Object-Oriented Formulation and Simulation of Models using Linear Implicit Equilibrium Dynamics

Dirk Zimmer

Institute of System Dynamics and Control,
German Aerospace Center (DLR),
dirk.zimmer@dlr.de

Abstract

New robust and yet powerful Modelica libraries have been developed such as the DLR ThermoFluid Stream library or the introduction of the Dialectic Mechanics library. These libraries apply a special modeling approach that uses linear implicit equilibrium dynamics. In this paper, we study the basic motivation of this approach, its benefits and drawbacks before we finally demonstrate how to get from models to applicable simulation code.

Keywords: Object-oriented modeling, Code Generation, Modeling principles

1 Introductory Example

Classic continuous laws of physics can be interpreted as communicating by means of waves. When you read these lines, your eye's photon receptors measure the electromagnetic waves communicating the corresponding visual information. When we speak, pneumatic waves communicate our audible voices. In a mechanics, pressure waves distribute the impulse in seemingly rigid bodies.

Even for things that we consider not to be alive, this analogy may be applied. A famous example is called: "communicating vessels" (in German: "kommunizierende Röhren") where various vessels filled with a homogenous liquid (let us use water) agree on a common surface level. This agreement is reached by hydraulic pressure waves going through the pipes, finally establish the hydrostatic equilibrium.

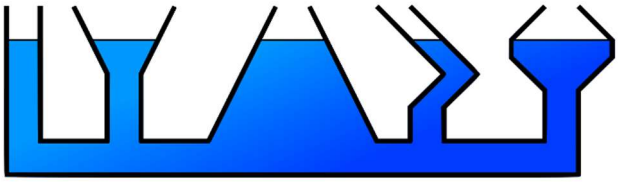


Figure 1: Depiction of communicating vessels displaying the hydro-static equilibrium. Public domain from Wikipedia.

Evidently, the macroscopic motion of a system can be interpreted as the emerging behavior of wave functions agreeing on an (quasi-) equilibrium state. One straight-forward way to model and simulate classic physics is thus simply to implement the corresponding wave equations directly using a spatial discretization scheme.

1.1 Example: Communicating Vessels

We can implement this in Modelica for the example of the communicating vessels by using a staggered grid, where the inertia and compression of the fluid is alternately placed such as in the lower half of Figure 2.

When we model the wave equation in an object-oriented way, we need an interface to connect the distributed elements. Since a wave can be interpreted as the rotation within two dimensions as in Figure 3, it is a natural choice to choose two variables on the two corresponding orthogonal axes. Each of these variables thereby indicates a different form of energy storage.

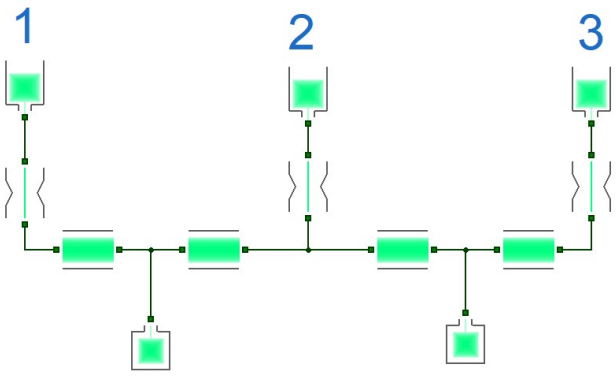


Figure 2: A model of 3 communicating vessels using a simple hydraulics library. Different from the depiction in Figure 1, the speed of flow is modulated by three narrow orifices at each tank. The one-dimensional hydraulic wave is modeled using a staggered grid for discretization. From top to bottom the layered icons represent the following elements: open-tank, non-linear pressure drop, fluid inertia, fluid compressibility.

In our hydraulic example, these two axes are: pressure p and volume flow $\dot{V}$. The pressure represents the potential energy of the compressed element whereas the volume flow rate represents the kinetic energy. We may call one of them a potential variable and the other one a flow variable. Since we work with an Eulerian framework, choosing the volume flow as flow is the natural choice.

Figure 4 shows the simulation result corresponding to our example. We can see that at the end of the simulation, we reach the desired equilibrium point. However, the computational efficiency is abysmal if this point is the

only result we are interested in. The pressure waves have a very high frequency (artificially lowered here) and so the simulation had to take many, very small time-steps.

