Tool Support for Modelica Real-time Models

Michaela Huhn[1], Martin Sjölund[2], Wuzhu Chen[1], Christan Schulze[1], and Peter Fritzson[2]
[1]Clausthal University of Technology, Department of Informatics
Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Germany
[2]Linköpings Universitet, Dept. of Computer and Information Science
SE-581 83 Linköping, Sweden

Abstract

The challenges in the area real-time simulation of physical systems have grown rapidly. To prepare a simulation model for execution on a real-time target, an experienced developer usually performs several adaptations on the model and the solver in order to reduce runtime and communication needs.

Two-folded tool support for evaluating the effect of such adaptations is presented here: (1) A *ModelComparator* for the systematic comparison of simulation results from different versions of the model and (2) an *RT-Profiler* for measurements and analyses of function calls during RT simulations. The ModelComparator facilitates verification of a model adapted for real-time execution to ensure that it will produce sufficiently accurate results at selected operation points. The RT-Profiler takes the specific code structure of simulation models into account when measuring execution times. It directs the developer to those parts that are most promising for model adaptations.

We consider OpenModelica and SimulationX as modeling and code generation frameworks for real-time simulation. The procedure of model adaptations and the use of the analysis tools therein are exemplified in small case studies.

Keywords: simulation, Modelica, RT-profiling, optimization, hardware-in-the-loop

1 Introduction

Modeling and simulation has become an essential part of the design process in mechanical engineering and mechatronics. The object-oriented language Modelica is widely accepted for physical modeling in many industries. Simulation has been applied for analysis and validation in the concept and rapid prototyping phase for a long time, but nowadays simulation models are re-used in latter design phases for testing and verification or even as part of the running system. The potential of simulation models in later design phases is to partially substitute costly physical components or complicated conditions of the surroundings that are required to verify the control of a complex system within its mechatronic environment. However, these usages are most often based on online simulation. Hence, simulation models need not only reflect the physics of the modeled components properly from a functional viewpoint, but also with respect to their timing behavior. Thus, the interest in real-time simulation is increasing rapidly.

Prominent usage scenarios for real-time simulation of physical systems are Rapid Control Prototyping (RCP) and Hardware-in-the-Loop (HIL)[12] where the real-time simulation substitutes mechatronic components during detailed design and verification of embedded controllers. Another usage is Model Predictive Control (MPC)[12] where the model becomes a part of the controller used for predicting the short term behavior of a physical component.

The challenge of simulating a Modelica model as part of a real-time system is to meet the timing constraints and meanwhile keep the result accuracies and resource consumption in a acceptable range. Especially when simulating in a hard real-time (HRT) context, any violation of timing constraints during the simulation may cause a fatal failure of the whole system. Since the HRT case is more critical than a soft real-time (SRT) case, it is worth the subject in this paper.

We present two tools to assist the modeler with the validation and verification of real-time simulation models which may be used independently from any specific Modelica framework: The *ModelComparator* aims for verifying whether a model optimized for real-time execution will operate with an acceptable accuracy by comparing its outputs to a reference model. On the other hand, the *RT-Profiler* will aid the modeler

to understand better the timing behaviors and the internal complexity of the model. In profiling the code for measurement of execution times, call frequencies or resource consumptions will be inserted into the model at places most interesting, which is known as instrumentation. As a consequence, instrumentation will add extra instructions to the model and hence increase the total execution time of it. So in general, the overhead caused by instrumentation should be kept in a minimal extent.

The rest of the paper is structured as follows: In Sec. 2 some techniques for adapting models for real-time simulation are sketched. In the Sec. 3 the *ModelComparator* is presented, whereas *RT-Profiler* in described in Sec. 4. Sec. 5 illustrates tool usage in small case studies. Section 6 concludes.

2 Strategies to adapt a model for real-time execution

A key characteristic of HRT systems is time-determinism. Thus, the first requirement on simulation models for real-time execution is to ensure predictable timing behaviors. The second requirement is that the simulation model shall react as fast as the original (physical) system would do, which means that the execution time must not exceed the step size.

Several Modelica simulation environments are capable of exporting Modelica models for real-time simulation already, to name but a few, SimulationX®, Dymola®, OpenModelica, etc. In many cases, models for real-time simulation are derived from off-line simulation models developed in an earlier design phase. Such adaptations on a model are called real-time optimization (RTO) of the design model. Some parts can be automated, but due to the diversity of model domains and simulation goals, model engineers commonly need to manually optimize the models to enhance time determinism and performance for RT simulations [2].

The numerous variants to be performed on off-line models can be roughly categorized into two groups:

- RTO by adapting the system behaviors
- RTO by mathematical reduction of complexity

2.1 RTO by adapting the system behavior

Hybrid dynamic systems can be adequately described as a set of nonlinear differential algebraic equations (DAEs) having a general implicit form as:

$$0 = f(x, \dot{x}, y, z, u, p, t) \tag{2.1}$$
$$0 = g(x, y, z, u, p, t) \tag{2.2}$$

with

x	continuous state variables
$\dot{x}$	time derivative of x
y	outputs of the system
z	discrete state variables
u	inputs of the system
p	parameters
t	time

where equation (2.1) gives the evolutionary rule of the state variables inside the dynamic system and equation (2.2) implies the algebraic constraint of the system. According to *Implicit Function Theorem*, assuming $D_y g$ (Jacobian of g) is not singular, $\exists \phi(x, z, u, p, t) = y$. Inserting this result into equation (2.2) then gives the following DAEs:

$$0 = f(x, \dot{x}, \phi(x, z, u, p, t), z, u, p, t) \tag{2.3}$$
$$0 = g(x, \phi(x, z, u, p, t), z, u, p, t) \tag{2.4}$$

