Model-based control with FMI and a C++ runtime for Modelica

Rüdiger Franke[1] Marcus Walther[2] Niklas Worschech[3] Willi Braun[4] Bernhard Bachmann[4]

[1]ABB, ruediger.franke@de.abb.com, [2] TU Dresden, marcus.walther@tu-dresden.de,
[3] Bosch Rexroth, niklas.worschech@boschrexroth.de,
[4] FH Bielefeld, {willi.braun, bernhard.bachmann}@fh-bielefeld.de

Abstract

Modelica describes physical systems on a high level, using model objects, multi-dimensional arrays and other data structures as well as graphical representations. Modelica models are translated to differential-algebraic equation systems and compiled to executable code prior to their execution in numerical solvers. The translation gives a lot of possibilities for code optimization. This is particularly important for model-based control applications.

This paper investigates the exploitation of C++ for Modelica code optimization. C++ supports advanced programming concepts and at the same time aims to “leave no room for a lower-level language … (except for assembly code in rare cases)” (B. Stroustrup: The C++ Programming Language, 2014). The features exploited here include polymorphism, templates, built-in exception handling and object destructors.

The ideas have been implemented in the OpenModelica C++ runtime. The paper describes its enhancement with new array features and with an FMI 2.0 interface. FMI serves as interface between modeling tools and control applications. In particular the new FMI 2.0 meets requirements of numerical optimization solvers in model-based control.

A publically available application example demonstrates the achievements. CPU times obtained with the OpenModelica C++ runtime are significantly faster than CPU times obtained with the C runtime or with Dymola.

Keywords: Modelica, OpenModelica, FMI, C++, model-based control, MPC, MHE, SQP, HQP.

1 Introduction

The development of the Functional Mock-up Interface (FMI) was originally driven by automotive industries. The goal was to improve simulation model exchange between component suppliers and OEMs during product development. The FMI standard supports model exchange and co-simulation of dynamic models using a combination of xml-files and compiled C-code (FMI, 2014).

FMI 2.0 for model exchange introduces major enhancements, like sparse model structures and directional derivatives. These enhancements make FMI applicable beyond functional mock-ups for model-based control and optimization as well, evolving it to a Functional Model Interface.

Real-time control applications pose further requirements, like small code size, high quality of generated binaries and fast execution speed. The C++ runtime of OpenModelica is focusing on real-time requirements (Worschech and Mikelsons, 2012). This makes it superior for model-based control applications.

It must be noted that the real-time applications addressed here require cycle times of seconds, sometimes going down to milliseconds or up to minutes. We assume an execution platform with relatively high performance, starting from devices like Raspberry PI and ranging up to distributed server farms. We don’t consider smaller devices with only few kilobytes of memory or lacking floating point arithmetic, because we rate engineering efficiency exploiting high-level technologies like Modelica or C++ more important than extreme hardware savings.

2 Model-based control with mathematical programs

Modelica models are typically used for initial-value simulations. The strict separation of Modelica models from numerical solvers opens further application areas. The models may serve as constraints in mathematical programs as well. Mathematical programming is a technology to solve tasks described with constraints and objective function. This section outlines how Modelica and Mathematical Programming are brought together for model-based control.

2.1 Related work and design rationale

The Optimization Library developed by DLR adds a numerical optimization solver to the Dymola modeling and simulation environment (Pfeiffer, 2012). The focus is on usability, supporting engineering design simulations. A mathematical program is formulated with optimization

DOI
10.3384/ecp15118339

attributes, like bounds or weights, which are added to a readily compiled simulation model using regular Modelica parameter dialogs. The idea of custom attributes has been generalized as custom annotations (Zimmer et al, 2014).

Alternatively to adding an optimization specification on top of a simulation model, there has been an attempt to extend the Modelica syntax with the Optimica language (Åkesson et al, 2010). The approach is conceptionally questionable because it mixes the physical modeling language Modelica with mathematical programs. It is claimed that this improves the treatment of large-scale optimization programs for optimal control. The optimization specific extensions of the Modelica language hinder the re-use of simulation technologies like FMI for optimization. Work basing on Optimica typically involves the development of specific complex tool chains. Results published so far show feasibility, but no advantages over earlier simpler approaches, see e.g. (Lazutkin et al, 2014; Magnusson et al, 2014; Ruge et al, 2014). Recent publications report a convergence towards simpler approaches that re-use simulation technologies, like BLT transformation (Magnusson et al, 2014; Ruge et al, 2014).

