Fundamentals of Synchronous Control in Modelica

Hilding Elmqvist[1] Martin Otter[2] Sven Erik Mattsson[1]
[1]Dassault Systèmes AB, Ideon Science Park, SE-223 70 Lund, Sweden
[2]DLR Institute of System Dynamics and Control, D-82234 Wessling, Germany
Hilding.Elmqvist@3ds.com Martin.Otter@dlr.de SvenErik.Mattsson@3ds.com

Abstract

The scope of Modelica 3.3 has been extended from a language primarily intended for physical systems modeling to modeling of complete systems by allowing the modeling of control systems and enabling automatic code generation for embedded systems.

This paper describes the fundamental synchronous language primitives introduced for increased correctness of control systems implementation. The approach is based on associating clocks to the variable types. Special operators are needed when accessing variables of another clock. This enables clock inference and increased correctness of the code since many more checks can be done during translation.

Keywords: Modelica; Synchronous; Control; Sampled Data Systems, Periodic Systems

1 Introduction

The scope of Modelica has been extended from a language primarily intended for physical systems modeling to modeling of complete systems by allowing the modeling of control systems and by enabling automatic code generation for embedded systems.

This paper describes the fundamental synchronous language primitives introduced for increased correctness of control systems implementation since many more checks can be done at compile time. A companion paper (*Elmqvist, et.al, 2012*) describes the state machine features of Modelica 3.3. Yet another companion paper (*Otter, et.al, 2012*) describes a Modelica library, Modelica_Synchronous, which supports a graphically oriented approach to synchronous control systems implementation.

The new language elements follow the synchronous approach (*Benveniste et. al. 2002*). They are based on the clock calculus and inference system proposed by (*Colaco and Pouzet 2003*) and implemented in Lucid Synchrone version 2 and 3 (*Pouzet 2006*). However, the Modelica approach also uses multi-rate periodic clocks based on rational arithmetic introduced by (*Forget et. al. 2008*), as an extension of the Lucid Synchrone semantics. Additionally, the built-in operators introduced in Modelica 3.3 also support non-periodic and event based clocks.

In the following sections the new language elements are discussed. Afterwards, in section 5, a rational is given why they have been introduced by comparing the new possibilities with the features of Modelica 3.2 to model sampled data systems.

2 Synchronous Features of Modelica

The synchronous features of Modelica 3.3 will be gradually introduced by means of examples illustrating how to use them. This paper uses a completely textual approach. The companion paper (*Otter, et.al, 2012*) describes a Modelica library, Modelica_Synchronous, which supports a graphically oriented approach to synchronous control systems implementation.

2.1 Plant and Controller Partitioning

We will consider control of a mass and spring-damper system with a force actuator. A Modelica model is shown below:

```
model MassWithSpringDamper
  parameter Modelica.SIunits.Mass m=1;
  parameter Modelica.SIunits.TranslationalSpringConstant k=1;
  parameter
        Modelica.SIunits.TranslationalDampingConstant d=0.1;
  Modelica.SIunits.Position x(start=1,fixed=true) "Position";
  Modelica.SIunits.Velocity v(start=0,fixed=true) "Velocity";
  Modelica.SIunits.Force f "Force";
equation
  der(x) = v;
  m*der(v) = f - k*x - d*v;
end MassWithSpringDamper;
```

A simple discrete-time speed controller can be implemented as follows:

```
model SpeedControl
  extends MassWithSpringDamper;
  parameter Real K = 20 "Gain of speed P controller";
  parameter Modelica.SIunits.Velocity vref = 100 "Speed ref.";
  discrete Real vd;
```

```
  discrete Real u(start=0);
equation
  // speed sensor
  vd = sample(v, Clock(0.01));

  // P controller for speed
  u = K*(vref-vd);

  // force actuator
  f = hold(u);
end SpeedControl;
```

The SpeedControl model extends the continuous-time plant model MassWithSpringDamper. The speed controller is a discrete-time controller. The boundaries between continuous-time equations and discrete-time equations are defined by the operators sample and hold.

The sample operator samples a continuous-time variable and returns a discrete-time variable. The sample rate is specified by the second Clock argument to sample. In this case, a periodic clock which ticks with a period of 0.01 second is specified.

Since sample returns a discrete-time result that is associated to clock Clock(0.01), the variable vd becomes discrete-time and is associated to the same clock as well. Variable vd appears in equation u = K*(vref-vd) and therefore all time varying variables in this equation, i.e., u, must be also discrete-time and associated to the same clock. If further equations would be present, then all equations in which vd and u appear, would be again associated to the same clock. This approach to identify the equations belonging to the same clock is called *clock inference* and is a key element in the new approach.

The hold operator converts from discrete-time to continuous-time by holding the value between the clock ticks. More precisely, the hold(u) operator returns the start value of u if the operator is called before the first tick of the clock of u. Otherwise, the most recently available value of u is returned.

To summarize, the sample(v..) and hold(..) operators define the boundaries between clocked and continuous-time partitions. Equations and variables belonging to the same clocked partition are identified by clock inference.

2.2 Discrete-time State Variables

More advanced features will be introduced using a position controller using an inner P controller and an outer PI controller. The first version is using one clock:

```
model ControlledMassBasic
  extends MassWithSpringDamper;
  parameter Real KOuter = 10 "Gain of position PI controller";
  parameter Real KInner = 20 "Gain of speed P controller";
  parameter Real Ti = 10 "Integral time for pos. PI controller";
  parameter Real xref = 10 "Position reference";

  discrete Real xd;
  discrete Real eOuter;
  discrete Real intE(start=0);
  discrete Real uOuter;

  discrete Real vd;
  discrete Real vref;
  discrete Real uInner(start=0);
equation
  // position sensor
  xd = sample(x, Clock(0.01));

  // outer PI controller for position
  eOuter = xref-xd;
  intE = previous(intE) + eOuter;
  uOuter = KOuter*(eOuter + intE/Ti);

  // speed sensor
  vd = sample(v);

  // inner P controller for speed
  vref = uOuter;
  uInner = KInner*(vref-vd);

  // force actuator
  f = hold(uInner);
end ControlledMassBasic;
```

In this model, the sample operator for v does not have an associated Clock specification since it is inferred (sample(v) is implicitly associated to clock Clock(0.01) because xd is on this clock, and therefore eOuter, and therefore uOuter, and therefore vref and therefore vd, and therefore sample(v)).

