electronics

MDPI

Article

Modelling and Simulation of Physical Systems with Dynamically Changing Degrees of Freedom

Andrea Neumayr * and Martin Otter *

German Aerospace Center (DLR), Institute of System Dynamics and Control (SR), 82234 Wessling, Germany
* Correspondence: andrea.neumayr@dlr.de (A.N.); martin.otter@dlr.de (M.O.)

Abstract: A new approach is introduced to model and simulate equation-based systems where variables can appear and disappear during simulation without re-generation and re-compilation of code when the numbers of equations and states change during events. The method is presented in a generic, mathematical way and can be in principle applied to all types of declarative, equation-based modelling languages, such as Modelica. A concrete implementation is given for the Julia-based experimental modelling language Modia, which is similar to Modelica. However, Modia has a much simpler semantics based on hierarchical collections of name/value pairs, and is capable of supporting domain-specific algorithms, expecially for multibody systems with collision handling. The new method is demonstrated with heat-transfer in a rod, separation of stages of a rocket and gripping operations of a robot.

Keywords: Modelica; Julia; Modia; multibody; multi-mode; variable structure systems; segmented simulation; built-in component

check for updates

Citation: Neumayr, A.; Otter, M. Modelling and Simulation of Physical Systems with Dynamically Changing Degrees of Freedom. *Electronics* **2023**, *12*, 500. https://doi.org/10.3390/electronics12030500

Academic Editor: Xue (Shelley) Lin

Received: 19 December 2022
Revised: 14 January 2023
Accepted: 16 January 2023
Published: 18 January 2023

1. Introduction

The standardized, declarative and equation-based Modelica language [1] and the open source and commercial tools supporting Modelica [2] are in widespread use in scientific and industrial applications to model, simulate and design cyber-physical systems. Modelica can be seen as a format to define large sets of differential, algebraic and discrete equations in a standardized way on a high, user-friendly level.

Modelling languages that are declarative and equation-based have the principle advantage that complex models can be defined on a high level because sophisticated symbolic algorithms allow automatic transformation into a low-level format that can be solved with standard numerical solvers for ODEs (ordinary differential equations). The principle drawbacks are that (a) the approach does not scale for large systems because the equations of n instances of a model component are present n times in the generated code, and (b) specialized modelling techniques and algorithms that are successfully utilized in various physical domains cannot be directly applied. For example, the multibody community has designed specialized methods for efficient simulation of 3D-mechanical systems (see, e.g., [3]) that cannot be directly utilized by an equation-based language.

Modia [4] is an experimental, open source modelling and simulation system that is used to develop new approaches to overcome the limitations of declarative, equation-based modelling languages, for example, by combining equation-based modelling with domain-specific algorithms, especially from the multibody and fluid fields, or by using very simple, yet powerful, language semantics that define models with hierarchical collections of name/value pairs. Modia consists of a set of Julia packages, most importantly of packages Modia.jl (https://github.com/ModiaSim/Modia.jl, accessed on 13 January 2023) and Modia3D.jl (https://github.com/ModiaSim/Modia3D.jl, accessed on 13 January 2023), and relies heavily on the powerful programming language Julia [5] and the Julia eco-system (https://julialang.org/, accessed on 13 January 2023).

In this article, we present a new approach to modelling and simulating equation-based systems where variables can appear and disappear during a simulation without re-generation and re-compilation of code when the numbers of equations and states change during events. The method is presented in a generic, mathematical way and can be in principle applied to all types of declarative, equation-based modelling languages, such as Modelica. A concrete implementation is given for Modia, together with several applications that are based on this new feature.

Equation-based languages define models with DAEs (differential algebraic equations). With structural analysis methods such as the Pantelides algorithm [6] or Pryce's ∑-method [7], along with further symbolic transformation techniques, it is possible to transform DAEs to ODEs that can be solved with standard numerical methods.

Variable-structure systems are models where equations change during simulation. The idea of multi-mode modelling is to define model components with state machines where component equations change whenever a transition to another state occurs; see, e.g., [8]. One difficulty is to efficiently treat such models. Another difficulty is that a transition can lead to Dirac impulses. Benveniste, Caillaud et al. [9–11] extended the structural analysis with the Pantelides algorithm and Pryce's ∑-method for multi-mode models. In [11], it is demonstrated how a multi-mode Modia model is treated by re-generating and re-compiling code on-the-fly with Julia when a state transition occurs and initializing in the new states even if Dirac impulses occur.

Höger [12] also worked on Pryce's ∑-method for variable-structure modelling. Zimmer [13] used a run-time interpreter to process the DAE equations at run-time, when the structure and/or the DAE index was changing. The limitations of this approach are that impulsive behaviour is not supported and that the simulation time is one or more orders of magnitude larger than if compiled code is used. Pepper et al. [14] described the semantics of variable-structure modelling with state machines. Mehlhase [15] provided a Python-based approach where transitions can be made between pre-defined models. Elmqvist, Mattsson et al. [8,16] proposed high-level descriptions of multi-mode models in Modelica by extending the synchronous clocked state machines to continuous-time state machines.

Tinnerholm et al. [17] provided a Julia-based implementation of Modelica called OpenModelica.jl that supports variable-structure systems. A distinction is made between an explicit and an implicit variable structured system. For the explicit variable structured system the transition between states of the system are explicitly encoded by the user. Thus, all equations and variables of the system are known beforehand and the compiler and the simulation runtime need to process the entire model at once. For the implicit variable structured systems predefined events trigger a re-compilation of the model with Julia on-the-fly during simulation.

All current proposals for variable-structure systems either need to know the entire models for all modes beforehand and switch between these models during simulation, or the entire model is newly processed and code re-generated and re-compiled (or interpreted) whenever the equation structure is changed. In this article, several novel features are introduced that overcome current limitations:

1. The sizes of array equations can be changed after code generation. A simple example is shown in Section 2, where the parameter matrices of an LTI (Linear Time-Invariant) system can still be provided with different dimensions after generation and compilation of code.
2. Built-in model components are introduced that scale for large systems because the component equations are split into a (usually) large part that is encoded in a small set of pre-compiled functions and into a (usually) small part in form of standard equations which is processed with the entire model. A simple example is shown in Appendix B.1, where the core part of a discretized partial differential equation is present in pre-compiled functions and the (acausal) component is defined with four scalar equations that are independent of the discretization.

3. Variables and equations of built-in model components can appear and disappear during simulation, without re-generation and re-compilation of code, provided these variables and equations are part of the pre-compiled functions. Simple examples are shown in Appendix B, where (a) the number of volumes of a discretized partial differential equation can be changed after generation and compilation of code, and (b) the number of equations and states of a two-stage rocket are changed during simulation due to the separation of stages.

The limitation of the new approach is that built-in components cannot be designed for arbitrary (useful) connection scenarios. For example, built-in components cannot be used in a way so that one or more of its pre-compiled functions need to be differentiated. Therefore, the approach is not as general as if all equations of an entire model are newly processed when the model's structure is changing. However, the class of systems that can be practically handled is still large and has the advantage that it scales for large systems and leads to efficient simulations because code is *not* re-generated and re-compiled on-the-fly during simulation.

This article is organized as follows: An overview is given in Section 2 of how to handle Modia models where the number of states changes after the model code is generated and compiled but before simulation begins. In Sections 3 and 4, a new, general method is described in which states and other variables can be introduced and removed during simulation without having to re-generate and re-compile the model code. This approach is specialized for 3D mechanical systems in Section 5, and two applications of multibody systems with dynamically changing degrees of freedom are presented in Section 6. Additionally, in Appendix A, a short overview of Modia is given, and in Appendix B, other examples with the new approach are provided.

2. Changing Number of States after Model Translation

In this section, an overview is given of how the symbolic and simulation engine of Modia can treat models where the number of states can be changed after generation and compilation of the model's code and before simulation is started. Appendix A gives an overview of the modelling language Modia. In Section 4, models are treated in which the number of states can be changed during simulation. All this is done without re-translation, and the simulation speed is therefore hardly influenced when using these new techniques.

Traditional object-oriented modelling languages, in particular, the Modelica language, define variable types and array sizes precisely. This information is used by the symbolic engine when generating code that is compiled into binary form. Modia does this differently in order to take full advantage of the Julia language. In particular, the goal is that Modia models can use all variable types that can be described with Julia. Since Julia has a very rich type system with type inference, it makes no sense that a language such as Modia tries to replicate this very powerful underlying engine. As a consequence, the Modia language and Modia's symbolic engine do not have the complete information about the variable types because a variable type can be determined in the Julia compiler inference pass.

For this reason, a Modia model basically defines a set of unknown variables, without necessarily knowing the types of these variables, and a set of equations, along with a set of known variables (=parameters). Note, an unknown variable might also be an instance of a mutable Julia struct, as will be shown below. The basic requirement is that the number of unknown variables and the number of equations must be equal. For example, if a variable is a vector, then an equation must be present that is able to compute this vector. Whether this requirement is fulfilled or not might only be detected by the Julia compiler during compilation of the generated model code, or even only during execution of the model code. Furthermore, a variable is typically treated as one symbol and the associated equation as one symbol-equation, even if the symbol is an array. Two examples are given in Listing 1.

Listing 1. An array equation must be defined for an array variable and an array must be declared with an init or start array value when its sizes cannot be inferred.

```
# Correct code
m1 = Model(v = Var(init=zeros(2)),          # size of v cannot be inferred,
                                            # therefore, init needed
           equations = :[a = der(v)         # size of a can be inferred
                         m*a = [2.0, 3.0]]  # array equation for a
)

# Wrong code
m2 = Model(equations = :[wd = der(w)      # sizes of w,wd cannot be inferred
                         m*wd[1] = 2.0    # no array equation for wd
                         m*wd[2] = 3.0])
```

As a consequence, the size of a variable typically has no influence on the symbolic engine or the code generation: The generated equations are basically the same, whether a variable is a scalar or has, let us say, 1000 elements. Note, all this is different to current Modelica tools, where variables and equations are typically scalarized before symbolic processing takes place (e.g., a vector of 1000 elements is replaced by 1000 scalars, so 1000 symbols are used in the symbolic engine).

Array sizes of parameters and of variables defined with `init` or `start` attributes can be changed in Modia after code generation, provided they are not defined as static arrays. Some examples are given in Listing 2.

Listing 2. Examples of array equations. The sizes of the statically sized arrays `A1`, `y1` cannot be changed after compilation. The sizes of arrays `A2`, `y2` can be changed after compilation.

```
using StaticArrays

LinearODEs = Model(
    A1 = parameter | SMatrix{2,2}([-1.0  0.0;   # statically sized matrix
                                   0.0 -2.0]),
    A2 = parameter | [-1.0  0.0;                # variable sized matrix
                      0.0 -2.0],
    y1 = Var(init = SVector{2}(1.0, 2.0)),      # statically sized vector
    y2 = Var(init=[1.0, 2.0]),                  # variable sized vector
    equations = :[der(y1) = A1*y1               # static array equation
                  der(y2) = A2*y2])             # variable array equation

linearODEs = @instantiateModel(LinearODEs)  # generate and compile code
simulate!(linearODEs, stopTime = 2,
            merge = Map(A2 = [-1.0  0.0  0.0;   # change sizes of A2
                               0.0 -2.0  0.0;
                               0.0  0.0 -3.0],
                        y2 = [1.1, 2.1, 3.1]))  # change size of y2

# Generated code (simplified)
function getDerivatives(_der_x, _x, _m, _time)
  ...
  _p = _m.evaluatedParameters   # _p is a hierarchical dictionary
  A1 = _p[:A1]::SMatrix{2,2,Float64,4} # _p[:A1] is the value of symbol :A1
  A2 = _p[:A2]::Matrix{Float64}  # ::Matrix{Float64} is the type of _p[:A2]
  y1 = SVector{2,Float64}(_x[1], _x[2])
  y2 = _m.x_vec[1]
  var"der(y1)" = A1 * y1  # var"der(y1)" is a macro defining name "der(y1)"
  var"der(y2)" = A2 * y2
  ...
end
```

`A1`,`y1` are statically sized arrays, and their dimensions cannot be changed after `@instantiateModel(..)` has been called. In contrast, `A2`,`y2` are standard Julia arrays, and their dimensions can be changed with the `merge` attribute of the `simulate!(..)` command after compilation of the generated `getDerivatives` function.

In the generated function `getDerivatives`, the statically sized state vector `y1` is always newly generated by utilizing the corresponding elements from the model state vector `_x` in

the `SVector` constructor. This constructor allocates memory on the *stack*, and operations on `y1` are efficiently implemented in package `StaticArrays`. New arrays needed in calculations are automatically constructed again on the stack (which is an efficient operation). For example, Julia's multiple dispatch feature will deduce at compile time, that `A1 * y1` is a static array (because `A1` and `y1` are static arrays and the operator `*` is overloaded to return a static array), and therefore, `var"der(y1)"` is generated on the stack as a static array.

Auxiliary memory `_m.x_vec[1]` is allocated for the state vector `y2` whenever the `merge`-attribute has been processed. Before function `getDerivatives` is called, the corresponding elements of the model state vector `_x` are copied into `_m.x_vec[1]`, and this vector is then accessed with the alias name `y2` due to `y2 =_m.x_vec[1]` (`y2` is a reference to `_m.x_vec[1]`). The generated array equations' code does not depend on the array sizes of the involved variables. The drawback of non-static arrays is that intermediate computations, and left-hand side variables such as `var"der(y2)"`, allocate new memory on the heap, whenever the corresponding statements are executed. If there are many such statements, this can reduce the efficiency of the simulation. In the next section, built-in components are introduced that do not have this drawback. Arrays used in functions of built-in components operate on memory that is allocated once on the heap and not in every model evaluation.

3. Built-In Components

In this and the next section, a new, general method for handling equation-based models is described, where states and other variables can be introduced and removed during simulation without re-generation and re-compilation of the model code. The method is described in a generic, mathematical way and is practically demonstrated with an implementation in Julia/Modia. In principle, the method can also be used for other modelling systems, for example, in an extended version of Modelica.

