Enabling the Compilation of Individual Components for Systems of Linear Implicit Equilibrium Dynamics

Dirk Zimmer,
Institute of System Dynamics and Control

08.02.2024 – MODPROD Workshop, Linköping Sweden

DLR

Motivation: From Necessary to Sufficient

- Our current standard interfaces, ultimately result from Hamilton's trick to double the dimension. They are what is ***necessary*** for object-oriented modeling.

Domain	Translational Mechanics	Rotational Mechanics	Hydraulics	Electrics	Thermal	…
Potential	r	φ	P	V	T	
Flow	f	τ	$\dot{Q}$	i	Q	

- We can find extended interfaces that offer a ***sufficient*** form. (Unfortunately hardly anyone is looking for these forms)

Domain	Translational Mechanics	Rotational Mechanics	Thermo Fluids	Electrics	?	…
Potential	v_{kin}	ω_{kin}	r	?	…	
Flow	f	τ	$\dot{m}$	?	…	
Signal	r	φ	$\boldsymbol{\Theta}$	?	…	

Definition of a Linear Equilibrium Dynamics System

- The way of modeling that we derived leads to a special class of DAE systems: Linear Implicit Equilibrium Dynamics.

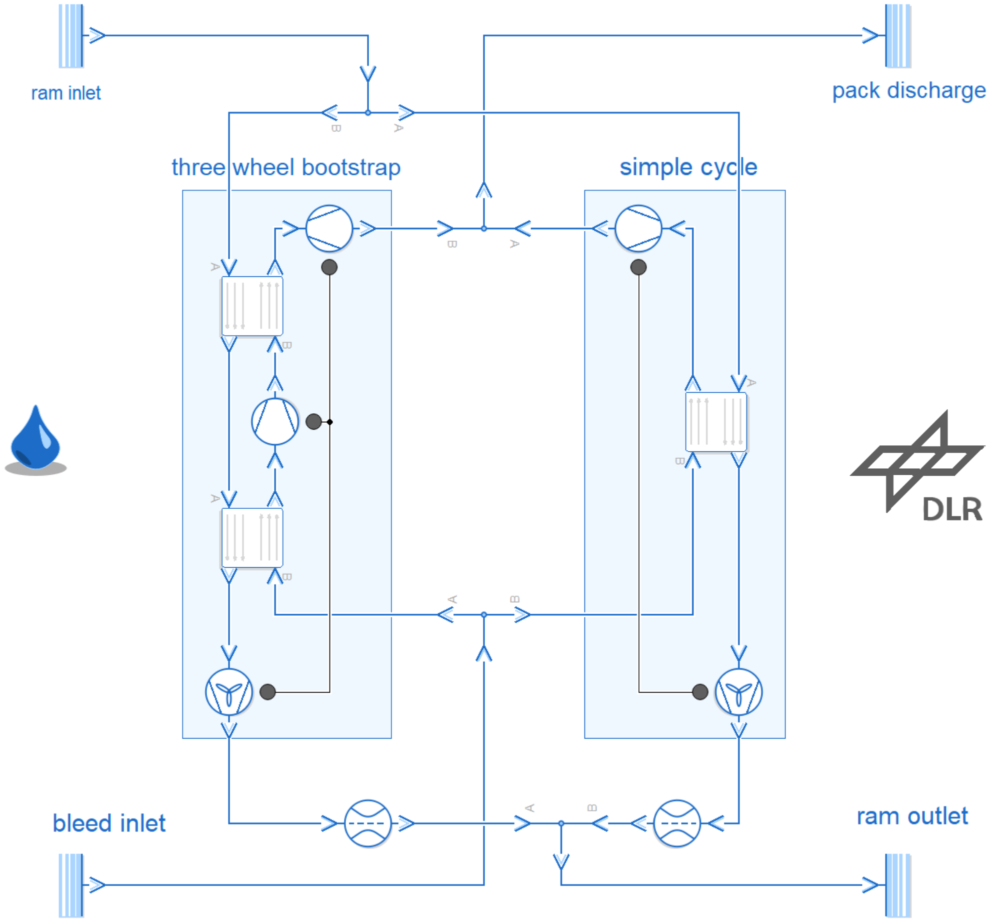
$$\mathbf{0} = \mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{w}, t) \quad \Rightarrow \quad \mathbf{A}(\mathbf{x}) \begin{bmatrix} \dot{\mathbf{x}} \\ \mathbf{w} \end{bmatrix} = \mathbf{b}(\mathbf{x}, t)$$

The DAE is linear in the state derivatives $\dot{\mathbf{x}}$ and the algebraic variables $\mathbf{w}$
(besides small nonlinear algebraic equations that may arise inside components to compute local variables of a component and where the component developer ensures that a solution can be reliably computed. When hidding such nonlinear algebaric equations inside functions (at least conceptually), the above structure holds)

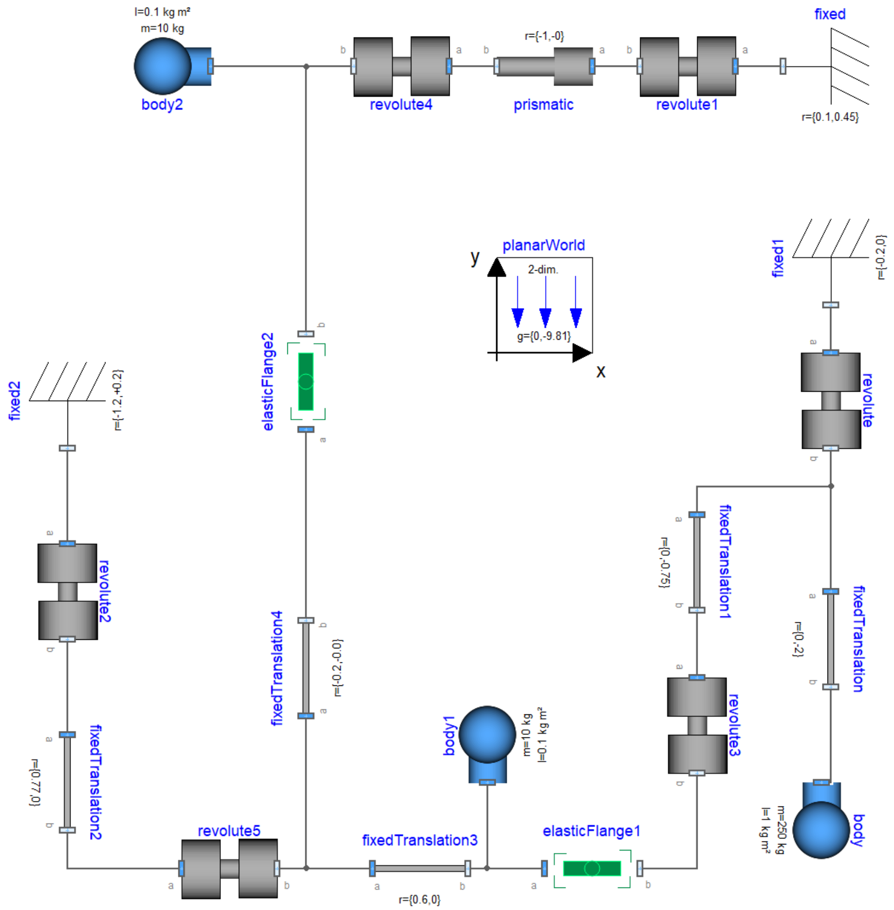
- What looks like a very restrictive class of models is actually much more powerful than expected.

