Supporting Infinitely Fast Processes in Continuous System Modeling

John Tinnerholm[1] Francesco Casella[2] Adrian Pop[1]

[1]Department of Computer and Information Science (IDA), Linköping University, Sweden,
{firstname.lastname}@liu.se
[2]Dipartimento di Elettronica Informazione e Bioingegneria, Politecnico di Milano, Italy,
francesco.casella@polimi.it

Abstract

This article examines the consequences of introducing a new language construct into an equation-based language to model infinitely fast processes. We do this by extending the equation-based language Modelica with a special time constant, Θ. Θ provides modelers with an additional language construct that they can utilize both to improve the performance of numerical integration for existing models as well as express and simulate models that existing tools may struggle with. In this paper we exemplify this with two examples. The first is an artificial DAE-System using a monotonic function; the second is an electrical circuit with and without a parasitic capacitance.

Based on our observations, we believe that by enabling modelers to express common idealizations using Θ we can improve both performance and maintainability. This is the case since it is possible to express the relevant idealizations can now be expressessd using Θ and are thereby explicitly encoded in the model.

Keywords: continuous system modeling, Modelica, modeling, nonlinear systems, simulation

1 Introduction

In the context of Modeling and Simulation (M&S) a commonly used modeling language is the Modelica Language which enables modelers to model complex systems using object orientation and equations to represent various physical components. By using equations, the Modelica language can model any domain that can be expressed using equations. The goal of the language is to provide modelers with the necessary abstractions to express complex cyber-physical systems.

However, modeling complex cyber-physical systems is a challenging task, and when designing such systems modelers frequently utilize idealization techniques in order to formulate models that can be simulated efficiently. Still, as of this writing, there are some techniques common in modeling practice that the Modelica language, and, according to our best knowledge and mainstream equation-based languages in general, do not support.

One such idealization technique is the method of *Artificial States* where the modeler extends the dynamics of systems by introducing additional state variables and associated equations, and in that way, reducing the complexity of the resulting nonlinear equation system, allowing efficient and reliable solving.

The use of artificial states is generally seen as malpractice in the modeling community, however, Zimmer (2013) provides recommendations concerning how M&S frameworks, and consequently equation-based modeling languages, can be extended such that modelers may express this idealization explicitly.

This idea is further expanded upon in (Zimmer 2014), where he proposes an augmentation to existing equation-based languages by introducing a new time constant, Θ.

1.1 Motivation

While the method proposed by Zimmer has been previously discussed and suggestions for implementing it have been described in (Zimmer 2013; Zimmer 2014), it has not yet been integrated into any mainstream equation-based language. In this article, we investigate the method initially proposed by Zimmer empirically. Hence, we aim to provide additional insights concerning the applicability Θ in pratice. We do this by examining and discussing the practical consequences of integrating these concepts into the equation-based language Modelica and provide details concerning how to integrate it in compilers for equation-based languages. To illustrate the concept we use a common modeling scenario with a nonlinear circuit and a DAE-System that existing Modelica environments have difficulties solving.

1.2 Organization

The remainder of this article is organized as follows: Section 2 further expands on the background provided in the introduction by discussing Θ and summarizing the mathematical background. We then provide details concerning how to extend a Modelica Simulation environment by adapting the structural transformation phases to accommodate this new operator in Section 3. Section 4 presents the results using the proposed new operator, and Section 5 presents additional related work. Finally, the conclusion is presented in Section 6.

2 Handling infinitely fast processes in continuous system modeling

In an equation-based language, the system of equations of the final system that is to be simulated is derived by collecting and merging the equations resulting from the components of some model. Systems in the following form are typical:

$$\frac{dx}{dt} = f(x,u,p,t)$$

where x is the set of state variables, u is the set of algebraic variables, and p is a vector representing parameters and constants. These systems consisting of both algebraic and differential equations are called **DAE**s. Compilers for equation-based languages such as Modelica translate such systems into executable code for simulation by techniques such as index reduction and topological sorting of the dependencies between its constituent parts.

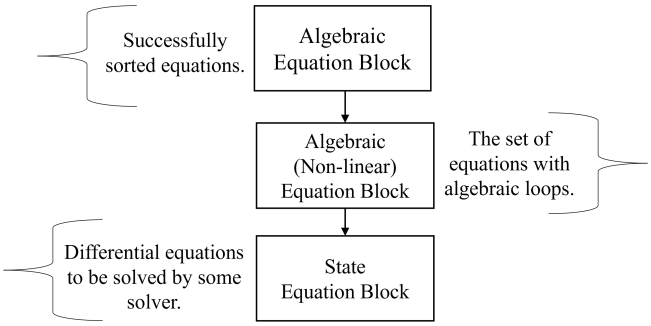

Figure 1. A sketch on how simulation code might look for a model that contains algebraic loops.

Due to the presence of algebraic loops, such systems may contain a nonlinear subsystem of equations and a system of ordinary differential equations. Figure 1 illustrates how a DAE-System might be translated by a model compiler. In this example, the part of the system that can be fully causalized is denoted the **Algebraic Equation Block**. Following this block is the **Algebraic (Nonlinear) Equation Block**. This block represents a nonlinear system of algebraic equations, caused by the existance of algebraic loops in the underlying model. Finally, in this example is the block of the differential equations, the **State Equation Block**.

