Discrete-time models for control applications with FMI

Rüdiger Franke[1] Sven Erik Mattsson[2] Martin Otter[3] Karl Wernersson[2] Hans Olsson[2]
Lennart Ochel[4] Torsten Blochwitz[5]

[1]ABB, ruediger.franke@de.abb.com, [3]DLR, martin.otter@dlr.de,
[2]Dassault Systémes, {svenerik.mattsson, karl.wernersson, hans.olsson}@3ds.com,
[4]Uni Linköping, lennart.ochel@liu.se, [5]ESI ITI, torsten.blochwitz@esi-group.com

Abstract

The paper proposes an extension of FMI 2.0 for the rigorous treatment of discrete-time models. This includes the introduction of discrete-time states, the declaration of clocks in the model description and an extension of the calling interface for the external activation of clocks by an importing environment.

The synchronous discrete-time extension enables for the first time the synchronization of FMUs with the environment and with other FMUs. It specializes the existing generic event mechanism of FMI 2.0 and maps to synchronous features of Modelica.

Discrete-time FMUs are needed for the generation of controller code from functional models. This paper outlines different use cases, including a simple PI controller, feed forward control with a nonlinear inverse model and nonlinear model predictive control.

The FMI change proposal FCP-001 and the Modelica change proposal MCP-0024 describe the proposed extensions in more detail. Test implementations exist in the simulation tools Dymola and OpenModelica and in the importing optimization solver HQP. The use cases given in this paper served for further refinement of the change proposals and the test implementations.

Keywords: Modelica, Synchonous modeling, Inline Integration, Model-based Control, Nonlinear Inverse Model, Feed Forward Control, NMPC.

1 Introduction

Control systems are composed of interconnected control blocks that must synchronize with each other and with real time. This requires precise time event handling and discrete states.

Modelica 3.3 extends the scope from a language primarily intended for physical systems modeling to modeling of complete systems. In particular, new synchronous language primitives were introduced for increased correctness of control systems implementation (Elmqvist et al, 2012).

Version 2.0 of the FMI standard omitted precise time event handling. The design was considered complicated at the time of the release of FMI 2.0 since several aspects have to be considered (Blochwitz et al, 2012):

- The synchronous features of Modelica 3.3 should be supported.
- FMI should also be useable by tools that do not support synchronous time event handling.
- The time event handling is to be defined in a way that allows backward compatible extensions.

This paper discusses the progress made recently. The work resulted in a new version of the FMI change proposal FCP-001 (Otter et al, 2016) and in the Modelica change proposal MCP-0024 (Franke, 2016). This paper summarizes the change proposals, provides use cases and investigates examples using test implementations in the simulation tools Dymola and OpenModelica and in the optimization solver HQP (Franke and Arnold, 1997).

2 Synchronous Modelica

Modelica has always supported continuous-time variables and discrete-time variables defined as piecewise continuous and piecewise constant functions of time, respectively. Both may change discontinuously at time instants, so called events. Events are treated at runtime.

The synchronous features of Modelica 3.3 introduce a new Clock type. Clock variables $c(t^k)$ are special discrete variables that are active (are ticking) at particular time instants, see Figure 1.

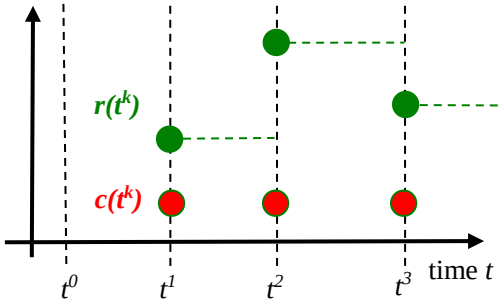

Figure 1: Clock variable c and clocked variable r

A clocked discrete-time variable $r(t^k)$ is associated with exactly one clock. This enables the partitioning of a model into sub-models for each clock at translation time. A clock defined for one variable of a partition automatically propagates to all other variables of this partition. This enables generic discrete-time models with inferred sample times.

A clocked discrete-time variable only has a value when the clock ticks. Continuous-time variables may be converted to clocked variables with the sample operator. A clocked variable may be converted to a continuous-time variable with the hold operator.

A clocked partition is mathematically defined as:

$$x^k = f^k(x^{k-1}, u^k, t^k), \quad k = 0,1,2,\dots,k_{end}-1, \quad x^{-1} = x_{start} \tag{1}$$

$$y^k = g^k(x^{k-1}, u^k, t^k), \quad k = 0,1,2,\dots,k_{end} \tag{2}$$

Here x^k are discrete-time states, u^k are inputs, y^k are outputs, k is the k-th tick of the associated clock and k_{end} is the final tick. The discrete-time states are defined with difference equations as function f^k of the previous values x^{k-1} and the inputs u^k.

2.1 Clocked continuous-time models

A clocked partition may contain differential equations. This allows the embedding of regular continuous-time models from given Modelica libraries. The Modelica translator brings the equations of a clocked partition to the form of an ODE or semi-explicit index-1 DAE:

$$\frac{dx(t)}{dt} = f[x(t), u(t)]$$
$$0 = h[x(t), u(t)] \tag{3}$$

The translator then applies a specified solver method to convert continuous-time differential equations to discrete-time difference equations. This mixed symbolic/numeric approach is also known as inline integration (Elmqvist et al, 1995).