Basis for new Modelica Libraries

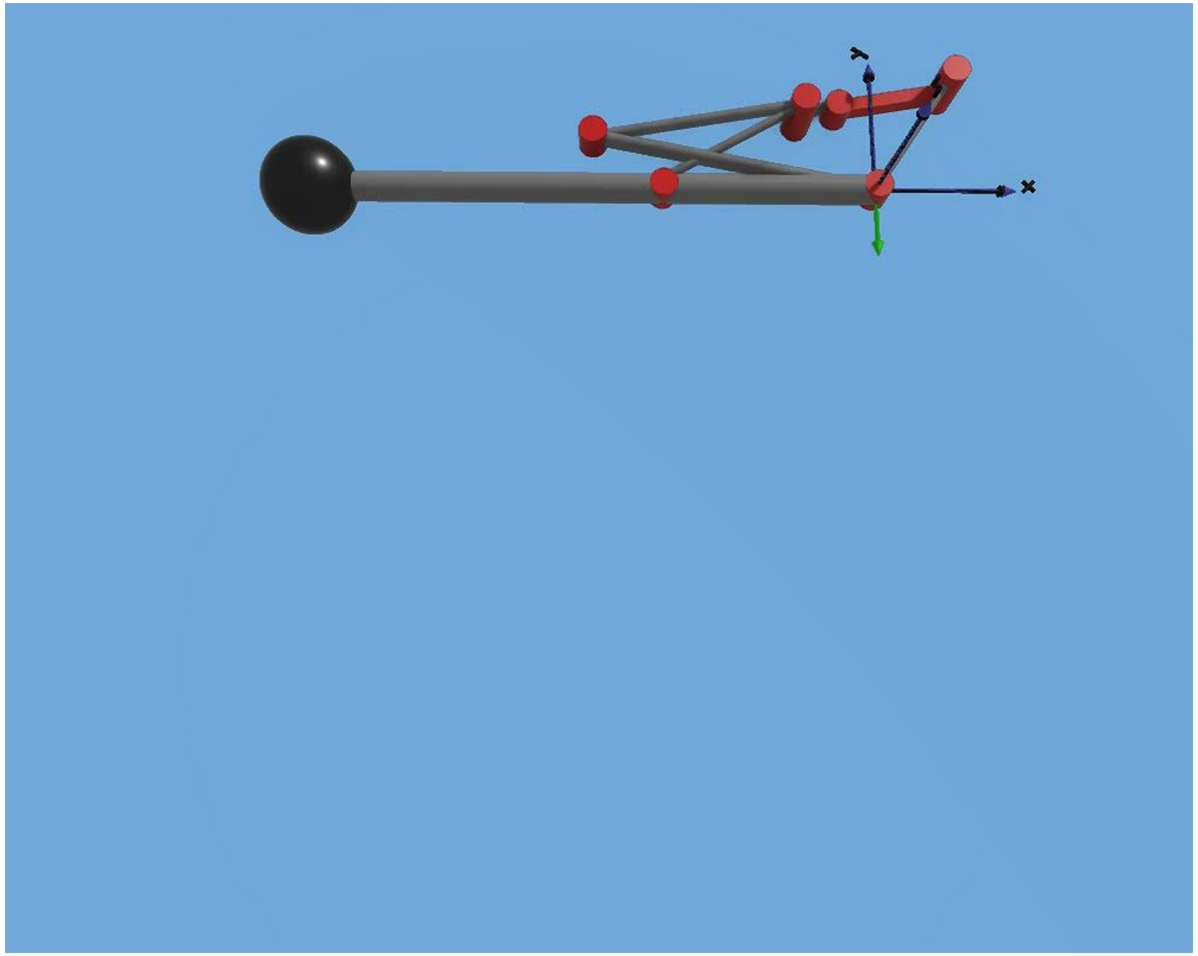
- DLR ThermoFluid Stream: github.com/DLR-SR/ThermofluidStream



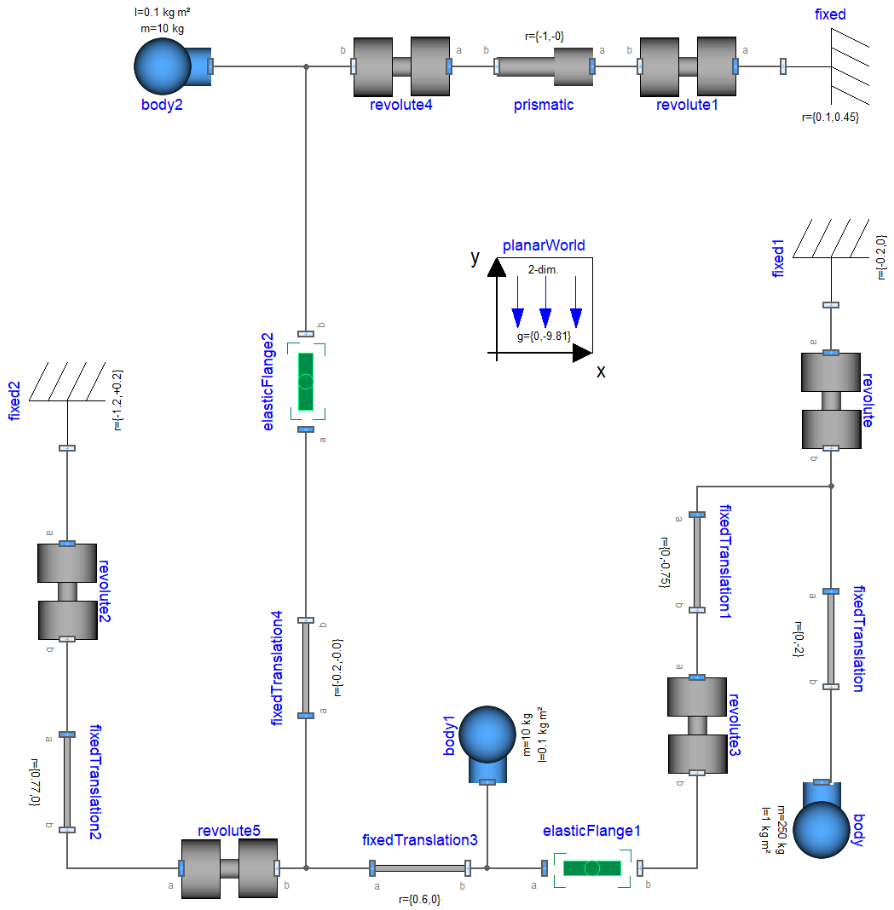
- Dialectic Mechanics: (internal development)



