SETL Newsletter # 157

J.T. Schwartz
August 7, 1975

'Whenever' Dictions

1. Introduction, Specification.

Dictions of the 'whenever' type are of much potential interest, since they facilitate description and simulation of very general system models without requiring heavy use of ordinary procedural dictions. This newsletter will propose a family of dictions of this kind as supplements to SETL, and will give a few examples showing the use of the dictions proposed.

When the dictions in which we are interested are used, an environment of (pseudo -) parallel processes will come into being. For this reason, we begin by determining some important facts concerning the semantics of processes.

i. Processes are introduced as additional semantic objects within SETL. These objects are treated essentially as atoms; they do have internal components whose significance it is important to understand, but most of these components are not directly accessible. Basically a process can *execute*, and can *wait* for particular conditions to arise before it executes.

ii. The components of a process are

a. a blank atom, the *process identifier*, which identifies the process uniquely.

b. its *internal stack*, which at any given moment will show some chain of procedure invocations, bindings of the variables in these procedures to abstract addresses, return locations, a current instruction location, etc.

c. an integer *priority*.

d. an abstract address, the *error address* of the process (see below)

iii. To create a new process p, use the expression

(1) newprocess (f, e, $x_1,\ldots,x_n$).

Here the value of f must be a procedure, and e must be a variable. We call f the *initial procedure* of the process p. The abstract address bound to the variable e when (1) is evaluated becomes the error address of the process p. The procedure f must have n parameters. When p first begins to execute, the procedure f will be called, and the creation-time values of $x_1,\ldots,x_n$ will then be transmitted to f as its initial argument values.

When p is first created, its priority is zero and its internal state is null. If a SETL error occurs while a process p is executing, the value bound to the error address of the process becomes Ω.

Processes can be set members and tuple components.

iv. Namescoping rules very much like those of SETL continue to be used. Note that dynamic compilation is not being supported, so that we can continue to assume, as in SETL, that all the program text entering into a group of processes is presented for compilation at one time, and that this text is organised into a set S of namescopes and procedures; cf.O.P. II, pp. 69-89. Each variable in this text 'belongs to', i.e., is 'owned by' some procedure (in the standard SETL sense; cf. O.P. II, pp. 88-89). The creation of a process will create a base-level, process-local copy of(the variables owned by) each procedure of the set S, and recursion will create additional copies of these variables.

To make it possible for processes to interact, we allow processes to access and to write into each other's variables. For this purpose, 'process qualified' variables are made available. Such a variable is written

(2) pexpn.varname

where *pexpn* is a process-valued expression, and *varname* is a variable name. The process-qualified variable (2) is at any given moment bound to precisely the abstract address to which *varname* is bound in the process designated by *pexpn*. The operator '.' appearing in (2) has maximal priority and associates to the left.

It will often be the case that certain of the variables frequently used in some group of routines should be taken to refer, not to the data environment of the process p in which they are accessed, but to the copy of v that exists in some other process q. It is inconvenient in such cases to have to write q.v instead of v repeatedly. To make this unnecessary, we make available a declaration of the form

<u>subordinate</u> pexpn (v_1,...v_n), pexpn' (v_1'...v_n'),...;

Here v_1,...,v_n,v_1',...v_n',... are variable names, and pexpn, pexpn' are expressions (which, when evaluated, should have processes as values). This declaration acts much like a macro, substituting pexpn.v_j for each occurence of v_j within the zone in which the declaration is active. Note that a <u>subordinate</u> declaration, like a macro, belongs to some particular SETL scope and is active there.

v. When the primitive nulladic operator

<u>self</u>

is executed within a process p, the value p is returned.

The monadic primitive

priority.p

retrieves the priority of the process p, and can be used in sinister position to set the priority of p.

vi. A process can wait for a given condition to be satisfied by executing the statement

<u>await</u> C;

where C is a boolean expression. This boolean expression can involve function calls, but is not allowed to have any side effects, and should not involve any construction which could possibly loop. A process for which C evaluates to true is said to be ready; of all the processes ready at a given moment, some process of maximum priority is chosen to actually execute. It is intended that await should be implemented efficiently, and actually as a 'nonbusy' wait.