However, the above mentioned equation system is still in an implicit form. Symbolic analysis and possibly associated manipulations for the index reduction need to be performed on the DAE system to produce an explicit ODE system. Although this can be automatically processed either by commercial Modelica compilers or free ones, some non-linear implicit relations often remain on the RHS of the ODE system. These mathematically complex interdependencies between state variables cause algebraic loops that significantly contribute to execution times during simulation. Breaking up algebraic loops, i.e. decoupling state variables, decomposes the system model into a set of subsystems that is computationally simpler to handle. We briefly sketch some strategies for decoupling:

The most straightforward approach is to eliminate state variables which are of minor relevance to system dynamics, like for instance the fluid temperature in a 1-D momentum equation. Library elements offering different model variants of the same physical entity support the modeler with this strategy: The variants, which may be selected via parameters, may be optimized for the calculation of different sets of input/output variables. Such variants may exploit specific solvers for subproblems or differ wrt. numerical

accuracy or completeness of the physical effects being modeled. All this will result in differences in the computational complexity and the real-time behavior.

As described in [3], knowledge on weak dynamic interactions between system parts may be used to decouple the system by introducing pre-announced weak dynamic state variables which are solved by implicit integration. Such a decoupling leads to strong improvements on the Block Lower Triangle (BLT) transformation, since the off-diagonal entries will decrease inside the BLT structure.

An another alternative is the mixed-mode integration approach described in [11], which tries to partition an entire dynamic system into separate fast/slow subsystems by analyzing the eigenvalues of the system. In contrast to weak dynamic decoupling, mixed-mode integration can be fully automated.

It has to be noted that all these approaches require a modeler with strong backgrounds on the problem domain, as they are only applicable under certain constraints, e.g. a loose coupling system, a reasonable difference between subsystem dynamics, etc.

2.2 RTO by mathematical reduction of complexity

A system's complexity is characterized by the number of its state variables (or number of dimensions of the model). Some state variables de facto dominate the dynamic behavior of the system, which gives us a possibility to reduce the complexity of the system by disregarding some subordinate ones. Provided that a DAE system has been transformed into the following continuous generalized state-space form:

$$\mathbf{E}\dot{\boldsymbol{x}} = \mathbf{A}\boldsymbol{x} + \mathbf{B}\boldsymbol{u} \qquad \text{(state equation)} \tag{2.5a}$$

$$\boldsymbol{y} = \mathbf{C}\boldsymbol{x} + \mathbf{D}\boldsymbol{u} \qquad \text{(output equation)} \tag{2.5b}$$

with $\boldsymbol{x} \in \mathbb{R}^n$ (state variables), $\boldsymbol{u} \in \mathbb{R}^m$ (inputs), $\boldsymbol{y} \in \mathbb{R}^l$ (outputs), $\mathbf{E} \in \mathbb{R}^n \times \mathbb{R}^n$ (descriptor matrix), $\mathbf{A} \in \mathbb{R}^n \times \mathbb{R}^n$ (system or state matrix), $\mathbf{B} \in \mathbb{R}^n \times \mathbb{R}^m$ (input matrix), $\mathbf{C} \in \mathbb{R}^l \times \mathbb{R}^m$ (output matrix) and $\mathbf{D} \in \mathbb{R}^l \times \mathbb{R}^m$ (feed-through matrix).

In (2.5), if the number of state variables is very large compared to the number of inputs and outputs ($n \gg l, n \gg m$), it implies that redundancies might exist in the system. Hence, there is a chance to to come up with an alternative system model not only with much lower dimensions but also with adequate accuracies and the preservation of important system properties. This technique is called *model order reduction* (MOR).

A brief introduction of the so-called projection-based model oder reduction (PBMOR) will be given here to show the philosophy of MOR. The PBMOR can be roughly performed in three steps:

1) Choice of ansatz:

The state variables $\boldsymbol{x} \in \mathbb{R}^n$ is approximated by $\boldsymbol{x}_q \in \mathbb{R}^q$ ($q \ll n$):

$$\boldsymbol{x} \approx \boldsymbol{x}_q = \sum_{j=0}^{q} \boldsymbol{v}_j \xi_j = \mathbf{V}_q \xi \tag{2.6}$$

where $\boldsymbol{v}_j$ is the basis spanning a subspace $\mathscr{V}_q = \text{span}\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_q\}$ called ansatz space of the reduced model, $\xi \in \mathbb{R}^q$ and the matrix $\mathbf{V}_q = [\mathbf{v}_1, \ldots, \boldsymbol{v}_q]$ ($q \ll n$). The choice of decent basis $\boldsymbol{v}_j$ plays a significant role in PBMOR, there exist many methods on the market [8, 5].

2) Insert the ansatz:

Inserting the ansatz (2.6) into (2.5) delivers the following over-determined system:

$$\mathbf{E}\mathbf{V}_q\dot{\xi} = \mathbf{A}\mathbf{V}_q\xi + \mathbf{B}\boldsymbol{u} \tag{2.7}$$

$$\boldsymbol{y} = \mathbf{C}\mathbf{V}_q\xi + \mathbf{D}\boldsymbol{u} \tag{2.8}$$

As there are more equations than unknowns in this reduced model representation, in general the residual of the system will not equal zero.

3) Projection of the residual:

The residual is projected onto a subspace $\mathscr{W}_q$ spanned by the columns of so-called weighting factors $\mathbf{W}_q$.

$$\mathbf{W}_q^T\mathbf{E}\mathbf{V}_q\dot{\xi} = \mathbf{W}_q^T\mathbf{A}\mathbf{V}_q\xi + \mathbf{W}_q^T\mathbf{B}\boldsymbol{u} \tag{2.9a}$$

$$\boldsymbol{y} = \mathbf{C}\mathbf{V}_q\xi + \mathbf{D}\boldsymbol{u} \tag{2.9b}$$

Thus, an optimal solution of the over-determined system can be attained in a least square sense. The input-output properties of the reduced model described in (2.9) depends only on matrices $\mathbf{V}_q$ and $\mathbf{W}_q$.