Basis for new Modelica Libraries

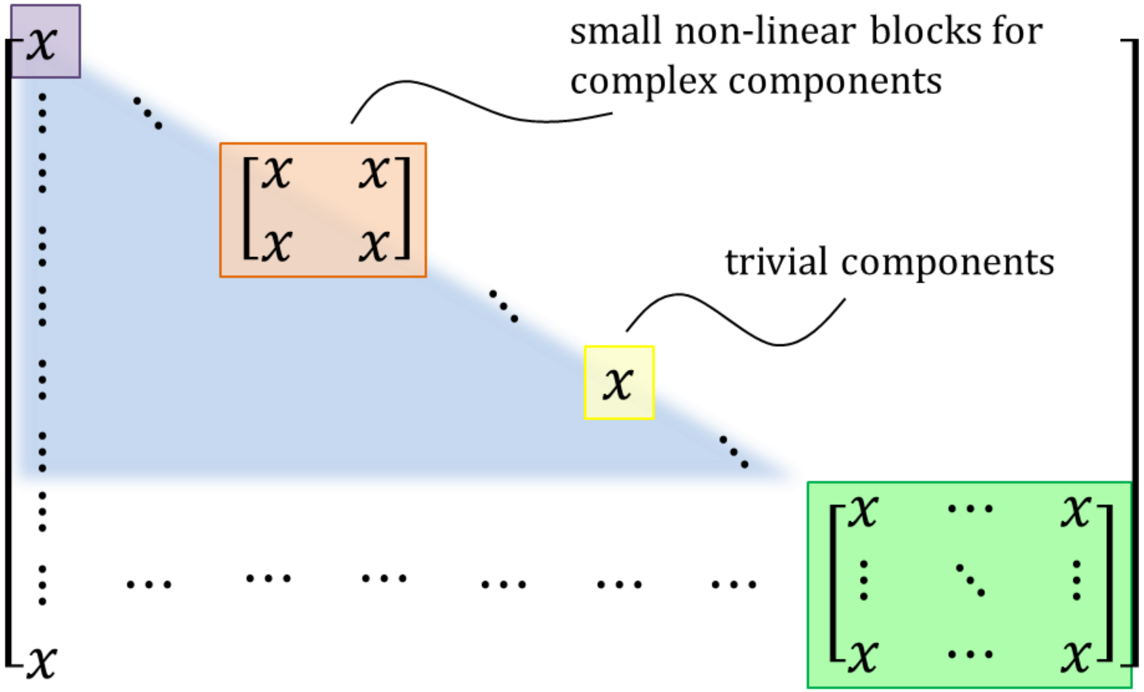


- Dialectic Mechanics:
  (internal development)



Robustness as Original Motivation

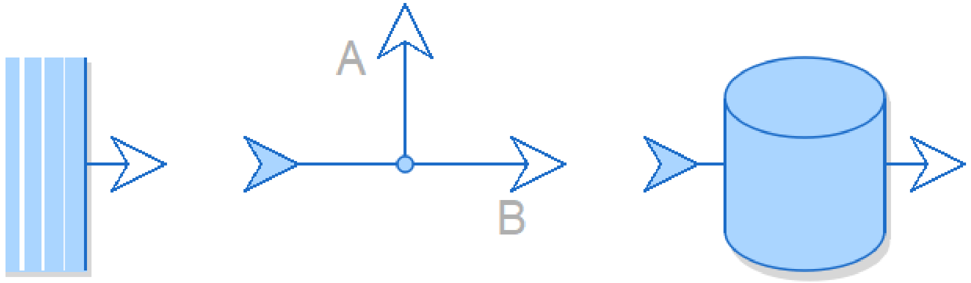
$$\mathbf{0} = \mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{w}, t) \quad \Rightarrow \quad \mathbf{A}(\mathbf{x}) \begin{bmatrix} \dot{\mathbf{x}} \\ \mathbf{w} \end{bmatrix} = \mathbf{b}(\mathbf{x}, t)$$



- The original motivation was simply to enable a robust solution of the total system by insisting on the **implicit system to be linear**.
- However, it then later revealed to us that the enforced linearity also is rewarded by a **structural certainty** *(w.r.t causality, state-selection, dummy derivaties and residuals)*

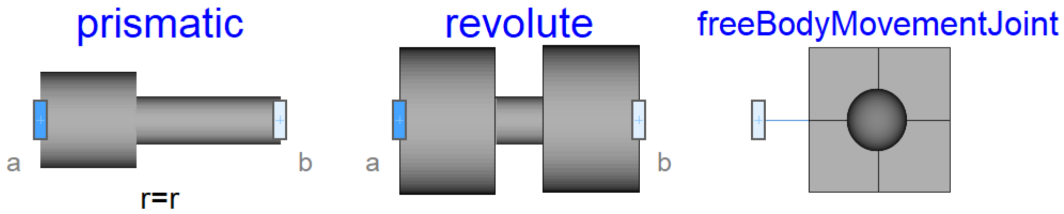
Observation regarding State Selection

- State Selection in TFS components:



$\dot{m}$ $\quad$ $\dot{m}$ $\quad$ $\dot{m}, \mathbf{\Theta}$

- State Selection in Dialectic Mechanics:



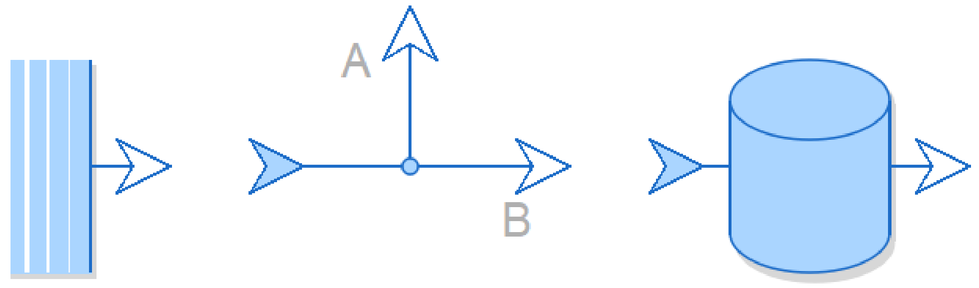
s, v $\quad$ φ, ω $\quad$ $\{x, y, \varphi\}, \{v_x, v_y, \omega\}$

- Mass-flows are selected for the source of any new branch.

- Each joint defines the position and (kinetic) velocity as state variables of the system

Observation regarding the Linear Equations

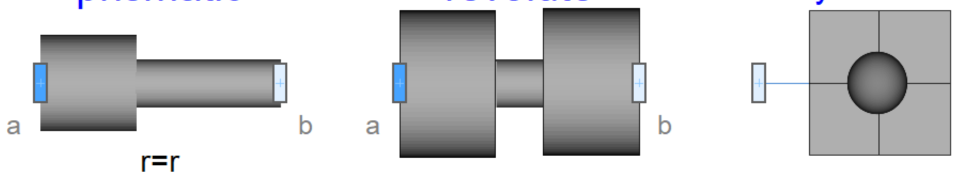
- Linear System in TFS components:



$$\left(\frac{d\dot{m}}{dt}, \Delta r\right)$$

- The derivative of each mass selected mass flow is a good tearing variable. The difference in inertial pressure is a good residual.

- State Selection in Dialectic Mechanics:



$$\left(\frac{ds}{dt}, f\right) \qquad \left(\frac{d\varphi}{dt}, \tau\right) \qquad \left(\frac{dx}{dt}, f_x\right), \left(\frac{dy}{dt}, f_y\right), \left(\frac{d\varphi}{dt}, \tau\right)$$

- The derivative of each selected positional state is a good tearing variable the collective force on the joint is a good residual.

Consequences for code generation

With this knowledge we can basically pre-compile each component:

- we stipulate the states
- we stipulate the tearing variables of the linear system and the corresponding residuals
- we perform the dummy derivative method on those equations where necessary.
- we define the causality of the interface variables
- we causalize all equations into assignments in a particular order
- we group the list of assignments depending on their dependence of the inputs.

