The Value of Enforcing a Strict Modeling Methodology within Modelica

Dirk Zimmer[1]

[1]Institute of Robotics and Mechatronics, German Aerospace Center (DLR), Germany,
dirk.zimmer@dlr.de

Abstract

Self-restriction to a certain modeling style can enable the modeling of large-scale systems and the robust modeling of complex system architectures. This paper discusses how such a self-restriction can be achieved within the Modelica language and provides a corresponding example.

Keywords: Modeling Methodology, Language Design, Object-Oriented

1 Motivation

Modelica provides the modelers with a lot of freedom on how to create and compose their models. This freedom contributed significantly to the appeal of the language. It enabled many modelers to solve even exotic problems while the powerful algorithms for symbolic index-reduction in Modelica compilers still enable the generation of efficient simulation code. Hence Modelica has become the language of choice for many experts in their fields, especially in thermal and energy related fields where many specialized solutions are needed.

As always, this freedom is associated with a price to be paid for it. First, what is appealing to an expert maybe overwhelming for a novice. Beginners in Modelica often struggle with decrypting the error-messages and often yearn for more guidance. Even after 25 years, the entry barrier to the world of equation-based modeling is high. Second, since equations can be entered in any form, the code generation has to be relegated to the very last stages in processing. For monolithic simulation code, this is fine but code for large-scale systems on GPUs or for variable structure systems require differently structured code that becomes available before final system assembly. This is difficult to fulfill for current Modelica models, despite notable efforts from (Benveniste 2023, Neumayr 2023).

Being self-aware of its flaws, the Modelica community has always remained open to take inspiration from new language developments. Whether these are experimental languages like MOSILAB (Nytsch-Geusen 2006), SOL (Zimmer 2010) or larger efforts such as Modia (Elmqvist 2021). However, these developments also revealed how strong the Modelica legacy is and that maybe a new language or a major extension is not what is needed.

Indeed, this paper argues that not an extension of the Modelica language is needed but that a self-imposed restriction would benefit a significant set of applications. The conflict between an expert and a novice does not need to exist. We can formulate basic models by restricted means and then let the experts extend from this giving them free choice of their tools. The conflict between different possible compile paths also does not need to exist. We can provide models in a restricted form that enable local analysis and local compilation and when these restrictions are lifted, full global analysis for elaborate index-reduction is enabled.

We shall remind ourselves that it is the strength of a declarative language that it does not stipulate its computational realization. When we are stricter about the declarative form, we typically gain freedom on how to process the content.

2 Robust Modeling Methodology

The greatest source of complexity in the processing of equation-based models are systems that are over-idealized. Over-idealization means that the system has been idealized so strongly that the information on how to solve this system has been lost in the process of modeling. A great educational collection of such examples has been provided by Peter Junglas (Junglas 2025):

- A singular model of gear shift for a car transmission: the over-idealization neglected the necessary synchronization during the shift.
- A singular model of an RS Flip-Flop: the over-regularization neglected the signal delays within the components
- A model for the distribution of compressed air with unsolvable non-linear equations: the over-idealization neglected the kinetic energy of the fluid that would lead to the flow balance.
- etc.

Unfortunately, the formulation of over-idealized systems has a strong legacy in text-books and modeling. To make matters worse the awareness of this phenomenon is often little to none. Over-idealized systems not necessarily lead to singular systems but often also to structural regular systems involving non-linear equation system in implicit

form and states with non-linear constraints between potential state variables.

Because the information on how to solve these models has been removed, a compiler for such Modelica models has essentially to perform blunt guessing (often euphemistically formulated as "applying clever heuristics"). No matter how elaborated such guesswork may be, it involves severe disadvantages:

- It needs be performed on a global level to exploit the full context and hence a global analysis of the model is needed.
- It does not scale. While guessing may work for system of low dimensions, it will eventually fail for increasingly large dimensions.
- The compile process becomes non-deterministic. Different compilers may have different guessing strategies.

It is important to emphasize that this is not a problem of the compiler's algorithms or of any of the simulation tools. It is a problem of the input to the compiler. If the information entering the system is incomplete, no algorithm can magically retrieve this missing information. This is basic information theory.

