COMBINEDSIMULATION

Reference Manual

CONTENTS:

1. BASIC CONCEPTS

The class outline below shows the most essential user-attributes of class `COMBINEDSIMULATION`.

```
SIMULATION class COMBINEDSIMULATION;
begin
  class CONTINUOUS;
  begin
    procedure START; ... ;
    procedure STOP; ... ;
    Boolean procedure ACTIVE; ... ;
  end;

  class VARIABLE(STATE); real STATE;
  begin
    real RATE;
    procedure START; ... ;
    procedure STOP; ... ;
    Boolean procedure ACTIVE; ... ;
  end;

  class REPORTER;
  begin
    procedure SETFREQUENCY(F); real F; ... ;
    procedure START; ... ;
    procedure STOP; ... ;
    Boolean procedure ACTIVE; ... ;
  end;

  procedure WAITUNTIL(B); name B; Boolean B; ... ;

  real DTMIN,DTMAX, MAXRELERROR,MAXABSERROR;
end;
```

COMBINEDSIMULATION is a subclass of class SIMULATION. The discrete events of the simulated system may be described in the usual way by means of the class PROCESS. Objects of class PROCESS are called "discrete processes".

The continuous state changes of the system may be described by means of the two new classes - CONTINUOUS and VARIABLE.

In one or more subclasses of class CONTINUOUS the user can express the difference and/or differential equations of the model in question. A CONTINUOUS-object, called a "continuous process", will "continuously" execute its actions after its START-procedure has been called. The continuous process is said to be "active" during this period. The active phase will cease when the STOP-procedure of the process is called.

Objects of class VARIABLE is used to represent state variables whose continuous variation is expressible as difference or differential equations. For such a variable the attribute STATE denotes its current value, while RATE denotes its derivative with respect to time. STATE is changed "continuously" according to the value of RATE, as computed by the active continuous processes. The VARIABLE-object is said to be "active" during this period. The active phase will cease when the object's STOP-procedure is called.

Class REPORTER is used for reporting purposes. A REPORTER-object may have its user-defined actions executed with a specified frequency. The frequency is specified by procedure SETFREQUENCY and the object's continual execution is started and stopped by the procedures START and STOP, respectively.

Discrete events, which are scheduled to occur at a specified point in time (EVTIME), are called "time-events". This type of event is well-known to a user of class SIMULATION.

The procedure WAITUNTIL makes it possible to schedule a new type of discrete event, a so-called "state-event". This is an event planned to take place as soon as the state of the model fulfils a specified condition.

Between the discrete events the state of the model is advanced in steps using numerical integration. DTMIN and DTMAX are used to specify the minimum and the maximum allowable step-size, respectively. MAXRELERROR and MAXABSERROR may be used to specify the relative and absolute integration error that can be tolerated at each step.

2. EXECUTION OF A SIMULATION

A simulation is controlled behind the scenes, so to speak, by an object called the "monitor".

It is the monitor's responsibility that

(1) the state of the model changes "continuously" between the discrete events,

(2) the events (time- as well as state-events) take place at the right time and in correct order, and

(3) the active REPORTER-objects have their user-defined actions executed with the specified frequency.

The monitor updates the state of the model between the discrete events as prescribed by the active CONTINUOUS processes. The updating is performed in time steps using numerical integration (see Section 2.2.1). The step-size is adjusted so that the requested accuracy requirements are met, and so that no events take place within a step. In order to reduce rounding errors, "quasi double-precision summation" is used in updating the state of the model (see Section 2.2.3).

The monitor ensures that the scheduled events take place. The time-events take place precisely a their given time points (EVTIME); on the other hand, state-events, that is, events scheduled by procedure WAITUNTIL, are time-determined with a certain error (DTMIN, see Section 3.1 and 3.3).

Moreover the monitor takes care that the user-defined actions (inner) of the active REPORTER-objects are executed as frequently as prescribed, which for a REPORTER-object with frequency F means:

F > 0: at uniformly spaced intervals of length F time units and also at event times.

F = 0: at the end of each time step (which includes event times).

F < 0: only at event times.

When an event takes place, the user-defined actions of all active REPORTER-objects, regardless of frequency, are executed - both immediately before and after the event.

The monitor is active between the events, even when they are "concurrent". However, an event which is initiated by "direct" activation - that is, by `activate P` or the like - will not be "detected" by the monitor.

The work of the monitor is outlined below.

```
class MONITOR;
while more projected events do
begin
  DT:=0;
  execute all active CONTINUOUS-objects;
  execute all active REPORTER-objects;

  while no event now do
  begin
    take an integration step, DT, fulfilling the
    accuracy requirements;

    if a state-event was passed
    then determine event time and reduce step accordingly;

    execute active REPORTER-objects when requested;
  end;

  let an event take place now;
end;
```

In order to perform its tasks, the monitor has access to the following lists:

1) class SIMULATION's event list, SQS, which represents the scheduled time-events

2) a list of "wait-notices" representing the planned state-events

3) a list of active CONTINUOUS-objects

4) a list of active VARIABLE-objects

5) a list of active REPORTER-objects

The actions of the active CONTINUOUS- and REPORTER-objects are executed by the monitor as so-called "RESUME-chains":

(1) the monitor RESUMEs the first object in the chain,

(2) each object RESUMEs its successor, and

(3) the last object in the chain RESUMEs the monitor.

The monitor will take care that all processes of the model are synchronised. Two PROCESS-objects, called CONTROLLER1 and CONTROLLER2, control the activity of the monitor, and ensure that the user does not destroy the synchronisation (for example by activating a discrete process directly from a continuous process).

2.1. Time advance

Between discrete events the state of the model is advanced in time steps using numerical integration. The step-size varies within limits specified by the user (DTMIN, DTMAX) and is adjusted so that the integration error in each step is less than the error limit specified by the user (RELERROR, ABSERROR).

Usually the step-size will vary between DTMIN and DTMAX. However, steps smaller than DTMIN may be necessary in order to assure that no time-events are passed within a step. In using any of the fixed step-size integration methods (see Section 2.2) DTNEXT is equal to DTMAX. The event times of state-events are determined with an accuracy of DTMIN.

If the step-size becomes too small, that is, time "stands still", an error message is given and the simulation stops (see Section 2.2.3).

The execution of a single step is described below.

First the actual step-size, DTNOW, is chosen. If there are active continuous processes, then DTNOW is set to

```
MIN(DTNEXT,NEXTEVENTTIME-LASTTIME)
```

where DTNEXT denotes a proposal for the step-size, NEXTEVENTTIME denotes the event time of the next event, and LASTTIME denotes the start time of the step.

If there are no active continuous processes, then DTNOW will instead be set to

```
MIN(DTMAX,NEXTEVENTTIME-LASTTIME)
```

unless there are no state-events, in which case DTNOW is set to

```
NEXTEVENTTIME-LASTTIME
```

Next, using numerical integration, a STATE increment, DS, corresponding to the TIME increment, DTNOW, is determined for each active VARIABLE-object.

If, however, the integration error for merely one VARIABLE-object is greater than the acceptable error, the step computation is redone with a new step of size

```
MAX(1/2*DTNOW,DTMIN)
```

The process is repeated until either acceptable accuracy is achieved, or the step-size becomes smaller than DTMIN. In the latter case the virtual procedure INTEGRATIONERROR is called and the simulation is stopped. The user may rewrite this procedure if he wishes an alternative course of action.

When the step is acceptable, the assertion holds that

```
TIME = LASTTIME+DTNOW
```

and for the active VARIABLE-objects that

```
STATE = OLDSTATE+DS
```

where OLDSTATE denotes the value of STATE at the beginning of the step, that is, at LASTTIME (quasi double-precision summation is not taken into account in this explanation).

Next, at the end point of the step the monitor determines if a state-condition has been fulfilled, that is to say, if a state-event has been passed within the step. In this case, the event time of the earliest state-event is determined and the step is reduced accordingly. The event time is determined with an accuracy of DTMIN by a binary search within the step; the model state is determined by Hermite interpolation - see Section 2.2.2.

Lastly, the active REPORTER-objects have their user-defined actions executed at the correct times. For those reporting times which were passed within the actual step, the model state is determined by Hermite interpolation - see Section 2.2.2.

Below is given an algorithm outline showing how, in principle, the monitor advances time between events.

```
while TIME<NEXTEVENTTIME do
begin
  LASTTIME:=TIME;

  if there are any active CONTINUOUS-objects then
  begin
    for each active VARIABLE-object do OLDSTATE:=STATE;
    DTNOW:=MIN(DTNEXT,NEXTEVENTTIME-LASTTIME);

  INTEGRATION:
    for each active VARIABLE-object do DS:= ... ;
    if the error is unacceptable then
    begin
      if DTNOW<=DTMIN then
      begin
        INTEGRATIONERROR;
        if REPEATSTEP then goto INTEGRATION;
      end;
      DTNOW:=DTNEXT:=MAX(1/2*DTNOW,DTMIN);
      goto INTEGRATION;
    end;
    for each active VARIABLE-object do STATE:=OLDSTATE+DS;
    if DTNOW=DTNEXT then DTNEXT:= ... ;
    DTNEXT:= ... ;
  end
  else if there are any planned state-events
       then DTNOW:=MIN(DTMAX,NEXTEVENTTIME-LASTTIME)
  else DTNOW:=NEXTEVENTTIME-LASTTIME;

  TIME:=LASTTIME+DTNOW;

  if a state-event was passed and TIME<NEXTEVENTTIME then
  begin
    DTLOWER:=0;
    while DTNOW-DTLOWER>DTMIN do
    begin
      DT:=MAX(1/2*(DTLOWER+DTNOW),DTMIN);
      TIME:=LASTTIME+DT;
      determine the state of the model at TIME (using
      interpolation);
      if a state-condition is fulfilled then DTNOW:=DT
      else DTLOWER:=DT;
    end;
    TIME:=LASTTIME+DTNOW;
    if DT=DTLOWER
    then determine the state of the model at TIME;
  end;

  while NEXTREPTIME<=TIME do
  begin
    determine the state of the model at
    NEXTREPTIME (using interpolation);
    execute relevant active REPORTER-objects;
    if NEXTREPTIME>TIME
    then re-establish the state of the model at LASTTIME+DTNOW;
  end;

  execute all active REPORTER-objects having FREQUENCY=0;
end;
```

2.2. Numerical methods

2.2.1. Integration

COMBINEDSIMULATION allows the user to describe continuous state changes by a system of ordinary first-order differential equations:

$$dy/dt = f(t,y)$$

where t is the independent variable, y is a vector of differentiable state variables, and f is a vector function of t and y.

The variable t is the model time, TIME, while y is represented by active VARIABLE-objects, where

d STATE / d TIME = RATE

The function f is defined by active CONTINUOUS-objects which "continuously" compute the VARIABLE-objects' RATEs. Note that f may be exchanged in connection with a discrete event.

The monitor will integrate the actual equation system numerically, that is, it will cause the STATEs of the active VARIABLE-objects to change according to their RATEs.

The user is offered the choice of six numerical integration methods: Runge-Kutta-England, Euler, Trapezoidal, Adams, Simpson, and Improved Heun. Any of these methods may be chosen at any time during a simulation. If the user does not specify an integration method, the program uses the fourth-order Runge-Kutta-England method, RKE for short (ref. 2).

The Boolean variables EULER, ADAMS, TRAPEZ and SIMPSON can be used to select the desired integration method according to the following table.

EULER	ADAMS	TRAPEZ	SIMPSON	Method	Order	Step-size
false	false	false	false	RKE	4	variable
true	false	false	false	EULER	1	fixed
-	true	false	false	ADAMS	2	fixed
-	false	true	false	TRAPEZ	2	fixed
-	true	true	false	HEUN	2	fixed
-	false	-	true	SIMPSON	3	fixed
-	true	-	true	no name	4	fixed

In using the RKE-method the step-size is variable and it is possible to control the integration error by setting MAXRELERROR and/or MAXABSERROR. The tolerated error may even be specified separately for each VARIABLE-object by setting the VARIABLE-attributes RELERROR and ABSERROR.

In using a fixed step-size integration method the step-size remains constant at the maximum prescribed by the user, DTMAX, unless an event occurs within a step.