To simplify the description and focus on the new technique, time-discrete systems, time events, event iteration, and super-dense time (see, e.g., [1] Appendix B and [18] Section 3.1) are not discussed in this article. However, in the Modia implementation, these features are included.

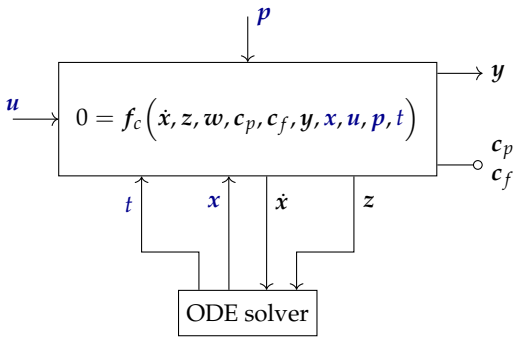
3.1. Acausal Components

Declarative, equation-based modelling languages, such as Modelica or Modia, define acausal component models as shown in Figure 1. In this context, acausal means that interface variables are present that are neither inputs nor outputs of the component ($\boldsymbol{c}_p, \boldsymbol{c}_f$ in Figure 1). Instead, the connection between a component and the symbolic transformation defines the order equations are evaluated, and whether an interface variable is provided to the component equations or is computed by the component equations.

In Figure 1, the following definitions are used: Let $\mathbb{R}$ be the set of real numbers and assume $k \in \mathbb{N}_0$. $C^k(\mathbb{R})$ is the space of functions which are bounded and k-times continuously differentiable in $\mathbb{R}$; see, e.g., [19]. This means $C^1(\mathbb{R})$ is the function space of 1-time continuously differentiable functions. $C^0(\mathbb{R})$ is the function space of continuous functions. Furthermore, due to events, there are non-continuous jumps. $C^1_{\text{pw}}(\mathbb{R})^n$ is the space of piecewise (pw) one-time continuously differentiable functions in n dimensions, and $C^0_{\text{pw}}(\mathbb{R})^n$ is the space of piecewise (pw) continuous functions in n dimensions. Figure 1 shows the minimal smoothness requirements of the variables. Depending on the structure of the equations $0 = \boldsymbol{f}_c(\ldots)$ and how the component is connected with other components, higher smoothness might be required; see, e.g., [20].

An acausal component, according to Figure 1, consists of a set of implicit equations $0 = \boldsymbol{f}_c(\ldots)$, e.g., the equations in the `equation` section of a Modelica model or the equations in the quoted vector that are assigned to variable `equations` in a Modia model. It can be connected with other components via inputs $\boldsymbol{u}$, outputs $\boldsymbol{y}$, and connectors containing pairs of potential $\boldsymbol{c}_p$ and flow variables $\boldsymbol{c}_f$. As usual, when connecting potential and flow variables via connectors, the corresponding potential variables are set equal, and the sum of the corresponding flow variables is set to zero. It is required that a connector has only

equal pairs of potential and flow variables to ensure that any connection of acausal components is globally balanced. In other words, the number of equations and the number of unknowns of any set of connected components is equal, provided every component is locally balanced. A component is called locally balanced if $\dim(\boldsymbol{c}_p) = \dim(\boldsymbol{c}_f) = n_c$ and $\dim(\boldsymbol{f}_c) = n_y + n_c + n_w + n_z$. For details, see [21].



$t \in \mathbb{R}$	time
$\boldsymbol{p} \in \mathbb{R}^{n_p}$	parameters
$\boldsymbol{u}(t) \in C^0_{\text{pw}}(\mathbb{R})^{n_u}$	inputs
$\boldsymbol{x}(t) \in C^1_{\text{pw}}(\mathbb{R})^{n_x}$	continuous states
$\boldsymbol{y}(t) \in C^0_{\text{pw}}(\mathbb{R})^{n_y}$	outputs
$\boldsymbol{c}_f(t) \in C^0_{\text{pw}}(\mathbb{R})^{n_c}$	flow variables
$\boldsymbol{c}_p(t) \in C^0_{\text{pw}}(\mathbb{R})^{n_c}$	potential variables
$\boldsymbol{w}(t) \in C^0_{\text{pw}}(\mathbb{R})^{n_w}$	local variables
$\boldsymbol{z}(t) \in C^0_{\text{pw}}(\mathbb{R})^{n_z}$	event indicators
$\dim(\boldsymbol{f}_c) = n_y + n_c + n_w + n_z$	

Figure 1. Mathematical description of an acausal component. Components can be connected via $\boldsymbol{u}, \boldsymbol{y}, \boldsymbol{c}_p, \boldsymbol{c}_f$. Events are defined by zero crossings of elements of $\boldsymbol{z}$. At events, variable values can change discontinuously. $\mathbb{R}$ is the set of real numbers. $C^k_{\text{pw}}(\mathbb{R})^n$ is the space of piecewise (pw) k-times continuously differentiable functions in n dimensions. $0 = \boldsymbol{f}_c(\ldots)$ is a set of implicit equations. Variables in dark blue are assumed to be known: variables $t, \boldsymbol{x}$ are provided by the ODE solver, $\boldsymbol{p}$ are parameters that get constant values before simulation starts, $\boldsymbol{u}$ are inputs and are provided externally to the component. There are n_c equations missing in order to solve $0 = \boldsymbol{f}_c(\ldots)$ for its unknowns $\dot{\boldsymbol{x}}, \boldsymbol{z}, \boldsymbol{w}, \boldsymbol{c}_p, \boldsymbol{c}_f, \boldsymbol{y}$. These missing equations are provided when connecting the component via the connection equations of $\boldsymbol{c}_p, \boldsymbol{c}_f$.

The equations of all components of a model, together with the connection equations, form a set of DAEs. The set of DAEs is transformed into a set of ODEs and is solved by an ODE integrator. In every model evaluation, the time t and the continuous states $\boldsymbol{x}$ are supplied to the model by the solver. Using the symbolically processed equations, the derivative of the states $\dot{\boldsymbol{x}}$ and the event indicators $\boldsymbol{z}$ are calculated and returned to the solver. Hereby, linear and/or non-linear algebraic equation systems might be solved within the model; see, e.g., [1] Appendix B.

3.2. Acausal Components with Pre-translated Mathematical Functions

Elmqvist [22] proposes a generic method to split the equations of an acausal component (see Figure 1) into causal and acausal partitions. The intuition is that the causal partitions are always evaluated in the same order, regardless of how the component of Figure 1 is connected with other components. These partitions are sorted, explicitly solved for the unknowns, and pre-translated. In contrast, the evaluation order of the acausal partition depends on the actual connection of the component, and this partition is kept as an implicit equation system. The method of Elmqvist defines the causal partitions as mathematical functions where the states and state derivatives are known in the calling environment. It is then possible that the symbolic engine differentiates these functions if needed. In this article a variant of this method is used, where the information about states and state derivatives is hidden in these functions and then a symbolic engine cannot differentiate these functions any more, resulting in limitations on how a component can be utilized in an overall model. The benefit of this variant is that the number of states can be changed during simulation without influencing the symbolic transformation and code generation of the overall model.

For certain cases, it is possible to find better partitioning (smaller acausal part) if special connection topologies are being considered. For example, a component without potential and flow variables (input/output block) is usually used in a way that the inputs are provided externally, and the outputs are computed from the component equations. Partitioning is then performed for this common situation. All equations can be sorted and

solved for the unknowns, and the entire code can be pre-translated. When the inverse of an input/output block shall be determined, the outputs are provided externally, and the inputs are computed from the component equations. It could be that this inverse model cannot be determined from a pre-translated block, because it may be necessary to differentiate equations and this is not possible if the information about the states is hidden in the pre-translated block as done below.

If all code of a pre-translated block is included in one function, an implicit equation system might occur when connecting the block, whereas no implicit equation system might occur if the code is included in two functions. For an example, see [18] Figure 5. This example demonstrates that even if a partitioning in causal and acausal parts is made, there is still the difficulty of deciding whether to put all causal code in one or in several functions.

If potential and flow variables are present in a component, common connection scenarios are that either the potential or flow variables or a combination of both are provided externally. In order to prepare for all these cases, n_c implicit dummy equations

$$0 = \boldsymbol{g}_c(\boldsymbol{c}_p, \boldsymbol{c}_f), \tag{1}$$

are defined. Every element of every argument appears in every equation of $\boldsymbol{g}_c$, so (1) has full incidence of all of its arguments. Mathematically, (1) defines a large set of potential connection possibilities. Note, more general scenarios could be treated, if $\boldsymbol{u}$ and $\boldsymbol{y}$ would be arguments of $\boldsymbol{g}_c$ too. However, this results usually in a larger acausal part. Sorting the following equations,

$$0 = \begin{bmatrix} \boldsymbol{f}_c\left(\dot{\boldsymbol{x}}, \boldsymbol{z}, \boldsymbol{w}, \boldsymbol{c}_p, \boldsymbol{c}_f, \boldsymbol{y}, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, t\right) \\ \boldsymbol{g}_c(\boldsymbol{c}_p, \boldsymbol{c}_f) \end{bmatrix} \tag{2}$$

under the assumption that $\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, t$ are known and utilizing only structural information (whether a variable appears or does not appear in an equation, see, e.g., (https://modiasim.github.io/ModiaBase.jl/stable/Tutorial.html, accessed on 11 December 2022, Sections 1.1–1.4), results in a sorted set of equations that has at least one implicit equation system,

$$0 = \begin{bmatrix} \boldsymbol{f}_{c,\mathrm{eq}}(\ldots) \\ \boldsymbol{g}_c(\ldots) \end{bmatrix} \tag{3}$$

that cannot be split into smaller implicit equation systems by sorting. All equations of (1) are included in (3) because (1) has full incidence. Equations $0 = \boldsymbol{f}_{c,\mathrm{eq}}$ are a subset of $\boldsymbol{f}_c$ and form the acausal part of the component because these equations are needed to compute all the arguments of $\boldsymbol{g}_c$, and so the potential and flow variables of Figure 1. All other sorted equations can be explicitly solved (possibly by solving linear and/or non-linear equation systems) and packed into functions $\boldsymbol{f}_{c,i}(\ldots)$ that are called either before or after (3). Removing (1) from the sorted equations results in Figure 2.

Note, the unknown variables $\dot{\boldsymbol{x}}, \boldsymbol{z}, \boldsymbol{w}, \boldsymbol{c}_p, \boldsymbol{c}_f, \boldsymbol{y}$ of Figure 1 are split into three parts; for example, $\boldsymbol{y} = (\boldsymbol{y}_1, \boldsymbol{y}_{\mathrm{eq}}, \boldsymbol{y}_2)$, where $\boldsymbol{y}_1$ is an output argument of function $\boldsymbol{f}_{c,1}$, $\boldsymbol{y}_{\mathrm{eq}}$ is computed from the implicit equation system $0 = \boldsymbol{f}_{c,\mathrm{eq}}(\ldots)$, and $\boldsymbol{y}_2$ is an output argument of function $\boldsymbol{f}_{c,2}$. Furthermore, $\boldsymbol{f}_{c,1}, \boldsymbol{f}_{c,2}$ might be split in several functions, depending on the expected usage scenarios. When the acausal component of Figure 2 is connected with other components and the overall model is *sorted*, $\boldsymbol{f}_{c,1}$ is called before equations $0 = \boldsymbol{f}_{c,\mathrm{eq}}(\ldots)$ are evaluated, because all output arguments of $\boldsymbol{f}_{c,1}$ are (possible) arguments of $\boldsymbol{f}_{c,\mathrm{eq}}$. Function $\boldsymbol{f}_{c,2}$ is called afterwards, because variables computed by $\boldsymbol{f}_{c,1}$ and $\boldsymbol{f}_{c,\mathrm{eq}}$ are (possible) input arguments of $\boldsymbol{f}_{c,2}$.

The big advantage of an acausal component according to Figure 2 is that functions $\boldsymbol{f}_{c,1}, \boldsymbol{f}_{c,2}$ can be pre-translated once in advance, so that symbolically processing an overall

model, and generation and compilation of code, can be made much more efficiently as with the original formulation of Figure 1.

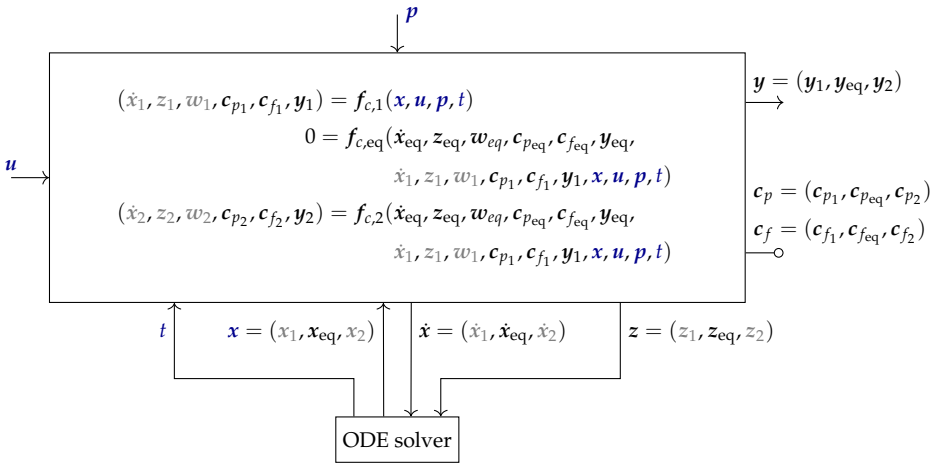


Figure 2. Mathematical description of an acausal component with pre-translated functions with the same interface as in Figure 1. These components can be connected via u, y, c_p, c_f. Arguments given in grey will be moved into an internal memory in Section 3.3. $f_{c,1}, f_{c,2}$ are explicit functions, whereas $0 = f_{c,\mathrm{eq}}(\ldots)$ is a set of implicit equations. Variables $\dot{x}, z, w, c_p, c_f, y$ are unknown and are split into three parts, e.g., $y = (y_1, y_{\mathrm{eq}}, y_2)$, where y_1 is an output argument of function $f_{c,1}$, y_{eq} is computed from the implicit equation system $0 = f_{c,\mathrm{eq}}(\ldots)$, and y_2 is an output argument of function $f_{c,2}$.