The only solid solution to this problem is thus to insist on input that is information complete, meaning models that are idealized (as all models are idealizations) but not over-idealized.

Fortunately, a suitable model class has been recently identified. We call them Linear Implicit Equilibrium Dynamics (LIED), (Zimmer 2024)

A DAE system with potential state derivatives $\dot{\boldsymbol{x}}$, time t and algebraic variables $\mathbf{w}$

$$\mathbf{0} = \boldsymbol{F}(\dot{\boldsymbol{x}}, \boldsymbol{x}, \mathbf{w}, t)$$

is defined as LIED system when it can be transformed into the following form:

$$\begin{bmatrix} \mathbf{w}_E \\ \dot{\boldsymbol{x}}_E \end{bmatrix} = \mathrm{g}(\boldsymbol{x}_I, \boldsymbol{x}_E, t)$$

$$\mathbf{A}(\boldsymbol{x}_I, \boldsymbol{x}_E, \mathbf{w}_E) \begin{bmatrix} \mathbf{w}_I \\ \dot{\boldsymbol{x}}_I \end{bmatrix} = \mathrm{f}(\boldsymbol{x}_I, \boldsymbol{x}_E, \mathbf{w}_E, t)$$

We see that both the algebraic variables as well as the state derivatives can be split into a fully explicit part $(\dot{\boldsymbol{x}}_E;\ \mathbf{w}_E)$ and a part $(\dot{\boldsymbol{x}}_I;\ \mathbf{w}_I)$ with a linear system in implicit form expressed by the regular matrix $\mathbf{A}$. Furthermore, the following conditions shall hold true:

$$\dot{\boldsymbol{x}}_E \cap \dot{\boldsymbol{x}}_I \subseteq \dot{\boldsymbol{x}}$$

$$\mathbf{w}_E \cap \mathbf{w}_I \supseteq \mathbf{w}$$

$$\dot{\boldsymbol{x}}_E \cap \dot{\boldsymbol{x}}_I \cap \mathbf{w}_I \supseteq \dot{\boldsymbol{x}}$$

$$\dot{\boldsymbol{x}}_E, \dot{\boldsymbol{x}}_I, \mathbf{w}_E, \mathbf{w}_I \text{ are all disjoint}$$

These conditions essentially mean that it is allowed to perform certain symbolic mechanism of index reduction such as the dummy derivative method (Mattsson1993) originating from Pantelides (Pantelides 1988). Using this method, states variables of $\boldsymbol{x}$ can be transformed to algebraic variables in $\mathbf{w}_I$ and further derivatives may be added to $\mathbf{w}_I$ or $\mathbf{w}_E$. In practice, this is important because it means that the linear implicit dynamics can be expressed by far fewer states than suggested by the vector $\boldsymbol{x}$ of the original DAE formulation.

Whereas this model class may seem to be overly restrictive in the first place we could demonstrate that it is not only applicable but also highly attractive for the modeling of stiff mechanical systems as in Figure 2 (Zimmer 2023) and complex thermo-fluid architectures (Zimmer 2022, Junglas 2023) as in Figure 1. Further library development for energy and electric systems is currently ongoing.

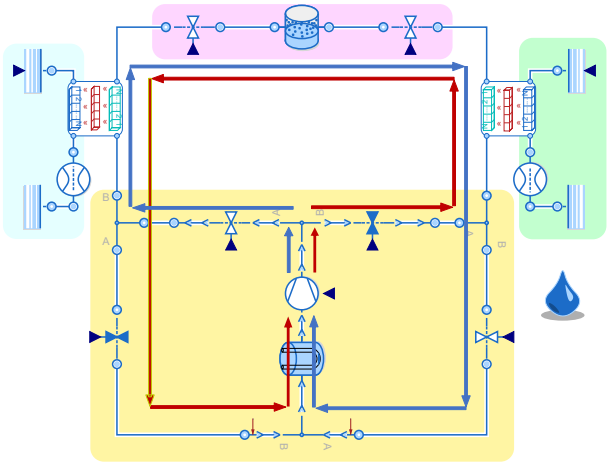
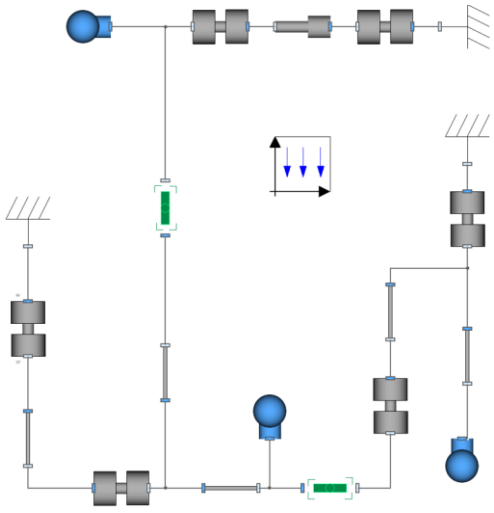


