A Toolchain for Real-Time Simulation using the OpenModelica Compiler

Niklas Worschech   Lars Mikelsons
Bosch Rexroth
Rexrothstraße 3, 97816 Lohr am Main

Abstract

Nowadays, simulation is the key technology to shorten development times, while increasing the functionality of products. In this context simulation is always used in order to verify characteristics of the product under consideration. In the past simulation was mostly done offline, i.e. not synchronized to real-time. Due to the increased computing power, the relevance of real-time simulation has increased in the last years. Therefore, several simulation environments offer a toolchain for real-time simulation, e.g. the Real-Time Workshop integrated in Simulink. In this paper such a toolchain (although not yet fully automated) for the OpenModelica Compiler (OMC) is presented using a hydro-mechanical system as an example. Thereby, this paper describes a modular C++ Simulation-Runtime for the OMC including a numerical integration method suitable for real-time simulation as well as modeling details of the example system using Modelica. *Keywords: real-time; simulation; runtime; OpenModelica*

1   Introduction

Simulation is always based on models. These models can be mind-models, scaled physical models or mathematical models. No matter what kind of model is used, the purpose of simulation is mostly the validation of characteristics of physical systems. Nowadays, even detailed mathematical models can be simulated in relatively short time. Hence, computer-simulation is an important tool in the mechatronic development cycle and helps to reduce costs by shorten the development process. The mechatronic-development cycle involving the validation process is visualized in the V-Model in figure 1.

Clearly, the level of detail of the employed model plays a very important role. To obtain a model with a higher level of detail, more modeling effort has to be invested and one has to expect longer simulation times.
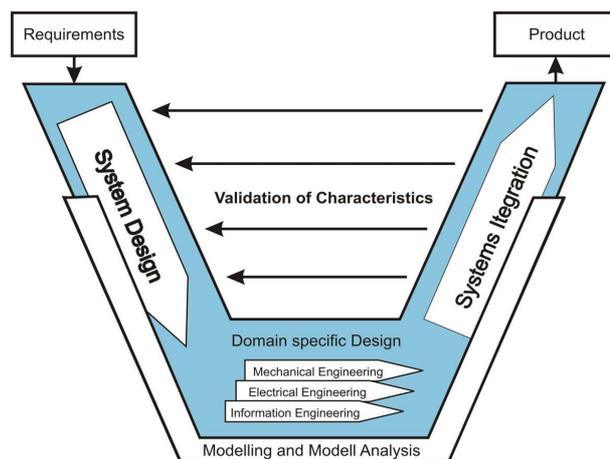
Figure 1: V-Model of the mechatronic development cycle

A proper model is as simple as possible, but still complex enough to reproduce the physical effects under consideration [9]. However, there exist tasks that can not be fulfilled satisfactorily with the help of non-real-time simulations regardless of which level of detail is used. These are among others:

- Setting up Simulators (e.g. driving simulator),

- Controller testing,

- Physical Component testing.

Real-time simulation refers to a mathematical model of a physical system including a numerical integration method that can execute at the same rate as actual "wall clock" time. Hence, using real-time simulation, the real system can be replaced by a virtual system which makes real-time simulation suitable for the applications mentioned above. Due to this possibility and the increased available computing power, real-time simulation became very popular in the recent years.

Consequently, many commercial simulation tools offer a complete toolchain for real-time simulation. Such

a toolchain consists of a modeling environment, a simulation-runtime and a compiler which can compile the model for a real-time-target. Simulink together with the Real-time Workshop form the toolchain offered by The MathWorks. Some other tools do not offer an own compiler, but an export to Simulink, so that the real-time Workshop can be used. There are also tools which offer an integrated solution. However, currently the OMC lacks such an automated toolchain at all. In this paper a C++ Simulation-Runtime is presented which forms the basis for a toolchain for real-time simulation. This modular C++ Simulation-Runtime contains a numerical integration method suitable for real-time simulations of hydraulic systems and can also be used for co-simulation.

This contribution is structured as follows. In section 2 the C++ Simulation-Runtime and its structure is presented. After that the toolchain for real-time simulation is explained using an application example in section 3. Here, the C++ Simulation-Runtime is compiled together with the application example for the real-time operating system Scale-RT [2] and executed on a real-time-target after that. The paper closes with a conclusion and an outlook.