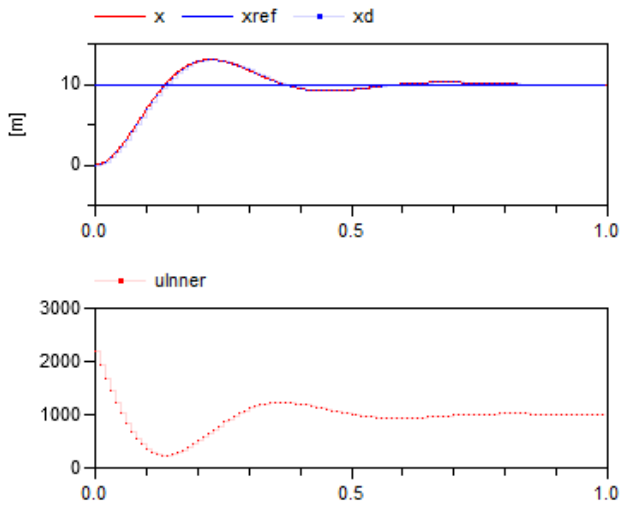
Since a PI controller is used, it is necessary to introduce a discrete-time state variable for the integral part. The operator previous(..) is used to access the value of intE at the previous clock tick. Note that due to this use of previous(..),intE becomes a discrete-time state and needs to have a start value specified in the declaration (at the first clock tick, previous(intE) returns the start value of intE).

The behavior of the system is shown in the figure below: x, xref and xd (upper diagram) and the actuator signal uInner (lower diagram).

2.3 Base- clocks and Sub-clocks

A Modelica model will typically have several controllers for different parts of the plant. Such controllers might not need synchronization and can have different *base clocks*. Equations belonging to different base clocks can be implemented by asynchronous tasks of the used operating system.

DOI 10.3384/ecp1207615



It is also possible to introduce sub-clocks that tick a certain factor slower than the base clock. Such sub-clocks are perfectly synchronized with the base clock, i.e. the definitions and uses of a variable are sorted in such a way that when sub-clocks are activated at the same clock tick, then the definition is evaluated before all the uses.

Such sub-clocks can, for example, be used to save CPU resources. In some cases, an outer controller of a cascade control architecture does not need to be evaluated as often as the inner controller.

When using several clocks, it is convenient and clear to declare them. Modelica 3.3 introduces a new base type, Clock, for this purpose:

```
Clock cControl = Clock(0.01);
Clock cOuter = subSample(cControl, 5);
```

The subSample operator creates a clock which is a factor slower; in this case cOuter becomes 5 times slower than cControl. The subSample operator can also operate on a discrete-time variable and then picks the value at every factor clock tick of the clock of this variable.

Such clock variables can then be used as argument to the sample operator:

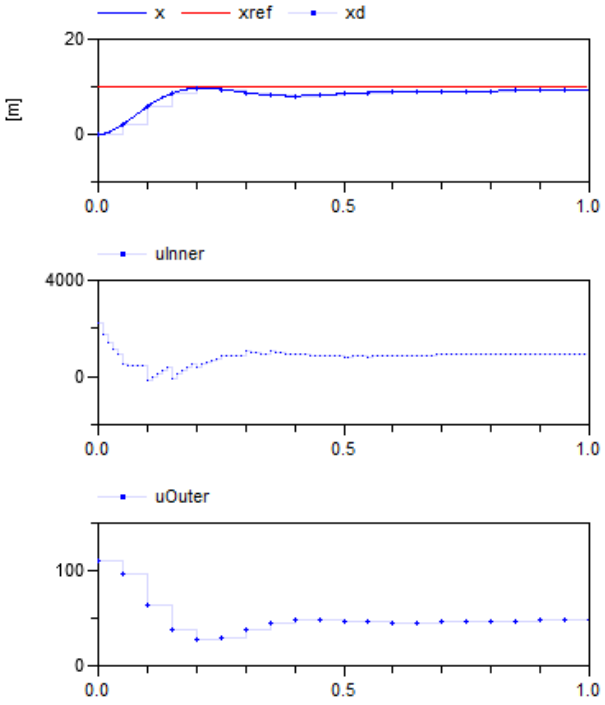
```
xd = sample(x, cOuter);
vd = sample(v, cControl);
```

The outer controller now calculates uOuter at the rate of cOuter. uOuter is the velocity reference, vref, for the inner controller which is compared to the sampled velocity measurement, vd. vd has clock cControl, i.e., 5 times faster than uOuter. Trying to directly calculate uOuter-vd would give a clocking error since the semantics is not clear. The user needs to state the intent by using a clock conversion operator. In this case uOuter needs to be converted to the faster clock by using the superSample operator:

```
vref = superSample(uOuter, 5);
```

superSample replicates a factor 5 times the value of the variable with the slower clock to have a clock a factor faster.

The simulation results are shown below. Note that xd and uOuter have a slower sample rate than uInner.



