Bloqqi
Feature-Oriented Control Programming

Niklas Fors, Lund University, @Zoom, 2020-08-18

Process Industrial IT and Automation

The Bloqqi Project

Goal: Improve code reuse

Method: Experiment with language constructs in a prototype language

Participants: Computer Science@LU, Control@LU, ABB Malmö, Modelon

Funding: PiiA (part of Vinnova)

Process Industrial IT and Automation

Participants

Computer Science@LU

Dr. Niklas Fors
Prof. Görel Hedin
Dr. Sven Gestegård Robertz

Control@LU

Prof. Anders Robertsson
(Prof. Charlotta Johnsson)

ABB Malmö

Ulf Hagberg
Christina Persson
Stefan Sällberg
Dr. Alfred Theorin

Modelon

Filip Stenström
Dr. Per-Ola Larsson
(Dr. Johan Åkesson)

Bloqqi – Improving Automation Programming

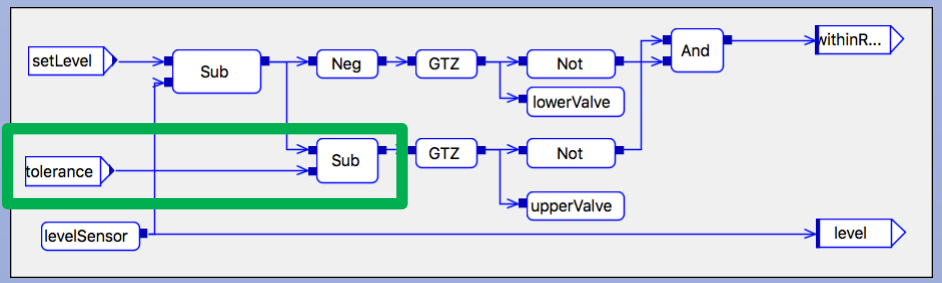
- Directed data-flow with periodic execution
- Both visual and textual
- Supports reuse: inheritance, features, ...
- Supports distributed execution: MQTT
- Supports interoperability: FMI
- Open source language and tools
- High-level implementation techniques: JastAdd RAGs

Bloqqi: Feature-based data-flow programming

Tolerance feature

Bloqqi program for tank control

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

1. Read liquid level

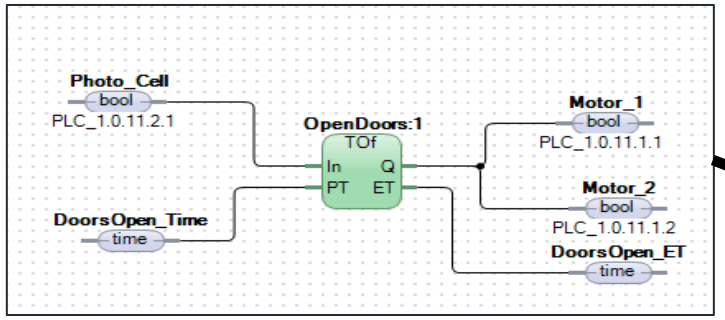
3. Open/ close valves

2. Compute control signal

Control system

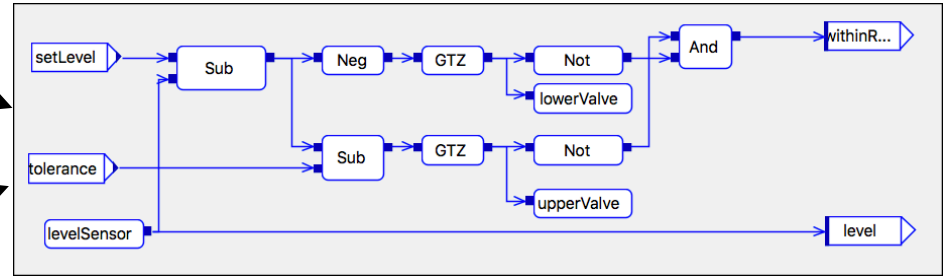
Inspiration

ABB Control Builder



- Directed data-flow
- Execution model
- Simplified examples

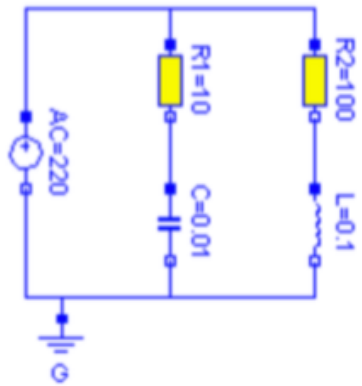
Bloqqi



Prototype language to explore different language constructs for code reuse.