2 A C++ Simulation-Runtime for OpenModelica

In order to set up an automated toolchain for real-time simulation, a new C++ Simulation-Runtime was designed. The design-guidelines were chosen to obtain a simulation-runtime that is easy to

- maintain,

- extend,

- configure.

Therefore, it is much easier to add new numerical integration methods, extend its functionality with new algorithms (e.g. for initialization) or just to fix bugs. In order to obtain a simulation-runtime that realizes these design-guidelines, the solver-component which implements the numerical integration method is separated from the system-component which represents the system of differential-algebraic equations (DAE). Note, that this design is completely contrary to the idea of inline-integration which was invented in order to increase the computational efficiency [8]. In the next section a general overview is given. After that the Event-Handling strategy is explained. In section 2.4

the chosen numerical integration method for real-time simulation is described.

2.1 Components Overview and General Interface Description
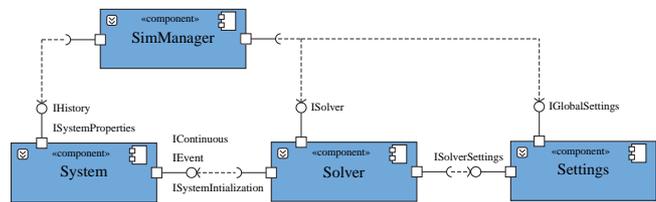
Figure 2: Components of the C++ Simulation-Runtime

In figure 2 the component diagram of the C++ Simulation-Runtime is pictured. The solver-component consists of a set of integration methods, e.g. CVode from the Sundials library [12]. The SimManager-component controls the simulation. Besides standard-tasks like starting and stopping of the simulation, the SimManager is able to synchronize different systems and solvers and hence allows for co-simulation. The settings-component is used to configure the simulation, e.g. set solver-tolerances. The system-component represents the DAE and therefore includes the Modelica-System class. This class is generated by a new code-generation module inside the OpenModelica compiler [10]. As mentioned above the solver-component is separated from the system-component and thus interfaces are used (see figure 3).
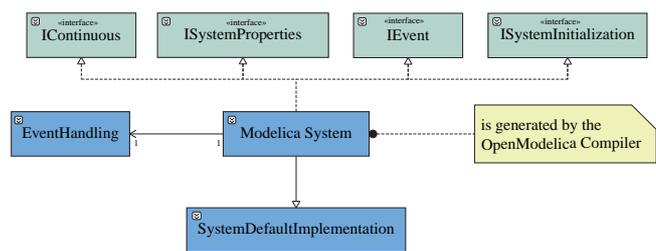
Figure 3: Modelica-System class

The C++ Simulation-Runtime is able to handle systems with a lot of different properties as shown in figure 4. Some of the properties (like *isAlgebraic*) are standard properties and used to automatically select a suitable numerical solution method for the corresponding system. Other properties are not yet reported by the OMC to the C++ Simulation-Runtime. A flag to use a symbolic jacobian for the numerical integration is part of current work. The generation of the symbolic

jacobian is described in [5]. The interface *ISystemI-*
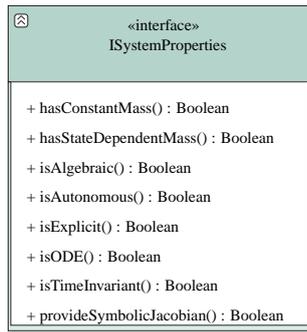
Figure 4: ISystemProperies Interface

nitialization is used to initialize the Modelica-System at the beginning of the simulation. Since the efficient initialization of models is part of current work [6], the currently implemented algorithms are rather basic. However, due to the design of the C++ Simulation-Runtime, new initialization-algorithms can be easily added. The communication between solver and sys-
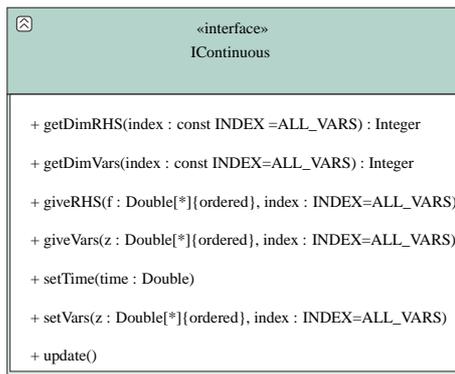
Figure 5: IContinous Interface

tem is defined by the interface *IContinuous* (see figure 5). The method *giveVars* returns the state-vector **z**. The state-vector is sorted according to the variable-index (see table 2.1) and hence it is possible to access a corresponding part of the state-vector by passing the variable-index. This sorting allows for efficient generation of the jacobian [11]. The remaining methods