2.3 Solver

In a real-time setting the well-established Euler Forward solver is considered most suitable for time integration. As an explicit fixed step solver it guarantees a limited number of calculations and thereby predictable execution times. Moreover, it offers acceptable inaccuracies near discontinuities [2]. If the system contains any non-linear behavior, then iterative solvers have to be employed for solving nonlinear equations. As a matter of fact, this will lead to unpredictable execution time, which is not preferable in real-time simulation. The execution time when applying an iterative solver depends on the number of iterations needed for

the solution to converge. Setting up an upper bound for the iteration number will be a work-around to ensure the predictable timing behavior, but such treatment might induce numerical inaccuracies and needs to be taken in to account carefully. Thus, the two central issues in real-time simulation, timing behavior and accuracy, are reflected by the *ModelComparator* and *RT-Profiler* in this paper.

3 Comparing model variants

In many cases, several RTO techniques have to be applied on a compound model to finally meet the real-time constraints. However, simulating in time is just a necessary but not a sufficient condition for RT simulations, because simulation results with instabilities or large deviation errors will not make any sense. So after having performed several RTOs on a model, the quality of simulation results has to be assured. This can be done with the help of the *ModelComparator* by comparing the simulation results between the original and optimized models. Although many Modelica simulation environments - like SimulationX, Dymola and OpenModelica - already offer the functionality for comparing results for variants, these are limited in that all variants must be modeled and simulated in the same framework. The *ModelComparator* has however surmounted this limit and it is capable to compare results from different frameworks. If the new Functional Mock-up Interface (FMI) [1] standard will be widely adopted by Modelica tool providers for model export, the *ModelComparator* can be further extended for direct manipulation of the model on source code level.

Moreover, the *ModelComparator* helps the developer to make a choice between a model leaning more towards the RT performance and a model leaning more towards the accuracy in a concrete application context, for instance, the anti-lock breaking system employed in automotive industry may prefer a more rapid response of the RT model for HIL testing.

The *ModelComparator* is an application that was developed with Java in Eclipse IDE with GUI support, its outputs are shown in Fig. 8 and 14. The user may load different models and select for each of them variables that will be plotted and compared to each other. As it allows users to handily modify the parameters for simulation and solver settings via the GUI, it is very useful when carrying out a parameter optimization, too. Another feature of *ModelComparator* is to automate series of simulations and analyze them wrt. thresholds: The user may specify threshold values before starting the simulation that indicate in some sense exceptional behavior. The *ModelComparator* filters the results and directs the user conveniently to those instances of a series of simulations where the values are exceeding the threshold. Currently it supports exported model from SimulationX under Windows, but the scope of environment support will be extended.

4 RT-Profiling for simulation models

The purpose of profiling in the context of real-time simulation is two-folded: (1) RT-profiling is a means to verify whether an optimized model satisfies its real-time requirements and (2) RT-profiling shall allow a detailed analysis to determine those parts of the model causing the major portion of the computational load. Thus, RT-profiling shall support the modeler to identify the primary candidates in further optimization and verification steps. Hence a RT-profiler tailored to the specific code structure of simulation models is preferred to general purpose profiling tools like *prof* [6].

The structure of the C code generated from simulation models was analyzed and the concepts for instrumenting the code for RT-profiling were introduced in [12]. The profiling was performed on the proprietary C code from SimulationX for the assessment of RT performance on RT target SCALE-RT® 5.1.4 [7]. However, the concepts can be transferred to other frameworks by adopting the instrumentation accordingly.

In general, source code automatically generated from simulation models has a flat structure for which profiling, i.e. an statistical evaluation of call frequencies and execution times of functions, seems to be sufficient. A model commonly consists of an initialization and a simulation phase. Each phase is split into global solver steps. In a global solver step a series of integration steps are performed followed by the calculations of the output variables. To guarantee the timely delivery of results, either a fixed step solver is employed or the number of iterations is limited[1]. Within each part, external functions may be called. In case the algebraic loops cannot be solved analytically, a local solver will numerically compute the solution. So, a global solver step can be described by:

- $n_I \cdot$ integration steps
  - $e_I \cdot$ external function calls
  - $c_I \cdot$ additional calculations
  - $a_I \cdot$ (non-)linear blocks

[1] with the well-known consequences on accuracy

  - $*$ $e_{aI}\cdot$ external function calls
  - $*$ $c_{aI}\cdot$ additional calculations

- $1\cdot$ output of variables
  - $e_O\cdot$ external function calls
  - $a_O\cdot$ additional calculations

To analyze the allocation of execution time within a global step the profiling will measure the execution time for each (non-)linear block and each external function call, as well as for each integration step as a whole and the calculation of outputs. The values are stored separately, and in a post processing step average, variance and the maximum, which is most important for verification of hard real-time requirements, are calculated.

4.1 RT-Profiling SimulationX models

The C code generated by SimulationX from a simulation model is instrumented in a post processing step as follows: Whenever entering or leaving a function or block a time stamp is recorded and a counter is incremented. As the generated C code is well structured, automation of code instrumentation is straightforward. After determining the tokens at which instrumentation has to be placed, template-based code transformation can be realized easily. We used ANTLR [9] and cc65 [13] as two alternatives for this tasks.