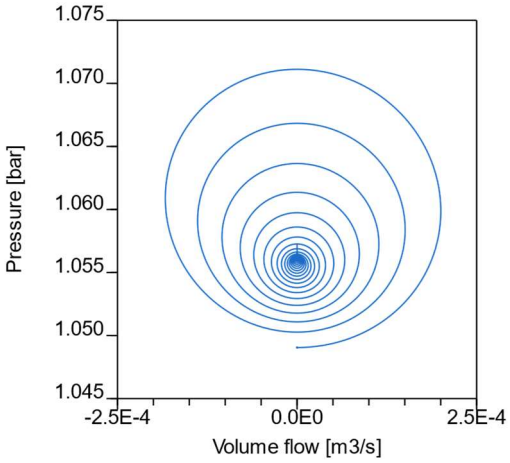


Figure 3: Trajectory of the pressure wave for the compressible volume in the two dimensions spanned by pressure and volume flow rate

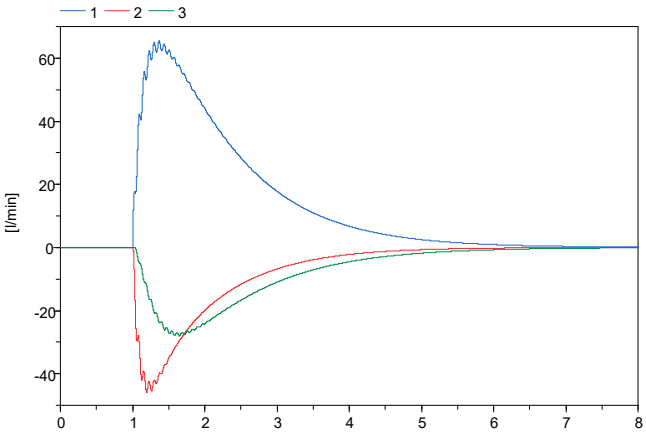


Figure 4: Step response of the communicating vessels after height of vessel 1 being increased at time = 1, showing the volume-flow through the valve openings. For the sake of illustration, the compressibility of water has been divided by 1000(!). Using the actual values, frequency would be much higher and ripples on volume flow barely visible.

Fortunately, we can avoid having to deal with high frequencies if we reduce the wave to its role as a conveyor of energy. The energy contained in our linear hydraulic wave is

$$E = \frac{1}{2}\rho_{lin} A^2 \omega^2 c$$

where ρ_{lin} is the linear density, A the amplitude, ω the frequency and c the speed of sound. If the macroscopic phenomenon of interest is orders of magnitudes larger than the amplitude and slower than the frequency, we can presume the wave to be an instantaneous transmitter of energy that simply has to uphold the conservation of energy (given that also the speed of sound is quick enough over the required distance).

This transfer of power can be modelled by the same pair of variables that we have used to describe the wave equation. In our case the product of the pair represents a flow of energy:

$$\dot{E} = p\dot{V}$$

The power produced by a component with two such pairs is thus:

$$P = p_1\dot{V}_1 + p_2\dot{V}_2$$

If we are simply interested in the exchange of potential gravitational energy between the vessels over dissipative valve openings, we can choose to ignore the modeling of the hydraulic wave completely and simply connect the elements directly as in Figure 5.

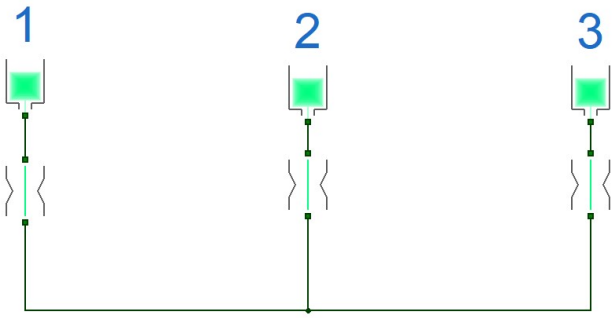


Figure 5: Modelling the communicating vessels by the sheer exchange of potential energy

By doing so, we have created an implicit non-linear algebraic equation system: the pressure level below the valves has to be found so that the corresponding volume flows resulting from the pressure drop are in balance. In our case, this system can be reliably solved, even for the case of the step response as depicted in Figure 6.

In general, we may have more than one solution or none at all. Also, the equation system is only available in implicit form. We thus replace the physical method to compute the transfer of energy with the solution for an algebraic system. Whether this works or not is simply down to luck in the general case. Here we were lucky.

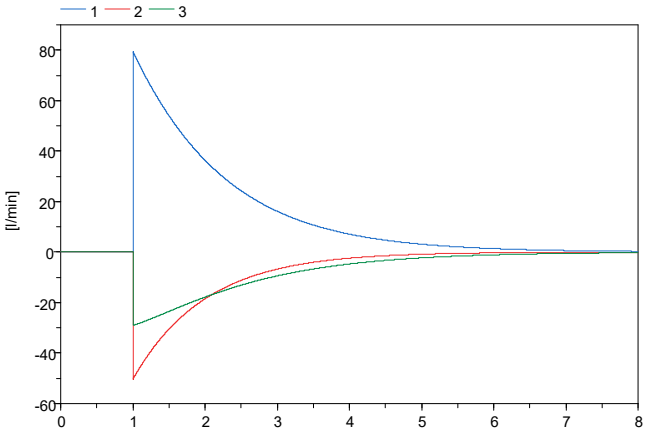


Figure 6: Corresponding step response of the direct model

DOI
10.3384/ecp204265

1.2 Comparing the Modeling Approaches

We have created our first model using *explicit wave dynamics*.