Variable Index	Description
VAR_INDEX0	States of systems of 1st order
VAR_INDEX1	1st order States of systems of 2nd order, e.g. positions
VAR_INDEX2	2nd order States of systems of 2nd order, e.g. velocities
DIFF_INDEX3	Constraints on position level only
DIFF_INDEX2	Constraints on velocity level only
DIFF_INDEX1	Constraints on acceleration level only
ALL_RESIDUALS	All constraints
ALL_STATES	
ALL_VARS	

Table 1: The Variable Index

are basic methods needed for the numerical integra-

tion process.

In case that the OMC returns algebraic equation
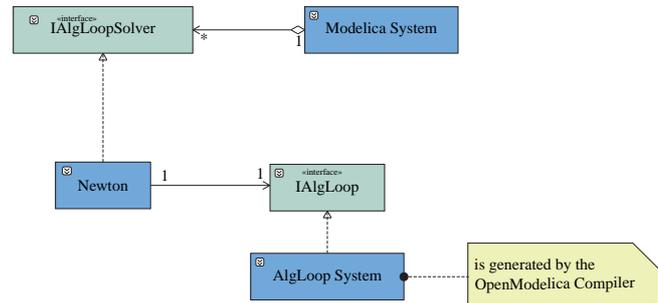
Figure 6: Solving Non Linear and Linear Systems

systems (as shown in figure 6), an instance of the AlgLoop-System class is created for each equation system. Once again, the Algloop-System class provides a method which allows to choose an adequate numerical solution method.

The simulation results are currently stored in a tabulator separated text-file. The Modelica-System class uses an instance of type *IHistory* to store the simulation results. Moreover, the storing instance uses a policy class for the implementation of the storing behavior [3]. This allows an extension of the output mechanism of simulation results, e.g storing the results in a buffer for further processing. In the future simulation results will be stored in the new Modelica result-file-format.

2.2 Integration Loop
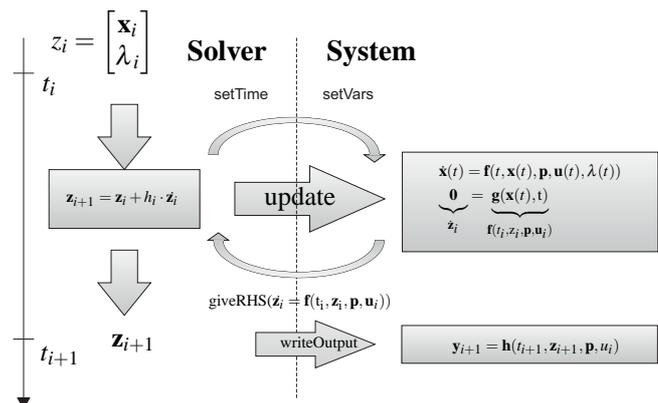
Figure 7: Integration loop in the C++ Simulation-Runtime

A scheme of the integration loop for a semi-explicit DAE

$$\dot{\mathbf{x}}(t) = \mathbf{f}(t, \mathbf{x}(t), \mathbf{p}, \mathbf{u}(t), \lambda(t)), \qquad (1a)$$

$$\mathbf{0} = \mathbf{g}(\mathbf{x}(t), t), \qquad (1b)$$

can be seen in figure 7. Here, **x** denotes the states, λ is the vector of algebraic variables, **p** are the parameters and $\mathbf{u}(t)$ are the system inputs. The time-step starts by setting the previously calculated state-vector and the current time. The right-hand-side of equation 1a is evaluated by calling *update*. Note that algebraic loops are solved within this call. After that *giveRHS* gives the right-hand-side to the numerical integration method which performs the integration step (e.g. using Forward-Euler).