A process which executes an await statement will often be awaiting one of several disjoint events; depending on the event which occurs first, it will then take one or another action. For use in such situations, we provide a generalised await statement, having the following syntactic form:

$$\text{await: } (C_1,C_2,\ldots,C_n)\ \ell_1,\ell_2,\ldots,\ell_n;$$

Here $C_1,\ldots,C_n$ are boolean conditions, and $\ell_1,\ldots,\ell_n$ labels. This generalised await statement is equivalent to the statement group:

$$\text{await: } C_1 \text{ or } C_2 \text{ or } \ldots \text{ or } C_n; \quad \text{/* and then: */}$$

go to if C_1 then ℓ_1 else if C_2 then ℓ_2 else if ... else ℓ_n;

vii. We shall now mention a useful extension of ordinary SETL that can be particularly useful when SETL is extended in the manner described in the preceeding pages. This has to do with the return of argument values after a procedure call

(1) subr (expn,...);

Suppose that an argument of such a call is, as shown, an expression *expn*. Let the principal operator of *expn*, i.e., the operator executed first when *expn* is evaluated, be f. Then if a standard sinister meaning is defined for f (this will be the case for certain primitive operators f, and can also be the case for user-defined f), and if *subr* changes the value of the formal parameter corresponding to f, then we can let this change of value be propagated back upon return, i.e., can treat (1) as if it were

(2)

```
...
temp = expn;
subr (temp,...);
expn = temp;
```

If such an expression must be applied to several of the arguments of a call, a left-to-right rule will govern.

From our present point of view the most significant thing in all of this is that it allows us to pass 'process qualified' variables to subprocedures which will change them. I.e., if we write

```
(3)                    subr (p.v,...);
```

then the value of *p.v* will be transmitted to *subr* and if *subr* changes its first parameter a corresponding change to *p.v* will be made after return from *subr*. Note however that the call (3) behaves like

```
(4)                     temp = p. v;
                        subr (temp,...);
                        p.v = temp;
```

so that *subr* will continue to use the parameter value transmitted to it even if an await is executed in subr and the value of the variable v of the process p is then changed by p or by some other process.

viii. The body of code constituting a single 'job' or 'program' consists of a 'main program' and a group of subprocedures. Execution starts with a single process in existence, just beginning to execute the main program.

2. How to represent other useful parallel-processing constructs.

i. Terminate. This can be represented as

```
                    await false;
```

ii. Suspend, resume. To give each process a 'ready flag' which prevents process execution when not set, introduce an additional variable *readyflag*, and reinterpret

```
    await C;    as    await readyflag and C;
```

then

```
    suspend  p;    and    resume p;
```

are respectively (for suspend:)

```
                (p is ptemp). readyflag = false;
if  ptemp eq self then await self.readyflag  eq true;
```

and (for resume:)

```
                (p). readyflag = true;
```

Note however that the suspend and resume operations thus introduced can be implemented (as primitives) more efficiently than the general await operation. In particular, it is never necessary to re-evaluate a condition C awaited by a process whose ready flag is not set. However, we choose not to consider suspend and resume as primitives since we are concentrating, in conformity with the SETL spirit, on logical power rather than optimization of this kind.