In order to compute the function value of f at a given time, that is, the RATE-values, the monitor executes the user-defined actions of all active CONTINUOUS-objects. The execution takes place as a RESUME-chain in the same order as the list of active CONTINUOUS-objects. The number of times that the user-defined actions of all active CONTINUOUS-objects are executed for each integration step is listed below:

Integration method	Times
RKE	9
EULER	1
ADAMS	2
TRAPEZ	2
HEUN	2
SIMPSON	3

In a continuous process the order in which the equations are written is left to the user. Because COMBINEDSIMULATION does not change the execution order of the equations, a correct sequencing is the responsibility of the user. To prevent unintentional time delays from being introduced into the model dynamics, the user must make sure that variables occurring on the right hand side of an equation have values which reflect the current state of the system. The user can determine the order of evaluation within each continuous process, and the continuous processes themselves may be ranked by giving each a priority (by calling procedure SETPRIORITY).

2.2.1.1. Euler's method

When Euler's method is selected, EULER:=true, time is advanced with a fixed step length of DTMAX. However, time steps smaller than DTMAX may be taken in order to assure that no event is passed within a step. User-specified accuracy requirements (RELERROR, ABSERROR) are not taken into account.

Let y be the solution to the initial value problem

$$dy/dt = f(t,y) , \quad y(t0) = y0$$

Euler's method approximates the solution y at the points t_0, t_0+h, t_0+2h, ... , with u, where

$$u(t+h) = u(t)+h*f(t,u(t)) \quad \text{and} \quad u(t_0) = y(t_0)$$

The method is efficient with respect to computer time because only one computation of f is performed per integration step (versus 4.5 when using RKE). On the other hand, the method is not very accurate. The integration error

$$u_i(t+h)-y_i(t+h)$$

for a given variable y_i and step-size h, is approximately proportional to h^2 when h is small, that is $O(h^2)$. In comparison, when RKE is used, the error is $O(h^5)$.

Euler's method is well-suited

- when the accuracy is of lesser importance (e.g., in the first trial simulations),

- when the continuous changes of the simulated system are described by difference equations (e.g., in simulations of models of the System Dynamics type where the equations are of the form `Y.STATE=Y.LASTSTATE+DT*`(expression)).

An integration step with Euler's method is executed by the monitor as described below.

```
 1.  LASTTIME:=TIME;
 2.  DTNOW:=MIN(DTMAX,NEXTEVENTTIME-LASTTIME)
 3.  VAR:-FIRSTVAR;
 4.  while VAR=/=none do inspect VAR do
 5.  begin
 6.    OLDSTATE:=STATE;
 7.    DS:=DTNOW*RATE;
 8.    STATE:=OLDSTATE+DS;
 9.    RATE:=0;
10.    VAR:-SUCVAR;
11.  end;
12.  DT:=DTNOW; TIME:=LASTTIME+DTNOW;
13.  RESUME(FIRSTCONT);
```

Comments:

2 The current step size is chosen. It is assumed that all active continuous processes have just been executed so that all `RATE`s have been computed.

3 The variable `VAR` traverses the list of active `VARIABLE`-objects. `FIRSTVAR` denotes the first object in this list.

7-8 The `STATE`-increment `DS` corresponding to the `TIME` increment `DTNOW` is determined and added to `LASTSTATE`, the value of `STATE` at the starting point of the step.

9 Unless `RATE` is computed by the active continuous processes, its value must be zero (difference equations are "integrated" with `RATE`=0).

12-13 Time is advanced and all active continuous processes are executed (to compute the `RATE`s). `FIRSTCONT` denotes the first `CONTINUOUS`-object of the list of active continuous processes. The execution takes place as `RESUME`-chain where the monitor itself is the last link.

2.2.1.2. Runge-Kutta-England's method

Unless the user sets one or more of the Boolean variables EULER, ADAMS, TRAPEZ, or SIMPSON to true, the program selects the fourth-order variable step-size Runge-Kutta-England integration method, RKE.

The method has been described by England (ref. 2, process 9) and is further examined by Shampine and Watts (ref. 5).

RKE is a fourth-order method for numerical solution of the initial value problem

$$dy/dt = f(t,y) \ , \ y(t_0) = y_0$$

where y is a vector of differentiable state variables (VARIABLE-objects), and y_0 is the vector's initial value.

The method has several advantages that make it appropriate for combined simulation. First of all, it is easy to change the step-size. This is very important in a combined simulation where events are not normally spaced uniformly in time. Secondly, RKE is self-starting, thus there is no loss of efficiency when restarting from an event. Thirdly, there is the possibility, by means of interpolation, of determining the state of the model at time points within an integration step (see Section 2.2.2).

At each step the RKE-method estimates the integration error (the local truncation error), and tries to meet the user-prescribed accuracy requirements (RELERROR, ABSERROR).

RKE is a two-step method involving nine function evaluations over the two steps. A step of length DTNOW is divided into two ordinary Runge-Kutta integration steps, each having a length of H=1/2*DTNOW. The integration error is estimated in the middle of the second sub-step. If the error is unacceptable, the step is discarded, and a fresh step of half the length, 1/2*DTNOW, is attempted. The following is a detailed description of one RKE-step.

Let $t_1=t_0+H$ and $t_2=t_1+H=t_0+2H$. Then a RKE-step from t_0 to t_2, a computation of $y(t_2)$, involves the evaluation of the following equations:

$$c_1 = H*f(t_0,y(t_0))$$
$$c_2 = H*f(t_0+H/2,y(t_0)+c_1/2)$$
$$c_3 = H*f(t_0+H/2,y(t_0)+(c_1+c_2)/4)$$
$$c_4 = H*f(t_0+H,y(t_0)-c_2+2*c_3)$$

$$y(t_1) = y(t_0)+(c_1+4c_3+c_4)/6$$

$$c_5 = H*f(t_1,y(t_1))$$
$$c_6 = H*f(t_1+H/2,y(t_1)+c_5/2)$$
$$c_7 = H*f(t_1+H/2,y(t_1)+(c_5+c_6)/4)$$

$$ERROR = (-c_1+4c_3+17c_4-23c_5+4c_7-r)/90 \ , \ \text{where}$$

$$r = H*f(t_1+H, y(t_0)+(-c_1-96c_2+92c_3-121c_4+144c_5+6c_6-12c_7)/6)$$

If ABS(ERROR)<=ABS(ABSERROR)+ABS(RELERROR*$y(t_1)$), then
$y(t_2) = y(t_1)+(c_5+4c_7+H*f(t_2,y(t_1)-c_6+2c_7))/6$ + ERROR.

Here the c's, y, ERROR, ABSERROR and RELERROR are all vectors.

Only if the estimated integration error, ERROR, is acceptable for all state variables will the $y(t_2)$ values be computed. If the step is acceptable, the algorithm involves nine function evaluations, that is, four and a half evaluations per Runge-Kutta step.

Local extrapolation is used to achieve a better accuracy in the computation of $y(t_2)$, thus the variable ERROR is used as a correction value.

In the implementation of the RKE-algorithm attempts have been made to hold the storage requirements down by using as few auxiliary variables as possible. The five VARIABLE-attributes A1, A2, A3, A4 and A5 are used with the following meaning:

A1 denotes c_1
A2 denotes c_2 and c_6
A3 denotes c_3 and c_7
A4 denotes c_4 and ERROR
A5 denotes c_5

The function value of f is evaluated by assigning TIME and the STATEs of all active VARIABLE-objects suitable values, and thereafter execute all active CONTINUOUS-objects.

The following assertions hold:

t_0 = LASTTIME, t_1 = LASTTIME+H, t_2 = LASTTIME+DTNOW
$y(t_0)$= OLDSTATE, $y(t_1)$= OLDSTATE+DSH, $y(t_2)$= OLDSTATE+DS

As long as the estimated error is unacceptable, the step-size is halved:

```
DTNOW:=MAX(1/2*DTNOW,DTMIN)
```

However, if the step-size is reduced to DTMIN and the error is still unacceptable, then the virtual procedure INTEGRATIONERROR is called. Unless the user has redefined this procedure, the simulation will be stopped with the following error message:

```
THE REQUESTED INTEGRATION ACCURAY CAN NOT BE ACHIEVED
```

RKE is a fourth-order method, which means that the integration error at each step, H, is $O(H^5)$. The integration error itself is estimated with an error of $O(H^6)$. This estimate is asymptotically correct, that is to say that the estimated error divided by the true error approaches one as the step-size approaches zero.

When the error is acceptable, it will be used for local extrapolation. It will also be used to predict the length of the next integration step. The method is described below.

Assume that the next step is of length DTNEXT = K*DTNOW. The factor K is determined in the following way.

Since the integration error for small values of the step-size, H, is approximately proportional to H^5, the integration error for the i'th variable, Yi, at the next step is expected to be

$$K^5 * ERROR_i$$

where $ERROR_i$ is the estimated error in Y_i.

Therefore a local optimal choice for the factor K is found by choosing K as large as possible under the restriction that

$$K^5 * ABS(ERROR_i) <= ABS(ABSERROR_i)+ABS(RELERROR_i*Y_i(t_1))$$

holds for every i, that is, by selecting

$$K = ERRORRATIO^{1/5}$$

where ERRORRATIO denotes

$$\min_i(ABS(ABSERROR_i)+ABS(RELERRROR_i*Y_i(t_1))/ABS(ERROR_i))$$

However, selecting K so that the expected error is exactly equal to the maximum error allowed may frequently cause the error to become too large so that the step have to be discarded. In order to avoid this situation a more conservative approach is used. K is selected so that the expected error is half the acceptable error, that is,

$$K = (1/2*ERRORRATIO)^{1/5}$$

In addition, K must be less than 2. That is to say, DTNEXT is at most twice DTNOW. Since DTNEXT must be less than DTMAX, DTNEXT is computed as follows:

```
DTNEXT:= MIN( MIN(2,K)*DTNOW , DTMAX)
```

The determination of K is performed by setting ERRORRATIO to $(2^5)*2$, which corresponds to a doubling of the step-size. During a pass through the list of active VARIABLE-objects ERRORRATIO is set to

$$(ABS(ABSERROR_i)+ABS(RELERROR_i*Y_i(t_1)))/ABS(ERROR_i)$$

each time a VARIABLE-object, Y_i, satisfies the following inequality

$$ABS(ABSERROR)+ABS(RELERROR*Y_i(t_1)) > ABS(ERROR_i)*ERRORRATIO$$

By using this procedure arithmetic overflow is avoided in the computation of ERRORRATIO when $ERROR_i$ is equal or close to zero.

2.2.1.3. The trapezoid method

The second-order trapezoid integration method is selected by setting the Boolean variable `TRAPEZ` to `true`. At the same time the Boolean variables `ADAMS` and `SIMPSON` must be `false`.

The trapezoid method is sometimes referred to as Improved Euler.

The trapezoid method is a fixed step-size method. As with the other fixed step-size methods the step-size is usually kept at its maximum value, `DTMAX`, and there is no integration error check.

The mathematics of this method is given below. The integration step, `DTNOW`, is divided into two equal sub-steps, each of width `H = 1/2*DTNOW`.

```
A1 = H*f(t,y(t))
A2 = H*f(t+h,y(t)+A1)
y(t+DTNOW) = y(t)+(A1+A2)/2
```

Each step involves two function evaluations, that is, at each step the user-defined actions of all active `CONTINUOUS`-objects are executed twice.

2.2.1.4. Adams' method

Adams' second-order fixed step-size integration method may be selected by setting the Boolean variable ADAMS to `true`. At the same time the Boolean variables `TRAPEZ` and `SIMPSON` must be `false`.

Below is given the mathematics of the method. The error criterion (`RELERROR`, `ABSERROR`) is not used.

```
DSH = 0.5*DTNOW*f(t-DTNOW,y(t-DTNOW))
A1 = DTNOW*f(t,y(t))
y(t+DTNOW) = 1.5*A1-DSH
```

Only one function evaluation is performed at each integration step. The method is a so-called implicit method, which means that a function value of the previous step, `f(t-DTNOW,y(t-DTNOW))`, is remembered and enters into the computation of the current step.

