Model-based Embedded Control using Rosenbrock Integration Methods

Hans Olsson[1] Sven Erik Mattsson[1] Martin Otter[2] Andreas Pfeiffer[2] Christoff Bürger[1] Dan Henriksson[1]

[1]Dassault Systèmes AB, Lund, Sweden,
{Hans.Olsson, SvenErik.Mattsson, Christoff.Buerger, Dan.Henriksson}@3ds.com
[2]DLR, Institute of System Dynamics and Control, Germany,
{Martin.Otter, Andreas.Pfeiffer}@dlr.de

Abstract

Directly generating controller code from models is important for advanced model-based design. This paper describes how Dymola can generate embedded C-code from Modelica models, designed to be easy to embed, with care about minimal foot-print, traceability, and straightforward integration in embedded platforms and gives actual application examples.

The paper focuses on using Rosenbrock methods for index-1 problems (instead of the normal transformation to index 0) that allows Dymola to handle stiff systems in a way that both is theoretically sound and has an upper bound on the execution time per sample.

The stiff systems in the control system often occur due to using an inverse (simplified) model of the real plant in the controller. A nonlinear feedforward controller and a controller with feedback linearization, both applying an inverse model, demonstrate the proposed process by using Rosenbrock methods for embedded code generation.

Keywords: Modelica, inverse models, real-time, embedded, Rosenbrock methods, inline integration, feedforward controller, feedback linearization

1 Introduction

Modelica and Modelica tools such as Dymola are very well suited to model and simulate complex physical systems with primary focus on offline simulation for design and assessment, as well as on online simulation on special purpose hardware, e.g. for hardware-in-the-loop simulations. Modelica models have been used in controller applications where nonlinear Modelica models are part of the real-time control system, see for example (*Looye et al., 2005*). The controller could be designed and assessed with Dymola, however, the actual real-time controller code had to be re-built manually either directly in C or with dedicated software for controller code generation.

There are several activities to extend the tool chains for Modelica models for real-time platforms, for example (*Satabin et al., 2015*) for generation of certified code of simple Modelica models via the SCADE-suite (*SCADE, 2017*), or (*Bertsch et al., 2015*) for utilizing Modelica code on automotive electronic control units.

This paper describes the steps to generate embedded real-time code using a new prototype functionality of Dymola. The goals are (a) to generate code that can be certified for critical applications, (b) to guarantee an upper number of operations so that hard real-time constraints can be fulfilled, and (c) to support advanced controllers that can utilize nonlinear Modelica models in the feedback or feedforward path of the controller, which may require solving nonlinear differential-algebraic equation systems.

Numerical integration in real-time is a challenging task. Explicit integration, such as explicit Runge-Kutta methods or explicit multistep methods provide integration schemes with a deterministic number of numerical operations, but they may fail for stiff systems due to stability problems. Choosing a rather small step size can help to overcome this issue, but the sample rate and the computational power of real-time platforms are (strongly) limited. Standard implicit methods like implicit Runge-Kutta methods or BDF methods are designed for stiff systems with an acceptable step size, but nonlinear systems of equations have to be solved in each time step. Linearly implicit one-step methods, in particular Rosenbrock methods, provide a compromise. They can solve stiff problems using larger steps than explicit methods at the cost of having to solve linear systems.

The paper describes the new contributions in the following order: Section 2 gives an overview of the new code generator, Section 3 explains implementation of the Rosenbrock methods, Section 4 gives realistic application examples, and finally Section 5 gives a summary and outlines possible future extensions.

2 Model-based Embedded Controller Development in Dymola

Controller code intended to be executed in real-time on embedded devices is subject to special requirements. For example, (*Bertsch et al., 2015*) discusses these challenges in the context of automotive embedded applications for the case of FMU source code

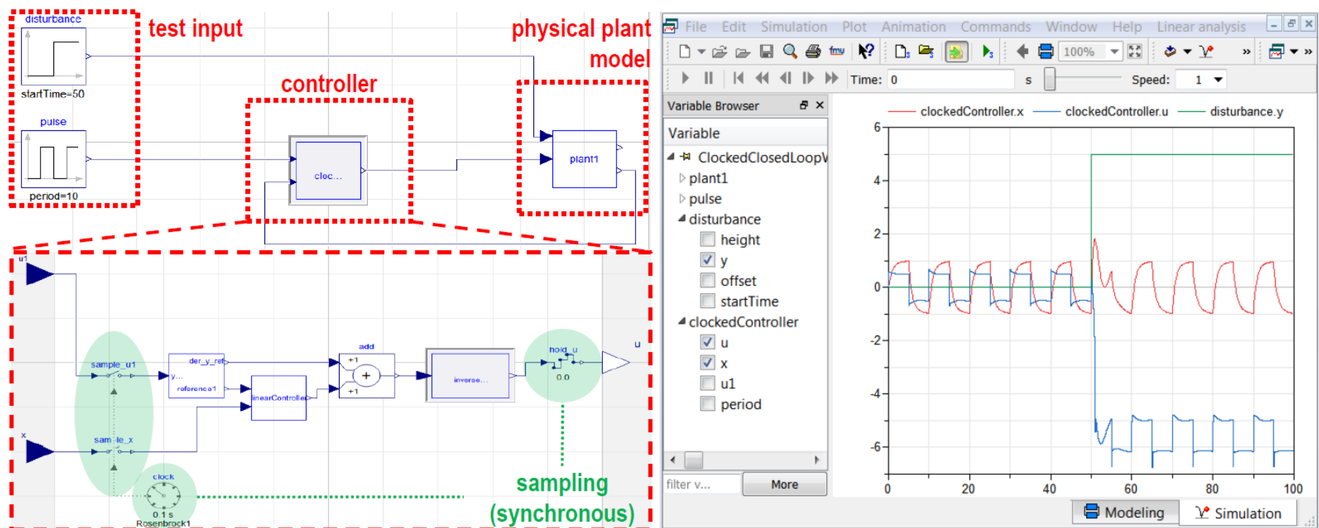

Figure 1. Embedded controller development scenario with Modelica/Dymola.

generation using Modelica tools. The standard C-code generated by Modelica tools is typically designed for desktop computer environments, where substantial hardware and software resources are available. Simulation is offline and without hard real-time constraints. Such standard code does not fulfill real-time system requirements, where code has to be deployed on embedded targets.

