Modular Multi-Rate and Multi-Method Real-Time Simulation

Bernhard Thiele[†] Martin Otter[†] Sven Erik Mattsson[‡]
Bernhard.Thiele@dlr.de Martin.Otter@dlr.de SvenErik.Mattsson@3ds.com
[†] German Aerospace Center (DLR), Institute for System Dynamics and Control, 82234 Wessling, Germany
[‡] Dassault Systèmes AB, Ideon Science Park, SE-223 70 Lund, Sweden

Abstract

The demand to ever increase realism and scope of models routinely exceeds the currently available computing power and thus requires thoughts on improving simulation efficiency. This is especially true for real-time simulations, where fixed timing constraints do not allow to just "wait a bit longer".

This paper presents a new approach in Modelica that allows a modeler to separate a model into different partitions for which individual solvers can be assigned. In effect, this allows to use multi-rate and multi-method time integration schemes that can contribute to improve the efficiency of a (real-time) simulation.

The first part of the paper discusses basic consideration relating to modular (real-)time integration. Afterwards, the implementation of a convenient Modelica library for the partitioning of physical models is briefly described. Finally, the presented library is used to partition a detailed six degree of freedom robot model for modular simulation. The simulation performance of that partitioned model is compared to the simulation performance achieved by using "conventional" global solvers.

Keywords: multi-rate / multi-method time integration; simulation; clocked discretized continuous-time partitions.

1 Introduction

Testing the actual embedded systems hardware in processor-in-the-loop (PIL) or hardware-in-the-loop (HIL) setups, usually requires that the "virtual" parts of the overall systems are simulated under real-time constraints. This means that the simulation must run as a hard real-time application that always meets its timing deadlines.

Inputs and outputs of a real-time simulation need to be processed at regular intervals. The length of this interval is called the *simulation frame time*. The *worst case* computation time needs to be less than the simulation frame time. Explicit fixed-step solvers are appropriate numerical integration routines for real-time simulations. Variable-step solvers are generally not appropriate for two reasons: a) because real-time simulation must normally perform I/O operations at regular intervals, and b) because the flexible number of performed simulation model evaluations impede deterministic prediction of worst case computations times.

Implicit fixed-step solvers are problematic in the context of real-time simulation, because the number of iterations required by implicit methods is theoretically unbounded. However, their numerical properties with regard to integrating *stiff systems*[1] is much more favorable compared to explicit solver methods. When integrating stiff systems using explicit methods, the largest possible step size is severely limited due to stability problems of the integration algorithms. Implicit methods can perform much better in such cases. Because of that, attempts have been made to use implicit methods also in real-time simulation. Elmqvist et al. [5, 4] describe techniques that allow to minimize the number of iteration variables for implicit methods (in some cases the number of iteration variables can be reduced to zero!). These algorithms are available in the commercial tool Dymola for real-time simulation purposes under the umbrella term *inline integration* algorithms. The development of linearly implicit or semi-implicit methods are another noticeable attempt in which implicit solver methods are "approximated" by methods that exhibit a bounded number of worst-case iterations. This improves the suitability of these methods for real-time simulation purposes

Computational resources are finite. As a consequence the computational requirements of the real-time simulation must accommodate with available

[1] Stiff systems typically contain dynamically fast and highly damped components.

hardware resources. If the computational load required for the simulation is too high, the simulation model needs to be adapted in order to meet the timing deadlines. Typical options for improving the computational performance include model simplification, use of more efficient algorithms and operations, or replacing computational intensive subsystems with fast table lookups. Another option is to split the simulation model into subsystems, which can be executed in parallel across multiple processors, or which can be executed with different frame rates. The later option is termed *multiframing* or *multi-rate integration* and can be attractive if some subsystems have significantly longer time constants than others. Details about typical techniques used in the context of real-time simulation can be found in relevant literature, e.g., [7, 3].

The new synchronous language elements extension to Modelica [8, Chapter 16] also provides language primitives that allow the developer to partition models into several parts that can be solved separately by different numerical solver methods. A crucial aspect of the partitioning task is to establish adequate *coupling* mechanism between the separate partitions. Following [6] the term *modular simulation* is used to underline that modular coupling approach.

During the partitioning the developer can take advantage of a-priori system knowledge to improve the performance of the simulation. Partitions can be executed with different frame rates and/or can utilize different numerical integration algorithms. In that way *multi-rate* and *multi-method* integration schemes can be realized. To the knowledge of the authors, this is a rather unique feature in a modeling language for physical systems. However, the suitable preparation of simulation models for multi-rate and multi-method integration is still a non trivial task.

In order to facilitate the preparation of simulation models for multi-rate and multi-method integration schemes a Modelica library named MULTIRATE has been build that wraps necessary methodological and technical knowledge in an easy-to-use framework. The theory behind this library, as well as its technical implementation and application to practical problems will be demonstrated in the following sections.

2 Clocked Discretized Continuous-Time

Using the synchronous language elements extension [8, Chapter 16] it is possible to define clocks that associate a continuous-time solver to the equations associated to that clock. This is illustrated in Figure 1. Line 4 defines a periodic clock with the in-

```
model ClockedDiscretizedContinuousTime       1
  Real x(start=0), u;                         2
  // 500 ms, no solver:                       3
  Clock clk = Clock(5,10);                    4
  // 500 ms, ExplicitEuler solver:            5
  Clock clk_solver=                           6
    Clock(clk, "ExplicitEuler");              7
equation                                      8
  // associate clock clk_solver to u:         9
  u = sample(1.0, clk_solver);                10
  der(x) = -x + u;                            11
end ClockedDiscretizedContinuousTime;         12
```

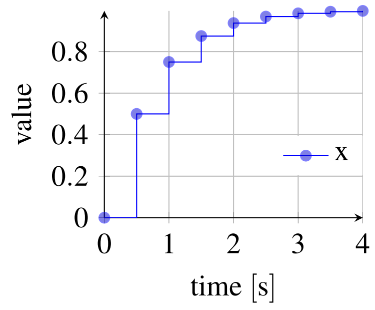


Figure 1: Clocked discretized continuous-time example.

terval 500 ms. Line 7 defines a *solver clock* based on the previously defined clock and assigns as solver method "ExplicitEuler". Other methods that are predefined by the language standard are discussed below. Line 10 associates clock clk_solver with the variable u (by "sampling" the literal constant 1.0 using clock clk_solver). Due to clock inference the differential equation in line 11 can be deduced to be also associated with clk_solver and therefore the differential equation needs to be solved by the "ExplicitEuler" method with a fixed-step integration step size of 500 ms. The corresponding plot of x is shown at the right-hand side of Figure 1.

The specification [8, Section 16.8.2] defines the conceptual solution algorithms of the predefined methods (tools may provide support for additional solver methods). Since the following discussion is based on these algorithms the respective part from the specifications is reproduced below in a slightly adapted form[2].

The solvers are defined with respect to the underlying ordinary differential equation in state space form to which the

[2]Most notably, in contrast to the specification text the solver methods are defined in terms of integrating from clock tick t_i to t_{i+1}, instead of from t_{i-1} to t_i. This is just a simple index shift. The advantage is, that it allows to present some equations in a slightly more concise and readable form.