Zimmer (2014) argues that a common occurrence in modeling practice is a system where parts of a system converge faster than others and provide the following example:

$$\begin{cases} \dfrac{dx}{dt} = f_x(x,y,u,t) \\ \dfrac{dy}{dt} = f_y(x,y,u,t) \end{cases} \tag{1}$$

In Equation 1, we assume that the process involves the state variable x, and converges faster than the process associated with the state variable y. Here, u denotes a set of algebraic variables. If we assume that the system in Equation 1 is stiff, and that the modeler is not interested in the dynamics of x a possible idealization is the system defined in Equation 2:

$$\begin{cases} 0 = f_x(x,y,u,t) \\ \dfrac{dy}{dt} = f_y(x,y,u,t) \end{cases} \tag{2}$$

In this case, the dynamics of the state variable x have been removed to make the system easier to solve. However, in this case we need to solve the nonlinear system:

$$0 = f_x(x,y,u,t)$$

The modeler, Zimmer (2014), argues, has to choose between two alternatives, either a stiff model represented by Equation 1 or a model with a possibly complex nonlinear system as in Equation 2. Zimmer (2014), argues that this selection is often made pragmatically; hence, once the modeler selects one alternative, the choice is not explicitly encoded in the model code. Consequently, future modelers that are to maintain such a model might not know that an idealization has been made, which arguably, makes maintenance more difficult.

Instead, Zimmer (2014) proposes the inclusion of a universal time constant, denoted Θ for equation-based languages to make this idealization explicit. Using this approach, Equation 1 can be formulated as follows:

$$\begin{cases} \dfrac{dx}{dt} \cdot \Theta = f_x(x,y,u,t) \\ \dfrac{dy}{dt} = f_y(x,y,u,t) \end{cases} \tag{3}$$

In Equation 3 the modeler explicitly expresses that $\frac{dx}{dt} \cdot \Theta = f_x(x,y,u,t)$ is an infinitely fast process. Consequently, this entails that the part of the system dependent on Θ is to be solved by means of a sub-simulation[1]. In other words, if Θ is used as in Equation 3, the system should be solved as in Equation 4. Where a sub-simulation provides the $\hat{x}$ value, by solving the differential equation $\frac{d\hat{x}}{dt} = f_x(\hat{x},y,u,t,\hat{t})$ where the state variable y, the variables in u and the global time[2] are treated as constants. Here, $\hat{t}$ is the artificial time used by the sub-simulation to model it as a infinitely fast process.

For additional details and mathematical background concerning Θ we refer to (Zimmer 2014).

$$\begin{cases} 0 = f_x(\hat{x},y,u,t) \\ \dfrac{dy}{dt} \cdot T_y = f_y(x,y,u,t) \end{cases} \tag{4}$$

[1] In the article **Handling infinitely fast processes in continuous system modeling** (Zimmer 2014), this is described to be solved by a continuation solver.

[2] The time t of the main simulation.

DOI 10.3384/ecp204423

```
model DAE_Example
  Real x(start = 1.0);
  Real y;
  Real a;
function s
  input Real a;
  output Real oa;
algorithm
  if (a < -1) then
    oa := a/4 -3/4;
  elseif (a > 1) then
    oa := a/4 + 3/4;
  else
    oa := a;
  end if;
end s;
equation
  der(x) = y;
  der(y) = -0.1*a - 0.4*y;
  der(a) = (10*x - s(a));
end DAE_Example;
```

Listing 1. A first attempt of a Modelica implementation of Equation 5.

2.1 Example 1: A Differential Algebraic Equation System

We turn now to a more concrete example of the previous discussion to exemplify how Θ can be used in the context of Modelica. Later in Section 4, we provide an example on how Θ may be used to significantly speed up simulations, and arguably, make models more maintainable by allowing modelers to explicitly state their intent. In Zimmer (2013) the following system is given:

$$f(x,y,a,t) = \begin{cases} \dfrac{dx}{dt} = y \\ \dfrac{dy}{dt} = -0.1a - 0.4y \\ \dfrac{dx}{da} = 10x - s(a) \end{cases} \tag{5}$$

With the monotonic increasing function s defined as:

$$s(a) = \begin{cases} a - 4 - 3/4 & \text{if } a < -1 \\ a/4 + 3/4 & \text{if } (a > 1) \wedge \neg(a < -1) \\ a & \text{otherwise} \end{cases} \tag{6}$$

If we formulate a model for the system defined by Equation 5 in the equation-based modeling language Modelica, see Listing 1, we end up with a stiff system. Still, if we attempt a similar idealization as described earlier in this section we could do so by substituting $\frac{dx}{da} = 10x - s(a)$ with $0 = 10x - s(a)$.

```
model DAE_Example2
  Real x(start = 1.0);
  Real y;
  Real a;
equation
  der(x) = y;
  der(y) = -0.1*a - 0.4*y;
  0 = (10*x - s(a));
end DAE_Example2;
```

Listing 2. An attempted idealization of the model in Listing 1, the function `s` has been omitted.

With this change we can formulate the Modelica model in Listing 2. If we attempt to simulate this system the state-of-the-art OpenModelica Compiler (Fritzson et al. 2020) is unable to simulate it correctly due to the resulting nonlinear system, unless a very small step size is selected. As discussed previously, this is clearly disadvantageous since it requires manual adjustments of solver settings. Furthermore, this impacts simulation performance negatively since a very small step size is needed.

(Zimmer 2013), presents a clear use case for introducing an operator called `balance`. The similarities to Θ means that it may be used in place of balance, much in the same way. If we apply Θ to Equation 5 we get the following system:

$$\begin{cases} \dfrac{dx}{dt} = y \\ \dfrac{dy}{dt} = -0.1a - 0.4y - \\ \dfrac{dx}{da} \cdot \Theta = 10x - s(a) \end{cases} \tag{7}$$

The resulting Modelica model and the associated code generation extensions needed for Θ will be discussed in Section 3, and the simulation resuls are presented in Section 4.