We have created our second model using *non-linear implicit power dynamics*

Generating code for explicit wave dynamics is rather easy. All the equations are in explicit form and can be directly written as an ODE. Setting up the simulation code is thus principally rather trivial.

Simulating explicit wave dynamics is often computationally very expensive. Worse than the potentially high number of state variables is the that the frequency of the wave dynamics is often several orders of magnitude higher than the frequency of the macro-phenomenon of interest.

Simulating non-linear implicit power dynamics is much more efficient. Assuming an instantaneous transfer of power enables us to ignore the high frequency and phase shifts of the wave and we only have to deal with the low frequency of the macro-phenomenon. Also, we may use significantly fewer states.

Generating code for implicit power dynamics however, is far from trivial. Our system above had permutation index 1, because it requires the solution of a non-linear equation system. In mechanical systems, higher-index systems are common that require a reduction of the differential index for instance by applying Pantelides (Pantelides 1988). Because the simulation code is of high algorithmic complexity, we like to have a Modelica compiler creating it for us.

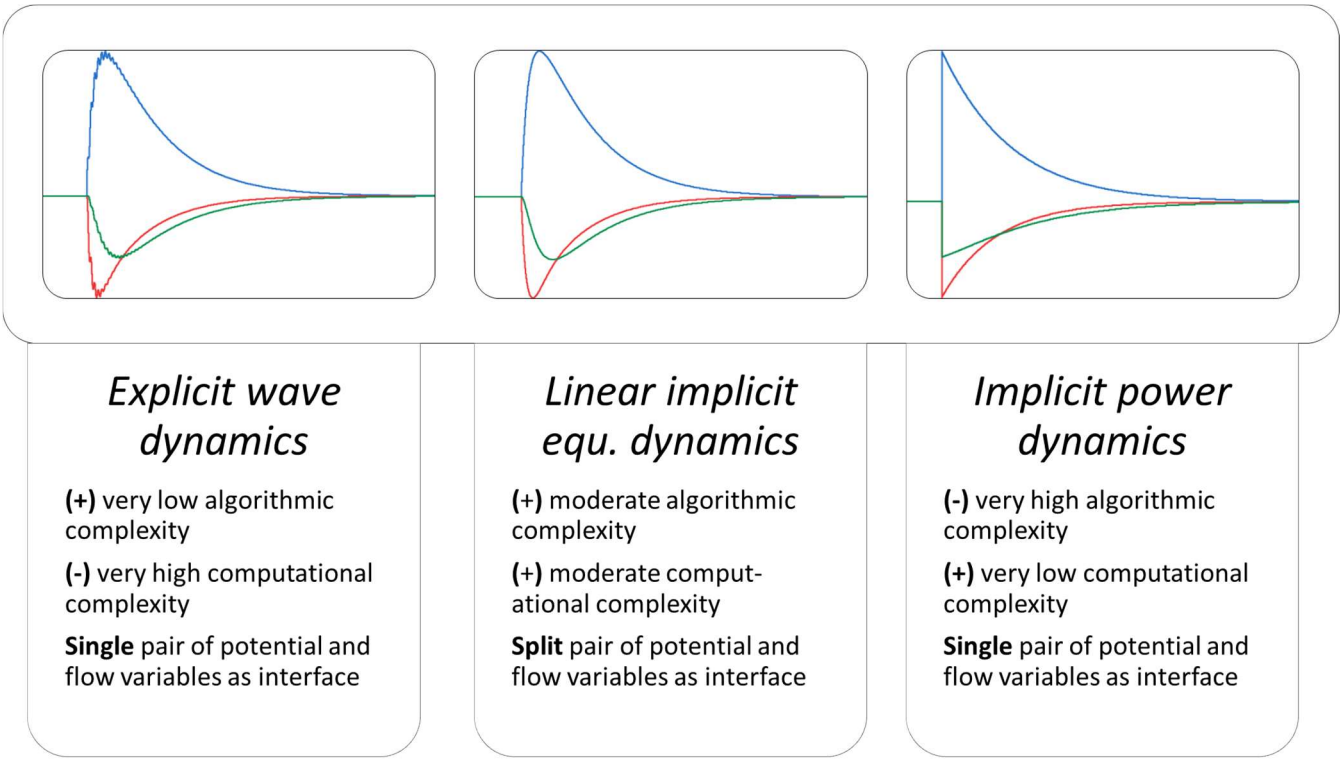
Choosing between these two can thus be seen as a trade-off between computational complexity (time needed for simulation) and algorithmic complexity (length of program for model generation) of the simulation code. This comparison is also highlighted in Figure 7 where wave dynamics is on the left and power dynamics is on the right.