Figure 1. Model diagram of a heat pump using the DLR ThermoFluid Stream library (Zimmer 2022)



DOI 10.3384/ecp218147

Figure 2. Model diagram of a mechanical system for a kinematic using the Dialectic Mechanics library (Zimmer 2023).

The goal is now to define a language that is restricted for such LIED systems and that also represent valid Modelica code. The expected advantages are naturally contrarian to the previous bullet list for over-idealized models:

- Models can be analyzed locally and early compilation of components is enabled.
- It scales. The dimension of the system will principally not impact its solvability barring other issues such as numerics or memory consumption.
- The compile process can be formulated in fully deterministic way increasing the value of a reference implementation.

3 Demonstration Example

Whereas the two preceding sections made very abstract arguments, we shall now provide a concrete example and then discuss the design ideas based on this code. Listing 1 was placed in the Appendix and presents the model of a 2D multibody library implementing components according to dialectic mechanics. The code has been stripped from most annotations, occasionally shortened and simplified for presentation purposes.

3.1 Assumed Causality

A key point of the robust modeling approach is that models are formulated under the assumption of causality. This enables two major benefits:

1. Code generation for individual components is enabled.
2. Error messages point to a single line holding either a equation whose causality is not correct or to a variable definition that is not causalized.

To achieve this, the connectors need to have an assumed causality and what derivates are needed. Also, we need to be able to formulate what is a state and what variables can be used to tear a linear equation system. Let us go through each of these points:

Connectors with assumed causality

Input and outputs are the natural way to formulate a causal connector. Whenever inputs and outputs are used we will use them as indicator of causality: the input is assumed to be known to its component and the output is unknown (and the reverse is true to the outside).

For the modeling of physical systems, conjugated pairs of potential and flow variables have proven to be useful. In order to formulate these pairs with an assumed causality, they are not allowed to stand on their own but need to be associated with a signal connector.

A potential variable may be declared in a connector after a signal and then inherits the assumed causality from this signal. The flow variable must then be declared after the potential variable and is assumed to be of inverse causality.

Two examples of such connectors are found in Listing 1, lines 35-51. The `FlangeOn` is supposed to be on the root side of a kinematic chain and hence its variable for position forms an output signal. The corresponding potential is the velocity and shares this causality. The flow is the force (or torque respectively and of inverse causality). The `FlangeTo` forms then the connector of opposite sex.

The example demonstrates another feature defined on Listing 1, line 14 and used in lines 30-34. A new special predefined type called `RealWithDerivative` that indicates that not only the potential variable is available in the connector but also its derivative. Providing this information means that differential index-reduction can be performed on the component level.

For a standard Modelica compiler, it is not necessary to have such a special type hence the definition on line 15 simply defines a `Real`. The definition exists so that other compilers or a compliance checker can replace this default definition by a special inner representation.

Given such causalized connectors, the model equations can now be provided in partially ordered form: each equation can be causalized based on the previous equations. A simple example for this is the body model in Listing 1, lines 130 – 150. It assumes the position and the derivatives to be known and calculates the forces and torques from the acceleration.

Declaring Tearing of Linear Equation Systems

Since the modeler shall formulate everything under the assumption of causality, this also impacts how implicit linear equation systems across components have to be formulated. Often the modeler expects the model to react to a variation by a linear response. This observation enables to formulate such a system in causalized and torn form. To indicate a tearing of a linear equation system, a special predefined model is being declared and then applied. The model is called `LinearResponse`.