Modelica

```
model circuit
  Resistor R1(R=10);
  Capacitor C(C=0.01);
  Resistor R2(R=100);
  Inductor L(L=0.1);
  VsourceAC AC;
  Ground G;
equation
  connect(AC.p, R1.p);
  connect(R1.n, C.p);
  connect(C.n, AC.n);
  connect(R1.p, R2.p);
  connect(R2.n, L.p);
  connect(L.n, C.n);
  connect(AC.n, G.p);
end circuit;
```



- Textual and visual syntax
- Inheritance
- Block redeclare

Language Constructs

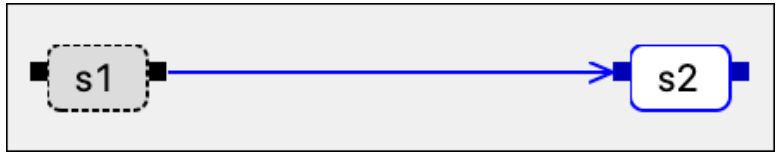
- Diagram inheritance
  - Connection interception
  - Block redeclare
  - Multiple inheritance
- Feature specifications
- State machines (simple)

Diagram Inheritance

A


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

B **extends** A




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

A subtype:

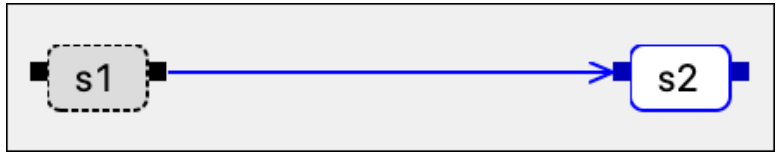
- Inherits all elements from its supertypes (depicted as grey/dotted)
- Can declare new elements: parameters, blocks, variables, connections (depicted as blue/solid)
- Can specialize even more: source/target connection intercept, block redeclare

(Target) Connection Interception

A

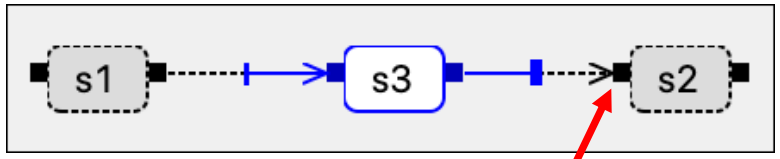

```
diagramtype A {
  s1: S;
}
```

B **extends** A




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

C1 **extends** B




```
diagramtype C1 extends B {
  s3: S;
  intercept s2.in with s3.in,s3.out;
}
```

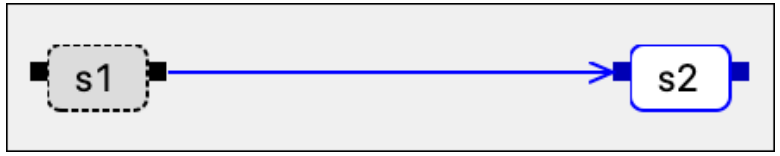
Connection to port **in** on **s2**
is *intercepted*
with a new block **s3**

Block Redeclare

A

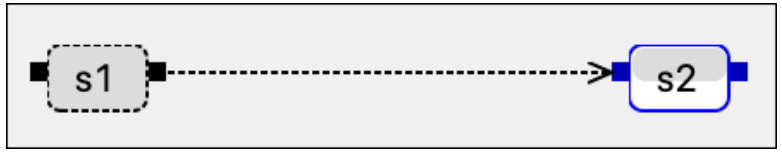

```
diagramtype A {
  s1: S;
}
```

B **extends** A




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

C2 **extends** B




```
diagramtype C2 extends B {
  redeclare s2: T;
}
```

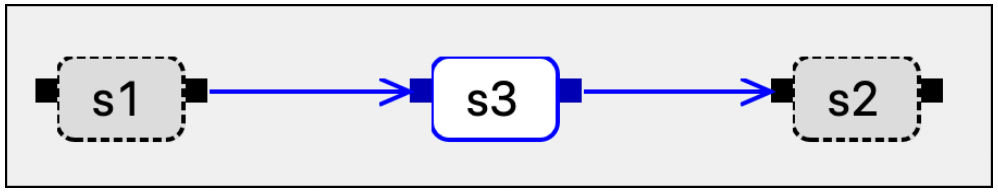
Block type for **s2** is
redeclared from **S** to **T**
(where T is a subtype of S)

Multiple Inheritance

Y1

Y2


```
Z extends Y1, Y2
```



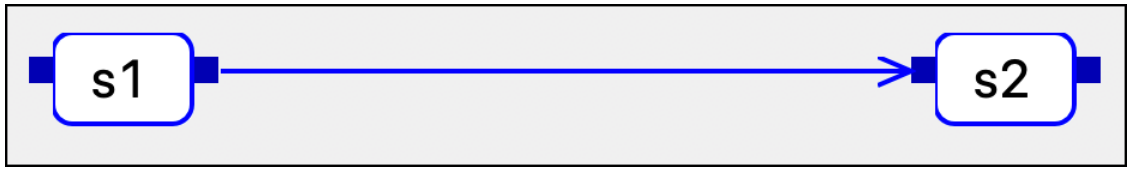
A diagram type can inherit from multiple supertypes.