Example: Main computational code

- A component can be pre-compiled into seperate blocks:

```
model PressureDrop
  TFSPlug inlet;
  TFSPlug outlet;
  parameter VolumeFlowRate v_ref;
  parameter Pressure dp_ref;
  VolumeFlowRate v_norm;
  SI.Pressure dp;
  SI.MassFlowRate m;
equation
  v=inlet.m.flow/rho(inlet.state);
  v_norm = v/v_ref;
  dp*2 = dp_ref*(v_norm+v_norm^2);
  inlet.m + outlet.m = 0;
  v = inlet.v;
  inlet.p - dp = outlet.p;

end PipeFrictionNL;
```

```
void PressureDrop::evalState() {
  const double v = inlet.m.flow/rho(inlet.state);
  const double v_norm = v/v_ref;
  const double dp = 0.5*dp_ref*(v_norm + v_norm*v_norm);
  outlet.state.h = inlet.state.h;
  outlet.state.p = inlet.state.p - dp;
}
```

```
void PressureDrop::evalFlow() {outlet.m = -inlet.m;}
```

```
void PressureDrop::evalInertial() {
  inlet.inertial.r = outlet.inertial.r
                     + L*inlet.m.flow_der;
}
```

Example: Meta Information

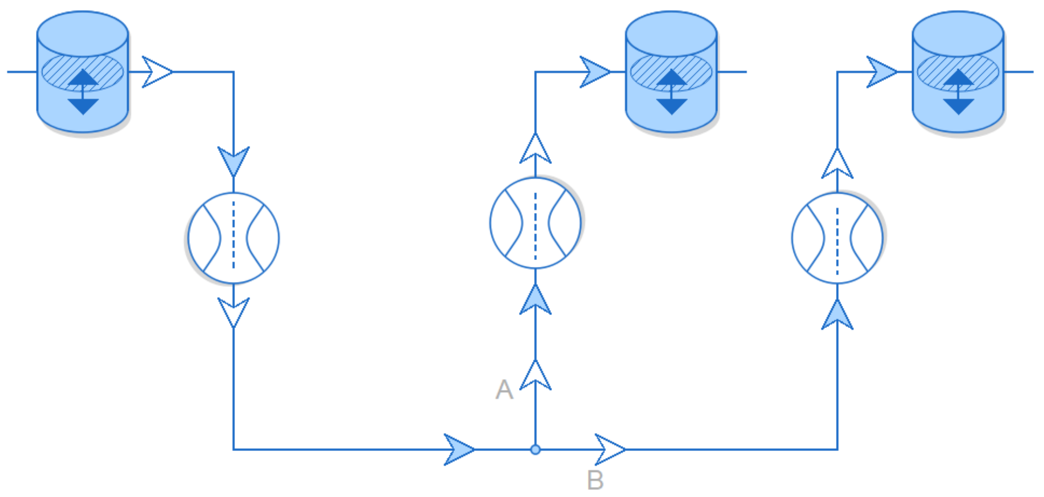
- For collecting, sorting and pruning, meta information is needed

```
model PressureDrop
  TFSPlug inlet;
  TFSPlug outlet;
  parameter VolumeFlowRate v_ref;
  parameter Pressure dp_ref;
  VolumeFlowRate v_norm;
  SI.Pressure dp;
  SI.MassFlowRate m;
equation
  v=inlet.m.flow/rho(inlet.state);
  v_norm = v/v_ref;
  dp*2 = dp_ref*(v_norm+v_norm^2);
  inlet.m + outlet.m = 0;
  v = inlet.v;
  inlet.p - dp = outlet.p;

end PipeFrictionNL;
```

```
void PressureDrop::metainfo(Meta& meta)
{
  meta.regComp (&inlet, “inlet”);
  meta.regComp (&inlet, “outlet”);
  meta.addBlock(this,
     LambdaFuncCalling(this->evalState()),
     Signals{&inlet.state,&inlet.m},
     Signals{&outlet.state});
  meta.addBlock(this,
    LambdaFuncCalling(this->evalFlow()),
    Signals{&inlet.m},
    Signals{&outlet.m});
  meta.addBlock(this,
    LambdaFuncCalling(this->evalInertial),
    Signals{&outlet.inertial,&inlet.m},
    Signals{&inlet.inertial});
}
```

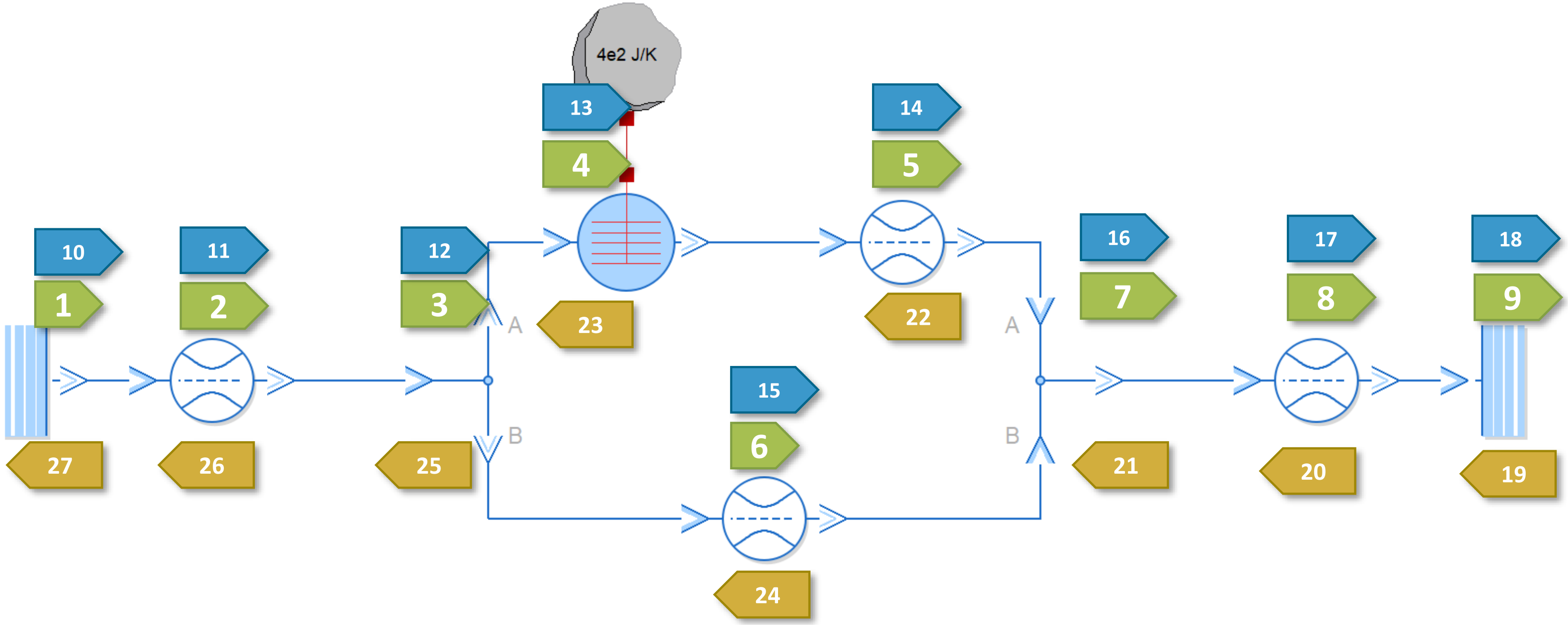
Example: Whole System in C++

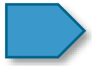
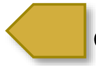
```
class ComVessels : public Component {
public:
  OutTank t1{};
  InTank t2{};
  InTank t3{};
  Splitter s{};
  PressureDrop p1{};
  PressureDrop p2{};
  PressureDrop p3{};

  Connections con {
    Connection{&t1.outlet, &p1.inlet},
    Connection{&p1.outlet, &s.inlet},
    Connection{&s.outlet1, &p2.inlet},
    Connection{&p2.outlet, &t2.inlet},
    Connection{&s.outlet2, &p3.inlet},
    Connection{&p3.inlet, &t3.inlet},
  };
...
```