As can be seen, this computation scheme gives problems both immediately before and immediately after a discrete event, because the required old function value, `f(t-DTNOW,y(t-DTNOW))`, has not been computed. In these cases, the trapezoid method is used (see Section 2.2.1.3).

2.2.1.5. Simpson's method

The user may select Simpson's fixed step-size integration method by setting the Boolean variable SIMPSON to true.

The Boolean variable ADAMS must be false; otherwise, a combination of Adam's and Simpson's method is used. This combination has not been analysed in detail. However, experiments have shown promising results.

The following gives the mathematics of Simpson's method. The integration step, DTNOW, is divided into two equal sub-steps, each of width H = 1/2*DTNOW. The error criteria, RELERROR and ABSERROR, are not used.

```
A1 = H*f(t,y(t))
A2 = H*f(t+H,y(t)+A1)
A3 = f(t+DTNOW,y(t)+(A1+A2)/2)
y(t+DTNOW) = y(t)+(A1+4*A2+A3)/6
```

Each step involves three function evaluations.

2.2.1.6. The improved Heun method

The improved Heun integration method may be selected by setting the two Boolean variables ADAMS and TRAPEZ to false. At the same time the Boolean variable SIMPSON must be false.

The mathematics of this third-order fixed step-size method is shown below (see also Sections 2.2.1.3 and 2.2.1.4).

```
DSH = 0.5*DTNOW*f(t-DTNOW,y(t-DTNOW))
A1 = DTNOW*f(t,y(t))
A2 = DTNOW*f(t+DTNOW,y(t)+1.5*A1-DSH)
y(t+DTNOW) = y(t)+(A1+A2)/2
```

Each step involves two function evaluations.

2.2.2. Interpolation

Interpolation is used to determine efficiently the state of the model at times within an integration step. The method is used both in connection with the time determination of state-events (WAITUNTIL-events) and with the regular reporting of REPORTER-objects with a positive frequency.

The interpolation is performed with polynomials in FRAC, where FRAC is the fraction DT/DTFULL, that is, the actual time increment DT (=TIME-LASTTIME) divided by the length of the full integration step, DTFULL.

When the RKE-method is used, a fifth-order polynomial

$$P(FRAC) = c_5*FRAC^5 + c_4*FRAC^4 + c_3*FRAC^3 + c_2*FRAC^2 + c_1*FRAC + c_0$$

is established for each active VARIABLE-object on the basis of the object's STATE and RATE at the three time points

```
LASTTIME , LASTTIME+1/2*DTFULL , LASTTIME+DTFULL
```

By means of such an interpolation polynomial for each active VARIABLE-object, it is possible to determine the model's state for TIME between LASTTIME and LASTTIME+DTFULL.

A VARIABLE-object's STATE at TIME=LASTTIME+FRAC*DTFULL (0<=FRAC<=1) is computed by Horner's scheme:

$$STATE:=(((c_5*FRAC+c_4)*FRAC+c_3)*FRAC+c_2)*FRAC+c_1)*FRAC+c_0$$

The six coefficients c_0 through c_5 are determined using the following results available from the RKE-integration:

OLDSTATE is STATE at LASTTIME
OLDSTATE+DSH is STATE at LASTTIME+1/2*DTFULL
OLDSTATE+DS is STATE at LASTTIME+DTFULL
A1 is the value of H*RATE at LASTTIME
A5 is the value of H*RATE at LASTTIME+1/2*DTFULL
RATE is the value of RATE at LASTTIME+DTFULL
(further, A4 is the estimated integration error over the full step)

The coefficients are determined as follows:

c_5 = 8*(-3*DS+A1+4*A5+H*RATE)
c_4 = 4*(4*DSH+13*DS-6*A1-20*A5-4*H*RATE)
c_3 = 2*(-16*DSH-17*DS+13*A1+32*A5+5*H*RATE)
c_2 = 16*DSH+7*DS-12*A1-16*A5-2*H*RATE
c_1 = 2*A1
c_0 = OLDSTATE

The VARIABLE-attributes A1, A2, A3, A4, A5 are used to contain the coefficients c_1, c_2, c_3, c_4, c_5, respectively.

When a fixed step-size integration method, instead of RKE, is used, the interpolation polynomial is of third order:

$$P(FRAC) = c_3*FRAC^3 + c_2*FRAC^2 + c_1*FRAC + c_0$$

It is established on the basis of the active VARIABLE-objects' STATE and RATE at the two time points

```
LASTTIME  ,  LASTTIME+DTFULL
```

The four coefficients are determined as follows:

c_3 = A1+DTFULL*RATE-2*DS
c_2 = DS-A1-A3
c_1 = A1
c_0 = OLDSTATE

This interpolation method, sometimes called Hermite interpolation, is examined and described in more detail in reference 3. It may be shown that the interpolation error is at least of the same order as the local integration error.

2.2.3. Quasi double-precision summation

To reduce rounding errors in the step-wise increase of the model's state, a method called "quasi-double precision summation" is used (ref. 4). The method is used both in updating the system time, TIME, and in updating the STATE-values of the active VARIABLE-objects. The method is especially useful when the simulation is carried out in many, relatively small, time-steps.

The summation procedure is illustrated in the algorithm outline below which shows how TIME, in principle, is advanced between discrete events. The variable EPSTIME denotes a small correction to the value of LASTTIME. In relation to LASTTIME, EPSTIME is so small that EPSTIME added to LASTTIME gives LASTTIME as a result.

```
while TIME<NEXTEVENTTIME do
begin
  LASTTIME:=TIME;
  DTNOW:= ... ;

  take a step of size DTNOW: TIME:=LASTTIME+(EPSTIME+DTNOW);

  if EPSTIME+DTNOW<=EPPSTIME and TIME<NEXTEVENTTIME then
  ERROR("7: THE CURRENT TIME STEP IS TOO SMALL TO ADVANCE TIME");

  EPSTIME:=(EPSTIME+DTNOW)-(TIME-LASTTIME);
end;
```

2.3. State-events

The planned state-events are represented in an ordered list of so-called "wait-notices". When the procedure WAITUNTIL is called, a wait-notice (an object of class WAITNOTICE) is inserted into the list and the active discrete process, CURRENT, becomes passive.

The user may determine the order of notices of the list by setting the global variables WAITPRIORITY and/or WAITPRIOR. The order of the notices is of significance only when there is a possibility of "simultaneous" state-events, that is, state-events occurring at the same instant of time. The monitor will examine the state-conditions in a sequence corresponding to the list order. In the case where two or more state-events may take place at the same time, the monitor will cause the state-event met first take place. Thereafter the list is examined from the beginning.

The class WAITNOTICE along with an outline of procedure WAITUNTIL is shown below.

```
class WAITNOTICE(PROC,PRIORITY);
ref(PROCESS) PROC; real PRORITY;
begin ref(WAITNOTICE) PREDWAIT,SUCWAIT; end;


 1. procedure WAITUNTIL(B); name B; Boolean B;
 2. inspect new WAITNOTICE(CURRENT,WAPRIORITY) do
 3. begin
 4.   rank this WAITNOTICE in the list of wait-notices;
 5.   PASSIVATE;

 6.   while PROC=/=CURRENT do
 7.   if B then
 8.   begin THEMONITOR.STATEEVENT:-PROC; RESUME(THEMONITOR); end
 9.   else
10.   RESUME(if SUCWAIT=/=none then SUCWAIT.PROC else THEMONITOR);
11.   remove this WAITNOTICE from the list of wait-notices;
12. end;
```

Each wait-notice has a reference to its waiting process, PROC, and a priority value, PRIORITY.

When procedure WAITUNTIL is called, a WAITNOTICE-object is created (line 2) with PROC referencing the calling discrete process, CURRENT, and with PRIORITY equal to the current value of the global real-variable WAITPRIORITY.

Next, the notice is inserted into the list of wait-notices (line 4). Its place in the list is determined from its priority, namely before all notices having a lower priority value and after all notices having a greater priority value. Thus the wait-notices are ranked in a high-value-first order (HVF). The value of the global Boolean variable WAITPRIOR determines if the notice is to be inserted before (WAITPRIOR is true) or after (WAITPRIOR is false) notices having the same priority value (cf. activate P prior).

Thereupon, the calling discrete process becomes passive (line 5).

To examine if a state-condition has been fulfilled, the monitor sets the PROCESS-reference STATEEVENT to none and RESUMEs the first waiting process (FIRSTWAIT.PROC). Every time a waiting process is RESUMEd, it will examine its own state-condition B (line 7). If this condition is true, then STATEEVENT is set to reference the process in question, PROC, and the monitor is RESUMEd (line 8). If, on the other hand, the condition is false, then the process will RESUME the next waiting process (SUCWAIT.PROC), if any; the last process RESUMEs the monitor (line 10).

When the monitor is RESUMEd, the condition

```
STATEEVENT=/=none
```

is true only if a waiting process, namely STATEEVENT, has its state-condition fulfilled.

The monitor's algorithm for state-event detection is sketched below.

```
STATEEVENT:-none;
if FIRSTWAIT=/=none then RESUME(FIRSTWAIT.PROC);
if STATEEVENT=/=none
then locate earliest state-event within current step;
```

If a waiting process is made active, either by the monitor because the state-event of the process must take place, or by some other discrete process (using `activate` or `reactivate`), then the wait-notice of the waiting process is removed from the list of wait-notices (line 11), and the process resumes its actions after the `WAITUNTIL` call.

The event times of state-events are determined with an accuracy of `DTMIN` using interpolation (see Section 2.2.2) and a binary search method (see Section 2.2.2). The monitor assure that no state-event takes place unless its corresponding condition is true. Moreover, no state-event is allowed to take place before all "simultaneous" time-events have occurred.

Generally, the following rules hold for "simultaneous" events:

(1) Time-events take place prior to state-events
(2) Time-events take place in their scheduled order
(3) State-events take place according to their priorities (HVF)

These rules may sometimes be of importance to the user. Note that they do not restrict the model formulation in any way since every time-event may be converted to a state-event.

2.4. Simulation control

A simulation is controlled by two `PROCESS`-objects, called `CONTROLLER1` and `CONTROLLER2`, which during the whole simulation period are represented successively in the event list of class `SIMULATION` (`SQS`), that is,

```
CONTROLLER1.NEXTEV==CONTROLLER2.
```

It is the responsibility of `CONTROLLER1` (an object of class `CONTROL1`) that the monitor becomes active after each discrete event.

`CONTROLLER2` (an object of class `CONTROL2`) is used to assure that the user does not attempt to call any event scheduling procedure (e.g., `HOLD` or `PASSIVATE`) between discrete events. All discrete changes are namely restricted to the discrete processes and must take place at event times. This control ensures that the `RESUME`-chains of (1) `CONTINUOUS`-objects, (2) `REPORTER`-objects and (3) `PROCESS`-objects calling `WAITUNTIL`, can not be destroyed by the user.

The two classes CONTROL1 and CONTROL2 are shown below.

```
PROCESS class CONTROL1;
while not THEMONITOR.ACTIVE do RESUME(THEMONITOR);

PROCESS class CONTROL2;
if THEMONITOR.CONTROLLER1.IDLE then
  begin
    if THEMONITOR.CONTROLLER1.TERMINATED
    then ERROR("17:  ILLEGAL CALL OF (RE)ACTIVATE");
    ERROR("18:  ILLEGAL CALL OF PASSIVATE (OR CANCEL(CURRENT))");
  end
else ERROR("19:  ILLEGAL CALL OF HOLD (OR REACTIVATE CURRENT)");
```

Each time CONTROLLER1 becomes active (CURRENT), it will RESUME the monitor (which is not a PROCESS-object). If the monitor already has control, that is, if its attribute ACTIVE is true, then the user must have made an error. In that case, CONTROLLER1 will terminate causing CONTROLLER2 to become active at once. The simulation is then stopped with an error message.

CONTROLLER2 becomes active if one of the following actions takes place during execution of a CONTINUOUS-object or a REPORTER-object, or as a side-effect of evaluating a state-condition (the name parameter of procedure WAITUNTIL).

1) HOLD or `reactivate CURRENT`,
2) PASSIVATE or CANCEL(CURRENT), or
3) `activate P` or `reactivate P`, where P is a PROCESS-object (direct activation).