2.4 Interval of Clock

It is possible to inquire the actual interval of a clock by using the interval() operator. One example of the need is when using difference approximations. Assume that no speed sensor is available and the speed needs to be estimated from changes of position. A first order approximation is shown below. It uses a faster sampling of the position, x:

```
Clock cFast = superSample(cControl, 2);

xdFast = sample(x, cFast);
vd = subSample((xdFast-previous(xdFast))/interval(), 2);
```

After approximating the derivative at the higher rate, the result is sub-sampled with a factor of 2 to get the required rate of vd.

2.5 Phase of Clock

To better control the scheduling of calculations, it is possible to shift the phase of a clock. For example, the calculation of the outer controller code will be done before the inner controller code due to the data flow. This might give jittering in the actuator signal uInner caused by the slight delay due to the computation time. One way to avoid this is to schedule the

outer code to be executed later in the cycle and to accept the use of an old value of uOuter. This is accomplished in the following way:

```
Clock cOuter = subSample(shiftSample(cControl, 2, 3), 5);
```

The shiftSample operator shifts the clock a part of the interval. In this case 2/3 of the interval of the clock cControl.

By changing the clock cOuter in this way, the calculation of uOuter will be delayed and will not be synchronized to vd. This needs to be compensated by using backSample which shifts the clock in the opposite direction to shiftSample:

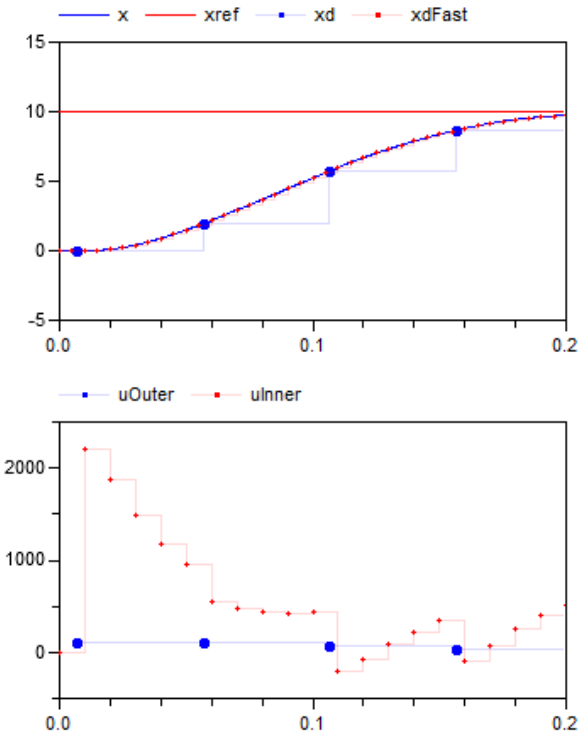
```
vref = backSample(superSample(uOuter, 5), 2, 3);
```

It should be noted that this means that a start value must be given to uOuter which is used before the clock of uOuter has started ticking.

The complete model including all aspects discussed above is given below:

```
model ControlledMass
  extends MassWithSpringDamper;
  parameter Real KOuter = 10 "Gain of position PI controller";
  parameter Real KInner = 20 "Gain of speed P controller";
  parameter Real Ti = 10 "Integral time for pos. PI controller";
  parameter Real xref = 10 "Position reference";

  discrete Real xd;
  discrete Real eOuter;
  discrete Real intE(start=0);
  discrete Real uOuter(start=0);

  discrete Real xdFast;
  discrete Real vd;
  discrete Real vref;
  discrete Real uInner(start=0);

  Clock cControl = Clock(0.01);
  Clock cOuter = subSample(shiftSample(cControl, 2, 3), 5);
  Clock cFast = superSample(cControl, 2);
equation
  // position sensor
  xd = sample(x, cOuter);

  // outer PI controller for position
  eOuter = xref-xd;
  intE = previous(intE) + eOuter;
  uOuter = KOuter*(eOuter + intE/Ti);

  // speed estimation
  xdFast = sample(x, cFast);
  vd = subSample((xdFast-previous(xdFast))/interval(), 2);

  // inner P controller for speed
  vref = backSample(superSample(uOuter, 5), 2, 3);
  uInner = KInner*(vref-vd);

  // force actuator
  f = hold(uInner);
end ControlledMass;
```

The simulation results are shown below. In particular it can be noted how uOuter is shifted 2/3 of the interval on uInner.



An interesting question is when a clock starts to tick. In principal there are two useful approaches: A clock starts ticking at time = 0 seconds or it starts ticking at the simulation start time (or when a device is switched on). The synchronous extensions of Modelica use the second approach because from the view of a hardware device, there is no absolute but only relative time.

Operator y = shiftSample(u, c, r) defines a new clock that basically shifts the first activation of the clock of y in time c/r*interval(u) later than the first activation of the clock of u. This definition gives not a precise time definition because interval(u) is of type Real. Furthermore, it only holds in special cases, such as for periodic clocks with a fixed period. The precise time definition that holds for all clocks is achieved by constructing (conceptually) a clock cBase:

```
Clock cBase = subSample(superSample(u, r), c);
```

and the clock of y = shiftSample(u, c, r) starts at the second clock tick of cBase and y is set to the most recently available value of u.

In a similar way the operator y = backSample(u, c, r) defines a new clock that basically shifts the first activation of the clock of y in time c/r*interval(u) before the first activation of the clock of u. Similarly to shiftSample, the precise time definition is achieved by constructing (conceptually) a clock cBase :

```
Clock cBase = subSample(superSample(u, r), c);
```


DOI 10.3384/ecp1207615

and the clock of y = backSample(u, c, r) is shifted a time duration before the clock of u, such that this duration is identical to the duration between the first and second clock tick of cBase.

The backSample(..) operator is more critical than the shiftSample(..) operator: The clock of v starts before the clock of u and therefore a start value for u is needed and before the first tick of the clock of u, the operator returns this start value. Additionally, there is the restriction that the clock of v cannot start before the simulation start time.