2. Examples.

It is now appropriate to give a few examples showing the use of the dictions that have just been described. Our first example is a simulation, which we shall write in a way indicating how more general simulations could be handled. In our example, customers appear at a first service window at which ka servers are present, join the shortest one of ka service queues there, receive service, and then proceed to a second window to receive service, this time from one of kb servers. The arrival and service-time distributions are poissonian with respective mean inter-event times ta, tsa, tsb. The simulation runs for a time tlim; time average queue lengths are produced as output.

```
/* start of main program, which will also function as a master */
/* timer.create servers, which will initialise their queues to null */
serversa = {newprocess(server,comervar, tsa,self), 1≤ n ≤ ka};
serversb = {newprocess(server, cometvar,tsb,self), 1≤ n ≤ kb};
/* comevar is a common error variable, which should never be used */
currentime = 0;     /* simulated time */
timealarmset = nℓ;  /* set of critical times */
/* create customer arrival generator */
p = newprocess(starter, comervar, ta,self);
/* drop to lower priority and enter event-management loop */
priority self = - 1;
(while timealarmset ne nℓ and time ℓt tlim)
     ([min: t ∈ timealarmset] t) is nextime) out timealarmset;
     if nextime gt tlim then quit;;  currentime = nextime;
end while;
/* now calculate waiting time sums and print out averages  */
(∀s∈(serversa + serversb)) updateq (server, currentime);;
suma = [+: s ∈ serversa] qtime(s);
sumb = [+: s ∈ serversb] qtime(s);
print 'average waiting time in first queue', suma/tlim;
print 'average waiting time in second queue', sumb/tlim;
/* end of main program */
```

```
/* next follow a group of routines for queue maintainance  */
definef time(s);  /* retrieves accumulated queue time. */
return  s.queue(1)(1);  /* s is the queue's server     */
end time;
definef update (s, timen);  /* updates accumulated queue time */
s.queue(1)(1)  =  s.queue(1)(1) + (timen-s.queue(1)(2))* (#s.queue-1)
s.queue(1)(2) = timen;
return;
end update;
define x inq s;  /* makes queue insertion  */
update(s, currentime. master)
s.queue = s.queue + <x>
return;
end j
definef qheadout  q;
update ( q,   currentime. master);
head = q(2);
q = <q(1)> + q(3:);
return head; end qheadout;
/* next follow the 'server' and the 'customer' procedures */
define server (tserv, master);  /* master is master timer process */
queue = <<0,0>>;  /* initialise queue. first component is
                                    <time-accumulated, time-lastchanged>*/
wait:  await (# queue)  gt 1; /* queue is local to this routine and
                                                       process */
     customr  = qheadout queue;
     holdfor - tserv * log(random);
/* note that random generates a random real in the interval (0,1] */
/* the logarithm converts this to a poissonian random quantity */
customr . go = true;  /* resume customer activity */
go to wait;
end server;
```

```
define customer (master);
s = [minserv: s ∈ serversa.master] s;  /* find minimum length queue */
self inq s;
go = false;  await go eq true;
s = [minserv: s ∈ serversb. master] s; /* proceed to second window */
self inq s;
go = false; await go eq true;
terminate;
end customer;
definef sa minserv sb;  /* auxiliary function-selects shortest queue */
return if #(sa. queue)  le # (sb.queue) then sa else sb;
end minserv;
/* next follows the routine for generating new customers,    */
/* together with a 'hold for specified period' function.     */
define starter (timst, master);
start:  p = newprocess (customer, comervar, master);
/* cf. earlier comment concerning comervar */
holdfor  - timst * log (random); /* to secure Poissonian arrivals */
go to start;
end starter;
define holdfor time;
(currentime + time is timewant) in master.timealarmset;
await  master. currentime ge timewant;
return;
end holdfor;
```

Our next example is a considerably more complex simulation; namely a variant of the 'Stanford elevator' simulation described in Knuth, v. I, pp. 280-293. This involves a single elevator serving a building of *nfloors + 1* floors, where floor 0 is the basement, and floor 1 is the elevator's homing position. For this simulation we use six classes of processes: a master timer, an elevator, passenger processes, a passenger starter, and two auxiliary processes, one of which repushes dropped buttons for waiting passengers who have not been able to get on an elevator because it was full, the other of which can set the elevator's direction of motion before anyone gets into it if it has been waiting with doors open for a sufficiently long time (because of numerous people exiting). The master timer and passenger starters have quite conventional structures, and the two auxiliary processes are simple also. The elevator acts as follows: initially, it waits, empty, with doors closed, at floor 1, waiting for an external call button (up or down) to be pressed. If the first button pressed is on another floor, it prepares to move in the appropriate direction; otherwise, it prepares to open its doors. Once the elevator starts to move in a given direction it will continue to do so as long as any call remains to be serviced in that direction (at any given moment the elevator is *goingup* or *goingdown* state, or in neither state, i.e., in a *neutral* state.) When preparing to open its doors, the elevator first sees if it has more calls to service in the direction in which it has been moving, and, if not, reverses its state of motion if it has work to reach in the opposite direction. Otherwise, it passes into neutral state. Then the doors begin to open, and are fully open 2 sec. later. At this point, an anxiliary *resolver* process is started; if the elevator is still in neutral condition with its doors not closed 28 sec. later, this process will send it in the direction of the lowest floor on which a call has been signalled. After the doors have been open for 5.6 sec., or 2.5 sec. after a passenger steps into a neutral elevator if this is earlier, the doors will, if they have been unblocked, become partly closed; but if blocked during this time they will have sprung open, and will only reach the partly closed condition 4.0 sec. after the blockage was removed. Once partly closed, the doors will become fully closed in 2 sec. more; but if a passenger seeking entry appears in this period, they will open again, returning to the fully open condition after 2 sec.

