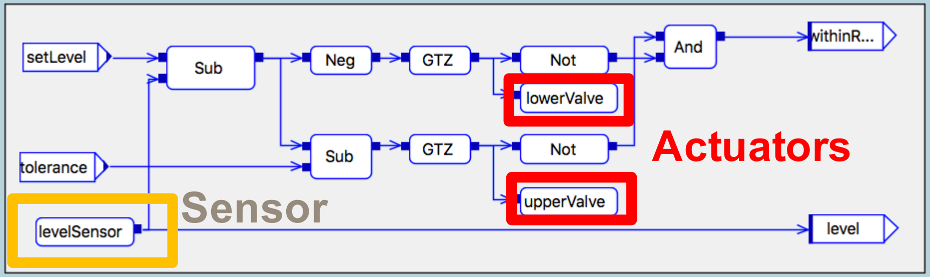
Feature-oriented automation programming with Bloqqi

NIKLAS FORS, GÖREL HEDIN, LUND UNIVERSITY, MODPROD, 2017-02-07

Bloqqi: data-flow programming
for control systems

Bloqqi program

Visual view

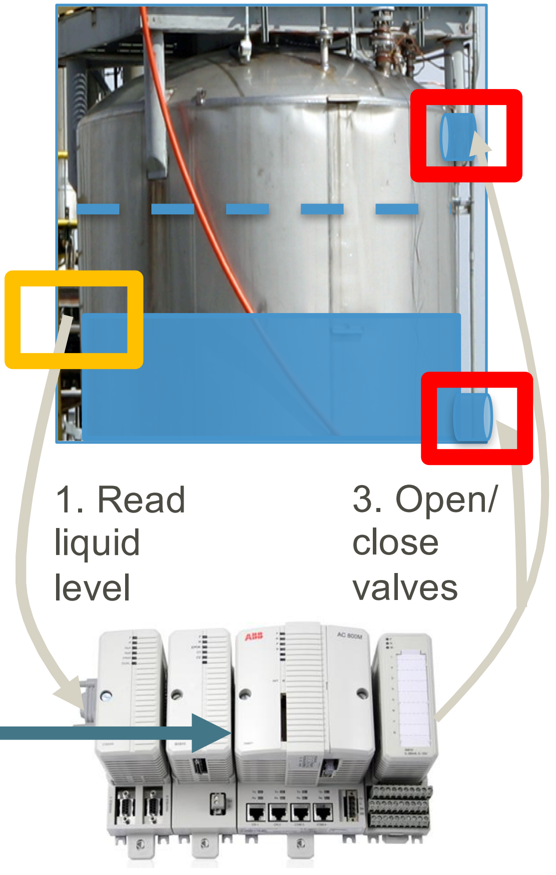


Textual view

```
diagramtype  Tank(setLevel:  Int, tolerance:  Int
    => level: Int, withinRange:  Bool) {
  upperValve:  Valve;
  lowerValve:  Valve;
  levelSensor:  Sensor;
  ...
  connect(setLevel,  Sub_1.in1);
  connect(levelSensor.out,  Sub_1.in2);
  connect(levelSensor.out,  level);
  ...
}
```

Runs in

Real world



2. Compute control signal

Control system

Problem

Diagram **variant**: a combination of **features** *(on a base diagram)*

Example optional features:

- For *Tank*: **heating, agitation**, etc.
- For *Proportional controller*: **derivative part, integral part**, etc.

How to handle variants?

- With **modular** libraries
- **Easy programming** for automation engineer

Alternative solutions

Alternative solutions

- **Copy-paste**
  - Problem: code duplication
- **Template**
  - Problem: not extensible and complex diagrams

Our solution:

- **Bloqqi**

Template: Diagram with all anticipated features that are turned on/off using parameters

The Bloqqi language

Some inspiration from Modelica

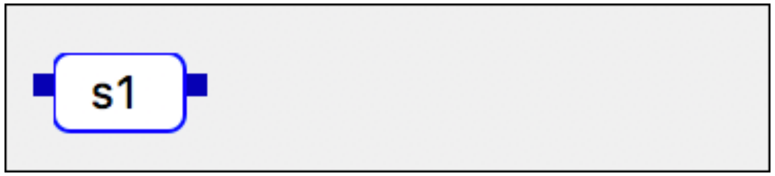
- Inheritance (and redeclare)
- Both textual and visual syntax