4.2 Implementation on a RT Target

Whereas in [12] version 4.1.2 of Scale-RT was used, we now have moved to the current version Scale-RT 5.1.4 which offers improved support for simulation models by providing a framework that automatically embeds a model in a kernel module that iterates the global solver steps. It was already observed in [12] that storing and evaluating the profiling data significantly contributes to execution time. Thus it should not be done by the task executing the model in the Scale-RT real-time kernel, but by a task running in the (non-real-time) user space. Consequently, we applied the producer-consumer pattern and implemented the profiling as two task communicating via a FIFO-buffer (see Figure 1).

For each global solver step of the model, the profiling methods record the following information:

- execution time and frequency of an integration step
- execution time and frequency of each external function call within the integration step that does not reside inside a (non-)linear block
- execution time and number of loops of each (non-)linear block within the integration step
- execution time and frequency of each external function call within each (non-)linear block
- execution time of outputting variables at the end of the current step

4.3 RT-profiling OpenModelica models

The OpenModelica implementation of the code instrumentation was done in the compiler itself, with only slight modifications. The instrumentation is performed by compiling a model with a preprocessor macro set, and running the executable with the time measurement flag. The time measurement uses the real-time clock available on the platform used. All measurements are accumulated using integer math and output at each time step.

The code runs on all platforms supported by OpenModelica and is not limited to RT systems. Profiling is of general interest because small changes to a model may have a large impact on the simulation time. By providing a profiler to both developers and users of a simulation tool, performance issues can more easily be discovered. As a result it should be easier to improve the quality of the tool. By observing the output of the profiler, you can see that in the `SimpleNonLinear` example (Listing 1), `sin(x)` will be called 3 times in each time step. While it is possible to determine the value of `x` during compile-time or initialization, OpenModelica does not yet perform these optimizations.

Listing 1: Simple non-linear equation

```
class SimpleNonLinear
  Real x = cos(x);
end SimpleNonLinear;
```

The profiling also works for any user-defined function that is called. In the `ArrayCall` example (Listing 2), `tenCos` is called 10 times because arrays were not handled properly by the compiler in this case. This means `cos` is called 10^2 times in every timestep. If the function is inlined, `cos` is only called 10 times instead.

Listing 2: Binding equation is an array

```
class ArrayCall
  function tenCos
    input Real r;
```

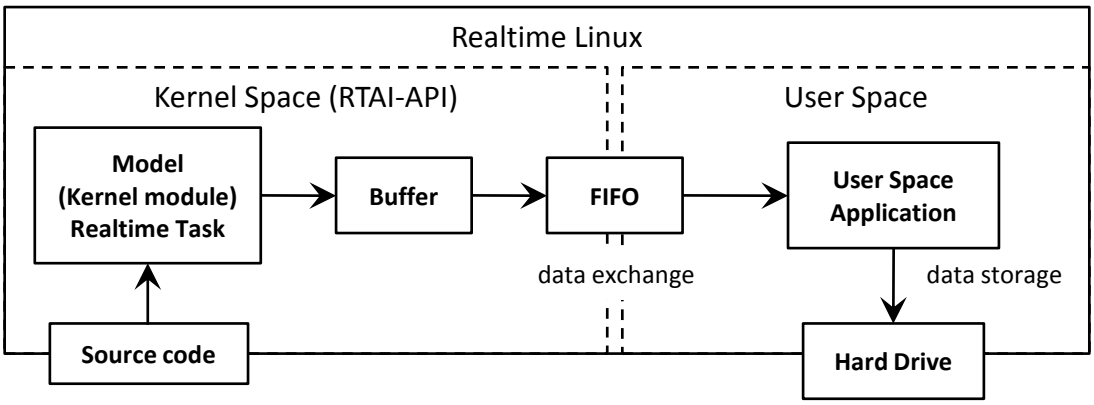

Figure 1: Communication between user task and model task

```
    output Real array[10];
  algorithm
    array := cos(r*(1.0:10.0));
  end tenCos;

  Real x[10] = tenCos(time);
end ArrayCall;
```

The output only contains the number of the equation blocks and names of functions. In the future, this information will be augmented with the names of variables defined by the block and the line numbers where those variables originate from. It is also of interest to add the lines of the equations involved in solving the block since it is hard for a user to understand which of his equations caused a particular error. This is related to the general problem of bug localization in debugging equation-based models [10].

Much of this information is present in the OpenModelica backend, but only part of it is present in the generated code. Once the information is present in the code, the runtime system should be able to generate a detailed report. In the code generated by SimulationX information is present as well, however, it would be the task of the instrumentation to extract and relate it to the profiling results.

4.4 Mapping Profiling Results to Model Positions

Furthermore, additional work is planned for improving the final report that the profiling gives. For example, when displaying the time spent in a non-linear system of equations, the tool should also report the variables involved, what line of code they are defined in. When possible, the tool should also display the line numbers where the original equations were defined.

Much of this information is already available in the OpenModelica Compiler backend but is not yet propagated all the way into the source code. In the code generated by SimulationX information is present as well, however, it would be the task of the instrumentation to extract and relate it to the profiling results. Adding this information as part of the runtime environment is also of a more general interest to a user since it is hard for a modeler to try and understand which of the equations caused a particular runtime error.

This is related to the general problem of bug localization in debugging equation-based models [10].

5 Case Studies

The case studies presented in this section are going to show the procedures of carrying out RTO on original models, testing model RT performance and validating the optimized models with facilities from RT-Profiler and ModelComparator. It is also shown here how the results from RT-Profiler can guide developers to perform adaptations on design model for a better RT performance. However, the design, modeling and code exporting phases from physical dynamic systems to RT models are not covered in this paper.