On first view, one could have only provided one operator to shift the start of a clock forward or backward in time. However, shifting backwards in time requires providing a start value, whereas this is not the case when shifting forward in time. Since these are therefore structurally different cases, it is better to use two different operators.

2.6 Exact Periodic Clocks

In the previous sections, periodic clocks are defined with the Clock(period) constructor, where period is of type Real and defines the sample period. The semantics is that two clocks of this kind are not time synchronized to each other. Example:

```
Clock c1 = Clock(0.1);
Clock c2 = superSample(c1,3);
Clock c3 = Clock(0.1/3);
```

Clock c1 and c2 are precisely time synchronized to each other and at every third tick of c2, clock c1 ticks. However, clock c3 is not time synchronized to c1 or c2 and there is no guarantee that c3 ticks at every third tick of c1. The reason is that calculations with Real numbers are not exact and subject to small numerical errors.

Alternatively, a periodic clock can be defined with the Clock(c,r) operator, where c and r are of type Integer, and the fixed sample period is defined as the rational number c/r. The semantics is that all clocks defined in this way are precisely time synchronized to each other. Example:

```
Clock c1 = Clock(1,10);          // period = 1/10
Clock c2 = superSample(c1,3);  // period = 1/30
Clock c3 = Clock(1,30);          // period = 1/30
```

Clocks c1, c2, and c3 are precisely time synchronized to each other and at every third tick of c2 and of c3, clock c1 ticks.

An interesting question is which periods can be defined with exact periodic clocks? Basically, a period is defined as the quotient of two Modelica Integer numbers, which are usually 32 bit integers. Therefore, periods in the range 10^{-9} ... 10^{9} s can be directly defined. However, clocks can be sub- and super-sampled, e.g,

```
superSample(Clock(1, 1000000000) , 1000000000);
```

The resulting clock will have a period of 10^{-18} s. In other words, from a Modelica point of view, any period that can be represented by a rational number with unlimited precision can be defined. In the Modelica 3.3 specification it is stated that "it is required that accumulated sub- and super sampling factors in the range of 1 to 2^{63} can be handled". Therefore, every tool should support internally at least 64 bit integers and therefore periods in the range 10^{-18} ... 10^{18} s.

2.7 Clocked When Clause

Although the new synchronous operators allow defining clocked equations *implicitly* due to clock inference, it is sometimes still useful to *explicitly* define that a group of equations is associated with the same clock. In order to not introduce yet another new keyword, the already existing when-clause is overloaded for this purpose. Example:

```
 import Modelica.Utilities.Streams.print;
equation
 when Clock(0.1) then
    x = A*previous(x) + B*u;
    y = C*previous(x) + D*u;
    print("Clock ticks at time = " +  String(sample(time)));
 end when
```

If a clock is used in a when-clause then all equations in the when-clause are associated with this clock. In such a case, the equations in the when-clause can be arbitrary equations (recall that for standard when-clauses with a Boolean condition, all equations in the when-clause must have a variable reference on the left hand side of every equation, i.e., equations must be of the form "x = expr").

In the example above, all three equations in the when-clause belong to the same partition that is are associated to clock Clock(0.1). When-clauses might be used to clearly define that equations are associated with the same clock. Furthermore, there are exceptional cases as in the example above, where it would be not possible to associate the print(..) statement to Clock(0.1) without a when-clause because no variable of the clocked partition is used in the print statement. If the clock of the when-clause is defined somewhere else and shall be deduced by clock inference, then the clock Clock() needs to be used in the when-clause:

```
when Clock() then       // clock is inferred
   x = A*previous(x) + B*u;
   y = C*previous(x) + D*u;
   print("Clock ticks at time = " +  String(sample(time)));
end when
```

In Modelica 3.3, clocked when-clauses are restricted: The condition must be a clock (and not, say a Boolean expression of clocks such as "c1 or c2"), an else-

when part is not allowed, and the clocked when clause can only appear in an equation section.

2.8 Varying Interval Clocks

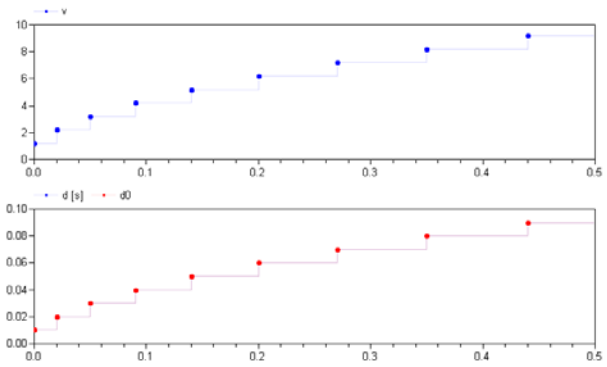
It is also possible to define clocks with a varying interval between the sampling points. As an example, consider

```
model VaryingClock
  Integer nextInterval(start=1);
  Clock c = Clock(nextInterval, 100);

  Real  v(start=0.2);
  Real  d = interval(v);
  Real  d0 = previous(nextInterval)/100.0;
equation
  when c then
    nextInterval = previous(nextInterval) + 1;
    v = previous(v) + 1;
  end when;
end VaryingClock;
```

It defines a Clock c with varying interval, nextInterval. A definition of the form

```
Clock c = Clock(nextInterval, 100)
```