For both forms of complexities, it is in practice nearly impossible to determine their theoretical limits. Since the computational complexity includes the ODE solver, we would need to determine the solver that reaches the desired precision within the shortest amount of time. The algorithmic complexity is to be interpreted in terms of algorithmic information theory (Chaitin 1987) and we would need to find the shortest possible program for code generation. In practice, it is however feasible to work with the numbers at hand: measuring the code-length of the compiler and measuring the time the simulation took. For our considerations, the general concepts suffice.

Explicit wave dynamics is computationally complex but can be low in algorithmic complexity. Non-linear implicit power dynamics is often of high algorithmic complexity but lower in computational complexity.

However, Figure 7 also shows that there seems to be an interesting middle ground in between these two classes of models, that might offer a very favorable trade-off. I denote this class: Linear Implicit Equilibrium Dynamics (LIED) (you can keep the pun). This class is typically not found in classic text-books and I presume that the primary reason for this is simply that we have to use an extended interface, which is not intuitive to come up with in the first place.

Figure 7: Illustration of different modeling approaches and their impact on algorithmic and computational complexity



2 The Idea behind Linear Implicit Equilibrium Dynamics

Figure 9 illustrates the desired result. To a step change we react neither with a high-frequency wave function nor with a discrete jump but by approaching the desired equilibrium with replacement dynamics. These dynamics shall reach the same steady-state behavior than the original wave dynamics and exhibit only a limited deviation for slow-mode behavior. Any deviation shall be of dissipative nature in case energy conservation cannot be upheld. There is one additional catch though: we shall limit our equations which are in implicit form to constitute a purely linear system.

The motivation for restricting ourselves to linearity for the implicit part is, to enable a robust solution of the system at all time, something that cannot be guaranteed for non-linear systems in general.

To put these statements in formal terms: if a system is described by differential algebraic equations (DAEs) in the following implicit form:

$$\mathbf{0} = F(\mathbf{x}_P, \dot{\mathbf{x}}_P, \mathbf{u}, t)$$

where $\boldsymbol{x}_P$ is the vector of potential states, $\dot{\boldsymbol{x}}_P$ represents all time derivatives, $\boldsymbol{u}$ the input vector and t time.

We aim to transform this system into the following form with an implicit linear part and an explicit non-linear part:

$$\mathbf{L}\dot{\mathbf{x}}_L = g(\mathbf{x}_L, \mathbf{x}_N, \mathbf{u}, t)$$

$$\dot{\mathbf{x}}_N = f(\mathbf{x}_L, \mathbf{x}_N, \mathbf{u}, t)$$

where $\mathbf{x}_L$ and $\mathbf{x}_N$ are both disjoint sub-vectors of $\mathbf{x}_P$. $\mathbf{L}$ is a linear matrix and f and g are non-linear functions. The original DAE system F is defined as a LIED system if and only if the functions f and g can be constructed just by ordering the corresponding equations of F.

Techniques for symbolical reduction of the differential index (Leimkuhler 1985) or the permutation index (Campbell 1995) may hence only be applied to derive the matrix $\mathbf{L}$. Hence, all non-linearities have to be brought into an explicit form and placed in either f or g. Equations in implicit form (including constraints between potential states) have to be linear and to be placed in $\mathbf{L}$.

How can we construct such DAEs for classic physical systems? And how to do this in an object-oriented form? The basic idea is simple: we find a part in the transient dynamics that is suitable for linear approximation and that completely vanishes at steady-state. A suitable candidate is often the dynamics of kinetic energy since it has a linear characteristic for a wide range of systems.

To enable this extraction, we have to split our interface, especially suited are variables that contain the flow of impulse (force, pressure, etc.) because here we can apply the superposition principle. Otherwise it may be very hard to separate the linear part in implicit from the non-linear part in explicit form.

All of the above is much easier said than done. I have spent several months figuring it out for thermo-fluid domain and later for the mechanical domain. Refinement took years for thermo-fluids and is still in the process for mechanics. The good news is: once we have identified a suitable interface, the remaining part of implementation is straightforward, often even easy.