2.2 Example 2: Nonlinear Circuit

Let us now consider a less artificial example exemplified by using two configuration examples of an electrical circuit model.

The first example in question is a nonlinear circuit with a few diodes. The diodes are real diodes with an exponential voltage-current characteristic, not ideal diodes with either zero voltage or zero current. The model `Circuit1Static`, see Figure 2, has a series connection between the diodes and a large resistor. The result of this connection is a very strongly nonlinear system of equations. In this case the nonlinearity index (Casella and Bachmann 2021), will be $>> 1$. As a consequence, if simulated by the OpenModelica Compiler (Fritzson et al. 2020) the nonlinear algebraic equation solver experiences convergence issues, causing the master ODE integration

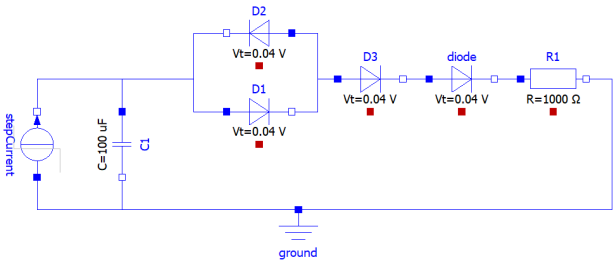

Figure 2. An electrical circuit with a nonlinear system that is difficult to solve. Model `Circuit1Static`.

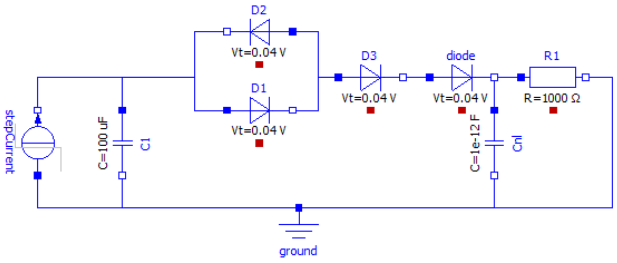

Figure 3. An electrical circuit with a less complex nonlinear system due to the parasitic capacitance `Cnl`. Model `Circuit1Dynamic`.

method to reduce the time step, and to eventually give up after 0.015 seconds.

The second circuit `Circuit1Dynamic`, see Figure 3 solves the problem by connecting a small parasitic capacitance between the diodes and the large resistor. The introduction of an additional state variable to the model makes the voltage at that node known at each time step during the simulation, hence significantly easing the solution of the system. Hence, by utilizing the method of artificial states, we ensure that the simulation can proceed without issues.

To conclude, the examples in Subsection 2.1 and Subsection 2.2 exemplify how various idealizations might be used in practice. Still, sometimes it might be difficult to get the correct simulation results as exemplified in the discussion of Subsection 2.1. In other cases we can get a system to simulate at the cost of introducing additional states. However, and as previously discussed and argued by Zimmer (2014) this might be a future detriment in terms of model maintainability. As we will see in Section 4, Θ may be used to significantly speed up simulations in this case, and arguably make the idealization more maintainable.

3 Implementation

In this section we present implementation details concerning the introduction of Θ in a Modelica compiler.

3.1 OpenModelica.jl

We implemented the ideas presented in this paper in **OpenModelica.jl**, a Julia-based Modelica Compiler (Tinnerholm, Pop, and Sjölund 2022). OpenModelica.jl is written in the programming language Julia and supports some experimental features not currently available in mainstream Modelica Compilers. This compiler integrates several Julia packages such as ModelingToolkit (MTK) (Ma et al. 2021) and DifferentialEquations.jl (Rackauckas and Nie 2017). The main feature of this compiler being its modularization and extensions that introduce support for Variable Structure System Modeling for Modelica. As a part of this work a new code generator was written to export the intermediate representation produced by MTK models to a more portable low-level representation. Furthermore, we implemented support for a significant subset of the Electrical Library of the Modelica Standard Library for this new code generator.

3.2 Extending Modelica with Θ

The typical compilation process of a compiler for an equation-based language is to transform the provided model into a suitable format for some solver. The general process is described in Figure 4.

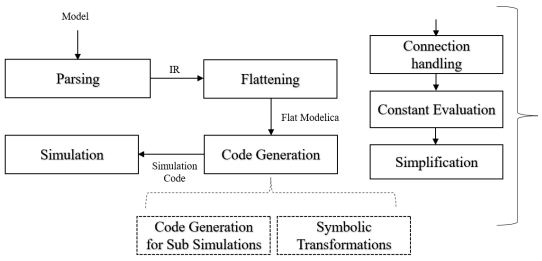

Figure 4. An illustration of a typical compiler pipeline for an equation-based language. The frontend is similar to that of an ordinary compiler; it performs parsing, syntactical, and semantical analysis of the input model. Finally, a compiler for an equation-based language typically generates code targeting a solver such as DASSL (Petzold 1982).

In principle introducing Θ involves only slight changes to key parts of this process. First of all, Θ should not only be used as a low-level operator; instead, Θ should be applicable in a non-invasive way such that the internal equations of models that it is applied to are untouched. Furthermore, Θ should be propagated and not be removed during any optimization phase. As such, Θ should be available and taken into consideration by the various structural transformation phases, such as sorting performed by the compiler backend.