When the doors are closed, the in-elevator call light for the floor just serviced is dropped, and also the call button for the direction in which the elevator will now move, unless the elevator is in its neutral state. But if the elevator is in its neutral state, a decision is taken as to its next direction of motion: toward the call on the lowest floor, if any call exists; otherwise, toward floor 1 (if the elevator is in neutral on floor 1, it merely waits). When the doors close, the auxiliary resolver process is cancelled. The elevator then accelerates for 1.5 sec. (if goingup; a little longer if goingdown), after which it proceeds to the nearest floor at which it has reason to stop. The inter-floor transit time is 5.1 sec. (if goingup; slightly longer if goingdown). On approaching its target floor, the elevator takes 1.4 sec. to decelerate, and then returns to the start of its door opening procedure.

The passenger procedure acts as follows: a passenger entering the system records his entry time and establishes a deadline after which he will leave the system (unless the elevator is on his floor, doors opening, proceeding in the desired direction). Suppose that the passenger wishes to go up. If at the moment of the passenger's arrival the doors are partly but not fully open, he will signal the door for immediate opening (possibly reopening) and join a queue of persons wishing to proceed upward. When he has advanced to the front of this queue, and provided that the elevator doors are fully open, that the elevator is proceeding in the direction desired and not full, that all those wishing to get off the elevator at its present position have done so, and that the elevator door is not blocked by another person, he steps on the elevator, presses a button indicating his current floor, leaves the queue of persons waiting at his service floor, and joins the stack of persons who will leave the elevator at his destination floor.

The elevator door is blocked for 2.5 sec. as each passenger gets on or off. Note that if the elevator is in a neutral condition when a passenger p reaches the head of his waiting queue, p will step on if either no passenger is proceeding in the opposite direction from the same floor or if p has arrived in the system before the first passenger (at the same floor) proceeding in the opposite direction. In this neutral case, the first passenger entering the elevator determines the direction in which it will move. However, if this determination is not made soon enough, the direction of motion will be determined by the pattern of call buttons pressed on other floors.)

If a passenger cannot get on the elevator by his personal deadline, he will leave the system, (and walk) if confronted by a closed elevator door, a full elevator, or an elevator proceeding in the wrong direction.

Once a passenger gets on the elevator, he remains on it until his destination floor is reached, and then gets off, leaving the system. People with a given destination floor step off the elevator in inverse order of their elevator entry.

An independent process is used to ensure that an appropriate call button will be pressed whenever a passenger is standing before closed doors waiting for an elevator.