The optimization approach used here was developed and first published many years ago. It combines the advantages of optimization formulations using custom attributes with the efficient treatment of large-scale optimization programs for optimal control. A front-end for the optimization solver HQP (Franke and Arnold, 1997) converts optimal control problems formulated for simulation models to large-scale nonlinear programs treated internally. The design rationale was to re-use existing modeling and simulation technologies for optimization, minimizing additional development effort and dependencies on specific tools. The new FMI standard fits well into the long standing design rationale.

Meanwhile there find many industrial applications of HQP, including the control of water canal systems (Wagenpfeil et al, 2014), boom cranes (Neupert et al, 2010) and polymerization reactors (Nagy et al, 2007). HQP has been integrated with the ABB control system and is being applied to the model-based optimal control of power plants worldwide since a decade (Franke and Vogelbacher, 2006; Franke et al, 2008). Recent applications address the real-time optimization of large numbers of renewable power units in virtual power plants and smart grids (Franke et al, 2014).

2.2 Treating model-based control with mathematical programs

Many advanced model-based control applications can be treated as mixed discrete/continuous optimal control problems. Examples include moving horizon estimation (MHE) and model predictive control (MPC).

Discrete-time model equations result from the implementation of control systems on digital computers with cyclically running tasks. They are described with difference equations of the form

$$\begin{gathered}\boldsymbol{x_d(k+1) = f_d[k, x_d(k), x_c(t_k), u_d(k)]},\\ \boldsymbol{x_d(0) = x_{d0}}, \qquad \boldsymbol{k = 0, 1, \dots, K}\end{gathered} \tag{1}$$

Here $x_d(k)$ are the discrete-time states at interval k with the corresponding sample time points t_k, $t_0 < t_1 < \cdots < t_K$. The control inputs $u_d(k)$ are optimized per sample time point. Optimized model parameters can be treated as additional states that are constant and have free initial values.

The continuous-time states $x_c(t)$ describe physical processes, like devices for energy conversion or storage. They are defined with continuous-time differential equations of the form

$$\begin{gathered}\boldsymbol{\frac{dx_c(t)}{dt} = f_c\big[t, x_d\big(k(t)\big), x_c(t), u_c(t)\big]},\\ \boldsymbol{x_c(t_0) = x_{c0}}, \qquad \boldsymbol{t \in [t_0, t_K]}\end{gathered} \tag{2}$$

Numerical solvers generally require the parameterization of continuous-time trajectories $u_c(t)$ with a finite number of control inputs $u_c(k)$, such that $u_c(t) = f_u[t, u_c(k(t))]$. Typically the control inputs describe the control trajectories piecewise constant or piecewise linear.

The optimization has to consider physical and legal limitations that are formulated as constraints of the form

$$\boldsymbol{g\big[t, x_d\big(k(t)\big), x_c(t), u_d\big(k(t)\big), u_c(t)\big] \geq 0} \tag{3}$$

Remaining degrees of freedom are covered with the objective function

$$\sum_{k=0}^{K} f_0\left[k, \begin{pmatrix} x_d(k) \\ x_c(t_k) \end{pmatrix}, \begin{pmatrix} u_d(k) \\ u_c(t_k) \end{pmatrix}\right] \quad \rightarrow \quad \begin{array}{c} \min \\ x_d(0)\ u_d(0) \\ x_c(t_0)'\, u_c(t_0) \end{array} \tag{4}$$

Typical objectives are the minimization of costs or the maximization of results. Multiple objectives can often be expressed monetary and summed up.

2.3 The HQP solver

HQP treats mixed discrete/continuous optimal control problems as large-scale mathematical programs (Franke and Arnold, 1997). Continuous-time differential equations are approximated numerically over given discrete time intervals, either with fixed polynomials or using a variable step size solver. Discrete and continuous-time states and controls are combined into the state vector $x = (x_d, x_c)$ and the control vector $u = (u_d, u_c)$. This gives the discrete-time optimal control problem:

$$J = f_0(x^K) + \sum_{k=0}^{K-1} f_0(x^k, u^k) \quad \rightarrow \quad \min_{x^0,\, u^k}$$

with the discrete-time state equations

$$x^{k+1} = f^k(x^k, u^k), \qquad k = 0, \dots, K-1$$

and the constraints

$$\begin{aligned} & c^k(x^k, u^k) \geq 0, \quad k = 0, \dots, K-1 \\ & c^K(x^K) \geq 0 \end{aligned} \tag{5}$$