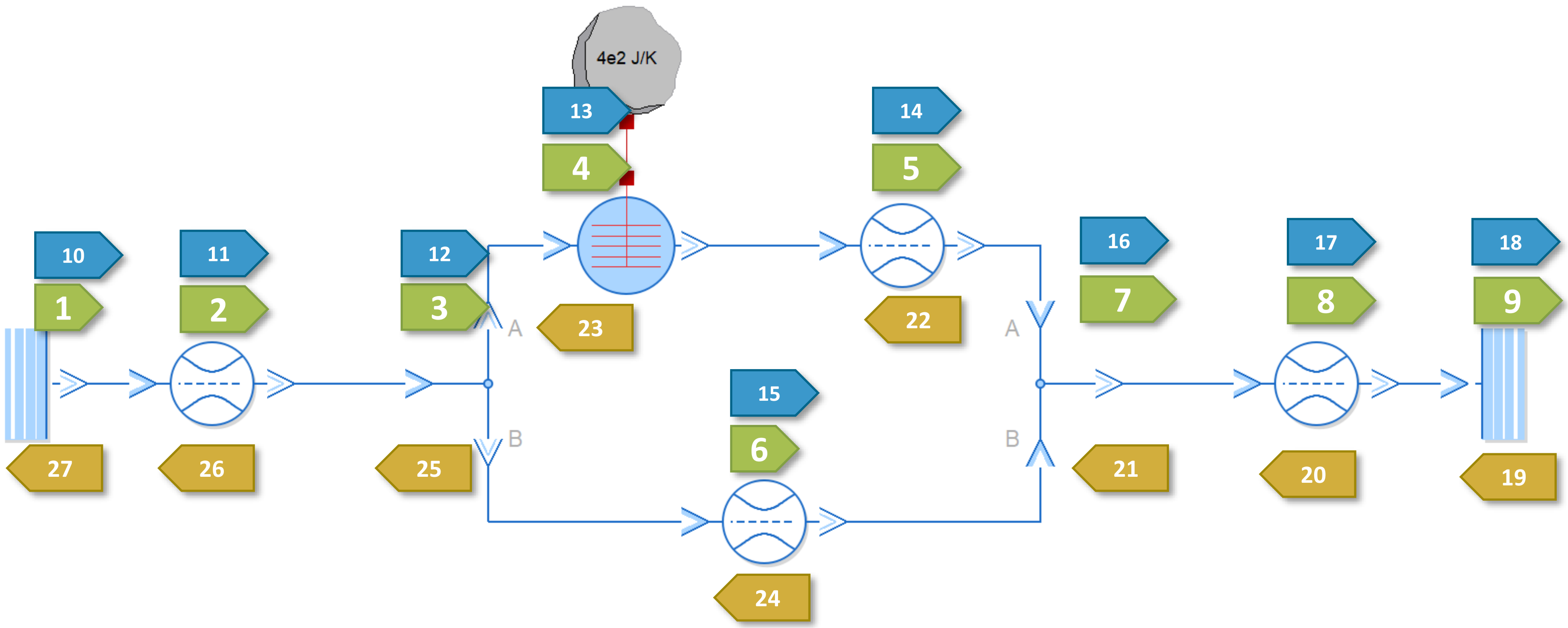

```
…
virtual void metainfo(Meta& meta) override{
    meta.regComp(&t1, "t1: first vessel");
    meta.regComp(&t2, "t2: second vessel");
    meta.regComp(&s,  "s: flow split");
    meta.regComp(&t3, "t3: third vessel");
    meta.regComp(&p1, "p1: first valve");
    meta.regComp(&p2, "p2: 2nd valve");
    meta.regComp(&p3, "p3: third valve");
   };
};
```

Example: Sorting



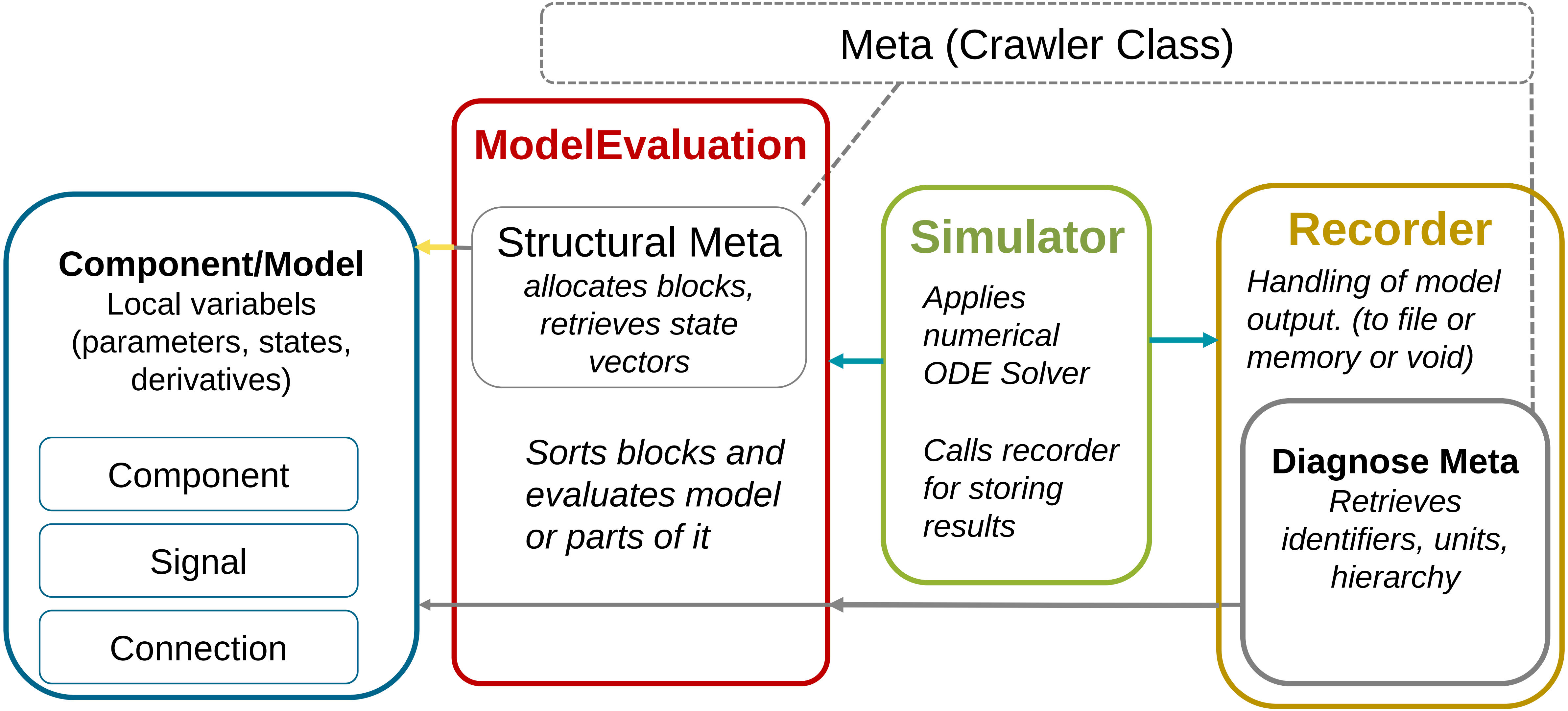
- For LIED systems, the code blocks would then be sorted. This can be done at run-time.
  - In our example:
  -  `evalState()` and
     `evalFlow()` are both sorted downstream
  -  `evalInertia()` is sorted upstream.