Table 1: Predefined solver methods for solver clocks

Solver Method	*Solution method* (for all methods: $y_i := g(x_i, u_i, t_i)$)
`Explicit-Euler`	$x_{i+1} := x_i + h \cdot \dot{x}_i$ $\dot{x}_i := f(x_i, u_i, t_i)$
`ExplicitMid-Point2`	$x_{i+1} := x_i + h \cdot f(x_i + \frac{1}{2}h \cdot \dot{x}_i, \frac{u_i + u_{i+1}}{2}, t_i + \frac{1}{2}h)$ $\dot{x}_i := f(x_i, u_i, t_i)$
`Explicit-Runge-Kutta4`	$k_1 := h \cdot \dot{x}_i$ $k_2 := h \cdot f(x_i + \frac{1}{2}k_1, \frac{u_i + u_{i+1}}{2}, t_i + \frac{1}{2}h)$ $k_3 := h \cdot f(x_i + \frac{1}{2}k_2, \frac{u_i + u_{i+1}}{2}, t_i + \frac{1}{2}h)$ $k_4 := h \cdot f(x_i + k_3, u_{i+1}, t_{i+1})$ $x_{i+1} := x_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$ $\dot{x}_i := f(x_i, u_i, t_i)$
`Implicit-Euler`	$x_{i+1} := x_i + h \cdot \dot{x}_{i+1}$ † $\dot{x}_i := f(x_i, u_i, t_i)$
`Implicit-Trapezoid`	$x_{i+1} := x_i + \frac{1}{2}h \cdot (\dot{x}_i + \dot{x}_{i+1})$ † $\dot{x}_i := f(x_i, u_i, t_i)$

† Equation system with unknowns: x_{i+1}, $\dot{x}_{i+1}$.

continuous-time partition can be transformed, at least conceptually (t is time, $u(t)$ is the real vector of input variables to the partition, $x(t)$ is the real vector of continuous-time states, and $y(t)$ is the real vector of algebraic and/or output variables to other partitions):

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

The solver methods (with exception of `"External"`[3]) are defined by integrating from clock tick t_i to clock tick t_{i+1} and computing the desired variables at t_{i+1}, with $h = t_{i+1} - t_i = \mathrm{interval}(u_{i+1})$ and $x_{i+1} = x(t_{i+1})$.

Table 1 shows the definitions of the predefined solver methods using the notation from above.

3 Multi-Rate

Multi-rate integration can be attractive if some subsystems have significantly longer time constants than others. This is often the case for multi-domain physical systems since the components of different physical domains often exhibit significant different time constants. A typical example are systems with slow mechanical parts which are controlled by fast electrical circuits. If an explicit integration method is used, the numerical stability of the whole system depends on the fastest time constant and it is necessary to choose a respective small step size for integration.

The MULTIRATE library doesn't impose limits on the number of partitions with different frame rates that may be executed together. However, for clarity the basic idea of multi-rate integration is demonstrated with two ODE partitions[4] that will be discretized for two different execution rates:

$$\dot{x}_f(t) = f_f(x_f, x_s, t) \quad (1a)$$
$$\dot{x}_s(t) = f_s(x_f, x_s, t) \quad (1b)$$

The sub-index f stands for the "fast" partition and s stands for the "slow" partition. Using the `ExplicitEuler` method from Table 1 for discretization results in a system of recurrence equations of the form:

$$\begin{aligned} x_f(t_i + j \cdot h_f) = {} & x_f(t_i + (j-1) \cdot h_f) \\ & + h_f \cdot f_f\big(x_f(t_i + (j-1) \cdot h_f), \\ & x_s(t_i + (j-1) \cdot h), t_i + (j-1) \cdot h\big) \quad (2a) \end{aligned}$$
$$x_s(t_{i+1}) = x_s(t_i) + k \cdot h_f \cdot f_s\big(x_f(t_i), x_s(t_i), t_i\big) \quad (2b)$$

where k and j are integers, k is the ratio of the two step sizes, $j = 1 \ldots k$, h_f is the step-size of the fast partition, and $t_{i+1} - t_i = k \cdot h_f = h_s$ is the step-size of the slow partition.

Note that Equation (2a) does not specify how $x_s(t_i + (j-1) \cdot h)$ is calculated. Equation (2b) is not defined for intermediate values between t_i and t_{i+1}. Therefore, an interpolation or extrapolation scheme needs to be used to estimate the intermediate values.

For real-time simulation the sequence in which the computation of fast and slow frames are interspersed is important. Ledin [7] describes three typical execution schemes. The timing diagram in Figure 2 shows that execution schemes by means of an example where the ratio of the slow and the fast step size, k, is $k = 3$. The three schemes are described briefly below:

1. *Multiframing in a single task with no fast-frame real-time I/O.* The slow frame rate is treated as a "master" frame rate in which the slow frame is executed first, followed by a burst of the k fast frames. This scheme is only acceptable if the fast frames do not perform any real-time I/O.

[3] The solver method `"External"` means that the solution method is defined in the simulation environment and not in the Modelica model.

[4] In general, partitions in the MULTIRATE library may consist of differential and algebraic equations and the partitions are coupled over designated input and output variables, but this is omitted here in favor of a more succinct presentation of the basic idea of multi-rate integration.

2. *Multiframing in a single task with fast-frame real-time I/O*. The fast frames are executed at fixed intervals of h_f length. The computations needed in the slow frame are split into several subframes which are interspersed after the fast frame calculations. However, splitting the slow frame into several suitable subframes is rarely a simple thing to do. This is a serious drawback of this method.

3. *Multiframing in a multitasking environment with rate monotonic scheduling* (RMS). In this case the scheduler will give CPU access to the task with the higher priority (computation of fast frames) and interrupt the lower priority task (computation of slow frames) until the higher priority task has finished its computations. During the times in which the higher priority task is idle, the CPU access is given back to the lower priority task to resume its computations. No (manual) splitting into subframes is needed which is a huge advantage compared to the previous method.

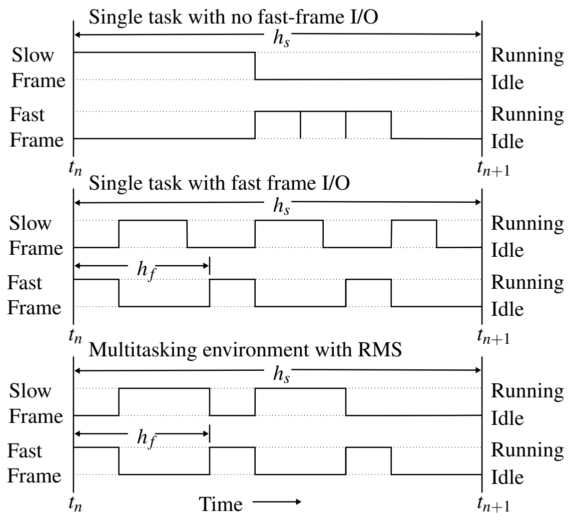


Figure 2: Three different multiframing schemes for real-time simulation.

4 Multi-Method

Multi-method integration (also called *mixed-mode integration*) is yet another option to improve the execution performance of (real-time) simulation. In contrast to multi-rate integration that uses different integration step sizes for distinct partitions, multi-method integration uses different integration methods for distinct partitions. Similarly to multi-rate integration, multi-method integration can be attractive if some partitions have significantly longer time constants than others.

A typical scenario is to split a system into fast parts and slow parts and use an implicit integration method for the fast parts and an explicit integration method for the slow parts. Schiela and Olsson [11] describe such a mixed-mode integration scheme (using explicit and implicit Euler methods) in which they employ an automatic partitioning approach based on linearization and eigenvalue analysis. This disburdens the developer from partitioning the system. However, if a system is highly nonlinear, inspecting eigenvalues becomes questionable since the eigenvalues of the linearized system move around with time. A user controlled partitioning, leveraging a-priori system knowledge, can be more adequate and effective in such cases.

Similar to multi-rate integration, the basic idea of multi-method integration is demonstrated on the basis of two ODE partitions in the form of equation system (1). Using the `ImplicitEuler` method from Table 1 for the “fast” partition and the `ExplicitEuler` method for the “slow” partition results in recurrence equations of the form:

$$x_s(t_{i+1}) = x_s(t_i) + h \cdot f_s\big(x_f(t_i), x_s(t_i), t_i\big) \quad (3a)$$
$$x_f(t_{i+1}) = x_f(t_i) + h \cdot f_f\big(x_f(t_{i+1}), x_s(t_{i+1}), t_{i+1}\big) \quad (3b)$$

where $h = t_{i+1} - t_i$ is the integration step-size. At first, $x_s(t_{i+1})$ is computed using the explicit Euler method. This value is afterwards used to compute $x_f(t_{i+1})$ using the implicit Euler method.