CONTROLLER2 is able to distinguish between these three error cases merely by examining the current state of CONTROLLER1:

re 1) CONTROLLER1 will be suspended which causes CONTROLLER2 to become active. Then the following assertion is true:

```
not CONTROLLER1.IDLE
```

re 2) CONTROLLER1 will be passivated which causes CONTROLLER2 to become active. Then the following assertion is true:

```
CONTROLLER1.IDLE and not CONTROLLER1.TERMINATED
```

re 3) An event of process P is inserted before CONTROLLER1. When CONTROLLER1 again becomes active, the monitor's attribute ACTIVE is `true` which causes CONTROLLER1 to terminate and thus CONTROLLER2 to become active. Then the following assertion is true:

```
CONTROLLER1.TERMINATED
```

The following algorithm outline shows how the monitor exploits CONTROLLER1 and CONTROLLER2. Note that the monitor itself assures that the next future event scheduled by the user is legal.

```
LINK class MONITOR;
begin
  real TIME,NEXTEVENTTIME;
  Boolean ACTIVE;
  ref(WAITNOTICE) FIRSTWAIT;
  ref(PROCESS) STATEEVENT,NEXTTIMEEVENT,CONTROLLER1,CONTROLLER2;

  CONTROLLER1:-new CONTROL1;
  CONTROLLER2:-new CONTROL2;
  reactivate CONTROLLER2 after MAIN;
  reactivate CONTROLLER1 before CONTROLLER2;
  DETACH;

  NEXTTIMEEVENT:-CONTROLLER2.NEXTEV;

  while NEXTTIMEEVENT=/=none or FIRSTWAIT=/=none do
  begin
    ACTIVE:=true;
    comment *** Immediately AFTER an event;
    .
    .
    .
    while TIME<NEXTEVENTTIME do take a step;
    .
    .
    .
    comment *** Immediately BEFORE an event;
    if STATEEVENT=/=none then
    begin
      reactivate STATEEVENT at TIME;
      NEXTTIMEEVENT:-STATEEVENT; STATEEVENT:-none;
    end;
    reactivate CONTROLLER2 after NEXTTIMEEVENT;

    if CONTROLLER1.NEXTEV=/=NEXTTIMEEVENT then
      begin
        if NEXTTIMEVENT.IDLE
        then ERROR("20:  ILLEGAL CALL OF CANCEL");
        ERROR("17:  ILLEGAL CALL OF (RE)ACTIVATE");
      end
    else
    if not TIME=NEXTTIMEEVENT.EVTIME
    then ERROR("17:  ILLEGAL CALL OF (RE)ACTIVATE");

    ACTIVE:=false;
    reactivate CONTROLLER1 before CONTROLLER2;
    comment *** Now NEXTTIMEEVENT has taken place;
    NEXTTIMEEVENT:-CONTROLLER2.NEXTEV;
  end;

  ERROR("3:  THERE ARE NO DISCRETE EVENTS SCHEDULED");
end;
```

Moreover, between discrete events it is forbidden to call the CONTINUOUS- and REPORTER-procedures START, STOP, SETPRIORITY and SETFREQEUNCY, and the procedures WAITUNTIL, CANCELSTATEEVENT and PAUSE. Any attempt to make such a procedure call will be detected by the procedure in question in that the monitor's attribute ACTIVE is tested and found to be true.

3. THE ATTRIBUTES OF COMBINEDSIMULATION

The class skeleton below shows all the attributes of class COMBINEDSIMULATION. The user-attributes are printed in large type, whereas attributes which should be hidden from the user are printed in small type.

```
 SIMULATION class COMBINEDSIMULATION;
virtual: procedure SIMULATIONERROR,INTEGRATIONERROR;
begin
  LINK class CONTINUOUS; virtual: procedure PRELUDE;
  begin
    procedure PRELUDE;;
    procedure START; ... ;
    procedure STOP; ... ;
    Boolean procedure ACTIVE; ... ;
    ref(CONTINUOUS) procedure SETPRIORITY(R); real R; ... ;
    real procedure PRIORITY; ... ;
    real pri;
    ref(CONTINUOUS) predcont; ref(LINK) succont;
    ...;
  end;

  LINK class VARIABLE(STATE); real STATE;
  begin
    real RATE;
    procedure START; ... ;
    procedure STOP; ... ;
    Boolean procedure ACTIVE; ... ;
    real procedure LASTSTATE; ... ;
    real RELERROR,ABSERROR;
    real oldstate,epsstate,ds,dsh,a1,a2,a3,a4,a5;
    ref(VARIABLE) predvar,sucvar;
  end;

  LINK class REPORTER; virtual: procedure PRELUDE;
  begin
    procedure PRELUDE;;
    procedure START; ... ;
    procedure STOP; ... ;
    Boolean procedure ACTIVE; ... ;
    ref(REPORTER) procedure SETFREQUENCY(F); real F; ... ;
    real procedure FREQUENCY; ... ;
    real procedure REPORTTIME; ... ;
    real frq,reptime;
    ref(REPORTER) predrep; ref(LINK) sucrep;
    ...;
  end;
```

```
  procedure WAITUNTIL(B); name B; Boolean B; ... ;
  real WAITPRIORITY;
  Boolean WAITPRIOR;
  procedure CANCELSTATEEVENT(P); ref(PROCESS) P; ... ;

  real DTMIN,DTMAX;
  real procedure TIME; ... ;
  real procedure LASTTIME; ... ;
  real procedure DT; ... ;

  real MAXRELERROR,MAXABSERROR;
  Boolean EULER,ADAMS,TRAPEZ,SIMPSON;

  procedure PAUSE; ... ;
  ref(PROCESS) procedure NEXTTIMEEVENT(P); ref(PROCESS) P; ... ;
  procedure INTEGRATIONERROR; ... ;
  ref(VARIABLE) procedure ERRORVARIABLE; ... ;
  Boolean REPEATSTEP;
  procedure SIMULATIONERROR;
  ref(CONTINUOUS) procedure ERRORCONTINUOUS; ... ;
  ref(REPORTER) procedure ERRORREPORTER; ... ;

  real procedure maxreal; ... ;
  procedure abort; ... ;
  procedure error(message); value message; text message; ... ;

  class waitnotice(proc,priority);
  ref(PROCESS) proc; real priority;
  begin ref(waitnotice) predwait, sucwait; end;

  PROCESS class control1; ... ;

  PROCESS class control2; ... ;

  LINK class monitor;
  begin
    real time,lasttime,epstime,nexttime,nexteventtime,
         nextreporttime,dt,dtnow,dtnext,dtfull,dtlower,
         h,frac,errorratio,temp;
    Boolean active;
    ref(CONTINUOUS) firstcont,lastcont;
    ref(VARIABLE) firstvar,var;
    ref(REPORTER) firstposreporter,firstzeroreporter,
                  firstnegreporter;
    ref(waitnotice) firstwait,lastwait;
    ref(PROCESS) stateevent,nextstateevent,nexttimeevent,
                 controller1,controller2;
    ref(CONTINUOUS) errorcontinuous;
    ref(VARIABLE) errorvariable;
    ref(REPORTER) errorreporter;
    ...;
  end;

  ref(monitor) themonitor;

  themonitor:-new monitor;
  inner;
stopsimulation:;
end;
```

3.1 The SIMULATION-prefix

COMBINEDSIMULATION is a subclass of SIMULA's system-defined class for discrete event simulation, class SIMULATION.

All of class SIMULATION's properties are at the user's disposal. Thus class PROCESS may be used in the familiar manner for the description of discrete processes of a model.

3.2. Class CONTINUOUS

Class CONTINUOUS is used to describe the continuous processes of a model.

The continuous state changes are described in one or more subclasses of class CONTINUOUS, so that objects of these subclasses through their actions compute the current values or derivatives of state variables. It is the user's responsibility to make sure that the sequence of these computations is correct so that the quantities involved always reflect the current state of the model. Sometimes the sequence may be decisive for correct model behaviour. This applies for example when a VARIABLE-object's RATE appears on the right-hand side of an assignment statement. Execution of the active continuous processes is governed by a sequence of decreasing priorities (high-value-first). Processes with equal priorities are executed according to when they became active (earliest-first).

Only continuous changes may be described with class CONTINUOUS. All discrete state changes should be handled by discrete processes (PROCESS-objects).

A skeleton of class CONTINUOUS is shown below.

```
LINK class CONTINUOUS;
 virtual: procedure PRELUDE;
 begin
   procedure PRELUDE;;

   procedure START; ... ;

   procedure STOP; ... ;

   Boolean procedure ACTIVE; ACTIVE:=SUCCONT=/=none;

   ref(CONTINUOUS) procedure SETPRIORITY(R); real R;
   begin
     PRI:=R;
     if ACTIVE then begin START; STOP; end;
     SETPRIORITY:-this CONTINUOUS;
   end;

   real procedure PRIORITY; PRIORITY:=PRI;

   real PRI;

   ref(CONTINUOUS) PREDCONT; ref(LINK) SUCCONT;

   PRELUDE;
   DETACH;
 EXECUTE:;
   inner;
   RESUME(SUCCONT);
   goto EXECUTE;
 end;
```

The active continuous processes are placed in a list controlled by the monitor. When the monitor RESUMEs the first process of the list (FIRSTCONT), the user-defined actions (inner) of all active continuous processes will be executed. Each CONTINUOUS-object will namely, after having executed its own user-defined actions, RESUME its successor in the list (SUCCONT), and the last object of the list (LASTCONT) will RESUME the monitor (because LASTCONT.SUCCONT== THEMONITOR).

3.2.1. The LINK-prefix

The LINK-property is at the user's disposal. The CONTINUOUS-objects are placed in the list of active continuous processes independently of their LINK-property.

3.2.2. Procedure PRELUDE

The procedure PRELUDE, defined with an empty procedure-body in class CONTINUOUS, is called at generation of each CONTINUOUS-object.

The procedure is defined virtual and therefore can be redefined by the user in subclasses of class CONTINUOUS.

3.2.3. Procedure START

START inserts the object into the list of active continuous processes. Its place in the list is determined by the priority value of the object, PRIORITY (high-value-first). If there are other objects with the same priority value, then the object in question is inserted after the others.

Calling START when the object is already active has no effect.

Note that each object is inactive until its START-procedure is called.

START may only be called by a discrete process. Violating this rule leads to the error message

```
ILLEGAL CALL OF START (CLASS CONTINUOUS)
```

after which the simulation is stopped.

3.2.4. Procedure STOP

STOP removes the object from the list of active continuous processes.

Calling STOP has no effect unless the object is active.

STOP may only be called by a discrete process. Violating this rule leads to the error message

```
ILLEGAL CALL OF STOP (CLASS CONTINUOUS)
```

after which the simulation is stopped.

3.2.5. Boolean procedure ACTIVE

ACTIVE returns the value `true` if the object is in the list of active continuous processes; otherwise, `false`.

Since the monitor is the successor of the last object in the list (LASTCONT.SUCCONT== THEMONITOR), and an object which is not in the list has no successor (SUCCONT==none), ACTIVE is always equivalent to the condition

```
SUCCONT=/=none
```

3.2.6. Procedure SETPRIORITY(R); real R

The list of active continuous processes is ordered according to decreasing priority values (high-value-first). Calling `SETPRIORITY(R)` sets the object's priority to `R`.

The priority may be changed as often as necessary. Each continuous process has priority zero until its `SETPRIORITY`-procedure is called.

If the object is active at the time of the call, then it probably should be given a new place in the list. In this case, `SETPRIORITY` calls `STOP` followed by `START`.

`SETPRIORITY` may be called not only when the object is active, but also when it is inactive. However, `SETPRIORITY` may only be called by a discrete process. Violating this rule leads to the error message

```
ILLEGAL CALL OF SETPRIORITY (CLASS CONTINUOUS)
```

after which the simulation is stopped.

Actually, `SETPRIORITY` is a `ref(CONTINUOUS)`-procedure that returns a reference to this `CONTINUOUS`-object. This allows the user to write as follows:

```
new DERIVATIVES.SETPRIORITY(2).START
```

where `DERIVATIVES` is a subclass of class `CONTINUOUS`.

3.2.7. Real procedure PRIORITY

`PRIORITY` returns the current value of `PRI` (see below).