2.3 Event-Iteration
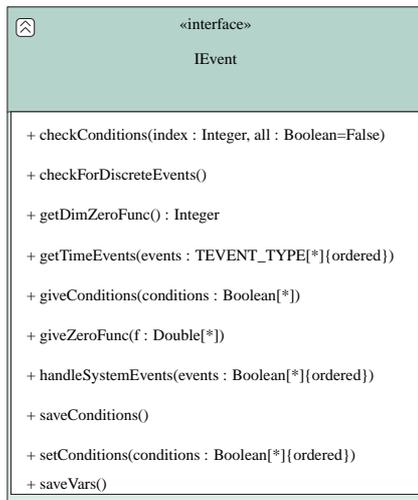
Figure 8: IEvent Interface

is an event-vector **e** indicating which zero-crossing-function (i.e. transition) is active. The relevant relation expressions are evaluated and stored in a condition-vector using *checkConditions*. This condition-vector is used in the *update* method to evaluate the right-hand-side of equation 1a. The method *saveVars* is called to save the predecessor values of all variables.
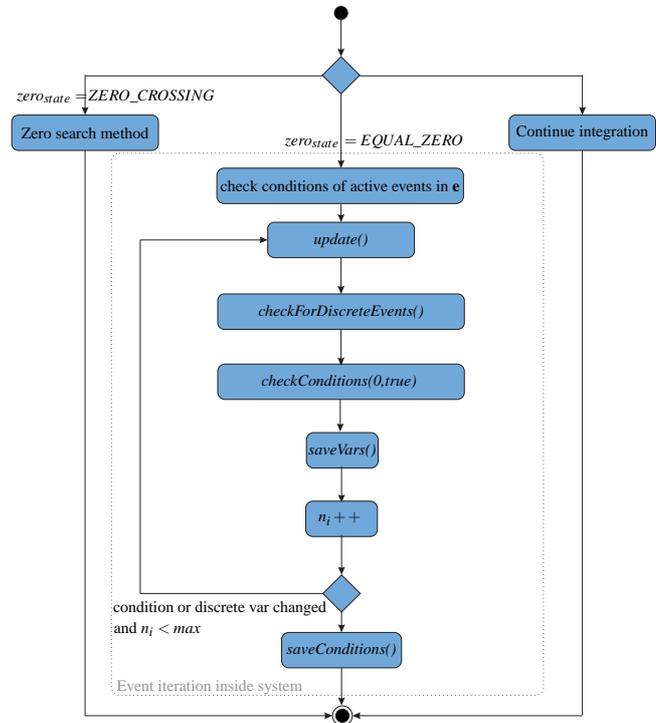
Figure 9: Event Iteration within an integration step

To handle discontinuities the Modelica-System implements the *IEvent* interface (figure 8). For each continuous event from the Modelica model, a zero-crossing- function and a corresponding condition variable is created. Thereby, the zero-crossing-functions are interpreted as transitions in a state-graph. To be more precise, the zero-crossing-functions are always negative as long as no event occurs. A positive zero-crossing-function indicates an event and in the consequence the event is handled (and the event-iteration is started) such that the corresponding zero-crossing-function is negative again. Note, that this is fundamental difference to the treatment of events in the current C Simulation-Runtime and allows the use of the built-in zero-detection algorithms of the Sundials library. These algorithms are very efficient since all ODE/DAE solvers of the Sundials library are multi-step methods and hence the solution polynomial is at hand with no additional effort.

When a zero is found an event-iteration is started as pictured in figure 9. The input of the event-iteration

2.4 Real-time Simulation

Real-time Simulation refers to a mathematical model of a physical system including a numerical integration method that can execute at the same rate as actual "wall clock" time. Hence, two requirements have to be met:

- The simulation has to be faster than the "wall clock" time.

- A predictable worst-case runtime is required.