The MULTIRATE library allows to combine any of the solver methods listed in Table 1. Note that it is easily possible to combine multi-rate and multi-method integration within the framework of the MULTIRATE library. This allows to exceed the benefits compared to applying the methods separately.

5 Partition Coupling

An important aspect when applying multi-rate and multi-method integration is the coupling between the involved partitions. In the context of the framework provided by the MULTIRATE library the coupling scheme of Figure 3 depicts the basic idea (liberally abstracting from the details). Note that although the coupling is discussed by considering the special case of two coupled partitions, the basic principles carry over to the general case of n partitions.

Partition 1 and 2 may be discretized by using one of the solver methods defined in Table 1. Different solver

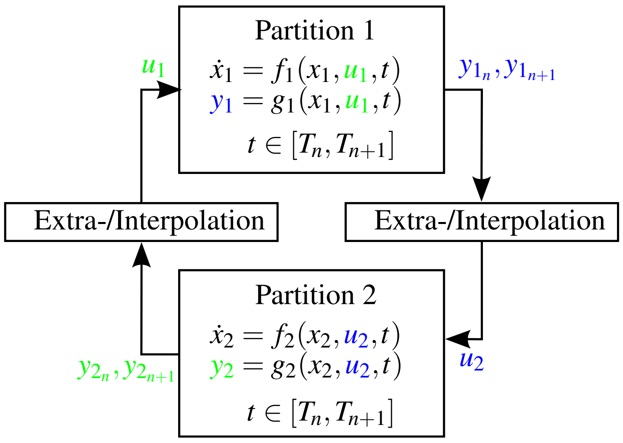


Figure 3: Coupling of two partitions. Communication takes place at discrete time instants $t = T_0, T_1, \ldots, T_k$. Depending on the applied coupling scheme the approximation of the coupling terms u_1, u_2 can be based on interpolation (if $y_{1_{n+1}}$, or respectively $y_{2_{n+1}}$ is available) or it must be based on extrapolation.

methods, as well as different step sizes can be used as long as the ratio of the step sizes is an integer.

Data exchange between clocked discretized continuous-time partitions is only possible at clock ticks. Therefore, the clock ticks of the slower partition determine the discrete time grid in which data is exchanged between the two partitions. This is quite similar to the situation encountered in co-simulation scenarios, in which the communication and data exchange between two distinct systems is restricted to discrete synchronization points T_n.

However, in contrast to co-simulation the modular integration considered in the MULTIRATE framework has some distinctive characteristics:

- It inherits the characteristics of the synchronous model of computation that has been introduced in the "Synchronous Language Elements" extension in Modelica 3.3 [8, Chapter 16]. This has the advantage that the formal model of various coupling schemes can be expressed in a high-level, declarative manner which is close to the underlying conceptual mathematical model. However, the drawback is that optimizations that require a more low-level control can not be realized.

- Modelica uses *acausal* connectors to assemble models of physical components. However, the coupling of partitions according to Figure 3 requires *causal* data-flow. It is not obvious how convenient and effective coupling schemes can be realized when a model should be partitioned at acausal connectors.

5.1 Mathematical Model

For the further analysis more detailed mathematical models than the one indicated in Figure 3 are proposed. As before, the discussion is based on two partitions, but carries over to more general settings involving n partitions.

Figure 4 depicts a continuous-time domain model for two coupled partitions, including inputs (u_1, u_2) and outputs (y_1, y_2) due to real-time I/O hardware devices. The dynamics of the coupled partitions are modeled as differential-algebraic equations (DAEs) in autonomous semi-explicit form

$$\dot{x}_1 = f_1\big(x_1(t), \tilde{x}_2(t), z_1(t), \tilde{z}_2(t), \tilde{u}_1(t)\big) \tag{4a}$$
$$0 = \gamma_1\big(x_1(t), \tilde{x}_2(t), z_1(t), \tilde{z}_2(t), \tilde{u}_1(t)\big) \tag{4b}$$
$$y_1 = g_1\big(x_1(t), \tilde{x}_2(t), z_1(t), \tilde{z}_2(t), \tilde{u}_1(t)\big) \tag{4c}$$
$$\dot{x}_2 = f_2\big(\tilde{x}_1(t), x_2(t), \tilde{z}_1(t), z_2(t), \tilde{u}_2(t)\big) \tag{4d}$$
$$0 = \gamma_2\big(\tilde{x}_1(t), x_2(t), \tilde{z}_1(t), z_2(t), \tilde{u}_2(t)\big) \tag{4e}$$
$$y_2 = g_2\big(\tilde{x}_1(t), x_2(t), \tilde{z}_1(t), z_2(t), \tilde{u}_2(t)\big) \tag{4f}$$

with differential variables $x_i \in \mathbb{R}^{n_{x_i}}$, algebraic variables $z_i \in \mathbb{R}^{n_{z_i}}$, (real-time) inputs $u_i \in \mathbb{R}^{n_{u_i}}$, and outputs $y_i \in \mathbb{R}^{n_{y_i}}$ where $n_{x_i}, n_{z_i}, n_{u_i}, n_{y_i} \in \mathbb{N}$ and consistent initial conditions $x_i(t_0) = x_{i,0}$, $z_i(t_0) = z_{i,0}$. The (continuous-time) variables with tilde, $\tilde{x}_i, \tilde{z}_i, \tilde{u}_i$ are reconstructed from a number of $m \geq 1$ sampled (discrete-time) values of the variables without tilde of the same name by means of extrapolation or interpolation using a "reconstruction" operator denoted Ψ

$$\tilde{x}_i(t), \tilde{z}_i(t) = \Psi_i(\chi_i, \zeta_i)(t) \quad t \in [T_n, T_{n+1}] \tag{4g}$$
$$\tilde{u}_i(t) = \Psi_{u_i}(\upsilon_i)(t) \tag{4h}$$

where $\chi_i, \zeta_i, \upsilon_i$ are sampled at time instants $t_k \in (T_{n-m}, T_n]$ (extrapolation), or $t_k \in (T_{n+1-m}, T_{n+1}]$ (interpolation)

$$\begin{gathered}\chi_{i_k}, \zeta_{i_k} = x_i(T_k), z_i(T_k) \quad k = 1 \ldots k_i,\ T_k < T_{k+1},\\ k_i \in \mathbb{N},\ \chi_i \in \mathbb{R}^{n_{x_i} \times k_i},\ \zeta_i \in \mathbb{R}^{n_{z_i} \times k_i}\\ \upsilon_{i_k} = u_i(T_k) \quad k = 1 \ldots k_{u_i},\ T_k < T_{k+1},\\ k_{u_i} \in \mathbb{N},\ \upsilon_i \in \mathbb{R}^{n_{u_i} \times k_{u_i}}.\end{gathered}$$

The operator Ψ is loosely borrowed from the mathematical framework described in [12, p. 1495], where it is defined as extrapolation operator. The definition there is mathematically more technical and rigorous than considered necessary for this work.

