Design aspects of Bioprocess Library® for Modelica

Jan Peter Axelsson
Vascaia AB, Stockholm, Sweden
OpenModelica workshop 2021-02-02

Note, “Bioprocess Library” is a registered trademark.
Outline

• Scope
• Structure Library /Application – flexibility needed
• Key is parametrization of the library
  • Media packages
  • Reactor model – part library part application
• Jupyter notebook – Python3, PyFMI, FMU
  • Command line interaction!
• Conclusion
Scope

• Quicker to setup first model – key as consultant

• Document model with focus on the customer
  • Play down standard components

• Clarity of code!

• Useful also in teaching context for biotech-people

• Teach myself Modelica…. 😊
Example: Batch cultivation
Example: Fedbatch cultivation
Fixed feedrate profile
Example: Fedbatch cultivation
Feedprofile with on-line control
Example: Perfusion cultivation
Flexibility needed

• Process configuration may vary
  • Batch, Fedbatch, Continuous, Perfusion, ... (up-stream)
  • Scale-down
  • Later include down-stream processing

• Process control
  • Substrate levels, Dissolved oxygen, pH, temperature...
  • Feeding strategies...
  • Optimization

• Cells cultivated
  • Yeast, Ecoli, CHO.... Hosts for recombinant proteins
  • Liquid- and gas-phase varies
Object orientation...  
- connectors

- Liquid pipes - LiquidCon
- Gas pipes - GasCon
- Electrical signals - MSL RealInput, RealOutput etc
- Provide flexibility
  - Re-configuration of process setup
  - Change of control systems

- Provide flexibility for different cultures, liquids...?
  - Parametrize EquipmentLib with Application def parts!
Structure of code

Application

Library

• Liquid-, gas-phases
• EquipmentLib
  • Tank
  • Pump
  • Reactor
  • Sensor
  • ...
• ControlLib
Structure of code

Application
• LiquidphaseYeast
• GasphaseYeast

Library
• Liquid-, gas-phases, signals
• EquipmentLib
  • Tank
  • Pump
  • Reactor
  • Sensor
  • ...
• ControlLib
Structure of code

Application
• LiquidphaseYeast
• GasphaseYeast

Library
• Liquid-, gas- phases, signals
• EquipmentLib
  • Tank
  • Pump
  • Reactor
  • Sensor
  • ...
• ControlLib
Structure of code

Application
- LiquidphaseYeast
- GasphaseYeast
- CultureYeast model
- BufferYeast model
- GasLiquidTransferYeast

Library
- Liquid-, gas- phases, signals
- EquipmentLib
  - Tank
  - Pump
  - Reactor
  - Sensor
  - ...
- ControlLib
Structure of code

Application
- LiquidphaseYeast
- GasphaseYeast
- CultureYeast model
- BufferYeast model
- GasLiquidTransferYeast

Library
- Liquid-, gas-phases, signals
- EquipmentLib
  - Tank
  - Pump
  - Reactor
  - Sensor
  - ...
- ControlLib
Try to improve...

Application...
- Yeast in airated reactor

Work from “templates” partial packages, models...

Extend partial package LiquidphaseBase -> LiquidphaseYeast
Extend partial package GasphaseBase -> GasphaseYeast
(Define number of components, provide index etc)

Extend partial model ReactorInterface -> CultureYeast model
Extend partial model ReactorInterface -> GasLiquidTransfer
(Define mappings concentration to flows: c[i] -> q[i] - static or dynamical systems)
encapsulated package BPL_YEAST_AIR
  package LiquidphaseYeast
  package GasphaseYeast
  model CultureYeast
  model GasLiquidTransferYeast
  package EquipmentYeast
    import BPL.EquipmentLib
    .... extend, redeclare
  model Fedbatch
    ... configure with redeclared BPL components
end BPL_YEAST_AIR;
package EquipmentYeast

import BPL.EquipmentLib;

