SINTEF

Simulator Design based on Variable Graph for Complex Physical Systems

Motivations

Code Reusability

- When two models have something in common, how can we reuse the code developed for one model for the other?
- What does

  having something in common

  mean for two models?

Model Complexity

- Model complexity **will always grow**.
- It happens when we
  - **couple** models
  - add/replace new physical process in a given model.

SINTEF

Motivations

Rapid Prototyping

- We target a **couple of weeks** to setup a model
- includes:
  - parameter input
  - control
  - calibration tool

Model Exploration

- What are the **physical variables** that your model knows about?
- How they **relate** with each other?

Interactive Use

- We provide tools for the developer
- The tools **have to be interactive** to help **during** the implementation.

SINTEF

Variable Graph (VG)

Physical Model

- **Physical quantities**
- **Relationships** between quantities

Physical quantities

- Physical quantities have **names** carefully chosen by the experts
- The relationship between variables can be **explicit** or **implicit**
- implicit relationship gives an equation.

Model = Variable Graph

Variable Graph

- The **nodes** are the variables names
- The **edges** provides the dependence relationship between the variables.
- Directed acyclic graph

Variable Graph (VG)

Physical Model

- **Physical quantities**
- **Relationships** between quantities

Physical quantities

- Physical quantities have **names** carefully chosen by the experts
- The relationship between variables can be **explicit** or **implicit**
- implicit relationship gives an equation.

Model = Variable Graph

Variable Graph

- The **nodes** are the variables names
- The **edges** provides the dependence relationship between the variables.
- Directed acyclic graph

Example

- We have the following chemical systems

$$\mathrm{Li}^{+} + \mathrm{e}^{-} \longleftrightarrow \mathrm{Li}$$

- The equations are

$$\frac{dc_e}{dt} = R \qquad \frac{dc_s}{dt} = -R$$

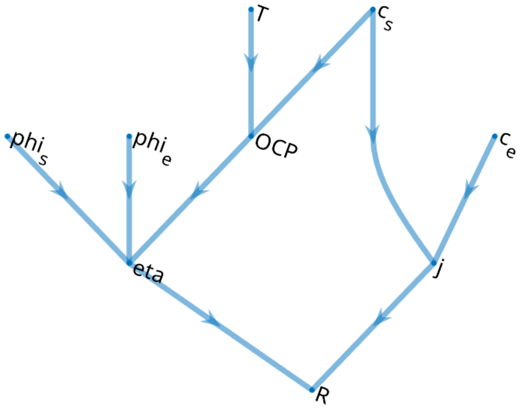
with

$$R = j(c_e, c_s)\sinh(\eta)$$
$$\eta = \phi_e - \phi_s - \mathtt{OCP}(c_e, T)$$

The named variables are

c_e	:	Li concentration in electrolyte
c_s	:	Li concentration in electrode
R	:	reaction rate
j	:	exchange current density
ϕ_e	:	Electric potential in electrolyte
ϕ_s	:	Electric potential in electrode
η	:	overpotential
$\mathtt{OCP}$	:	open circuit potential

Example



(Only the dependency of the reaction term *R*)

Variable graph (VG)

A model development approach based on variable graph enables to

BREAK THE COMPLEXITY

by using a **two-step** approach

First Step	Second Step
Design Variable Graph	Implement Declared Functions

Implementation Requirement

To get a **reusable** code, we need to be able to do the following operations

Import A Model	Add/Remove Variables to a Model	Add/Change Functional Dependencies
Import a VG and use the functions that are defined.	Add/Remove nodes to the VG keeping all the rest.	Add/Change directed edges to VG, keeping all the rest

FULL GRAPH EDITING CAPABILITIES - AT EVERY LEVEL

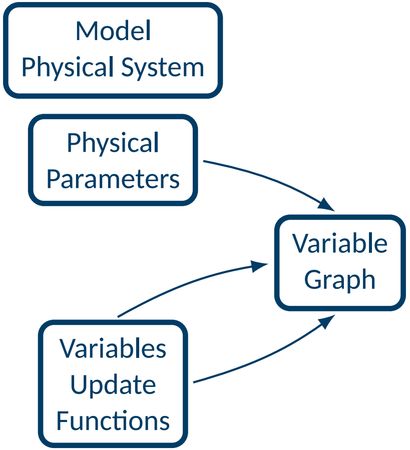
Later, we will go through an example where we see these requirements are met by our implementation

