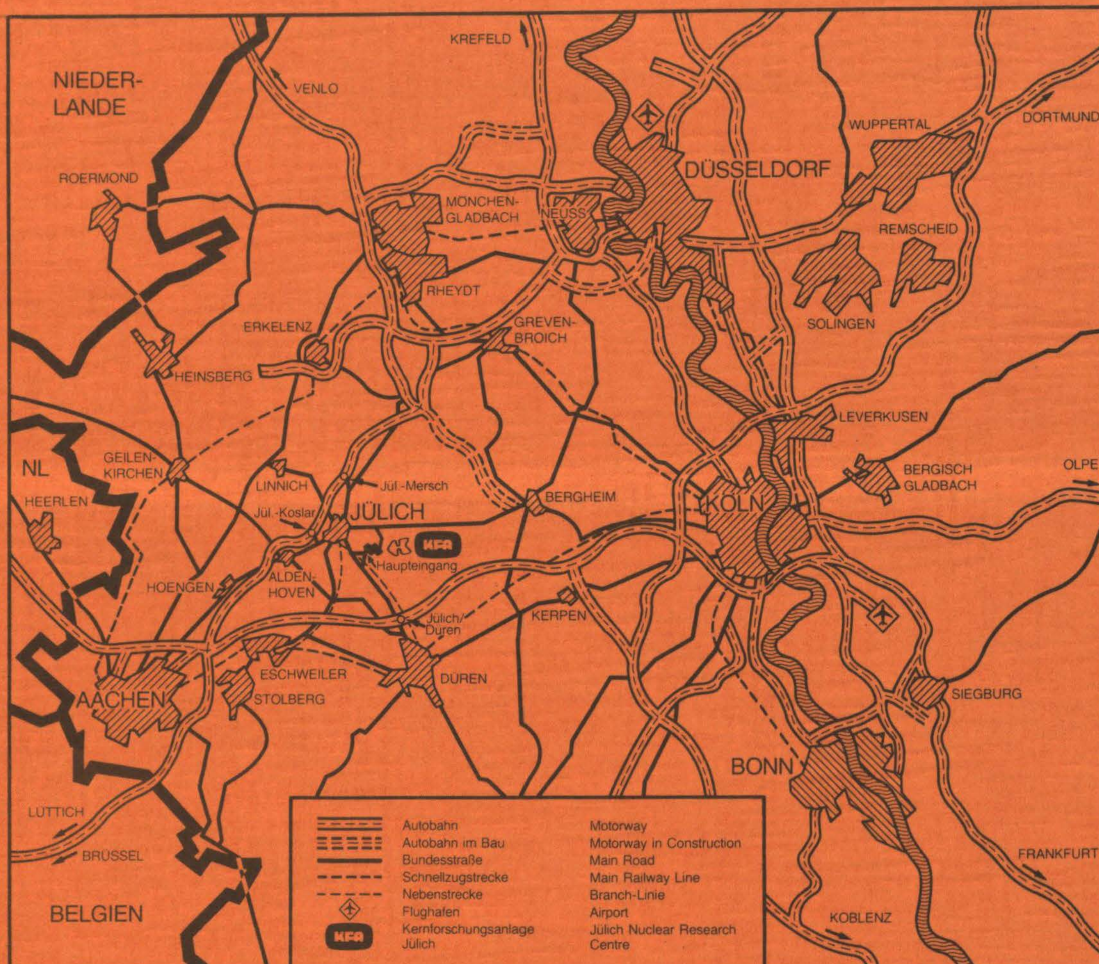
KERNFORSCHUNGSANLAGE JÜLICH GmbH
Programmgruppe Technik und Gesellschaft

EMULATING SIMULA IN TURBO PASCAL

by

Morton John Canty

Jül-Spez-467
September 1988
ISSN 0343-7639



Spezielle Berichte der Kernforschungsanlage Jülich – Nr. 467

Programmgruppe Technik und Gesellschaft Jül-Spez-467

Zu beziehen durch: ZENTRALBIBLIOTHEK der Kernforschungsanlage Jülich GmbH
Postfach 1913 · D-5170 Jülich (Bundesrepublik Deutschland)
Telefon: 02461/610 · Telex: 833556-0 kf d

EMULATING SIMULA IN TURBO PASCAL

by

Morton John Canty

EMULATING SIMULA IN TURBO PASCAL

M. J. Canty

1. Introduction

Although the computer language SIMULA /1,2/ is now over 20 years old it remains an excellent general purpose programming tool as well as a popular and powerful simulation language, the purpose for which it was originally developed. SIMULA is an extension of ALGOL 60, which it contains as a true subset, and its advanced concepts have served as a model for modern object-oriented languages such as SMALLTALK.