5.1 Case 1: Electric circuit with saturating inductors

The first case is a simple electric circuit with inductors showing non-linear behaviors (due to the saturation effect of ferromagnetic materials): It is taken a variant of the basic components from the Modelica Standard Library 2.2.1 [4]. The saturation of an inductor is approximately described by a non-linear function relating the actual inductance with the changes in drive current. The Modelica model is shown graphically in Figure 2, where A and B are the observation points.

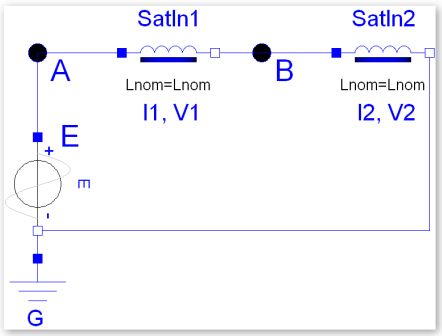

Figure 2: Simple circuit with saturating inductors

5.1.1 System Description

In this model, a time-dependent sinusoidal voltage source E and two nonlinear inductors SatIn1 and SatIn2 are connected in series. By providing the necessary parameters, e.g. nominal inductance L_{nom}, nominal current I_{nom}, inductance near zero current L_{zer} and inductance at large current L_{inf}, the actual inductance L_{act} can be calculated via $L_{act} = f(I(t))$, where $I(t)$ is the current flowing through the inductor, $f : \mathbb{R} \mapsto \mathbb{R}$ is a nonlinear mapping. From Maxwell Equations it is known that the magnetic flux $\Phi^{(1)}$ and $\Phi^{(2)}$ of the two inductors can be computed through:

$$\Phi^{(1)} = L_{act}^{(1)} \cdot I_1 = f^{(1)}(I_1) \cdot I_1$$
$$\Phi^{(2)} = L_{act}^{(2)} \cdot I_2 = f^{(2)}(I_2) \cdot I_2$$

After manually performing an electric circuit analysis, the following DAEs are obtained:

$$V_A = E \qquad \text{Voltage of the source} \tag{5.1a}$$
$$V_A = V_1 + V_2 \tag{5.1b}$$
$$V_1 = der(\Phi^{(1)}) = der(f^{(1)}(I_1) \cdot I_1) \tag{5.1c}$$
$$V_2 = der(\Phi^{(2)}) = der(f^{(2)}(I_2) \cdot I_2) \tag{5.1d}$$
$$I_1 = I_2 \tag{5.1e}$$

5.1.2 Nonlinearity of the System

The DAE system (5.1) contains highly nonlinear behaviors due to the relation between voltage and current within the saturating inductors. Moreover, the algebraic constraint on V_1 and V_2 forces the system to stay as a holistic system. In order to solve this monolithic (non)linear block, a numerical solver has to be called iteratively, which leads to excessive and indeterministic computation time. A workaround to handle the non-determinism is to limit the maximal number of iterations, but this will sometimes lead to numerical instability. However, the main issue is that it depends on the tool whether such a workaround is possible or not. So the timing behavior for solving the (non)linear block are of a great interest in RT profiling.

Applying the profiling tool on the model in SCALE-RT 5.1.4 [7] real-time environment yields the results given in the following figure:

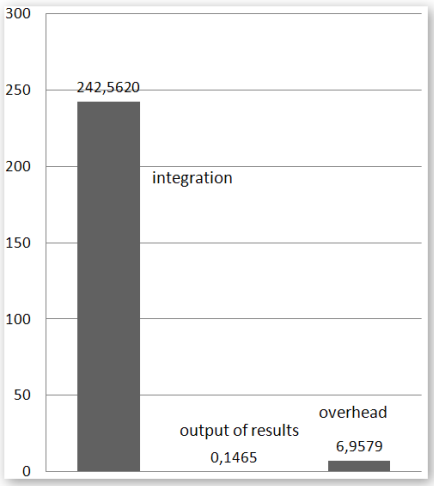

Figure 3: Workload in global steps

It is noticed from Figure 3 that the model runtime of a global solver step is dominated by the number of integration steps and the execution time of each integration step. In this RT simulation, the 10 integration steps contribute more than 97% of the total work load. The time for each integration, 24.2562 μs, is taken as an average of 10 second simulation results. Time for outputs, 0.1465 μs,is relatively small in this case and the overhead caused by auxiliary operations, 6.9579 μs, is small as well. Time for outputs and overhead are also averages of all measured results.

An insight of the workload in each integration step can be achieved by tracing down to the generated source code and through results from the profiling results of each integration step. This is given in Figure 4:

The average execution time for the nonlinear block is 3.7001 μs and in each integration step the nonlinear solver is called 4.4281 times on the average, so the total time for solving the nonlinear equations is $\tau_{tot\_non} = 16.3844\ \mu s$. There still exists a linear system after the solution of the nonlinear one is obtained, which is also solved iteratively. The average execution time and the number of loops are 3,5555 μs and 1.999 times, respectively, in each integration step, which leads to a total time $\tau_{tot\_lin} = 7.1106\ \mu s$. Compared to the total time for solving nonlinear equations, τ_{tot_lin} is less than half of τ_{tot_non}. Thus, the monolithic

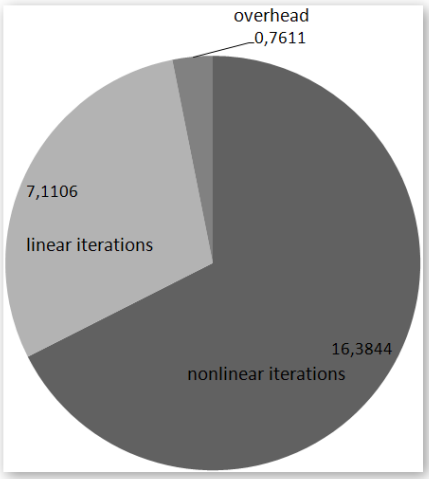

Figure 4: Workload in integration steps

(non)linear block might be a candidate of bottlenecks in the RT model and needs to be dealt with.