Basic solver methods are implicit Euler, explicit Euler and semi-implicit Euler. Application of implicit Euler results in:

$$\frac{x^k - x^{k-1}}{t^k - t^{k-1}} = if\ k = 0\ then\ 0\ else\ f(x^k, u^k)$$
$$0 = h(x^k, u^k) \tag{4}$$

Explicit Euler avoids the implicit equation system for the states x^k in (4) for non-stiff models. It results in:

$$\frac{x^k - x^{k-1}}{t^k - t^{k-1}} = if\ k = 0\ then\ 0\ else\ f(x^{k-1}, u^{k-1})$$
$$0 = h(x^{k-1}, u^{k-1}) \tag{5}$$

The use of u^{k-1} in (5) leads to the introduction of additional discrete-time states for the delay of inputs by one sample period, even though this is typically not wanted. Semi-implicit Euler avoids the delay of inputs and implicit dependencies of states for non-stiff models. It results in:

$$\frac{x^k - x^{k-1}}{t^k - t^{k-1}} = if\ k = 0\ then\ 0\ else\ f(x^{k-1}, u^k)$$
$$0 = h(x^{k-1}, u^k) \tag{6}$$

Many more solver methods exist with specific advantages and drawbacks. The choice of the best solver method depends on the model at hand. This is why it is advantageous that inline integration embeds the most appropriate solver method into an exported model.

Modelica 3.3 defines the operators `previous(x)` to access x^{k-1} and `interval()` to determine $t^k - t^{k-1}$.

The Modelica change proposal MCP-0024 introduces the operator `firstTick()` to determine if $k = 0$ (Franke, 2016).

3 FMI extension

FMI 2.0 defines a generic event mechanism that also covers synchronous models. The drawbacks of this generic mechanism are that discrete states are hidden in the FMU and that the environment does not know any details about the events. This makes it impossible to synchronize events with the environment of an FMU. Thus, it is not possible to re-import an exported FMU with synchronous discrete-time features and achieve a deterministic behavior. Neither it is possible to exploit a discrete-time FMU for advanced applications such as parameter estimation or model predictive control, because the discrete states are hidden.

It is proposed to extend FMI by the following:

1. Declare clocks in modelDescription.xml
2. Declare discrete-time states in modelDescription.xml
3. Let the environment activate clocks in order to enable synchronization with the environment and with other FMUs.

This extension is optional. A model can always hide event details according to FMI 2.0.

3.1 Extension of modelDescription.xml

The “TypeDefinitions” section is extended with a “Clocks” subsection that contains one or more “Clock” entries.

DOI 10.3384/ecp17132507

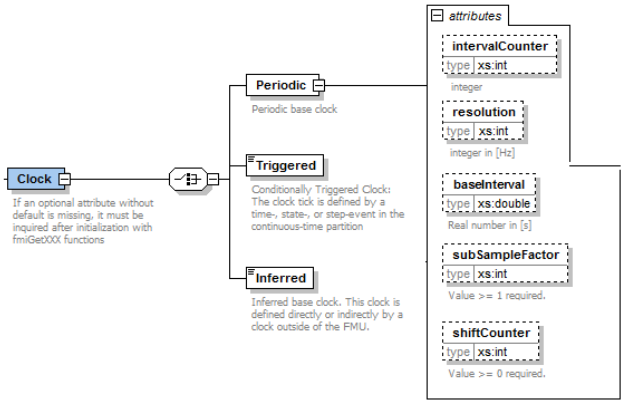

Figure 2: Kinds of Clock

Each Clock may be one of (see Figure 2):

- **Periodic**: the clock ticks periodically with an a priori known interval specified in the model description XML file. A priori known values make the sampling a structural model property for increased correctness at runtime.
- **Triggered**: the clock is activated by a Boolean condition in the model, e.g. for an interval that depends on model variables.
- **Inferred**: the clock is activated from outside the model, e.g. for a generic discrete-time model with arbitrary sample interval. Synchronous models do not require a parameter for the sample time; the clock propagates with clocked variables. Synchronous Modelica models use the interval operator instead of a parameter.

The attributes of Periodic define the clock interval and offset time. The basic clock interval is either specified with a double valued `baseInterval` or with integer valued `intervalCounter` and `resolution`. Both definitions relate to each other with

baseInterval = intervalCounter/resolution

Periodic clocks may be further refined with the attributes `subSampleFactor` and `shiftCounter`. This results in the actual

interval = baseInterval/subSampleFactor

that is delayed by

offsetTime = interval*shiftCounter

The attributes of "ScalarVariable" are extended with two new attributes:

- **previous** marks a discrete-time state, similar to the derivative attribute of continuous-time states. The value is an index to the variable providing the previous value of the discrete-time state.
- **clockIndex** associating a variable uniquely with a clock in the "Clocks" section.