The properties which make SIMULA especially suitable for simulation tasks are

1. a hierarchic class concept with inheritance,

2. sophisticated list handling facilities and

3. concurrent programming capability.

Of these three, the most essential property to allow for programming of discrete time simulation tasks is the third one, concurrent programming. By this is meant the ability to sustain parallel autonomous entities (called processes or co-routines) in memory. Allowing an arbitrary number of such processes to interact with each other along a time axis forms the basis of SIMULA's model for discrete time simulation.

Although virtually all major programming languages have been implemented in one form or another on personal computers, SIMULA is a notable exception. Perhaps the main reason for this is that, while enjoying great popularity in Europe, SIMULA is not as well known on the North American continent. Pascal belongs to the same Algol family as SIMULA, and the dialect Turbo Pascal of the firm Borland International has become one of the most wide spread high-level languages for MS-DOS personal computers. Unfortunately, neither the ANSI-Pascal specification nor Turbo-Pascal in particular allow for concurrent programming.

A recent article in BYTE by Krishnamoorthy and Agnarsson /3/ presented an extension to Turbo Pascal 3.0 which enables the creation of parallel processes. In the

present report, their extension is modified for the latest version (4.0) of Turbo Pascal /4/ and integrated into a Turbo Pascal unit (pre-compiled module) which emulates the elementary simulation constructs of the SIMULA language. A simple application illustrating the use of the unit is provided.

2. A Discrete Time Simulation Model

We begin by defining a time axis consisting of an ordered sequence of "slots". Each slot is a Pascal record with three attributes: a reference to an associated process, the time at which the process is next to be activated, and a pointer to the next slot in the sequence. This is realized by the type declaration *slot_type* in the implementation part of the unit simula (Appendix I). The slots are ordered according to their time attributes, earliest times first.

A process is also a Pascal record structure consisting of a pointer to the process's stack, an identifying text and an arbitrary number of variable declarations (perhaps zero) specific to the processes to be created. These are variables which are to be accessible globally, i.e. from other processes, and of course must be provided by the user as needed. (See the declaration *process_type* in the interface part of the unit.) When a new process is created (as described in the following section), the process's stack is associated with a parameterless procedure which describes the actions to be carried out by the process when it is activated.

A process can be in one of three possible states:

1. If it is not represented as a slot on the time axis, the process is *passive.*

2. If it is in the first slot on the time axis the process is *active.*

3. Otherwise the process is *suspended* , i.e. scheduled for later activation.

A discrete time simulation then consists simply of the creation and scheduling of processes, with control being given to the currently active process. An active process can reschedule itself, or create, activate or reschedule other processes. The passage of time corresponds to control passing from the current process to the next one scheduled. The simulation terminates with the exhaustion of all actions of the last process on the time axis.

3. The Unit Simula

The unit *simula* (Appendix I), once compiled, is linked to the user's simulation program with the Turbo Pascal directive *uses simula;*.

The user now has at his disposal the following functions and procedures for discrete time simulation:

function CURRENT: process; returns a reference to the process in the first slot on the time axis, i.e. to the currently active process.

function TIME: real; returns the event time of the active process, i.e. the current time of the simulation.

function EVTIME(p: process): real; returns the scheduled time of process *p* if it is on the time axis (i.e. active or suspended) else -1.0.

function IDLE(p: process): boolean; returns false if *p* is active or suspended, else true.

procedure NEWP(var p: process; ident: id_type; prog_seg,prog_ofs,size: integer); initializes *p* as a new process. *prog_seg* and *prog_ofs* are the segment and offset of the parameterless procedure associated with the process. *ident* is an identifying string and *size* is the stack size for the process.

procedure ACTIVATE(p: process); makes *p* the active or current process. The calling process is suspended.

procedure CANCEL(p: process); removes process *p* from the time axis, i.e. it becomes passive.

procedure PASSIVATE; removes the current process from the time axis. Same effect as *CANCEL(CURRENT);*

procedure REACTIVATE(p: process; keyword: keyword_type; t: real); places process *p* onto the time axis, according to the value of keyword:

If *keyword*= *AT*, *p* is given event time *t* and scheduled after any other events with the same event time.

If *keyword*= *PRIOR*, *p* is given event time *t* and scheduled before any other events with the same event time.