The states and the control inputs of all time intervals are collected into one large vector of optimization variables

$$v = (x^0, u^0, x^1, u^1, \dots, x^{K-1}, u^{K-1}, x^K). \tag{6}$$

This results in the mathematical program

$$\begin{aligned} & J(v) \underset{v}{\rightarrow} min && J\colon \mathbb{R}^n \rightarrow \mathbb{R}^1 \\ & h(v) = 0 && h\colon \mathbb{R}^n \rightarrow \mathbb{R}^{m_e} \\ & g(v) \geq 0 && g\colon \mathbb{R}^n \rightarrow \mathbb{R}^m \end{aligned} \tag{7}$$

HQP treats large-scale nonlinear optimization with Sequential Quadratic Programming (SQP). Basing on the Lagrangian

$$\begin{aligned} L(v, \lambda, \mu) &= J(v) - \lambda^T h(v) - \mu^T g(v) \\ L&\colon \mathbb{R}^n \times \mathbb{R}^{m_e} \times \mathbb{R}^m \rightarrow \mathbb{R}^1 \end{aligned} \tag{8}$$

the solution must fulfill the Karush Kuhn Tucker (KKT) conditions

$$\begin{aligned} \nabla_v L(v, \lambda, \mu) &= \nabla J(v) - \nabla h(v)^T \lambda - \nabla g(v)^T \mu = 0 \\ \nabla_\lambda L(v, \lambda, \mu) &= -h(v) = 0 \\ g(v) &\geq 0 \\ \mu &\geq 0 \\ g(v)^T \mu &= 0 \end{aligned} \tag{9}$$

HQP applies Lagrange Newton iterations

$$\begin{aligned} \nabla^2 L(v, \lambda) \begin{pmatrix} \Delta v \\ \Delta \lambda \end{pmatrix} &= -\nabla L(v, \lambda) \\ \begin{pmatrix} v^+ \\ \lambda^+ \end{pmatrix} &:= \begin{pmatrix} v + \Delta v \\ \lambda + \Delta \lambda \end{pmatrix} \end{aligned} \tag{10}$$

to find the solution. The Lagrange Newton iteration is given here for the case *m=0*. HQP augments the Lagrangian to treat inequality constraints with an Interior Point method.

The Hessian of the Lagrangian $\nabla^2 L(v, \lambda, \mu)$ is formed numerically applying a rank 2 update in each time interval basing on the progress over subsequent iterations. This efficient multi-rank update is possible because the discrete-time model equations make the large-scale nonlinear program partial separable. There are only linear couplings between subsequent time intervals. This is also why no analytical second order derivatives are required.

The Jacobian of the Lagrangian $\nabla L(v, \lambda, \mu)$ is obtained by forming partial derivatives and by solving sensitivity equations along with the continuous-time differential equations of the model in each time interval. There exist different solvers, including fixed or variable step size and implicit or explicit.

2.4 Relation to other optimization approaches

The mathematical description given above basically applies to all optimization approaches mentioned in section 2.1. Only the following details differ:

1. The constraints (3) and optimization objective (4) may either be formulated as custom attributes for existing equations or as specific new equations for optimization (see Optimica).
2. The vector of optimization variables (7) contains optimized control variables and state variables. This results in large-scale sparse optimization programs. It has advantages if state constraints are present or for long time horizons. Alternatively the state variables may be hidden, resulting in smaller dense optimization programs (see the Optimization Library).
3. Explicit discrete-time state equations and constraints in (5) lead to partial separability, localizing non-linear terms inside individual time steps (also referred to as multiple shooting).

This gives the ability to apply efficient multi-rank updates to the numerical formation of information about second order derivatives. Alternatively the discrete-time equations may be replaced with nonlinear terms spanning two subsequent time steps (known as collocation and typically applied with Optimica). The lost partial separability may be compensated with analytic second order derivatives.

4. The nonlinear optimization program may be treated with a Newton method (typically used with Optimica basing on actual second order derivatives) or with a Quasi-Newton method basing on numerical updates (also known as SQP method and used with the Optimization Library and here).
5. Finally there finds different methods for the treatment of inequality constraints. Well known approaches include active set methods (see the Optimization Library) and interior point methods used with Optimica and here.

3 FMI for model based control

There exist a couple of different powerful Modelica tools, each having its particular pros and cons. FMI offers the advantage of being tool independent. This makes it possible to exploit the best features of different modeling tools depending on the application at hand. Once a control system can import FMI, it may be used together with any exporting tool.

Even with one and the same modeling tool the runtime code can be exchanged without effecting the optimization solver, e.g. from C code to C++ code as discussed below.