The first requirement is a requirement on the computational complexity and hence a requirement for the model as well as for the numerical integration method. An approach for the generation of models suitable for real-time simulation can be found in [13]. The choice of the numerical integration method is even more restricted by the second requirement which is mostly harder to meet than the first one. A predictable worst

case runtime can only be obtained with non-iterative algorithms. To be more precise implicit numerical integration methods can not be used (without modification) inside a real-time process. Note that this requirement is rather problematic in the context of stiff ODEs and DAEs. Furthermore, step-size control produces a non-predictable runtime and can thus also not be used. The same holds for many algorithms for the detection of zero-crossings.

Since explicit numerical integration methods are not suited for many practical problems and implicit methods are not allowed inside a real-time process, linear-implicit integration methods with fixed step size are very common for real-time simulation [4]. Using a linear-implicit integration method, not a non-linear, but a linear system of equations has to be solved. This operation can be performed with an upper bound for the computational effort and hence linear-implicit integration methods can be used in real-time processes. Linear-implicit methods can for example be obtained by linearizing the numerical integration method. In that case linear-implicit methods inherit the stability properties of the corresponding implicit method due to the linearity of Dahlquist's test equation [11].

The most popular linear-implicit integration scheme is the linear-implicit Euler-method due to its simplicity and stability properties, i.e. it is A- and L-stable like the Backward-Euler [7]. These properties make it better suited for practical (i.e. stiff) problems than explicit methods. Unfortunately, it is of the same order as Backward-Euler which might be problematic in combination with a fixed-step size for low tolerances. An alternative is the linear-implicit trapezoidal-rule. This method has the same complexity as Backward Euler but is of order two. However, the linear-implicit trapezoidal rule is not L-stable due to the stability properties of the trapezoidal-rule and should thus not be used for stiff problems.

The C++ Simulation-Runtime offers an A- and L-stable linear-implicit integration method of order three which will be called LI3 in the following. This method was designed for the solution of discretized unsteady incompressible Navier-Stokes equations originally and has not been used for real-time simulation yet (to the author's knowledge) [14]. For an ODE as

in equation 1a the method can be written as

$$\mathbf{k}_1 = \mathbf{x}_n + \frac{2h}{3}\mathbf{L} \cdot \mathbf{f}(\mathbf{x}_n, t_n), \tag{2}$$

$$\mathbf{k}_2 = \mathbf{L}(\mathbf{x}_n - \frac{h}{2}\mathbf{J} \cdot \mathbf{k}_1 + \frac{h}{3}\mathbf{f}(\mathbf{x}_n, t_n) + \frac{h}{3}\mathbf{f}(\mathbf{k}_1, t_n + \frac{2h}{3})), \tag{3}$$

$$\bar{\mathbf{k}} = \frac{9}{4}\mathbf{k}_1 - \frac{3}{4}\mathbf{k}_2 - \frac{1}{2}\mathbf{x}_n, \tag{4}$$

$$\mathbf{k}_3 = \mathbf{L}(\mathbf{x}_n - \frac{h}{2}\mathbf{J} \cdot \bar{\mathbf{k}} + \frac{h}{4}\mathbf{f}(\mathbf{x}_n, t_n) + \frac{3h}{4}\mathbf{f}(\mathbf{k}_1, t_n + \frac{2h}{3})), \tag{5}$$

$$\mathbf{x}_{n+1} = \mathbf{L}(\mathbf{x}_n - \frac{h}{2}\mathbf{J} \cdot \bar{\mathbf{k}} + \frac{h}{4}\mathbf{f}(\mathbf{x}_n, t_n) + \frac{3h}{4}\mathbf{f}(\mathbf{k}_2, t_n + \frac{2h}{3})), \tag{6}$$

where

$$\mathbf{L} = (\mathbf{E} - \frac{h}{2}\mathbf{J})^{-1}. \tag{7}$$

Here $\mathbf{J}$ denotes the jacobian of $\mathbf{f}$ (or at least an approximation) and h is the step-size. Thus, one time-step requires three evaluations of the right-hand side of the ODE. Moreover, four linear systems of equations of the same dimension as $\mathbf{x}$ have to be solved. Thus, the structure of LI3 is similar to the structure of a linear-implicit method obtained from a diagonally-implicit Runge-Kutta method. Note that the solution of these four systems is computationally cheaper than solving a system of dimension $4 \cdot \dim(\mathbf{x})$ which would result from a linear-implicit method obtained from a implicit Runge-Kutta method. The proof for the stability properties as well as for the order can be found in [14].