The standard C-code generated by Dymola from Modelica models is no exception; it is highly optimized to cope with several application scenarios including offline simulation and hardware-in-the-loop simulation of complex plant models on dedicated hardware platforms. However, this code includes many features not needed (*and* fails to fulfill constraints) for real-time controller code to be executed on embedded targets where minimalistic, self-contained, and human readable code is required.

On the other hand, Dymola provides convenient tooling for the development of full multi-domain system models and their simulation. It would be very convenient if embedded code for the controller parts also could be automatically generated and evaluated in software-in-the-loop simulations. The advantages of Modelica regarding complete system modeling and simulation are then leveraged also for real-time and embedded controller development.

Figure 1 summarizes the embedded development scenario we like to support. Physical plant models, controllers and test inputs for typical use cases can be fully modeled (left part) and simulated (right part) on system level. Throughout *iterative development* of *all* components, the *whole system* can be evaluated using standard simulation facilities. Embedded code can then be generated for the controller and co-simulated with the rest of the system. The results of such a software-in-the-loop co-simulation are shown on the right. The control signal (blue curve) is computed using the code generated by the embedded code generator. The red curve is the controlled plant output and the green signal is a disturbance that becomes active midway through the simulation. The embedded code generator to support this process is described below.

2.1 Embedded Development Process

Given a physical system model in Modelica, the experimental Dymola embedded code generator considers the following four tasks for the design and implementation of controllers for an embedded target:

(1) Controller modeling: Implement controllers as Modelica models with continuous model equation parts as done in Modelica since many years.

(2) Model decomposition: Use the controller models in a synchronous environment as described in (*Otter et al., 2012*). Sample and hold blocks are used to incorporate the controller inputs and outputs. As integration scheme for the clocked blocks the Rosenbrock methods presented in Section 3 can be used. In Modelica terms, controllers are therefore just synchronously clocked sub-models. Their synchronous clock models the interval in which the embedded environment provides new real-time inputs and queries for respective control actions.

(3) Embedded code generation: To generate the code to be embedded for the controller parts, apply the embedded code generator on the total model. Dymola extracts the synchronously clocked parts and generates C-code which is a self-contained, real-time simulator of its clocked parts. The code is well-suited for embedded deployment.

(4) Embedded deployment: Adapt, integrate and test the generated controller code on a real-time platform, like a rapid prototyping platform or embedded device.

The four tasks can be iteratively performed, in interrelation with the development of the model of the controlled physical systems. Co-simulation of the generated controllers is achieved by binding the generated C-code of controllers as external C functions to Modelica and calling them at every sample point

throughout the system simulation. Examples of this procedure are given in Section 4.

2.2 Properties of Generated Code

In addition to the standard optimizations performed by Dymola's symbolic manipulation facilities (equation systems are automatically torn to solve as much symbolically as possible, constant expressions are folded and shared expressions are eliminated to be computed at most once), the controller source code generated by the embedded code generator complies with the following requirements for execution on embedded devices:

Code Integration

- All types (model variables, states and records) relevant for user code and further code integration are encapsulated in header files.
- Proper C data types are deduced. Substitutions are performed to reduce memory footprint.
- A clear interface (with separate C-functions for initialization, output calculations, etc.) enables easy integration within external embedded environments.
- A generic interface to a solver for linear equation systems enables the usage of solvers tailored for specific applications and targets. The code for a default LU-solver is provided.
- The generated code is self-contained, without dependencies on further libraries (including the C standard library), supporting embedded devices without operating system or restricted software availability.

Traceability

- Comments link the generated code to its Modelica model, enabling traceability of computations and declarations. An XML file describing all variables is generated.

Real-Time Execution

- No heap memory allocation or recursion enables deterministic static memory allocation and therefore memory requirement predictions.
- The Rosenbrock integration methods described in Section 3 are applied to achieve deterministic execution times and enable predictable response times by preventing iterative loops with unknown number of iterations.
- Equations and variables are only considered when relevant for *controller outputs*; irrelevant computations are removed from the code.

There are also some restrictions on the generated embedded code:

- It does not (yet) fulfill all requirements of the MISRA-C standard (*MISRA, 2013*), which is important for safety-critical systems.
- Simplified event handling is applied. Only state events can occur, since the models do not use time directly.
- Nonlinear systems of equations to be solved in real-time are currently not directly supported. By using linearly implicit integration methods with an index-1 formulation these systems are automatically avoided, see Section 3.2.

2.3 A Simple PI-Controller Example

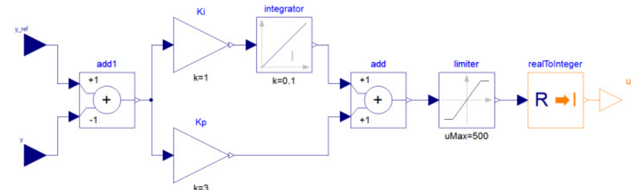

Figure 2. Simple PI-controller with output saturation and integer quantization.

Figure 2 shows a simple linear PI-controller. Since the synchronous model decomposition is only required to "mark" the controller for embedded code generation, but irrelevant for the actual embedded code generated, we can ignore the controller's clock and in- and output samplings. Relevant for embedded code generation is that the output u of the controller model is declared with *min* and *max* attributes defining its saturation:

```
Modelica.Blocks.Interfaces.IntegerOutput
    u (min = -500, max = 500) "Controller output";
```

The embedded code generator generates two C source code files: a header file defining the controller's in- and output types (*dsembedded.h*) and its actual implementation (*dsembedded.c*).

The header file in Figure 3 defines a C struct that holds all relevant model variables, each annotated with a comment referring to its original Modelica declaration and description. Note, that the type of the output u is a signed 16-bit integer which Dymola has deduced from the *min* and *max* attributes declared in the controller's Modelica model.