3.2.8. Real PRI

`PRI` denotes the current priority value of the `CONTINUOUS`-object.

The user has access to the value of `PRI` through the procedure `PRIORITY` and may assign a value to `PRI` by procedure `SETPRIORITY`.

The initial value of `PRI` is zero.

3.2.9. Ref(CONTINUOUS) PREDCONT; ref(LINK) SUCCONT

PREDCONT and SUCCONT denote the object's predecessor and successor in the list of active continuous processes.

The first object of the list has PREDCONT==none. The last object has SUCCONT== THEMONITOR (the monitor is a LINK-object). For objects not in the list PREDCONT== SUCCONT==none.

3.3. Class VARIABLE

Class VARIABLE is used to represent state variables that vary between discrete events according to ordinary first-order differential equations and/or difference equations. These equations are expressed in subclasses of class CONTINUOUS.

A skeleton of class VARIABLE is shown below.

```
LINK class VARIABLE(STATE); real STATE;
begin
  real RATE;

  procedure START; ... ;

  procedure STOP; ... ;

  Boolean procedure ACTIVE; ACTIVE:=PREDVAR=/=none;

  real procedure LASTSTATE; LASTSTATE:=OLDSTATE;

  real RELERROR,ABSERROR;

  real OLDSTATE,EPSSTATE,DS,DSH,A1,A2,A3,A4,A5;

  ref(VARIABLE) PREDVAR,SUCVAR;

  RELERROR:=MAXRELERROR; ABSERROR:=MAXABSERROR;
end;
```

The only actions of the class consist of assigning the attributes RELERROR and ABSERROR the value of MAXRELERROR and MAXABSERROR, respectively.

Active VARIABLE-objects are held in a list controlled by the monitor. Between discrete events the STATEs of these objects is updated by the monitor as defined by the active continuous processes.

The continuous variation may be expressed either as first-order "differential equation", for example,

```
V.RATE := 2+TIME*V.STATE
```

or as first-order "difference equation", for example,

```
V.STATE := V.LASTSTATE+DT*(2+LASTTIME*V.LASTSTATE)
```

where V is a VARIABLE-object.

3.3.1. The LINK-prefix

The LINK-property is at the user's disposal. The VARIABLE-objects are placed in the list of active VARIABLE-objects independently of their LINK-property.

3.3.2. Real STATE

STATE denotes the current value of the VARIABLE-object in question.

The initial value is passed as a parameter at object generation.

When the object is in the list of active VARIABLE-objects, the monitor will "continuously" change the object's STATE according to the value of its RATE.

3.3.3. Real RATE

RATE denotes the derivative of STATE with respect to TIME.

RATE is to be computed by the user by means of active CONTINUOUS-objects. When RATE is not computed in any CONTINUOUS-object, its value is equal to zero.

3.3.4. Procedure START

START inserts the object foremost in the list of active VARIABLE-objects.

Calling START when the object is already active has no effect.

Note that each VARIABLE-object is inactive until its START-procedure has been called. Note also that it is allowed to call START between discrete events, for example in connection with a call of the virtual procedure INTEGRATIONERROR (Section 3.17).

3.3.5. Procedure STOP

STOP removes the object from the list of active VARIABLE-objects and sets the object's RATE to zero.

Calling STOP has no effect unless the object is active.

Note that it is allowed to call STOP between discrete events, for example in connection with a call of the virtual procedure INTEGRATIONERROR (Section 3.17).

3.3.6. Boolean procedure ACTIVE

ACTIVE returns the value `true` if the object is in the list of active VARIABLE-objects; otherwise, `false`.

Since the predecessor of the first object in the list is defined as the object itself (FIRSTVAR.PREDVAR==FIRSTVAR), and an object not in the list has no predecessor (PREDVAR==none), ACTIVE is always equivalent to the condition

```
PREDVAR=/=none
```

3.3.7. Real procedure LASTSTATE

LASTSTATE returns the value of OLDSTATE, that is, the value of STATE at the starting point of the current step (see Section 3.3.9).

LASTSTATE may be used to describe continuous changes defined by difference equations.

3.3.8. Real RELERROR, ABSERROR

RELERROR and ABSERROR may be used by the user to set an upper bound for the relative and the absolute integration error allowed in each integration step.

When the RKE-method is used, the monitor at each step will assure that, for each active VARIABLE-object, the integration error is less than

```
ABS(ABSERROR) + ABS(RELERROR*m)
```

where m denotes the value of STATE in the middle of the current integration step (that is, at OLDTIME+1/2*DTFULL). If this condition can not be fulfilled, not even with the minimum step-size, DTMIN, and if the user has not redefined the virtual procedure INTEGRATIONERROR (Section 3.17), then the following error message is output

```
THE REQUESTED INTEGRATION ACCURACY CAN NOT BE ACHIEVED
```

after which the simulation is stopped.

When a fixed step-size integration method is used, the values of RELERROR and ABSERROR are irrelevant.

Note that, at the generation of a VARIABLE-object, its RELERROR and ABSERROR are automatically assigned the values of the two global variables MAXRELERROR and MAXABSERROR, respectively.

3.3.9. Real OLDSTATE

When the VARIABLE-object is active, OLDSTATE holds the value of STATE at the starting point of the current step, that is, at OLDTIME. When the object is inactive, the value of OLDSTATE will not be updated.

The user has access to the value of OLDSTATE through the procedure LASTSTATE (Section 3.3.7).

3.3.10. Real EPSSTATE

EPSSTATE is used in connection with the quasi-double precision summation of STATE-increments, DS, and denotes a small correction to OLDSTATE. The method is described in Section 2.2.3.

3.3.11. Real DS

DS is the increment of STATE for the current integration step (DTNOW). The following assertion holds

```
STATE = OLDSTATE+(EPSSTATE+DS)
```

During RKE-integration DS is also used for storing intermediate results.

3.3.12. Real DSH

During RKE-integration DSH is used to hold increment of STATE corresponding to half the current integration step (that is, 1/2*DTNOW).

If Adams' implicit integration method is applied, DSH is used to store the value of STATE at the starting point of the previous step (that is, at OLDTIME-DTNOW).

3.3.13. Real A1, A2, A3, A4, A5

The variables A1 through A5 are auxiliary variables used by the monitor for both integration (see Section 2.2.1) and interpolation (see Section 2.2.2).

3.3.14. Ref(VARIABLE) PREDVAR, SUCVAR

PREDVAR and SUCVAR denote the object's predecessor and successor, respectively, in the list of active VARIABLE-objects. When the object is not in this list, PREDVAR and SUCVAR both have the value none.

If the object is the first object of the list, then PREDVAR points to the object itself, that is, FIRSTVAR.PREDVAR==FIRSTVAR. The last object of the list has no successor, SUCVAR== none.

During RKE-integration the monitor may, for the sake of efficiency, change the list order. When the step-size is too large to achieve the user-requested accuracy for a VARIABLE-object, the object in question will be moved to the front of the list.

3.4. Class REPORTER

Class REPORTER may be used for reporting purposes. In one or more subclasses of class REPORTER the user may define actions for gathering information about the model's behaviour.

Each object of class REPORTER may automatically have its user-defined actions executed with a specified frequency, namely either (1) at uniformly spaced intervals, (2) after each time step, or (3) at event times.

Note that all state changes should be restricted to PROCESS- and CONTINUOUS-objects. Class REPORTER must not be used for any kind of state change.

A skeleton of class REPORTER is shown below.

```
LINK class REPORTER;
virtual: procedure PRELUDE;
begin
  procedure PRELUDE;;

  procedure START; ... ;

  procedure STOP; ... ;

  Boolean procedure ACTIVE; ACTIVE:=SUCREP=/=none;

  ref(REPORTER) procedure SETFREQUENCY(F); real F;
  begin
    if not ACTIVE or SIGN(FRQ)=SIGN(F) then
      begin REPTIME:=TIME; FRQ:=F; end
    else begin STOP; FRQ:=F; START; end;
    SETFREQUENCY:-this REPORTER;
  end;

  real procedure FREQUENCY; FREQUENCY:=FRQ;

  real procedure REPORTTIME; REPORTTIME:=REPTIME;

  real FRQ,REPTIME;

  ref(REPORTER) PREDREP,SUCREP;

  PRELUDE;
  DETACH;
EXECUTE:;
  inner;
  ...;
  RESUME(SUCREP);
  ...;
  goto EXECUTE;
end;
```

According to its frequency - positive, zero, or negative - an active REPORTER-object is in one of three lists controlled by the monitor. When the monitor RESUMEs the first object in one of these lists, the user-defined actions (inner) of all the list's REPORTER-objects are executed. Each REPORTER-object will namely, after having executed its own user-defined actions, RESUME its successor in the list, SUCREP, an the last object will RESUME the monitor (because its SUCREP ==THEMONITOR).

Each REPORTER-object assures that its user-defined actions are executed only as determined by the specified frequency, and that the frequency is not so small that time "stands still".

3.4.1. The LINK-prefix

The LINK-property is at the user's disposal. The REPORTER-objects are placed in one of the lists of active REPORTER-objects independently of their LINK-property.

3.4.2. Procedure PRELUDE

The procedure PRELUDE, defined in class CONTINUOUS with an empty procedure-body, is called at generation of each REPORTER-object.

The procedure is defined virtual and therefore can be redefined by the user in subclasses of class REPORTER.

3.4.3. Procedure START

START inserts the object in one of the three lists of active REPORTER-objects. The list the object is placed in is determined by the value of FREQUENCY - positive, zero, or negative. The first object of the three lists is denoted FIRSTPOSREPORTER, FIRSTZEROREPORTER and FIRSTNEGREPORTER, respectively.

Calling START when the object is already active has no effect.

Note that each object is inactive until its START-procedure is called.

START may only be called by a discrete process. Violating this rule leads to the error message

```
ILLEGAL CALL OF START (CLASS REPORTER)
```

after which the simulation is stopped.

3.4.4. Procedure STOP

STOP removes the object from the list of active REPORTER-objects of which the object is a member.

Calling STOP has no effect unless the object is active.

STOP may only be called by a discrete process. Violating this rule leads to the error message

```
ILLEGAL CALL OF STOP (CLASS REPORTER)
```

after which the simulation is stopped.

3.4.5. Boolean procedure ACTIVE

ACTIVE returns the value true if the object is a member of one of the three lists of active REPORTER-objects; otherwise, false.

Since the monitor is the successor of the last object in each of these lists (SUCREP== THEMONITOR), and an object which is not in the list has no successor (SUCREP==none), ACTIVE is always equivalent to the condition

```
SUCREP=/=none.
```

3.4.6. Procedure SETFREQUENCY(F); real F

SETFREQUENCY(F) sets the object's frequency to F (for the meaning of F, see Section 3.4.9).

The frequency may be changed as often as necessary. The frequency of each REPORTER-object is zero until its SETFREQUENCY-procedure is called.

If the object is active and has to be moved to another list (that is, if ACTIVE and SIGN(F)= SIGN(FREQUENCY)), then SETFREQUENCY calls STOP followed by START.

SETFREQUENCY may be called not only when the object is active, but also when it is inactive. However, SETFREQUENCY may only be called by a discrete process. Violating this rule leads to the error message

```
ILLEGAL CALL OF SETFREQUENCY (CLASS REPORTER)
```

after which the simulation is stopped.

Actually, SETFREQUENCY is a ref(REPORTER)-procedure that returns a reference to this REPORTER-object. This allows the user to write as follows:

```
new PLOTTER.SETFREQUENCY(0.1).START
```

where PLOTTER is a subclass of class REPORTER.

3.4.7. Real procedure FREQUENCY

FREQUENCY returns the value of FRQ, the frequency of the REPORTER-object (see Section 3.4.9).

3.4.8. Real procedure REPORTTIME

REPORTTIME returns the value of REPTIME (see Section 3.4.10).

3.4.9. Real FRQ

FRQ is the current frequency of the REPORTER-object.

The meaning of FRQ is explained by the following:

FRQ>0: Execution takes place at uniformly spaced intervals and also at event times

FRQ=0: Execution takes place at the end of each time step (which includes event times)

FRQ<0: Execution takes place only at event times

When a discrete event takes place, all active REPORTER-objects, regardless of frequency, have their user-defined actions executed twice at the event time, namely both immediately before and immediately after the event.