The revolute joint element shows how to use it. The modeler expects that a linear response in force results from a change in acceleration. He formulates it accordingly by declaring the model `LinearResponse`. The assumed causality of the model now naturally follows from this.

Within Modelica the `LinearResponse` model can be formulated as in Listing 1, line 6-12. This definition ensures that the model will be correctly understood by any existing Modelica tool. Special tools for this modeling

methodology may replace this definition with an internal representation.

Declaring States of the System

A predefined type is also used to select the state of the system. In this case the state variable is declared as `ContinuousState`.

The obvious question is why using this mechanism and not the already available `StateSelect` attribute? The reason is to enable an orthogonal mechanism for state selection.

The special model is used for the restricted modeling methodology presented here and aims to enable component-wise compilation. The state-select attribute is to be set by an expert for full complex (non-restricted) Modelica models. The motives for selecting states can be quite different and hence it seems a good idea to avoid any potential conflict.

When translating a model, the state variable is assumed to be known and the derivative is assumed as unknown. If you take the derivative of any non-state variable than this variable has to be known at this point and the derivative is derived from this. The `Body` component is an example from this. Its derivative does not define a state.

3.2 No Look-ahead

In our restricted modeling methodology, we shall declare or define items before we use them. Basically, a compiler or a compliance checker for this restricted set shall be able to "understand" each line without having to read ahead. This concession significantly reduces the complexity of a compiler since many processes can now be performed directly on the stack instead of having to work on intermediately stored parts. Far less internal data-structures are thus needed.

One result of this requirement is that packages must not be mutually depending on each other anymore. The common pattern that is used by many existing Modelica libraries does not match this requirement. We have to abandon this pattern and separate interfaces from components from examples in order to shed unnecessary dependencies.

We can see this in Listing 1 where packages for interfaces, planar components and examples are separated and all encapsulated. Import statements are restricted to encapsulated packages. By separating interfaces, components and examples the interdependencies are reduced.

While providing a short-term inconvenience, this restriction leads to code that is better maintainable in the long-term and avoids "spaghetti" packages. Even a modern language like Go enforces such a rule for good reasons.

3.3 Feature Elimination

When we restrict ourselves to a certain modeling style, this also offers an opportunity to trim the feature set of the language. There are a number of small features which have only very specific purpose or application in the standard Modelica language that we shall abandon for this modeling methodology:

- connections.branch/root (used mostly in MultiBody)
- stream (used in Modelica.Fluid)
- operator overload
- expandable connectors
- delay
- homotopy
- inline
- etc.

As the reader can see from the above list of abandoned language features, it is back to basics again. Another significant simplification is that annotations can be part of the code but are completely ignored for the process of compliance checking or compiling. The information within the annotation is hence only regarded as auxiliary meta-information for non-compilation processes as it was the original intention in early Modelica (this is also an additional reason why to use special predefined types and models such as in lines 2-19 rather than custom annotations)

For the moment, the focus is also on continuous systems but of course, discrete events also need to be handled. This shall follow the same guidelines yielding a fully deterministic process that can be locally analyzed. However, this is still upcoming work.

3.4 Compatiblity to Modelica

Evidently, the restrictions that are imposed to the modeler are significant. The Modelica Standard Library clearly does not comply to these restrictions and other existing libraries are also unlikely to do so.

We hence need new libraries developed under this methodology. This is no surprise because there was previously no sufficiently practical guidance on how to model in an object-oriented way that is also information complete.

What is important however that libraries that are developed under these restrictions are fully usable with any tool supporting the Modelica standard. Even if these tools do not exploit the special characteristics of the restricted modeling methodology, they still profit from the fact that these models are information complete. Not only shall these libraries work in any Modelica tool, they shall also be (automatically without further tool support) be very robust in their usage.

DOI 10.3384/ecp218147

We currently engage in the development of 4 libraries:

1. Signal Blocks
2. Dialectic Mechanics
3. Controlled Energy Flows
4. ThermoFluid Stream Lite

4 Implementation

To support the development of these new libraries, a Compliance checker is being implemented. It serves two functions:

1. The implementation formalizes the restrictions that have been informally stated above.
2. It provides instant feedback to modelers whether their library abides to these restrictions.