Figure 4 shows parts of the generated model implementation, in this case the start of the routine for the calculation of controller outputs. Each calculation is preceded by a comment that traces back to the original component, class and equation within the controller's Modelica model responsible for the code generated. The comments also contain information about alias substitutions and deduced array sizes.

Similar code is generated for model update used to provide new sampling inputs and dynamics to compute complex simulation steps. Using the code and the types provided by the generated header file, the generated controller implementation can be integrated in the embedded software system actually deployed on some target device. More advanced application examples combining Rosenbrock integration methods and this embedded code generator are given in Section 4.

```
/* dsembedded.h
 * Model variables for Modelica model PIController */
#ifndef _dsembedded_h_
#define _dsembedded_h_
#include <dse_types.h>
#include <dsembedded_structs.h>    /* structs for records */
#include <dsembedded_prototypes.h> /* function prototypes */

/* Model variables */
struct PIController_variables {
  /* input Modelica.Blocks.Interfaces.RealInput y
      "Measured variable" */
  real_t y;

  /* input Modelica.Blocks.Interfaces.RealInput y_ref
      "Reference signal" */
  real_t y_ref;

  /* output Modelica.Blocks.Interfaces.IntegerOutput
      u(min=-500, max=500) "Controller output" */
  integer16_t u;
  ...

  /* parameter Modelica.Blocks.Types.Init
      integrator.initType (min=1, max=4) =
       Modelica.Blocks.Types.Init.InitialState
        "Type of initialization (1: no init,
         2: steady state, 3,4: initial output)" */
  uinterger8_t integrator_initType;

  /* parameter Boolean limiter.strict = false
      "= true, if strict limits with noEvent(..)" */
  boolean_t limiter_strict;
  ...
};
```

Figure 3. C header file generated for the PI-controller.

```
/* dsembedded.c
 * Model equations for Modelica model PIController */
#include <dsembedded.h>
#include <dsembedded_codes.c>      /* functions code */

/* Model outputs */
static int model_outputs(PIController_variables* v,
    PIController_states* s)
{
  /* Component add1 */
  /* Class     Modelica.Blocks.Math.Add */
  /* y = k1*u1+k2*u2; */
  /* y = Ki.u; */
  /* u1 = y_ref; */
  /* u2 = y; */
  v->Ki_u = v->add1_k1*v->y_ref+v->add1_k2*v->y;

  /* Component Kp */
  /* Class     Modelica.Blocks.Math.Gain */
  /* y = k*u; */
  /* u = Ki.u; */
  v->Kp_y = v->Kp_k*v->Ki_u;

  /* Component add */
  /* Class     Modelica.Blocks.Math.Add */
  /* y = k1*u1+k2*u2; */
  /* u1 = integrator.y; */
  /* u2 = Kp.y; */
  v->add_y = v->add_k1*v->integrator_y+v->add_k2*v->Kp_y;
  ...
}
```

Figure 4. Generated PI-controller implementation.

3 Rosenbrock Methods

For real-time applications of stiff systems Dymola has historically reduced the model's equation system to index 0 (an ODE system) and used a nonlinear solver to handle the implicit Euler discretization by a limited number of Newton iterations, see e.g. (*Elmqvist, Mattsson et al., 2004*).

One main advantage of Rosenbrock methods is to directly solve stiff systems using only a *linear* solver. A certain variant of the implicit Euler method doing only one Newton iteration per step is equivalent to the corresponding Rosenbrock method of order 1.

In the following subsection Rosenbrock methods are introduced for index-1 DAEs which are known from the literature. Further, the advantages of the index-1 formulation and their application on Modelica models are presented. Finally, some properties and details of their implementation in Dymola are discussed.

3.1 Rosenbrock Methods for Index-1 DAEs

The supported Rosenbrock methods consider non-autonomous DAE systems with index 1 of the form

$$E\dot{y} = f(t, y)$$

where E is a constant and possibly singular matrix.

The Rosenbrock methods (*Hairer, Wanner, 1991*) are defined by s stages for a single step from t_0 to $t_1 \coloneqq t_0 + h$ with the initial state vector $y_0 = y(t_0)$ to get an approximation of the state vector $y(t_1)$:

$$y_1 = y_0 + \sum_{i=1}^{s} m_i u_i,$$

$$J_i = \frac{1}{h\gamma_{ii}} E - f_y(t_0, y_0),$$

$$J_i u_i = f(t_0 + \alpha_i h, y_0 + \sum_{j=1}^{i-1} a_{ij} u_j) + E \sum_{j=1}^{i-1} \frac{c_{ij}}{h} u_j + \gamma_i h f_t(t_0, y_0) \quad (i = 1, \dots, s).$$

Fixed method coefficients are γ_{ii}, a_{ij}, α_i, c_{ij}, γ_i and m_i. To compute the stage vectors u_i a linear system of equations has to be solved in each stage. Especially interesting are methods with $\gamma \coloneqq \gamma_{ii}$ $(i = 1, \dots, s)$, because then the iteration matrix J_i of the linear system is the same in each stage – and we can drop the index. So, only one decomposition of the iteration matrix J is required in each time step. Rosenbrock methods require the evaluation of the Jacobian f_y and the derivative with respect to time f_t.

For systems with input variables u (which must not be mixed up with the stage vectors u_i):

$$E\dot{y} = f(t, y) \coloneqq \varphi\big(t, y, u(t)\big)$$

this means $f_t = \varphi_t + \varphi_u \dot{u}$ with the derivatives $\dot{u}$ of the external input signal u to be provided.

There exist coefficients of Rosenbrock methods with convergence orders from 1 to 4 with different stability properties. In (*Lubich, Roche, 1990*) an L-stable Rosenbrock method of order 3 with $s = 4$ stages is developed for index-1 systems. In (*Rang, 2013*) the coefficients of Rosenbrock methods are improved to get methods without order reduction for (very) stiff problems.

Rosenbrock methods are interesting for real-time simulation of stiff systems, because the computational procedure for a step includes the solution of linear

systems but not of nonlinear systems. For linear systems a fixed number of computations guarantee finding the numerical solution in contrast to the iteration process for solving nonlinear systems.