The elevator simulation runs for some pre-specified period of time. As output, we record the average time that passengers require to be carried to their destination, the number of passengers carried, the average time wasted before giving up and walking, and the number of would-be passengers who do give up and walk.

```
/* main program of elevator simulation; also functions as */
/* master timer.initialisation, creation of processes.    */
currentime = 0;      /* simulated time */
servicetime = 0;     /* accumulated time spent getting elevator service*/
servicenumber = 0;   /* number of passengers serviced */
wastedtime = 0;      /* accumulated time wasted waiting for service */
wastednumber = 0;    /* number of passenger who have given up */
master = self;       /* this will be the master process       */
priority master = -1;  /* this will run at low priority */
/* create elevator process */
elevator = newprocess(relevator, comervar, master);
/* comervar is a error variable which is formally required but */
/* which should never be used.                                 */
/* create auxiliary resolver process                           */
resolver = newprocess (rresolver, comervar, master, elevator);
/* create auxiliary button-push process, which operates at     */
/* higher priority                                             */
pusher = newprocess (rpusher, comervar, elevator, master);
priority pusher = 1;
/* initialise button settings and waiting queues              */
upcall = nult; downcall = nult; upqueue = nult; downqueue = nult;
(0 ≤ ∀n ≤ nfloors)
     upcall(n) = false;  /* no up - calls */
     downcall(n) = false;   /* no  down - calls */
     upqueue(n) = nult;     /* no up-waiting passengers */
     downqueue(n) = nult;   /* no down-waiting passengers */
end ∀n;                     /* end of initialisation loop */
/* create passenger starter process */
starter = newprocess (rstart, comervar, elevator, master);
timealarmset = nℓ;  /* set of critical times */
(while timealarmset ne nℓ and time lt tlim)  /* timing loop */
     ([min: t ∈ timealarmset] t) is nextime out timealarmset;
     if nextime gt tlim then quit;;
end while;
```

```
/* at this point simulation is over. print results */
print  'average time spent in waiting and elevator transit',
                                          servicetime/servicenumber;
print  'average number of persons who give up and walk, '
        per unit time', wastednumber/tlim;
print  'average time wasted before giving up', wastedtime/wastednumber;
/* end of main program */
/* here follow various auxiliary routines */
definef critical (time);  /* designates time as critical moment */
time in timealarmset;
return time;
end critical;
/* now follow various queue and stack manipulating routines */
/* the queue - manipulating routines are,like the queues they */
/* manipulate, associated with the 'master' process; the stack- */
/* manipulating routines and the stacks they handle are        */
/* associated with the elevator                                */
define inupqueue (p, infloor);
/* inserts a passenger in an up queue */
master.upqueue (infloor) = master.upqueue   (infloor) + <p>;
return;
end;
define indownqueue (p, infloor)
master.downqueue(infloor)= master.downqueue(infloor) + <p>;
return;
end
definef firstinupque(ifloor);
return master.upqueue (ifloor)(1);
end;
definef firstindownqueue(ifloor)
return master.downqueue (ifloor)(1);
end;
```

```
define firstoutupqueue(ifloor); /*deletes first member of upqueue */
master.upqueue (ifloor) = master.upqueue  (ifloor)(2:);
return;
end;
define firstoutdownqueue (ifloor); /* deletes first member of
                                                       downqueue  */
master.downqueue(ifloor) =  master.downqueue (ifloor)(2:);
return;
end;
define inoutstack (ifloor,p);
/* adds member to stack of passengers with given destination */
/* outstack and elevator are assumed to be global */
subordinate  elevator (outstack);
outstack (ifloor) = outstack (ifloor) + <p>;
return;
end;
define outoutstack (ifloor);
subordinate  elevator (outstack);
/* deletes member from stack of passengers with given destination */
/* outstack and elevator are assumed to be global */
outstack (ifloor) (# outstack(ifloor)) = Ω;
return;
end;
definef firstoutstack(ifloor);
/* elevator and outstack are assumed to be global */
return elevator.outstack(ifloor) (# outstack(iflooor));
end firstoutstack;
/* now follow the procedures which define the main processes of */
                                                  the simulation */
define relevator (master);        /* elevator procedure */
subordinate  master (currentime, nfloors);
resolver = master.resolver;  /* obtain resolver process pointer from
                                                          master */
/* initialise elevator state and stacks of riders */
dooropen = false; doorfullopen = false; floor = 1;gettinginout= false;
goingup = false; goingdown = false; neutral = true;
holdit = false;
```

```
        carcall = nult; outstack = nult;
        (0 ≤ ∀n ≤ nfloors)
            carcall(n) = false;  /* initally, car not called */
            outstack(n) = nult;  /* initially, no passengers */
        end ∀n;
waitingposn: await 0 ≤ ∃n ≤ nfloors | (upcall(n) or downcall(n));
        if not (upcall(1) or downcall(1))  then