Formalizing the restricted modeling methodology will ultimately result in a new language that forms a sub-set of Modelica. The compliance checker is thus not developed from an existing Modelica compiler or tool but as a new tool from scratch. This is the most effective way to shed complexity and question all the existing requirements.

The compliance checker is implemented as a stand-alone console application in C++17 using only the Standard Template Library (STL). Given that the portability of C++ code between operating systems and compilers has significantly improved over the last decade the application may be applied on different operating systems.

We can use the compliance checker to keep track of the complexity of the new restricted language. Figures 3 and 4 measure the complexity of the C++ code compared to the Modelica specification which is written in English language. The counting for the code includes comments and the counting for the specification includes also the examples.

Although we compare an executable code against a mere description in natural language, we still have far less complexity. This shows that (at least for now) the reduction in complexity is very effective.

How is this compliance checker supposed to be used? As a console application it runs next to the Modelica development environment of your choice and automatically checks whenever a modification is saved whether the file(s) comply to the restricted modeling methodology. Otherwise it provides an error message that always refers to a specific line of code.

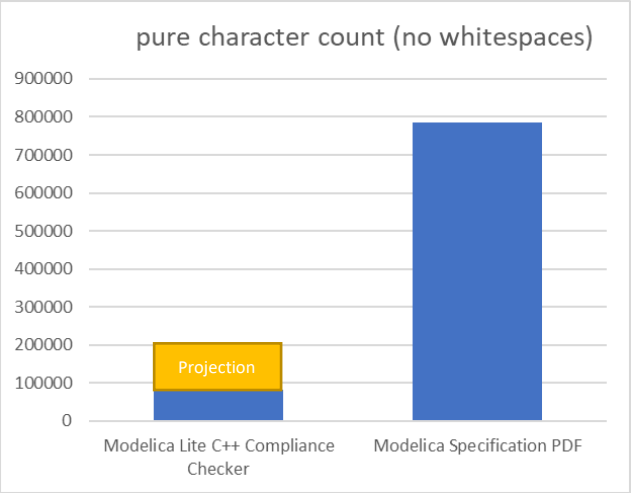


Figure 3. Complexity of the compliance checker implementation in C++ for a restrictive Modelica vs Modelica Specification PDF version in character count. Projection indicates an estimate for the complete compliance checker.

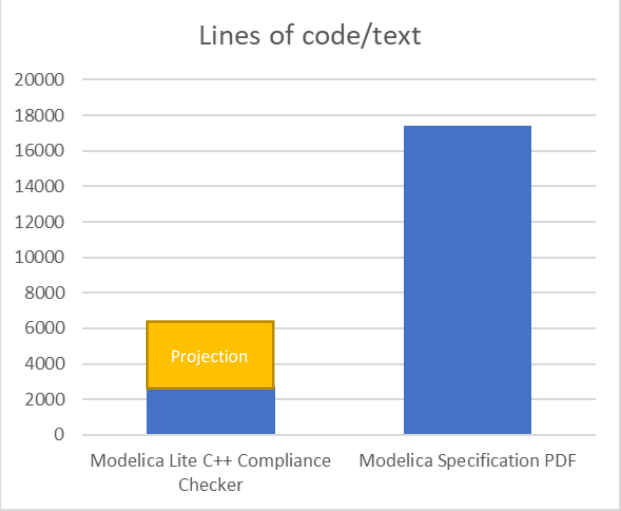


Figure 4. Complexity of the compliance checker in C++ for a restrictive Modelica vs Modelica Specification PDF version in lines of code.

In this way, the existing IDEs for Modelica can be effectively reused but the modeler is restricted with instant feedback. In future, the compliance checker can be extended to enable different compile paths. After all the compliance checker already performs many processing steps of a full compiler such as: lexer, parser, type generation, identifier resolution, partial casualization, etc.

5 Conclusions

Modelica has gained its popularity by providing the modeler ample freedom in the formulation of its models.