Model Object

```
classdef:

properties:

methods:
```



Model Hierarchy

Model A

nodes: $\{\mathtt{var}_1, \cdots, \mathtt{var}_{n_A}\}$

Model B

nodes: $\{\mathtt{var}_1, \cdots, \mathtt{var}_{n_B}\}$

Graphs are imported as subgraphs at model initialization

Submodel names are used as prefix

Namespace mechanism

Model

`modelA`
`modelB`

nodes: $\{\mathtt{modelA.var}_1, \cdots, \mathtt{modelA.var}_{n_A}$

$\mathtt{modelB.var}_1, \cdots, \mathtt{modelB.var}_{n_B}$

$\mathtt{var}_1, \cdots, \mathtt{var}_n\}$

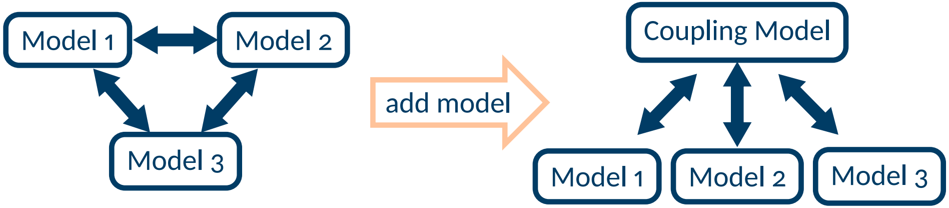
additional edges (always if coupling !)

Model Couplings

- We include **model names**

  model name = namespace for the model variables

- Hierarchical approach : We couple models by adding a coupling model.



VG for Automatic Assembly

automatic assembly setup

roots of the graph ⇒ primary variables

leaves of the graph ⇒ equations

Pre-Processing Step on VG

1. Order the graph using the declared dependency
2. Identify the **primary variables** and the **equations**
3. Setup the **sequence of function call** to update sequentially all variables

Newton Step

1. Instantiate primary variables as **MRST-AD**
2. Run sequence of function calls
3. Extract from the leaves the equations, stab them and send them to the MRST Newton solver

VG for Automatic Assembly

automatic assembly setup

roots of the graph ⇒ primary variables

leaves of the graph ⇒ equations

Pre-Processing Step on VG

1. Order the graph using the declared dependency
2. Identify the **primary variables** and the **equations**
3. Setup the **sequence of function call** to update sequentially all variables

Newton Step

1. Instantiate primary variables as **MRST-AD**
2. Run sequence of function calls
3. Extract from the leaves the equations, stab them and send them to the MRST Newton solver

Graph Setup

Declarative Graph Setup

We do not have to figure out the layout of the graph beforehand.

We use a **declarative** approach : We register the nodes (variables) and the edges (functional dependencies). They can be registered in a **arbitrary order**.

The code **builds the graph** for us and provides **interactive tools** for exploration and visualization.

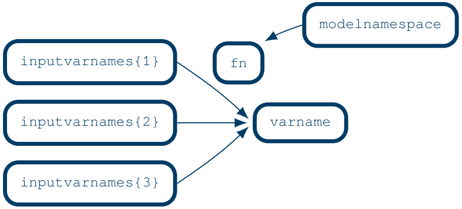
Graphs can be complicated objects...

Simple structures for the architecture implementation

In `BaseModel`, we have the methods

- `registerVarNames` to add variable names
- `registerPropFunction` to add functions

with some **syntactic sugar**



The nodes (variable names)

```
classdef VarName
 properties
  namespace % list of model names
  name      % name of the variable
  index     % if array variables
  dim       % if array variables
 end
end
```

The edges (functional relationships)

```
classdef PropFunction
 properties
  varname         % target variable
  inputvarnames   % list of input variables
  modelnamespace  % model name space
                  % for the parameters
  fn % function handler to update target
 end
end
```

Example

We register the variables

```
classdef ReactionModel < BaseModel
  methods
   function model = registerVarAndPropfuncNames(model)

    model = registerVarAndPropfuncNames@BaseModel(model);

    varnames = {};
    varnames{end + 1} = 'phi_s';
    varnames{end + 1} = 'c_s';
    varnames{end + 1} = 'phi_e';
    varnames{end + 1} = 'c_e';
    varnames{end + 1} = 'eta';
    varnames{end + 1} = 'R';
    varnames{end + 1} = 'OCP';
    varnames{end + 1} = 'j';

    model = model.registerVarNames(varnames);
```