3.3. Acausal Built-In Components

In order that the number of variables, and especially the number of states, can change during simulation *without* re-generating and re-compilation of code, the scheme from the previous subsection is changed by storing the grey variables of Figure 2 inside an internal memory m; see Figure 3.

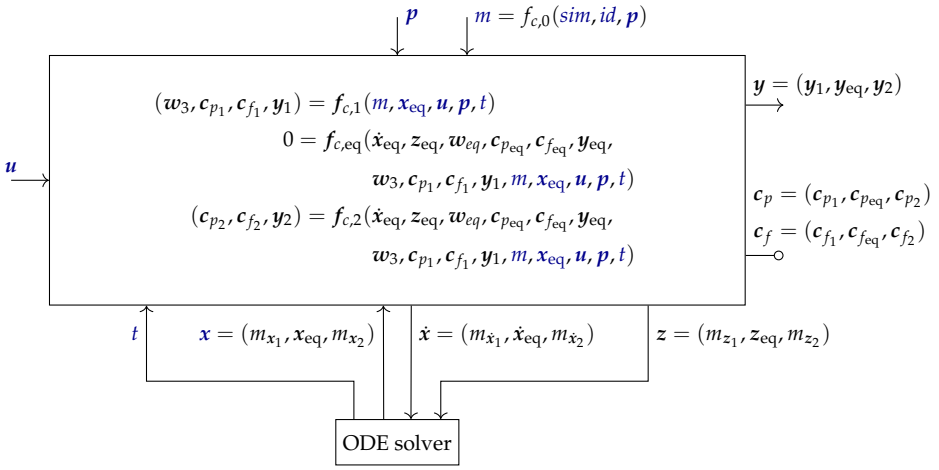


Figure 3. Description of an acausal built-in component with functions of a programming language that has internal memory and with the same interface as in Figure 1. $m = f_{c,0}(\mathrm{sim}, \mathrm{id}, p)$ is an instance of the built-in component. It is constructed before simulation starts given a reference to the simulation engine sim, a unique identification id of the instance, and the parameters p. The hidden state derivatives $m_{\dot{x}_1}$ are computed in $f_{c,1}$ and $m_{\dot{x}_2}$ in $f_{c,2}$. In some cases, (new) local, algebraic variables w_3 need to be introduced to hide, e.g., state variables in $f_{c,\mathrm{eq}}$.

In the sequel, such components are called acausal built-in components. Contrary to the previous figures in this section, $f_{c,1}, f_{c,2}$ in Figure 3 are no longer mathematical functions but functions of a programming language where argument m is both an input and an output argument to the respective function. The memory m is exchanged between

the functions $f_{c,1}, f_{c,2}$. Function $f_{c,1}$ copies states x_1, x_2 from the solver into m. Functions $f_{c,1}, f_{c,2}$ copy the state derivatives $\dot{x}_1, \dot{x}_2$ and the event indicators z_1, z_2 from these functions to the solver.

One issue is that the states $\boldsymbol{x}$ and the output variables $\dot{x}_1, z_1, w_1$ of function $f_{c,1}$ are (possibly) present in $f_{c,\text{eq}}$, as seen in Figure 2. It would then not be possible to add or remove these variables during simulation without re-translation. In such a case, re-formulations are needed, e.g., by computing some part of the expressions present in $f_{c,\text{eq}}$ in a function that has the memory m as argument, so that variables $\dot{x}_1, z_1, w_1, x_1, x_2$ are no longer visible in $f_{c,\text{eq}}$. It might also be necessary to hide some of these variables in (new) local, algebraic variables w_3 that are returned from $f_{c,1}$ and used in $f_{c,\text{eq}}$; see Figure 3.

The big benefit of an acausal built-in component is that the hidden states x_1, x_2 and their derivatives are not visible in the model equations. Consequently, it is in principle possible to change the number of states during simulation, as is shown in Section 4. Furthermore, the code parts inside functions $f_{c,1}, f_{c,2}$ are pre-translated and present once, independent of the number of instances of the built-in component that are used in a model. As a result, the effort to symbolically process and translate the equations can be drastically reduced. A drawback of acausal built-in components with internal memory m is that they might be used in a way requiring one to differentiate functions $f_{c,1}, f_{c,2}$, which is not possible due to this memory.

A simple acausal built-in component of an electrical capacitor is discussed in Section 3.4. A more involved acausal built-in component describing heat transfer in a rod is discussed in Appendix B.1.

3.4. Application: Capacitor

In Figure 4, an equation-based model of a capacitor is shown that is defined by three equations. A Modia implementation of this model is given in Appendix A.2.

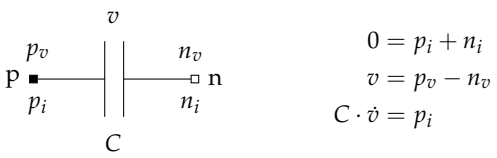


Figure 4. Equation-based model of a capacitor with parameter C, state v, connectors p, n with potential variables p_v, n_v (electrical potentials), and flow variables p_i, n_i (electric currents).

Additionally, the same model is defined as (a) an acausal component with mathematical functions and (b) as a built-in component with functions of a programming language (that have internal memory) in the form of pseudo-code in Table 1. Note, function $f_{c,0}$ of the built-in component is called once during setup of the simulation run. This function allocates a record or a struct that holds the internal memory m for the component and copies the parameters into this memory.

Table 1. The capacitor is defined as a component with mathematical functions (left column) and with functions that have an internal memory (right column) to hide the state and the state derivative of the component in the equation section.

as component (with math. functions)	**as built-in component** (with functions of a prog. language that have internal memory m)
equation section	
$w = f_{c,1}(v)$	$w = f_{c,1}(m)$
$w = p_v - n_v$	$w = p_v - n_v$
$0 = p_i + n_i$	$0 = p_i + n_i$
$\dot{v} = f_{c,2}(C, p_i)$	$f_{c,2}(m, p_i)$ # prog. language function without return argument
function definitions	
function $f_{c,1}(v)$ **return** v	**function** $f_{c,0}(\text{sim}, \text{id}, C)$ # called once < allocate new record m > $m_{\text{sim}} := \text{sim}$; $m_{\text{id}} := \text{id}$; $m_C = C$; **return** m
function $f_{c,2}(C, p_i)$ **return** p_i / C	**function** $f_{c,1}(m)$ < copy m_v from states in m_{sim} for m_{id} >; **return** m_v
	function $f_{c,2}(m, n_i)$ $m_{\dot{v}} := p_i / m_C$; < copy $m_{\dot{v}}$ into state derivatives of m_{sim} for m_{id} >

This capacitor model is just used as demonstration of the principle, due to its simplicity. The formulation as a built-in component does *not* give an advantage, because the equation section has four equations, and the function bodies are tiny, whereas the pure equation-based model consists of three equations. Note, when two capacitors defined as built-in components with internal memory are connected in parallel, then both capacitors return the respective state w_1, w_2 from function $f_{c,1}$. Since the parallel connection introduces an equation $w_1 = w_2$, an implicit equation system with three equations for two unknowns is present. Therefore, such a model will be rejected (the issue is that due to the parallel connection, w_2 can be computed from w_1, but then w_2 cannot be a state as defined in the built-in component).

4. Changing the Number of States during Simulation

In order that the number of states can be changed during simulation (without newly generating and translating code) the generic concept sketched in Figure 5 is used. The variables of the solver (state vector $\boldsymbol{x}$ and the vector of event indicators $\boldsymbol{z}$) are split into an invariant and a variant part: $\boldsymbol{x} = (\boldsymbol{x}^{\text{inv}}, \boldsymbol{x}^{\text{var}})$ and $\boldsymbol{z} = (\boldsymbol{z}^{\text{inv}}, \boldsymbol{z}^{\text{var}})$. The dimensions of the invariant parts were fixed before the simulation starts. The dimensions of the variant parts can change during events in the simulation. The variables of a model are characterized by the following attributes:

- Invariance (inv): Variable name, type, and number of dimensions are fixed before translation. The dimensions of an invariant variable (e.g., length of a vector) can be changed before simulation starts. The solver provides vector $\boldsymbol{x}^{\text{inv}}$ to the model function that contains the sorted and solved equations. The elements of $\boldsymbol{x}^{\text{inv}}$ are copied into the elements of the invariant state variables of the model. The computed derivatives of the invariant state variables are copied into vector $\dot{\boldsymbol{x}}^{\text{inv}}$ and the computed invariant event indicators are copied into $\boldsymbol{z}^{\text{inv}}$ before the model's function returns. All variables defined and used in an equation section are invariant variables. This includes all input/output arguments of the called functions.
- Variant (var): Variables can appear and existing variables can disappear *during events*. The type and the number of dimensions of a variant variable cannot be changed after it is first introduced in a simulation run. However, its dimensions (e.g., the length of a vector) can be changed at event initiation. All model variables defined inside functions of built-in components are variant variables. Before a variant state variable of the model is used in a function of a built-in component, its elements are retrieved

from vector $\boldsymbol{x}^{\text{var}}$ provided by the solver. After the derivatives of variant state variables of a model are computed, they are copied into vector $\dot{\boldsymbol{x}}^{\text{var}}$ provided by the solver.

As defined above, all variables present in the sorted and solved equations, including the variables that are input/output arguments of built-in component functions, must be invariant variables. Under this restriction, it is possible to sort and solve the equations of all components/built-in components and generate/translate code, provided (a) the symbolic transformation algorithms treat an array as one symbol during the assignment phase, as sketched in Section 2, and (b) no function of a built-in component needs to be differentiated. Note, this implies that all names, types, and number of dimensions of all interface variables of a component ($\boldsymbol{u}, \boldsymbol{y}, \boldsymbol{p}, \boldsymbol{c}_p, \boldsymbol{c}_f$ of Figure 1) are fixed before translation starts and cannot be changed after translation.

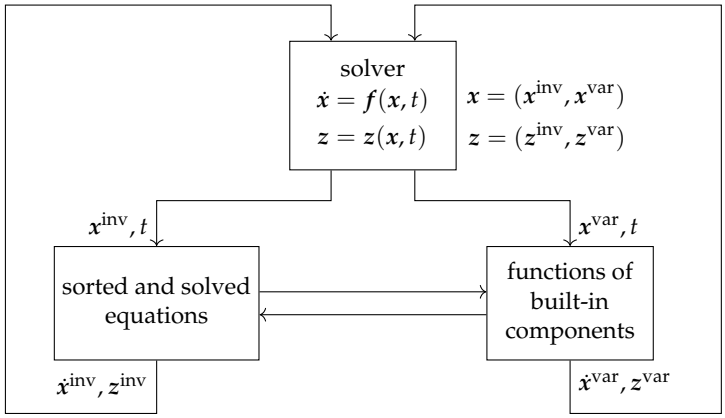


Figure 5. Communication between the solver, the sorted and solved equations, and the functions of the built-in components. The state vector $\boldsymbol{x}$ and the event indicators $\boldsymbol{z}$ are split into an invariant and a variant part: $\boldsymbol{x} = (\boldsymbol{x}^{\text{inv}}, \boldsymbol{x}^{\text{var}})$, $\boldsymbol{z} = (\boldsymbol{z}^{\text{inv}}, \boldsymbol{z}^{\text{var}})$. The variant parts consist of $\boldsymbol{x}_{j,1}, \boldsymbol{x}_{j,2}, \boldsymbol{z}_{j,1}, \boldsymbol{z}_{j,2}$ from all built-in components j (see Figure 3) present in the overall model. The dimensions of the invariant parts are fixed before simulation starts. The dimensions of the variant parts can change at events during simulation.

A simulation run is partitioned into phases that are called segments or modes, as sketched in Figure 6.

A run starts with mode $i = 1$, and the corresponding system is initialized with the initial states $\boldsymbol{x}_{1,0} = (\boldsymbol{x}_0^{\text{inv}}, \boldsymbol{x}_{1,0}^{\text{var}})$. The ODE $\dot{\boldsymbol{x}}_i = \boldsymbol{f}_i(\boldsymbol{x}_i, t)$ of actual mode i is solved until either a termination condition is reached or a full restart (FR) event indicator $z_{\text{FR}_i,j}$ becomes positive. In the latter case, the system switches to the next mode $i + 1$ with potentially new equations and potentially different variant states than in the previous mode. Initial values $\boldsymbol{x}_{i+1,0}$ in mode $i + 1$ are (a) the invariant states at the current time instant t of mode i, $\boldsymbol{x}^{\text{inv}}(t)$; and (b) initial variant states that are computed from the states of mode i with function $\boldsymbol{h}_i$. Reinitialization is a complex topic because Dirac impulses can occur. For more details, see [11] Section 4, in which a general reinitialization method for a large class of multi-mode systems is presented. In all the examples discussed in this article, Dirac impulses do not occur, so reinitialization in these cases is straightforward. The re-initialized mode is solved until it is terminated or another full restart event is triggered. Note, the number of modes is usually not known in advance.