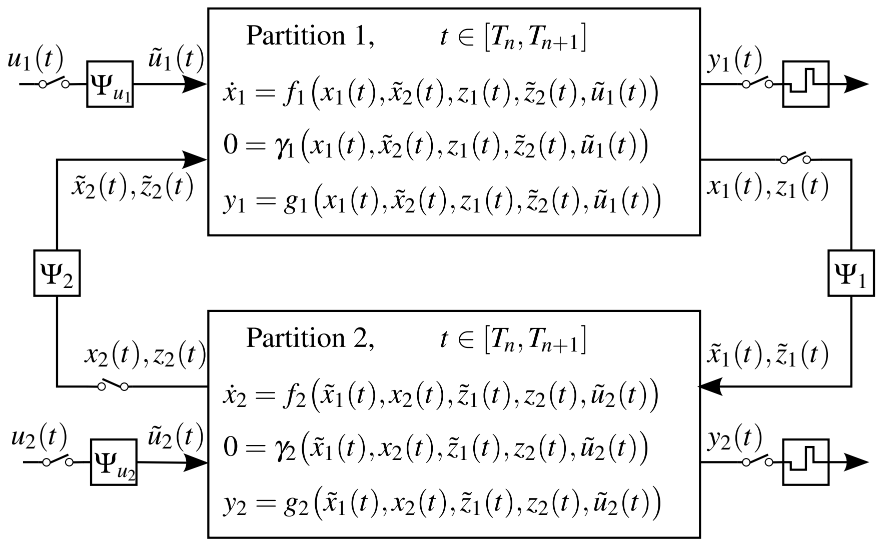

Partition 1, $t \in [T_n, T_{n+1}]$
$\dot{x}_1 = f_1(x_1(t), \tilde{x}_2(t), z_1(t), \tilde{z}_2(t), \tilde{u}_1(t))$
$0 = \gamma_1(x_1(t), \tilde{x}_2(t), z_1(t), \tilde{z}_2(t), \tilde{u}_1(t))$
$y_1 = g_1(x_1(t), \tilde{x}_2(t), z_1(t), \tilde{z}_2(t), \tilde{u}_1(t))$
Partition 2, $t \in [T_n, T_{n+1}]$
$\dot{x}_2 = f_2(\tilde{x}_1(t), x_2(t), \tilde{z}_1(t), z_2(t), \tilde{u}_2(t))$
$0 = \gamma_2(\tilde{x}_1(t), x_2(t), \tilde{z}_1(t), z_2(t), \tilde{u}_2(t))$
$y_2 = g_2(\tilde{x}_1(t), x_2(t), \tilde{z}_1(t), z_2(t), \tilde{u}_2(t))$
$u_1(t)$
Ψ_{u_1}
$\tilde{u}_1(t)$
$\tilde{x}_2(t), \tilde{z}_2(t)$
Ψ_2
$x_2(t), z_2(t)$
$u_2(t)$
Ψ_{u_2}
$\tilde{u}_2(t)$
$y_1(t)$
$x_1(t), z_1(t)$
Ψ_1
$\tilde{x}_1(t), \tilde{z}_1(t)$
$y_2(t)$

Figure 4: Illustration of the used continuous-time domain mathematical model for partition coupling, including external real-time I/O (u_1, u_2, y_1, y_2). The inputs to the partitions are first sampled at discrete time instants $t = T_0, T_1, \ldots, T_k$, subsequently the operators Ψ_i are applied to the (time-discrete) signals in order to provide extrapolated/interpolated continuous-time signals during a period of continuous-time system evolution ($t \in [T_n, T_{n+1}]$).

Data exchange between partitions and the update of coupling terms is restricted to the time-discrete synchronization points $t = T_0, T_1, \ldots, T_k$. In co-simulation the steps from $T_n \rightarrow T_{n+1}$ are referred to as *macro steps*. The union of all macro-time steps is a a macro-time grid where the partitions update their coupling terms. Generally, the real-time inputs and outputs of the respective partitions may be sampled at discrete time instants that are different to the macro-time grid instants. During a macro step $T_n \rightarrow T_{n+1}$ the dynamics of the partitions evolve according to the governing DAE using extrapolated (or interpolated) data. Note that $\tilde{x}_i, \tilde{z}_i, \tilde{u}_i$ are continuous in each macro step $T_n \rightarrow T_{n+1}$ but may have jump discontinuities at the synchronization points T_n.

Since this work is concerned with real-time simulation, it is natural to consider only equidistant macro-time grids with constant macro-step size $h = (T_{n+1} - T_n)$. Moreover, the utilized synchronous framework in conjunction with the discretization formula given in Table 1 suggests to describe the coupling within a (discrete-time) recurrence equation framework. The step-sizes h_i of the two partitions may differ, but the ratio between the slower and faster period must be an integer multiple. Without loss of generality assume that h_1 is the faster partition and denote $N = h_2/h_1$ as the frame ratio. The partitions are synchronized at the discretization points ($\equiv$ macro-time grid)

$$t = k \cdot N \cdot h_1 = k \cdot h_2 = k \cdot h, \; k \in \mathbb{N}. \qquad (5)$$

At these points the equations of both partitions have to be fulfilled concurrently (synchronous model of computation).

The overall discretized system equations can now be described in terms of the faster sampling period $h = h_1$, further also denoted as micro-time step. The precise time dependencies, i.e., at which instant $t_i = i \cdot h_1$ on the micro-time grid coupling variables from time instant $t_{i \leq j}$ are required depend on the utilized discretization method. Table 2 shows the dependencies for the solver methods supported by the Modelica standard. The important characteristics to be observed are:

1. Algebraic equations always, regardless of the utilized solver method, require the variable values at the current time instant on the micro/macro-time grid.

2. For state integration using `ExplicitEuler` coupling variable values and state variable values from previous activation times are sufficient.

3. The other two explicit methods require coupling variable values from the *current* time instant (due to the occurrence of term $\frac{u_i + u_{i+1}}{2}$ in the defining equations in Table 1)!

4. The implicit methods require coupling variable values and state variable values from the current time instant.

Table 2: Coupled variables time dependencies after discretization

Solver Method	*Time instant dependencies*
	For all methods: h_1 is the step-size of the fast partition, and $t_{i+1} - t_i = N \cdot h_1 = h_2 = h$ is the step-size of the slow partition, $N, j \in \mathbb{N}, N = h_2/h_1, j = 1 \ldots N$. $\tilde{x}_{1,i}, \tilde{z}_{1,i}, \tilde{x}_{2,i}, \tilde{z}_{2,i}$ are approximated from $x_{1,i}, z_{1,i}, x_{2,i}, z_{2,i}$ by a suitable extrapolation method. Time dependencies in the algebraic equations are always: $0 = \gamma_1(x_{1,iN+j}, \tilde{x}_{2,iN+j}, z_{1,iN+j}, \tilde{z}_{2,iN+j})$ $0 = \gamma_2(\tilde{x}_{1,(i+1)N}, x_{2,(i+1)N}, \tilde{z}_{1,(i+1)N}, z_{2,(i+1)N})$
`Explicit-Euler`	$x_{1,iN+j} = f_1(x_{1,iN+j-1}, \tilde{x}_{2,iN+j-1}, z_{1,iN+j-1}, \tilde{z}_{2,iN+j-1})$ $x_{2,(i+1)N} = f_2(\tilde{x}_{1,iN}, x_{2,iN}, \tilde{z}_{1,iN}, z_{2,iN})$
`Explicit-MidPoint2 / RungeKutta4`	$x_{1,iN+j} = f_1(x_{1,iN+j-1}, \tilde{x}_{2,iN+j}, z_{1,iN+j-1}, \tilde{z}_{2,iN+j})$ $x_{2,(i+1)N} = f_2(\tilde{x}_{1,(i+1)N}, x_{2,iN}, \tilde{z}_{1,(i+1)N}, z_{2,iN})$
`Implicit-Euler / Trapezoid`	$x_{1,iN+j} = f_1(x_{1,iN+j}, \tilde{x}_{2,iN+j}, z_{1,iN+j}, \tilde{z}_{2,iN+j})$ $x_{2,(i+1)N} = f_2(\tilde{x}_{1,(i+1)N}, x_{2,(i+1)N}, \tilde{z}_{1,(i+1)N}, z_{2,(i+1)N})$

This has the following consequences for clocked discretized continuous-time partitions that are coupled within Modelica's synchronous computation framework:

- In the general case, there is no scheduling of γ_1 and γ_2, that satisfies reciprocal data dependencies without resorting to extrapolation from previous values.

- State integration using `ExplictEuler` allows to use $\tilde{x}_{1,iN}, \tilde{z}_{1,iN} = x_{1,iN}, z_{1,iN}$ (no extrapolation needed, since values already available at $t_{i+1}N$). However, $\tilde{x}_{2,iN+j-1}, \tilde{z}_{2,iN+j-1}$ need extrapolation for $j > 1$. Scheduling of f_1, f_2 at the macro-time grid is always possible, since only past values are required at these points.