Finally, the "ModelStructure" section is extended with a subsection "DiscreteStates". It provides an ordered list of all exposed discrete states with their indices in the "ScalarVariable" list. Each entry of "DiscreteStates" may declare the dependencies from known inputs, continuous-time states and other discrete-time states. The dependencies are defined under the assumption that the respective clock ticks.

3.2 Extension of the C calling API

The C calling API is extended with four new functions that can be called during the event mode of an FMU.

A clock is activated by the environment for the current time instant by the function `fmi2SetClock`, and the status of a clock can be queried with the function `fmi2GetClock`:

```
fmi2Status fmi2SetClock (
   fmi2Component c,
   const fmi2Integer clockIndex[],
   size_t nClockIndex,
   const fmi2Boolean tick[],
   const fmi2Boolean* subactive);
```

Set the clock activation status by providing the indices of the corresponding clocks with respect to the xml element "`<TypeDefinitions><Clocks>`" and values. A clock is activated at the current time instant if `tick[i] = fmi2True`, otherwise the clock is deactivated. The environment may set `subactive[i] = fmi2True` to only evaluate the output equations (2) and replace the state equations (1) with

$$x^k = x^{k-1} \quad (7)$$

This is similar to the treatment of clocked continuous states at initial time, see (4), (5) and (6). The argument `subactive[i]` defaults to `fmi2False` if a NULL pointer is passed.

```
fmi2Status fmi2GetClock (
   fmi2Component c,
   const fmi2Integer clockIndex[],
   size_t nClockIndex,
   fmi2Boolean tick[]);
```

Query whether a set of clocks is active by providing the indices of the corresponding clocks with respect to the xml element "`<TypeDefinitions><Clocks>`".

A clock interval is set by the environment for the current time instant by the function `fmi2SetInterval`, and it can be queried with the function `fmi2GetInterval`:

```
fmi2Status fmi2SetInterval(
   fmi2Component c,
   const fmi2Integer clockIndex[],
   size_t nClockIndex,
   const fmi2Real interval[]);
```

Set the interval value between the previous and the present tick of the clock.

```
fmi2Status fmi2GetInterval(
   fmi2Component c,
   const fmi2Integer clockIndex[],
   size_t nClockIndex,
   fmi2Real interval[]);
```

Query the interval value for the provided clocks (periodic or non-periodic). If the clocks are non-periodic, the interval has to be queried at every clock tick, to define the follow-up clock tick.

3.3 Extension of importing environment

The importing environment parses the model description XML file and activates periodic and inferred clocks during simulation. It activates periodic clocks at sample intervals specified in the model description XML file. It activates inferred clocks as needed by the environment (e.g. with an externally specified sample interval or if the clock of a connected FMU ticks). The FMU itself activates Triggered clocks.

This extension does not change the overall calling sequence of C functions for model exchange. The environment calls the new API functions additionally during event mode as follows:

0. **Enter event mode:**
   FMI 2.0 enters the event mode either after initialization (call to function `fmi2ExitInitializationMode`) or during simulation (call to function `fmi2EnterEventMode`).
1. **Activate clocks and set inferred intervals:**
   An FMU activates triggered clocks itself. The environment may query the clock activation status with the function `fmi2GetClock`. The environment sets the activation status of periodic and inferred clocks by calling `fmi2SetClock`. Moreover, the environment calls `fmi2SetInterval` for inferred clocks. It may query the clock interval, e.g. for triggered clocks, with the function `fmi2GetInterval`.
2. **Evaluate clocked equations:**
   The evaluation is triggered by `fmi2GetXXX` for clocked variables during event mode or by `fmi2NewDiscreteStates`. The FMU copies x^k to x^{k-1} and evaluates the discrete-time equations, updating x^k, if the corresponding clock is active. The FMU resets the clock activation after one evaluation. This means that the environment must activate the clock again if it wants to re-evaluate clocked equations, for instance to treat an algebraic loop (see below)
3. **Leave event mode:**
   The functions `fmi2NewDiscreteStates` and `fmi2Reset` leave event mode and deactivate all clocks.

The environment might need to evaluate clocked discrete-time equations multiple times at one time instant, for instance to iteratively solve an algebraic loop among multiple connected FMUs or to calculate partial derivatives for optimization. The environment can either call `fmi2GetXXX` within event mode, triggering the evaluation of clocked equations if the respective clocks are active. The FMU will update discrete-time states and deactivate the clocks. The environment may reset discrete-time states by calling `fmi2SetXXX`, re-activate clocks and call `fmi2GetXXX` again for multiple evaluations. This also applies to all kinds of clocks, including also triggered clocks. Alternatively, the environment may enter event mode multiple times and reset discrete-time states for multiple evaluations.

The environment might be interested in the dependencies of model outputs from inputs and given discrete-time states, independently of the state equations. This can be achieved by passing `subactive=fmi2True` to `fmi2SetClock`.