FMI 2.0 for model exchange covers many aspects of hooking a model to an optimization solver, like:

- Initialization mode for steady-state models
- Continuous-time mode for differential equations
- Change of parameter values at runtime
- Directional derivatives for Jacobian evaluation
- ModelStructure defining the sparse pattern in modelDescription.xml
- Variable names, physical units and start values in modelDescription.xml

Two important features are missing in FMI 2.0. Differential-Algebraic Equation (DAE) systems are not covered. They must be converted to an explicit system of differential equations inside the FMU. This is a performance penalty for optimization solvers that may treat DAE constraints themselves. This is why FMI is not suited for models with many algebraic constraints, like network models.

Clocked equations are a powerful mechanism to model discrete systems. Unfortunately the FMI event mode and the ModelStructure do not cover discrete-time states resulting from clocked equations. This makes the treatment of discrete-time models clumsy.

4 OpenModelica

OpenModelica offers an open development process, including published nightly tests and a public discussion panel. This eliminates hidden problems. It ensures a high quality and makes OpenModelica well suited for code export to control applications that shall run 7/24 in possibly safety critical environments.

Moreover OpenModelica has outstanding support for localization. The GUI is delivered with 9 translations and the development environment fully supports UTF-8 for localized doc strings. This broadens the range of possible applications beyond nerds.

OpenModelica enables extensions by third parties for particular needs, like model-based control and real-time applications. Figure 1 gives an overview of the main modules and the data flow in the OpenModelica compiler (OpenModelica, 2014).

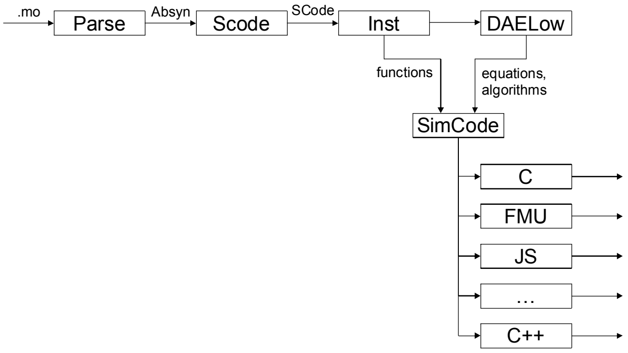

Figure 1: Overview of OpenModelica compiler

The parser generates an abstract syntax tree (Absyn), which is converted to the simplified intermediate code (SCode) and instantiated to a Differential Algebraic Equation system (DAE). The backend DAELow simplifies the equations and algorithms, applies DAE index reduction and brings the equations to the Block Lower Triangular (BLT) form.

The SimCode module applies a template mechanism to generate code for a specific target. There exist multiple code generators, covering the C runtime, FMU export, JavaScript and more. The C++ runtime was developed with particular requirements of real-time simulation in mind (Worschech and Mikelsons, 2012).

DOI
10.3384/ecp15118339

4.1 Common Sub-expression Elimination

Common sub-expressions, like a function called multiple times for the same arguments, may result from object-oriented libraries and minimal connector interfaces.

Take Modelica.Fluid as example. Only pressure *p* and specific enthalpy *h* appear in fluid connectors, besides fluid composition. Connected components may call the same function to obtain the temperature *T(p, h)* on each side of the connection.

A Modelica tool should eliminate common sub-expressions such that they are evaluated only once.

4.2 C++ runtime

Most Modelica tools translate models to C code and simulate them with a runtime written in C as well. OpenModelica offers the possibility to generate simulation code for various languages and runtimes. The default runtime of OpenModelica is also based on C code, but there is a powerful additional runtime written in C++ that can easily be used. By comparing these two runtimes, it can be noticed that the C++ code leads to a higher compilation time, but gives a better runtime performance. Besides that, the C code is less comprehensible and harder to maintain.

Especially features like memory management and exception handling need to be implemented manually in C, messing up the code. Similar code has to be written multiple times, for example to implement arrays of different data types like double, int, bool, string, and records.

C++ addresses many of these issues. It not only has built-in exception handling and object destructors for automated memory management, it also offers templates for an advanced reuse of written code. C++ compilers can instantiate one and the same template multiple times for different data types. Appropriate use of this feature also increases type safety. It may even shift effort from model execution to model translation, making the executable code more reliable and efficient.

While reducing source code size and improving type safety, the additional features of C++ lead to longer compilation times. This restricts its use for interactive sessions. Compilation time is less an issue for online applications, where a model is compiled once and then runs endlessly in the real-time control.

4.3 Arrays

"I have never seen a perfect matrix class. In fact, given the wide variety of uses of matrices, it is doubtful whether one could exist" (Stroustrup, 2014).

Modelica models use multi-dimensional matrices and arrays to a large extent. Unrolling these arrays during model translation is not acceptable, as it leads to large code and long translation times. This is why the simulation runtime needs good support for arrays.