2.1 LIED for Thermofluid Systems

Here is the full interface for thermo-fluid streams:

- r: inertial pressure (potential)
- $\dot{m}$: mass-flow rate (flow)
- Θ: Vector repr. state of medium (signal)
  - $\hat{p}$: steady-mass flow pressure
  - $\hat{h}$: steady-mass flow enthalpy
  - X: mass fractions

For the thermo-fluid streams, we have to split the potential variable into two parts: The steady-state pressure $\hat{p}$ and the inertial pressure r. The dynamics for the inertial pressure can be described by an implicit linear system using the law of inertance using the fluids inertia L:

$$r = L\frac{d\dot{m}}{dt}$$

For the steady-state with a constant mass flow rate, r will thus go to zero. To enable the approximation during transients, the impact of r on the thermodynamic state has to be neglected and hence Θ is composed using $\hat{p}$.

When the interface is used correctly, the whole thermofluid system will be a LIED system. $\mathbf{x}_L$ will form a vector that describes all mass-flow rates of the system in non-redundant manner. Typically, the dummy derivatives method (Mattsson 1993. Pantelides 1988) needs to be applied to construct the Matrix $\mathbf{L}$. Its coefficients are then formed by linear combinations of the inertances. $\mathbf{x}_N$ will contain all other states (such as specific enthalpy, etc.). Using these states the functions f and g can be computed in a downstream manner. More details on this interface and the implementation of a full library can be found in (Zimmer 2020, 2022). Models using this interface are especially suitable for the simulation of complex thermal architectures with bypasses and switches even under hard real-time constraints.

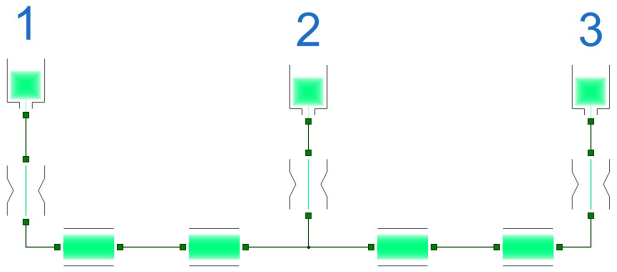


Figure 8: For this particular model of the communicating vessels, the LIED approach has an equivalent counterpart using conventional connectors. Modeling the inertia but leaving out the compressibility does the trick here.

DOI
10.3384/ecp204265

In the particular case of our example with the communicating vessels, the LIED approach is equivalent to using only inertias for the fluid but disregarding the compressibility. Figure 8 shows the equivalent model diagram and Figure 9 depicts the corresponding simulation results.

This simple equivalence does however only work in this example because we treat the water as having constant density and also neglect any influence of temperature. Hence in this example we can mimic the LIED approach using the basic connectors. Using more realistic media models, the ThermoFluid Stream approach works more subtly and the split interface is needed.

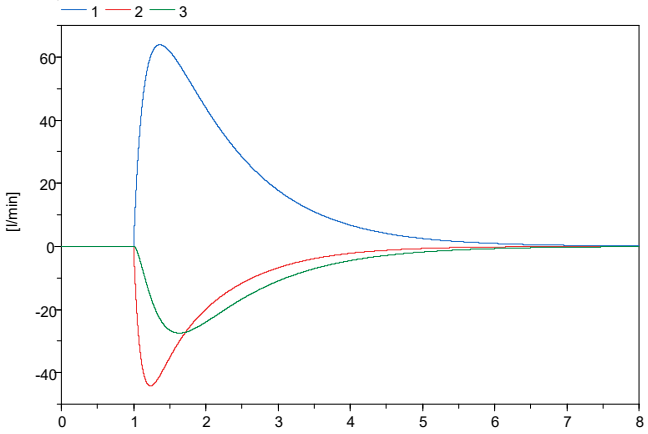


Figure 9: Modelling the communicating vessels by the sheer exchange of potential energy

2.2 LIED for Mechanical Systems

For mechanical systems, the interface is defined as follows:

- s: position (potential)
- f_{el}: elastic force (flow)
- v: velocity (potential)
- f_{ki}: kinetic force (flow)

We have thus two pairs of effort and flow not one. The derivative of the position s is thereby defined as v_{el}. The velocity v is also denoted as v_{ki}. The difference $\Delta v = v_{el} - v_{ki}$ should ideally be zero at all times. To enable a linear implicit approximation, we tolerate non-zero values for Δv at fast transients but establish a first order dynamics that ensures zero is approached for slow dynamics with the dialectic time constant T_D:

$$\frac{d\Delta v}{dt} T_D = -\Delta v$$

Because this interface separates the regimes of elastics and kinetics, I have denoted it as dialectic mechanics. First implementations and analysis are presented in (Zimmer 2023) and (Oldemeyer 2023). Models using this interface are especially suitable for the simulation of contacts and limited joints also under hard-real time constraints.