3.4 Relation to Simulink S-functions

The basic concept of the proposed FMI extension is well known from other simulation technologies. The widely used simulation tool Simulink, for example, supports an arbitrary number of discrete sample times in an S-function, in addition to continuous-time equations. Lacking an XML file, the sample times are defined in S-function methods (C functions). The most important methods are listed here and related to the proposed FMI extension.

`mdlInitializeSizes(SimStruct *S)`

This method declares the number of sample times with

```
ssSetNumSampleTimes(S, n);
```

It corresponds to the number of Clock entries in the model description XML file.

`mdlInitializeSampleTimes(SimStruct *S)`

This method initializes each sample time `i = 0,…,n-1` with an interval and an offset time by calling

```
ssSetSampleTime(S, i, interval);
ssSetOffsetTime(S, i, offsetTime);
```


DOI 10.3384/ecp17132507

The argument `interval` may take the special values `CONTINUOUS_SAMPLE_TIME` for a continuous-time model and `INHERITED_SAMPLE_TIME`, corresponding to an inferred sample time of this proposal.

Moreover, the argument `interval` may take the special value `VARIABLE_SAMPLE_TIME` and the argument `offsetTime` may take the special value `FIXED_IN_MINOR_STEP_OFFSET`, relating discrete-time sub-models to numerical integration steps of continuous-time sub-models. Such sampling can be implemented with triggered clocks of this proposal, if the FMU activates clocks itself during transitions between continuous-time mode and event mode.

Simulink will activate any sample time from outside S-functions in the case of sample hits and call the function

```
mdlUpdate(SimStruct *S, int_T tid)
```

A model must query the activation status and evaluate the respective discrete-time equations.

```
if (ssIsSampleHit(S, i, tid)) {
  // update discrete states that belong
  // to sample time i
}
```

Discrete states are accessed with

```
real_T *x = ssGetRealDiscStates(S);
```

This FMI proposal uses variable references to access discrete states. It introduces optional previous values for discrete-time states. Previous values allow the definition of dependencies on x^{k-1} in the model structure, see (1), (2). The environment only sets the actual value x^k. An FMU with previous values copies x^k to x^{k-1} prior to the evaluation of clocked equations.

4 Use Cases

This section lists use cases for control applications. A chemical process model serves as an example.

4.1 Exemplary chemical process model

We consider a continuous stirred-tank reactor (CSTR) with cooling jacket published by (Engell, Klatt, 1993). This highly nonlinear model exhibits interesting properties, like nonminimum phase behavior and change of steady-state gain at the main operating point. (Chen et al, 1995) propose this example as a benchmark problem for nonlinear control system design.

The following reaction describes the chemical process:

$$A \overset{k_1}{\rightarrow} B \overset{k_2}{\rightarrow} C$$

$$2A \overset{k_3}{\rightarrow} D \tag{8}$$

The reactor primarily transforms cyclopentadiene (substance A) to the product cyclopentenol (substance B). An unwanted subsequent reaction transforms B to cyclopentanediol (substance C). Another unwanted parallel reaction transforms A to the by-product dicyclopentadiene (substance D). The mathematical model contains the component balances for A and B:

$$\frac{dc_A}{dt} = \frac{\dot{V}_F}{V_R}\left(c_{A,F} - c_A\right) - k_1(T)c_A - k_3(T){c_A}^2$$

$$\frac{dc_B}{dt} = -\frac{\dot{V}_F}{V_R}c_B + k_1(T)c_A - k_2(T)c_B \tag{9}$$

with the reaction coefficients

$$k_i(T) = k_{i,0}e^{\frac{E_i}{T}}, \quad i = 1,2,3 \tag{10}$$

as well as the energy balances for the reactor and the cooling jacket:

$$\frac{dT}{dt} = \frac{\dot{V}_F}{V_R}(T_F - T) + \frac{k_w A_R}{\rho C_p V_R}(T_K - T) - \frac{1}{\rho C_p}[k_1(T)c_A H_1 + k_2(T)c_B H_2 + k_3(T){c_A}^2 H_3]$$

$$\frac{dT_K}{dt} = \frac{1}{m_K C_{p,K}}\left[\dot{Q}_K + k_w A_R(T - T_K)\right] \tag{11}$$

Table 1 lists the model parameters.

Table 1: Parameters of CSTR model

Name	Value	Description
$k_{1,0}$	$1.287\ h^{-1}$	Collision factor one
$k_{2,0}$	$1.287\ h^{-1}$	Collision factor two
$k_{3,0}$	$9.043\ (molA\ h)^{-1}$	Collision factor three
E_1	$-9758.3\ K$	Activation energy one
E_2	$-9758.3\ K$	Activation energy two
E_3	$-8560\ K$	Activation energy three
H_1	$4.2\ kJ/molA$	Reaction enthalpy one
H_2	$-11.0\ kJ/molB$	Reaction enthalpy two
H_3	$-41.85\ kJ/molC$	Reaction enthalpy three
ρ	$0.9342\ kg/l$	Density reactant
C_p	$3.01\ kJ/(kg\ K)$	Heat capacity reactant
k_w	$1.12\ kW/(m^2\ K)$	Heat transfer jacket
A_R	$0.215\ m^2$	Surface reactor
V_R	$0.01\ m^3$	Volume reactor
m_K	$5.0\ kg$	Mass cooling jacket
$C_{p,K}$	$2.0\ kJ/(kg\ K)$	Heat capacity coolant