Clearly, a time-step with LI3 is computationally more expensive than a time-step with the linear-implicit Euler-method. However, LI3 allows to use larger step-sizes due to the higher order. This is expressed in the engineers rule of thumb that a method of order p should be used for a tolerance of 10^{-p}.

Consequently, stability properties, order and computational complexity make LI3 suitable for real-time simulation of stiff problems and hence hydro-mechanic systems.

Since no iterative algorithm for the detection of zero-crossings can be used, the zero-crossing is assumed to be in the middle of the last solution interval. Note that this leads to an increase in the worst-case runtime of a factor of three.

3 Application Example

In the last section a C++ Simulation-Runtime for the OMC was presented. This simulation-runtime forms

Figure 10: Pieter Schelte (picture taken from [1])

Figure 11: Object diagram of the Y-drive

the basis for an automated toolchain for real-time simulation. The workflow of this toolchain is explained in this section using a hydro-mechanical heavy-duty system as an example. In the next subsection the set-up of the real-time simulation is explained. After that some modeling details and simulation results are given.

3.1 Real-Time Simulation Set-Up

The toolchain consists of the OMC as a Modelica Compiler, the C++ Simulation-Runtime, a cross-compiler for ScaleRT and the real-time operating system itself. The hardware setup to execute the real-time simulation of the Modelica model using SCALE-RT requires a host and a target PC. The host PC is standard Windows PC while the target PC uses ScaleRT (Linux with Xenomai real-time extension).

The output of the OMC is coupled to the ScaleRT interface and cross-compiled for ScaleRT. The automation of this step is part of future work. After that the code can be transfered to the target and started via the ScaleRT software in a graphical-user-interface.

Note that in contrast to the OMC neither ScaleRT nor the coupling of the C++ Simulation-Runtime to the real-time interface is Open Source.

3.2 Modeling of the Example System

The application example is a part of a hydro-mechanical heavy-duty system, which is designed to operate on a ship for the installation and removal of oil-platforms. The ship is currently under construction and is pictured in figure 10. The complete hydro-mechanical system consists of eight beams, each with a clamp (or gripper) at the end. During operation the beams move towards the legs of a platform and grip

them. After that the platform can be lifted and removed (details can be found at [1]). Each beam can be divided into a Y- and a Z-drive. In this paper only the Y-drive is modelled and simulated. The Y-drive is used to compensate sea motion, driving the beam towards the leg and applying a constant force towards the leg in case of a contact (in order to avoid hammering). It consists of

- a hydraulic cylinder,

- a 3-way hydraulic valve,

- an electrical drive,

- the beam,

- gears,

- and a force controller.

The electrical drive moves the beam towards the leg using position control, while the cylinder applies a constant force towards the leg during contact using force control.

The Y-drive was modeled in Modelica, where custom models were set up for all Rexroth specific components of the system. Thereby, an incompressible fluid is used. The object diagram is shown in figure 11. Here the hydraulic unit consists of a tank, a pressure source and a three way valve. In order to deal with the large forces inside the system a special kind of cylinder is used and modelled. The flat model consists of 360 equations, while the translated model has 25 state variables and two algebraic loops. The algebraic loops exhibit real as well as discrete variables.
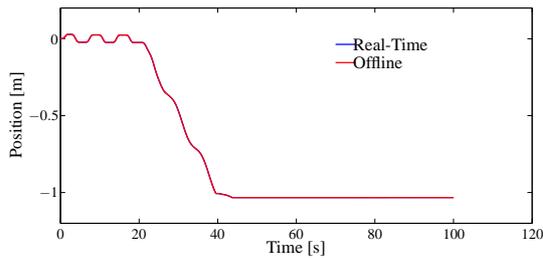
Figure 12: Position of the clamp
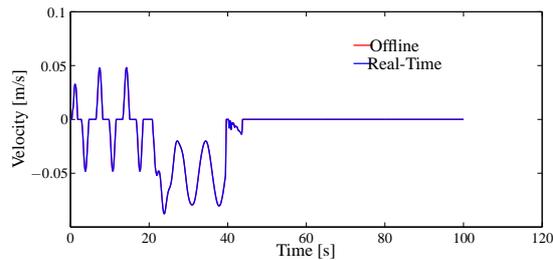
Figure 13: Velocity of the clamp