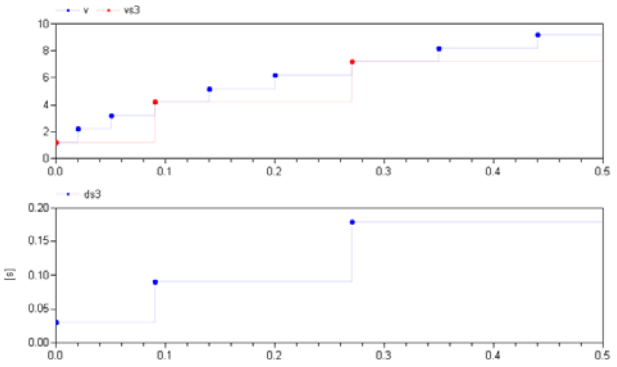
states that clock c ticks at the simulation start and then every nextInterval/100 seconds, and at every clock tick, nextInterval can be newly computed. Since at the first clock tick, previous(nextInterval) is equal to the start value of nextInterval (= 1), the value of nextInterval at the first clock tick is 1+1 = 2, and therefore the second clock tick is at 2/100 seconds. The further ticks are at 5/100, 9/100 etc. The behavior of the variable v is shown in the following plot:



The variables d = interval(v) and d0 = previous(nextInterval)/100.0 are equal.

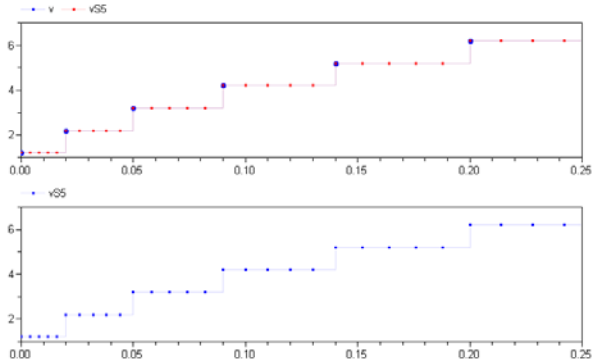
Let us sub-sample v by adding to the model:

```
Real vs3 = subSample(v, 3) ;
Real ds3 = interval(vs3);
```



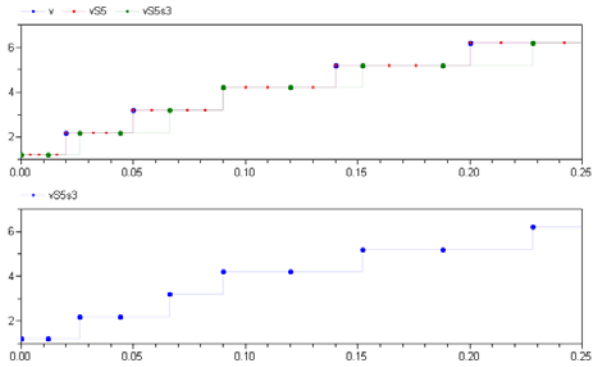
As the plot shows, vs3 samples each third point of v. We can also super-sample:

```
Real vS5 = superSample(v, 5) ;
Real dS5 = interval(vS5);
```



The 5 super-sampling points are evenly distributed in time within the intervals of clock c as shown by the plot of dS5. Let us now sub-sample dS5:

```
Real vS5s3 = subSample(vS5, 3) ;
Real ds3S5 = interval(vs3S5);
```



The result is that vS5s3 is every third sample of vS5 resulting in a more irregular sampling interval. The equation vS5s3 = subSample(vS5, 3) can be expanded as vS5s3 = subSample(superSample(v, 5), 3).

What is the result if we do it in the reverse order, vs3S5 = superSample(subSample(v, 3), 5)? For the clock c, the time to the next tick is known at the current tick.

DOI 10.3384/ecp1207615

However, this is not the case for the clock of subSample(v, 3). The interval to its next tick is the sum of 3 future intervals of c and only the first term is known. The definition of super-sampling does not require the intervals of super-sampling to be equidistant in time. The definition is instead based on counting ticks. It means that vs3S5 = vS5s3.

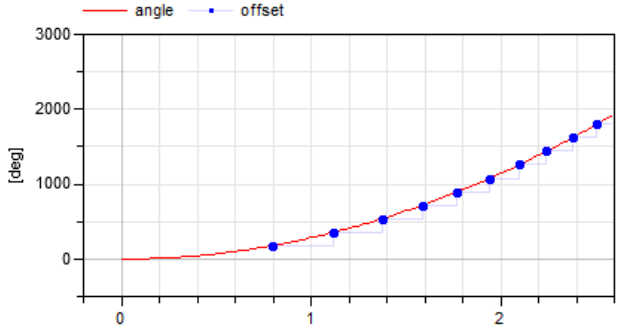
In Modelica, a non-periodic clock can only be introduced by using an explicit clock constructor. The factors of sub-sampling or super-sampling must be parameter expressions, which mean that neither sub-sampling nor super-sampling can construct a clock with varying interval from a periodic clock. It is also required that there must be only one clock constructor, c, in the same base-clock partition if c is a non-periodic clock. All this means that we can construct a new clock c0 that is a super-sampled clock of c, such that all other clocks can be modeled as pure sub-sampling clocks of c0. As we have described, there are no issues in making a faster clock c0 by super-sampling c. The sub-sampling of c0 to implement all the sub-clocks is then just a matter of counting ticks and picking the nth samples.