If *keyword*= *AFTERCURRENT, p* is given the current event time and scheduled immediately after the current process. The value of *t* is ignored.

procedure HOLD(t: real); reschedules the current process at *TIME* + *t*.

The use of these functions and procedures will be illustrated in the example discussed in the next section, and we shall concern ourselves here only with a brief discussion of the way in which the unit works.

The key components of the unit are the function *NewProcess* and the procedure *Transfer*, the latter being written in assembler, compiled and then linked to the unit with the {L} compiler directive /4/. These procedures have been taken from /3/ and modified slightly to conform to Turbo Pascal 4.0. The modification takes account of the fact that all units are in different core segments so that procedure calls over unit boundaries are far. The way in which these two procedures enable concurrent programming is explained in detail in /3/. Essentially, the stack conventions of Turbo Pascal are circumvented by creating additional stacks in the heap area and associating them with parameterless procedures defining the behavior of the processes. This is accomplished by *NewProcess.* One procedure can be associated with arbitrarily many stacks, corresponding to different "incarnations" of a single process. The procedure *Transfer* switches control from one process to another by manipulating the stack segment and stack pointer registers. (This is a strictly forbidden activity in "normal" Turbo Pascal!) The unit *simula* does not export either of these two routines, but rather uses them internally. The creation of a new process, for example, is accomplished with the exported procedure *NEWP*, which itself calls *NewProcess.*

Caution is advised in setting the process stack size with the procedure *NEWP.* It is safest to be as generous as possible, since the Turbo Pascal compiler makes extensive use of the stack and "doesn't know" that it is working with many different stacks simultaneously. Overflows usually have disastrous consequences. A good rule is to use the normal default stack size of 16K to begin with and work down if necessary. (The author has found it convenient to write simple stack overflow checking routines during the debugging phase of simulation program development.)

Event scheduling is accomplished by the procedure *Update*, again not directly available to the user, and involves the manipulation of the singly linked list of slots using standard insertion/deletion algorithms /5/. The internal procedure *Scheduler*, in an infinite loop, calls *Update* to reorder the time axis and then gives control to the active process via a call to *Transfer.*

4. An Illustration

Use of the unit *simula* is illustrated here with a simple example taken from /2/. The program is listed in Appendix II. In fact this example does not involve time at all, but uses the discrete time simulation model to merge several ordered series of integers into a single ordered series. The program works as follows:

> For each input series (up to a maximum of 10) a new instance of the process defined by the parameterless procedure *v_proc* is created and then activated.
>
> Upon activation, the process reads its integer series k[i] from the input file and then enters a FOR-loop. Here, it reschedules itself for reactivation at "time" k[1] and control is relinquished to *main_proc.* Now the next instance of *v_proc* is created, and so on.
>
> When all instances are created, *main_proc* waits for 100 "time" units, using the *HOLD* statement, which is assumed to be the maximum size of any integer in an input series. The various instances of *v_proc* then "fire" in an orderly fashion, each time writing their k[i] values to the output stream and rescheduling themselves for k[i+1]. This produces the desired merged series. As each process exhausts its series, it removes itself from the time axis with the *PASSIVATE* statement.
>
> Eventually, *main_proc* again becomes active and the program terminates in an orderly way.

Note the structure of the sample program. The body consists only of two statements: the creation of a process called *main* and the activation of that process. This is done for compatibility of style to SIMULA, which automatically distinguishes the simulation program body as a "main process" which can be scheduled just like any other process.

Of course far more sophisticated applications are possible. The author has used the unit *simula* to write a simulation program in Turbo Pascal for the head end of a nuclear processing plant. Over 50 components (tanks, centrifuges, flow switching devices, even the plant operator and the safeguards inspector) are modelled as concurrent processes. The program is distributed over five units (including *simula*) in about 3000 lines of code and runs successfully on a 640K MS-DOS PC.

References

/1/ SIMULA Users Guide, Norwegian Computing Center, Revised Edition, 1975.

/2/ H. Rohlfing, SIMULA Eine Einführung, Bibliographisches Institut AG, Mannheim, 1973.

/3/ M. S. Krishnamoorthy and S. Agnarsson, Concurrent Programming in Turbo Pascal, BYTE, 12, No. 4, April, 1987, p. 127.

/4/ Turbo Pascal 4.0 User Manual and Reference Manual, Borland International INC, 1988.

/5/ N. Wirth, Algorithmen und Datenstrukturen, B. G. Teubner Stuttgart, 1979.