New language constructs for variants

- Connection interception } Specialize diagram
- Wirings
- Recommendations } Describe features

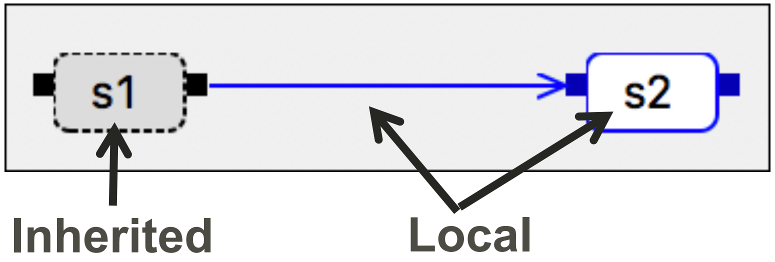
Inheritance and Connection Interception

A



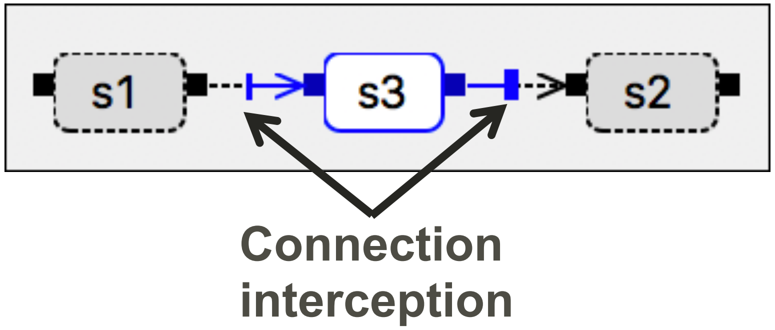
```
diagramtype A {
  s1: S;
}
```

B extends A



```
diagramtype B extends A {
  s2: S;
  connect(s1.out, s2.in);
}
```

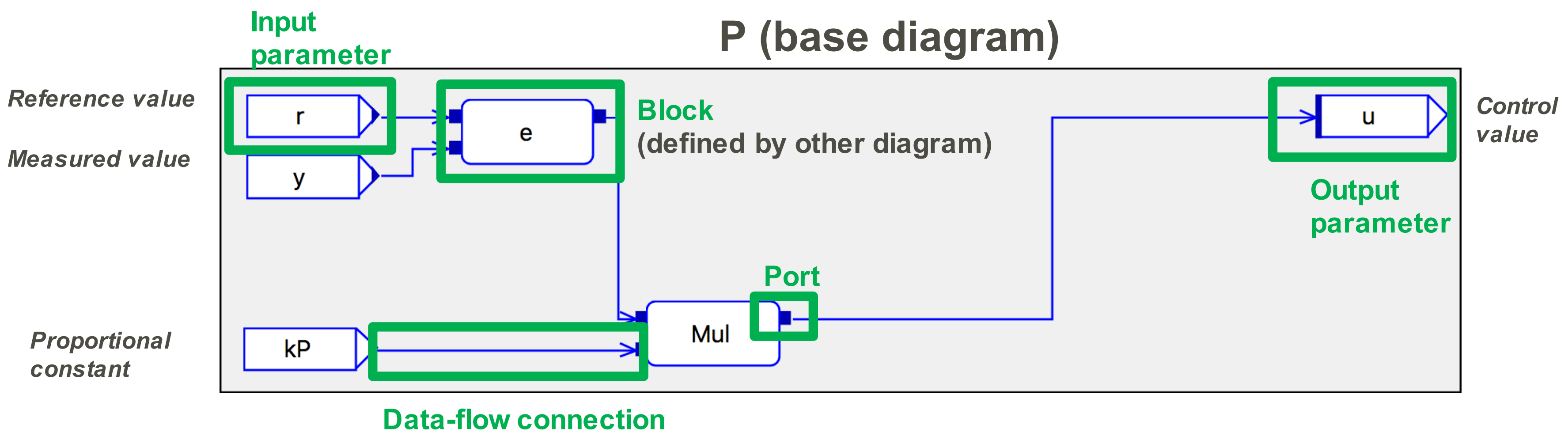
C extends B



```
diagramtype C extends B {
  s3: S;
  intercept s2.in with s3.in,s3.out;
}
```