However, there are many applications that profit from a more restrictive modeling methodology:

- For complex architectures, (especially when modeled across different organizational entities) the robustness of the model often becomes the limiting factor in model development. If a strict modeling methodology allows stronger a-priori statements on solvability and pinpoint error messages, it will be highly welcome.
- For large scale systems or variable structure systems, a global analysis of the system increasingly becomes infeasible. If a strict modeling methodology requires only a local analysis and enables compilation of components, it will be of high value.
- Modern compute architectures like GPUs (Kirk 2022) demand different forms of code generation. Modern numerical methods such as multi-derivative methods (Krivovichev 2024) also require different compilation schemes. If a strict modeling methodology eases the compiler development by massively reducing complexity, it will find quick adaptation.

As shown in the previous section, it is possible to support such a strict methodology fully within the current version of Modelica without demanding any changes to the existing language. It is basically an act of self-discipline. Some models may seem unusual for a Modelica purists but in view of the listed advantages, one should advocate for pragmatism.

The currently undergoing implementation of a compliance checker will ease this discipline, formalize the restrictions and open up the development of independent compile paths. We plan to release the compliance checker as open-source software.

In this way, we can have the best of both worlds. Confined modeling efforts for special systems can be performed by experts using the full feature-set of Modelica. Modeling efforts of larger architectures or with versatile compute architectures can be performed using a strict modeling methodology such as the one presented here.

Acknowledgements

This work was partially supported by the ITEA4 research project OpenScaling with financial support of the German federal ministry of education and research referring to the grant number: 01IS23062C

SPONSORED BY THE

Federal Ministry of Education and Research

References

Benveniste, A. B. Caillaud, M. Malandain, J. Thibault (2023) Towards the separate compilation of Modelica: modularity and interfaces for the index reduction of incomplete DAE systems. *Proceedings of the 15th International Modelica Conference*, Aachen.

Elmqvist H., M. Otter, A. Neumayr and G. Hippmann (2021) Modia - Equation Based Modeling and Domain Specific Algorithms *Proceedings of the 14th International Modelica Conference*, Linköping.

Junglas, P. (2025) Mathematical Problems due to Oversimplification *Keynote at the ASIM Workshop* STS, GMMS, EDU. https://www.peter-junglas.de/fh/talks/2025-oberpfaffenhofen/oversimp/html/index.html

Junglas, P. (2023) Implementing Thermodynamic Cyclic Processes Using the DLR Thermofluid Stream Library. *Simulation News Europe* E 33(4)

Kirk, David B., Izzat El Haji, Wen-mei W. Hwu (2022). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 4th Edition.

Krivovichev, G. V. (2024). Stability Optimization of Explicit Runge–Kutta Methods with Higher-Order Derivatives. *Algorithms*, 17(12), 535. https://doi.org/10.3390/a17120535

Mattsson, S.E., Gustaf Söderlind (1993). "Index Reduction in Differential-Algebraic Equations Using Dummy Derivatives" In: *SIAM Journal on Scientific Computing* 1993 14:3, 677-692

Neumayr, A.; Otter, (2023) M. Modelling and Simulation of Physical Systems with Dynamically Changing Degrees of Freedom. Electronics, doi.org/10.3390/electronics12030500

Nytsch-Geusen, C. et al. (2006) Advanced Modeling and Simulation Techniques in MOSILAB: A System Development Case Study. In *Proc. of the 5th International Modelica Conference*, Vienna Austria, Vol. 1, pp. 63-71.

Pantelides, C. (1988), The consistent initialization of differential-algebraic systems, *SIAM J. Sci. Statist. Comput*., 9, 213–231

Zimmer, D. (2010), *Equation-Based Modeling of Variable Structure Systems*, PhD-Thesis, ETH Zurich.

Zimmer, D. (2020), Robust Object-Oriented Formulation of Directed Thermofluid Stream Networks . *Mathematical and Computer Modelling of Dynamic Systems*, Vol 26, Issue 3.

Zimmer, D., N. Weber, M. Meißner (2022) *The DLR ThermoFluid Stream Library*. MDPI Electronics - Special Issue.

Zimmer, D., C. Oldemeyer (2023). "Introducing Dialectic Mechanics". *Proceedings of the 15th International Modelica Conference*, Aachen.