3.2 Linear and Nonlinear Systems of Equations

In comparison to the ODE representation of a Modelica model, the index-1 formulation in Section 3.1 has some advantages in combination with Rosenbrock methods. Consider the example system of index 1

$$\dot{x} = f(t, x, y),$$
$$0 = g(t, x, y),$$

where a possibly nonlinear function g couples states x and algebraic variables y. The typical transformation to ODE form would lead to

$$\dot{x} = f\big(t, x, y(x, t)\big),$$
$$y = g^{-1}(x, t).$$

Here, maybe a nonlinear or at least a linear system of equations has to be solved when inverting the function g with respect to y. This can be avoided, if the index-1 formulation is used:

$$\begin{pmatrix} I & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} f(t, x, y) \\ g(t, x, y) \end{pmatrix}.$$

When applying a Rosenbrock method only the right hand side and its derivatives are evaluated. The one step method provides an approximation of the solution vectors x, y just by solving linear systems in the stages of the method. So, no nonlinear or nested linear system has to be solved. This property is very helpful for real-time simulation, because nonlinear loops in the original Modelica model can be replaced by linear systems in this way – leading to predictable computation times, see Section 4.1.2 for an example.

3.3 Rosenbrock Methods in Dymola

The support of Rosenbrock methods has recently been implemented in Dymola. The integration schemes rely on the index-1 formulation of the manipulated Modelica model equations. By the symbolic manipulation algorithms of Dymola, it is structurally guaranteed, that the system $E\dot{y} = f(t, y)$ has index 1. Currently, in Dymola four different Rosenbrock methods with orders 1-4 are available. The method of order 1 is the linearly implicit Euler method. All the methods are available as global inline integration methods in Dymola, see the menu in Figure 5.

Further, a Rosenbrock method can be specified as a solver method for a clocked part, by setting the argument *solverMethod* of the Modelica Clock constructor *Clock(c, solverMethod)*. This functionality is then used in the *Modelica_Synchronous* library to define the integration method of clocked equations. In the example in Figure 6 the first order Rosenbrock method “Rosenbrock1” is used.

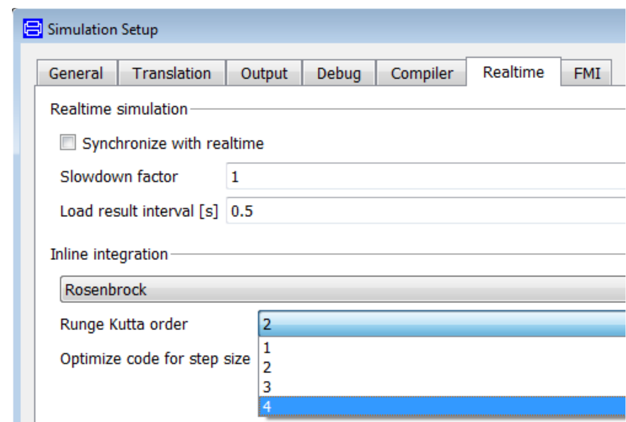

Figure 5. Menu to select a Rosenbrock integration method in Dymola.

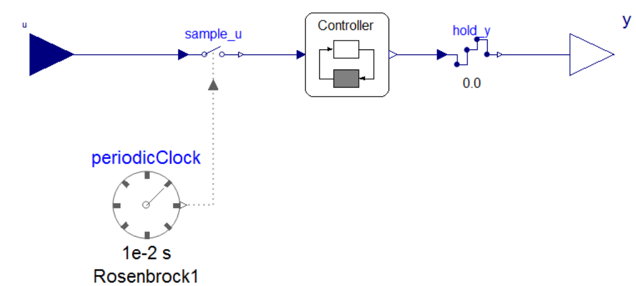

Figure 6. Inclusion of a continuous-time controller into a clocked environment using Rosenbrock integration.

To complete the tool chain, the Rosenbrock methods are also supported by Dymola’s embedded code generator. The symbolic machinery transforms the Modelica model equations after index reduction and fixed state selection to the form $E\dot{y} = f(t, y)$ and generates code for calculating E and f. Additionally the matrix f_y corresponding to the analytic Jacobian is straightforward to construct and generate code for.

It is more complicated to construct the vector f_t. The symbolic machinery normally deduces a total derivative with respect to time, but for Rosenbrock methods a partial derivative is needed. We will explain the difference with an example. If for example $f(t, y) = t^2 + y$, the total derivative with respect to time would be $d/dt\, f(t, y(t)) = 2t + \dot{y}$, but the partial derivative is $f_t(t, y) = 2t$. The symbolic machinery has also to deal with intermediate variables (e.g. z, if we rewrite the previous equation as $f(t, y) = z + y$; $z = t^2$; and the partial derivative with respect to time should differentiate those, but not the states).

Moreover, this time-derivative is not used by other standard numerical integration methods, and thus some Modelica functions do not provide the necessary derivatives (this can be handled either by assuming that the functions are smooth even if not specified or some minor modifications of libraries such as Modelica

Standard Library to specify this). This is especially the case, if the model follows some time-dependent trajectory $r(t)$ – because we need the derivative $\dot{r}(t)$, which is not needed for other methods. For input dependent models the derivative $\dot{u}$ of the input is involved in f_t (as explained in Section 3.1) and could be approximated by a difference quotient or the influence of $\dot{u}$ could be neglected in the method equations ($\dot{u} = 0$), when we assume that the input signal is piecewise constant. But this introduces some non-smoothness into the right hand side f, which could lead to numerical errors especially when applying Rosenbrock methods with orders greater than two. A more sophisticated solution for such input dependent models would require the additional input $\dot{u}$ for the model. This approach has not been investigated so far.

The matrix E is generally sparse or even diagonal with just zeros and ones on the diagonal and it would be worth to exploit the structure of the matrix when generating tailored code for the application of a Rosenbrock method to a specific Modelica model – but this is not yet realized in the implementation.

Dymola's implementation of Rosenbrock is generic, and some method-specific optimizations are not yet included, e.g. some Rosenbrock methods have several rows of the matrix (a_{ij}) that are identical, and in those cases we could avoid re-evaluating the right hand side f. This can intuitively be explained as performing exactly two iterations of the nonlinear solver for that point.