3.3 Preparing Code For Simulation

Zimmer (2014) provides an initial sketch for simulation code generation when expanding an equation-based language with Θ. In summary, the steps are as follows:

1. Treat Θ as an irreducible variable.
2. Analyze that Θ has been applied correctly.

DOI
10.3384/ecp204423

3. Generate code for simulation with respect to how Θ has been applied.

To provide initial support for Θ we implemented it as a special parameter by introducing a new reserved keyword `THETA`[3]. By reserving a name we can via static analysis follow the def-use chains in the frontend and abstain from removing the parameter during the backend optimization phases. Hence, concerning the first step, we fulfill it by omitting certain optimization phases such as not removing simple equations that have structural dependencies on `THETA`. This means that Θ remains in the system of equations, after the sorting, matching, and index-reduction phases are completed.

The second step entails adhering to several constraints, Zimmer (2014) proposes the following constraints:

1. The resulting system should be successfully balanced.

2. The resulting system should be successfully causalized.

3. Furthermore, each variable may be expressed as a factor of Θ^n where n is an integer. Moreover, Θ may not be used as a function argument for nonlinear functions such as `sin`, `cos`. Furthermore, the variables in $\frac{dx}{dt}$ should be multiplied by Θ or Θ^0, where multiplication by Θ indicates that sub-simulation code should be generated for that variable[4]

These requirements were fulfilled by augmenting the compiler backend with additional checks before proceeding with simulations, however, the check concerning invalid usage of Θ in nonlinear functions was omitted.

3.4 Generating code for state variables depending on Θ.

As previously stated, variables in $\frac{dx}{dt}$ should be multiplied by either Θ or Θ^0 where the first indicates that code representing an infinitely fast process should be generated for that part of the system. For more mathematical detail concerning the code generation for this process, we refer to (Zimmer 2014).

To exemplify the current state of our code generator let us consider the Modelica model in Listing 1. Using the aforementioned new parameter THETA we can augment our code and write a new model as in Listing 3.

The structural analysis is simple for the model depicted in Listing 3. Code for a sub-simulation is generated for

```
der(a) * THETA = (10*x - s(a))
```

```
model DAE_Example_THETA
  Real a;
  Real x(start = 1);
  Real y;
  parameter Real THETA = 1.0;
equation
  der(x) = y;
  der(y) = -0.1*a - 0.4*y;
  der(a)*THETA = (10*x - s(a));
end DAE_Example_THETA;
```

Listing 3. A Modelica implementation of Equation 5, here the function `s` is omitted.

For the main simulation, this equation is replaced with the following nonlinear equation as described in Section 2:

```
0 = (10*x - s(a))
```

During the simulation, the sub-simulation is solved using the implicit Euler integration algorithm, providing $\hat{x}$ for the main simulation. The current termination criterion for the sub-simulation is running the artificial time $\hat{t}$ from $\hat{t} = 0$ until $t_{current}$[5]. For a high-level description of the code generated for the solvers, we refer to Algorithm 1 and Figure 5.

Algorithm 1 High-level description of the code generated when translating the Modelica model in Listing 3.

function K(u)
  Initialize x and y using u.
  $ox[1] \leftarrow 10y[1] - s(x[1])$
end function
function H(dy, y, u, t)
  sub-simulation(u)
  nonlinear-solve(k)
  $dy[1] \leftarrow y[2]$
  $dy[2] \leftarrow -0.1x[1] - 0.4y[2]$
end function
function SUB-SIMULATION(u)
  extract x from u.
  solve $\frac{da}{dt} = 10 \cdot x - s(a)$
  Update u, provide $\hat{a}$ for the main simulation.
end function
function SIMULATION-FUNCTION
  Simulate by integrating the H function.
  Report results.
end function

In general, however, the structural analysis needed, may be more involved. Consider, the electrical circuit in Figure 3, just as in Listing 3 the model is a suitable candidate

[3]We note that this might break existing models using parameters with the same name, however, we use it in the initial implementation to illustrate the concept.

[4]Θ^0 means that Θ has not been applied.

[5]It should be noted that a dedicated algorithm to describe the sub-simulation is presented in (Zimmer 2013). Compared to the algorithm suggested in (Zimmer 2013) we currently take more steps in the sub-simulations then necessary.

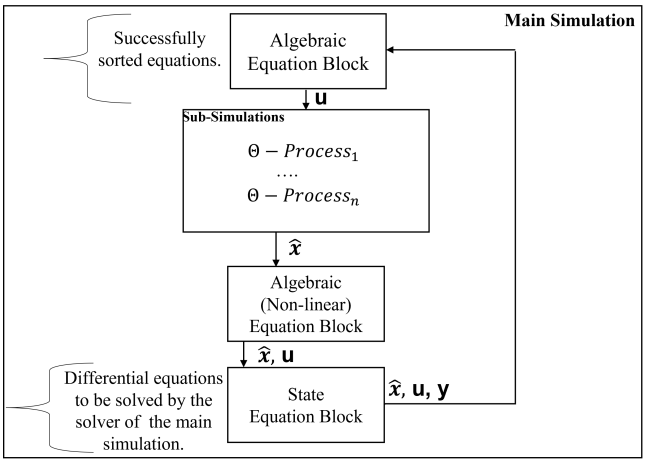

Figure 5. Graphical illustration of the simulation code generated for a model with n non-nested subprocesses. Showing where Θ appears in generated code, the Θ-processes supply the nonlinear solver with initial values.

```
package CircuitTest
  model ThetaCircuit2Dynamic
    parameter Real THETA = 1.0;
    extends Circuit1Static;
    Capacitor Cp(C = 1e-12 * THETA);
  equation
    connect(Cp.n, ground.p);
    connect(diode.n, Cp.p);
  end ThetaCircuit2Dynamic;
end CircuitTest;
```