The OpenModelica C++ runtime addresses the wide variety of requirements on arrays with polymorphism. Figure 2 shows the different array classes.

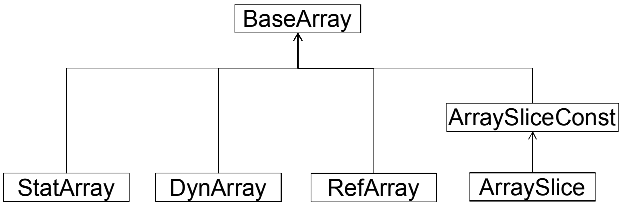

Figure 2: Inheritance diagram of array classes

BaseArray defines a common interface. It is also used as type for array arguments to functions.

StatArray implements that interface with an array of fixed size, known at compilation time. The array data is stored inside the object itself, in order to improve runtime performance.

DynArray can be resized at runtime. It stores array data in dynamically allocated memory. It is typically used in functions that operate on arrays of variable size, like `input Real[:,:] A`.

RefArray is a static array of pointers to simulation variables. This way the array elements may be distributed to optimize performance (see section 4.4 below).

ArraySlice holds a reference to a BaseArray and gives access to a sub-array without necessarily copying the data. It directly maps the Modelica slice syntax to C++, e.g. for `A[1:3,:]`.

ArraySliceConst implements a subset of the functionality of ArraySlice, giving read access only. This is needed for slices of const arrays.

4.4 Performance optimizations with RefArray

The RefArray-type is a simple data structure that can help to improve the performance of large model simulations. Most Modelica tools generate simulation code that stores the array elements consecutive in memory. A solver algorithm is used to calculate all equations respectively equation systems of the model. The execution order of these equations is defined during model translation in the Modelica compiler itself. Because the array variables of the model are often unrolled in the backend of the Modelica Compiler, the single array elements are not solved in the same equation or equation system, but interspersed throughout different equations. This can lead to caching problems, because modern CPU cache memories follow the principle of locality (Denning, 2005). The hardware will automatically prefetch values that are stored besides the memory locations that were used in the last instructions. Thus, the required variables should be stored as dense as possible in memory in order to reduce cache misses and enable efficient computation. With the RefArray-type it is possible to store all variables that are required to solve one equation as dense as possible in memory, because all array elements are references that can be distributed arbitrarily.

4.5 Array storage order

The storage order of multi-dimensional arrays (row major or column major) is generally arbitrary.

Most Modelica runtimes follow the common C convention of row major order. On the other hand most external functions require column major order (in particular LAPACK called from Modelica.Math.Matrices). Thus transposition operations are necessary on each external function call (e.g. when a matrix `A` shall be inverted with `inv(A)` and `inv` calls `LAPACK.dgetri)`.

The C++ runtime stores array data in column major order. External functions can be called without overhead this way. The C++ array implementation hides the storage layout behind its interface. For example the second element of the first row is accessed in Modelica with `A[1,2]` and in the C++ runtime with `A(1,2)`, independent of the internal storage order.

4.6 Mapping of base types

Table 1: Mapping of Modelica base types to C++ types

Modelica type	C++ type
`Real`	`double`
`Integer`	`int`
`Boolean`	`bool`
`String`	`std::string`

Table 1 shows the mapping of Modelica base types to C++. It is assumed that the C++ compiler maps its default base types to the most appropriate and efficient binary representations of the respective hardware platform.

It might be considered a drawback that `int` will typically be 32 bit even on a 64 bit architecture. On the other hand the mapping to standard base types improves platform independence and it simplifies the integration with other software packages, like numerical FORTRAN routines. Moreover note that the IEEE 754 representation of `double` has 64 bits and can treat exact integers with up to 53 bits in a platform independent way.

The C++ language is paired with a powerful standard library. The `std::string` gives the ability to treat a character string like a regular base type. This simplifies the coverage of strings by the C++ runtime.

4.7 Real-time behavior

There are some critical facts that have to be considered for real-time simulations. First of all, most of the program execution time should be spent in user mode and not in kernel mode, to prevent context switches that are expensive and can bloat the simulation time. For the generated simulation code, the most critical part that leads to these kind of context switches is memory allocation. Therefore, the C++ simulation runtime allocates the required memory during initialization and frees it after the simulation run. One exception was described in section 4.3 with the DynArray type, which is rarely used in the evaluated simulation models and thus not a problem for the real time behavior.

Secondly the time integration solver and its event handling are important for real-time simulation. The C++ simulation runtime offers clear interfaces to change the solver and adapt it for real-time criteria. No further details are given here because this was already described in (Worschech and Mikelsons, 2012).