- For state integration using the remaining explicit and implicit methods, there is no scheduling of f_1 and f_2 that satisfies reciprocal data dependencies in the general case without resorting to extrapolation from previous values.

Note that in Modelica's synchronous computation framework it is not allowed to have algebraic loops *spanning* clocked discretized continuous-time partitions (however, it is allowed to have algebraic loops *within* a partition!). Therefore, there must be a sorting for the coupling equations at macro-time grid points that allows to evaluate them in a sequential order that satisfies data dependencies.

The previous discussion already allows to identify some of the consequences when using the synchronous framework for partition coupling:

- A staggered method (Gauss-Seidel scheme) that would first integrate the slow partition, extrapolating the inputs from the coupled (fast) partition, and after that integrate the fast partition, *interpolating* from the results of the slow partition, is not feasible. This is because within the synchronous framework new values are accessible only at the points at time where they are *valid* and not directly after they have been computed. Also, it is not possible for the modeler to directly influence the sequence of calculations. This is at the discretion of the simulation tool that will only guarantee to respect data flow dependencies.

- Nevertheless, an execution scheme similar to the "*Single task with no fast-frame real-time I/O*" depicted in Figure 2 is feasible, however, the interpolation of coupling variables during execution of the fast partitions is not possible (extrapolation is required).

- The synchronous model of computation makes the abstractions that equations at time instants are evaluated instantaneously. However, in reality, computation takes time. From a real-time I/O timing perspective it is desirable that the simulation time instants of inputs and outputs closely fit the real-time instants. For that reason, dedicated real-time integrators are typically designed to require only past inputs for the integration up to the current time instant. This allows to compute the output values required at real-time instants $t = T_n$ during a computational period scheduled at $t < T_n$. This computational timing details are beyond the scope of the Modelica specification. However, observe that all solvers specified in Table 1 (except `ExplicitEuler`) require the (real-time) inputs at $t = T_n$ in order to compute the outputs at $t = T_n$. Therefore, the real-time outputs will be inevitable afflicted with a (potentially considerable) computational delay ($t_{\text{outputs}} = T_n + t_{\text{delay}}$).

Obviously, it will often be desirable to keep at least t_{delay} as small as possible. For the "*Single task with no fast-frame real-time I/O*" scenario it is therefore advisable to schedule the computation of the slow frame's real-time outputs before executing the fast frames (just as depicted in Figure 2)[5].

- Parallel coupling methods using solely extrapolated coupling terms (Jacobi-scheme) can be realized within the available synchronous framework. Therefore, parallel execution of frames as depicted at the bottom of Figure 2 should be feasible. This is expected to be particularly attractive if a simulation can benefit from multi-core hardware and distribute the computational load on the available cores[6].

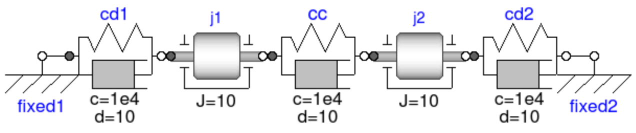
- Some of the solver methods in Table 1 are *multi-pass* methods[7], i.e., they require several "intermediate" evaluation of $f(..)$ during integration. Since communication within the synchronous framework can only occur at clock ticks, the needed intermediate values of the inputs u are computed by interpolation. The alternative to actually acquire (real-time) samples of u for this intermediate evaluations is not possible in the current framework. However, it should be noted that for dedicated real-time integrators the use of intermediate input samples is typically considered [1, 9].

5.2 Partition Coupling at Acausal Connectors

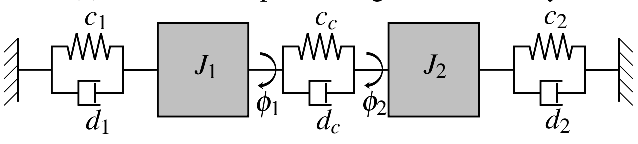
In Figures 3 and 4 the coupling of partitions is accomplished by causal (directed) dataflow. However, physical modeling in Modelica relies on acausal connectors. It is not obvious how to accomplish partition coupling at the boundary of acausal connectors.

The following discussion is based on couplings at the boundary of rotational mechanical connectors from the Modelica Standard Library. However, the presented principles are easily transferable to other physical connectors.

Consider the academic example model in Figure 5 and assume the model shall be partitioned somewhere between the inertias J_1 and J_2.



(a) Modelica component diagram of assembly.



(b) Schematic diagram of mechanical assembly.

Figure 5: The example model: a linear 2-DOF oscillator.

Figure 6 shows two common approaches to partition the model for modular integration [2]:

Force/displacement coupling Partition P_1 provides the cut torque at its boundary as output which is the input to partition P_2. Conversely, partition P_2 provides the displacement at its boundary as output which in turn is the input to partition P_1 (Figure 6a).

Displacement/Displacement coupling The coupling force element of S_1 is duplicated in S_2. The two partitions are coupled by the displacements of S_1 and S_2 (Figure 6b).

5.3 Implementation in Modelica

The Modelica MULTIRATE library provides convenient building blocks for partitioning a model for multi-rate and multi-method simulation. It is implemented on top of the synchronous language elements, partly reusing functionality provided by the MODELICA_SYNCHRONOUS library [10].

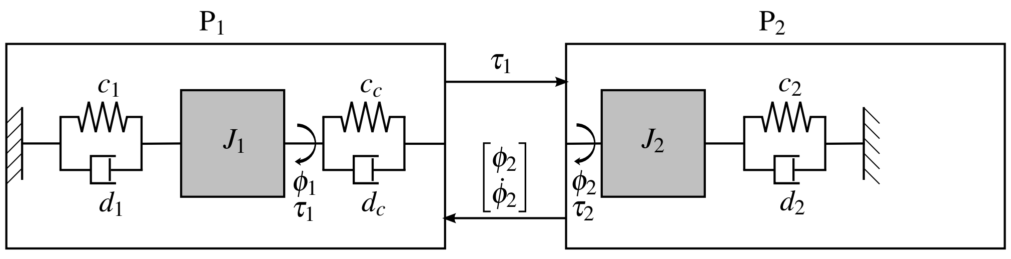
5.3.1 Force/Displacement Coupling

The component diagram in Figure 7a shows the oscillator from Figure 5a in a *force/displacement* coupling configuration. The coupling component `subSample1` is an instance of the `SubSampleForceDisp` class from the MULTIRATE library. The `SubSampleForceDisp` class provides a few parameters to modify the coupling characteristics, namely `inferFactor` to define that the

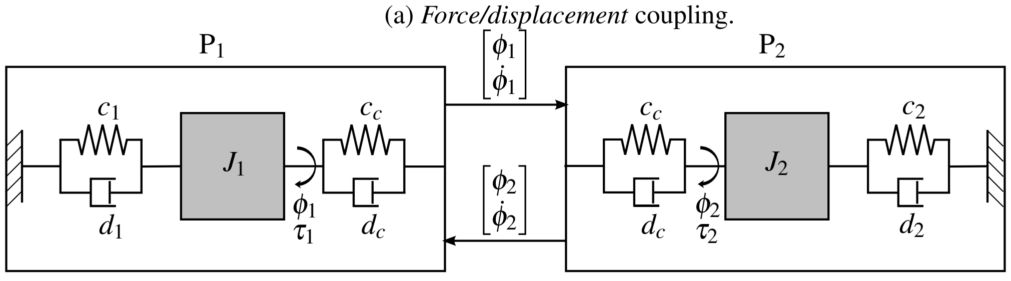
[5]The Modelica standard doesn't provide any possibility for a modeler to control the scheduling of computations. Therefore, a reasonable scheduling of real-time I/O can be seen as an implementation quality trait of a Modelica tool.

[6]However, the Modelica standard doesn't provide any means for a modeler to control whether computations are parallelized. At the time of writing this article the authors are not aware of any Modelica tool that supports parallelization of clocked partitions.