5.1.3 RTO by Introducing Capacities

As described above, the nonlinearity of the original system causes computationally expensive algebraic loops during the integration steps. The execution times for solving the (non)linear blocks are more significant than those for other operations. In order to break down the large nonlinear system into smaller subsystems, a capacitor can be introduced into the original system as illustrated in Figure 5. As a result, the original DAE

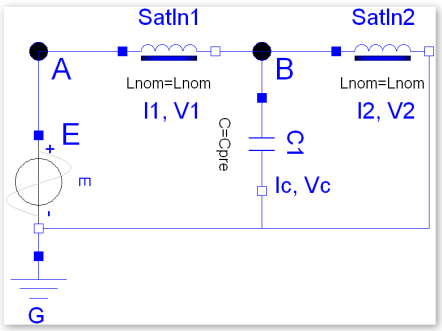

Figure 5: Optimized simple circuit

system (5.1) is transformed as follows:

$$V_A = E \tag{5.2a}$$
$$V_A = V_1 + V_C \tag{5.2b}$$
$$V_1 = der(\Phi^{(1)}) = der(f^{(1)}(I_1) \cdot I_1) \tag{5.2c}$$
$$I_C = C \cdot der(V_C) \tag{5.2d}$$
$$I_1 = I_2 + I_C \tag{5.2e}$$
$$V_2 = V_C \tag{5.2f}$$
$$V_2 = der(\Phi^{(2)}) = der(f^{(2)}(I_2) \cdot I_2) \tag{5.2g}$$

Now the voltage V_1 is directly related to the voltage V_C of the capacitor C instead of voltage V_2. A consequence of this transformation is the decoupling of the dependency between V_1 and V_2. Thus, the equation system in (5.2) is now decomposed into two smaller subsystems. The profiling results of this decomposed system are shown below.

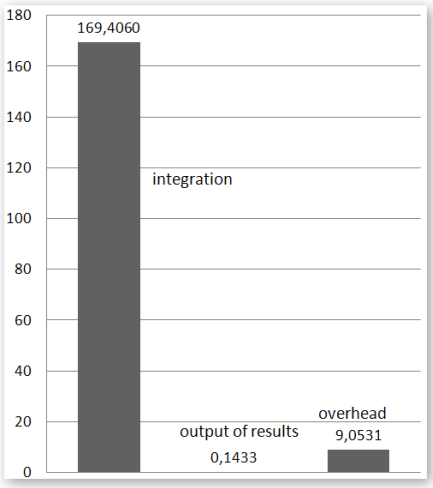

Figure 6: Workload in global steps (optimized)

The results show a decrease of overall runtime in each global step of about 30% workload compared to the original model. The number of integration steps is still 10, but now with 16.9406 μs average , and together they contribute about 95% of the total workload. The overhead has increased to 9.0531 μs and the calculation of outputs remains almost the same 0.1433 μs as given in Figure 6

To check whether the reason for the runtime improvement is the decoupling of algebraic loops, an analysis of the underlying integration steps has been carried out in Figure 7.

From the automatically generated source code it also can be seen there are two nonlinear subsystems. When solving these two nonlinear subsystems, each takes 5.9968 loops and each has execution times of 1.3746

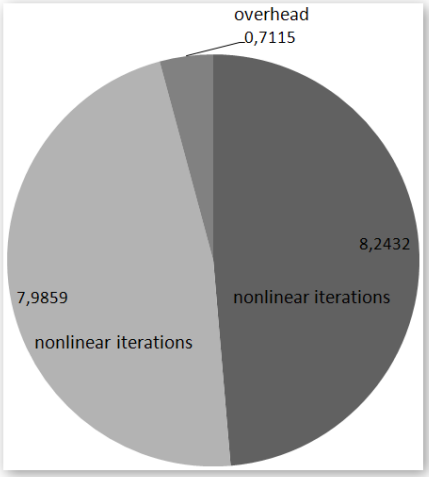

Figure 7: Workload in integration steps (optimized)

μs and 1.3317 μs. Summing them up, the total time for solving the first nonlinear block is $\tau^1_{tot\_non} = 8.2432$ μs, while the total time for solving the second nonlinear block is $\tau^2_{tot\_non} = 7.9859$ μs. Comparing these results with the ones shown in Figure 4, although the decoupling of the original system leads to two nonlinear subsystems, which require more iterations, these subsystems are usually easier and more efficiently to be solved. The final effect is an improvement of RT performance.

5.1.4 Deviation Analysis

A side effect of this RTO that introduces a capacitor is an unwanted oscillation which can be observed for instance at V_B. This is because of the nature of capacitor inside a dynamic electric circuit, which is described as a differential equation in (5.2d). Nevertheless, these oscillating errors are so small that it makes almost no difference to the results as it can be seen from the screenshot of the ModelComparator Figure 8.

5.2 Case 2: Nonlinear Thermal Resistor Circuit

An analogue to electric circuit in heat transfer context is a thermal circuit. Consequently the heat flow, temperature, thermal resistance, thermal capacity and temperature source are represented respectively by the current, voltage, resistor, capacitor and voltage source in a thermal circuit. However, any components in a thermal circuit might show nonlinear behaviors. In this case study, the nonlinearities of two thermal resistors are considered.