Example: Pruning

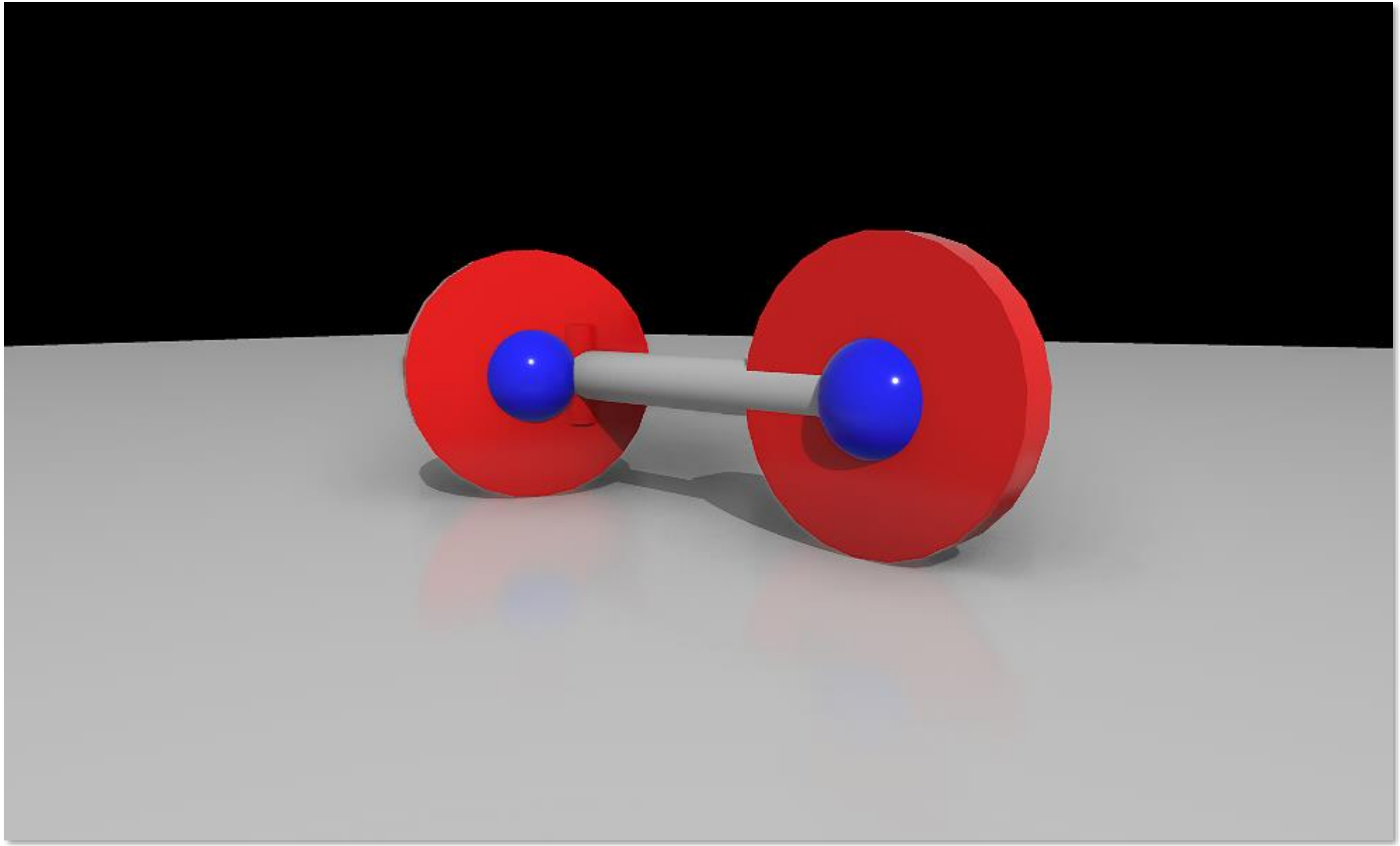


- A full model evaluation is not always needed. For the solution of the linear system only a partial evaluation is needed.

Overview of the object-oriented simulator code



Demo



From Modelica to Compiled Components in C++

- For the compilation of components, a causal interface needs to be provided.
- The causal interface requires the presence of additional derivatives
- The signal types can be classified as:
  - State-signal
  - Tearing signal
  - Residual signal
  - Other causal signals.
- The tearing signal determines yields a linear response on the residual signal and determines the derivative of the state signal
- Everything else is purely causal signal-based.

Example Rotational Dialectic Mechanics

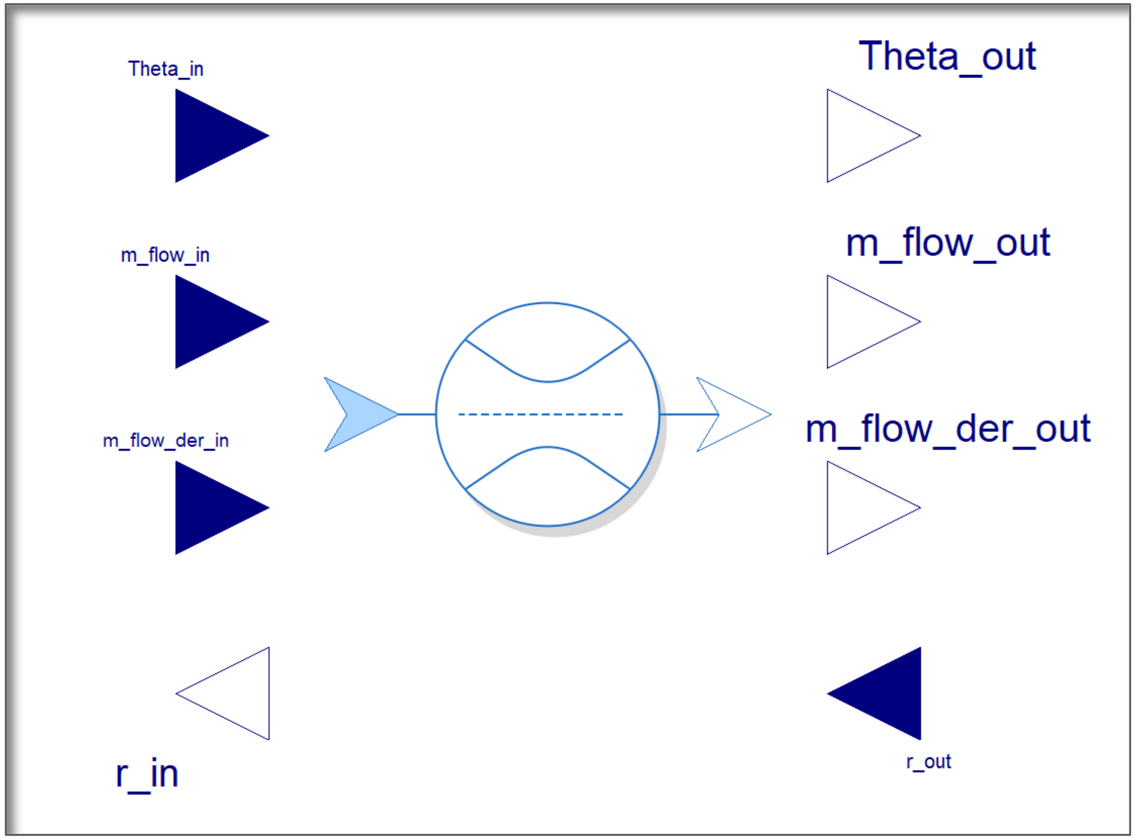
Modelica	**Causal Interface** (also inverted)	**C++** (also inverted)
φ (output)	φ (output: state signal)	`FlangeOn.state.phi`
ω (potential)	ω (output: state signal)	`FlangeOn.state.omega`
	$\dot{\varphi}$ (output: tearing signal)	`FlangeOn.tear.phi_der`
	α (output: tearing signal)	`FlangeOn.tear.alpha`
τ (flow)	τ (input: residual signal)	`FlangeOn.impulse.tau`

Example ThermoFluid Stream (unidirectional)

Modelica	**Causal Interface** (also inverted)	**C++** (also inverted)
θ (output)	θ (output: other signal)	`outlet.state`
$\dot{m}$ (flow)	$\dot{m}$ (output: state signal)	`outlet.m.flow`
$\xrightarrow{\frac{d}{dt}}$	$\frac{d\dot{m}}{dt}$ (output: tearing signal)	`outlet.m.flow_der`
r (potential)	r (input: residual signal)	`outlet.inertial.r`