Finally, it should be noticed that C++ object oriented code can lead to small code size and small binaries, which is important for real-time simulation as well.

5 Application example

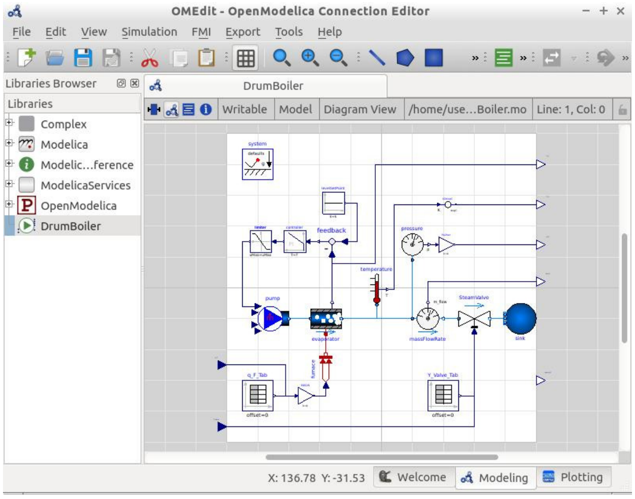

Figure 3: DrumBoiler model in OMEdit

The DrumBoiler example was first introduced in (Franke et al, 2003). Meanwhile it has been added to the Modelica Standard Library and was re-formulated using regular Modelica.Media and Modelica.Fluid. Extended versions of the model, including also once-through boilers, superheaters, reheaters and turbine stages, have been installed in many steam power plants worldwide, optimizing boiler startup control and plant performance.

DOI
10.3384/ecp15118339

The distribution of the optimization solver HQP contains two optimization examples for the basic DrumBoiler model: a steady-state set point optimization and a dynamic start-up optimization. The FMI based examples obtain Jacabians with finite differences so far.

Figure 3 shows the model in the OpenModelica editor OMEdit. The model has three states: drum pressure, liquid water level and integrated error of the feed water controller. The model contains discrete events for flow reversal and control limits. These events are irrelevant here because they are not triggered during the startup optimization.

The objective of the startup optimization is to reach given set points for steam pressure p and flow rate q_m:

$$J = \int_{t=t_0}^{t_f} w^T \begin{Bmatrix} [p_S(t) - p_{ref}]^2 \\ [q_{m,S}(t) - q_{m,ref}]^2 \\ \left[\frac{dq_F(t)}{dt}\right]^2 \end{Bmatrix} dt \rightarrow \min_{u(t)}$$

subject to bounds on the control $u = (q_F, Y_{Valve})$, i.e. fuel flow rate and valve position:

$$0 \leq \quad q_F \quad \leq 500\,MW$$

$$0 \leq Y_{Valve} \leq 1$$

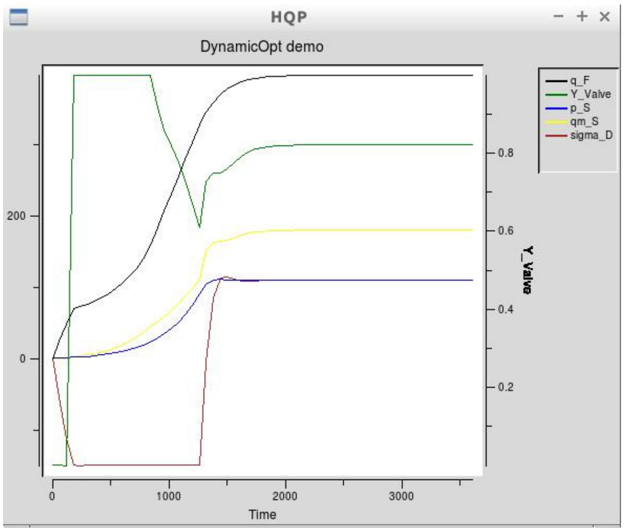

Figure 4: Results of the startup optimization example

rate of change bounds:

$$-24\,MW/min \leq \quad \frac{dq_F}{dt} \quad \leq 24\,MW/min$$

as well as an output constraint on thermal and membrane stress that is a function of drum temperature differences and pressure:

$$-150\frac{N}{mm^2} \leq \ \sigma_{Drum} \ \leq 150\frac{N}{mm^2}$$

$$\sigma_{Drum} \ = 10^{-3}\frac{dT_{Drum}}{dt} + 10^{-5}p_{Drum}$$

The time horizon spans over one hour. It is split into 60 equally spaced intervals with a length of 60 seconds. The control trajectories are parameterized piecewise linear. The continuous-time differential equations are solved in each interval with two fixed steps of 30 seconds applying the Implicit Midpoint Rule (IMP).