            decision; /* call decision routine to set direction of motio
            go to preparemove;
        end if;
preparetoopen:  /* first adjust elevator state */
        ishighercall = floor < ∃n ≤ nfloors | (upcall(n) or
                                              downcall(n) or carcall(n))
        islowercall = floor > ∃n ≥ 0 |(upcall(n) or downcall(n) or
                                              carcall(n));
        if goingup and not ishighercall then
            goingup = false;
            if islowercall then goingdown = true;;
        else if goingdown and not islowercall then
            goingdown = false;
            if ishigercall then goingup = true;;
        end if;
        if not (goingup or goingdown) then neutral = true;;
        dooropen = true;  /* now door starts to open */
opening:  holdfor 20;
opened:   doorfullopen = true;  /* now door is fully open */
        holdit = false;     /* drop holdit if it has been set */
        /* start direction - resolution process */
        resolver.proceed = true; resolver.ringtime = currentime + 280;
        startclosetime = critical (currentime + 16);
        await currentime ge startclosetime;
startclose:  closetime = critical (currentime + 40);
        await:(currentime ge closetime, gettinginout)
                                       almostclosed,  startclose;
        doorfullopen = false;
```

```
almostclosed: closetime = critical (currentime + 20);
     await:   (currentime ge closetime, holdit) preparemove, opening;
preparemove:  dooropen = false;
     carcall(floor) = false;
     if not goingdown then upcall (floor) = false;;
     if not goingup then downcall (floor) = false;;
     if neutral then decision;; /* call routine to set direction
                                                       of motion */
     if neutral then  /* must be sitting at floor 1 */
          go to waitingposn;
     end if;
     resolver.proceed = false;  /* cancel automatic resolver·process */
     holdfor if goingup then 15 else 23; /* period of acceleration */
moving: floor = floor + if goingup then 1 else - 1;
     holdfor if goingup then 51 else 61; /* interfloor transit time */
     reasontostop = carcall(floor) or goingup and upcall(floor)
                         or goingdown and downcall(floor) or
     (goingingup and not floor < ∃n ≤ nfloors |(upcall(n) or
                                                 downcall(n) or carcall(
     or goingdown and not floor > ∃n ≥ 0 |(upcall(n) or downcall(n) or
                                                      carcall(n)))
     and (floor eq 1 or upcall(floor) or downcall(floor));
     if reasontostop then
          holdfor  14;  /* deceleration time */
          go to preparetoopen;
     end if;
     /* else */   go to moving;
     end elevator;                /* end of elevator procedure */
     define decision;
/* auxiliary routine to decide direction of motion when elevator is
                                                  in neutral state */
subordinate  master(upcall,downcall),elevator(carcall, goingup
                                                  goingdown, neutral);
j = 0;
if not 0 ≤ ∃j ≤ nfloors | j ne floor and (carcall(j) or upcall(j) or
                    downcall(j)) then if floor ne 1 then j = 1; endif;
if j ne 0 then if j lt floor then goingdown = true else
                                        goingup = true; end if;end if;
```

```
if goingdown or goingup then neutral = false;;
return;
end;
define rpassenger (myfloor, mypatience, destfloor, master,elevator);
/* passenger procedure */
subordinate    master (currentime, upcall, downcall,
               servicenumber, wastednumber,servicetime,wastedtime),
               elevator(dooropen,doorfullopen,goingdown, goingup,
               neutral, holdit, floor, outstack, gettinginout,
                                    carcall, startclosetime);
elapsedtime = 0; arrivedtime = currentime;
deadline = critical (arrivedtime + mypatience);
if destfloor lt myfloor then go to down;;
if dooropen and floor eq myfloor and not doorfullopen and not
                               goingdown then holdit = true;;
inupqueue (self, myfloor);    /* join queue of up waiters */
waitmoreup:  /* main waiting point for up waiters */
     await: (currentime ge deadline,
          doorfullopen and not goingdown and floor eq myfloor and
          firstinupqueue(myfloor)) eq self and not full( ) and
          (# outstack(myfloor)) eq 0) expiredup, amfirstup;
expiredup:  /* if miss this turn then leave system */
     await:  (not dooropen or floor ne myfloor or not goingup
             or (doorfullopen and (# outstack(myfloor))eq 0 and full( )
             doorfullopen and (# outstack(myfloor))eq 0 and
             floor eq myfloor and not full( ) and not goingdown and
             firstinupqueue (myfloor) eq self) giveup, amfirstup;
amfirstup;  /* here are at head of upgoing group - */
            /* compare with priority of first downgoer */
     cango = if firstindownqueue(myfloor) is fdown eq Ω then true
             else arrivedtime lt fdown.arrivedtime;
if not cango then
     await:  fdown ne firstindownqueue (myfloor);
             go to waitmoreup;
end if;
```

```
     /* at this point passenger will be able to get on elevator */
     await not gettinginout;  /* nobody else getting in or out */
     firstoutupqueue(myfloor);  /* leave queue */