Listing 4. Modelica model showcasing how the Θ is used at the top level of the component hierarchy. The components used are from the Modelica Standard Library; the package paths have been omitted.

for applying Θ. To illustrate how it can be applied to existing models without changing any equations at a lower abstraction level consider the model depicted in Listing 4 where Θ is applied at the top level.

$$\begin{cases} 0 = Cp\_v - R1\_R\_actual \cdot R1\_i \\ 0 = diode\_i - (10^{-9}(tmp53 - 1) - (10^{-8}))diode\_v \\ 0 = D2\_i + diode\_i - D1\_i \\ \frac{Cp\_v}{dt} = Cp\_i/(10^{-12}\Theta) \\ \frac{C1\_v}{dt} = 9999.99999999999 \cdot C1\_i \end{cases} \tag{8}$$

When used as in Listing 4, the compiler initially generates the equations listed in Equation 8, then, the compiler starts Θ specific code generation. During this process structural analysis is used to extract the processes that should be run as sub-simulation from the resulting equations. This is achieved by using the following steps:

1. Construct a graph based on equation-variable dependencies.
2. Extract equations were the Θ operator is used.
3. Extract the set of variables depending on Θ.
4. Return the strongly connected components of the equation-variable dependency graph.

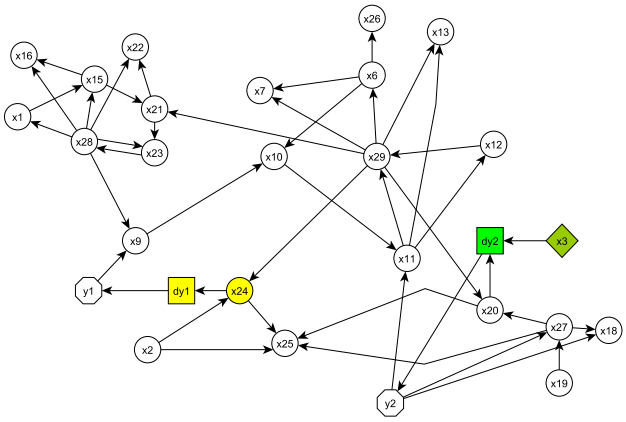

Figure 6. Excerpt of the dependency graph for `ThetaCircuit2Dynamic` in Listing 4. The variables that depend on Θ is marked in green, the other state is marked in yellow.

The last step to extract the strongly connected components uses Tarjans algorithm (Tarjan 1971). To illustrate this process graphically we refer to Figure 6. We refer to the algorithm in (Zimmer 2014) for a more formal description of the steps involved.

After the structural analysis, the compiler generates Equation 9 for the main simulation process and Equation 10 for the sub-simulation.

$$\begin{cases} 0 = Cp\_v - R1\_R\_actual * R1\_i \\ 0 = diode\_i - 10^{-9}(tmp53 - 1)10^{-8}diode\_v \\ 0 = D2\_i + diode\_i - D1\_i \\ \frac{C1\_v}{dt} = 9999.99999999999 \cdot C1\_i \end{cases} \tag{9}$$

$$\frac{Cp\_v}{dt} = Cp\_i/(10^{-12}\Theta) \tag{10}$$

DOI
10.3384/ecp204423

4 Simulation Results

In Section 3 we discussed the practical integration of Θ in a Modelica compiler. In this section we will present our findings concerning concrete practical benefits of using this new construct. We do so by presenting two motivating examples, the first being a description of our results when simulating the model in Listing 3. The second example concerns simulation speedup when simulating the circuit depicted in Figure 3 compared to the circuit using Θ listed in Listing 4.

4.1 Simulating the DAE_Example

As discussed previously simulating **DAE_Example** without Θ resulted in undeseriable results, see Listing 2 both when using OpenModelica.jl and OpenModelica.

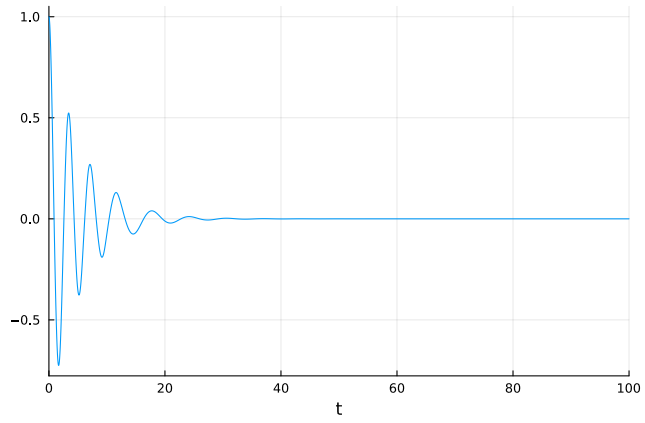

Figure 7. Simulation result showing the oscillation of x for Listing 2.

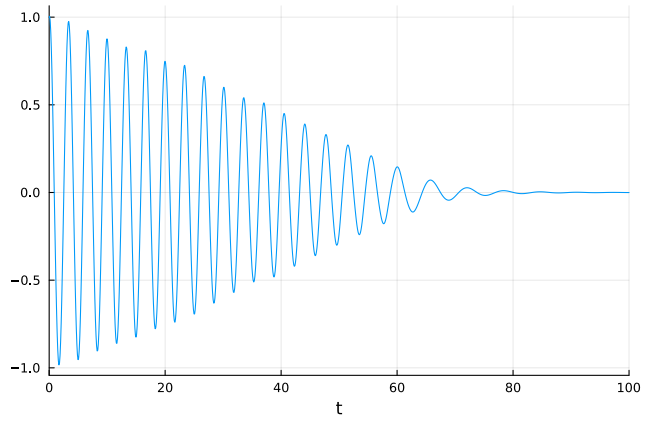

Figure 8. Simulation result showing the oscillation of x after theta has been applied.