[7]Namely, the solver methods `ExplicitMidPoint2` and `ExplicitRungeKutta4`.



(a) *Force/displacement* coupling.


P$_1$
$\begin{bmatrix}\phi_1\\ \dot\phi_1\end{bmatrix}$
P$_2$
c_1
J_1
c_c
d_1
ϕ_1
τ_1
d_c
$\begin{bmatrix}\phi_2\\ \dot\phi_2\end{bmatrix}$
c_c
J_2
c_2
d_c
ϕ_2
τ_2
d_2

(b) *Displacement/displacement* coupling.

Figure 6: Partitioning the example model.

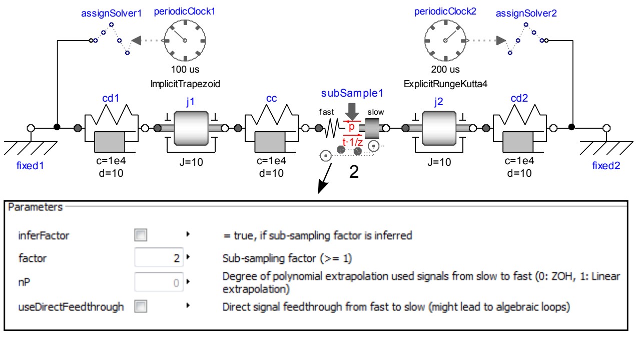
tool shall determine the sub-sample factor by clock inference. Otherwise, the sub-sample `factor` can be entered manually. Efficient polynomial extrapolation of arbitrary degree `nP` is supported for extrapolating values stemming from the slow partition during execution of the fast partition[8]. With default settings the output of the fast partition will be delayed one clock tick, however parameter `useDirectFeedthrough` allows to avoid that delay. The delay is necessary if both partitions need at macro-time steps $t = iN$ the current coupling values of the respective other partition. As has been discussed in Section 5.1 this reciprocal data dependencies lead to illegal algebraic loops spanning the coupled partitions. Note that if at least one of the partitions uses `ExplicitEuler` as solver method, a scheduling without algebraic loops may become feasible and `useDirectFeedthrough` may be set to **`true`**.

The design of the icon of the `SubSampleForceDisp` class gives a visual clue regarding the intended placement within a model. The component at the left hand side receives a displacement (ϕ and derivatives of ϕ), and needs to provide a torque τ. This is typical for spring like elements. The right hand side component receives a torque and reacts with a displacement. This is typical for inertia like components. The sub-sampling `factor` is displayed at the bottom of the icon (provided that `inferFactor=`**`false`**).

The internal structure of the `SubSampleForceDisp` class is depicted in Figure 7b. The parallel branches with component `uDirect1` and `unitDelay1`[9] are conditional branches. Their activation is mutually exclusive and depends on the value of the parameter `useDirectFeedthrough`. The `subSample1` block wraps Modelica's **`subSample`**`(..)` operator, which performs fast-to-slow rate transitions. Component `absoluteSensor` returns an array with the displacement variables $\{\phi, \dot\phi, \ddot\phi\}$. Upsampling and (polynomial) extrapolation of the variables is performed by the `superSample1` block. Since ϕ is a discrete-time (sampled) signal (see Figure 4), it carries no information about its derivatives. It is the task of component `move` to force the movement of flange `flange_a` according to signals ϕ, $\dot\phi$ and $\ddot\phi$. In the implementation smoothness information of ϕ is recovered at the sampling points by using the sampled values of $\dot\phi$ and $\ddot\phi$ and setting the recovered signal equal to `flange_a.phi`. This is accomplished by using the `derivative` annotation as described in [8, Section 17.7, "Declaring Derivatives of Functions"]. Components `torqueSensor` and `torque1` are from the Modelica Standard Library.

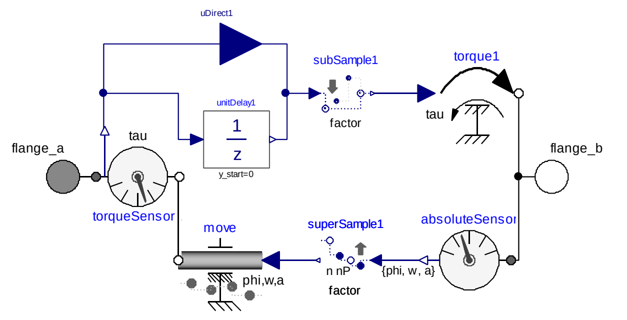
[8]Note that `nP=0` is equivalent to holding the value constant and `nP=1` is equivalent to linear extrapolation. Moreover, while increasing the order of extrapolation *may* result in improved numerical stability and accuracy in some applications it may also deteriorate numerical stability (see [2] for comprehensive numerical experiments regarding extrapolation and interpolation of coupling signals in co-simulation scenarios). In many cases, `nP=0` or `nP=1` seems to be a good choice.

[9]The `unitDelay1` block wraps the **`previous`** operator. According to the current specification [8, Section 16.8.1] the use of **`previous`** within a clocked discretized continuous-time partition is forbidden. However, the Dymola 2014 FD01 tool used for this work is more lenient and allows it. This is a desirable feature in order to enable the described implementation.



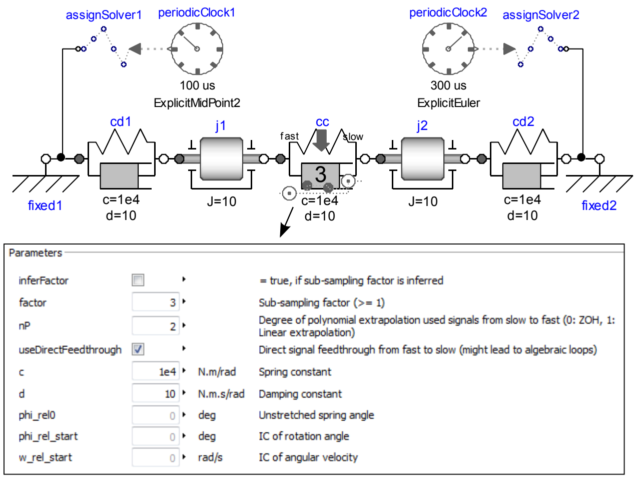
inferFactor	☐	= true, if sub-sampling factor is inferred
factor	2	Sub-sampling factor (>= 1)
nP	0	Degree of polynomial extrapolation used signals from slow to fast (0: ZOH, 1: Linear extrapolation)
useDirectFeedthrough	☐	Direct signal feedthrough from fast to slow (might lead to algebraic loops)

(a) Example model using a *force/displacement* coupling in a multi-rate and multi-method configuration. The coupling component `subSample1` is an instance of class `SubSampleForceDisp` from the MULTIRATE library.



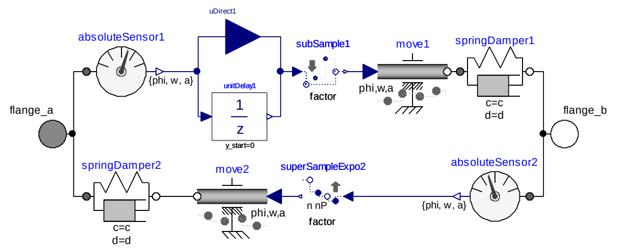
(b) Component composition diagram of class `SubSampleForceDisp`. This class implements the *force/displacement* coupling.

Figure 7: *Force/displacement* coupling using the MULTIRATE library.



inferFactor	☐		= true, if sub-sampling factor is inferred
factor	3		Sub-sampling factor (>= 1)
nP	2		Degree of polynomial extrapolation used signals from slow to fast (0: ZOH, 1: Linear extrapolation)
useDirectFeedthrough	✓		Direct signal feedthrough from fast to slow (might lead to algebraic loops)
c	1e4	N.m/rad	Spring constant
d	10	N.m.s/rad	Damping constant
phi_rel0	0	deg	Unstretched spring angle
phi_rel_start	0	deg	IC of rotation angle
w_rel_start	0	rad/s	IC of angular velocity

(a) Example model using *displacement/displacement* coupling in a multi-rate and multi-method configuration.