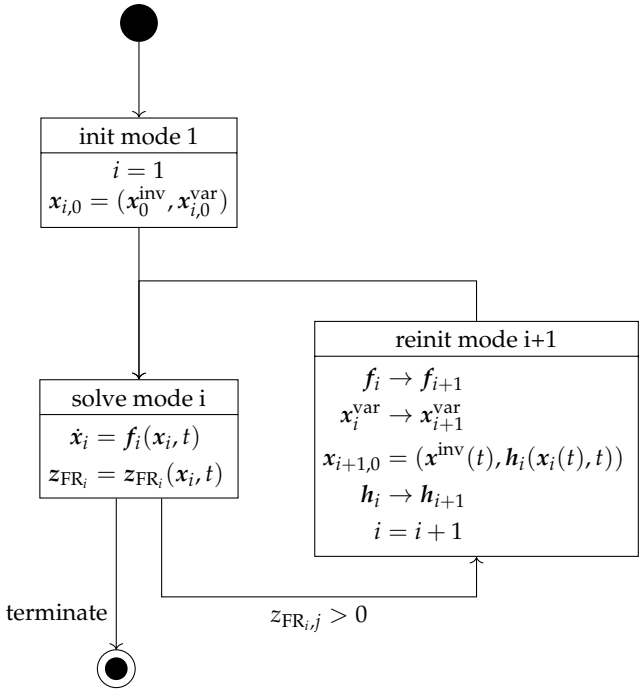


Figure 6. State machine of segmented simulation. The first mode $i = 1$ is initialized with its start values. The ODEs of the actual mode i are solved until they are terminated or interrupted by a *full restart* zero-crossing event indicator $z_{\text{FR}_i,j}$ that becomes positive. In the latter case, the model is re-initialized in mode $i + 1$. Hereby, variant variables can appear or disappear. The number of modes is unknown beforehand.

Since variables may appear and disappear at events and theses changes are not known in advance, new schemes are needed to store result data. In [23], such a proposal is given, introducing signal tables as a format for exchanging data associated with simulations based on dictionaries and multi-dimensional arrays with `missing` values. This format was developed and evaluated with the open source Julia package SignalTables (https://github.com/ModiaSim/SignalTables.jl, accessed on 12 December 2022) and is used in Modia.

The general scheme presented above is implemented in Modia. Some details of the Modia implementation are shown in Figure 7. Typically, whenever the number of equations changes at an event, some internal data structure must be updated that is used to efficiently compute the equations of a built-in component. This internal data structure is constructed from the dictionary of the Modia `Model` that defines the interface and the equations of the built-in component. This data structure and the functions to evaluate it are called the execution scheme in Figure 7.

When the model is initialized in mode $i = 1$, the execution scheme for mode $i = 1$ is defined, together with the variant states and their start values. The ODE $\dot{\boldsymbol{x}}_i = \boldsymbol{f}_i(\boldsymbol{x}_i, t)$ of the actual segment i is solved for $t \in [t_{i,0}, t_{\text{stop}}]$ and its start values $\boldsymbol{x}_{i,0} = (\boldsymbol{x}_0^{\text{inv}}, \boldsymbol{x}_{i,0}^{\text{var}})$ until t_{stop} is reached, or an event for a structural change is triggered. In the latter case, *actions* are defined and stored internally in the built-in components that define how to construct mode $i + 1$. More details and examples for execution schemes and actions are given in Section 5.3 and Appendix B.2. When the built-in component is reinstantiated in mode $i + 1$, the execution scheme is redefined, together with new states $\boldsymbol{x}_{i+1} = (\boldsymbol{x}^{\text{inv}}, \boldsymbol{x}_{i+1}^{\text{var}})$ and their start values $\boldsymbol{x}_{i+1,0} = (\boldsymbol{x}^{\text{inv}}(t), \boldsymbol{x}_{i+1,0}^{\text{var}})$. Afterwards, the ODE for segment $i + 1$ is solved. Applications of changing number of states during simulation are given in Section 6 and Appendix B.2.

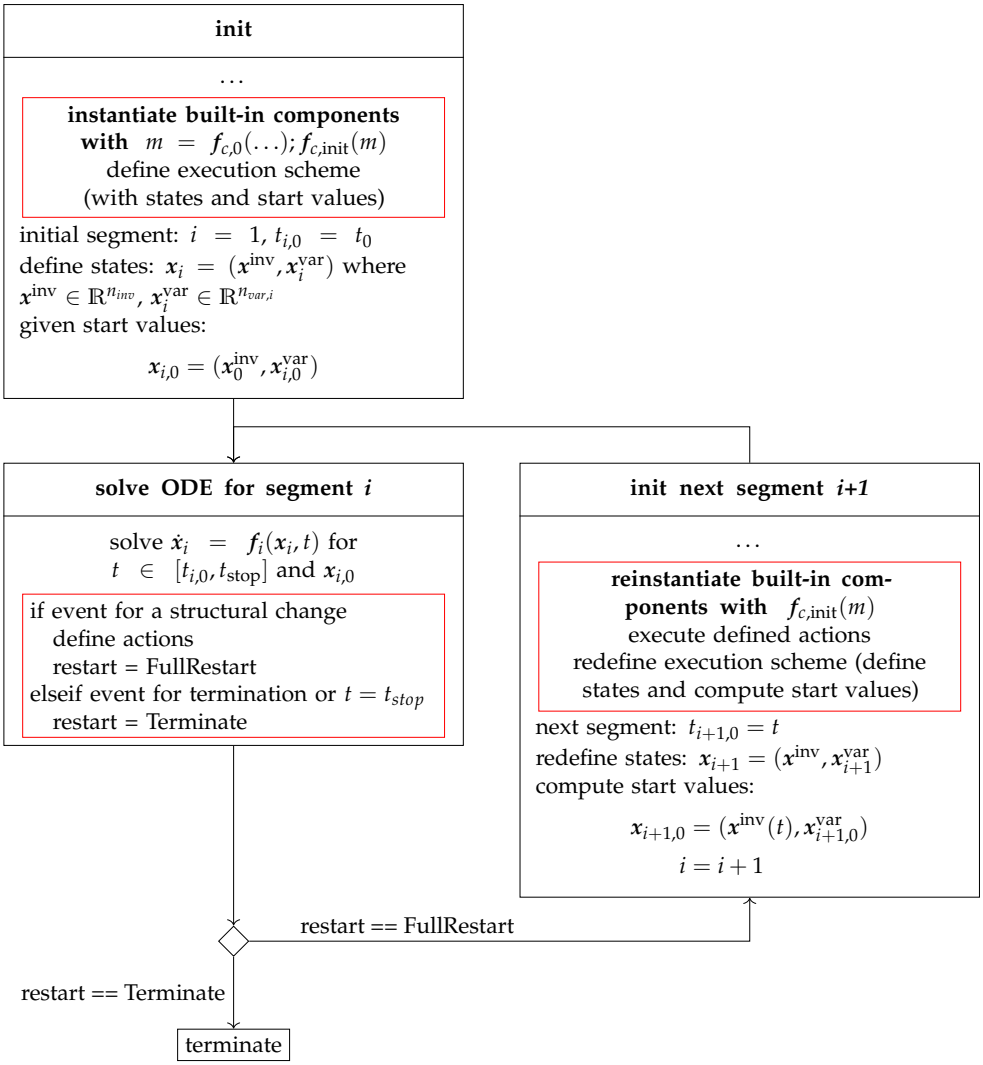


Figure 7. State machine of the segmented simulation as used by Modia.

5. Segmented Simulation with Built-In Component Modia3D

5.1. Overview Modia3D

The open source Julia package Modia3D (https://github.com/ModiaSim/Modia3D.jl, accessed on 17 January 2023, release v0.12.0) is a multibody tool for 3D mechanical systems implemented as a built-in component of Modia and can therefore be combined with other Modia components. Modia3D is targeted for solvers with adaptive step-size control to compute results close to real physics including collision handling using the Minkowski portal refinement (MPR) algorithm [24,25] and collision response for elastic contacts [26–28]. Furthermore, it is inspired by the generic component-based design pattern of modern game engines, allowing very flexible and modular definitions of 3D systems: A coordinate system located in 3D is used as a container with optional components (geometry, solid and collision properties, visualization data, light, camera, etc.); see [29], Unity [30], Unreal Engine [31], and three.js [32].

The core component of Modia3D is an *Object3D*. It is a coordinate system moving in 3D with associated optional features; see Figure 8. An Object3D's position and orientation is defined relative to an optional parent Object3D by translation and rotation. The Object3D with feature `Scene` is the root of all other Object3Ds and defines a global inertial system. The feature `Visual` is for 3D animation and defines shapes such as box, sphere, cylinder, beam, and 3D meshes with visualization properties. The feature `Solid` defines solid bodies. It has mass properties and can be considered in collision situations if `collision = true` is set. It can have a shape and visualization properties. For a more extended description, see [4]

and the Modia3D Tutorial (https://modiasim.github.io/Modia3D.jl/stable/, accessed on 11 December 2022).

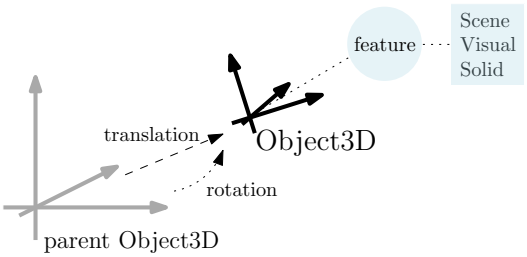


Figure 8. Object3D defined relative to its parent with translation and rotation. An Object3D can have one optional feature: Scene, Visual or Solid.

An example of a simple pendulum model (https://github.com/ModiaSim/Modia3D.jl/blob/main/test/Tutorial/Pendulum3.jl, accessed on 14 December 2022) with damping in its joint is given in Listing 3 with the already described features of the Modia3D built-in component. The remaining elements of the Pendulum use predefined models of a small Modia library: the Modelica.Mechanics.Rotational library. In particular, a rotational 1D `Damper` is connected to a fixed point and to the flange of the revolute joint, see Listing 4, to model damping in the joint.

Listing 3. Simple damped pendulum defined with constructors of Modia3D (`Object3D, RevoluteWithFlange`) and equation based components of Modia (`Damper, Fixed`).

```
Pendulum = Model3D(
    world = Object3D(feature=Scene()),
    obj1  = Object3D(feature=Solid(solidMaterial="Steel", collision=true,
                shape=Beam(length=1.0, width=0.2, thickness=0.2, <...>))),
    obj2  = Object3D(parent=:obj1, translation=[-0.5, 0, 0],
                feature=Visual(shape=Cylinder(diameter=0.1, length=0.21),
                visualMaterial = VisualMaterial(color="Red"))),
    rev   = RevoluteWithFlange(obj1=:world, obj2=:obj2),

    # Equation based components
    damper = Damper | Map(d=100.0),
    fixed  = Fixed,
    equations = :[connect(damper.flange_b, rev.flange),
                  connect(damper.flange_a, fixed.flange)]
)
pendulum = @instantiateModel(Pendulum, <...>)
simulate!(pendulum, stopTime=3.0)
```

Listing 4. Definition of a Modia3D revolute joint containing a Modia 1D rotational flange that can be connected with Modia components.

```
Flange = Model(phi=Var(potential=true), tau=Var(flow=true))
RevoluteWithFlange(; obj1, obj2, <...>) = Model(; _constructor = <...>,
    flange = Flange,
    equations = :[
        phi = flange.phi
        w   = der(phi)]
)
```

Modia3D offers two kinds of joints: The first kind of joints contains Modia equation sections with invariant variables, including invariant states, according to Figure 5. These joints are visible for Modia and cannot be removed or added during simulation. In order that state constraints can be defined and index reduction on invariant states can be performed, the interface to the Modia3D functionality is designed to define differential equations only on the Modia side in Modia equation sections. The definition of a `RevoluteWithFlange` joint (a revolute joint that has a Modia 1D rotational `Flange`) is shown in Listing 4; for more

details, see [4]. During the instantiation of an overall model, the model is traversed, and each built-in component can inject equations into the model definition that is used during symbolic processing. As a result, in Listing 5, the code generated for the pendulum of Listing 3 is shown.

Listing 5. Generated code for pendulum of Listing 3.

```
function getDerivatives(_der_x, _x, _m, _time)
    <...>
    var"pendulum.rev.phi" = _x[1]
    var"pendulum.rev.w"   = _x[2]
    _mbs1 = Modia3D.openModel3D!(<...>) # equation "injected" from Modia3D
    _mbs2 = Modia3D.setStatesRevolute!( # equation "injected" from Modia3D
                _mbs1, var"pendulum.rev.phi", var"pendulum.rev.w")
    <...>
end
```

As can be seen, the states `_x` of the solver are copied into the Modia variables `phi` and `w` of the revolute joint which are in turn passed to function `Modia3D.setStatesRevolute!(..)` of the Modia3D built-in component.

The joints of the second kind define variant variables, including variant states, according to Figure 5, which are visible only in the built-in component Modia3D. These variables can be added or removed during simulation. For example, an Object3D has an optional keyword `fixedToParent` with a default value of `true`. In this case, the Object3D is rigidly connected to its parent Object3D; this means it has zero degrees of freedom. If the value is set to `false`, the Object3D is allowed to move freely with respect to its parent, meaning it has 6 degrees of freedom and 12 variant states. At events, keyword `fixedToParent` can be changed from `false` to `true` and vice versa, as will be shown below.

5.2. Super-Objects

Rigidly connected Object3Ds are grouped together into so-called super-objects [33]. An example is given in Figure 9. Super-objects are disjunct via joints. Based on the features of Object3Ds in super-objects, different actions are performed: For example, all Object3Ds in the same super-object cannot collide with each other, but they can collide with all other Object3Ds that are enabled for collision handling. A common mass, common inertia tensor, and common center of mass are computed for a super-object taking into account the mass properties of all Object3Ds inside this super-object. For further information, see [33].

An Object3D can be marked to be the root of an assembly, or it can be marked to be lockable. As an example, Figure 9 shows 14 Object3Ds. In segment 1, they are grouped into six super-objects at initialization. If defined in an action program, two Object3Ds (e.g., obj2 and obj10, which are both defined as lockable Object3Ds) are locked, and a full restart is triggered, resulting in a new segment 2, as shown in Section 5.3, if the two Object3Ds are close to each other. During re-instantiation of segment 2, see Figure 7, the internal data structure of the Modia3D built-in component is regenerated, resulting in five super-objects in Figure 9. This is a very cheap operation in the milli-seconds range. More details are given in Section 5.3.