Dialectic mechanics are also LIED systems: the vector $\mathbf{x}_L$ will contain all the (generalized) positions in a non-redundant form so that all degrees of freedom are described. $\mathbf{x}_N$ then typically consists in the corresponding kinetic velocities. f and g can then be computed from the mechanical root of the system to the branches. Kinematic loops are explicitly closed using elastic elements with high stiff springs which is the preferred way in dialectic mechanics since high frequencies can be suppressed.

The details of the domain specific implementation shall not be the topic of this paper. But evidently this class of models is very useful and hence we shall further investigate its implications for the generation of simulation code.

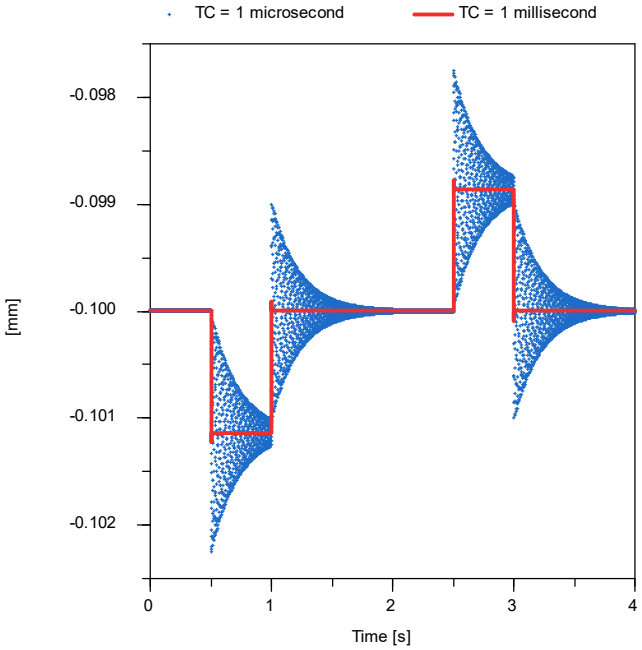


Figure 10: Penetration depth into the left claw represented by an elasto-gap, for the choice of two different time constants. Both agree on the time-averaged solution.

Just for the sake of quick illustration: Figure 10 from (Zimmer 2023) is repeated here again that shows the dynamics of a lightweight object moved in clamp modeled by a very stiff spring. The figure simply illustrates how the oscillatory dynamics is replaced with a replacement dynamics leading to the same (quasi) steady-state solution.

3 How to Create Simulation Code for LIED Systems?

The original intention of the LIED approach was simply to ensure that no non-linear system is created that spans across the components and hence a robust solution of the model evaluation could be taken for granted, given robust component models. When we started with it, we expected it to be the only notable change from other Modelica models and that all other features of a Modelica compiler (state selection, differential index-reduction, tearing of

linear equation systems, etc.) would basically remain untouched.

However, over time, we realized that LIED systems are much simpler to transfer to simulation code than general DAEs resulting from non-linear implicit power dynamics. Let us go through the observed simplifications one-by one:

- Because we avoid the creation of non-linear equation systems, we do not need a non-linear equation system solver anymore.
- For the same reason, constraint equations among potential states cannot be non-linear and hence no dynamic state selection is needed (Mattsson 2000).
- Even stronger: we can select the states on component level. This is less obvious but ultimately the connection rules that enforce the linearity of the system also enforce this rule.
- Because we can select the states on component level, this means that the dummy-derivative method can be applied also on component level before system composition.
- Since the goal of the linear equation system is to have a synchronized replacement dynamic towards the equilibrium, we know suitable tearing variables for this system. These will be the linear state derivatives: $\dot{\mathbf{x}}_L$ or at least a subset of it.
- The residual for a tearing variable can be attributed to the same component as the tearing variable.

The items above represent observations resulting from modeling many components and system examples using the LIED approach. However, these observations have profound implications: For each component we know:

- the set of pairs of state-variables and their derivatives it adds to the system.
- the set of pairs of tearing and residual variables it adds to the system.

If this is the case, we can basically causalize everything already on the component level. In concrete terms, this means for 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.

Practical experience so far indicates that performing index reduction to construct the matrix **L** can be performed in a very methodical and deterministic manner. It is thus far easier to generate simulation code for the LIED modeling approach than it would be for general higher-index DAEs. Neither there is a need for global flattening anymore nor are elaborate heuristics needed for the selection of state or tearing variables. Indeed, the generation of simulation code is so easy that a direct implementation in C++ becomes feasible. The following code excerpts illustrate the implementation for a ThermoFluidStream Library (using idealized water) in C++.