We register the functions

```

    fn = @ReactionModel.updateReactionRateCoefficient;
    inputnames = {'c_e', 'c_s'};
    model = model.registerPropFunction({'j', fn, inputnames});

    fn = @ReactionModel.updateOCP;
    inputnames = {'c_s'};
    model = model.registerPropFunction({'OCP', fn, inputnames});

    fn = @ReactionModel.updateEta;
    inputnames = {'phi_e', 'phi_s', 'OCP'};
    model = model.registerPropFunction({'eta', fn, inputnames});

    fn = @ReactionModel.updateReactionRate;
    inputnames = {'eta', 'j'};
    model = model.registerPropFunction({'R', fn, inputnames});

   end
  end
end
```

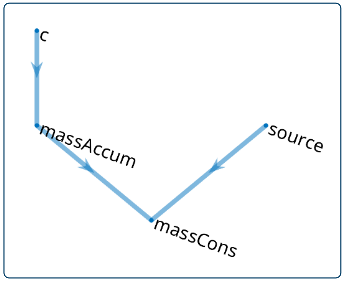
Coupling model - Example

Example from tutorial. We build-up from bottom to top

- `ConcentrationModel`
- `ReactionModel`
- `ThermalModel`
- `ConcentrationReactionModel`
  couples `ConcentrationModel` and `ReactionModel`.
- `ConcentrationReactionThermalModel`
  couples `ConcentrationReactionModel` and `ThermalModel`.

Concentration and reaction model

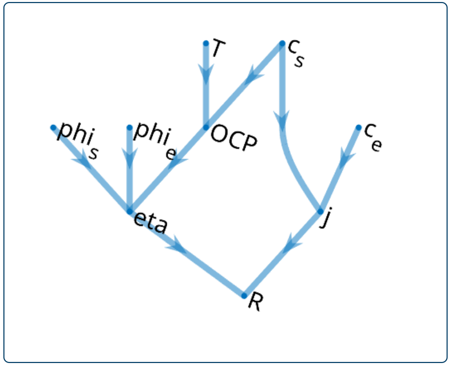
$$\frac{dc}{dt} = S$$



`ConcentrationModel`

$$R = j(c_e, c_s)\sinh(\eta)$$
$$\eta = \phi_e - \phi_s - \mathtt{OCP}(c_e, T)$$

`ReactionModel`



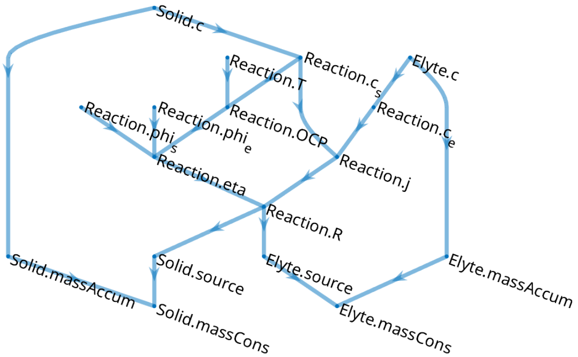
Concentration and reaction model

$$\frac{dc_e}{dt} = R \quad \frac{dc_s}{dt} = -R$$

with

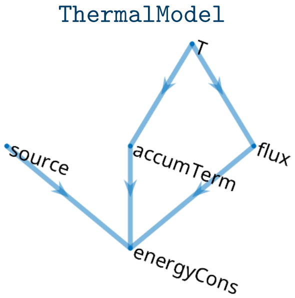
$$R = j(c_e, c_s)\,\sinh(\eta)$$
$$\eta = \phi_e - \phi_s - \mathtt{OCP}(c_e, T)$$

```
ConcentrationReactionModel
⇒ ConcentrationModel (Elyte)
⇒ ConcentrationModel (Solid)
⇒ ReactionModel (Reaction)
```