There are variations of Rosenbrock methods (W-methods) that keep the factorized matrices for several steps. We have not considered them for real-time applications. The reason is that for real-time code the goal is to ensure a maximum computation time for each sampling point – not for the average one; and we will anyway need new factorized matrices after each event. If we do not explicitly detect events, the problem with W-methods would be more severe since the continuity assumptions are silently broken.

4 Application Examples

In this section two application examples are given to demonstrate how the embedded code generation and the Rosenbrock methods can be used to generate real-time code for nonlinear controller structures with guaranteed upper number of operations.

4.1 Nonlinear Feedforward Controllers

We consider a continuous-time controller with two structural degrees of freedom and an inverse plant model in the feedforward path. See (*Looye et al., 2005*) for details on this controller structure and its implementation in Modelica. In case the inverse plant and the plant model are identical, they start at the same initial values and the plant is stable, then the control error is equal to zero, so the plant output follows the filtered reference input. The feedback controller is used to compensate for differences in the plant and inverse plant model, as well as for external disturbances, and it stabilizes a plant in case it is unstable.

4.1.1 Implementation in Modelica

In Modelica an inverse plant model can be constructed by using the model component

Modelica.Blocks.Math.InverseBlockConstraints

to exchange inputs and outputs and by connecting a filter to the input of the inverse model. As filter the model *Modelica.Blocks.Continuous.Filter* with parameters *filterType* = *LowPass* and *analogFilter* = *CriticalDamping* or *Bessel* can be used, see Figure 7. The minimum order of the filter results from the structural analysis of the inverse plant model resp. the corresponding DAE in order to only provide the input u but not derivatives of it. The derivatives of the smoothed input signal are computed inside the filter model.

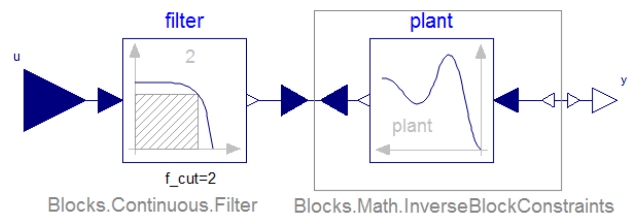

Figure 7. Definition of an inverse plant model.

In order that the controller can be used on a real time system, the process of Section 2 is applied. The continuous controller model is transformed to a clocked system with sample and hold blocks and an appropriate inline integrator needs to be selected for the clock. In simple cases, an *Explicit Euler* method might be enough. If the controller contains nonlinear algebraic equations or if the model is stiff, a *Rosenbrock* integrator has to be selected, see also Section 3.3. Note, that the filter might be stiff even if the inverse plant model might be non-stiff.

It follows an application example for the automatic construction of nonlinear feedforward controllers that can be used in an embedded system.

4.1.2 Example 1: Slider Crank Mechanism with Feedforward Controller

The following example is a slider-crank mechanism that is directed in vertical direction. At the top a spring-mass system is present. The goal is to move the revolute joint of the slider-crank mechanism, such that the mass follows a pre-defined path without vibrations.

When kinematically driving the revolute joint with constant velocity, then the vertical coordinate of the top mass moves as shown

DOI 10.3384/ecp17132517

in Figure 8. As can be seen, significant vibrations are present in the movement of the mass. The goal is to develop an embedded controller according to Section 2. The problem is rather challenging, because the slider crank mechanism introduces a nonlinear algebraic equation system in the plant, as well as in the plant inverse.

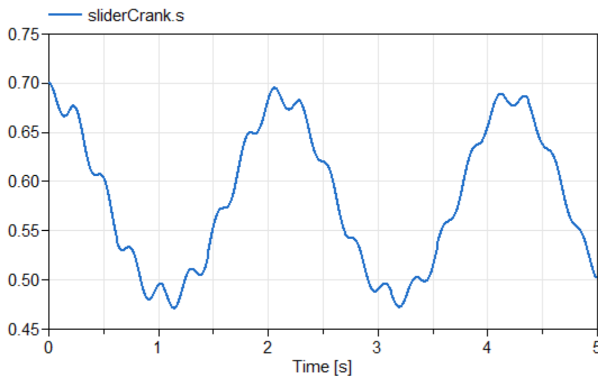

Figure 8. Vertical movement of top-mass of the slider-crank mechanism.

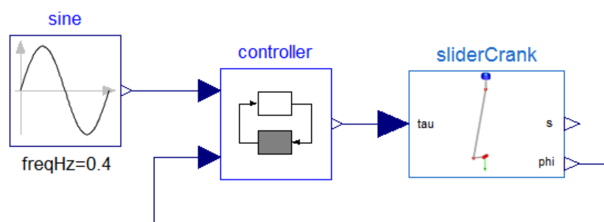

Figure 9. Controlled slider crank mechanism.

In Figure 9 the overall system including a controller is shown. The controller is detailed in Figure 10 where for the feedforward path of the controller an inverse model of the slider crank mechanism is present such that the input of the inverse model is the vertical position *s* of the top mass, and the outputs are (a) the reference torque *tau* for the revolute joint, and (b) the reference angle *phi* for the revolute joint. As filter a third order critical damping filter is used. The control error is the difference between the reference angle phi computed by the inverse slider crank model and the measured angle phi from the plant. A simple P controller is used in the feedback loop.

Although a nonlinear system of equations appears in the model equations of the inverse slider-crank model, it is possible to generate embedded code for the sampled data controller according to Section 2 by using the newly supported Rosenbrock integrators of Section 3. The detailed explanation of this effect is found in Section 3.2. A proper step size of the tested Rosenbrock methods for the controller is 1 ms.

Some simulation results are shown in Figure 11. The Rosenbrock method of order 1 (the linearly implicit Euler method) leads to very accurate results with respect to the reference solution generated by a highly accurate DASSL simulation. The numerical solution of the Rosenbrock method with order 3 is also rather accurate, only in the torque signal some vibrations are visible. One reason could be neglecting the input derivatives of *s_ref* and *phi* in the integration scheme of Rosenbrock methods as described in Section 3.1.