2.9 Boolean Clocks

It is also possible to define clocks that tick when a Boolean expression changes from false to true. For example assume that a clock shall tick whenever the shaft of a drive train passes 180°. This can be defined as (*Otter, et.al. 2012*):

```
      w = der(angle);
J*der(w) = tau;
when Clock(angle >= hold(offset)+Modelica.Constants.pi) then
  offset = sample(angle);
end when;
```

At the simulation start the discrete variable offset has a start value of zero. Therefore, the first clock tick appears when angle becomes larger as 180°. Then, offset is set to the actual angle, and the next clock tick appears at another full rotation of the shaft. Note, that the Boolean expression is continuous-time, and therefore the clocked variable offset cannot be directly used, but must be casted from a clocked to a continuous-time variable with operator hold. A typical simulation result is shown in the next figure:



Operators subSample, superSample, shiftSample and backSample can also be applied on Boolean clocks. However, there are restrictions. For example, superSample(..) cannot introduce new ticks because the next clock tick is not known in advance. Example:

```
Clock u =  Clock(sine(time) > 0);
Clock y1 = subSample(u,4);
Clock y2 = superSample(y1,2);    // fine y2 = subSample(u,2)
Clock y3 = superSample(u, 2);    // error
```

2.10 Discretized continuous time

A partition (i.e., a set of equations) that is marked by sample, hold, subSample, superSample etc. operators is called a "clocked partitions". There are two different kinds of clocked partitions:

Clocked discrete-time partition
This is the type of partition discussed so far, consisting of algebraic equations, potentially using operators previous(..) and interval(..) in the equations.

Clocked discretized continuous-time partition
This is a partition where the operator der(..) is used (and then previous(..) and interval(..) must not be present). In such a case a set of differential and algebraic equations is marked to be a clocked partition. The semantics is that at clock ticks these equations are solved with a specified integration method from the previous to the next clock tick. The integrator for such a partition is propagated (inferred) similarly as a clock and therefore it suffices to define it at a few places.

This is a powerful feature since in many cases it is no longer necessary to manually implement discrete-time components but it suffices to just build-up a controller with continuous-time components and then sample the input signals and hold the output signals.

In the following example a continuous-time PI controller that gets a reference and a measurement signal as input is automatically transformed to a clocked partition:

```
model ClockedPI
  parameter Real k;
  parameter Real T;
  input Real y_ref;
  input Real y_mes;
  output Real u(start=0.0);
  discrete Real e;
  discrete Real x;
  discrete Real ud;
  Clock c = Clock(Clock(0.1), solverMethod="ImplicitEuler");

equation
  // Sampling the inputs
  e = sample(y_ref,c) - sample(y_mes);
```

```
  // PI controller
  der(x) = e/T;
  ud = k*(x + e);

  // Holding the output
  u = hold(ud);
end ClockedPI;
```

With the declaration Clock(c, solverMethod), the solverMethod (defined as String) is associated to clock c and the partitions to which this clock is associated are solved with the specified solver method (= integration method). As already mentioned, this feature can be used to discretize continuous-time blocks. Also, nonlinear plant models can be inverted and the inverse model can be discretized and used, say, as feedforward controller part in a sampled data controller, see (*Otter, et. al. 2012*). Furthermore, this feature can be utilized for multi-rate real-time simulations where a model is partitioned in different parts and these parts are solved with different integration methods and step sizes.

3 Synchronous Operators

All newly introduced operators of the synchronous extension to Modelica have been sketched so far. In this section, a short overview of these operators is given:

Clock Constructors

Clock(): Returns a clock that is inferred

Clock(i,r): Returns a variable interval clock where the next interval at the current clock tick is defined by the rational number i/r. If i is parameteric, i.e., a literal, a constant, a parameter or an expression of those kinds, the clock is periodic.

Clock(ri): Returns a variable interval clock where the next interval at the current clock tick is defined by the Real number ri. If ri is parametric, the clock is periodic.

Clock(cond, ri0): Returns a Boolean clock that ticks whenever the condition cond changes from false to true. The optional ri0 argument is the value returned by operator interval() at the first tick of the clock.

Clock(c,m): Returns clock c and associates the solver method m to the returned clock .

Base-clock conversion operators

sample(u,c): Returns continuous-time variable u as clocked variable that has the optional argument c as associated clock.

hold(u): Returns the clocked variable u as piecewise constant continuous-time signal. Before the first tick of the clock of u, the start value of u is returned.

Sub-clock conversion operators

subSample(u,factor): Sub-samples the signal or clock u by the integer factor. If factor is not present, it is inferred.

superSample(u,factor): Super-samples the signal or clock u by the integer factor. If factor is not present, it is inferred.

shiftSample(u,c,r): Shifts the clock of a signal or clock u forward in time.

backSample(u,c,r): Shifts the clock of a signal or clock u backward in time. Before the first tick of the clock of u, the start value of u is returned.

Other operators

previous(u): At the first tick of the clock of u, the start value of u is returned. At subsequent clock ticks, the value of u from the previous clock activation is returned.

interval(u): Returns the interval between the previous and the present tick of the clock to which signal u is associated. The interval is returned as a Real number.

4 Base-clock and Sub-clock Partitioning

Consider the example SpeedControl in section 2.1. The variables and equations of MassWithSpringDamper form a well-defined continuous-time model together with the equation f = hold(u) from SpeedControl if we view u as a known input. Similarly the variables and equations added in SpeedControl when extending from MassWithSpringDamper form a well-defined discrete system if we disregard the equation f = hold(u), which already is used in the continuous time system and if we view v, referred in the equation vd = sample(v, Clock(0.01)) as a known input. We have now decomposed the system in a *continuous-time partition* and in a *discrete-time partition*.

For the general case, we observe that the sample and hold operators serve an important role as identifying the interfaces between the two kinds of partitions. The first argument of sample identifies inputs to discrete-time partitions that must be provided by continuous time partitions. Similarly the first argument of hold identifies inputs to continuous-time partions that must be provided by discrete-time partitions. If the first arguments are expressions, auxiliary variables are introduced.

DOI 10.3384/ecp1207615

The idea of the base-clock decomposition is to decompose the variables and the equations into sets where the equations only refer to variables of its own set if we neglect references of the first argument of sample and hold. There are simple algorithms for doing this, for details, see (*Modelica Association 2012*).

It must then be possible to classify a partition as either continuous-time or discrete-time. Use of previous, subSample, superSample, shiftSample or backSample or appearances of clocks or clock constructors requires the partition to be discrete-time. The global variable time can only be referenced in a continuous time partition.