The initial value of FRQ is zero.

The user has access to the value of FRQ through the procedure FREQUENCY and may assign a value to FRQ with the procedure SETFREQUENCY.

3.4.10. Real REPTIME

For an active REPORTER-object with a positive frequency REPTIME denotes the time when the next regular execution of the object's user-defined actions will take place. Executions due to discrete event occurrences are not taken into account.

REPTIME may be viewed as a counterpart to the PROCESS-attribute EVTIME.

The user has access to the value of REPTIME through the procedure REPORTTIME.

If the object is inactive, the value of REPTIME is of no interest to the user and is not updated.

The initial value of REPTIME is zero. By calling START or SETFREQUENCY, REPTIME is set to TIME. The active REPORTER-objects themselves are responsible for updating REPTIME and for the determination of the earliest regular execution time, NEXTREPORTTIME. If during updating of REPTIME it is discovered that FREQUENCY is so small that the addition REPTIME + FREQUENCY gives REPTIME as a result, then the simulation is stopped with the following error message

```
FREQUENCY IS TOO SMALL TO ADVANCE TIME
```

3.4.11. Ref(REPORTER) PREDREP,SUCREP

PREDREP and SUCREP are the object's predecessor and successor, respectively, in the relevant list of active REPORTER-objects.

PREDREP and SUCREP both have the value none when the object is not a member in any of the three lists of active REPORTER-objects.

3.5. Procedure WAITUNTIL(B); name B; Boolean B

The procedure WAITUNTIL may be used to define state-events, that is, events whose time of occurrence is dependent upon a given state-condition.

WAITUNTIL(B), where B is a Boolean expression of arbitrary complexity, causes the active discrete process, CURRENT, to become passive (IDLE) over a period which is planned to last until B evaluates to true. However, this passive period will end sooner if the waiting process is activated by another discrete process.

It is possible to schedule a time-event for a waiting process, so that the process has simultaneously a state-event and a time-event scheduled (e.g., activate P delay 10, where P is a waiting PROCESS-object). When the first of these events takes place, the other one will be annulled.

A state-event takes place as soon as the corresponding state-condition is fulfilled. The event time will be determined with an accuracy of DTMIN (see Section 2.3).

Discrete processes operate in quasi-parallel, which means that "simultaneous" events occur in a certain order. With regard to simultaneous events, the following rules apply:

(1) Time-events take place before state-events

(2) Time-events take place in their scheduled order, that is to say, in the same sequence as they are represented in class SIMULATION's list of event notices (SQS)

(3) State-events take place in accordance with their priorities (WAITPRIORITY, high-value-first).

A state-event takes place only if its condition is true. Notice that the occurrence of a simultaneous event can change the condition's truth value.

WAITUNTIL must only be called by a discrete process. Violating this rule leads to the error message

```
ILLEGAL CALL OF WAITUNTIL
```

after which the simulation is stopped.

3.6. Real WAITPRIORITY

The global `real`-variable `WAITPRIORITY` may be used to give a state-event a priority.

When procedure `WAITUNTIL` is called, the state-event in question is assigned a priority equal to the current value of `WAITPRIORITY`.

The priority of a state-event has importance only if two or more state-events can take place simultaneously. In this case, the state-event having the highest priority will take place first.

`WAITPRIORITY` will only changed by the user. Initially its value is zero.

3.7. Boolean WAITPRIOR

The global Boolean-variable `WAITPRIOR` may be used to rank state-events having equal priority.

When procedure `WAITUNTIL` is called, the value of `WAITPRIOR` determines if the state-event in question is to be ranked higher (WAITPRIOR is `true`), or lower (`WAITPRIOR` is `false`) than all planned state-events having the same priority (cf. the construction `activate P at T prior`).

`WAITPRIOR` will only changed by the user. Initially its value is false. `WAITPRIOR` will only be changed

3.8. Procedure CANCELSTATEEVENT(P); ref(PROCESS) P

`CANCELSTATEEVENT` may be used to cancel a planned state-event.

Calling `CANCELSTATEEVENT(P)` causes the planned state-event of the discrete process `P`, if any, to be annulled (cf. the procedure `CANCEL` of class `SIMULATION`).

First, `CANCELSTATEEVENT` determines if the process P has any state-event associated with it, that is, if in the list of wait-notices there exists one notice having `PROC==P` (see Section 3.26). In this case, the notice in question is removed from the list. By letting the predecessor and the successor of the removed notice be the notice itself (`PREDWAIT==SUCWAIT==this WAITNOTICE`), it is assured that no problems arise when the `WAITUNTIL`-procedure attempts to remove the notice from the list later on.

Calling `CANCELSTATEEVENT(P)` is without effect if `P==none`, or `P` has not any planned state-events.

3.9. Real DTMIN, DTMAX

During the simulation time advances in steps of variable size. The step-size, among other things, is governed by the event times and the user's requirements regarding integration accuracy. `DTMIN` and `DTMAX` are used to specify the minimum and the maximum allowable step-size, respectively.

In general, step-size will vary within these bounds. There are the following exceptions, however:

(1) A time step less than `DTMIN` can be taken in order to assure that a time-event is not passed. In addition, the length of the first step taken after an event is equal to zero.

(2) A time step greater than `DTMAX` can be taken whenever there are neither planned state-events nor active continuous processes. In this case, step-size is as large as possible without passing a time-event.

The first trial step of a simulation is of length `DTMAX`. In using a fixed step-size integration method the step-size remains constant at `DTMAX`, unless an event occurs within a step.

Eventual active `REPORTER`-objects have no influence on the integration step-size. The state of the model at the regular reporting times is determined by interpolation (see Section 2.2.2).

Before a step is completed, the conditions for planned state-events are examined at the end of the step. If a condition is met, a more precise time determination is made, so that the earliest state-event is found within the step. The step-length is reduced accordingly. Observe that a state-event can be passed unnoticed, if `DTMAX` is so large that the corresponding condition is met within a step, but not at its endpoint.

The time of a state-event's occurrence is determined (by binary search) with an accuracy of `DTMIN`. One may set `DTMIN` to zero. In this case, state-events will be time-determined with the best accuracy obtainable on the computer in question.

`DTMIN` and `DTMAX` are two global variables which initially both have the value zero.

Assignment of their values should be made such that `0<=DTMIN<=DTMAX`. If this is not the case, the simulation is stopped with one of the following error messages

```
DTMIN<0

DTMIN>DTMAX
```

These errors can only be discovered immediately after an event, or immediately after a call of the virtual procedure `INTEGRATIONERROR` (Section 3.17).

If the time step ever becomes so small that time "stands still", the error message

```
THE CURRENT TIME STEP IS TOO SMALL TO ADVANCE TIME
```

is output, after which the simulation is stopped.

3.10. Real procedure TIME

TIME returns the value of THEMONITOR.TIME, that is, the current model time.

3.11. Real procedure LASTTIME

LASTTIME returns the value of THEMONITOR.LASTTIME, that is, the starting point of the current time step.

The value of LASTTIME may be used, for example, to describe first-order difference equations (see Section 3.3).

3.12. Real procedure DT

DT returns the value of THEMONITOR.DT, that is, the current time increment, TIME-LASTTIME.

The value of DT may be used, for example, to describe first-order difference equations.

Note that DT is equal to zero immediately after the occurrence of an event, so that division by DT should be avoided.

3.13. Real MAXRELERROR, MAXABSERROR

When a VARIABLE-object is generated the upper bound for the relative and the absolute error is set to the current value of MAXRELERROR and MAXABSERROR, respectively.

MAXRELERROR and MAXABSERROR are two global variables which initially both have the value zero.

Their values have meaning only when VARIABLE-objects are generated (see Section 3.3.8).

3.14. Boolean EULER, TRAPEZ, ADAMS, SIMPSON

Unless specified otherwise by the user, Runge-Kutta-England's fourth-order variable step-size integration method, RKE, is used. However, by setting one or more of the Boolean variables EULER, TRAPEZ, ADAMS or SIMPSON to true, an alternative integration method may be selected (see Section 2.2.1).

In contrast to RKE, the other integration methods operate with constant time steps and without consideration of user-specified error bounds. The step-size with these methods is equal to DTMAX, unless, of course, this causes an event to be bypassed.

When the continuous parts of the model are defined exclusively by difference equations, EULER, for the sake of efficiency, should be set to `true`. This causes namely the user-defined actions of the active continuous processes to be executed only once at each time step (in contrast to 9 times with RKE).

EULER, TRAPEZ, ADAMS and SIMPSON are global variables which initially are false. Their values may be changed at any time during the simulation.

3.15. Procedure PAUSE

The actions of the active continuous processes will, as a rule, be executed immediately after each event. Through the use of procedure PAUSE a so-called "event internal" execution of the actions of all active continuous processes can be achieved. Such a call causes the user-defined actions of all active CONTINUOUS-objects together with the user-defined actions of all active REPORTER-objects to be executed instantaneously (DT=0). Afterwards, the discrete process which called PAUSE resumes its actions.

PAUSE may only be called by a discrete process. Violating this rule leads to the error message

```
ILLEGAL CALL OF PAUSE
```

after which the simulation is stopped.

The desired effect of PAUSE is achieved by causing the monitor to become active (CONTROLLER1 ==CURRENT).

3.16. ref(PROCESS) procedure NEXTTIMEEVENT(P); ref(PROCESS) P

The simulation is automatically controlled by two PROCESS-objects, CONTROLLER1 and CONTROLLER2, which during the whole simulation are always present in class SIMULATION's list of event notices, SQS (see Section 2.4).

In order to avoid unintentional referencing these two PROCESS-objects, the user should not use the PROCESS-attribute NEXTEV. In stead, it is strongly recommended that procedure NEXTTIMEEVENT be used.

NEXTTIMEEVENT "hides" CONTROLLER1 and CONTROLLER2 from the user, but in other respects it has the same effect as NEXTEV.

The call NEXTTIMEEVENT(P) with P==none returns the value none.

3.17. Procedure INTEGRATIONERROR

The procedure INTEGRATIONERROR is automatically called if the requested integration accuracy (RELERROR and ABSERROR) can not be achieved without taking a step smaller than DTMIN.

INTEGRATIONERROR stops the simulation with the following error message

```
THE REQUESTED INTEGRATION ACCURACY CAN NOT BE ACHIEVED
```

However, the procedure is defined virtual in class COMBINEDSIMULATION and therefore can be redefined by the user. Thus the user may determine the course of action to be taken in the cases where the accuracy requirements can not be met. The user may, for example, (1) choose to decrease DTMIN, (2) slacken the accuracy requirements, (3) change from RKE-integration to one of the fixed step-size integration methods, (4) output a warning, or (5) totally ignore the error (by defining INTEGRATIONERROR with an empty procedure-body).

In connection with a redefinition of INTEGRATIONERROR the procedure ERRORVARIABLE (Section 3.18) and the Boolean procedure REPEATSTEP (Section 3.19) might be useful.

3.18. Ref(VARIABLE) ERRORVARIABLE

In the virtual procedure INTEGRATIONERROR (Section 3.17), the user is able to determine which VARIABLE-object gave rise to the call. ERRORVARIABLE returns a reference to the VARIABLE-object in question.

When INTEGRATIONERROR is not called, ERRORVARIABLE returns none.

ERRORVARIABLE returns the value of THEMONITOR.ERRORVARIABLE (Section 3.18.28).

3.19. Boolean REPEATSTEP

In connection with a redefinition of the virtual procedure INTEGRATIONERROR (Section 3.17), the user is able to specify that the actual integration step must be repeated after a call of the procedure. If REPEATSTEP is true, the step is repeated from its beginning; otherwise, the step is completed even though INTEGRATIONERROR has been called.

REPEATSTEP is a global Boolean variable which initially has the value false. Its value may be changed at any time during the simulation.

3.20. Procedure SIMULATIONERROR

If the simulation is to be stopped due to the occurrence of an error, the procedure SIMULATIONERROR is called just prior to termination. A complete list of fatal errors can be found in Appendix A.

SIMULATIONERROR prints the values DT, DTMIN and DTMAX.