Appendix I. Source Listing of Unit Simula

```
UNIT simula;
{-----------------------------------------------------
   SIMULA Emulation in Turbo Pascal 4.0
   Reference: Co-routining  BYTE, Vol 12, No. 4, p. 127
                    M. J. Canty, June, 1988.
 -----------------------------------------------------}


INTERFACE

type ProcessPointer = ^integer; {Pointer to the process procedure's stack}
    id_type     = string[48];
    process = ^process_type;
    keyword_type = (AT,AFTERCURRENT,PRIOR);
    {*** Application-specific type declarations ***}

    process_type   = record
                  pr: ProcessPointer;
                  id: id_type;
                  {*** Application-specific variable declarations ***}
                end;


function CURRENT: process;
{Returns a reference to the currently active process}

function TIME: real;
{Returns the current time}

function EVTIME(p: process): real;
{Returns the scheduled time of process p if active or suspended, else -1.0}

function IDLE(p: process): boolean;
{Returns false if p is active or suspended, else true}

procedure NEWP(var p: process;ident: id_type;prog_seg,prog_ofs,size: integer);
{Initializes a new process p. prog_seg and prog_ofs are the
segment and offset of the parameterless procedure associated
with the process. size is the stack size for the process.}

procedure ACTIVATE(p: process);
{Process p becomes active. The calling process is suspended.}

procedure CANCEL(p: process);
{Process p is removed from the time axis.}

procedure PASSIVATE;
{The current process is removed from the time axis.}

procedure REACTIVATE(p: process; keyword: keyword_type; t: real);
{Process p is placed onto the time axis, according to the value of
keyword. If keyword= AT, p is given event time t and scheduled after any
other events with the same event time. If keyword= PRIOR, p is given
event time t and scheduled before any other events with the same event
time. If keyword= AFTERCURRENT, p is given the current event time and
scheduled immediately after the current process. The value of t
is ignored.}
```

```
procedure HOLD(t: real);
{The current process is rescheduled at TIME + t.}
```