The derivative operator is clearly a continuous-time operator. However, it may appear in a discrete-time partition, because there are features to have them automatically discretized by defining appropriate solver clocks, see section 2.10.

The discrete time partitions are further divided into sub-clock partitions by the same procedure while treating the first argument of the operators subSample, superSample, shiftSample or backSample as known inputs.

The result of sub-clock partitioning for the model ControlledMass in section 2.5 is:

Continuous-time partition:
```
der(x) = v;
m*der(v) = f - k*x - d*v;
f = hold(uInner);
```

Discrete-time sub-partition 1:
```
xd = sample(x, cOuter);
eOuter = xref-xd;
intE = previous(intE) + eOuter;
uOuter = KOuter*(eOuter + intE/Ti);
```

Discrete-time sub-partition 2:
```
xdFast = sample(x, cFast);
aux1 = (xdFast-previous(xdFast))/interval();
```

Discrete-time sub-partition 3:
```
vd = subSample(aux1, 2);
vref = backSample(aux2, 2, 3);
uInner = KInner*(vref-vd);
```

Discrete-time sub-partition 4:
```
aux2 = superSample(uOuter, 5);
```

5 Rationale for Clocked Semantics

This section describes why the synchronous language elements have been introduced in Modelica 3.3, by analyzing the issues of Modelica 3.2 regarding control systems implementation.

Modelica 3.2 has both continuous-time and discrete-time equations. Discrete-time equations are enclosed in when-clauses and are only executed at certain events, i.e. these equations are only valid instantaneously, not always. Furthermore, the discrete-time equations are not general equations, since the left hand-side of an equation in a when-clause must be a variable reference. It is for example not allowed to write in a when-clause: "A*x = b". The synchronous features of Modelica 3.3 remove this restriction and general equations are allowed in clocked partitions and in particular also in clocked when-clauses.

In order to handle such instantaneous equations, a special semantics regarding the definition of variables was introduced. A variable that is assigned by an instantaneous equation keeps its value until the next event when it is assigned again (= automatic "hold" semantics). This implies that the value of such a discrete-time variable could be read at any time by another instantaneous equation or continuous-time equation.

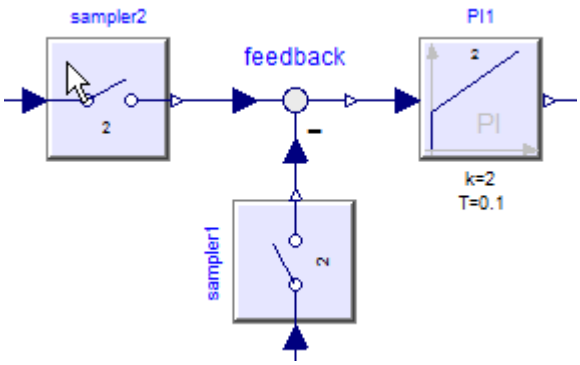
Such semantics can, however, be error prone when different discrete-time equations are not correctly synchronized (see example below). The synchronous features of Modelica 3.3 remove this problem.

Periodically sampled control systems can be defined with standard Modelica 3.2 when-clauses and the sample operator. For example:

```
when sample(0,3) then
  xd = A*pre(xd) + B*y;
   u = C*pre(xd) + D*y;
end when;
```

This approach to define periodically sampled data systems has the following drawbacks that are not present with the solution using clocks and clocked equations described earlier in this paper:

Sampling errors cannot be detected:
All current Modelica libraries modeling sampled data systems, such as Modelica.Blocks.Discrete, or Modelica_LinearSystems2.Controller (*Baur, et.al. 2009*) provide a set of blocks where at *every* block *instance* the sample period has to be defined in some way. For example, the following figure shows part of a control system modeled with the Modelica_LinearSystems2 library:



At every discrete block (here: sampler1, sampler2, PI1) a sampleFactor has to be given defining that the block equations are sampled at a multiple of a base sampling rate (which is propagated via inner/outer to all instances). This factor is shown in the icons (here: "2"). If the modeler accidentally gives a different number at one of the blocks (e.g., at "sampler1"), then this is still a correct Modelica model and a translator has to accept it, although this controller is erroneous.

Furthermore note that component "feedback" is still a continuous-time model without a when-clause. If everything is correctly modeled, the "effect" of the above model is that of a sampled data system with one periodic sampling rate. However, it is easy to make a mistake (e.g. forgetting "sampler1", or using a sampleFactor of 3 at one component), and then the resulting model does no longer describe the desired controller, but is still a valid Modelica model.

Worse, there is no easy way for a tool to figure out which equations belong to one partition that should be downloaded to a hardware device (e.g., describes the above figure one controller with one sample rate, or three different controllers that are connected by the continuous-time block "feedback"?). Due to the automatic sample and hold semantics of when-clauses in Modelica, it is not possible to fix this with Modelica 3.2 language elements.

With the synchronous language elements partitions are identified that belong to the same clock. The sampling rate has to be defined only at one place. Sampling errors can be easily detected, since then the requirement is violated somewhere that all variables in a clocked equation must belong to the same clock.

Unnecessary initial values have to be defined:
Due to the automatic sample and hold semantics, all variables assigned in a when-clause must have an initial value because they might be used before they are assigned a value the first time. Example:

```
when b then
  y1 = 2*x;
end when;
y2 = 2*y1;
```

Since the continuous-time equation `y2 = 2*y1` is valid all the time, including during initialization, a value for `y1` is needed all the time. The when-clause in the example is not active during initialization, and therefore an initial value for `y1` has to be provided. In general, it is too difficult and probably impossible that a tool can figure out whether an initial value for a discrete-time variable in Modelica 3.2 is needed or not. The only safe way is therefore to provide initial values for all discrete-time variables, although in reality, only a small sub-set of the discrete-time variables needs an initial value.^

With the synchronous language elements this is different: Start values are required for the first arguments of some operators (previous, hold, backSample). For all other variables, it is guaranteed that a start value is not needed for initialization (it might be needed as guess value for an iteration variable of a nonlinear equation system).