Figure 4 shows optimization results. The fuel flow rate and the valve position are controlled such that the constraint on thermal stress is fully exploited while approaching the target operating point. The optimized control trajectories consisting of 60 linear line segments appear smooth in the plot.

5.1 Runtime performance

Table 2: CPU times obtained in a VirtualBox running Linux jessie 3.16, x86_64 on a MacBook Pro Late 2013 with 2.4 GHz Intel Core i5. The gcc version is 4.9.2.

Modelica Tool for FMU export	CPU time with gcc flag		
	-O0	-O2	-Ofast
OpenModelica 1.9.3	16.6 s	15.5 s	13.5 s
OpenModelica 1.9.3 +cseCall	6.0 s	5.5 s	5.2 s
Dymola 2015FD01	3.4 s	1.7 s	1.3 s
OpenModelica 1.9.3 +simCodeTaget=Cpp	5.6 s	1.9 s	1.0 s
OpenModelica 1.9.3 +simCodeTaget=Cpp +cseCall	2.7 s	1.0 s	0.6 s

Table 2 lists CPU times obtained for different FMUs of the same model. The results for the regular OpenModelica C runtime show that the elimination of common subexpressions with the flag +cseCall is crucial for the fluid model. The Dymola results serve as reference.

The C++ runtime is selected with the flag `+simCodeTarget=Cpp`. It uses the same SimCode input as the C runtime and generates C++ code from it. The C++ runtime has its own FMI implementation.

The gcc optimization flag has only minor impact on the OpenModelica C runtime. An improvement by a factor of 2-3 is seen for Dymola C code. The OpenModelica C++ runtime shows a speedup by a factor of 4-6 with compiler optimization. This huge improvement underlines that the higher level expressiveness of C++ is actually exploited by modern compilers.

The CPU times reported here are the average of 10 runs. The deviations between different runs as well as the impact of the virtualization environment on the CPU times are minor. This is important because repeated runs in virtual production environments mark a major use case.

5.2 Reproduction of results

The results reported here can be reproduced on a Posix compliant machine with reasonable development tools installed (on a Debian based system these are the packages: git, gcc, g++, tcl-dev). One might invoke the commands:

```
  $ git clone https://github.com/omuses/hqp.git
  $ cd hqp
  $ ./configure
  $ make
  $ cd odc
  $ ./run drumboiler
```

The last command evaluates drumboiler.tcl. This file contains the optimization specification and all solver settings. They are equal, no matter how the FMU was generated.

The Tcl script calls the OpenModelica compiler omc with default settings to generate an FMU from the simulation model DrumBoiler.mo. Alternatively the FMU can be generated separately and copied to the odc directory before running the optimization.

Note that the first run with a newly generated FMU contains an unzip operation. Each run parses the file modelDescription.xml. The impact of the XML parser vanishes if multiple runs are performed in one process. The average time of ten runs in one process is obtained with:

```
$ ./odc
% time {source drumboiler.tcl} 10
```

6 Conclusions

FMI 2.0 for model exchange provides an efficient interface for hooking simulation models to optimization solvers and running model-based control applications. The approach discussed in this paper offers several advantages over alternative optimization approaches. The standardized FMI hides details of particular modeling tools or components thereof, enabling innovations without comprising a whole tool chain. Several Modelica tools support FMI.

OpenModelica offers a modular environment that can be customized and, thanks to the open source setup, further developed for particular needs.

The default C runtime is a good compromise between fast compilation speeds and high runtime performance. It is suited for interactive modeling and simulation sessions. It has limitations for model-based control applications though. Especially the garbage collection can produce issues.

The C++ runtime is particularly developed for real-time simulation. It exploits object destructors for deterministic memory management. C++ has a rich syntax to express programming concepts on a high level. This not only improves readability by humans, it also enables more code optimization by C++ compilers. Exploiting templates, the amount of manually written code is minimized and type safety is increased. For instance an array class only needs to be implemented once for arbitrary types of array elements. This boosts development efficiency and reduces the probability of bugs.

Some missing features were added to the C++ runtime throughout the work reported here. In particular the existing FMI 1.0 export was upgraded to FMI 2.0 and some issues were solved in the OpenModelica backend for FMI export. The array implementation was enhanced, an external FORTRAN interface was added. The array storage order was changed from row major to column major to minimize the overhead when calling external functions, like LAPACK functions from Modelica.Math.Matrices.

The OpenModelica development process with nightly tests and public issue tracking helped significantly. It provides immediate feedback on the progress made and possible negative side effects.