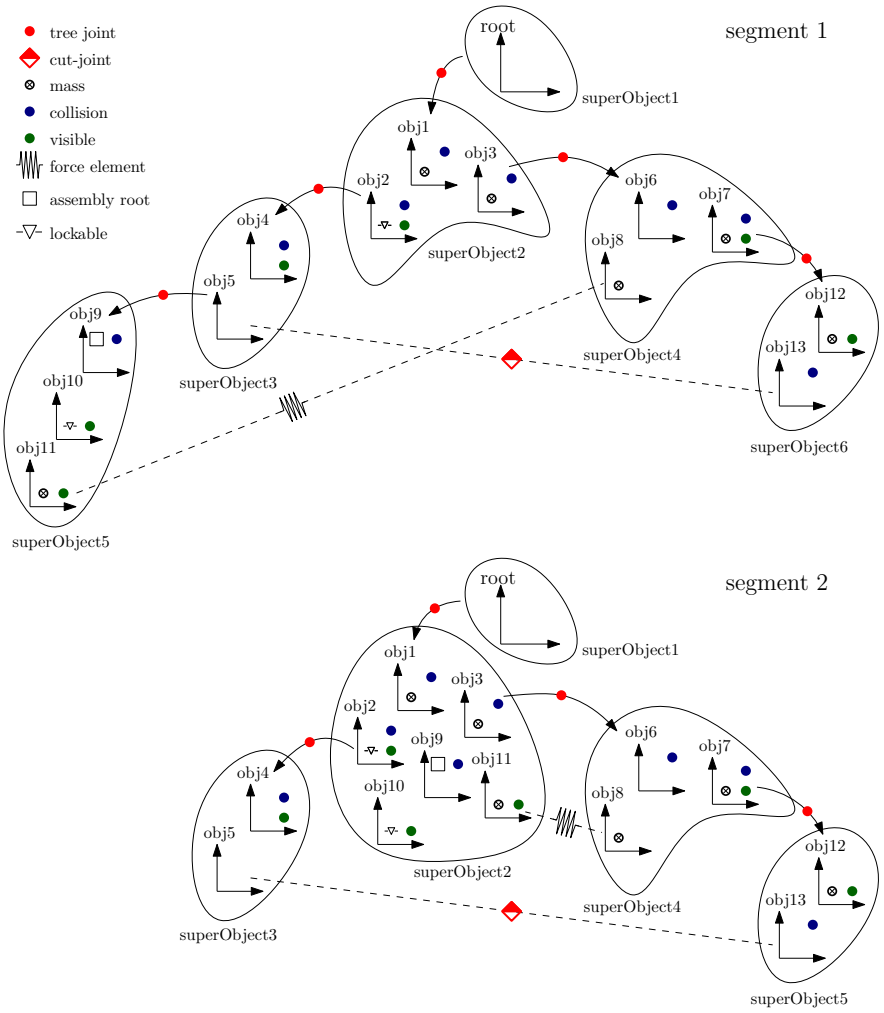


Figure 9. Internal execution scheme at initialization (**segment 1**) and after re-initialization (**segment 2**). (**Segment 1**): Fourteen Object3Ds with different properties are defined: They are allowed to collide, can have a mass, are visible and/or can be connected by force elements. They are grouped into six super-objects that are disjunct via tree- and cut-joints; see [33]. SuperObject5 is an assembly with a lockable Object3D (obj10). Only the assembly root (obj9) is allowed to change joints and states during simulation. SuperObject2 is able to interact with this assembly via its rigidly attached locking mechanism (obj2). (**Segment 2**): A full restart is triggered with `ActionAttach(..., obj10, obj2)` to initialize the second segment if both locking mechanisms (obj10 and obj2) are close to each other with negligible relative velocity. The lockable obj10 identifies its assembly root, which is obj9. The joint and states of the assembly root obj9 are removed, and all Object3Ds of the assembly are attached to superObject2. This results in five rigidly attached super objects after re-instantiation.

5.3. Segmented Simulation of Modia3D Models

In this subsection, a brief overview is given how Modia3D supports the generic segmented simulation method of Figure 7. At initialization, the Modia3Ds execution scheme is built up based on the Modia3Ds model definitions; see Figure 9. All information about the multibody systems' components (e.g., Object3Ds, joints, and solids; for an example, see Listing 3) and their functionality (e.g., collision properties) is sorted and mapped to an internal data structure with super-objects that can be efficiently evaluated during simulation. This execution scheme includes the definitions of the states of the multibody systems and of its initial values that are deduced from the utilized joints. The execution scheme is executed during the simulation of the current segment, until one of the defined actions requests a full restart for a structural change at the time of an event or when the simulation is terminated. If a full restart is required, the execution scheme is restructured,

as shown in the example of Figure 9 (basically, this means that some internal data structures are changed).

Rigidly connected Object3Ds can form an assembly by setting `assemblyRoot = true` for the freely moving Object3D, i.e., `fixedToParent = false`. All rigidly connected children of such an Object3D belong to the assembly. Additionally, any Object3D, whether it is part of an assembly or not, can be a locking mechanism by setting `lockable = true` in the Object3D constructor. In Figure 9, segment 1, superObject5 is an assembly because obj9 is marked as an assembly root. This assembly is able to interact with superObject2 because obj10 of the assembly and obj9 of superObject2 have `lockable = true` defined.

Actions on a Modia3D model and especially on assemblies are executed according to the construction sketched in Listing 6; for an application, see Listing 7. A collection of action commands is defined in a Julia function (e.g., `modelProgram`). This function is passed as `actions` argument to the Modia3D constructor `ModelActions` that returns a reference (e.g., `currentAction`) to an internal data structure. This data structure is passed to `executeActions`, which is called in a Modia `equations` section.

Listing 6. Defining actions for a Modia3D built-in model with commands of Tables 2 and 3 to interact with Modia.

```
function modelProgram(actions)
    <...>  # action commands
end

myModel = Model3D(
    world = Object3D(feature=Scene()),
    <...>
    modelActions  = ModelActions(world=:world, actions=modelProgram),
    currentAction = Var(hideResult=true),
    equations=:[
        currentAction = executeActions(modelActions)],)
```

The action commands in Table 2 increase or decrease the number of degrees of freedom, and therefore trigger a full restart of a new segment. If the number of degrees of freedom increases, new states are defined, and their initial values are computed based on the last configuration. Two actions (`ActionAttach`, `ActionReleaseAndAttach`) are only possible if the referenced lockable Object3Ds are close together and the relative velocity and angular velocity are close to zero. Currently, the following cases are treated:

- A freely moving assembly is rigidly connected to an Object3D with `ActionAttach`. This action reduces the number of degrees of freedom by six.
- If an assembly has at least two lockable Object3Ds (objA, objB) and is rigidly connected via objA, this rigid connection is removed, and another rigid connection via objB is introduced with `ActionReleaseAndAttach`. This action does not affect the number of degrees of freedom but changes the structure of the super-objects.
- A rigidly connected assembly, i.e., rigid connection to an Object3D, is unlocked with `ActionRelease` to get a freely moving assembly. This action increases the number of degrees of freedom by six.
- An assembly that is either freely moving or is rigidly connected to an Object3D is deleted with `ActionDelete`. All Object3Ds of this assembly are removed from the Modia3D model.

Whenever one of these actions is executed, the internal data structure with its super-objects must be restructured because the relations and connections between parents and their children have changed. As a result of this restructuring, objects may no longer be able to collide with each other, or the common mass properties of super-objects may have changed.

Table 2. Modia3D actions that trigger a full restart for a structural change. For applications, see Section 6.

Function	Description
`ActionAttach(...)`	Rigidly attaches the specified assembly.
`ActionReleaseAndAttach(...)`	Changes one rigid connection to another rigid connection.
`ActionRelease(...)`	Releases the specified assembly.
`ActionDelete(...)`	Deletes the specified assembly.

To initialize the next segment, a full restart is triggered in Figure 9 (segment 1) with `ActionAttach(..., obj10, obj2)`, see Table 2, provided both lockable Object3Ds (obj2 and obj10) are close to each other. Hereby, the joint connecting obj9 with obj5 is removed and obj10 is rigidly connected to obj2. The re-instantiation reduces the number of super objects and states and results in the execution scheme of Figure 9 (segment 2).

Other Modia3D actions that can be utilized but do not result in a full restart are listed in Table 3.

Table 3. Modia3D actions. For applications, see Section 6.

Function	Description
`EventAfterPeriod(...)`	Triggers an event after a specific period of time.
`ActionWait(...)`	Waits a specific period of time.
`addReferencePath(...)`	Adds a new reference path.
`ptpJointSpace(...)`	Generates a point-to-point trajectory.

6. Applications

Two applications with segmented simulations with the built-in component Modia3D are presented in this section. The first one deals with a two-stage rocket where the two stages are separated after a while. The second application is a robot where gripping and releasing actions are modelled with dynamically changing degrees of freedom.

6.1. Two-Stage Rocket

In Appendix B.2, an extremely simplified model of a two-stage rocket with mass points is shown, implemented as a Modia built-in component. The same model was implemented as a Modia3D model in Listing 7. The full code is available from (https://github.com/ModiaSim/Modia3D.jl, accessed on 17 January 2023, release v0.12.0, TwoStageRocket3D.jl). In Listing 7, one stage is modelled with submodel `RocketStage` consisting of a cylinder with lockable Object3Ds at the top and at the bottom and a thrust at the bottom. Model `TwoStageRocket` builds up the rocket system with the `world` object, two instances of `RocketStage`, and a `ModelActions` constructor that defines the actions with function `rocketProgram`:

1. Initially, the two stages are not rigidly connected. At initialization, the top of stage1 and the bottom of stage2 are attached with `ActionAttach(actions, "stage1.top", "stage2.bottom")`; see the visualization in the left part of Figure 10.
2. An event is triggered after 5 s with `EventAfterPeriod(actions, 5)` to release `state1.top` from `state2.bottom` with `ActionRelease(actions, "stage1.top")`. This separates the two stages. Furthermore, thrust1 is switched off.
3. Again, an event is triggered after 5 s with `EventAfterPeriod(actions, 5)`; see the right part of Figure 10. Since the movement of stage1 is no longer of interest in this scenario, `state1.top` and all Object3Ds connected to it are deleted with `ActionDelete(actions, "stage1.top")`. Thus, stage1 is removed from the simulation run.

Plots of the relevant variables are shown in Figure 11.

Listing 7. Modia3D model of a simple two stage rocket. The stages are instances of sub-model RocketStage that is a cylinder of length `L`, diameter `d`, mass `m`, with lockable Object3Ds at the top and at the bottom, and a thrust at the bottom.

```
function rocketProgram(actions)
  ActionAttach(actions, "stage1.top", "stage2.bottom")
  EventAfterPeriod(actions, 5)          # Trigger an event after 5s
  ActionRelease(actions, "stage1.top")  # Release stage1.top
  EventAfterPeriod(actions, 5)          # Trigger an event after 5s
  ActionDelete(actions, "stage1.top")   # Delete stage1
end

RocketStage(; L=1.0, d=0.1, m=100.0, color="blue", r_init,
              thrustFunction) = Model( # rocket stage is a cylinder
  body = Object3D(parent=:world, fixedToParent=false,
            translation=r_init, assemblyRoot=true,
            feature=Solid(...),
  # Lockable Object3Ds at cylinder bottom and top
  bottom = Object3D(parent=:body, translation=:[0.0, -$L/2, 0.0],
             lockable=true),
  top    = Object3D(parent=:body, translation=:[0.0,  $L/2, 0.0],
             lockable=true),
  # thrust is applied at bottom
  thrust = WorldForce(objectApply=:bottom, forceFunction=thrustFunction)
)

TwoStageRocket = Model3D(
    world  = Object3D(feature=Scene()),
    stage1 = RocketStage(L=2.0, d=0.2, color="blue", r_init=[0,1,0],
               thrustFunction=thrust1),
    stage2 = RocketStage(L=1.0, d=0.15, color="red", r_init=[0,2.5,0],
               thrustFunction=thrust2),

    modelActions = ModelActions(world=:world, actions=rocketProgram),
    currentAction = Var(hideResult=true),
    equations=:[
        currentAction = executeActions(modelActions)
        ],
)

rocket = @instantiateModel(TwoStageRocket)
simulate!(rocket, stopTime=15u"s")
```

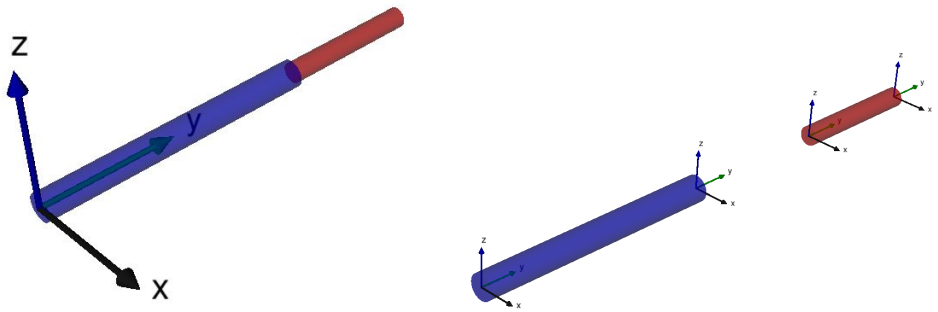


Figure 10. A two-stage rocket (stage 1: blue cylinder, stage 2: red cylinder). (**Left**): At initialization. (**Right**): After separating (release). The locking mechanisms at the top and bottom are each visualized with a coordinate system.