First, we have to define the interface. This is naturally more tedious than in Modelica because there is no direct support in the C++ language. Yet, it is feasible and after all, interfaces only need to be defined once:

Listing 1. ThermoFluid Interface in C++

```
class ThermodynamicStateOut: public Signal{
  public:
    double p;
    double h;
    […]
};

class ThermodynamicStateIn:
  public ThermodynamicStateOut
{
  public:
    void connect(ThermodynamicStateOut* o);
    […]
};

class MassFlowOut : public Signal{
  public:
    double flow;
    double flow_der;
    […]
};

class MassFlowIn : public MassFlowOut{
  public:
     void connect(MassFlowOut* o);
    […]
};

class InertialPressureOut : public Signal{
  public:
    double r;
    […]
};

class InertialPressureIn :
  public InertialPressureOut
{
  public:
    void connect(InertialPressureOut* o);
    […]
};

class ThermalPlugOut : public Signal{
   public:
     ThermodynamicStateOut state{};
     MassFlowOut m{};
     InertialPressureIn inertial{};
     […]
};
```


DOI
10.3384/ecp204265

```
class ThermalPlugIn : public Signal{
  public:
    ThermodynamicStateIn state{};
    MassFlowIn m{};
    InertialPressureOut inertial{};
    void connect(ThermalPlugOut* o);
    […]
};

class Connection {
  public:
    Connection(ThermalPlugOut* o,
               ThermalPlugIn* i) {
                i->connect(o);
     };
};
typedef std::vector<Connection>  Connections;
```

To best understand the interface, let us look at the classes `ThermalPlugOut` for a nominal outlet flow and at `ThermalPlugIn` for a nominal inlet flow first. These contain the same 3 components as the corresponding Modelica connector of the DLR ThermoFluid Stream library.

There are two notable differences however. In Modelica, inertial pressure and mass flow were not causalized signals as in the C++ implementation. Also the mass-flow signal in the C++ library consists of the mass-flow rate and its derivative. In Modelica, this is not necessary since symbolic differentiation can be applied by the Modelica compiler. Using this interface, we can now implement a component such as the pressure drop:

Listing 2. Implementation of a pressure drop component

```
class PressureDrop : public Component{
  public:
    ThermalPlugIn inlet;
    ThermalPlugOut outlet;
    PressureDrop(double v_ref,double dp_ref)
    void evalState();
    void evalFlow();
    void evalInertial();
    double v_ref;
    double dp_ref;

    virtual void metainfo(Meta& meta)
      override;
    […]
};
```

First, we declare our interface for outlet and inlet. Then we have to implement three blocks represented by methods. The first is `evalState` and computes the thermodynamic state downstream:

Listing 3. Calculation of the pressure drop by the corresponding method

```
void PressureDrop::evalState() {
   const double v =
     inlet.m.flow / density(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;
};
```

The second method is `evalFlow` to ensure what flows in is what flows out. However, this constraint is restated for the derivative. This is because the dummy derivative method is applied on the component level.

Listing 4. Trivial implementation of `evalFlow`

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

The third one is `evalInertia` that implements the law for the inertance as in the ThermoFluid Stream Library.

Listing 5. Calculation of the inertial pressure

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

Meta-information can be collected by a dedicated virtual method to register state and tearing variables as well as to track the signal dependence of the computing blocks.

Listing 6. The meta information of the component is described in a virtual method.

```
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});
}
```

For the pressure drop the signal dependencies of the methods have to be registered as vital structural information. Because of the horribly bad support of method function pointers in C++, the implementation requires the use of a lambda function which is done here in pseudo-code for the sake of readability.

In similar manner the other components of our introductory example can be implemented. Each of these components declares its interfaces, defines and implements methods representing the computational blocks and then registers these blocks as well as states, etc

by overriding the virtual `metainfo` method. It is not as convenient as Modelica but also not overburdening.

Finally, we can compose the introductory example:

Listing 7. Total system composition

```
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: second valve");
    meta.regComp(&p3, "p3: third valve");

  };
};
```

Regarding that C++ is a statically compiled imperative general-purpose language, the end result is astonishingly close to what we are used to from Modelica.

When an instance of the class is coupled to a simulator, a crawler for meta information collects all blocks recursively as well as the structural information regarding the signals, the states and the tearing variables for the linear equation system. Then the blocks are put into right order for complete or partial model evaluation. The full model evaluation is thus simply a list of method calls that are called in their respective order.

This also means that all component code is statically compiled but the system composition is performed at run-time. Advanced tasks such as variable structure systems would thus be comparably easy to achieve.

This is also the purpose of this code demonstration. It is not suggesting that we should use C++ instead of Modelica but to highlight that for a certain class of models the object-oriented Modelica code could be translated to object-oriented imperative code that can be statically compiled even before system composition.