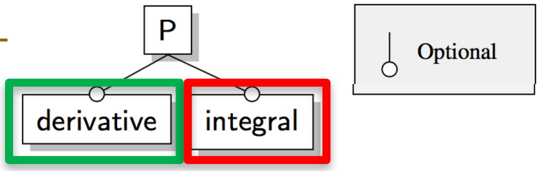
Example: Proportional controller

Visual syntax:

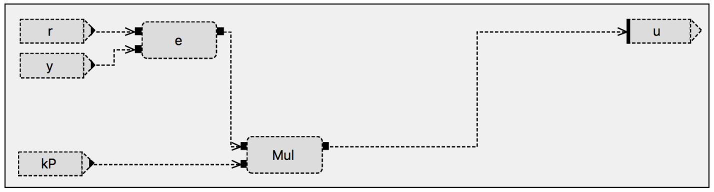


4 controller variants

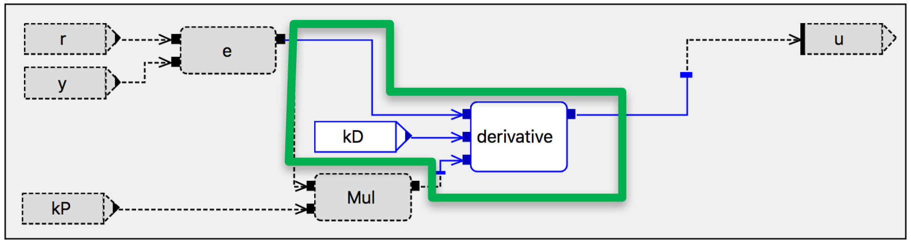
Corresponds to feature model



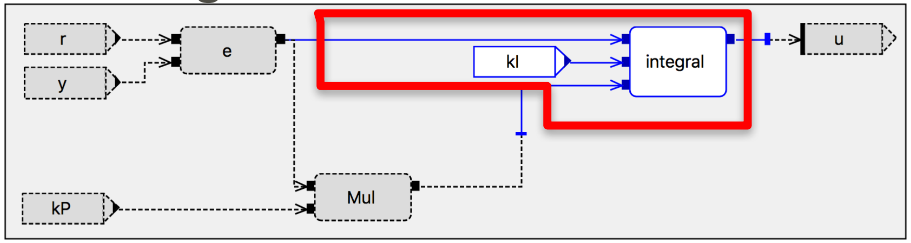
P (base diagram)