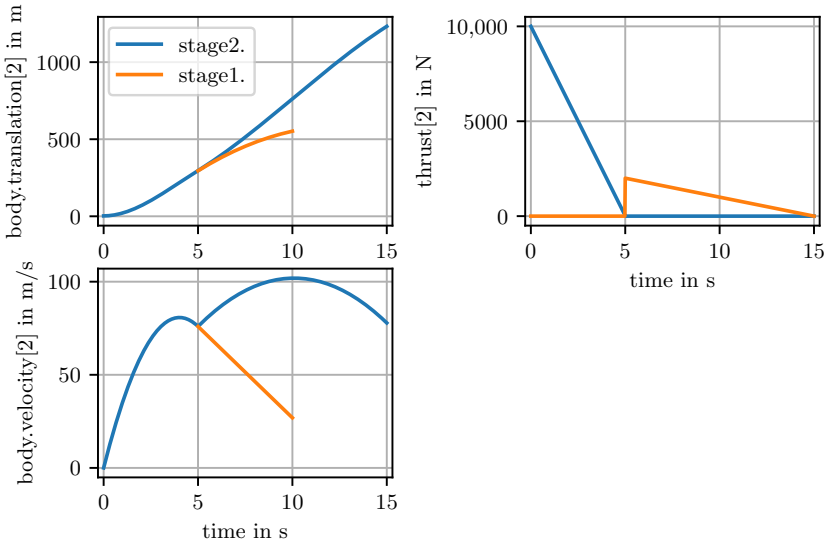


Figure 11. Plots of relevant variables from the simulation of model TwoStageRocket of Listing 7. Variable values are not shown in the plots, if they are not defined in the respective phase.

6.2. Gripping Robot

The new support of segmented simulations in Modia3D allows one to carry out gripping operations without elastic contact handling. For example, an assembly A is locked at object B (meaning, A is lying on B) and is then gripped from an object C, and then A is released from B and is locked (so rigidly fixed) to C. The advantage of this procedure is that collision handling issues can be avoided. For example, simulations can be performed with larger tolerances and fine-tuning of the details of the gripping operations, and the transport of the gripped object is no longer required. Thus, the simulation is faster and more robust, and the setup of the scenario is easier (see Scenario 2). It is also possible to model gripping operations that are not based on frictional contacts but use rigid mechanical connections, such as a bayonet lock. The drawback is that the details of the gripping are not modelled, but this could be essential, e.g., when designing a control system that carries out an assembly task.

This approach is demonstrated with a gripping operation of a KUKA YouBot robot. This robot has a five degrees of freedom arm and was manufactured in the years 2010–2016. The robot was modelled with Modia (drive trains and controllers) and with Modia3D (3D mechanics); see [4].

Scenario 1: Adding and removing states during simulation. Therefore, a slightly different version of model [4] was created for this article; see (https://github.com/ModiaSim/Modia3D.jl, accessed on 17 January 2023, release v0.12.0, YouBotDynamicState.jl) using the approach sketched above. Listing 8 shows parts of the action commands, and Figure 12 (left) shows a screenshot of the animation. The transportation procedure started with a free sphere lying on the plate of the robot (six DoF) that became rigidly attached (zero DoF) for transportation by the robot's gripper, until it was released and fell freely downwards (six DoF), bouncing on the plate. This application combined the benefits of segmented simulation for the gripping and transportation of the sphere (no collision handling) with the collision response of the freely bouncing sphere.

Listing 8. Action definitions for the KUKA YouBot robot. The transportation procedure starts with initializing a new reference path for driving 5 arm angles and the gripper itself, with unique names, initial positions, and maximum angular velocity v_{max}, and maximum angular acceleration a_{max} of the revolute joints. The arms and gripper are driven to the given points to rigidly attach the resting sphere with the gripper. The robot transports the rigidly attached sphere until it is released, falls freely and bounces on the plate.

```
function robotProgram(robotActions)
    addReferencePath(robotActions,
        names = ["angle1", "angle2", "angle3", "angle4", "angle5", "gripper
            "],
        v_max = [2.68512, 2.68512, 4.8879, 5.8997, 5.8997, 2.0],
        a_max = [1.5, 1.5, 1.5, 1.5, 1.5, 0.5],
        position = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
    ptpJointSpace(robotActions, [pi pi/4 pi/4 1.057 0.0 diameter+0.01])
    ActionAttach(robotActions, "sphereLock", "youbot.gripper.gripperLock")
    ptpJointSpace(robotActions, [pi 0.0 pi/2 0.0 0.0 diameter-0.002])
    ActionWait(robotActions, 0.2)
    ActionRelease(robotActions, "sphereLock")
    ...
end
```

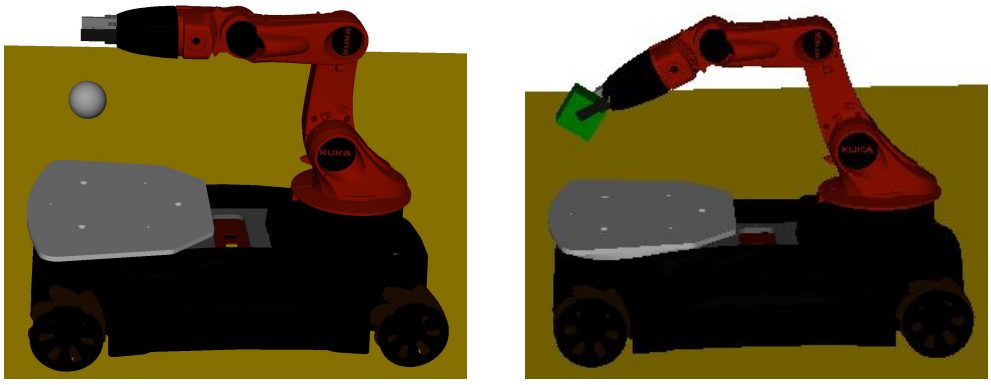


Figure 12. (**Left**) Scenario 1: A YouBot robot after releasing a free falling sphere (six DoF). The sphere will bounce on the grey plate. (**Right**) Scenario 3: A YouBot transporting a box with segmented simulation.

Scenario 2: Transportation of a sphere by segmented simulation compared to collision handling. Thus, a sphere was gripped by a robot and transported, until it was released and placed on a plate. Finally, it was gripped and transported again. This procedure was modelled (a) with a segmented simulation (https://github.com/ModiaSim/Modia3D.jl, accessed on 17 January 2023, release v0.12.0, YouBotFixSphere.jl) and (b) with collision handling (https://github.com/ModiaSim/Modia3D.jl, accessed on 17 January 2023, release v0.12.0, YouBotSphereTransport.jl). The simulation of (a) took about 0.22 s, whereas the simulation of (b) took about 6.67 s on a standard notebook. Therefore, the simulation time of (a) was about 30 times faster than (b). The reason is that (a) was basically a non-stiff system where the solver could use large step-sizes, and the time for the reconfiguration of the multibody system (for gripping and releasing) was negligible, whereas (b) was a stiff system, since the gripper held the sphere via elastic contact and friction forces that varied during the transportation, and therefore, the solver had to use much smaller step-sizes.

Scenario 3: Transportation of a box by segmented simulation. The models of scenario 3 were similar to those of scenario 2, but only the sphere was replaced by a box; see Figure 12, right. The simulation time of model (a) with segmented simulation was approximately the same as that of scenario 2. It is not possible to simulate model (b) with a box, because Modia3D collision handling currently only supports point contacts with elastic contact laws due to the used MPR algorithm. For parallel or nearly parallel surfaces, such as a box and gripper, or box and plate, no unique point contact can be computed that is continuous over time, as required from an adaptive step-size control.

7. Conclusions

This article introduced a new method to extend declarative, equation-based modelling systems so that variables can appear and disappear during simulation without re-generation and re-compilation of code when the numbers of equations and states change during events. This was introduced in a generic, mathematical way and was demonstrated with an extended version of Modia and of Modia3D. The main advantages of this approach are (a) that the modeller does not need to define in advance how the system changes the equations because this is decided at run-time and (b) that the transformation to a new mode is typically very efficient, because no code needs to be newly generated and compiled on the fly. This is, in particular, the case for the multibody built-in component Modia3D, which allows a superfast re-arrangement of the execution scheme for a new mode and opens up new applications, because gripping operations are much more efficient and more robust as sketched with the scenarios of the previous section.

As a side-effect, it is now possible to implement acausal components in Modia that consist to a large extent of pre-translated functions that can, for example, hide space discretization schemes of PDEs (partial differential equations) from equation-based code and its transformation algorithms, so that, for example, the number of discretization elements can be changed after code generation and even during simulation. This was demonstrated with heat transfer in a rod and can be generalized for many more PDE-based components, especially for thermo-fluid systems. It is planned to support much more models of this kind in Modia. Another benefit is that the transformation of a model into ODE form and generation and compilation of code is much faster with such built-in components, especially for large models with many states, because the core part of the equations is compiled once, independently of how many instances of the component are used in a model.

A drawback of this method is that the manual implementation of built-in components is quite involved, when compared to the simple definition of pure equation-based components. In the future, it might be possible to automatically transform a higher-level component description to a built-in component. Furthermore, built-in components cannot be used in an arbitrary way. For example, it is usually not possible to invert a model that contains built-in components, even if this would work for the underlying, pure equation-based version of the model. In the future, it would be also useful to combine the technique of built-in components with on-the-fly translation during simulation.

Author Contributions: Conceptualization, methodology, software, validation, writing—original draft preparation, review and editing, visualization: A.N., M.O., project administration: M.O.. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The Julia packages Modia (https://github.com/ModiaSim/Modia.jl, accessed on 17 January 2023, release v0.10.0) and Modia3D (https://github.com/ModiaSim/Modia3D.jl, accessed on 17 January 2023, release v0.12.0) are publicly available under the MIT open source license.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

3D	three dimensional
AST	abstract syntax tree
DAE	differential algebraic equation
DoF	degrees of freedom
FR	full restart
MDPI	Multidisciplinary Digital Publishing Institute

MPR	Minkowski portal refinement
ODE	ordinary differential equation
PDE	partial differential equation
inv	invariant
var	variant

Appendix A. Overview of Modia

This appendix section provides a concise introduction to Modia as needed to understand the code fragments used in this article. It is a shortened and modified version of [4] Section 2. Much more details are given in the Modia tutorial (https://modiasim.github.io/Modia.jl/stable/, accessed on 11 December 2022).

Modia is a Julia package and provides a modelling and simulation environment for declarative, equation-based models. Contrary to other modelling languages, a Modia model is defined directly with the Julia language using some pre-defined helper functions to define the model AST (Abstract Syntax Tree), so there is no language that is parsed and transformed to Julia code. This is possible and is reasonably user-friendly due to the feature-rich Julia language.

A Modia model is defined with hierarchical collections of name/value pairs, together with merging of such collections. This unified scheme is used for models, variables, equations, hierarchical modifiers, inheritance, and replaceable components.

Appendix A.1. Variables and Models

Variables are implicitly defined by their references in equations. A constructor `Var` allows one to define variables with attributes.

```
name = Var(attribute=value, ...)
```

`Var` is a function taking name/value pairs, building and returning a corresponding dictionary. The currently introduced attributes are: `value`; `min`; `max`; `init`; `start`; and the Booleans: `parameter`, `constant`, `input`, `output`, `potential`, and `flow`. `Par` is an abbreviation for `Var(parameter=true)`. Example:

```
T1 = Var(parameter=true, value=0.2, min=0)
T2 = Par(value=0.2, min=0)    # definitions are equivalent for T1 and T2
```

If the value contains references to other declared variables in the model, the expressions must be enclosed in quotes `:( )`. A parameter can also be defined by `name = literal-value`.

A model (Listing A1) is defined as a collection of name/value pairs with the constructor `Model`.

Listing A1. Syntax of a Modia model.

```
name = Model(
  <variable-or-component-definition>,
  ...,
  equations = :[
    <equation1>
    <equation2>
    ...]
  )
```

The equations have Julia expressions on both left and right sides of the equals sign. Note that the entire array of `equations` is quoted, since it is enclosed in `:[ ]`. This enables later processing, such as symbolically solving the equation, since an AST is built-up, instead of evaluating the expressions. The Modia-specific operator `der(v)` defines the time derivative dv/dt of variable v. Definition of units is done with a string macro `u"..."` from Julia package Unitful.jl [34]. For example, a low pass filter can be defined as in Listing A2.

Listing A2. Modia model of a low pass filter.

```
LowPassFilter = Model(
  T = 0.2u"s",
  u = Var(input=true),
  y = Var(output=true),
  x = Var(init=0.0),
  equations = :[
    T * der(x) + x = u
    y = x]
  )
```

This corresponds to the Modelica model in Listing A3.

Listing A3. Modelica model of a low pass filter.

```
block LowPassFilter
  parameter SIunits.Time T = 0.2;
  input  Real u;
  output Real y;
  Real x(start=0.0, fixed=true);
equation
  T * der(x) + x = u;
  y = x;
end LowPassFilter;
```

Appendix A.2. Connectors and Components

Models which contain any flow variable (with attribute `flow = true`) are considered connectors. Connectors must have an equal number of flow and potential variables (with attribute `potential = true`) and have matching array sizes. Connectors may not have any equations. An electrical connector with potential `v` and current `i` is defined as:

```
Pin = Model(v = Var(potential=true), i = Var(flow=true))
```

Components are declared by using a model name as a value in a name/value pair. An electrical capacitor with two Pins `p` and `n` can be described as in Listing A4.

Listing A4. Capacitor model.

```
Capacitor = Model(
  C = 0.1u"F",
  p = Pin,
  n = Pin,
  equations = :[
    0 = p.i + n.i
    v = p.v - n.v
    C*der(v) = p.i ]
  )
```

Appendix A.3. Merging

Models and variables are defined with hierarchical collections of name/value pairs. Setting and modifying parameters of components and attributes of variables are also naturally performed in the same way. A constructor `Map` is used for that. For example, modifying the parameter `T` of the `LowPassFilter` model defined in Listing A2 can be implemented by:

```
lowPassFilter = LowPassFilter | Map(T = Map(value=2u"s", min=1u"s"))
```

The achieved semantics is the same as for hierarchical modifiers in Modelica and result in:

```
lowPassFilter = Model(T = Par(value=2u"s", min=1u"s"), ...)
```

The used merge operator `|` is an overloaded binary operator of bitwise *or* with recursive merge semantics. Merging of `equations` is handled in a special way by concatenating the equations vectors.

Appendix A.4. Connections

Connections are described as a special equation of the form:

```
connect( <connect-reference-1>, <connect-reference-2>, ... )
```

A "connect-reference" has either the form "connect instance name" or "component instance name." "connect instance name" is either a connector instance, input, or output variable. For connectors, all the corresponding potentials of the connected connectors are set as equal. The sum of all incoming corresponding flows to the model is set equal to the sum of the corresponding flows into sub-components, i.e., the same semantics as in Modelica.