Simulating the model using Θ as done in Listing 3 produces the correct plot; see Figure 8. As we did not have access to the original code nor to the model simulated as the small application example in (Zimmer 2013) the initial values of the system were assumed to be $x_0 = \{1,0,0\}$ for x, a and y respectively.

4.2 Simulating the Dynamic circuit using Θ

As previously mentioned, simulating the static circuit depicted in Figure 2 resulted in failure for the nonlinear solver.

Simulating the same system using the parasitic capacitance as depicted in Figure 3, leads to a successful simulation, however, the nonlinear system is complicated to solve leading to the solver having to take several time steps to integrate the system successfully, see Figure 9 for the plot of $C1.v$ for this circuit.

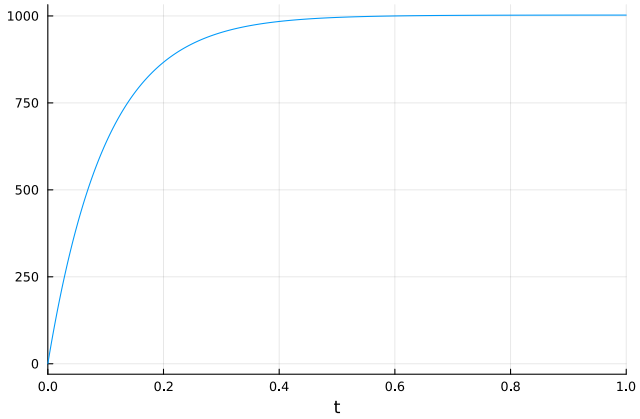

Figure 9. Simulation of $C1.v$ for the circuit in Figure 3 using Rodas5.

Using the method for code generation described in Section 3 simulation code was successfully generated for the model in Listing 4. We validated the solution of simulating the system using an infinitely fast sub-simulation by comparing the obtained results to the original results. There were no notable differences between the two simulations, see Figure 9 and Figure 10.

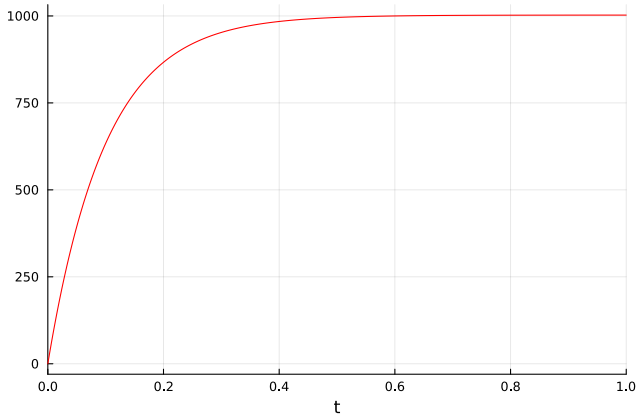

Figure 10. Simulation of $C1.v$ for the circuit model in Listing 3 using Tsit5.

Similarly, to the `DAE_Example` using Θ for the circuit model permits using solvers for non-stiff problems rather than stiff solvers such as DASSL (Petzold 1982). In this example, we compare the results of simulating `Circuit1Dynamic` using the following setup:

1. Simulating the System using the OpenModelica Compiler with the DASSL solver.
2. Simulating the System using OpenModelica.jl, the Julia-based Modelica Compiler with the Rodas5 solver[6]
3. Simulating the System using OpenModelica.jl, the Julia-based Modelica Compiler using Θ and the Rodas5 solver.
4. Simulating the System using OpenModelica.jl, the Julia-based Modelica Compiler using Θ with the explicit Tsit5 solver (Tsitouras 2011).

The simulations in the experiment had the absolute and relative tolerance levels set to $1e-6$. The experiment was run on a Laptop with an **AMD Ryzen 7 PRO 5850U with Radeon Graphics** and 32.0 GB internal memory, using Microsofts Subsystem for Linux with version **5.10.102.1-microsoft-standard-WSL2**. In terms of software, the Julia version used was **1.9-RC1** and the version of OpenModelica was **v1.21.0**.

Table 1. Solver statistics when simulating the dynamic circuit in Figure 3. C_{Rodas} refers to the simulation of the circuit without using Θ with the Rodas5 solver. $C_{\Theta Rodas}$ and $C_{\Theta TSIT5}$ refers to the result of simulating the same circuit using Θ.

Statistic	C_{Rodas}	$C_{\Theta Rodas}$	$C_{\Theta TSIT5}$
#Accepted Steps	109	36	39
#Rejected Steps	5	0	0
#Jacobians Created	109	36	0
#Linear Solves	912	288	0

The solver statistics for the models generated by the OpenModelica.jl are available in Table 1. As expected, we can see that using the Θ not only allows us to use explicit solvers such as Tsit5, it also reduces the amounts of integration steps needed to complete the simulation.

It is interesting to also compare the results with respect to the total simulation time for existing Modelica Compilers. For this purpose, we also compared the result of running the simulation using the OpenModelica Compiler. To benchmark the Julia code generated by OpenModelica.jl, we used Benchmarking software (Chen and Revels 2016) with the maximum number of samples set to 100. The simulation time for the OpenModelica Compiler was obtained by sampling the simulation statistics 10 times.