Experiments show for the Explicit Euler method a maximum step size of 0.5 ms can be used; otherwise the numerical integration cannot be run due to difficulties with solving the nonlinear system. There remains still a nonlinear system of equations due to the index-0 formulation of the translated model equations. This also means that currently no embedded code can be generated for this controller example when using an Explicit Euler method as integrator.

There are two advantages of Rosenbrock methods: Because they are implicit methods, generally greater step sizes can be used than for explicit methods and nonlinear system of equations present in the model equations can be approximately solved by a Rosenbrock solver resulting in only linear systems.

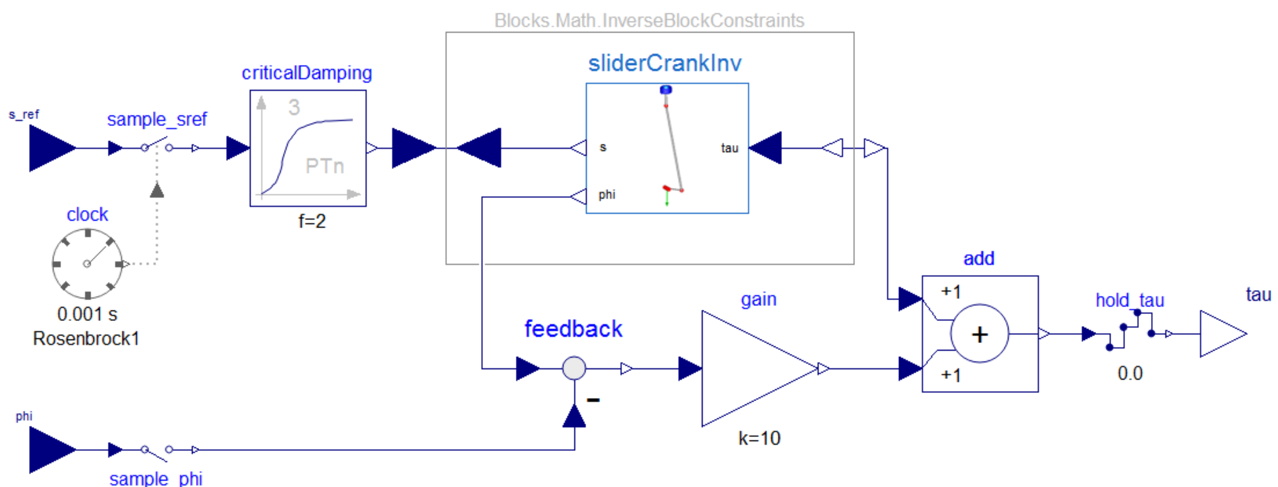

Figure 10. Sampled data controller of a slider crank mechanism consisting of inverse plant model in the feedforward path, a filter of order 3 and a P controller in the feedback loop.

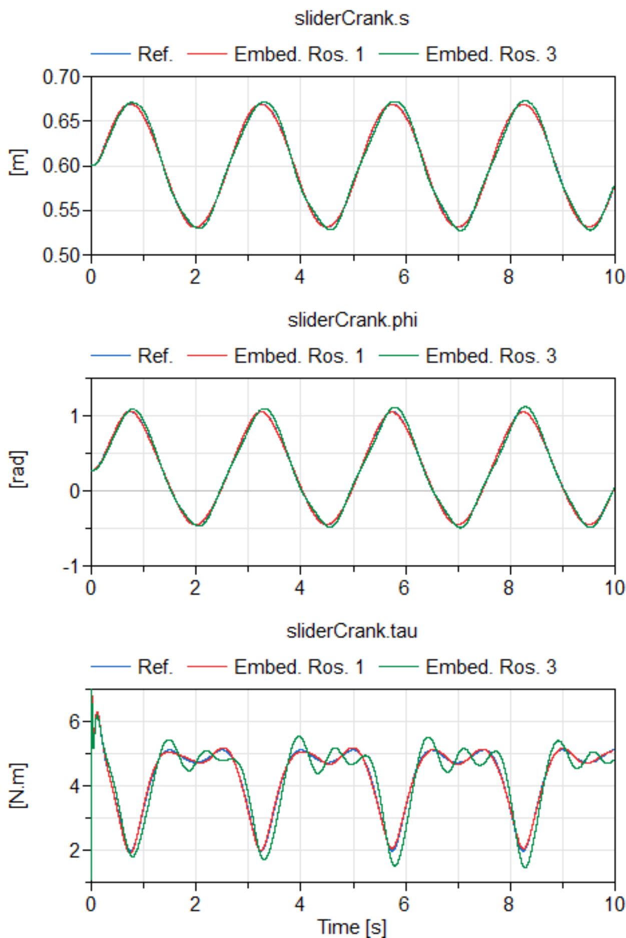

Figure 11. Simulation results of the slider crank mechanism with a nonlinear feedforward path and a P controller in the feedback loop. The reference solution using the continuous controller in Modelica is generated by highly accurate BDF-methods with DASSL (blue lines) whereas the embedded controllers contain Rosenbrock methods of order 1 (red lines) and order 3 (green lines) with a constant step size of 1 ms.

4.2 Feedback Linearization Controllers

A further important controller structure using nonlinear plant models is feedback linearization, see (*Looye et al., 2005*) for more details including the implementation in Modelica.

4.2.1 Implementation in Modelica

The first part of this subsection is a summary of material provided in (*Looye et al., 2005*). The principal differences between a controller with feedback linearization and a feedforward controller of Section 4.1 are that for the feedback linearization

- the inverse plant model is in the feedback part of the controller and
- the states in the inverse model are obtained from the actual plant via measurement and/or estimation and not via solving a DAE (but algebraic equations might need to be solved).