extends EquipmentLib(
    redeclare package Liquidphase = LiquidphaseYeast,
    redeclare package Gasphase = GasphaseYeast,
    redeclare model Culture = CultureYeast(
        redeclare package Liquidphase=LiquidphaseYeast),
    redeclare model GasLiquidTransfer=GasLiquidTransferYeast(
        redeclare package Liquidphase=LiquidphaseYeast,
        redeclare package Gasphase=GasYeast),
    redeclare model Buffer = NoBuffer(
        redeclare package Liquidphase=LiquidphaseYeast));

disable EquipmentYeast;
Library – formal parameter

package BPL

... package EquipmentLib
    replaceable package **Liquidphase**=LiquidphaseBase
        constrainedby LiquidphaseBase;
    replaceable package **Gasphase**=GasphaseBase
        constrainedby GasphaseBase;
    replaceable model **Culture** = NoCulture
        constrainedby ReactorInterface;

... package ControlLib...
end BPL;
Liquidphase - "template"

partial package LiquidphaseBase
    constant String name;
    constant Integer nc;
    type Concentration = Real[nc];
end LiquidphaseBase;
cont’ Liquidphase

package LiquidphaseYeast

    import BPL.LiquidphaseBase;
    extends LiquidphaseBase
        (name="Yeast medium...", nc=3);
    constant Integer X=1;
    constant Integer G=2;
    constant Integer E=3;
    constant Real[nc] mw = {24.6, 180.0, 46.0};

end LiquidphaseYeast;
cont’ Liquidphase

Record LiquidphaseYeast_data
   constant String name = LiquidphaseYeast.name;
   constant Integer nc = LiquidphaseYeast.nc;
   constant Integer X = LiquidphaseYeast.X;
   ...
   constant Real[nc] mw = LiquidphaseYeast.mw
End LiquidphaseYeast_data;
--

Repetious but seems to be needed
Connector LiquidCon - ”template”

package EquipmentLib
    replaceable package Liquidphase = LiquidphaseBase
        constrainedby LiquidphaseBase;
connector LiquidCon
    stream Liquidphase.Concentration c;
    flow Real F;
    Real p;
end LiquidCon;
model Pump, Tank, Reactor, Sensor etc
BPL ReactorType

General reactor with
• n_inlets, n_outlets, n_ports

• connector LiquidCon – common for EquipmentLib
• connector GasCon – common for EquipmentLib

• model Culture
• model GasLiquidTransfer
• model Buffer
Library

Library code
- Define general liquid- and gas-phases
- Define general reactor
- Define general pumps, tanks, filters, sensors...
- Define controllers…
Application

- Use library media partial model
  Specify components
- Define cell culture
- Define gas-liquid-transfer
- Define buffer reactions

- Adapt library to application
- Make instances for application process and connect sub-models

Gas-Liquid-transfer

Gasphase

Liquid phase

Cell culture

Buffer reactions
cont’ BPL ReactorType

Reactor concentration $c[i]$ affects everything
User inner/outer connection to application
• model Culture
• model GasLiquidTransfer
• model Buffer

These models static or dynamic of $c[i]$
• culture_q[i] – cell specific flow $q[i]$
• gas_to_liquid_transfer_Q[i] – total flow $Q[i]$
• liquid_to_gas_transfer_Q[i]
• buffer_Q[i]
cont’ BPL ReactorType

// Mass-balance for the liquid phase of the reactor:
for i in 1:Liquidphase.nc loop
    der(m[i]) = culture_q[i]*m[X] + buffer_Q[i]
    + gas_to_liquid_transfer_Q[i]
    + sum(actualStream(inlet[j].c[i])*inlet[j].F for j)
    + sum(c[i]*outlet[k].F for k);
    for j in 1:n_inlets loop inlet[j].c[i] = c[i]; end for;
end for;
der(V)  =  sum(inlet[j].F for j) + sum(outlet[k].F for k);
Applications with Yeast
- processes Batch and Fedbatch

package EquipmentYeast

    import BPL.EquipmentLib;
    extends EquipmentLib(redeclare package Liquidphase = LiquidphaseYeast
                          redeclare model Culture = CultureYeast
    ...
end EquipmentYeast;

model Batch

    LiquidphaseYeast_data liquidphase;
    EquipmentYeast.ReactorType bioreactor(X=liquidphase.X);

equation

end Batch;

model Fedbatch....
... and process Fedbatch

model Fedbatch "Fedbatch cultivation of yeast"
  LiquidphaseYeast_data liquidphase;
  EquipmentYeast.ReactorType bioreactor
    (X=liquidphase.X, n_inlets=1);
  EquipmentYeast.Feedsystem feedtank;
  Control.DosageSchemeExp dosagescheme;
  equation
    connect(bioreactor.inlet[1], feedtank.outlet);
    connect(dosagescheme.F, feedtank.Fsp);
end Fedbatch;
Jupyter notebook and Bioprocess Library for Modelica