(b) Implementation of *displacement/displacement* coupling class.

Figure 8: *Displacement/displacement* coupling using the MULTIRATE-library.

5.3.2 Displacement/Displacement Coupling

The component diagram in Figure 8a shows the oscillator in a *displacement/displacement* coupling configuration. The component `cc` is an instance of class `SubSampleDispDisp` from the MULTIRATE library. It constitutes both: the dynamic equations of motion for the spring/damper element and the partition coupling equations.

The upper part of parameters of the `SubSampleDispDisp` class is identical to the parameters provided by the `SubSampleForceDisp` class. The lower part provides parametrization for the spring/damper element.

The internal structure of the `SubSampleForceDisp` class is depicted in Figure 8b. Note that the dynamic equations for the spring/damper element are duplicated: while component `springDamper1` is assigned to the slow partition, `springDamper2` is assigned to the fast partition. This "overlapping" integration often leads to more favorable numerical stability properties (see Busch [2]). Aside from this, the components appearing in Figure 8b are already known from Figure 7b.

While this section described the partitioning at the example of 1-dimensional, rotational mechanics, it is needless to say that the basic approach carries over to other physical domains. Furthermore, the mindful reader may miss a `SubSampleDispForce` class. That class was omitted, since it basically results by swapping and adapting respective components in Figure 7b.

6 Application to a 6-DOF Robot Model

In order to understand whether the partitioning is suitable for "real-world" systems with considerable complexity, the `RobotR3` example (a detailed model of a robot with six degrees of freedom) from the `MultiBody` package of the Modelica Standard Library (MSL) was adapted and partitioned into three parts (see Figure 9):

1. **"`clockControl`" partition**. The partition consists of a clocked discrete-time path planning component and clocked discrete-time P-PI cas-

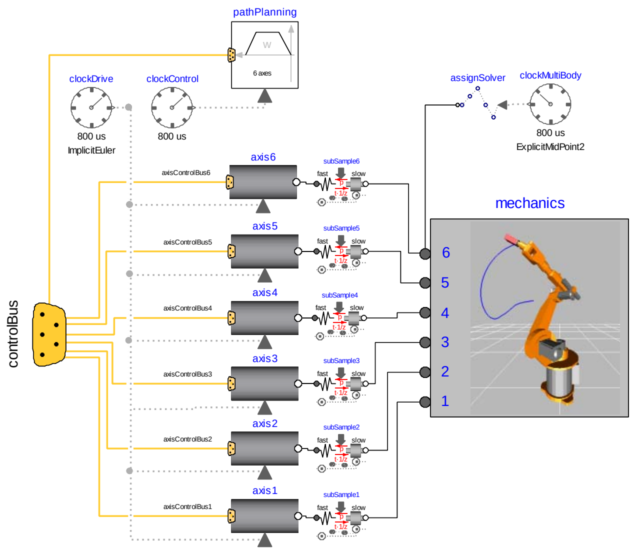


Figure 9: 6-DOF robot example adapted from the `RobotR3` example of the multi-body package of the MSL.

cade controllers for the six axes (inner PI-controllers to control the motor speeds, and outer P-controllers to control the motor positions). Therefore, the continuous-time controllers and the path planning component from the original `RobotR3` example were replaced by a discrete-time (digital) implementation. The sample period of that partition is set to 800 μs.

2. **"`clockDrive`" partition**. The remaining parts in each axis (motor including current controller and the gearbox including gear elasticity and bearing friction) are combined into a clocked discretized continuous-time partition. The steady-state initialization found in the original `RobotR3` example was removed since the initialization of clocked partitions differs from the standard scheme of initialization in Modelica [8, Section 16.9, "16.9 Initialization of Clocked Partitions"]. Instead, compatible initial values have been set at appropriate places. "`ImplicitEuler`" with a step size of 800 μs is used as solver method.

3. **"`clockMultiBody`" partition**. Except for setting compatible initial values, the multi-body part is identical to the original `RobotR3` model. "`ExplicitMidPoint2`", also with a step size of 800 μs, is used as solver method.

Partitions "`clockDrive`" and "`clockMultiBody`" are coupled at the mechanical flanges connecting the axes with the `mechanics` multi-body system by *force/displacement* components with constant extrapolation (`nP=0`) and no direct feedthrough.

The simulation performance using the solver clocks was compared against simulations performed on the same model (using various solvers), but without using coupled clocked discretized continuous-time partitions (see Figure 10).

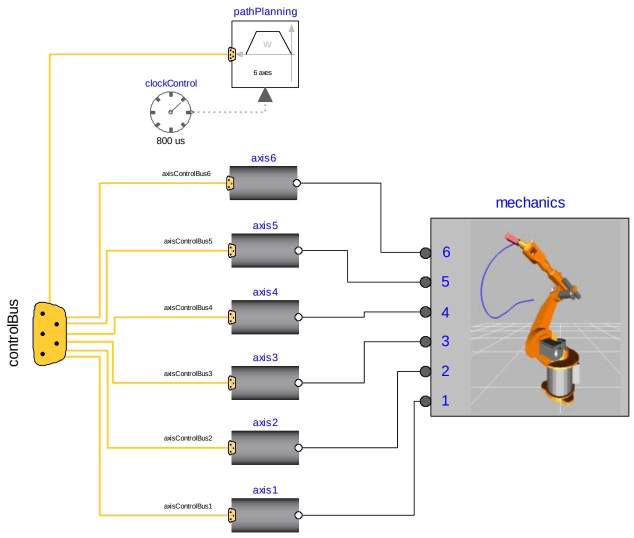


Figure 10: Comparison model: 6-DOF robot example without coupled clocked discretized continuous-time partitions.

The numerical experiment is conducted on a notebook with an Intel Core 2 Duo CPU P9700 @ 2.8 GHz and 4.0 GB of RAM. The simulation tool is Dymola 2014 FD01 running on a 64-bit Microsoft Windows 7 operating system.

The reference simulation result is obtained by simulation of the comparison model (Figure 10) using DASSL as integrator. The solver parameter "Tolerance" is set to 0.0001 for all simulation runs. The simulation interval is always set to $[0,2]$ seconds. The step size of the tested solver methods is iteratively increased until either integration fails (which for the considered model typically means that too large residuals appear while solving (nonlinear) systems of equations), or the simulation result deviates considerably from the DASSL reference solution.

The decision whether a result deviates considerable from the reference solution is made by two criteria: a) visual inspection of the trajectory of the load at the robot arm tip, and b) deviation of the trajectory to the reference solution defined by the norms

$$E_2 = \sqrt{\int_{t_0}^{t_e} e_r(t)^2 \, \mathrm{d}t} \tag{6a}$$

$$E_\infty = \sup_{t \geq t_0}(e_r(t)) \tag{6b}$$

where t_0 is the simulation start time, t_e is the simulation stop time and

$$e_r(t) = \frac{\|r(t) - r_{\mathrm{DASSL}}(t)\|}{\|r_{\mathrm{DASSL}}(t)\|} \qquad r, r_{\mathrm{DASSL}} \in \mathbb{R}^3$$

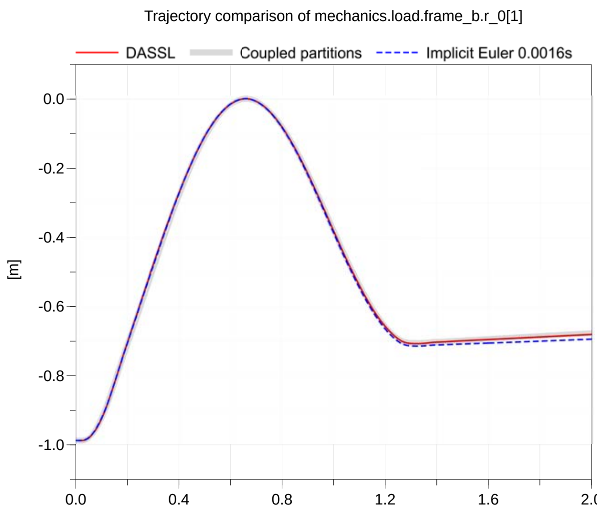
provides a relative error measurement by relating the distance between the obtained solution $r(t)$ and the reference solution $r_{\mathrm{DASSL}}(t)$ to the magnitude of the vector $r_{\mathrm{DASSL}}(t)$. For the actual computation of E_2 and E_∞, (6a) and (6b) are numerically evaluated over an uniformly spaced grid with spacing $\Delta t = 0.0008$. Therefore, the integral in (6a) is approximated by numerical summation.