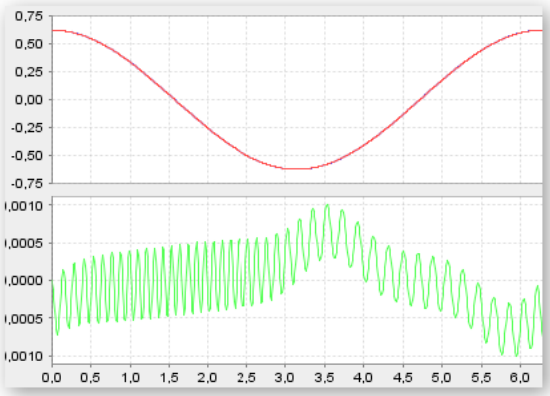

Figure 8: Error oscillations of V_B

5.2.1 Case Description

In the thermal circuit given in Figure 9, there are two temperature sources (T1 and T2) providing output temperature consisting of constant offset temperatures and small pure sinusoidal perturbations $T_{out} = T + \sin(t)$. Thermal resistances of the resistors (R1 and R2) are nonlinear functions of the temperature $R = f(T)$, where $f : \mathbb{R}^d \mapsto \mathbb{R}, d \in \mathbb{N}$ is a nonlinear mapping. For instance, the positive temperature coefficient (PTC) and negative temperature coefficient (NTC) thermistor, $R = f(T)$ is an expected physical behavior.

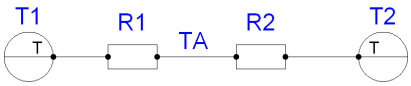

Figure 9: Thermal Circuit with nonlinear resistors

In order to calculate temperature T_A at point A, the following purely algebraic equations have to be solved:

$$q = \frac{T_1 - T_2}{R_1 + R_2} \tag{5.3a}$$

$$q = \frac{T_1 - T_A}{R_1} \quad \text{or} \quad q = \frac{T_A - T_2}{R_2} \tag{5.3b}$$

where q is the heat flow, T_A is the temperature at point A, $R_1 = f(T_1, T_A)$ and $R_2 = f(T_2, T_A)$ are the equivalent thermal resistances of the two resistors. As (5.3) forms a nonlinear equation, algebraic loops are expected in each integration step. The profiling results performed on the code executed on SCALE-RT are given in Figure 10.

The average number of integration steps of a global step is 99.9021 and the average execution time per in-

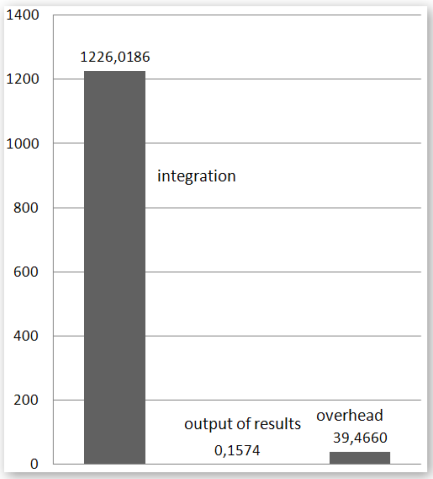

Figure 10: Workload in global steps

tegration is 12.2722 μs. Due to the ascent of the number of integrations per global step, the accumulated overhead has a value of 39.4660 μs. At each global step a calculation of outputs is performed and it takes 0.1574 μs, which is negligible compared to the average runtime 1266 μs in a global step. From these results, it is obviously to recognize that if a reduction can be achieved on the execution times or number of the integration steps, the RT performance will be enhanced significantly. A detailed view of the workload contribution in every integration step is given in Figure 11. The data in this figure further stresses the fact that the time for solving the nonlinear problem is the bottleneck of this model, which needs be handled for RT purpose.

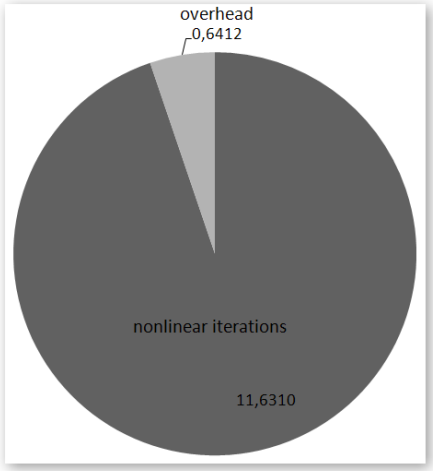

Figure 11: Workload in integration steps

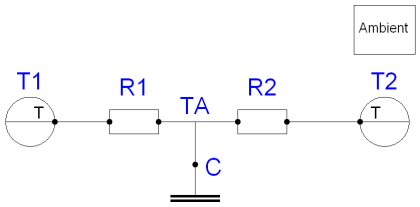

Figure 12: Optimized thermal circuit

5.2.2 RTO by Introducing Capacities

After introducing a thermal capacitor to the original thermal circuit, a decoupling of thermal resistances R_1 and R_2 is obtained Figure 12. Now the temperature at point A can be substituted by the temperature of the capacitor C instead of solving the underlying nonlinear equation. Assuming C.T, C.C, C.α and C.q are temperature, thermal capacity, thermal coefficient and heat flow of the capacitor C. q_1 and q_2 are heat flows in R_1 and R_2. Amb.T is the temperature of the surrounding ambiance. Then we have:

$$T_A = C.T \tag{5.4a}$$
$$R_1 = f(T_1, T_A) \tag{5.4b}$$
$$R_2 = f(T_2, T_A) \tag{5.4c}$$
$$q = \frac{T_1 - T_2}{R_1 + R_2} \tag{5.4d}$$
$$q_1 = \frac{T_1 - T_A}{R_1} \tag{5.4e}$$
$$q_2 = \frac{T_A - T_2}{R_2} \tag{5.4f}$$
$$C.q = q_1 - q_2 \tag{5.4g}$$
$$C.C \cdot der(C.T) = C.q - C.\alpha(Amb.T - C.T) \tag{5.4h}$$

The calculation here is pretty straightforward and no algebraic loops should be observed during a RT simulation, so a performance improvement should be expected. This is proved by the profiling results, since no algebraic loop inside integration steps has been measured. Instead of breaking up the algebraic loops into small ones as shown in section 5.1, the formal existing algebraic loops have been completely eliminated. Although the number of integration steps remains the same as in the original model, the execution time has been drastically reduced to 1.7042 μs. So the average computation time for a global step has been reduced to 219.6088 μs, which can be seen as a triumph of optimizations, see Figure 13.