In [3]:
```python
newplot()
for x in [0.7, 1.0, 1.3]: init(Vx_0=x); simu()
```

Batch cultivation

In [4]:
```python
describe('bioreactor.V')
```

Reactor broth volume [ L ]

Comment: In the diagram above we see the impact of variation in the initial cell concentration during batch cultivation.
More pedestrian way...

```
In [5]:  # Script bp6a_batch_setup - defines fmu_model, parDict[] and more

    # - newplot()
    plt.figure()
    ax1=plt.subplot(2,1,1); ax1.grid(); ax1.set_ylabel('X and S [g/L]')
    ax2=plt.subplot(2,1,2); ax2.grid(); ax2.set_ylabel('mu [1/h]'); ax2.set_xlabel('Time [h]')
    lines=['-','--',':']; linecycler = cycle(lines)

    for x in [0.7, 1.0, 1.3]:
        # - init(VX_0=x)
        parDict['bioreactor.n_0[1]'] = x
        # - simu(7)
        model = load_fmu(fmu_model)
        for key in parDict.keys(): model.set(key, parDict[key])
        sim_res = model.simulate(0, 7, options=opts)
        linetype = next(linecycler)
        ax1.plot(sim_res['time'], sim_res['bioreactor.c[1]'], color='r', linestyle=linetype); ax1.legend(['X', 'S'])
        ax1.plot(sim_res['time'], sim_res['bioreactor.c[2]'], color='b', linestyle=linetype)
        ax2.plot(sim_res['time'], sim_res['bioreactor.culture_q[1]'], color='b', linestyle=linetype)
```
Jupyter with Python3 and PyFMI

• Jupyter combine: code, diagrams, and text
• PyFMI (Modelon) – runs FMU
  (Modelon but on Github since a year)
• OpenModelica export FMU...
• Teaching – simplify command line interaction
  • newplot()
  • par(), init()
  • simu()
  • disp(), describe()

Run Modelica as if Simnon! (H. Elmqvist, 1975)
Larger example
- Yeast continuous (incl gas-phase)
Conclusion

- Application code clearly separated from library
- Got flexibility needed
  - ”Work with structure”
    - Object-orientation, connectors...
  - “Keep structure, change content”
    - Type level: Redeclaration... formal parameter: packages, models ... “polymorphism”
    - Instance level: Smaller changes for readability, eg what conc variable is cell conc etc
  Key words: Formal parameter, redeclare, replaceable, see Fritzon section 4.4
- Adapt EquipmentLib at one place, cf MSL Fluid differs
  - DRY still difficult...
- Modelica can bring very neat and compact code, cf library ReactorType
- FMU
  - OpenModelica do not propagate constants, cf LiquidphaseYeast.mw etc
  - List of continuous time state variables available, but not discrete time?
- Jupyter notebooks very useful also in teaching context
  - Good to simplify command line interaction!
Acknowledgement
Stackoverflow – name: janpeter

• Extending packages and access to the content
• Parametrised Modelica library and possibility to use models as parameters – part 1, 2, 3
• How to construct a balanced connector for liquids in Mocelica

Discussion are around simplified code examples that can be run – see you there!
6. Design Rationale – Code Reuse

Code reuse is a desirable but hard-to-reach goal for software development. Modelica contributes to this goal in several ways. Its non-causal equation-based modeling style permits model components to be reused in different contexts, automatically adapting to the data flow order in specific simulation applications, i.e. the Modelica compiler automatically arranges equations for solution with particular inputs or outputs. Object orientation and polymorphism significantly enhances the potential for reuse of Modelica model components.