IMPLEMENTATION

```
type slot      = ^slot_type;
    slot_type  = record
                 time :  real;
                 proc :  process;
                 next :  slot;
                end;

var HeadSlot, TailSlot,
   cs        : slot;
   mpp,
   cpp        : processpointer;
   pp        : process;
   tp        : real;
   pf        : (act,reactat,reactac,reactpr,canc);

function NewProcess(prog_seg,prog_ofs,size: integer): ProcessPointer;
var stack: ^integer;
begin
  GetMem(stack,size);
  MemW[Seg(stack^): Ofs(stack^)+size-10] := prog_seg;
  MemW[Seg(stack^): Ofs(stack^)+size-12] := prog_ofs;
  MemW[Seg(stack^): Ofs(stack^)+size-14] := Ofs(stack^)+size-14;
  NewProcess := Ptr(Seg(stack^),Ofs(stack^)+size-14);
end;

{$F+}
procedure Transfer(var p1,p2: ProcessPointer); external;
{$F-}
{$L Transfer}
{   Assembler listing of external procedure Transfer
    ---------------------------------------------
code      segment byte public
        assume  cs: code
        public  transfer
transfer   proc    far
;
 push   bp
 mov    bp,sp
;
 les    bp,dword ptr [bp]+6
 mov    ax,es:[bp]+2
 mov    bx,es:[bp]
 mov    bp,sp
 les    bp,dword ptr [bp]+10
 mov    es:[bp],sp
 mov    es:[bp]+2,ss
 mov    ss,ax
 mov    sp,bx
 mov    bp,sp
;
 mov    sp,bp
```

```
  pop   bp
  ret   8
;
transfer   endp
code       ends
         end
}

procedure newp;
begin
  New(p);
  with p^ do begin
    id:= ident;
    pr:= NewProcess(prog_seg,prog_ofs,size);
  end;
end;

procedure update;

  procedure activate;
  var s,ns: slot;
  begin
    s:= HeadSlot;
    repeat s:= s^.next until (s^.proc = pp) or (s = nil);
    if s = nil then begin
      new(ns);
      ns^.proc:= pp;
      ns^.time:= tp;
      ns^.next:= HeadSlot^.next;
      HeadSlot^.next:= ns;
    end;
  end;

  procedure cancel;
  var s,ss: slot;
  begin
    s:= HeadSlot;
    repeat s:= s^.next until (s^.proc = pp) or (s = nil);
    if s <> nil then begin
      ss:= s^.next;
      s^:= ss^;
      dispose(ss);
    end;
  end;

  procedure reactivate;
  var s,ss,ns: slot;
  begin
    s:= HeadSlot;
    repeat s:= s^.next until (s^.proc = pp) or (s = nil);
    if s <> nil then begin
      ss:= s^.next;
      s^:= ss^;
      dispose(ss);
    end;
    if pf = reactat then begin
      s:= HeadSlot;
      while s^.time <= tp do s:= s^.next;
      New(ns);
      ns^:= s^;
```

```
        s∧.time:= tp;
        s∧.proc:= pp;
        s∧.next:= ns;
      end
      else if pf = reactpr then begin
        s:= HeadSlot;
        while s∧.time < tp do s:= s∧.next;
        New(ns);
        ns∧:= s∧;
        s∧.time:= tp;
        s∧.proc:= pp;
        s∧.next:= ns;
      end
      else if pf = reactac then begin
        New(ns);
        ns∧.time:= time;
        ns∧.proc:= pp;
        ns∧.next:= cs∧.next;
        cs∧.next:= ns;
      end;
    end;

begin  { update }
  if cpp <> nil then cs∧.proc∧.pr:= cpp;
  case pf of
    act:     activate;
    canc:    cancel;
    reactat,
    reactac,
    reactpr: reactivate;
  end;
end;  { update }

procedure scheduler;
begin
  while true do begin
    update;
    cs:= HeadSlot∧.next;
    cpp:= cs∧.proc∧.pr;
    transfer(mpp,cpp);
  end;
end;

procedure resume;
begin
  if cpp = nil then scheduler else Transfer(cpp,mpp);
end;

procedure activate;
begin
  pp:= p;
  tp:= time;
  pf:= act;
  resume;
end;

procedure reactivate;
begin
  pp:= p;
  tp:= t;
```

```
    if keyword = at then pf:= reactat else
    if keyword = aftercurrent then pf:= reactac else
    if keyword = prior then pf:= reactpr;
    resume;
  end;

  procedure cancel;
  begin
    pp:= p;
    pf:= canc;
    resume;
  end;

  function current;
  begin
    current:= HeadSlot∧.next∧.proc;
  end;

  procedure passivate;
  begin
    cancel(current);
  end;

  procedure hold;
  begin
    reactivate(current,at,time+t);
  end;

  function time;
  var t: real;
  begin
    t:= HeadSlot∧.next∧.time;
    if t=1000000.0 then time:=0.0 else time:= t;
  end;

  function evtime;
  var s: slot;
  begin
    s:= HeadSlot;
    repeat
      s:= s∧.next
    until (s∧.proc = p) or (s = nil);
    if s <> nil then evtime:= s∧.time else evtime:= -1.0;
  end;

  function idle;
  var s: slot;
  begin
    s:= HeadSlot;
    repeat
      s:= s∧.next
    until (s∧.proc = p) or (s = nil);
    if s <> nil then idle:= true else idle:= false;
  end;

  BEGIN         { initialization }
    cpp:= nil;
    New(HeadSlot);
    New(TailSlot);
    HeadSlot∧.time:= 0.0;
```

```
  HeadSlot∧.proc:= nil;
  HeadSlot∧.next:= TailSlot;
  TailSlot∧.time:= 1000000.0;
  TailSlot∧.proc:= nil;
  TailSlot∧.proc∧.id:= '';
  TailSlot∧.next:= nil;
END.
```

Appendix II. Source Listing of Sample Program Merge

```
program merge;
{ Merge n ordered series of integers into one series
using simulation. Example taken from
H. Rohlfing, "SIMULA Eine Einfuehrung", p. 215 }

uses simula;

var main  : process;
   v      : array[1..10] of process;
   infile: text;

procedure v_proc;
var n,i: integer;
   k: array[1..100] of integer;
begin
  read(infile,n);
  for i:=1 to n do read(infile,k[i]);
  for i:=1 to n do begin
    REACTIVATE(CURRENT,AT,k[i]);
    write(k[i]:4);
  end;
  PASSIVATE;
end;

procedure main_proc;
var i,n: integer;
begin
  assign(infile,'merge.dat');
  reset(infile);
  readln(infile,n);
  for i:=1 to n do begin
    NEWP(v[i],'',seg(v_proc),ofs(v_proc),1000);
    ACTIVATE(v[i]);
  end;
  HOLD(100);
  writeln;
  close(infile);
  halt;
end;

begin
  NEWP(main,'main',seg(main_proc),ofs(main_proc),1000);
  ACTIVATE(main);
end.
```