entered:  gettinginout = true;
     inoutstack(destfloor,self); /* join group in elevator */
     carcall(destfloor) = true;  /* press button */
     if neutral then
          neutral = false;
          startclosetime = startclosetime min critical(currentime + 25);
          if destfloor lt myfloor then
               goingup = true;
          else
               goingdown = true;
          end if;
     end if neutral;
     holdfor 25;
     gettinginout = false;
     await floor eq destfloor and firstoutstack(destfloor) eq self
          and doorfullopen and not gettinginout;
     gettinginout = true;
     outoutstack(destfloor);  holdfor 25;
     servicetime = servicetime + currenttime-arrivedtime;
     servicenumber = servicenumber + 1; gettinginout = false;
     terminate;
giveup:   wastedtime = wastedtime + currentime-arrivedtime;
     wastednumber = wastednumber + 1;
     terminate;
down:     /* note that the code which starts here is symmetrical */
          /* with the code for the upgoing passengers */
     if dooropen and floor eq myfloor and not doorfullopen and not
          goingup then holdit = true;
     indownqueue (self,myfloor);
```

```
waitmoredown:  /* main waiting point for down waiters */
     await:    (currentime ge deadline,
          doorfullopen and not goingup and floor eq myfloor and
          firstindownqueue(myfloor) eq self and not full( ) and
          (# outstack(myfloor))  eq 0) expiredown,amfirstdown;
expiredown:    /* if miss this turn then leave system */
     await:    (not dooropen or floor ne myfloor or not goingup
           or  (doorfullopen and (# outstack(myfloor)) eq 0 and full( )),
not goingup and doorfullopen and (# outstack(myfloor)) eq 0 and floor eq
           myfloor and not full( )and firstinupqueue(myfloor) eq self)
                    giveup, amfirstdown;
amfirstdown:   /* here are at head of downgoing group - */
               /* compare with priority of first upgoer */
          cango = if firstinupqueue (myfloor) is fup eq Ω then true
                    else arrivedtime le fup.arrivedtime;
          if not cango then
               await fup ne  firstindownqueue (myfloor);
               go to waitmoredown;
          end if;
     /* at this point passenger will be able to get on elevator */
     await not gettinginout;/* nobody else getting in or out */
     firstoutdownqueue(myfloor);    /* leave queue */
     go to entered;
     end rpassenger;
     define rresolver (master, elevator);
     /* process forcing decision about state from first call    */
     /* if door stays open for long time                        /*
wait:     await ringtime ge master.currentime and proceed;
          proceed = false;
decide:   decision;            /* call decision routine to attempt
                                          to set direction of motion */
     await:   (elevator.state ne neutral,
               0 ≤ ∃n ≤ nfloors |(upcall(n) or downcall(n))) wait,decide;
     end rresolver;
```

```
      define rpusher (elevator, master);
      /* process to push button if passengers are waiting */
wait:   await:(not elevator.dooropen and
              master.firstinupque(elevator.floor) ne Ω and
                     not elevator.upcall(elevator.floor),
                     not elevator.dooropen and master.firstindownqueue
                     (elevator.floor) ne Ω and not elevator.downcall
                     (elevator.floor) pushup, pushdown
pushup:    upcall(elevator.floor) = t; go to wait;
pushdown:   downcall(elevator.floor) = t; go to wait;
      end rpusher;
      define rstart(elevator,master);  /* creates passengers for entry
                                                      in system */
again:     newtime = critical (master.currentime-tbetween * log(random));
      /* the 'log' serves to generate Poissonian arrivals. tbetween */
      /* is an external parameter which sets the mean inter-arrival time
      await  master.currentime ge newtime;
      /* now generate random source and destination floors (though */
      /* a more realistic traffic pattern might be better) and use */
      /* a fixed time-to-exhaustion of patience for each passenger. */
      sourcefloor = intpart(random, master.nfloors)
      (while (intpart(random,master.nfloors) is destfloor) eq
                                                  sourcefloor) noop;;
      p = newprocess(rpassenger,comervar,sourcefloor,patienceconstant,
                                          destfloor, master,elevator);
      go to again;
      end rstart;
      definef full;
      /* parameterless function to determine whether elevator is full */
      /* the quantities elevator, master, maxcap,nfloors, and outstack
                                              are assumed to be global */
      return ([+: 0≤n≤ master.nfloors] elevator.outstack(n) ge
                                                  elevator.maxcap;
      end full;
      definef intpart(x,n);
      /* auxiliary routine to generate a random integer with maximum n *
```

```
        /* from x, which is a random real between 0 and 1 */
try:        if bot (x * (n + 1) is m) lt(n + 1) then return m;;
        go to try;
        end intpart;
        /*note that the holdfor routine is precisely as in the */
        /* preceeding simulation.                              */
```