Inverse models not supported in discrete systems:
It is not possible to use a continuous-time model in when clauses. However, this feature is highly desirable. For example, some advanced controllers use an inverse model of a plant in a controller, see (*Looye et. al. 2005*). This powerful feature of Modelica to use a nonlinear plant model in a controller is only available for continuous-time systems, but not for discrete-time systems. With Modelica 3.2, modelers therefore have to export an inverse plant model and, e.g. Dymola provides the export option to include an integration method and treat the exported component from the outside as discrete-time system. It is then possible to import this discrete-time component in another environment, but not in a Modelica model. With clocked equations of Modelica 3.3, clocked controllers with continuous-time models can be directly defined in Modelica, see section 2.10.

Efficieny degradation at event points:
Simulating a continuous-time plant and a discrete-time controller in Modelica 3.2 together results in an event iteration at a sample instant. A when-clause with a sample(..) condition is evaluated exactly once at such an event instant. However, the continuous-time model to which the sampled data controller is connected will be evaluated typically three times at a sample instant: Once, when the sample instant is reached, once to evaluate the continuous equations at the sample instant, and once when an event iteration occurs since a discrete variable v is changed and pre(v) appears in the equations. Since a sampled system is only evaluated once at a sample instant, i.e., at a particular time instant, event iteration should not be necessary since the discrete-time variables cannot be changed by the event iteration. However, it seems to be difficult to figure this out automatically for a Modelica 3.2 model and therefore Modelica tools, including Dymola, have usually at least one unnecessary evaluation of the continuous-time equations at a sample instant.

With clocked equations described in the next sections a tool does not need to trigger an event iteration, because it is guaranteed that all equations belonging to a periodic or non-periodic interval clock are evaluated exactly once at an event instant, and

DOI 10.3384/ecp1207615

variables computed in such a partition cannot be used outside of the partition (only with a cast operator the most recent available value of a clocked variable v can be inquired outside of the clocked partition, but not previous(v)), and therefore event iteration cannot give a different result. Therefore, it is easy for a tool to avoid the unnecessary re-evaluation of the continuous-time equations at an event triggered by a clock.

6 Conclusions

We have introduced synchronous features in Modelica. For a discrete-time variable, its clock is associated with the variable type. Special operators have to be used to convert between clocks. This gives an additional safety since correct synchronization is guaranteed by the compiler. It would have been very hard to correctly implement the last version of the example control system without such help from the compiler.

7 Acknowledgements

The authors are very thankful to Albert Benveniste, Marc Pouzet, Benoit Caillaud, Timothy Bourke, Francois Dupont, Daniel Weil, Fabien Gaucher, Torsten Blochwitz, Peter Fritzson, Hans Olsson and Modelica Association members for stimulating discussions and feedback during evolutions of the Modelica 3.3 specification.

Parts of this work were supported by the German BMBF (Förderkennzeichen: 01IS08002), and the Swedish VINNOVA (funding number: 2008-02291) within the ITEA2 MODELISAR project (http://www.itea2.org/project/result/download/result/5533). The authors appreciate the partial funding of this work.

References

Baur M., Otter M., and Thiele B. (2009): **Modelica Libraries for Linear Control Systems**. Proceedings of 7th International Modelica Conference, Como, Italy, September 20-22. www.ep.liu.se/ecp/043/068/ecp09430068.pdf

Benveniste A., Caspi P., Edwards S.A., Halbwachs N., Le Guernic P., and Simone R. (2003): **The Synchronous Languages Twelve Years Later**. Proc. of the IEEE, Vol., 91, No. 1. www.irisa.fr/distribcom/-benveniste/pub/synch_ProcIEEE_2002.pdf

Colaco J.-L., and Pouzet M. (2003): **Clocks as First Class Abstract Types**. In Third International Conference on Embedded Software (EMSOFT'03), Philadelphia, Pennsylvania, USA, October 2003. www.di.ens.fr/~pouzet/lucid-synchrone/papers/emsoft03.ps.gz

Elmqvist H., Gaucher F., Mattsson S.E, and Dupont F. (2012): **State Machines in Modelica**. Proceedings of 9th International Modelica Conference, Munich, Germany, September 3-5.

Forget J., F. Boniol, D. Lesens, C. Pagetti (2008): **A Multi-Periodic Synchronous Data-Flow Language**. In 11th IEEE High Assurance Systems Engineering Symposium (HASE'08), Dec. 3-5 2008, Nanjing, China, pp. 251-260. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reoad=true&arnumber=4708883&contentType=Conference+Publications

Modelica Association (2012): **Modelica Language Specification Version 3.3**. https://www.modelica.org/documents/ModelicaSpec33.pdf.

Otter M., Thiele B., and Elmqvist H. (2012): **A Library for Synchronous Control Systems in Modelica**. Proceedings of 9th International Modelica Conference, Munich, Germany, September 3-5.

Pouzet M. (2006): **Lucid Synchrone, Version 3.0, Tutorial and Reference Manual**. http://www.di.ens.fr/~pouzet/lucid-synchrone/

Looye G., Thümmel M., Kurze M., Otter M., and Bals J. (2005): **Nonlinear Inverse Models for Control**. Proceedings of 4th International Modelica Conference, ed. G. Schmitz, Hamburg, March 7-8. https://www.modelica.org/events/Conference2005/online_proceedings/Session3/Session3c3.pdf

DOI
10.3384/ecp1207615