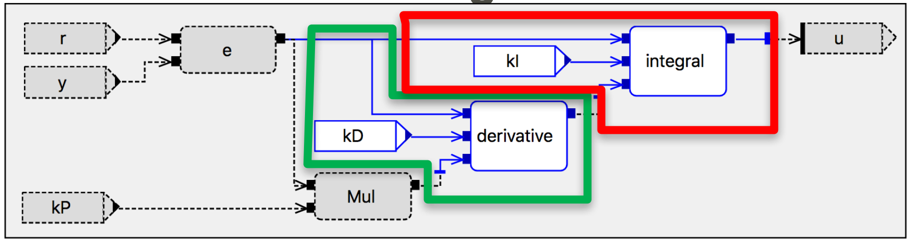
P + derivative



P + integral

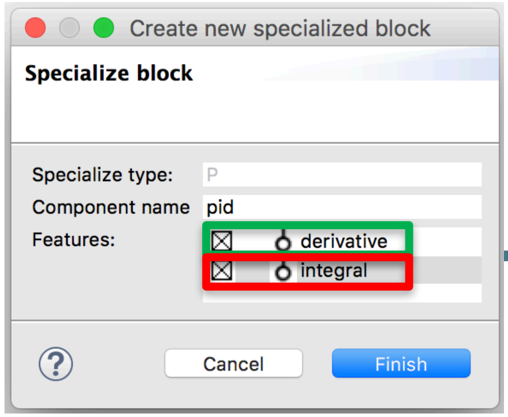


P + derivative + integral



We want:
wizard that automatically wires features

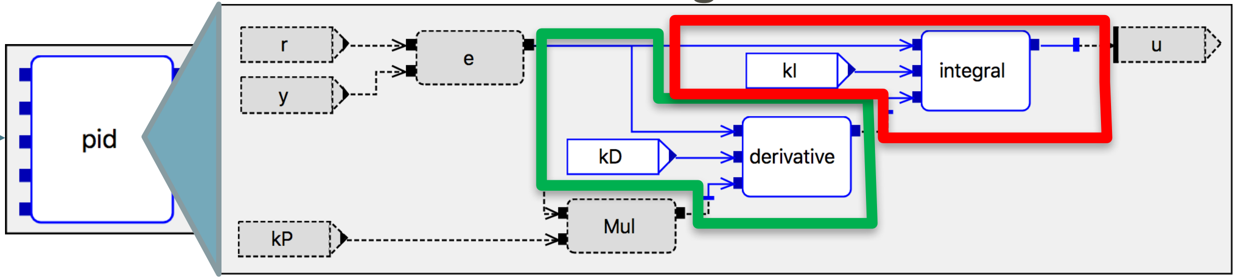
Feature wizard for P


Create new specialized block

Specialize block

Specialize type:	P
Component name	pid
Features:	☒ derivative
	☒ integral

P + derivative + integral

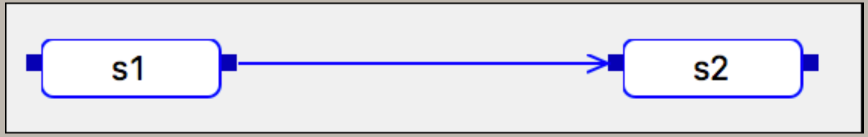


Automatic wiring of features

How can we compute feature wizard from library code?

Recommendations – simple example

Base diagram **A** with mandatory features



```
diagramtype A {
  s1: S;
  s2: S;
  connect(s1.out, s2.in);
}
```

Recommendations – optional features

```
recommendation A {
  f: F[s2.in];
}
```

Feature name

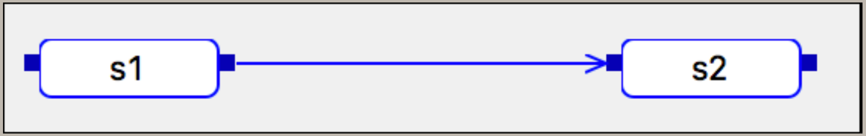
Feature type (another diagram)

How the wiring is done

A has an optional feature **f**, that is inserted before **s2**

Recommendations – simple example

Base diagram **A** with mandatory features



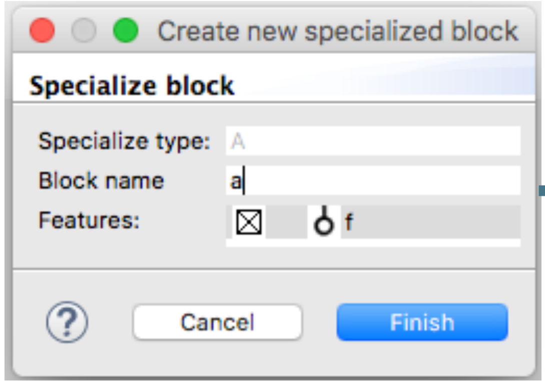
```
diagramtype A {
  s1: S;
  s2: S;
  connect(s1.out, s2.in);
}
```