Table 2: Desired steady operating point

Name	Value	Description
$c_{A,F}$	5.10 $molA/l$	Feed concentration
T_F	104.9 $°C$	Feed temperature
$\dot{V}_F/V_R$	14.19 h^{-1}	Feed flow rate
Q_K	−1113.5 kJ/h	Heat removal
c_A	2.14 mol/l	Concentration A
c_B	1.09 mol/l	Concentration B
T	114.2 $°C$	Reactor temperature
T_K	112.9 $°C$	Coolant temperature

Table 2 gives the desired operating point for optimal yield. The following subsections use this CSTR model to outline different use cases.

4.2 Functional Engineering

Modelica system models combine physical plant models with control models. This enables the study the functional behavior of a system with simulation. Having a functional model available, the actual controller code shall be generated automatically from the control models.

Figure 3 shows a system model with a CSTR and a PI control for the coolant temperature. The PI controller uses a clock and sample blocks from the Modelica_Synchronous library (Otter et al, 2012). The clock also defines the solver method ImplicitEuler to convert the controller model to discrete time.

The control task is to hold the coolant temperature at the desired operating point, in order to keep the desired concentration of product B.

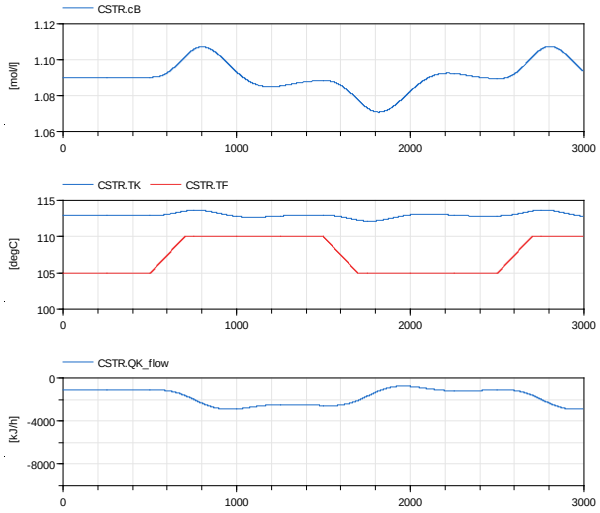

Figure 3: Simulation results for the functional model with plant and controller over 3000s

Figure 6 shows simulation results. The feed temperature CSTR.TF is increased periodically by 5 K. This results in higher reactor temperature and increased concentration CSTR.cB. The PI controller increases heat removal to bring the reactor back to the desired operating point.

Overall the disturbance leads to large deviations of the concentration of the product CSTR.cB from the desired operating point of 1.09 mol/l. This is because the controller sees the disturbance only indirectly if the coolant temperature increases. Moreover the reference value of the coolant temperature is not adjusted to the disturbance.