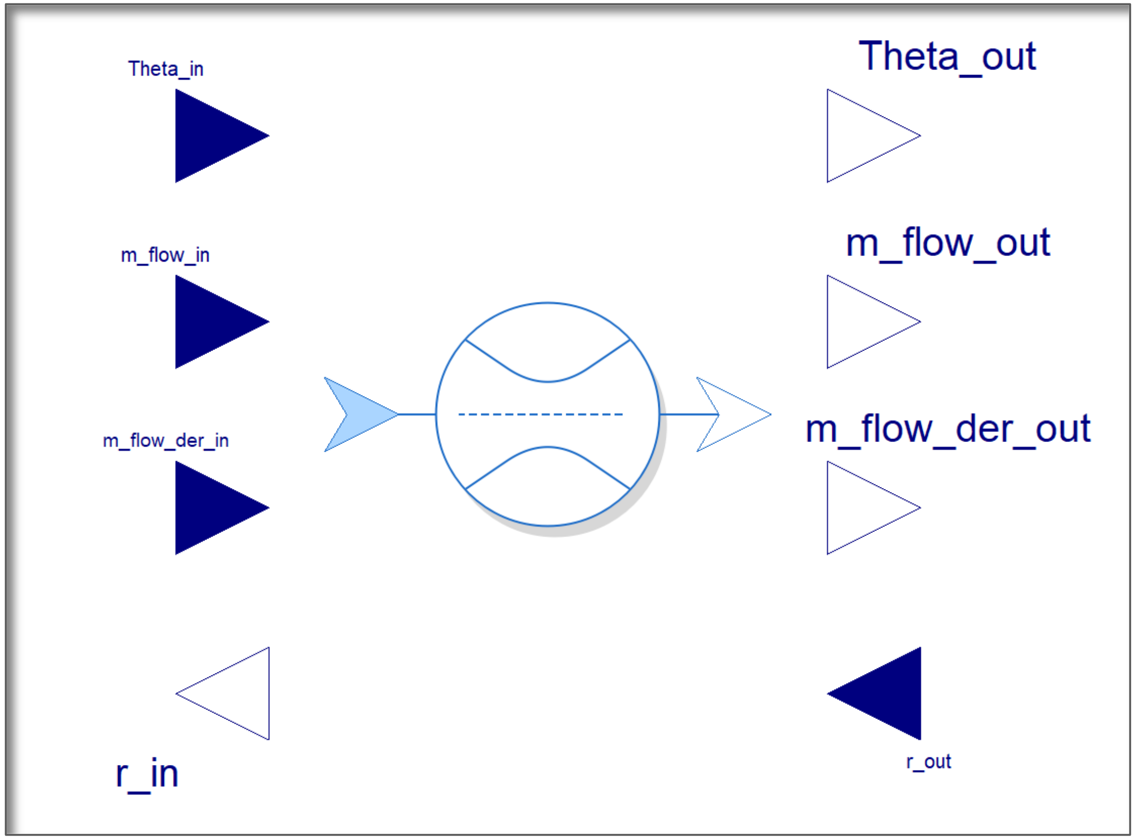
From Modelica to Compiled Components in C++

- Using this interface, we can then wrap LIED components
- Now compilation can be performed on the instance of this class, like with a FMU
- Similar to an FMU, we can enable the modification of non-structural parameters in the compiled version.
- The compiler should have all information that it needs from the interface wrapping.



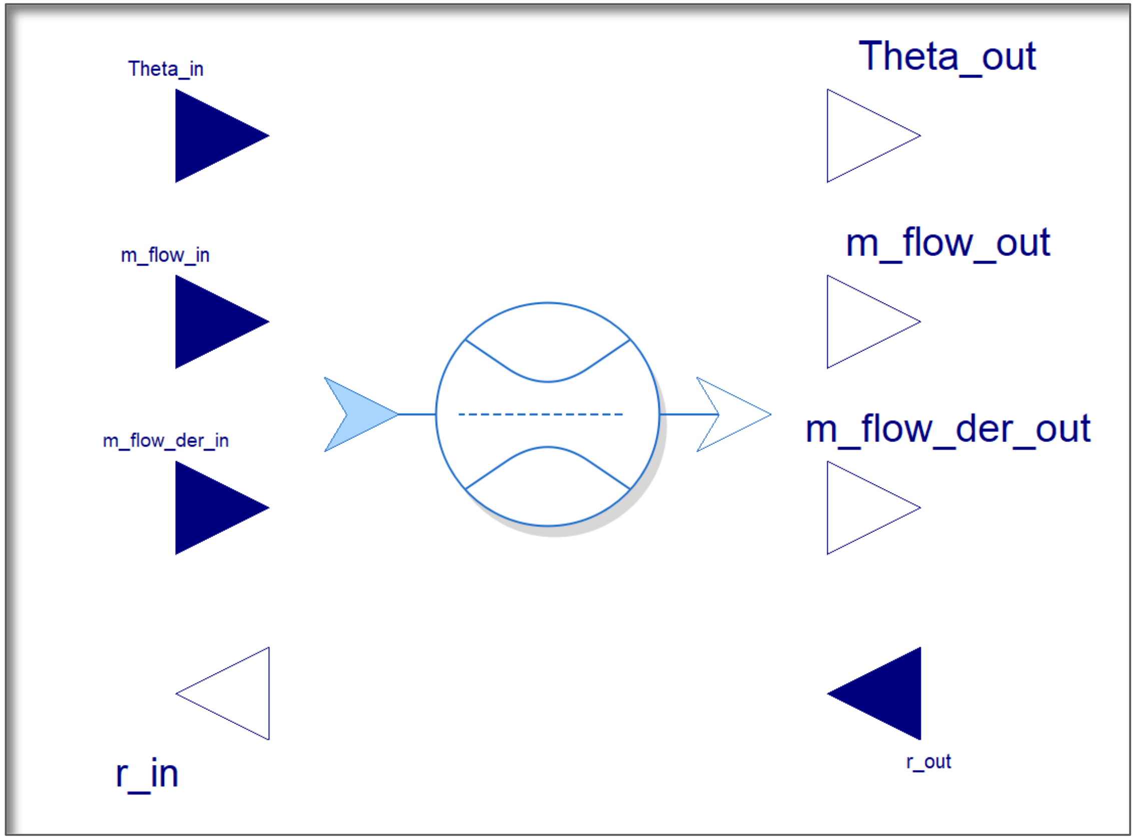
From Modelica to Compiled Components in C++

```
equation

  flowRes.inlet.Theta = Theta_in;
  flowRes.inlet.m_flow = m_flow;
  der(flowRes.inlet.m_flow) = m_flow_der;
  flowRes.inlet.r = r_in;


  flowRes.outlet.Theta = Theta_out;
  flowRes.outlet.m_flow = m_flow;
  der(flowRes.outlet.m_flow) = m_flow_out;
  flowRes.outlet.r = r_out;
```