Recommendations – optional features

```
recommendation A {
  f: F[s2.in];
}
```

A has an optional feature **f**, that is inserted before **s2**

Computed feature wizard for A

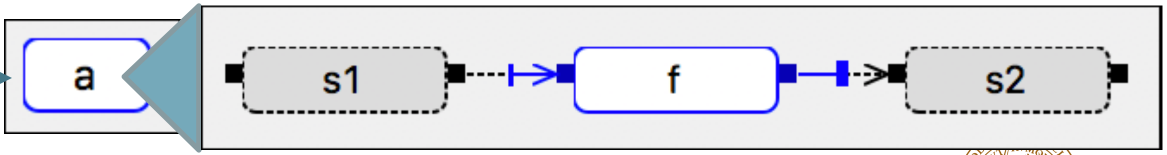

Create new specialized block

Specialize block

Specialize type: A

Block name: a

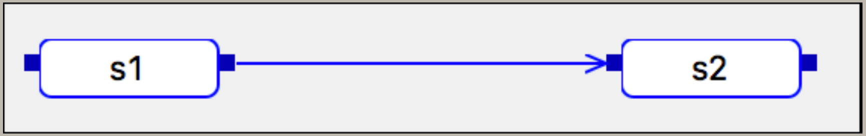
Features: ☒ f



f is before **s2**

Wirings

Base diagram **A** with mandatory features



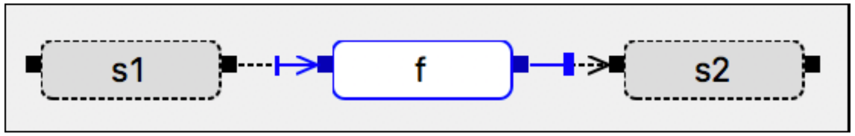
```
diagramtype A {
  s1: S;
  s2: S;
  connect(s1.out, s2.in);
}
```

Recommendations – optional features

```
recommendation A {
  f: F[s2.in];
}
```
What does this mean?

Interpreted as

```
f: F;
intercept s2.in with f.in, f.out;
```

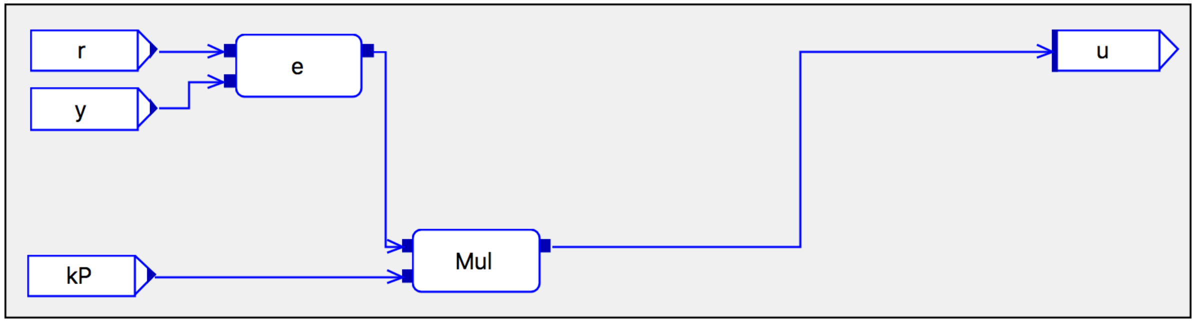


Wiring – how features are connected

```
diagramtype F(in: Int => out: Int) {
  ...
}
wiring F[=>v: Int] {
  intercept v with F.in, F.out;
}
```

P example again

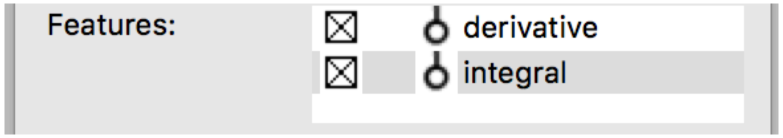
P (base diagram)



```
recommendation P {
  derivative: DPart[e.out, kD: Int, u];
  integral:   IPart[e.out, kI: Int, u];
}
```