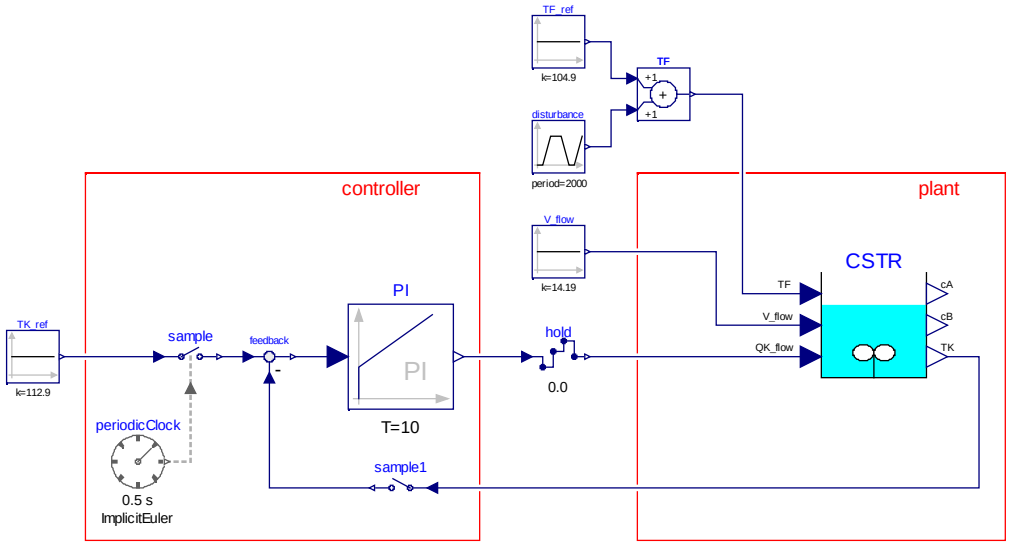

Figure 4: Functional model of a plant with controller

DOI 10.3384/ecp17132507

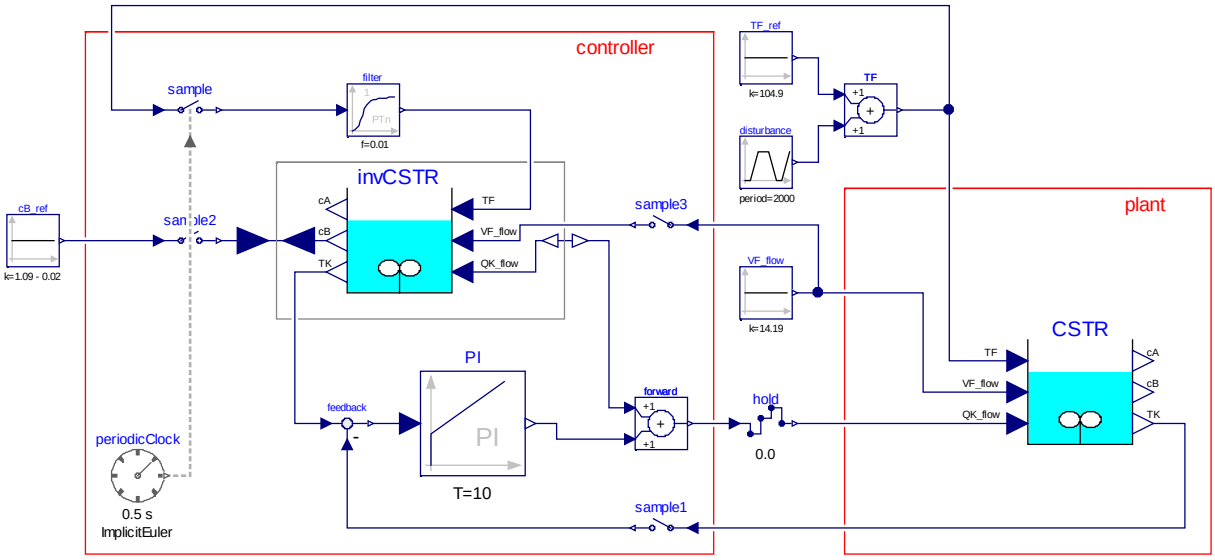

Figure 5: Functional model with advanced controller containing a nonlinear inverse plant model

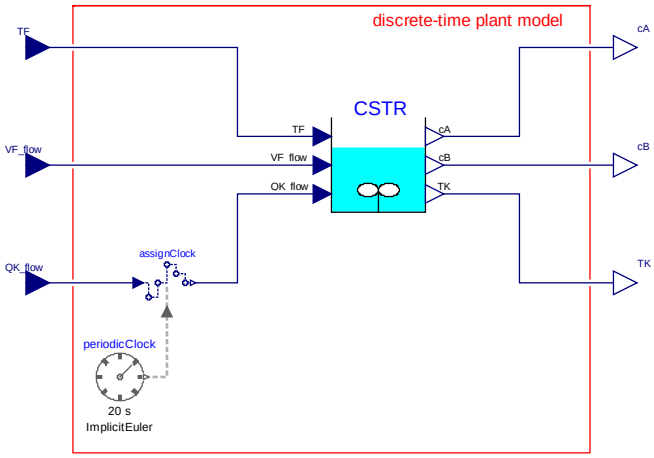

Figure 6: Discrete-time plant model for nonlinear model predictive control

4.3 Nonlinear inverse models for control

Feed forward is a well-known strategy to increase dynamic control performance. Modelica can invert a physical plant model analytically to get an inverse model for the feed forward path of a controller (Looye et al, 2005).

Figure 4 shows an advanced controller with nonlinear inverse model. This increases controller performance for disturbance rejection by converting feed temperature to an appropriate set point for heat removal and reference point for the coolant temperature TK_ref. Moreover, this enhances the controller with an external set point for the concentration of the product B.

Figure 7 shows simulation results. During the first 1000 s the controller adjusts the heat removal for the modified reference cB_ref of 1.07 mol/l. Afterwards the disturbance in the feed temperature is rejected considerably better with feed forward control.

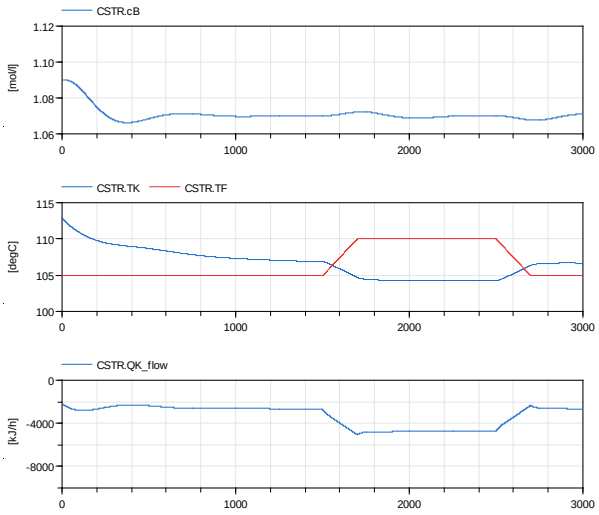

Figure 7: Simulation results for feed forward control with inverse plant model over 3000s

4.4 Discrete-time plant models for nonlinear model predictive control

Nonlinear model predictive control (NMPC) treats an optimal control problem for a given plant model at runtime. The model is used as is, without analytical inversion. This simplifies the treatment of multi-variable constrained problems at the cost of increased computing requirements for numerical optimization at runtime. A model predictive controller takes the following steps during each cycle (Franke et al, 2015):

1. Convert continuous-time physical model to discrete-time model for control.
2. Calculate model sensitivities.
3. Formulate a large-scale nonlinear optimization program spanning multiple time steps.
4. Solve the large-scale nonlinear optimization program.