When deriving feedback linearizing control laws manually, the outputs to be controlled are differentiated until an analytical relation with a control input is found. If the system model is available in Modelica, the derivation of the control laws can be automated using a similar procedure as described in Section 4.1.1. However, instead of a filter with a minimal order, a minimal set of integrators is added:

$$v := y^{(p)} := \frac{d^p}{dt^p} y \tag{1}$$

where v is the new model input corresponding to the output with relative degree p. We describe the procedure for a single output system with the controlled output y. The desired dynamic behavior of the closed-loop system is then imposed by application of an additional feedback law:

$$v = k_0(y_{Ref} - y) - \sum_{i=1}^{p-1} k_i \, y^{(i)} \tag{2}$$

with constant coefficients k_i. This feedback law requires availability of the $(p-1)$-th derivative of the controlled output y. The derivatives may be obtained from measurements or from the computed values in the inverse model. In case the inverted model exactly represents the true system, the closed loop system becomes the single output case:

$$y^{(p)} + \sum_{i=1}^{p-1} k_i \, y^{(i)} + k_0(y - y_{Ref}) = 0. \tag{3}$$

A disadvantage of feedback linearization is that the state vector of the plant must be fully available from measurement and/or estimation.

In Modelica, the inverse model is built in a similar way as for a feedforward controller, see Figure 12:

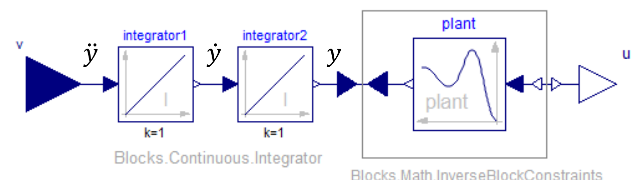

Figure 12. Definition of an inverse plant model for feedback linearization.

When translating this model with Dymola, typically an error message of the following kind is displayed: "*The model requires derivatives of some inputs as listed below: ...*". For example if derivatives of order 2 are required by v, then 2 more integrators have to be added.

This inverse model with the integrators of Figure 12 is placed in the feedback-loop of the controller. If the minimal number of integrators are added, then the model from $v \rightarrow y$ has the same number of states as the non-inverted plant. These states must be provided from measured and/or estimated values of the plant. To formulate this, Dymola has introduced an annotation to map a sampled input signal, say, *xs = sample(xc)* to a state *x*, say:

 DOI 10.3384/ecp17132517

```
Real xs annotation(useAsInputForState=x);
```

The meaning is that

1. *StateSelect.always* is defined for variable *x*.
2. After the usual index reduction and state selection *x* is deselected as state and the equation *x* = *xs* is added. Its derivative is set as a dummy derivative.

As a result, the inverse model is no longer a differential-algebraic equation system, but only an algebraic equation system. In case this system is nonlinear, the Rosenbrock method from Section 3 is used to solve it during run-time with a fixed upper bound on the number of operations.

Figure 13. Modelica block to apply the new annotation *useAsInputForState*.

A corresponding Modelica block has been implemented to use the annotation, see Figure 13. The main line of code in the block is

```
RealInput xs annotation (useAsInputForState=x);
```

to enforce, that the state *x* is set to the input *xs* according to the above logic.

In the next subsection an application example demonstrates the general tool chain for the automatic construction of feedback linearization controllers that can be used in an embedded system.

4.2.2 Example 2: Mixing Reactor with Feedback Linearization Controller

We use a mixing reactor model that is explained in detail in (*Looye et al., 2005*) – including different types of controllers for it. The reactor shall be controlled by a feedback linearization controller. For the feedback linearization it is assumed that the two system states, the concentration $c =: y$ of the chemical substance, as well as its temperature T, are measurable.

With the approach of Figure 12 it is determined that two integrators are needed. By Equation (3) we get the following feedback law:

$$v = \ddot{c} = k_0(c_{ref} - c) - k_1\dot{c} \qquad (4)$$

whereby c is available from measurement and $\dot{c}$ is computed from the inverse model (which in turn means that it is computed from the measured c and T). The following feedback coefficients are selected:

$$k_0 = 4.39\text{e-}4, \qquad k_1 = 0.0419.$$

The complete model including the controller and the controlled plant is shown in Figure 14. The controller is according to (4) and includes the “input to state” blocks for the states of the inverse plant model: the concentration c and the temperature T. As explained in the previous subsection this is necessary to provide the measured states to the inverse model of the feedback linearization controller.

Figure 15 shows the response of the closed loop system using the embedded controller generated according to the process described in Section 2. The simulation results are generated with Rosenbrock methods of order 1-4. It is obvious that the results are rather identical for the used step size of 5 s. It is also possible to use the Explicit Euler method, but then a step size of 1 s is necessary to achieve similar accuracy as the Rosenbrock methods do.

The main novelty of this example is the support of the newly introduced annotation *useAsInputForState* within a clocked system. This feature enables the user to develop and test a controller with feedback linearization in a purely Modelica environment. Previously (see *Looye et al., 2005*) this was only possible by exporting the inverse model to for example Simulink and building the controller in this environment. The additional support by the embedded code generator completes this feature.