3.3 Simulation Results

Real-time simulation requires a predictable worst case runtime. Therefore, the number of Newton-iterations in the algebraic loop solver had to be limited. Unfortunately, by doing so it is not guaranteed that a adequate solution is found. Nevertheless, for the used scenario (parameters and inputs) and step size (1ms) a maximum of 4 iterations was required. Hence, the maximum number of iterations was set to 6. The LI3 method described in the previous section was used as numerical integration method. In figure 12 the position of the clamp is shown. The blue line represents the solution computed on the real-time target, while the red line shows the solution computed offline using the C++ Simulation-Runtime and CVode as numerical integration method. It can be seen that the two lines are nearly overlaying. The same holds for the velocity of the clamp shown in figure 13.

4 Conclusion and Outlook

In this contribution the basis for a toolchain for real-time simulation using the OMC is presented. Therefore in section 2 a new C++ Simulation-Runtime was shown that is easy to extend and maintain. Moreover, this Simulation-Runtime includes numerical integration methods, that are suitable for real-time simulation. Due to its flexibility new solvers and algorithms (e.g. multi-rate integration, mixed-mode integration) can be

integrated in the future.

In section 3 the C++ Simulation-Runtime was coupled to the interface of the real-time operating system ScaleRT. That coupling enabled the execution of the C++ Simulation-Runtime together with simulation-code generated by the OMC on a real-time target. The toolchain was demonstrated using a hydro-mechanical heavy duty example system.

In the future this toolchain will be automated, in order to be in the position to generate code for real-time simulation just by a few mouse-clicks. Moreover, coupling of external hardware (e.g. a electronic control unit) is part of future work. This will allow for virtual commissioning using a low-cost toolchain.

5 Acknowledgements

References

[1] http://www.allseas.com/uk/19/equipment/pieter-schelte.html. website. Accessed: 11/07/2012.

[2] www.scale-rt.com. website. Accessed: 11/07/2012.

[3] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Professional, 2001.

[4] M. Arnold, B. Burgermeister, and A. Eichberger. Linearly implicit time integration methods in real-time applications: Daes and stiff odes. *Multibody System Dynamics*, 17(2):99–117, 2007.

[5] W. Braun and B. Bachmann. Symbolically derived jacobians using automatic differentiation-enhancement of the openmodelica compiler. *Modelica Conference, Dresden*, 2011.

[6] F. Casella. Open problems and research trends in oo modelling. Technical report, Politecnico di Milano, Dipartimento di Elettonica e Informazione.

[7] F.E. Cellier and E. Kofman. *Continuous system simulation*. Springer Verlag, 2006.

[8] H. Elmqvist, M. Otter, and F.E. Cellier. Inline integration: A new mixed symbolic/numeric approach for solving differential-algebraic equation systems. 1995.

[9] T. Ersal, H.K. Fathy, D.G. Rideout, L.S. Louca, and J.L. Stein. A review of proper modeling techniques. *Journal of Dynamic Systems, Measurement, and Control*, 130:061008, 2008.

[10] P. Fritzson, P. Aronsson, A. Pop, H. Lundvall, K. Nystrom, L. Saldamli, D. Broman, and A. Sandholm. Openmodelica-a free open-source environment for system modeling, simulation, and teaching. In *Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE*, pages 1588–1595. IEEE, 2006.

[11] E. Hairer, S.P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations: Stiff and differential-algebraic problems*. Springer Series in Computational Mathematics. Springer-Verlag, 1993.

[12] A.C. Hindmarsh and P.N. Brown. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396, 2005.

[13] L. Mikelsons and T. Brandt. Towards a generic vehicle model. *Journal of Computational and Nonlinear Dynamics*, 7:021013, 2012.

[14] N. Nikitin. Third-order-accurate semi-implicit runge-kutta scheme for incompressible navier-stokes equations. *International journal for numerical methods in fluids*, 51(2):221–233, 2006.