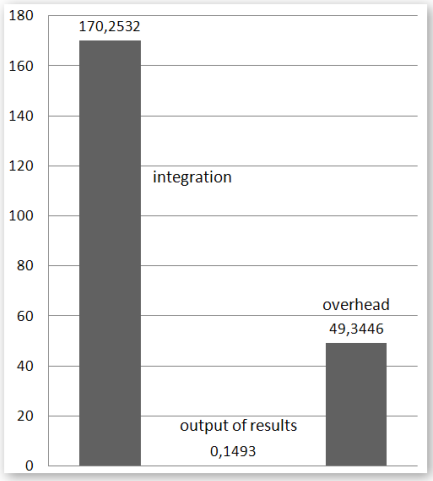

Figure 13: Workload in global steps (optimized)

5.2.3 Deviation Analysis

Model optimizations by introducing an element with some capacity is a good idea to break up the algebraic loops. But as shown in 5.1, one should be cautious when applying this technique on a model. If the time constant of the capacity is less than the integration step size, this will cause instability in a system. Unlike in case 1, where the voltage between those two inductors depends on the time derivatives of the current flowing through, here the thermal resistance R is just a function of the temperature. Hence R can be calculated directly from C.T. The following figure gives the comparison of the temperature T_A from original and optimized models.

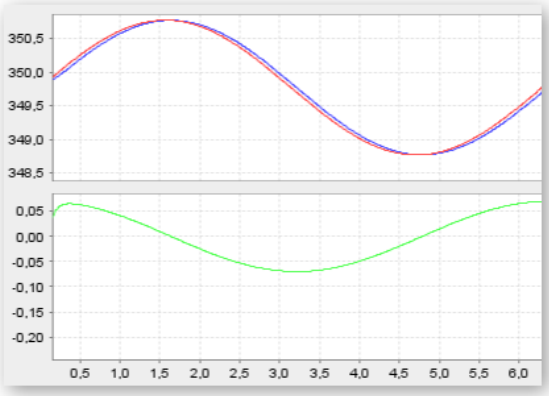

Figure 14: Error oscillations of T_A

As it is shown in the lower diagram, a small time shift of the solution T_A is observed in Figure 14, which causes the error in this case, but still in an acceptable range.

Acknowledgement

The work of the 3d and 4th authors was partially funded by the Federal Ministry of Education and Research (BMBF), Germany, in the projects TEMO (grant 01IS08013C) and OPENPROD (grant 01IS09029D).

We are thankful to Adina Aniculaesei and Mark Wessel for implementation support.

6 Conclusion

Adapting simulation models for execution in a real-time context is often a complex task that requires two-folded verification. First, accuracy of the results obtained with an optimized model and its stability have to be proven. Second, it has to be shown that the real-time constraints are met and if not, which are the most promising parts for further improvements. We presented two tools, a ModelComparator and the RT-Profiling, to support the developer with these tasks and illustrated their usage in two small case studies.

However, an open issue is how to relate the verification results from both, model comparison and RT-Profiling, back to the variables and equations of the optimized model. To solve this issue will be a precondition to enable real-time adaptation of models of another scale of complexity.

References

[1] MODELISAR (ITEA 2 07006). Functional Mock-up Interface for Model Exchange, January 26 2010.

[2] T. Blochwitz and T. Beutlich. Real-Time Simulation of Modelica-based Models. In *Proc. 7th Modelica Conference*, pages 386–392. The Modelica Association, 2009.

[3] F. Casella. Exploiting Weak Dynamic Interactions in Modelica. In *Proc. 4th Modelica Conference*, pages 97–103. The Modelica Association, 2005.

[4] C. Clauß and A. Schneider. Modelica Standard Library 2.2.1, 2007.

[5] W.H.A. Schilders et al. *Model Order Reduction*. Springer Verlag, 2008.

[6] S. Graham, P. Kessler, and M. McKusick. An Execution Profiler for Modular Programs. In

Software - Practice and Experience, volume 13, pages 671–685, 1991.

[7] Cosateq GmbH & Co. KG. Scale-RT, 2010.

[8] A.K. Noor. Recent advances and applications of reduction methods. *Appl. Mech. Rev.*, 1994.

[9] Terence Parr. ANTLR, 2010.

[10] A. Pop, D. Akhvlediani, and P. Fritzson. Towards Run-time Debugging of Equation-based Object-oriented Languages. In *Proceedings of the 48th Scandinavian Conference on Simulation and Modeling (SIMS' 2007)*, 2007. Göteborg, Sweden. October 30-31.

[11] A. Schiela and H. Olsson. Mixed-mode Integration for Real-Time Simulation. In *Modelica Workshop 2000 Proceedings*, pages 69–75. The Modelica Association, 2000.

[12] C. Schulze, M. Huhn, and M. Schüler. Profiling of Modelica Real-time Models. In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT)*, pages 23–32. Linköping Electronic Conference Proceedings, 2010.

[13] Ullrich von Bassewitz. cc65, 2010.