[6]https://docs.sciml.ai/DiffEqDocs/stable/solvers/ode_solve/ Accessed 2023-04-25.

Table 2. Simulation Statistics comparing the simulation of `Circuit1Dynamic` using the OpenModelica Compiler (OMC), and the results of running the same model using Θ with the Julia-based Compiler, OpenModelica.jl (shorten to OM.jl in the table). The sample mean, median and standard deviation of the total simulation time are denoted $\hat{x}$, $\hat{M}$, and $\hat{\sigma}$ respectively.

OMC (DASSL)	$\hat{M}$	$\hat{x}$	$\hat{\sigma}$
	41.016ms	44.2496ms	11.984ms
OM.jl (Tsit5)	$\hat{M}$	$\hat{x}$	$\hat{\sigma}$
	13.709ms	14.801ms	1.892ms

The results are presented in Table 2; from these results, we can see that there is a clear speedup in the experimental compiler using this method, in this case, by about 2.9 times.

5 Related Work

The techniques discussed and implemented in this paper were proposed in Zimmer (2013) and further elaborated upon in Zimmer (2014).

A technique similar to the extension of the Modelica language presented in this paper is the `homotopy` operator. The `homotopy` operator was added to the Modelica language to provide an option for more robust initialization (Sielemann et al. 2011).

Artificial time integration in the context of Partial Differential Equations has been proposed and investigated by Ascher, Huang, and Van Den Doel (2007).

6 Conclusions and Future Work

In this article, we have demonstrated the usefulness of introducing a new construct, Θ in the equation-based language Modelica. We integrated support for Θ in OM.jl and we used two examples with an associated microbenchmark to illustrate its advantages. The example presented in Subsection 4.1 illustrates how more robust simulations can be achieved using Θ. The second example presented in Subsection 4.2, shows how Θ may speed up the total simulation time in models constructed using existing standard components.

However, several open questions remain unanswered. The first question is selecting a suitable initial step size for the sub-simulation. Currently, if the step size is too small the solver will need to take many steps. If it is too long, it corresponds to more or less to solving the algebraic equilibrium equation outright; in that case, this method will not be used.

Furthermore, the current implementation relies on the implicit solvers provided by the MTK-ecosystem. This is not optimal in this case because such solvers tend to report failures late, whereas in this case failures should be reported as early as possible. Moreover, such solvers save intermediate values resulting in unnecessary high mem-

DOI 10.3384/ecp204423

ory consumption furthermore it also compute them with high precision and error control, which is not relevant in this case, only the asymptotic result is. While using the solvers from MTK worked for the example examined in this paper, a specialized embedded algorithm should be implemented instead.

An initial proposal of such an algorithm was proposed in (Zimmer 2013). Still, we believe such an algorithm could need further improvements. Improvements include utilize heuristics to select a suitable initial step size. Using an embedded subsimulation algorithm would also eliminate the need to save intermediate values, hence, reducing the memory footprint of the final simulation. As an extension to the work presented here further research should be invested to design and implement specialized algorithms and heuristics designed to be embedded for these purposes.

In this paper, we used the circuit model `ThetaCircuit2Dynamic` to illustrate how Θ could be applied to an existing model, `Circuit1Dynamic` listed in Listing 6, showing how existing models could integrate this without changing any low-level implementation. However, we have yet to investigate how well this new method scales when using nested sub-simulations. Hence, we should investigate the practical effects on larger models with more complex dependencies. This should be done both to finetune a possible heuristic for the initial step size of the sub simulations and gain even more insight concerning the robustness of the method.

To conclude we have examined the consequences when introducing a construct to an equation-based language to express infinitely fast processes; our experiments in Section 4 show clear net benefits of supporting Θ both in terms of speed and accuracy for the models that we tested.

Acknowledgements

This work has been supported by the Swedish Government in the ELLIIT project and has been partially funded by Vinnova in the context of the ITEA project EMBRACE. Furthermore, the authors gratefully acknowledge the financial support of the French Transmission System Operator RTE to the Open Source Modelica Consortium, which made this initial development possible.

References

Ascher, UM, H Huang, and K Van Den Doel (2007). "Artificial time integration". In: *BIT Numerical Mathematics* 47, pp. 3–25. DOI: 10.1007/s10543-006-0112-x.

Casella, Francesco and Bernhard Bachmann (2021). "On the choice of initial guesses for the Newton-Raphson algorithm". In: *Applied Mathematics and Computation* 398, p. 125991. ISSN: 0096-3003. DOI: https://doi.org/10.1016/j.amc.2021.125991. URL: https://www.sciencedirect.com/science/article/pii/S0096300321000394.

Chen, Jiahao and Jarrett Revels (2016-08). "Robust benchmarking in noisy environments". In: *arXiv e-prints*, arXiv:1608.04295. arXiv: 1608.04295 [cs.PF].

Fritzson, Peter et al. (2020). "The OpenModelica integrated environment for modeling, simulation, and model-based development". In: *Modeling, Identification and Control* 41.4, pp. 241–295. DOI: 10.4173/mic.2020.4.1.

Ma, Yingbo et al. (2021). *ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling*. arXiv: 2103.05244 [cs.MS].

Petzold, Linda R (1982). *Description of DASSL: a differential/algebraic system solver*. Tech. rep. Sandia National Labs., Livermore, CA (USA).