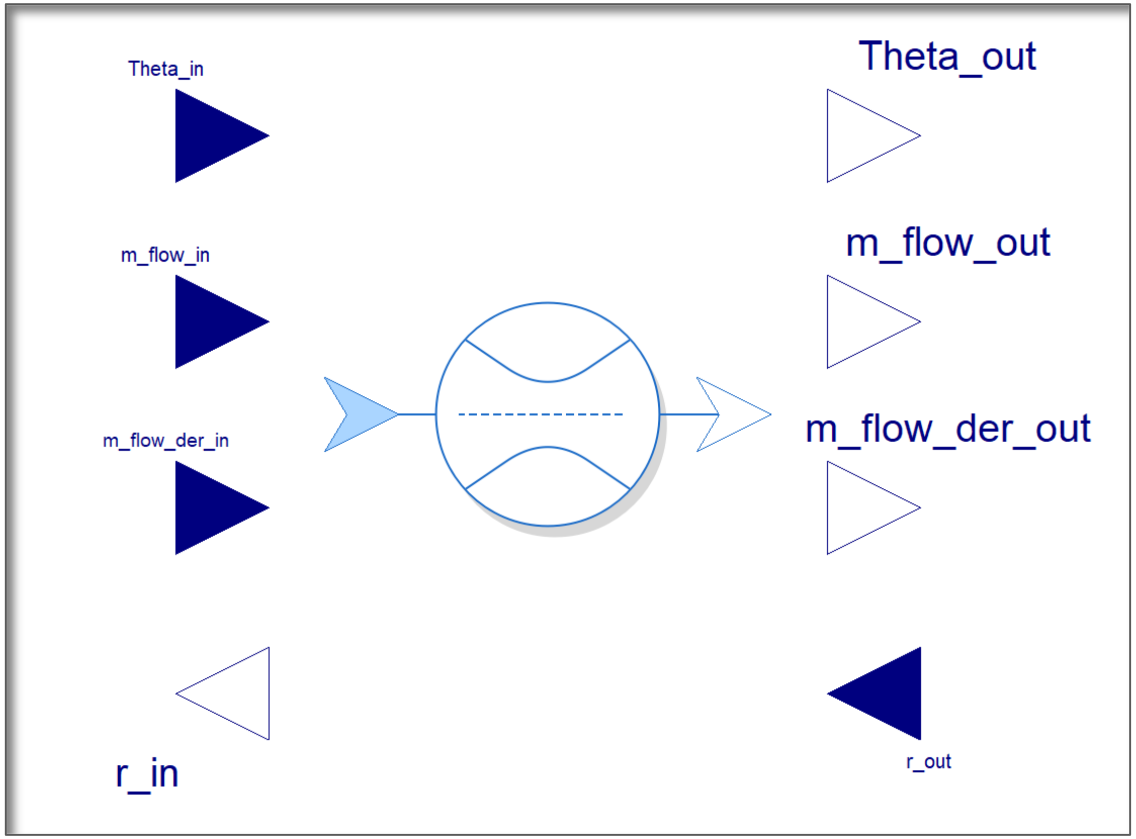
From Modelica to Compiled Components in C++

```
equation

   flowRes.inlet.Theta = Theta_in;
   flowRes.inlet.m_flow = m_flow;
   der(flowRes.inlet.m_flow) = m_flow_der;
   flowRes.inlet.r = r_in;


   flowRes.outlet.Theta = Theta_out;
   flowRes.outlet.m_flow = m_flow;
   der(flowRes.outlet.m_flow) = m_flow_out;
   flowRes.outlet.r = r_out;
```



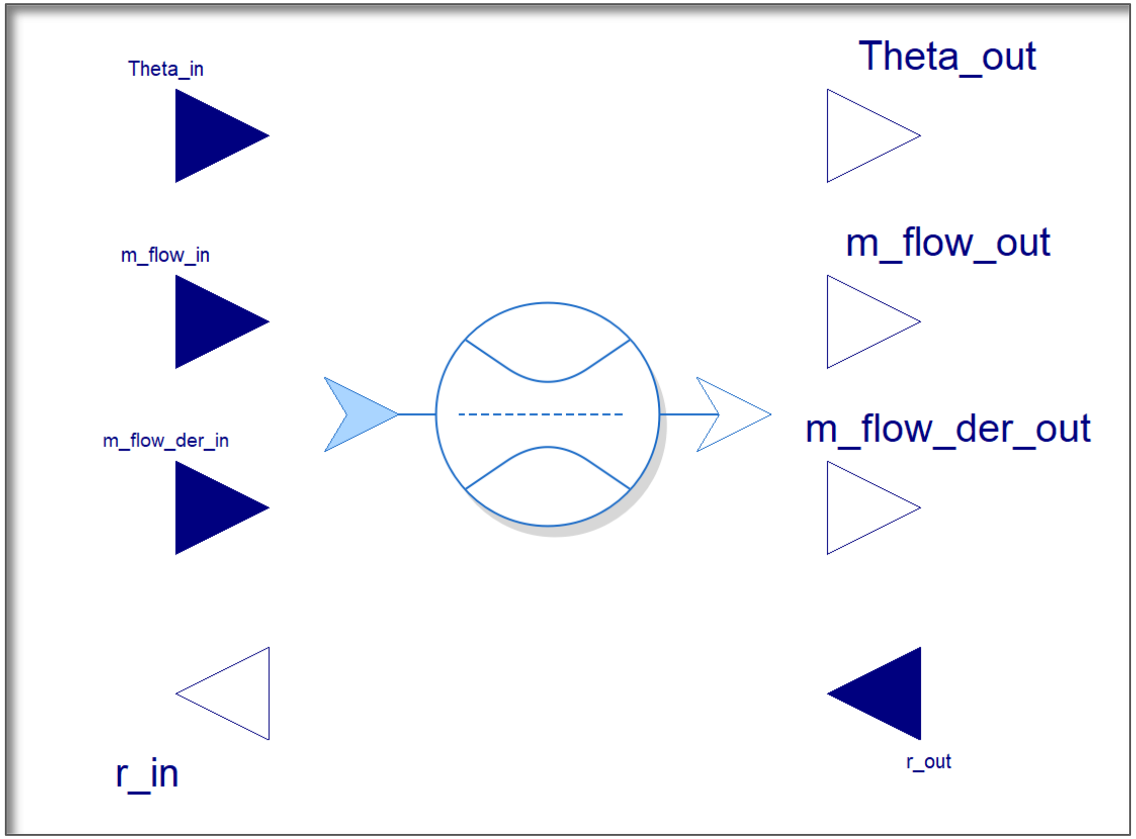
From Modelica to Compiled Components in C++

```
input ThermalState Theta_in
  annotation (LIED(
    signal=StdSignal,
    CPPref="inlet.state")
  );
input MassFlowRate m_flow_in
  annotation (LIED(
    signal=StateSignal,
    CPPRef="inlet.m.flow")
  );
input MassFLowAcc m_flow_der_in
  annotation (LIED(
    signal=TearingSignal,
    CPPRef="inlet.m.flow_der")
  );
output Pressure r
  annotation (LIED(
    signal=ResidualSignal,
    CPPRef="inlet.inertial.r")
  );
```



From Modelica to Compiled Components in C++

- The nice thing about this approach would be that it leaves the original Modelica component completely untouched.



Next Development / Research Steps

Goal for this summer:

- LIED Mechanical library in C++
- LIED Mechanical library in Modelica
- Causal interface in Modelica
- This will specify compilation source and compilation target.

Current Development on the C++ simulator:

- Implement better Diagnosis and Output
- Setup Regression Testing and Continuous Integration
- Implement pruning and measure scaling
- Make open-source and provide corresponding Modelica examples for compilation.

A More General Remark on LIED Systems.

It may be very tempting to allow for non-linear equation in implicit form…

…but it triggers lots of complexity down the line

- **Structural Uncertainty:**
  - States cannot be selected on component level
  - States may have to be determined at run-time
  - Tearing is non-deterministic
  - Structural changes very difficult
- **Flattening does not scale for large models**
  - Slow compilation, complicated algorithms
- **Algebraic limitations**
  - Unable to compute higher derivatives for multi-derivative methods
  - Unable to compute partial derivatives