.THIS.CNT;
when others =>
    raise DISTRIBUTION_ERROR;
end case;
end if;
end COUNT;
ALEA(HEADER.THIS); -- generate next random value
or
when CONTINUOUS =>
    accept VALUE (DIS : DISTRIBUTION; X : out FLOAT) do
        if (DIS=null) or else (DIS.RID=0) then
            -- this DIS has not been registered!
            raise DISTRIBUTION_ERROR;
        else
            case DIS.DID is
            when BINOMIAL | CUSTINT | POISSON | RANDI =>
                raise DISTRIBUTION_ERROR;
            when others =>
                ALIGN(HEADER, DIS);
                X := HEADER.THIS.VAL;
            end case;
        end if;
    end VALUE;
ALEA(HEADER.THIS); -- generate next random value
or
    terminate;
end select;
exception
    when DISTRIBUTION_ERROR =>
        ...;               -- handle it and put some message
when others =>
    null;                 -- ignore it and put some message
end;
ACKNOWLEDGMENT

The author would like to thank Mr. W. Murray of the Department of Computer Science, Brock University, for his care that this paper forms a cohesive whole.

ABOUT THE AUTHOR

Vladimir Wojcik is Associate Professor of Computer Science At Brock University in St.Catharines, Canada. Before coming to Brock he worked as management Scientist for DEC in their European HQ, Geneva, Switzerland, after having held the teaching and research posts at Ahmadu Bello University, Zaria, Nigeria and at the Technical University of Warsaw, Warsaw, Poland. His interests include: parallel processing, distributed simulation, software engineering, artificial intelligence with special emphasis on machine vision and cognition.