Both will intercept **u**

Results in the features for P

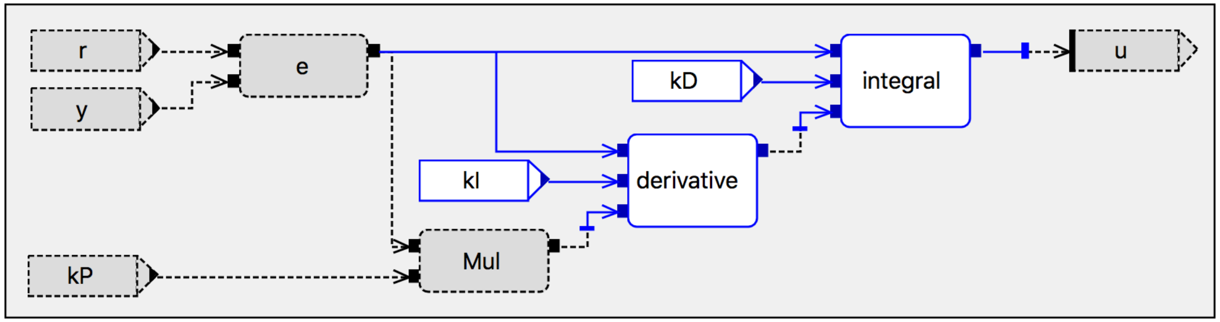


Features:

☒ derivative
☒ integral

P example again

If we select both **derivative** and **integral** then we get:

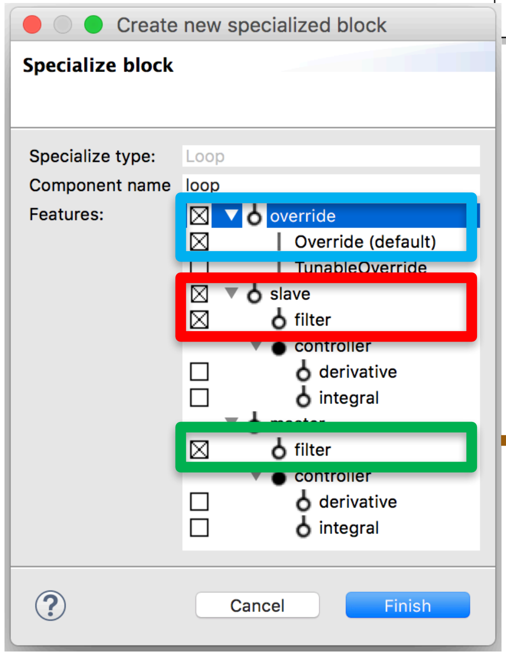


Why???

Answer: the order is defined by a **before** statement

```
recommendation P {
  derivative before integral;
}
```

A larger example: Control Loop

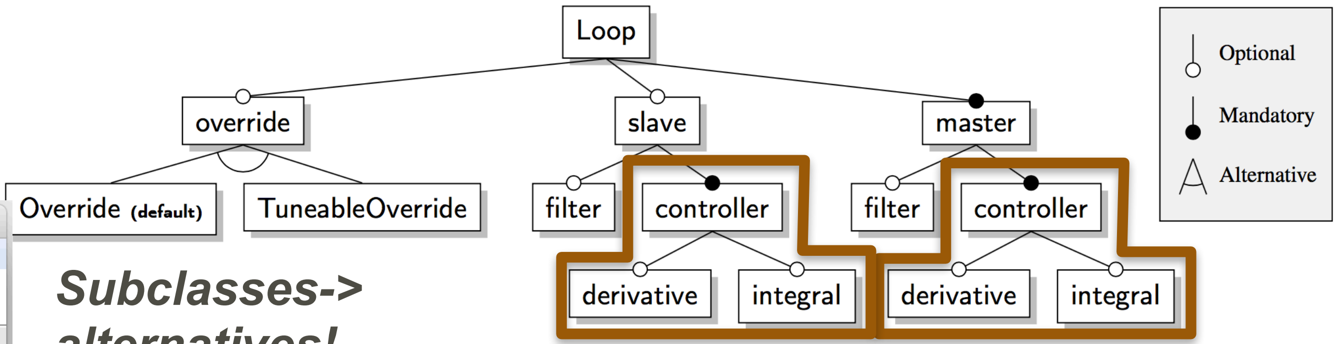

Create new specialized block

Specialize block

Specialize type: Loop
Component name: loop
Features:

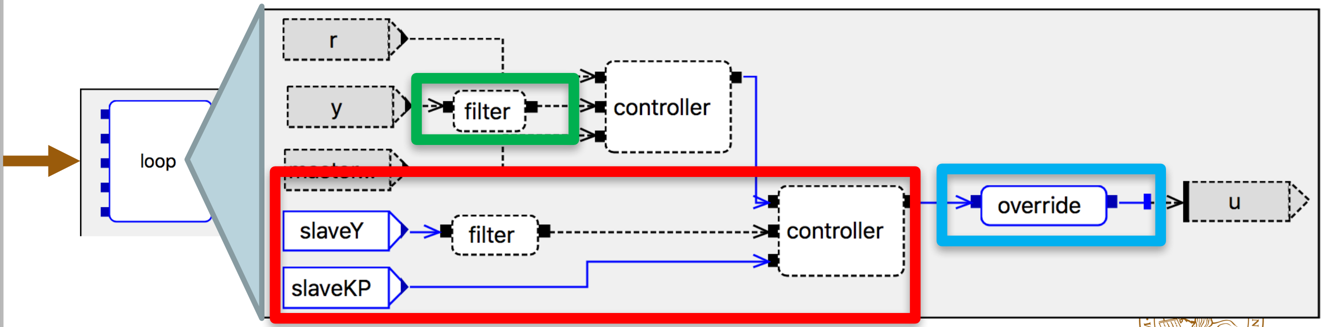
- [☒] override
  - [☒] Override (default)
  - [ ] TunableOverride
- [☒] slave
  - [☒] filter
  - controller
    - [ ] derivative
    - [ ] integral
- master
  - [☒] filter
  - controller
    - [ ] derivative
    - [ ] integral

Cancel | Finish



Subclasses-> alternatives!

Two controllers!