The synchronous features of Modelica and the discrete-time extension of FMI enable to shift steps 1 and 2 from the runtime to model translation time. Figure 5 shows the CSTR model with clock and solver method assigned.

The resulting exported FMU has the discrete-time states $x = (c_A; c_B; T; T_K)$, the inputs $u = \left(\dot{Q}_K; T_F; \frac{\dot{V}_F}{V_R}\right)$ and the outputs $y = (c_A; c_B; T_K)$. The control task is formulated as discrete-time optimal control problem over the time horizon of 3000s with $k_{end} = 150$ intervals of length 20s. The optimization objective is to minimize quadratic deviations of the concentration of substance B from the desired operating point. A second objective term applies a small penalty to control moves:

$$J = \sum_{k=0}^{k_{end}} (c_B^k - 1.07)^2 + \sum_{k=0}^{k_{end}-1} \left(\frac{\dot{Q}_K^{k+1} - \dot{Q}_K^k}{10^7}\right)^2 \rightarrow \min_{\dot{Q}_K^k} \tag{12}$$

The manipulated extraction of heat is constrained by

$$-9000\ kJ/h < \dot{Q}_K^k < 0\ kJ/h, \quad k = 0, \ldots, k_{end} - 1 \tag{13}$$

The discrete-time state equations in the FMU define further constraints:

$$x^{k+1} = f^k(x^k, u^k), \quad k = 0, \ldots, k_{end} - 1$$
$$x^0 = (2.14;\ 1.09;\ 114.2;\ 112.9) \tag{14}$$

The solver HQP collects all states and the control inputs of all time intervals into one large vector of optimization variables

$$v = (x^0, u^0, x^1, u^1, \ldots, x^{k_{end}-1}, u^{k_{end}-1}, x^{k_{end}}). \tag{15}$$

This results in the large-scale mathematical program

$$\begin{aligned} J(v) &\underset{v}{\rightarrow} min & J&: \mathbb{R}^n \rightarrow \mathbb{R}^1 \\ h(v) &= 0 & h&: \mathbb{R}^n \rightarrow \mathbb{R}^{m_e} \\ g(v) &\geq 0 & g&: \mathbb{R}^n \rightarrow \mathbb{R}^m \end{aligned} \tag{16}$$

with $n = \dim(v)$, $m_e = (k_{end} + 1)\dim(x)$ and $m = 2k_{end}$. HQP applies Sequential Quadratic Programming (SQP) with a sparse Interior Point QP solver to the numerical solution of the mathematical program.

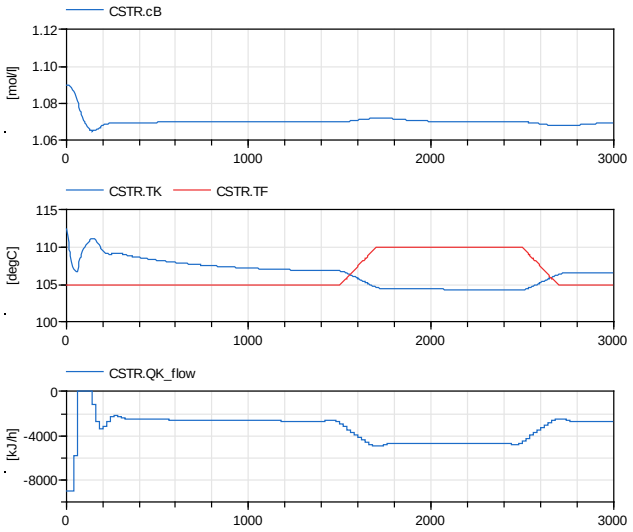

Figure 8: Results of the optimal control problem over a time horizon of 3000s

Figure 8 shows simulation results of the CSTR model for the optimized control trajectory 0 kJ/h The optimal solution exploits the full range between $-9000\ kJ/h$ and $0\ kJ/h$ to arrive at the new reference value $c_{B,ref} = 1.07\ mol/l$ significantly faster. It rejects the disturbance for the feed temperature T_F similar to the controller with nonlinear inverse model.

5 Conclusions

Modelica 3.3 introduced synchronous features that enable the rigorous treatment of discrete-time models. The Modelica_Synchronous library demonstrates the relevance of these features for control (Otter et al, 2012). The simulation tools Dymola and OpenModelica support Modelica_Synchronous so far.