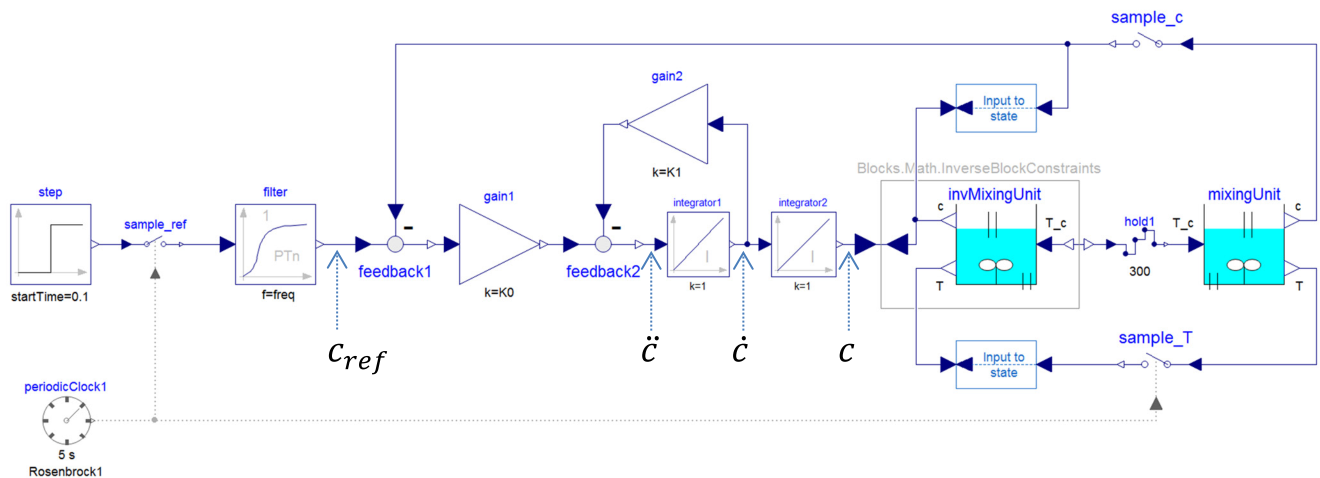

Figure 14. Mixing reactor controlled by a feedback linearizing controller. The “input to state” blocks are used to provide the states of the inverse model.

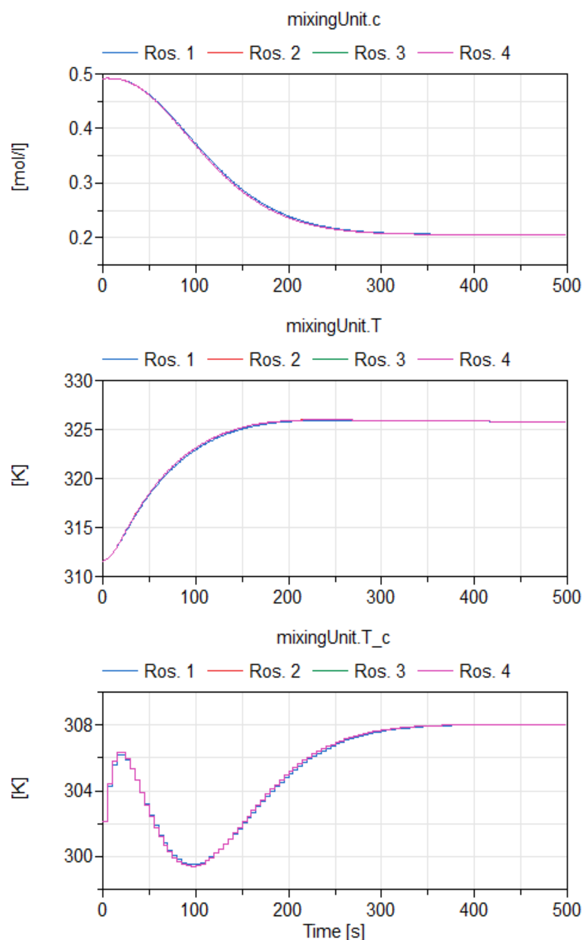

Figure 15. Step response of the mixing reactor controlled by a feedback linearization controller.

5 Summary and Outlook

Directly generating controller code from models is important for model-based design. This paper demonstrates how Dymola can generate embedded C-code for Modelica models (with several novel aspects) and demonstrates this for application examples.

By using Rosenbrock methods on the index-1 problem Dymola can handle stiff systems in a way that both is theoretically sound and has an upper bound on the execution time per sample. The stiff systems in the control system often occur due to using an inverse (simplified) model of the real plant in the controller.

Additionally Dymola's generated embedded C-code has been designed to be easy to embed, with care about minimal foot-print, traceability, and straightforward integration in embedded platforms.

Finally, a nonlinear feedforward controller and a feedback linearization controller have been implemented in Modelica for different application examples. They show the potential of the whole process by generating embedded real-time code for these nontrivial examples.

Future considerations include investigating how to handle inputs (piece-wise constant or extrapolation) for Rosenbrock methods, and nonlinear systems in initialization and event code. Additionally, the choice of the specific Rosenbrock methods will be re-investigated. A possible future extension would be to use Rosenbrock methods with step-size control in off-line mode; one benefit would be to quickly get an estimate of the step-size needed for the model.

6 Acknowledgment

We would like to thank Gertjan Looye from DLR for his help regarding modeling of feedback linearizing controllers.

References

C. Bertsch, J. Neudorfer, E. Ahle, S. S. Arumugham, K. Ramachandran, A. Thuy. FMI for Physical Models on Automotive Embedded Targets. *Proc. of 11th International Modelica Conference*, pp. 43-50. Versailles, France, 2015.

H. Elmqvist, S. E. Mattsson, H. Olsson, J. Andreasson, M. Otter, C. Schweiger, D. Brück. Realtime Simulation of Detailed Vehicle and Powertrain Dynamics. *Electronics Simulation and Optimization.* SAE 2004 World Congress, Detroit, USA, 2004.

E. Hairer, G. Wanner. *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems.* Springer, 1991.

G. Looye, M. Thümmel, M. Kurze, M. Otter, J. Bals. Nonlinear Inverse Models for Control. *Proceedings of 4th International Modelica Conference,* pp. 267-279, TU Hamburg-Harburg, Gemany, 2005.

C. Lubich, M. Roche. Rosenbrock Methods for Differential-algebraic Systems with Solution-dependent Singular Matrix Multiplying the Derivative. *Computing* 43, 325-342, Springer, 1990.

MISRA Consortium. *Guidelines for the Use of the C Language in Critical Systems*. 2013.

M. Otter, B. Thiele, H. Elmqvist. A Library for Synchronous Control Systems in Modelica. *Proc. of 9th International Modelica Conference*, pp. 27-36. Munich, Germany, 2012.

J. Rang. *Improved traditional Rosenbrock Wanner methods for stiff odes and daes*. Technical report, Institute of Scientific Computing, Technical University Braunschweig, 2013.

L. Satabin, J.-L. Colaco, O. Andrieu, B. Pagano. Towards a Formalized Modelica Subset. *Proc. of 11th International Modelica Conference,* pp. 637-646. Versailles, France, 2015.

SCADE: www.esterel-technologies.com/products/scade-suite

DOI 10.3384/ecp17132517