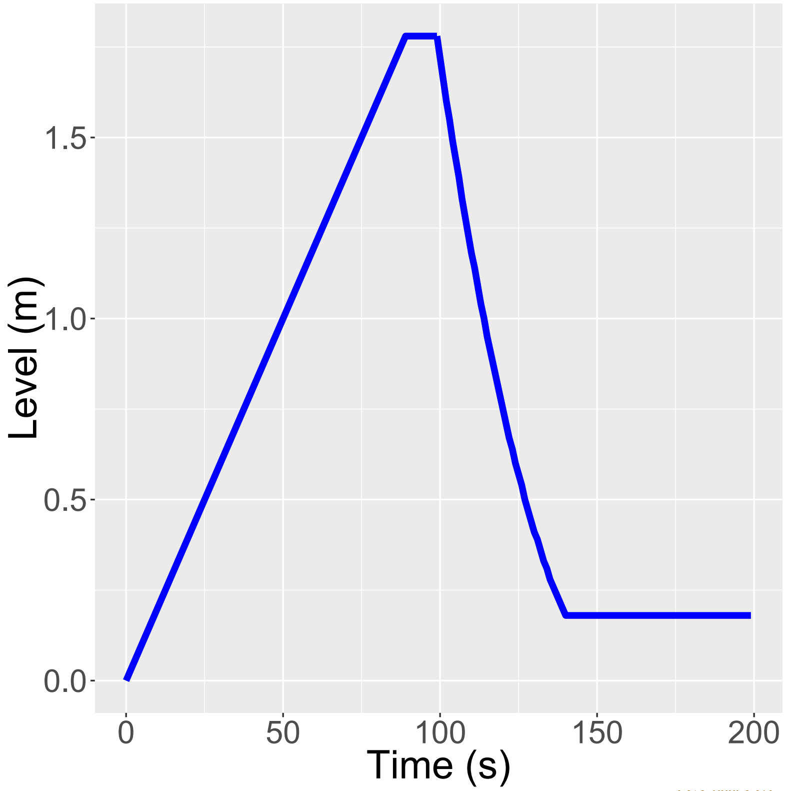


Demo

Experimentation with AC 800M and Modelica

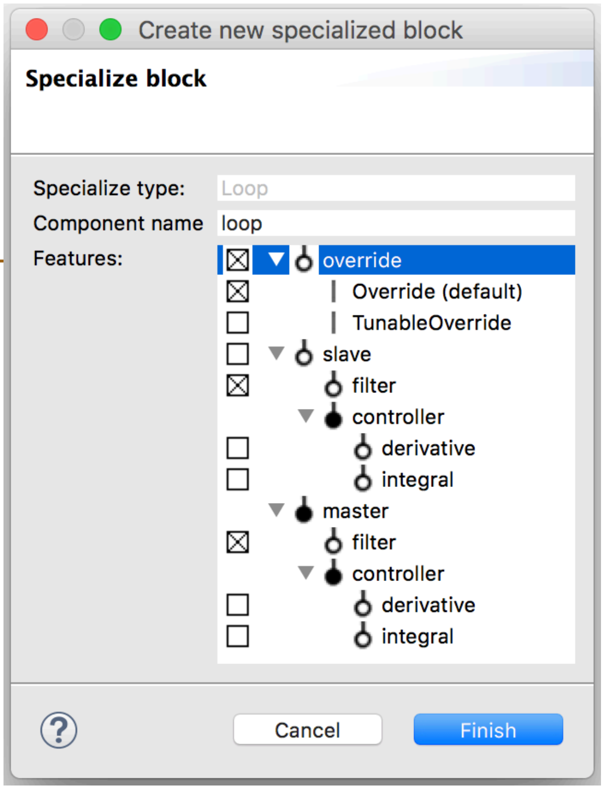
Running Bloqqi on controller hardware (ABB AC800M)

Running Bloqqi with simulated models (Modelica models exported using FMI)



Conclusion

- New language constructs
  - Connection interception, recommendations and wirings
  - Feature wizards computed based on modular library specification
- Future work
  - Combine with state-based languages



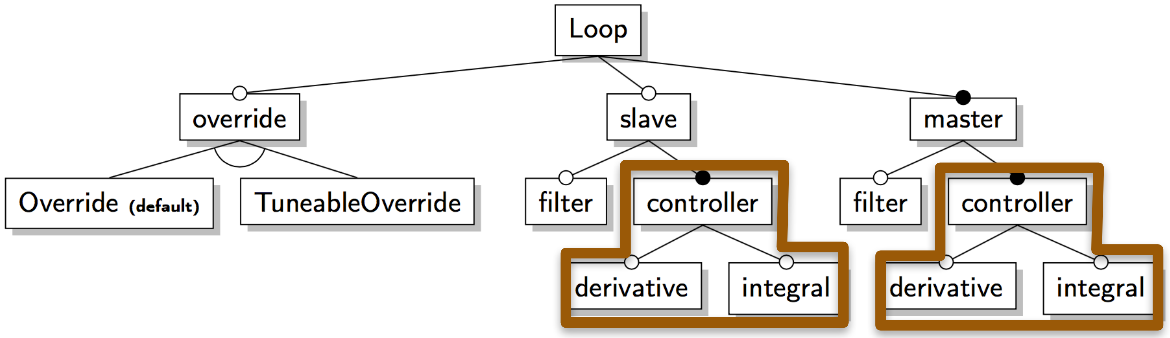
Create new specialized block

Specialize block

Specialize type: Loop
Component name: loop
Features:

```
[☒] ▼ override
[☒]     | Override (default)
[ ]     | TunableOverride
[ ] ▼ slave
[☒]     filter
      ▼ controller
[ ]         derivative
[ ]         integral
    ▼ master
[☒]     filter
      ▼ controller
[ ]         derivative
[ ]         integral
```

Computed from library



Features modularly defined

Read more

- **Bloqqi: modular feature-based block diagram programming**
  @ Onward 2016
  By Niklas Fors, Görel Hedin
- **The Design and Implementation of Bloqqi –**
  A Feature-Based Diagram Programming Language
  PhD thesis, 2016
  By Niklas Fors
- See **bloqqi.org** for pdf files

LUND
UNIVERSITY