3. Implementation and efficiency cost estimate.

We shall now sketch an implementation of the semantic mechanisms that have been described in the preceeding pages, and estimate the efficiency cost of providing these mechanisms. We propose to provide the basic 'monitoring' capability needed to support the await diction as follows: With each (name resolved) variable v in a process p, we will associate a list of processes q, namely those processes currently awaiting conditions involving v. We call this list the *monitoring list* of v. We will also maintain a *ready list* of processes, namely those processes awaiting a condition that might be true since one of the variables involved in it has changed since the condition was last tested. Each await condition will reference all the variables upon which it depends. At any given moment, the highest priority process on the active list will be executed. A process which executes an await whose condition fails is enqueued on the monitoring list of each variable appearing in the await, and control is then passed to the highest priority process on the active list. A process which has been waiting, but which has moved to the active list by virtue of a change in some variable v, is logically dequeued from the monitor list of v when it becomes active, and dequeued from the lists of all remaining variables involved in the await if the await condition is satisfied; otherwise it is logically requeued on any monitor lists from which it may have been removed.

Whenever the priority of a process p is changed, its new priority is compared to that of the process q currently executing, and also to the highest priority process r or the active list; then the process with highest priority executes while the others are returned to the active list.

The internal environment of a process is defined by its invocation stack and a table which gives the number of times each-subprocedure has been invoked within the process; of course, the invocation stack entries, each of which represents a generation of some variable, point to variable values stored in a common heap.

A process stack will be maintained as a list of stack segments, each segment corresponding to the local environment of one particular process. This imposes a call/return overhead which is slightly, but not significantly, larger than the corresponding overhead in ordinary SETL.

The implementation just outlined does not impose any overhead on load operations, but when a variable is changed its monitoring list must be checked, and processes of higher priority found on this list must be moved to the active list. Normally there will be no such processes, so the store operation will simply stretch out to 5 machine cycles from 1. The overhead cost of this should be modest both for SETL and for programs in a more array oriented language: probably less than 50%. Note that stores to compiler temporaries need not be checked, and that global analysis can be used to detect variables which never appear in an <u>await</u> statement; stores to such variables clearly do not need to be checked either.