Zimmer (2024) Object-Oriented Modeling of Classic Physical Systems using Linear Implicit Equilibrium Dynamics *Preprints* 2024, 2024031139

Appendix

Listing 1. Example code

```
encapsulated package Base "Base elements”

  type ContinuousState = Real;

  model LinearResponse "continuous state"
    input Real residual;
    output Real balance;
  equation
    residual = 0;
     annotationdefaultConnectionStructurallyInconsistent=true);
  end LinearResponse;

  type RealWithDerivative = Real;
  type RealWith2Derivatives = Real;

  package Units
    [...]
  end Units;

end package Base;


encapsulated package Mechanical

  encapsulated package Interfaces
    import B = Base;
    import SI = Base.Units.SI;

    type VelocityInt = B.RealWithDerivative
                         (unit = "m/s")
    type AngularVelocityInt =
      B.RealWithDerivative(unit = "rad/s")

    connector FlangeOnPlanar
      output SI.Position r0[2];
      output SI.Angle phi;
      VelocityInt v0[2];
      AngularVelocityInt w;
      flow SI.Force f0[2];
      flow SI.Torque t;
    end FlangeOnPlanar;

    connector FlangeToPlanar
      input SI.Position r0[2];
      input SI.Angle phi;
      VelocityInt v0[2];
      AngularVelocityInt w;
      flow SI.Force f0[2];
      flow SI.Torque t;
    end FlangeOnPlanar;

    [...]
  end package Interfaces;

  encapsulated package Planar
    import B = Base;
    import Interfaces = Mechanical.Interfaces;
    import SI = package Base.Units.SI;
```

```
    package Motion

      model Root "Frame fixed"

        parameter SI.Position r0_abs[2] = {0,0}
        "Fixed absolute x,y-position";
        parameter SI.Angle phi_abs = 0
        "Fixed angle";
        Interfaces.FlangeOnPlanar flangeOnP;

      equation
        flangeOnP.r0 = r0_abs;
        flangeOnP.phi = phi_abs;
        flangeOnP.v0 = {0,0};
        flangeOnP.w = 0;
      end model Root;

      model RevoluteJoint "Ideal Revolute Joint"
        Interfaces.FlangeToPlanar flangeToP;
        Interfaces.FlangeOnPlanar flangeOnP;
        parameter SI.Time TD
        "dialectic time constant";

      protected
        B.LinearResponse Cut;

        B.ContinuousState phi(unit="rad")
        "position of component";
        B.ContinuousState w(unit="rad/s")
        "(kinetic) velocity of component";
        SI.AngularAcceleration z
        "acceleration of component";
        SI.Torque t "actuation torque";

      equation

        z = Cut.balance;

        //dialectic damping term
        der(phi) = w + z*TD;
        der(w) = z;

        //rigidly connect positions
        flangeOnP.r0 = flangeToP.r0;
        flangeOnP.v0 = flangeToP.v0;
        flangeOnP.phi = flangeToP.phi + phi;
        flangeOnP.w = flangeToP.w + w;

         //balance forces
        flangeToP.f0 = -flangeOnP.f0;
        flangeToP.t = -flangeOnP.t;
        t = flangeOnP.t;
        Cut.residual = t;

      end RevoluteJoint;

    end Motion;
```

```
128    package Parts
129
130      model Body
131        Interfaces.FlangeToPlanar flangeToP;
132        parameter SI.Mass m  "Mass of the body";
133        parameter SI.Inertia J "Inertia";
134        parameter SI.Acceleration g[2]
135        "Gravity";
136        parameter Boolean enableGravity = true;
137
138      protected
139        SI.Acceleration a0[2] "Acceleration";
140        SI.AngularAcceleration z
141        "Angular acceleration";
142
143      equation
144        a0 = der(flangeToP.v0);
145        z = der(flangeToP.w);
146        flangeToP.f0 =
147          if enableGravity then m*a0-m*world.g
148          else m*a0;
149        flangeToP.t = J*z;
150      end Body;
151
152    end Parts;
153
154  end Planar;
155
156  encapsulated package Examples
157    import Planar = Mechanical.Planar;
158    [...]
159
160  end package Examples;
161
162end package Mechanical;
```


DOI
10.3384/ecp218147