Having electrical component models enables defining a filter (Listing A5) by instantiating components, setting parameters and defining connections. The filter model was instantiated and simulated, and the results are plotted:

Listing A5. Filter model defined with electrical components is instantiated, simulated and plotted.

```
using Modia
include("$(Modia.path)/models/Electric.jl") # include electrical components

Filter = Model(
  R  = Resistor,
  C  = Capacitor | Map(C=2.0u"F"),
  V  = ConstantVoltage | Map(V=10.0u"V"),
  equations = :[
    connect(R.n , C.p)
    connect(C.n , V.n) ]
  )

filter = @instantiateModel(Filter)
simulate!(filter, stopTime=2.0u"s", merge=Map(C=Map(C=100u"F")))

@usingModiaPlot    # Use selected plot package
plot(filter, ("R.v", "C.v"))
```

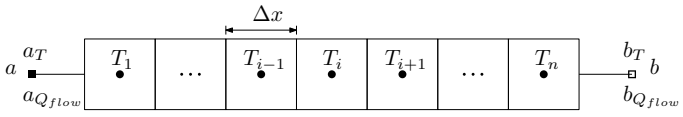
Julia macro `@instantiateModel` symbolically processes the model, generates, and compiles Julia code. Symbolic processing is performed with standard algorithms of object-oriented modelling languages and with extensions described in [35]. Function `simulate!` performs one simulation of the instantiated model with a solver from the Julia package DifferentialEquations.jl [36,37]. This package contains a large set of solvers. In Listing A5, the Modia default solver `CVODE_BDF` is used. With various keyword arguments, the simulation run can be defined, e.g., the stop time is set to 2 s. Parameters and initial values can be provided by a hierarchical `Map` that is merged with the current values via the `merge` keyword. The simulation results are stored within the instantiated model and are plotted with function call `plot`.

Appendix B. Examples

Appendix B.1. Heated Rod with Acausal Built-In Component

In Figure A1, an equation-based model of heat transfer in a rod with isolated surface is shown.

On the left and right sides of the rod, thermal connectors a, b are present that have potential variables a_T, b_T (temperatures) and flow variables $a_{Q_{\text{flow}}}, b_{Q_{\text{flow}}}$ (heat flow rates). The partial differential equation that mathematically describes heat transfer in one dimension is discretized in space by volumes $V_i = \Delta x \cdot A$ of equal lengths Δx and identical areas A. In the middle of volume i, a temperature T_i is defined. All temperatures are collected in vector $\boldsymbol{T} = [T_1, T_2, \ldots, T_n]$. In Listing A6, a Modia model is shown that uses a *built-in component* `InsulatedRod` of the rod that is connected at its left thermal connector `a` with a fixed temperature source `FixedTemperature` and at its right thermal connector `b` with a fixed heat-flow source `fixedHeatFlow` with the default zero heat-flow rate (so the rod is totally insulated on the right side and has a fixed temperature on the left side).



$$\Delta x = L/n$$

$$Q_{\text{flow},i} = \lambda \cdot A/\Delta x \cdot \begin{cases} 2(a_T - T_1) & i = 0 \\ T_i - T_{i+1} & i = 1,\ldots,n-1; \\ 2(T_i - b_T) & i = n \end{cases} \qquad \begin{aligned} a_{Q_{\text{flow}}} &= Q_{\text{flow},0} \\ b_{Q_{\text{flow}}} &= -Q_{\text{flow},n} \end{aligned}$$

$$\varrho \cdot c \cdot A \cdot \Delta x \cdot \dot{T}_i = Q_{\text{flow},i-1} - Q_{\text{flow},i}, \quad i = 1,\ldots,n; \quad T_i(t = t_0) = T_0$$

Figure A1. Space discretized partial differential equation of one-dimensional heat transfer in a rod with an isolated surface, defined with parameters L (length of rod), n (number of volumes), A (area), ϱ (density), c (specific heat capacity), λ (thermal conductivity), T_0 (initial value in every volume), states T_i (temperatures in the middle of every volume), thermal connectors a, b with potential variables a_T, b_T (temperatures), and flow variables $a_{Q_{\text{flow}}}, b_{Q_{\text{flow}}}$ (heat flow rates).

Listing A6. Simple usage of isolated rod `InsulatedRod` with one-dimensional heat-transfer that is connected on the left side with a fixed temperature source `FixedHeatFlow` with $T = 220\,^{\circ}\text{C} = 493.15$ K, and on the right side with a fixed heat flow source `FixedHeatFlow` with $Q_{\text{flow}} = 0$.

```
HeatPort = Model(T=Var(potential=true), Q_flow=Var(flow=true))
HeatedRod = Model(
    fixedT     = FixedTemperature | Map(T=493.15),    # temperature source
    fixedQflow = FixedHeatFlow,   # heat flow source with default Q_flow=0
    rod        = InsulatedRod | Map(L=1.0, T0=273.15, n=5),    # 5 volumes
    equations  = :[connect(fixedT.port, rod.a),
                   connect(rod.b      , fixedQflow.port)])
heatedRod = @instantiateModel(HeatedRod)
simulate!(heatedRod, stopTime = 1e5, merge=Map(rod = Map(n=8)) # 8 volumes
plot(heatedRod, ("fixedT.port.T", "rod.T"))
```

Command `@instantiateModel(HeatedRod)` symbolically processes this model, and generates and translates Julia code. The `simulate!(..)` statement changes the discretization (and therefore, the dimension of the temperature vector `T`) from 5 to 8 volumes before simulation starts (and without a new translation). With function `plot(heatedRod, ...)`, the plot of Figure A2 is generated, showing the temperatures at the temperature source and in the rod volumes. Two different implementations of the `InsulatedRod` model are shown in Table A1 in the form of pseudo-code. In the left column, the model's implementation with mathematical functions is shown. In the right column, the model's implementation is shown as a built-in component using functions of a programming language that have internal memory.

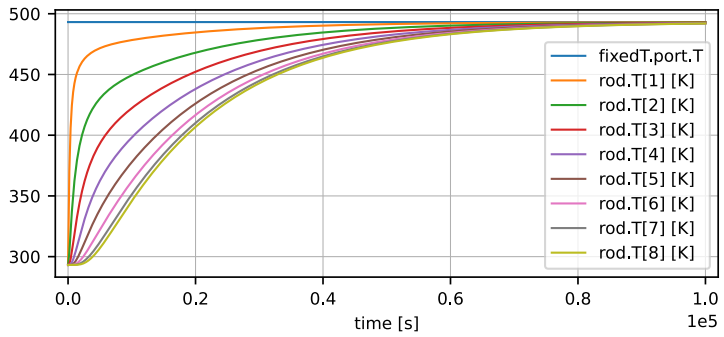


Figure A2. Plot of temperatures of heated rod model of Listing A6.

Table A1. Built-in component of an insulated rod with parameters $L, n, A, \varrho, \lambda, c, T_0$, state vector $\boldsymbol{T}$ of length n (temperatures at rod volumes), connectors a, b on the left and right sides of the rod with potential variables a_T, b_T (temperatures), and flow variables $a_{Q_{\text{flow}}}, b_{Q_{\text{flow}}}$ (heat flow rates).

as component (with mathematical functions)	**as built-in component** (with prog. lang. functions that have internal memory m)
evaluated once	
$dx = L/n$ $k_1 = (\lambda/dx)/(c \cdot \varrho \cdot dx)$ $k_2 = 2 \cdot \lambda \cdot A/dx$	$m = f_{c,0}(\text{sim}, \text{id}, L, n, A, \varrho, \lambda, c, T_0)$
equation section	
$(T_1, T_n) = f_{c,1}(\boldsymbol{T})$ $a_{Q_{\text{flow}}} = k_2 \cdot (a_T - T_1)$ $b_{Q_{\text{flow}}} = k_2 \cdot (b_T - T_n)$ $\dot{\boldsymbol{T}} = f_{c,2}(\boldsymbol{T}, k_1, a_T, b_T)$	$(k_2, T_1, T_n) = f_{c,1}(m)$ $a_{Q_{\text{flow}}} = k_2 \cdot (a_T - T_1)$ $b_{Q_{\text{flow}}} = k_2 \cdot (b_T - T_n)$ $f_{c,2}(m, a_T, b_T)$ # no return arguments
function definitions	
	function $f_{c,0}(\text{sim}, \text{id}, L, n, A, \varrho, \lambda, c, T_0)$ $<$ allocate new record $m >$; assert$(n \geq 2)$ $m_{\text{sim}} := \text{sim}$; $m_{\text{id}} := \text{id}$; $m_n := n$; $\Delta x := L/n$ $m_{k_1} := (\lambda/\Delta x)/(c \cdot \varrho \cdot \Delta x)$ $m_{k_2} := 2 \cdot \lambda \cdot A/\Delta x$; **return** m
function $f_{c,1}(\boldsymbol{T})$ **return** (T_1, T_n)	**function** $f_{c,1}(m)$ $<$ copy $m_{\boldsymbol{T}}$ from states in m_{sim} for $m_{\text{id}} >$ **return** $(m_{k_2}, m_{T_1}, m_{T_n})$
function $f_{c,2}(\boldsymbol{T}, k_1, a_T, b_T)$ $\dot{T}_1 := k_1 \cdot (2 \cdot (a_T - T_1) - (T_1 - T_2))$ $\dot{T}_n := k_1 \cdot (T_{n-1} - T_n - 2 \cdot (T_n - b_T))$ **for** $i = 2 : n-1$ $\dot{T}_i := k_1 \cdot (T_{i+1} - T_i - (T_i - T_{i-1}))$ **return** $\dot{\boldsymbol{T}}$	**function** $f_{c,2}(m, a_T, b_T)$ $\boldsymbol{T} := m_{\boldsymbol{T}}$; $n := m_n$; $k_1 = m_{k_1}$ $m_{\dot{T}_1} := k_1 \cdot (2 \cdot (a_T - T_1) - (T_1 - T_2))$ $m_{\dot{T}_n} := k_1 \cdot (T_{n-1} - T_n - 2 \cdot (T_n - b_T))$ **for** $i = 2 : n-1$ $m_{\dot{T}_i} := k_1 \cdot (T_{i+1} - T_i - (T_i - T_{i-1}))$ $<$ copy $m_{\dot{\boldsymbol{T}}}$ into state derivatives of m_{sim} for $m_{\text{id}} >$

A Modia model of the built-in component is available from (https://github.com/ModiaSim/Modia.jl, accessed on 17 January 2023, release v0.10.0, TestHeatTransfer2.jl). The advantage of the acausal built-in component implementation is its very compact definition in the equation section with four scalar equations. Once the used functions are translated (once), the symbolic processing and the code generation have to handle only four scalar equations per insulated rod, independent of the number of temperature nodes.

Note, since the left connector a of the rod is connected to a temperature source, $a_{Q_{\text{flow}}} = k_2 \cdot (a_T - T_1)$ is kept in this form in the generated code (a_T is provided by the temperature source, T_1 is provided, since it is a state, and $a_{Q_{\text{flow}}}$ is computed from this equation). Since the right connector b of the rod is connected to a heat flow source, equation $b_{Q_{\text{flow}}} = k_2 \cdot (b_T - T_n)$ is transformed into the statement $b_T := T_n$ during symbolic processing because $b_{Q_{\text{flow}}} = 0$.

Appendix B.2. Two-Stage Rocket with Acausal Built-In Component

The approach of Section 4 is demonstrated with an extremely simplified acausal built-in component of a rocket with two stages. In Section 6.1, a 3D version is shown defined with a Modia3D built-in component.

The two stages $i \in \{1, 2\}$ of a two-stage rocket are approximated by mass points m_i (constant masses are used; the effect of the variable masses due to the burned fuel is taken into account by decreasing the thrusts over time, since bodies with variable masses are not yet supported in Modia3D, which is used for the 3D version of the rocket). Other important variables of the model are: h_i height over ground i, v_i velocity, F_i thrust, g gravitational

acceleration (height dependency is neglected), and time t. An enumeration to indicate the actual modes,

$$\text{phase} = \begin{cases} 1 & \text{one body with two stages } (m_1 + m_2), \\ 2 & \text{two bodies with separated stages } (m_1, m_2), \\ 3 & \text{one body}(m_2) \text{ left, stage 1 is removed from model,} \end{cases}$$

is introduced.

Initially, in phase $= 1$ the two stages are connected together, and the system has two states h_1, v_1 and is described by the equations:

$$v_1 = \dot{h}_1 \tag{A1}$$
$$(m_1 + m_2) \cdot \dot{v}_1 = F_1 - (m_1 + m_2) \cdot g. \tag{A2}$$

At $t = t_1$, phase $= 2$, the two stages are separated, and the thrust of stage 2 is switched on. In this phase, the system has four states h_1, v_1, h_2, v_2 and is described by the equations:

$$v_1 = \dot{h}_1 \tag{A3}$$
$$m_1 \cdot \dot{v}_1 = F_1 - m_1 \cdot g \tag{A4}$$
$$v_2 = \dot{h}_2 \tag{A5}$$
$$m_2 \cdot \dot{v}_2 = F_2 - m_2 \cdot g. \tag{A6}$$

At $t = t_2$, phase $= 3$, stage 1 is removed from the simulation, because it is no longer of interest, and the system has two states h_2, v_2 and is described by the equations:

$$v_2 = \dot{h}_2 \tag{A7}$$
$$m_2 \cdot \dot{v}_2 = F_2 - m_2 \cdot g. \tag{A8}$$

A Modia model `RocketSystem` implementing these equations is given in Listing A7 and is available from (https://github.com/ModiaSim/Modia.jl, accessed on 17 January 2023, release v0.10.0, TestTwoStageRocket.jl). The utilized built-in component `TwoStageRocket` has varying number of states and its pseude-code is shown in Table A2. Simulation results are shown in Figure A3.