However, the procedure is defined virtual in class COMBINEDSIMULATION and therefore may be redefined by the user. The user may, for example, use the procedure for printing information about the model's state at the time of the error, or close possibly open files. In connection with a redefinition of procedure SIMULATIONERROR the procedures ERRORVARIABLE (Section 3.18), ERRORCONTINUOUS (Section 3.21) and ERRORREPORTER (Section 3.22) might be useful.

3.21. Ref(CONTINUOUS) procedure ERRORCONTINUOUS

In the virtual procedure SIMULATIONERROR (Section 3.20) the user is able by means of procedure ERRORCONTINUOUS to determine if it was a CONTINUOUS-object that gave rise to the error. If this was the case (ERRORCONTINUOUS=/=none), then ERRORCONTINUOUS returns a reference to the CONTINUOUS-object in question.

ERRORCONTINUOUS returns the value of THEMONITOR.ERRORCONTINUOUS (Section 3.29.27).

3.21. Ref(REPORTER) procedure ERRORREPORTER

In the virtual procedure SIMULATIONERROR (Section 3.20) the user is able by means of procedure ERRORREPORTER to determine if it was a REPORTER-object that gave rise to the error. If this was the case (ERRORREPORTER=/=none), then ERRORREPORTER returns a reference to the REPORTER-object in question.

ERRORREPORTER returns the value of THEMONITOR.ERRORREPORTER (Section 3.29.28).

3.22. Real procedure MAXREAL

MAXREAL returns the largest real-value that can be represented in the computer.

MAXREAL is the only machine dependent part of class COMBINEDSIMULATION. When the class is installed on a computer, the procedure MAXREAL probably must be rewritten.

The monitor uses the value of MAXREAL when there are no time-events scheduled. In that case, NEXTEVENTTIME=MAXREAL.

3.24. Procedure ABORT

The procedure ABORT is called if a fatal error is detected during the simulation. This causes the simulation to be stopped at once.

Unfortunately, SIMULA has no built-in facility for aborting a program. A solution would seem to be a jump to a label, say STOPSIMULATION, placed last in the main program. However, such a jump is not legal when the main program is inactive and would cause a runtime-error.

Nevertheless, for lack of a better method, this solution is used anyway and has been programmed with

```
inner;
STOPSIMULATION:;
```

as the last statements of class COMBINEDSIMULATION.

3.25. Procedure ERROR(MESSAGE); value MESSAGE; text MESSAGE

Procedure ERROR is called when an error is detected which is so serious that the simulation must be stopped. A complete list of such errors can be found in Appendix A.

First, an error message is printed on the form

```
***COMBINEDSIMULATION
 ***ERROR m
 ***ENCOUNTERED AT TIME t
```

where m is the text parameter MESSAGE, and t is the current value of TIME.

Next, the virtual procedure SIMULATIONERROR (Section 3.20) is called, and finally, the simulation is stopped by calling procedure ABORT (Section 3.24).

3.26. Class WAITNOTICE

Planned state-events (WAITUNTIL-events) are represented as objects of class WAITNOTICE (see Section 2.3).

The class in full is as follows

```
class WAITNOTICE(PROC,PRIORITY);
ref(PROCESS) PROC; real PRIORITY;
begin ref(WAITNOTICE) PREDWAIT,SUCWAIT; end;
```

3.26.1. Ref(PROCESS) PROC

PROC is a parameter of class WAITNOTICE and references the waiting discrete process.

When WAITUNTIL is called, a WAITNOTICE-object is created having PROC==CURRENT.

3.26.2. Real PRIORITY

PRIORITY is a parameter of class WAITNOTICE and contains the priority of the state-event.

When WAITUNTIL is called, a WAITNOTICE-object is created having PRIORITY equal to the current value of the global real-variable WAITPRIORITY.

3.26.3. Ref(WAITPRIORITY) PREDWAIT,SUCWAIT

PREDWAIT and SUCWAIT denote WAITNOTICE-object's predecessor and successor in the list of wait-notices.

PREDWAIT and SUCWAIT both have the value none when the WAITNOTICE-object is not in the list of wait-notices. However, when the procedure CANCELSTATEEVENT (Section 3.8) has been used to remove the notice from the list, but the corresponding discrete process, PROC, is still waiting, then PREDWAIT==SUCWAIT==this WAITNOTICE.

3.27. PROCESS class CONTROL1

It is the responsibility of CONTROLLER1, an object of the class CONTROL1, that the monitor becomes active after each discrete event (see Section 2.4).

3.28. PROCESS class CONTROL2

CONTROLLER2, an object of the class CONTROL2, assures that between discrete event the user does not attempt to use the procedures HOLD, activate, reactivate, CANCEL or PASSIVATE (see Section 2.4).

3.29. Class MONITOR

The simulation is controlled behind the scenes, so to speak, by an object of class MONITOR, called the monitor (THEMONITOR).

The monitor is active between discrete events and accomplishes the following tasks:

(1) Time advance
The model time, TIME, is advanced in steps (see Section 2.1).

(2) Updating of state variables
Between discrete events the values of state variables are updated using numerical integration (see Section 2.2).

(3) Event control
The discrete events are triggered at the right time and in the correct sequence (see Section 2.3). Together with two PROCESS-objects, CONTROLLER1 and CONTROLLER2, the monitor assures that no event is planned or cancelled while the monitor is active (see Section 2.4).

(4) Reporting
The active REPORTER-objects have their user-defined actions executed with the specified frequency (see Section 2.1).

The following class skeleton shows all attributes of class MONITOR. An algorithm outline is given in Appendix C.

```
LINK class MONITOR;
begin
  real TIME,LASTTIME,EPSTIME,NEXTTIME,NEXTEVENTTIME,
       NEXTREPORTTIME,DT,DTNOW,DTNEXT,DTFULL,DTLOWER,
       H,FRAC,ERRORRATIO,TEMP;

  Boolean ACTIVE;

  ref(CONTINUOUS) FIRSTCONT,LASTCONT;

  ref(VARIABLE) FIRSTVAR,VAR;

  ref(REPORTER) FIRSTPOSREPORTER,FIRSTZEROREPORTER,
                FIRSTNEGREPORTER;

  ref(WAITNOTICE) FIRSTWAIT,LASTWAIT;

  ref(PROCESS) STATEEVENT,NEXTSTATEEVENT,NEXTTIMEEVENT,
               CONTROLLER1,CONTROLLER2;

  ref(CONTINUOUS) ERRORCONTINUOUS;
  ref(VARIABLE) ERRORVARIABLE;
  ref(REPORTER) ERRORREPORTER;

  ...;
end;
```

3.29.1. The LINK-prefix

Class MONITOR has been provided with the LINK-property so as to give the class a common prefix with class CONTINUOUS and class REPORTER. By this means, the monitor object, THEMONITOR, can be a successor of the last object in the list of active CONTINUOUS-objects, and also the successor of the last object in each of the three lists of active REPORTER-objects.

3.29.2. Real TIME

TIME denotes the current model time.

The user has access to the value of TIME through the global procedure TIME (Section 3.10).

The monitor assures that time does not "stand still". If the step-size, DTNOW, ever becomes so small that

```
EPSTIME+DTNOW = EPSTIME
```

and there is no event at that time, then the simulation will be stopped with the error message

```
THE CURRENT TIME STEP IS TOO SMALL TO ADVANCE TIME
```

3.29.3. Real LASTTIME

LASTTIME denotes the starting point of the current step.

The user has access to the value of LASTTIME through the global procedure LASTTIME (Section 3.11).

3.29.4. Real EPSTIME

TIME is advanced using quasi-double precision summation (see Section 2.2.3).

EPSTIME is a correction to LASTTIME. In relation to LASTTTIME it is numerically so small that the addition of EPSTIME to LASTTIME will produce LASTTIME as a result.

3.29.5. Real NEXTTIME

NEXTTIME denotes the ending point of the current step, DTNOW, that is,

```
NEXTTIME = LASTTIME+(EPSTIME+DTNOW)
```

It is always true that

```
TIME <= NEXTTIME <= NEXTEVENTTIME
```

3.29.6. Real NEXTEVENTTIME

NEXTEVENTTIME denotes the time point of the next known event. As long as a state-event has not been discovered by the monitor, NEXTEEVENTTIME is the time of the next time-event (MAXREAL, if no time-events are scheduled). When a state-event is time-determined, NEXTEVENTTIME is set to the event time of the state-event in question.

3.29.7. Real NEXTREPORTTIME

NEXTREPORTTIME denotes the time of the next regular reporting, namely the earliest REPORTTIME of the active REPORTER-objects having a positive frequency. However, NEXTREPORTTIME is never allowed to exceed NEXTEVENTTIME:

```
NEXTREPORTTIME <= NEXTEVENTTIME
```

The value of NEXTREPORTTIME is updated by the REPORTER-objects when executed. The monitor merely sets NEXTREPORTTIME to NEXTEVENTTIME before the execution of the REPORTER-objects.

The value of NEXTREPORTTIME has no influence upon the size of an integration step. Unless NEXTREPORTTIME is the end point of a step, the model's state at the reporting times will be determined using interpolation (see Section 2.2.2).

3.29.8. Real DT

DT denotes the current time increment.

The user has access to the value through the global procedure DT (Section 3.12).

The assertion that

```
TIME = LASTTIME+(EPSTIME+DT)
```

always holds.

DT is zero immediately after the occurrence of an event.

It is always true that

```
DT <= DTNOW
```

The value of DT is, for example less than DTNOW during RKE-integration where DT in each step takes the following values

```
1/4*DTNOW ,  1/2*DTNOW,  3/4*DTNOW  and  DTNOW
```

3.29.9. Real DTNOW

DTNOW denotes the size of the current step.

The assertion that

```
NEXTTIME = LASTTIME+(EPSTIME+DTNOW)
```

always holds.

Usually DTNOW is bounded by DTMIN and DTMAX:

```
DTMIN <= DTNOW <= DTMAX
```

However, no events may be passed within a step, that is, the condition

```
NEXTTIME <= NEXTEVENTTIME
```

must always be true.

When there are neither active continuous processes nor planned state-events, then

```
DTNOW = (NEXTEVENTTIME-LASTTIME)-EPSTIME
```

3.29.10. Real DTNEXT

DTNEXT is used during RKE-integration to denote a proposal for the size of the next integration step (see Section 2.2.1.2).

Its value will, if DTNOW=DTFULL=DTNEXT, be determined as follows:

```
DTNEXT:=MIN(MAX(1,MIN(2,(1/2*ERRORATIO)**(1/5)*DTNOW)),DTMAX)
```

During fixed step-size integration DTNEXT is equal to DTMAX.

Further, DTNEXT is equal to DTMAX at the beginning of the simulation.

The following condition is always true:

```
DTMIN <= DTNEXT <= DTMAX
```

3.29.11. Real DTFULL

DTFULL is used during interpolation to denote the size of the full integration step (see Section 2.2.2).

3.29.12. Real DTLOWER

DTLOWER is used in the binary search for a state-event (see Section 2.3).

During the process of locating the event time of a state-event, the following conditions hold:

(1) DTLOWER <= DTNOW.

(2) There is no state-event at the time

LASTTIME+(EPSTIME+DTLOWER), but

(3) there is at least one state-event (NEXTSTATEEVENT) at the time

LASTTIME+(EPSTIME+DTNOW).

The interval from DTLOWER to DTNOW is repeatedly halved until its length, DTNOW-DTLOWER, becomes smaller than DTMIN. When this happens a state-event, STATEEVENT, has been time-determined and occurs at the time

```
LASTTIME+(EPSTIME+DTNOW)
```

3.29.13. Real H

H is used during RKE-integration to hold the value 1/2*DTNOW.

3.29.14. Real FRAC

FRAC is used during interpolation to denote the fraction DT/DTFULL (see Section 2.2.2).

3.29.15. Real ERRORRATIO

ERRORRATIO is used during RKE-integration to hold the maximum value of the user-acceptable integration error divided by the estimated integration error (ABS(A4)) for all active VARIABLE-objects, that is:

```
(ABS(ABSERROR)+ABS(RELERROR*(OLDSTATE+(EPSSTATE+DSH)))/ABS(A4)
```

The value of ERRORRATIO is used to determine DTNEXT (see Section 2.2.1.2).