This would avoid the flattening of all equations before code generation and help to overcome many limitations of current Modelica compilers and you can of course choose a different target than C++.

4 Conclusions

That a more restrictive class of modeling enables a simpler compilation scheme is not surprising. The same can be said about the many conventional signal-based modeling schemes or simple modeling schemes as Forrester's System Dynamics (Junglas 2016). Typically, the disadvantage is that the easier generation of simulation code has to be paid by an inferior modeling approach and indeed modeling complex mechanics or thermo-fluid streams is painful when using signal-based approaches (nevertheless this pain has been taken in industrial practice all too often).

The remarkable thing about the LIED approach is that you have a simple scheme for code generation but you can conveniently model both mechanics and thermo-fluid streams in a very robust manner. The corresponding Modelica Libraries prove this (Zimmer 2022, Zimmer 2023). Both application domains are known to be rather difficult but LIED can even be applied to the challenging parts of these fields such as handling stiff contact mechanics or complex by-passes in thermal architectures. It is yet unclear for what other domains LIED is an attractive choice.

Figure 11 attempts to qualitatively depict the trade-off between computational complexity and algorithmic complexity. LIED forms a very exposed point on a hypothetical Pareto front. This means that for a large number of applications it is a very attractive choice.

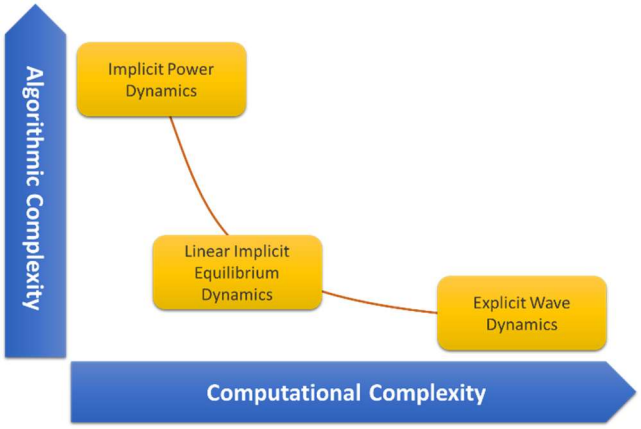


Figure 11: Hypothetical Pareto front weighing computational complexity against algorithmic complexity for code generation. LIED systems form an attractive compromise. The ultimate choice of the modeling approach depends however on the concrete application.

What does this mean for the Modelica community? We should recognize that fully within our standard, this particular class of LIED models has been hidden. Due to their non-conventional interfaces (that appear in no textbook), this class has been overlooked for more than 20 years. It is a robust class of models that scales for complex systems and also it is suitable for hard-real time simulation since everything non-linear is explicit and there are effective methods to manipulate fast eigendynamics.

DOI
10.3384/ecp204265

I think it is justified to give this class extra support, by enabling the following features:

- The modeler shall be enabled to mark components that are compatible to LIED, and provide additional meta-information to this end. The Modelica compiler can then check whether this is true.
- The Modelica compiler can then enable the component-wise compilation of such components, at least for explicit ODE solvers and for implicit solvers with the numerical computation of the Jacobian.

The primary motivation of this paper is to raise awareness on this class of models and the possibilities it enables for the generation of simulation code. The provided code examples are not necessarily the best approach and can be improved on. Furthermore, this modeling approach is still new and open for further investigation.

Many statements in this paper require further validation also the topic is not very tangible. Hence, I want to encourage the reader to play with the open-source library ThermoFluid Stream and study dialectic mechanics. The practical way is the best way to develop an understanding for this modeling style.

As a final remark, I shall say that I am very grateful for Modelica and its tool-set. I am not sure whether I would have ever found this particular class without it. Quick experimentation within Modelica was certainly extremely helpful.

References

Campbell, S.L., C. William (1995), The Index of General Nonlinear DAEs. In: Numerische Mathematik, Vol. 72, pp. 173–196

Chaitin, G.J. (1987). Algorithmic Information Theory. Cambridge University Press. ISBN 9780521343060

Junglas, P. (2016), Causality of System Dynamics Diagrams, SNE Simulation Notes Europe 26/3, 147-154

Leimkuhler, B., C.W. Gear, G.K. Gupta (1985), Automatic integration of Euler-Lagrange equations with constraints. In: J. Comp. Appl. Math., Vol. 12&13, pp. 77–90.

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

Sven Erik Mattsson, H. Olsson, H. Elmqvist (2000) “Dynamic Selection of States in Dymola”. Proceedings of Modelica Workshop 2000.

Oldemeyer, C., D. Zimmer (2023). “Dialectic Mechanics: Extension for Hard Real-time Simulation”. Proceedings of the 15th International Modelica Conference, Aachen.

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

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.