Rackauckas, Christopher and Qing Nie (2017). "DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia". In: *The Journal of Open Research Software* 5.1. DOI: 10.5334/jors.151. URL: https://app.dimensions.ai/details/publication/pub.1085583166%20and%20http://openresearchsoftware.metajnl.com/articles/10.5334/jors.151/galley/245/download/.

Sielemann, Michael et al. (2011). "Robust initialization of differential-algebraic equations using homotopy". In: *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany*. 063. Linköping University Electronic Press, pp. 75–85.

Tarjan, Robert (1971). "Depth-first search and linear graph algorithms". In: *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pp. 114–121. DOI: 10.1109/SWAT.1971.10.

Tinnerholm, John, Adrian Pop, and Martin Sjölund (2022). "A Modular, Extensible, and Modelica-Standard-Compliant OpenModelica Compiler Framework in Julia Supporting Structural Variability". In: *Electronics* 11.11, p. 1772.

Tsitouras, Ch (2011). "Runge–Kutta pairs of order 5 (4) satisfying only the first column simplifying assumption". In: *Computers & Mathematics with Applications* 62.2, pp. 770–775.

Zimmer, Dirk (2013-03). "Using Artificial States in Modeling Dynamic Systems: Turning Malpractice into Good Practice". In: *Proceedings of the 5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Ed. by Henrik Nilsson. Linköping University Press, pp. 77–85. URL: https://elib.dlr.de/84089/.

Zimmer, Dirk (2014). "Handling Infinitely Fast Processes in Continuous System Modeling". In: *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. EOOLT '14. Berlin, Germany: Association for Computing Machinery, pp. 31–34. ISBN: 9781450329538. DOI: 10.1145/2666202.2666206. URL: https://doi.org/10.1145/2666202.2666206.

A Models

This appendix contains the Modelica models of the circuits depicted in Figure 2, Figure 3. It also contains some example models illustrating different ways the Θ operator can be used. In **ThetaCircuit1Dynamic** it is used as a part of a low-level submodel, **ThetaCapacitator** and in **ThetaCircuit2Dynamic** it is used in the topmost model to enable efficient code simulation. The components used are from the Modelica Standard Library; the package paths have been omitted. The annotations have been omitted.

```
package DAE_Examples
  function s
    input Real a;
    output Real oa;
  algorithm
    if (a < -1) then
      oa := a/4 -3/4;
    elseif (a > 1) then
      oa := a/4 + 3/4;
    else
      oa := a;
    end if;
  end s;

  model DAE_Example
    Real x(start = 1.0);
    Real y;
    Real a;
  equation
    der(x) = y;
    der(y) = -0.1*a - 0.4*y;
    der(a) = (10*x - s(a));
  end DAE_Example;

 model DAE_Example2
   Real x(start = 1.0);
   Real y;
   Real a;
 equation
   der(x) = y;
   der(y) = -0.1*a - 0.4*y;
   0 = (10*x - s(a));
 end DAE_Example2;


  model DAE_Example_THETA
    Real a;
    Real x(start = 1);
    Real y;
    parameter Real THETA = 1.0;
  equation
    der(x) = y;
    der(y) = -0.1*a - 0.4*y;
    der(a)*THETA = (10*x - s(a));
  end DAE_Example_THETA;
end DAE_Examples;
```

Listing 5. The DAE Example models.

```
package CircuitTests
  //Details and annotations are omitted
  model Circuit1Static
    extends Modelica.Icons.Example;
    Ground ground;
    StepCurrent stepCurrent(I = 1);
    Capacitor C1(C(displayUnit = "uF") =
     ↪  0.0001000000000000001);
    Resistor R1(R = 1000);
    Diode D1(Ids = 1e-9, Maxexp = 40);
    Diode D2(Ids = 1e-9, Maxexp = 40);
    Diode D3(Ids = 1e-9, Maxexp = 40);
    Diode diode(Ids = 1e-9, Maxexp = 40);
  equation
    connect(C1.n, ground.p);
    connect(stepCurrent.p, ground.p);
    connect(R1.n, ground.p);
    connect(stepCurrent.n, C1.p);
    connect(C1.p, D1.p);
    connect(C1.p, D2.n);
    connect(D2.p, D3.p);
    connect(D1.n, D3.p);
    connect(D3.n, diode.p);
    connect(diode.n, R1.p);
  end Circuit1Static;

  model Circuit1Dynamic
    extends Circuit1Static;
    Capacitor Cnl(C (displayUnit = "F")=
     ↪  1e-12);
  equation
    connect(Cnl.n, ground.p)
    connect(diode.n, Cnl.p)
  end Circuit1Dynamic;

  model ThetaCapacitator
    extends OnePort(v(start=0));
    parameter Capacitance C(start=1)
     ↪  "Capacitance";
    parameter Real THETA;
  equation
    i = C*THETA*der(v);
  end ThetaCapacitator;

  model ThetaCircuit1Dynamic
    extends Circuit1Static;
    TestThetaMethod.ThetaCapacitator Cnl(C =
     ↪  1e-12);
  equation
    connect(Cnl.n, ground.p);
    connect(diode.n, Cnl.p);
  end ThetaCircuit1Dynamic;

  model ThetaCircuit2Dynamic
    parameter Real THETA = 1.0;
    extends Circuit1Static;
    Capacitor Cp(C = 1e-12 * THETA);
  equation
    connect(Cp.n, ground.p);
    connect(diode.n, Cp.p);
  end ThetaCircuit2Dynamic;

end CircuitTests;
```

Listing 6. The Circuit models discussing in Section 2.

DOI
10.3384/ecp204423