ERRORATIO is not allowed to become greater than what corresponds to a doubling of the current step-size, DTNOW. Accordingly,

```
ERRORATIO <= 2*(2**5)
```

3.29.16. Real TEMP

TEMP is an auxiliary variable which is used for temporary storage of a real-value.

3.29.17. Boolean ACTIVE

When the monitor is active, that is between discrete events, the value of ACTIVE is true; otherwise, ACTIVE is false. By means of ACTIVE it is possible to ensure that the user does not destroy the process synchronisation. For example, the illegal activation of a discrete process from a CONTINUOUS-object is detected and reported to the user (see Section 2.4).

3.29.18. Ref(CONTINUOUS) FIRSTCONT, LASTCONT

FIRSTCONT and LASTCONT denote the first and the last CONTINUOUS-object, respectively, in the list of active continuous processes.

Both have the value none when the list is empty.

3.29.19. Ref(VARIABLE) FIRSTVAR

FIRSTVAR denotes the first object in the list of active VARIABLE-objects. Its value is none when the list is empty.

3.29.20. Ref(VARIABLE) VAR

VAR is used to traverse the list of active VARIABLE-objects.

3.29.21. Ref(REPORTER) FIRSTPOSREPORTER, FIRSTZEROREPORTER, FIRSTNEGREPORTER

FIRSTPOSREPORTER, FIRSTZEROREPORTER and FIRSTNEGREPORTER denote the first object in (1) the list of active REPORTER-objects with a positive frequency, (2) the list of active REPORTER-objects with a frequency of zero, and (3) the list of active REPORTER-objects with a negative frequency, respectively. The value is none when the corresponding list is empty.

3.29.22. Ref(WAITNOTICE) FIRSTWAIT, LASTWAIT

FIRSTWAIT and LASTWAIT denote the first and the last WAITNOTICE-object in the list of wait-notices (see Section 2.3).

Both have the value none when the list is empty.

3.29.23. Ref(PROCESS) STATEEVENT

STATEEVENT is used by the monitor to determine if a state-condition has been fulfilled (see Section 2.3).

3.29.24. Ref(PROCESS) NEXTSTATEEVENT

NEXTSTATEEVENT is used during the time-determination of a state-event to store the value of STATEEVENT.

NEXTSTATEEVENT references the PROCESS-object which has planned next state-event.

3.29.25. Ref(PROCESS) NEXTTIMEEVENT

NEXTTIMEEVENT references the PROCESS-object which has scheduled the next time-event.

The following condition must always be true:

```
NEXTTIMEEVENT == CONTROLLER2.NEXTEV
```

This rule is used by the monitor to assure that the user does not destroy the process synchronisation (see Section 2.4).

3.29.27. Ref(CONTINUOUS) ERRORCONTINUOUS

When an error is discovered in the use of a CONTINUOUS-object, ERRORCONTINUOUS is set to reference the object in question. Thereafter, procedure ERROR is called. This is the case, for example, when CONTINUOUS-object is STARTed or STOPped between discrete events, that is to say, while the monitor is active.

The user has access to the value of ERRORCONTINUOUS through the global procedure ERRORCONTINUOUS (Section 3.21).

3.29.28. Ref(VARIABLE) ERRORVARIABLE

When the requested integration accuracy can not be achieved, ERRORVARIABLE is set to reference the VARIABLE-object that gave rise to the error. Thereafter, the virtual procedure INTEGRATIONERROR is called. If the user has redefined this procedure, ERRORVARIABLE will be set to none after a call.

The user has access to the value of ERRORVARIABLE through the global procedure ERRORVARIABLE (Section 3.18).

3.29.29. Ref(REPORTER) ERRORREPORTER

When an error is discovered in the use of a REPORTER-object, ERRORREPORTER is set to reference the object in question. Thereafter, procedure ERROR is called. This is the case, for example, when a REPORTER-object is STARTed or STOPped between discrete events, that is to say, while the monitor is active.

The user has access to the value of ERRORREPORTER through the global procedure ERRORREPORTER (Section 3.22).

3.30. Ref(MONITOR) THEMONITOR

THEMONITOR references the MONITOR-object. The object is generated by the main program, MAIN.

4. APPENDICES

4.1. Appendix A: Error messages

If an error is discovered during the simulation, the procedure ERROR is called. The causes an error message to be output after which the virtual procedure SIMULATIONERROR is called and the simulation is stopped.

The error message has the form:

```
***COMBINEDSIMULATION
***ERROR m
***ENCOUNTERED AT t
```

where m is a message describing the error and t is the current value of TIME.

The possible error messages with their associated numbers are given below.

1: THE REQUESTED INTEGRATION ACCURACY CAN NOT BE ACHIEVED

The virtual procedure INTEGRATIONERROR has not been redefined by the user and there is at least one VARIABLE-object, ERRORVARIABLE, which has an estimated integration error (A4) greater than the requested accuracy. That is to say,

```
ABS(A4) >
   ABS(ABSERROR)+ABS(RELERROR*(OLDSTATE+(EPSSTATE+DSH)))
```

2: THE CURRENT TIME STEP IS TOO SMALL TO ADVANCE TIME

EPSTIME+DTNOW=EPSTIME and TIME<NEXTEVENTTIME

3: THERE ARE NO DISCRETE EVENTS SCHEDULED

NEXTTIMEEVENT==none and FIRSTWAIT==none

4: DTMIN<0

5: DTMIN>DTMAX

The conditions that cause errors 4 and 5 are checked for only immediately after the occurrence of an event, and immediately after the execution of a user-defined version of the virtual procedure INTEGRATIONERROR.

6: FREQUENCY IS TOO SMALL TO ADVANCE TIME (CLASS REPORTER)

There is at least one REPORTER-object, ERRORREPORTER, with FREQUENCY>0 for which REPORTTIME+FREQUENCY=REPORTTIME.

7: TIME IS AT ITS MAXIMUM VALUE AND NO EVENTS OCCUR

TIME=MAXREAL and NEXTTIMEEVENT==none and STATEEVENT==none

8: ILLEGAL CALL OF PAUSE

9: ILLEGAL CALL OF CANCELSTATEEVENT

10: ILLEGAL CALL OF WAITUNTIL

11: ILLEGAL CALL OF SETPRIORITY (CLASS CONTINUOUS)

12: ILLEGAL CALL OF START (CLASS CONTINUOUS)

13: ILLEGAL CALL OF STOP (CLASS CONTINUOUS)

14: ILLEGAL CALL OF SETFREQUENCY (CLASS REPORTER)

15: ILLEGAL CALL OF START (CLASS REPORTER)

16: ILLEGAL CALL OF STOP (CLASS REPORTER)

17: ILLEGAL CALL OF (RE)ACTIVATE

18: ILLEGAL CALL OF PASSIVATE (OR CANCEL(CURRENT))

19: ILLEGAL CALL OF HOLD (OR REACTIVATE CURRENT)

20: ILLEGAL CALL OF CANCEL

Error messages 8 through 20 indicates that the procedure in question has been called between discrete events, that is to say, while the monitor is active.

Errors 17 and 20 can only be discovered if the event in question is within the "horizon" of the monitor, that is, if P.EVTIME<=NEXTEVENTTIME.

4.2. Appendix B: Efficiency and storage requirements

During the construction of class COMBINEDSIMULATION, generality and ease of use has been emphasized.

It is well-known that practical simulation problems often demand many time-consuming computer runs, and therefore it is of importance that the class is also efficient with respect to computer time.

Efficency has been enhanced through methods such as:

- Interpolation (Section 2.2.2)
- Binary search of state-events (Section 2.3)
- Step-size prediction (Section 2.2.1.2)
- RESUME-chain execution (Section 2)

The storage requirements are of lesser importance. Yet, this aspect has been taken into account. For example, the VARIABLE-attributes A1 through A5 are used repeatedly both during integration and interpolation.

The data storage requirements of COMBINEDSIMULATION are as follows:

The basic data storage requirements (THEMONITOR etc.)	136 words
Each VARIABLE-object	17 words
Each CONTINUOUS-object	9 words
Each REPORTER-object	10 words
Each idle PROCESS-object	6 words
Each suspended PROCESS-object	10 words
Each waiting PROCESS-object (WAITUNTIL)	16 words

4.3. Appedix C: Algorithm outline of the monitor

The actions of the monitor are sketched below.

```
while there are more planned events do
begin
  comment *** Immediately AFTER an event;
  DT:=0; LASTTIME:=TIME;

  if there are any active CONTINUOUS-objects then
  begin
    for each active VARIABLE-object do
    begin OLDSTATE:=STATE; RATE:=0; end;

    execute all active CONTINUOUS-objects;
    if DTNEXT=0 or a fixed step-size integration method is used
    then DTNEXT:=DTMAX;
  end;

  execute all active REPORTER-objects
  and determine NEXTREPORTTIME;
  NEXTEVENTTIME:=if no time-events are scheduled then MAXREAL
                 else EVTIME for the earliest time-event;
  if a state-condition is fulfilled and TIME<NEXTEVENTTIME
  then NEXTEVENTTIME:=TIME;

  while TIME<NEXTEVENTTIME do
  begin
    comment *** Between events;
    LASTTIME:=TIME;
    for each active VARIABLE-object do
    begin OLDSTATE:=STATE; RATE:=0; end;

    comment *** Determine step-size, DTNOW;
    DTNOW:=if there are any active CONTINUOUS-objects
           then MIN(NEXTEVENTTIME-LASTTIME,DTNEXT) else
           if there are any planned state-events
           then MIN(NEXTEVENTTIME-LASTTIME,DTMAX)
           else DTMAX;

    INTEGRATION:

    comment *** Take an integration step of size DTNOW;
    for each active VARIABLE-object do
    begin
      determine the STATE-increment DS using integration;
      STATE:=OLDSTATE+DS; RATE:=0;
    end;
```

```
    if the integration error is unacceptable then
    begin
      if DTNOW>DTMIN then
      begin DTNOW:=MAX(0.5*DTNOW,DTMIN); goto INTEGRATION; end;
      ERRORVARIABLE:-the VARIABLE-object with unacceptable error;
      INTEGRATIONERROR;
      ERRORVARIABLE:-none;
      if REPEATSTEP then
      begin
        for each active VARIABLE-object do
        re-establish STATE and RATE to their values at LASTTIME;
        goto INTEGRATION;
      end;
    end;

    DT:=DTNOW; TIME:=LASTTIME+DT;
    execute all active CONTINUOUS-objects and determine DTNEXT;

    comment *** Test if a state-event was passed;
    if a state-condition has been fulfilled then
    begin
      determine the time, TIME, for the earliest state-event
      within the step, and the state of the model at this point;
      NEXTEVENTTIME:=TIME;
    end;

    comment *** Test if a REPORTTIME was passed;
    if NEXTREPORTTIME<=TIME then
    begin
      while NEXTREPORTTIME<=TIME do
      begin
        determine the model's state at NEXTREPORTTIME
        using interpolation;
        execute all active REPORTER-objects
        having REPORTTIME=NEXTEREPORTTIME;
        re-establish TIME;
      end;
      re-establish the model's state at TIME;
    end;

    comment *** Now the step has been taken;
    execute all active REPORTER-objects having FREQUENCY=0;
  end;

  comment *** Immediately BEFORE an event;
  if DT>0 then execute all active REPORTER-objects
  having FREQUENCY<0;

  let an event take place now;
end;
```

5. REFERENCES

1. Dahl,O-J, Myhrhaug,B., Nygaard,K.:
   "Common Base Language".
   Publication no s-22, Norwegian Computing Center,
   Oslo 1970.

2. England,R.:
   "Error estimates for Runge-Kutta type solutions to systems of ordinary differential equations".
   Computer Journal, Vol. 12, 1969, pp. 166-170.

3. Helsgaun, K.:
   "On interpolation in class COMBINEDSIMULATION" (in Danish).
   Roskilde University Center, October 1978.

4. Møller,O.:
   "Quasi double-precision summation in floating point addition".
   BIT 5, 1965, pp. 37-50 and 251-255.

5. Shampine,L.F., Watts,H.A.:
   "Comparing Error Estimators for Runge-Kutta Methods".
   Mathematics of Computation, Vol. 25, 1971, pp. 445-455.

TABLE OF CONTENTS