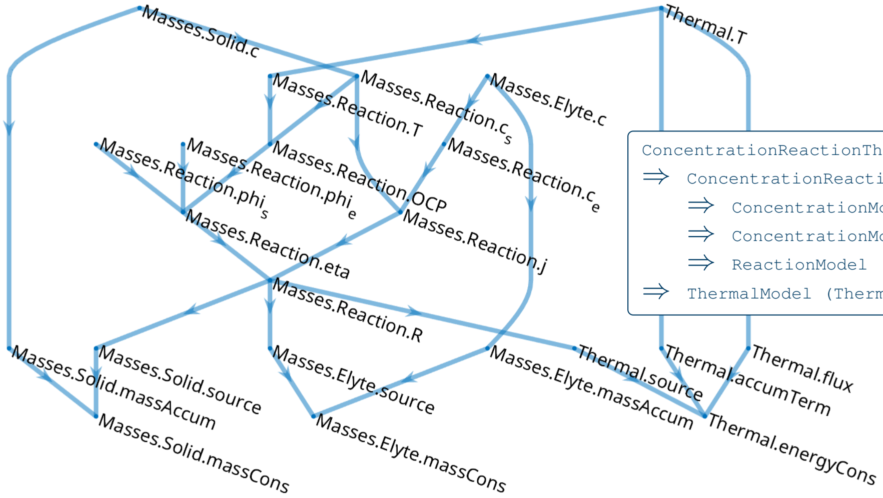


Thermal model

$$\frac{\partial}{\partial t}(cT) + \nabla \cdot (-\kappa \nabla T) = S$$

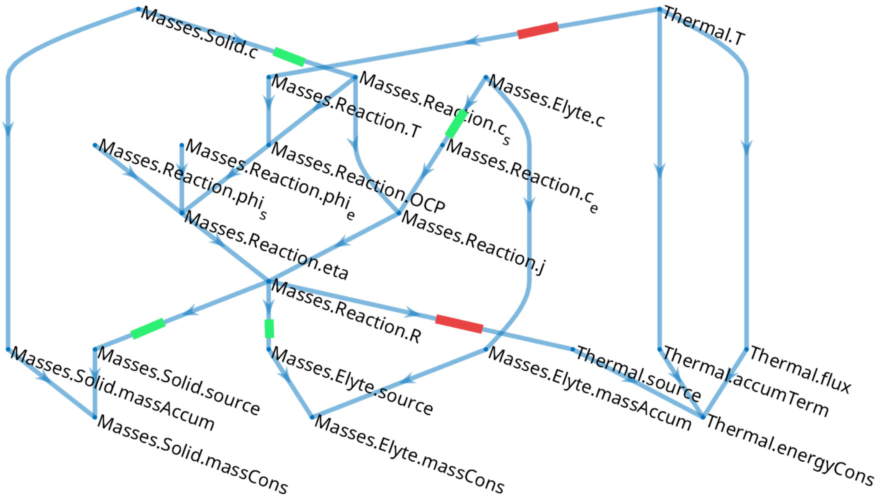


Coupled Thermal Reaction model



```
ConcentrationReactionThermalModel
⟹ ConcentrationReactionModel (Masses)
    ⟹ ConcentrationModel (Elyte)
    ⟹ ConcentrationModel (Solid)
    ⟹ ReactionModel (Reaction)
⟹ ThermalModel (Thermal)
```

Couplings in Thermal Reaction model



SINTEF

Example

Primary Variables

```
>> cgit.printRootVariables

3 Root Variables:
-----------------
Masses.Solid.c
Masses.Elyte.c
Thermal.T

2 Static Variables:
-------------------
Masses.Reaction.phi_s
Masses.Reaction.phi_e
```

Equations

```
>> cgit.printTailVariables

3 Tail Variables:
-----------------
Masses.Solid.massCons
Masses.Elyte.massCons
Thermal.energyCons
```

Ordered function call list

```
>> cgit.printOrderedFunctionCallList

state.Masses = model.Masses.updateReactionConcentrationS(state.Masses);
state.Masses = model.Masses.updateReactionConcentrationE(state.Masses);
state = model.updateTemperature(state);
state.Masses.Solid = model.Masses.Solid.updateMassAccum(state.Masses.Solid);
state.Masses.Elyte = model.Masses.Elyte.updateMassAccum(state.Masses.Elyte);
state.Thermal = model.Thermal.updateAccumTerm(state.Thermal);
state.Thermal = model.Thermal.updateFlux(state.Thermal);
state.Masses.Reaction = model.Masses.Reaction.updateOCP(state.Masses.Reaction);
state.Masses.Reaction = ...
    model.Masses.Reaction.updateReactionRateCoefficient(state.Masses.Reaction);
state.Masses.Reaction = model.Masses.Reaction.updateEta(state.Masses.Reaction);
state.Masses.Reaction = ...
    model.Masses.Reaction.updateReactionRate(state.Masses.Reaction);
state.Masses = model.Masses.updateConcentrationSource(state.Masses);
state = model.updateThermalSource(state);
state.Masses.Solid = model.Masses.Solid.updateMassCons(state.Masses.Solid);
state.Masses.Elyte = model.Masses.Elyte.updateMassCons(state.Masses.Elyte);
state.Thermal = model.Thermal.updateEnergyCons(state.Thermal);
```

This functions are evaluated **as such** using `eval` in MATLAB, see here.

macro mechanism

(Code that generates code)

Conclusion

Variable Graph based Simulator design enables us to

- Break the complexity
- Build a hierarchy of interacting physical simulation models
- Reuse update-functions which are preferably implemented at lowest scope
- Automate the assembly