In Figure 11 the DASSL reference solution is compared with the result when simulating the coupled partitions model of Figure 9 and the result when simulating the comparison model of Figure 10 with an inline implicit Euler solver.

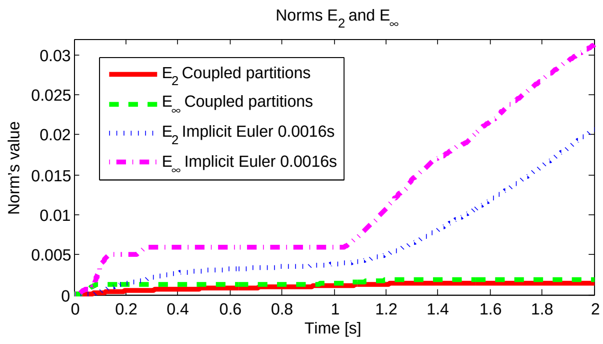
Figure 11a shows the first Cartesian coordinate of the trajectory of the robot arm tip. It can be observed that the solution computed by the implicit Euler solver with fixed step size 0.0016 s diverges considerably at the end of the simulation run. Examining the evolution of the norms E_2 and E_∞ confirms that observation (see Figure 11b). At $t = 2$ s the relative error of the implicit Euler solution compared to the DASSL reference is about 3%.

Beside using the DASSL integrator, various real-time inline integrators provided by the tool Dymola [5] were tested with the comparison model. Table 3 summarizes the results obtained by that simulation experiments and contrasts them to the result obtained by simulation of the coupled partitions model of Figure 9.

The *inline implicit Euler* solver displayed the best performance of the tested "conventional" solvers. However, for this scenario it was possible to even outperform that solver by a factor of about 2.9 (at comparable accuracy to the reference solution) by using a coupling of clocked discretized continuous-time partitions in combination with judiciously selected solver methods.



(a) Trajectory of the first Cartesian coordinate of the robot arm tip.



(b) Evolution of the norms E_2 and E_∞ over the simulation time.

Figure 11: Simulation results of coupled partition approach and implicit Euler solver (step size 0.0016 s) compared to the DASSL reference solution.

7 Conclusions

This paper presented a new approach in Modelica that allows a modeler to separate a model into different partitions for which individual solvers can be assigned. This effectively allows multi-rate and multi-method time integration schemes that can improve simulation efficiency in certain cases. Additionally, there is a potential to execute the partitions in parallel to gain further simulation speedups. However, this is not supported by currently available tools.

The approach is based on *clocked discretized continuous-time partitions*, a concept that was introduced as part of the synchronous language elements extension into the Modelica 3.3 language standard. However, until now it has not been applied in the context considered in this article.

The article started with a formal description of

Table 3: Comparison of solver methods

Solver	Step size (s)	CPU-time for integration (s)	E_2	E_∞
DASSL		10.8	0	0
Coupled partitions	$2 \times \mathbf{0.0008}$	**0.7**	0.0014	0.0019
Inline implicit Euler	**0.0008**	**2.0**	0.0058	0.0081
Inline implicit Euler (considerable deviation)	*0.0016*	(*1.2*)	0.0207	0.0313
Inline trapezoidal	0.0008	2.1	0.0059	0.0082
Inline trapezoidal (considerable deviation)	*0.0016*	(*1.3*)	0.0208	0.0313
Inline explicit Euler	0.00002	2.3	0.0009	0.0017
Inline explicit Euler	*0.00004*	*(Failed)*		
Mixed explicit/implicit Euler	0.0004	3.5	0.0004	0.0008
Mixed explicit/implicit Euler	*0.0008*	*(Failed)*		
Inline explicit RK 4	0.00002	8.6	0.0004	0.0009
Inline explicit RK 4	*0.00004*	*(Failed)*		

the mathematical prerequisites of coupling partitions within the synchronous language elements framework with special regard to timing requirements inherent to real-time simulations. In the following, the implementation of a Modelica library for the partitioning of physical models, denoted as MULTIRATE library, was sketched out. Finally, elements of that library were used to partition a detailed robot model and the simulation performance of that partitioned model was compared to "conventional" inline integrators provided by the Dymola tool. This numerical experiment illustrated: a) that the partitioning is feasible also for comprehensive, "real-world" models, and b) that by using a coupling of clocked discretized continuous-time partitions in combination with judiciously selected solver methods considerable simulation speedups can be achieved (the speedup factor was about 2.9 for the example model).

Despite this encouraging result, it also needs to be noted that it can take substantial efforts to find a *good* partitioning and select a combination of solver methods and corresponding integration step sizes that outperform a simulation using "conventional" solvers. Nevertheless, particularly for real-time simulations, investing that additional effort may be worthwhile if real-time constraints cannot be satisfied by using a conventional global solver approach.

Acknowledgements

Partial financial support of DLR by the German BMBF (within the ITEA2 project MODRIO) and of Dassault Systèmes by the European Union (within the CleanSky project MODELSSA) for this development is highly appreciated.

References

[1] Dennis S. Bernstein. The treatment of inputs in real-time digital simulation. *SIMULATION*, 33(2):65–68, 1979.

[2] Martin Busch. *Zur effizienten Kopplung von Simulationsprogrammen*. PhD thesis, Universität Kassel, 2012.

[3] Francois Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer Science + Business Media, Inc., 2006.

[4] Hilding. Elmqvist, Sven Erik Mattsson, and Hans Olsson. New Methods for Hardware-in-the-Loop-Simulation of Stiff Models. In Martin Otter, Hilding Elmqvist, and Peter Fritzson, editors, 2nd *Int. Modelica Conference*, Oberpfaffenhofen, Germany, March 18–19 2002.

[5] Hilding Elmqvist, Martin Otter, and Françoise E. Cellier. Inline Integration: A New Mixed Symbolic/Numeric Approach for Solving Differential-Algebraic Equation Systems. In *European Simulation Multiconference*, 1995.

[6] R. Kübler and W. Schiehlen. Modular simulation in multibody system dynamics. *Multibody System Dynamics*, 4(2-3):107–127, 2000.

[7] Jim Ledin. *Simulation Engineering*. CMP Books, Lawrence, Kansas 66046, first edition, 2001.

[8] Modelica Association. Modelica—A Unified Object-Oriented Language for Systems Modeling v3.3. Standard Specification, May 2012. available at http://www.modelica.org/.

[9] Stefan Nowack and Georg Feil. Reduction of time delays in Runge-Kutta integration methods. *SIMULATION*, 48(1):24–27, 1987.

[10] Martin Otter, Bernhard Thiele, and Hilding Elmqvist. A Library for Synchronous Control Systems in Modelica. In Martin Otter and Dirk Zimmer, editors, 9th *Int. Modelica Conference*, Munich, Germany, September 2012.

[11] Anton Schiela and Hans Olsson. Mixed-mode Integration for Real-time Simulation. In *Modelica Workshop 2000 Proceedings*, pages 69–75, Lund, Sweden, 2000.

[12] Tom Schierz and Martin Arnold. Stabilized overlapping modular time integration of coupled differential-algebraic equations. *Applied Numerical Mathematics*, 62(10):1491 – 1502, 2012.