DOI 10.3384/ecp17132507

This paper proposes an extension of FMI 2.0 to make rigorous discrete-time models available for control applications. The extension is backwards compatible. It specializes generic events towards clocks for discrete-time models. Tools that do not support synchronous time event handling can export the same model using generic events as known from FMI 2.0. An importing tool should parse the extensions of the XML file, in particular the Clocks section, activate periodic clocks at the specified intervals and activate inferred clocks on environment needs. Alternatively, an importing tool might reject the FMU if it finds inferred or periodic clocks in the Clocks section. Triggered clocks are activated by the FMU itself and need no support by the importing environment.

The basic concept of activation of sample times by a tool is well known from other simulation technologies, such as Simulink S-functions. The proposed FMI extension exploits the XML model description to associate clocks with variables. This enables deterministic clock propagation among multiple connected FMUs. The optional specification of integer valued clock intervals further enhances clock inference for system level design.

FMI export with synchronous features was implemented in the tools Dymola and OpenModelica. Import was implemented in the optimization solver HQP. The paper motivates the FMI extension with use cases for a highly nonlinear chemical process model. The use cases include functional engineering, nonlinear inverse models for control and nonlinear model predictive control.

The synchronous features of Modelica also include the automatic conversion of continuous-time models to discrete-time models with inline integration. This mixed symbolic/numeric approach simplifies model-based control applications considerably, because it releases an importing environment from the treatment of continuous-time differential equations and sensitivity equations. Run-time efficiency increases.

Discrete-time FMUs with inline integration are a work in progress. Development versions of OpenModelica, Dymola and HQP were used for the optimal control problem in section 4.4. Dymola 2017 was used for the nonlinear inverse model in section 4.3.

Discrete-time FMUs will serve for the investigation of parallel algorithms for automatic differentiation and numerical optimization in the PARADOM project.

Acknowledgements

This work was supported in parts by the Federal Ministry of Education and Research (BMBF) within the project PARADOM (PARallel Algorithmic Differentiation in OpenModelica) – BMBF funding code: 01IH15002E.

References

T. Blochwitz, M. Otter, J. Åkesson, M. Arnold, C. Clauß, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, A. Viel: Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models, 9th International Modelica Conference, Munich, 2012. http://www.ep.liu.se/ecp/076/017/ecp12076017.pdf

H. Chen, A. Kremling, F. Allgöwer: Nonlinear Predictive Control of a Benchmark CSTR, Proceedings 3rd European Control Conference ECC'95, Rome, 1995.

H. Elmqvist, M. Otter, S.E. Mattsson: Fundamentals of Synchronous Control in Modelica, 9th International Modelica Conference, Munich, 2012. http://www.ep.liu.se/ecp/076/001/ecp12076001.pdf

H. Elmqvist, M. Otter, F. Cellier: Inline integration: A new mixed symbolic/numeric approach for solving differential-algebraic equation systems. In Proceedings ESM European Simulation Multiconference, Prague, 1995.

S. Engell, K.-U. Klatt. Nonlinear control of a nonminimum phase CSTR. In Americal Control Conference, Los Angeles, 1993.

R. Franke, E. Arnold: Applying new numerical algorithms to the solution of discrete-time optimal control problems. In: Computer Intensive Methods in Control and Signal Processing: The Curse of Dimensionality, Birhäuser, Basel, 1997.

R. Franke, M. Walther, N. Worschech, W. Braun, B. Bachmann: Model-based control with FMI and a C++ runtime for Modelica. Proceedings of 11th International Modelica Conference, Paris 2015. https://www.modelica.org/events/modelica2015/proceedings/html/submissions/ecp15118339_FrankeWaltherWorschechBraunBachmann.pdf

R. Franke: Initialization of Clocked Discrete States, Modelica Change Proposal MCP-0024 2016. https://svn.modelica.org/projects/MCP/public/MCP-0024_InitializationClockedStates/MCP-0024_InitializationClockedStates.docx

Functional Mock-up Interface for Model Exchange and Co-Simulation, Version 2.0, July 2014.

G. Looye, M. Thümmel, M. Kurze, M. Otter, J. Bals: Nonlinear Inverse Models for Control. Proceedings of 4th International Modelica Conference, Hamburg, 2005. https://www.modelica.org/events/Conference2005/online_proceedings/Session3/Session3c3.pdf

Modelica Association: Modelica – A Unified Ob-ject-Oriented Language for Systems Modeling. Language Specification, Version 3.3. May 9, 2012.

M. Otter, S.E. Mattsson, R. Franke, H. Elmqvist, T. Blochwitz: Discrete States and Time Events in FMI (#353), FMI Change Proposal FCP-001, 2016. https://svn.fmi-standard.org/fmi/trunk/FMI_ChangeProposals/FCP_001_SampledDataSystemsForModelExchange/FMI_Proposal_DiscreteStates_TimeEvents.docx

M. Otter, B. Thiele, H. Elmqvist: A Library for Synchronous Control Systems in Modelica, 9th International Modelica Conference, Munich, 2012. http://www.ep.liu.se/ecp/076/002/ecp12076002.pdf