As a result the C++ runtime is applicable to model-based control using FMI 2.0 for model exchange along with the widely used optimization solver HQP. A speedup of up to 8 is seen with gcc optimization flags. An example shows an FMU exported with the C++ runtime performing significantly faster than the FMU exported with the C runtime or with Dymola.

The price to pay with C++ is longer compilation times. This is less an issue for online control applications, where a model is compiled once and then runs endlessly in the real-time control.

Another possible drawback of C++ is stronger coupling between compilation modules, as required for improved type safety, performance, and exception handling. These things are hidden behind FMI.

FMI needs to be further developed towards supporting DAE constraints, e.g. arising from network models, and discrete states arising from clocked equations. The OpenModelica C++ runtime offers a promising basis for this future work.

Acknowledgements

This work was supported in parts by the Federal Ministry of Education and Research (BMBF) within the ITEA2 project MODRIO (Model Driven Physical Systems Operation) – BMBF funding code: 01IS12022A.

References

P. J. Denning: The locality principle, Communications of the ACM - Designing for the mobile device, July 2005.

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

R. Franke, M. Rode, K. Krüger: On-line Optimization of Drum Boiler Startup, 3rd International Modelica Conference, 2003. https://www.modelica.org/events/Conference2003/papers/h29_Franke.pdf

R. Franke, L. Vogelbacher. Nonlinear model predictive control for cost optimal startup of steam power plants. at – Automatisierungstechnik, 54(12):630–637, 2006.

R. Franke, B.S. Babji, M. Antoine, A. Isaksson: Model-based online applications in the ABB Dynamic Optimization framework, 6th International Modelica Conference, Bielefeld, March 3-4, 2008.

R. Franke, S. Saliba, A. Frick: Virtual Power Plants for Smart Markets, PowerGen Europe, Cologne, June 2014.

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

E. Lazutkin, A. Geletu, S. Hopfgarten, P. Li: Modified Multiple Shooting Combined with Collocation Method in JModelica.org with Symbolic Calculations, Proceedings of the 10th International ModelicaConference March 10-12, 2014, Lund, Sweden. http://www.ep.liu.se/ecp/096/104/ecp14096104.pdf

F. Magnusson, K. Berntorp, B. Olofsson, J. Åkesson: Symbolic Transformations of Dynamic Optimization Problems, Proceedings of the 10th International ModelicaConference, Lund, Sweden, March 10-12, 2014.

Z.K. Nagy, B. Mahn, R. Franke, F. Allgöwer. Evaluation study of an efficient output feedback nonlinear model predictive control for temperature tracking in an industrial batch reactor. Control Engineering Practice, 15(7):839 – 850, 2007.

J. Neupert, E. Arnold, O. Sawodny, and K. Schneider: Tracking and anti-sway control for boom cranes. Control Engineering Practice, 18(1):31–44, 2010.

OpenModelica System Documentation, February 2014.

A. Pfeiffer: Optimization Library for Interactive Multi-Criteria Optimization Tasks, Proceedings of the 9th International Modelica Conference, September 3-5, 2012, Munich, Germany. http://www.ep.liu.se/ecp/076/068/ecp12076068.pdf

V. Ruge, W. Braun, B. Bachmann: Efficient Implementation of Collocation Methods for Optimization using OpenModelica and ADOL-C, Proceedings of the 10th International ModelicaConference, March 10-12, 2014, Lund, Sweden.

B. Stroustrup: The C++ Programming Language, Fourth Edition, Addison-Wesley Pearson Education Inc., 2014.

J. Wagenpfeil, E. Arnold, H. Linke, O. Sawodny: Modeling and optimized water management of artificial inland waterway systems. Journal of Hydroinformatics, 15(2):348–365, 2013.

N. Worschech, L. Mikelsons: A Toolchain for Real-Time Simulation using the OpenModelica Compiler, 9th International Modelica Conference, Munich, 2012. http://www.ep.liu.se/ecp/076/086/ecp12076086.pdf

D. Zimmer, M. Otter, H. Elmqvist, G. Kurzbach: Custom Annotations: Handling Meta-Information in Modelica, Proceedings of the 10th International ModelicaConference, March 10-12, 2014, Lund, Sweden. https://www.modelica.org/events/modelica2014/proceedings/html/submissions/ECP14096173_ZimmerOtterElmqvistKurzbach.pdf

J. Åkesson, K.-E. Årzén, M. Gäfvert, T. Bergdahl, H. Tummescheit. Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problems. Computers and Chemical Engineering, 34(11):1737–1749, November 2010.