Listing A7. Modia model of simple two stage rocket. `TwoStageRocket` is a built-in component with varying number of states.

```
using Modia

RocketSystem = Model(
  rocket = TwoStageRocket(m1    = 100.0, m2    = 100.0,
                          F1Max =   1e4, F2Max = 0.2e4, g = 9.81,
                           t1   =   5.0, t2    = 10.0, t3 = 15.0),
)
rocketSystem = @instantiateModel(RocketSystem)
simulate!(rocketSystem, stopTime=15.0)
```

The Modia built-in component with pseudo-code implementation for this component is shown in Table A2. Note that function $f_{c,\text{init}}$ defined in Table A2 is called before a new segment is initialized, also including segment 1. This function defines the variables of the respective segment and initializes them. Function $f_{c,2}$ is called for every model evaluation. In this function, the current states of the rocket instance are copied from the state vector of the solver to the rocket instance. The state derivatives are computed based on the phase of the rocket and are then copied into the derivative of the state vector of the solver.

Table A2. Pseudo-code for variable structure state handling of built-in component TwoStageRocket.

# equation section (events are triggered at $t == t_1$ and $t == t_2$)
phase = **if** $t < t_1$ **then** 1 **elseif** $t < t_2$ **then** 2 **else** 3 **end**
$f_{c,2}(m, \text{phase})$

---

function definitions
function $f_{c,0}(\text{sim}, \text{id}, m_1, m_2, g, f_{1,\max}, f_{2,\max}, t_1, t_2, t_3)$ # called once
  < allocate new record m and store parameters in m >
  $m_{\text{phase}} := 0$; $m_{\text{nextPhase}} := 1$; **return** m
function $f_{c,\text{init}}(m)$ # called before a segment is initialized
  $m_{\text{phase}} := m_{\text{nextPhase}}$
  if $m_{\text{phase}} == 1$
    < define new variables $h_1(\text{init} = 0), v_1(\text{init} = 0), \dot{h}_1, \dot{v}_1, F_1$ >
  elseif $m_{\text{phase}} == 2$
    < define new variables $h_1(\text{init} = m_{h_1}), h_2(\text{init} = m_{h_1})$,
      $v_1(\text{init} = m_{v_1}), v_2(\text{init} = m_{v_1}), \dot{v}_1, \dot{v}_2, F_1, F_2$ >
  elseif $m_{\text{phase}} == 3$
    < define new variables $h_2(\text{init} = m_{h_2}), v_2(\text{init} = m_{v_2}), \dot{h}_2, \dot{v}_2, F_2$ >
  end
function $f_{c,2}(m, \text{phase})$ # called in equation section
  if isEvent(m_{sim}) **and** $m_{\text{phase}} != \text{phase}$
    $m_{\text{nextPhase}} := \text{phase}$; setFullRestartEvent($m_{\text{sim}}$); **end**
  if $m_{\text{phase}} == 1$
    < copy m_{h_1}, m_{v_1} from states in m_{sim} for m_{id} >
    $\dot{h}_1 = m_{v_1}$
    $F_1 = F(m_{F_{1,\max}}, 0, m_{t_1})$ # Function F(..) to compute thrust is not shown
    $\dot{v}_1 = F_1/(m_1 + m_2) - g$
    < copy $\dot{h}_1, \dot{v}_1, F_1$ into states/local variables of m_{sim} for m_{id} >
  elseif $m_{\text{phase}} == 2$
    < copy $m_{h_1}, m_{v_1}, m_{h_2}, m_{v_2}$ from states in m_{sim} for m_{id} >
    $\dot{h}_1 = m_{v_1}$
    $F_1 = F(m_{F_{1,\max}}, 0, m_{t_1})$ # Function F(..) to compute thrust is not shown
    $\dot{v}_1 = F_1/m_1 - g$
    $\dot{h}_2 = m_{v_2}$
    $F_2 = F(m_{F_{2,\max}}, m_{t_1}, m_{t_3})$ # Function F(..) to compute thrust is not shown
    $\dot{v}_2 = F_2/m_2 - g$
    < copy $\dot{h}_1, \dot{v}_1, \dot{h}_2, \dot{v}_2, F_1, F_2$ into states/local variables of m_{sim} for m_{id} >
  elseif $m_{\text{phase}} == 3$
    < copy m_{h_2}, m_{v_2} from states in m_{sim} for m_{id} >
    $\dot{h}_2 = m_{v_2}$
    $F_2 = F(m_{F_{2,\max}}, m_{t_1}, m_{t_3})$ # Function F(..) to compute thrust is not shown
    $\dot{v}_2 = F_2/m_2 - g$
    < copy $\dot{h}_2, \dot{v}_2, F_2$ into states/local variables of m_{sim} for m_{id} >
  end

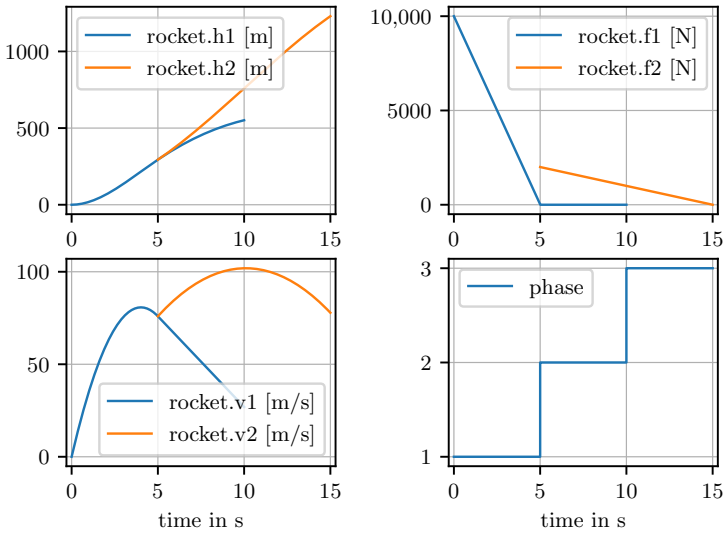


Figure A3. Plots of the variables from the simulation of model RocketSystem of Listing A7. Variable values are not shown in the plots, if they are not defined in the respective phase.

References

1. Modelica Association. Modelica—A Unified Object-Oriented Language for Systems Modeling, Language Specification, Version 3.5. 2021. Available online: https://specification.modelica.org/maint/3.5/MLS.pdf (accessed on 13 January 2023).
2. Modelica Tools. Available online: https://modelica.org/tools.html (accessed on 11 December 2022).
3. Arnold, M. DAE Aspects of Multibody System Dynamics. In *Surveys in Differential-Algebraic Equations IV*; Ilchmann, A., Reis, T., Eds.; Springer International Publishing: Cham, Switzerland, 2017; pp. 41–106. [CrossRef]
4. Elmqvist, H.; Otter, M.; Neumayr, A.; Hippmann, G. Modia—Equation Based Modeling and Domain Specific Algorithms. In Proceedings of the 14th International Modelica Conference, Linköping, Sweden, 20–24 September 2021; pp. 73–86. [CrossRef]
5. Bezanson, J.; Edelman, A.; Karpinski, S.; Shah, V.B. Julia: A fresh approach to numerical computing. *SIAM Rev.* **2017**, *59*, 65–98. [CrossRef]
6. Pantelides, C.C. The Consistent Initialization of Differential-Algebraic Systems. *SIAM J. Sci. Stat. Comput.* **1988**, *9*, 213–231. [CrossRef]
7. Pryce, J.D. A simple structural analysis method for DAEs. *BIT Numer. Math.* **2001**, *41*, 364–394. [CrossRef]
8. Elmqvist, H.; Matsson, S.E.; Otter, M. Modelica extensions for multi-mode DAE systems. In Proceedings of the 10th International Modelica Conference, Lund, Sweden, 10–12 March 2014; Linköping University Electronic Press: Linköping, Sweden, 2014; pp. 183–193. [CrossRef]
9. Benveniste, A.; Caillaud, B.; Malandain, M.; Thibault, J. Algorithms for the Structural Analysis of Multimode Modelica Models. *Electronics* **2022**, *11*, 2755. [CrossRef]
10. Caillaud, B.; Malandain, M.; Thibault, J. Implicit Structural Analysis of Multimode DAE Systems. In Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control, HSCC '20, Sydney, NSW, Australia, 22–24 April 2020. [CrossRef]
11. Benveniste, A.; Caillaud, B.; Elmqvist, H.; Ghorbal, K.; Otter, M.; Pouzet, M. Multi-Mode DAE Models—Challenges, Theory and Implementation. In *Computing and Software Science: State of the Art and Perspectives*; Springer International Publishing: Cham, Switzerland, 2019; pp. 283–310. [CrossRef]
12. Höger, C. Dynamic Structural Analysis for DAEs. In Proceedings of the 2014 Summer Simulation Multiconference, SummerSim'14, Monterey, CA, USA, 6–10 July 2014; Society for Computer Simulation International: San Diego, CA, USA, 2014.
13. Zimmer, D. Equation-Based Modeling of Variable-Structure Systems. Ph.D. Thesis, ETH Zürich, Zürich, Switzerland, 2010. [CrossRef]
14. Pepper, P.; Mehlhase, A.; Höger, C.; Scholz, L. A Compositional Semantics for Modelica-style Variable-structureModeling. In Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, EOOLT'11, Zürich, Switzerland, 5 December 2011; pp. 45–54.
15. Mehlhase, A. A Python framework to create and simulate models with variable structure in common simulation environments. *Math. Comput. Model. Dyn. Syst.* **2014**, *20*, 566–583. [CrossRef]
16. Mattsson, S.E.; Otter, M.; Elmqvist, H. Multi-mode DAE systems with varying index. In Proceedings of the 11th International Modelica Conference, Versailles, France, 21–23 September 2015; pp. 89–98. [CrossRef]
17. Tinnerholm, J.; Pop, A.; Sjölund, M. A Modular, Extensible, and Modelica-Standard-Compliant OpenModelica Compiler Framework in Julia Supporting Structural Variability. *Electronics* **2022**, *11*, 1772. [CrossRef]

18. Modelica Association. Functional Mock-Up Interface for Model Exchange and Co-Simulation—Version 2.0. 2014. Available online: https://fmi-standard.org/assets/releases/FMI_for_ModelExchange_and_CoSimulation_v2.0.pdf (accessed on 13 January 2023).
19. Steinbach, O. *Numerical Approximation Methods for Elliptic Boundary Value Problems: Finite and Boundary Elements*; Springer: New York, NY, USA 2007. [CrossRef]
20. Campbell, S.L.; Linh, V.H.; Petzold, L.R. Differential-algebraic equations. *Scholarpedia* **2008**, *3*, 2849. [CrossRef]
21. Olsson, H.; Otter, M.; Mattsson, S.; Elmqvist, H. Balanced Models in Modelica 3.0 for Increased Model Quality. In Proceedings of the 8th International Modelica Conference, Bielefeld, Germany, 3–4 March 2008; pp. 21–33.
22. Elmqvist, H. Pages 7–10 of Modia—A Prototyping Platform for Next Generation Modeling and Simulation Based on Julia. Jubilee Symposium 2019: Future Directions of System Modeling and Simulation. Available online: https://modelica.github.io/Symposium2019/slides/jubilee-symposium-2019-slides-elmqvist.pdf (accessed on 4 December 2022).
23. Otter, M. Signal Tables: An Extensible Exchange Format for Simulation Data. *Electronics* **2022**, *11*, 2811. [CrossRef]
24. Snethen, G. Xenocollide: Complex collision made simple. In *Game Programming Gems 7*; Course Technology; Charles River Media: Newton, MA, USA, 2008; pp. 165–178.
25. Neumayr, A.; Otter, M. Collision Handling with Variable-step Integrators. In Proceedings of the 8th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, EOOLT'17, Weßling, Germany, 1 December 2017; pp. 9–18. [CrossRef]
26. Hertz, H. On the contact of solids—On the contact of rigid elastic solids and on hardness. In *Miscellaneous Papers*; MacMillan: Stuttgart, Germany, 1896; pp. 146–183. Available online: https://archive.org/details/cu31924012500306 (accessed on 13 January 2023).
27. Flores, P.; Machado, M.; Silva, M.T.; Martins, J.M. On the continuous contact force models for soft materials in multibody dynamics. *Multibody Syst. Dyn.* **2011**, *25*, 357–375. [CrossRef]
28. Neumayr, A.; Otter, M. Collision Handling with Elastic Response Calculation and Zero-Crossing Functions. In Proceedings of the 9th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, EOOLT'19, Berlin, Germany, 5 November 2019; pp. 57–65. [CrossRef]
29. Nystrom, R. *Game Programming Patterns*; Genever Benning, 2014.
30. Unity Technologies. Unity—Manual: Unity User Manual 2021.3 (LTS). Available online: https://docs.unity3d.com/Manual/index.html (accessed on 27 April 2022).
31. Epic Games. Unreal Engine 5 Documentation | Unreal Engine Documentation. Available online: https://docs.unrealengine.com (accessed on 27 April 2022).
32. Three.js. Available online: https://threejs.org/docs/#api/en/core/Object3D (accessed on 13 October 2022).
33. Neumayr, A.; Otter, M. Algorithms for Component-Based 3D Modeling. In Proceedings of the 13th International Modelica Conference, Regensburg, Germany, 4–6 March 2019; Linköping University Electronic Press: Linköping, Sweden, 2019. [CrossRef]
34. Keller, A. Unitful.jl. Available online: https://github.com/PainterQubits/Unitful.jl (accessed on 12 December 2022).
35. Otter, M.; Elmqvist, H. Transformation of Differential Algebraic Array Equations to Index One Form. In Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, 15–17 May 2017; Linköping University Electronic Press: Linköping, Sweden, 2017. [CrossRef]
36. Rackauckas, C.; Nie, Q. DifferentialEquations.jl—A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. *J. Open Res. Softw.* **2017**, *5*, 15. [CrossRef]
37. DifferentialEquations.jl. Available online: https://github.com/SciML/DifferentialEquations.jl (accessed on 12 December 2022).