The order Y1, Y2 matters in some cases.

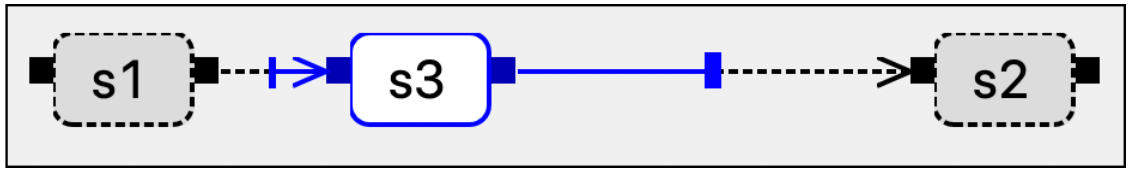
It’s fine if two blocks with the same names are added from different supertypes (names are prefixed).

The Diamond Problem

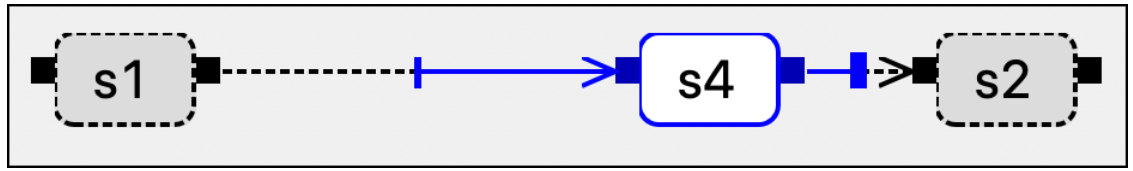
X



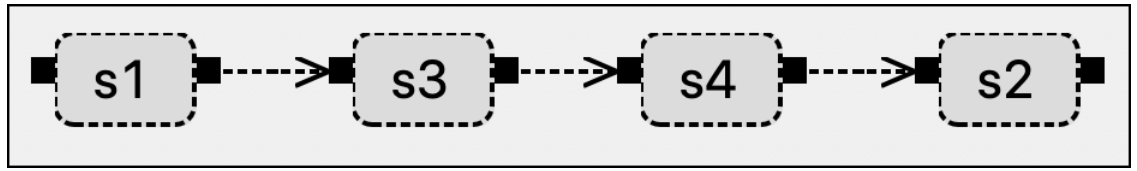
`Y1 extends X`



`Y2 extends X`



`Z extends Y1, Y2`



Block **s3** is before **s4** because
Z is extended with first **Y1** and then **Y2**

Elements are added in order: X, Z1, Z2, Z (a linearization order)

Linearization Order

Formally, the type hierarchy for a diagram type d with acyclic supertypes $b_1, b_2, ..., b_n$ is linearized as the following sequence:

$$\mathrm{L}(d) = \mathrm{L}(b_1) \oplus \mathrm{L}(b_2) \oplus ... \oplus \mathrm{L}(b_n)\ ,\ d$$

where

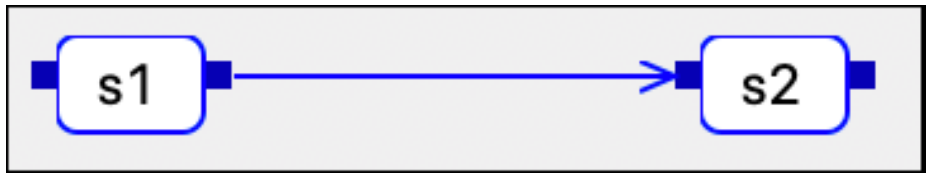
$$A \oplus (b, B) = \begin{cases} (A, b) \oplus B & \text{if } b \notin A \\ A \oplus B & \text{if } b \in A \end{cases}$$

$$A = a_1, ..., a_n \text{ and } B = b_1, ..., b_m$$

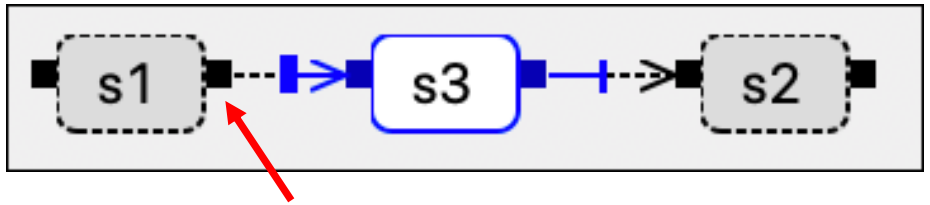
The operator $\oplus$ combines two sequences. It removes duplicates and favors elements on the left-hand side.

Source Interception

X

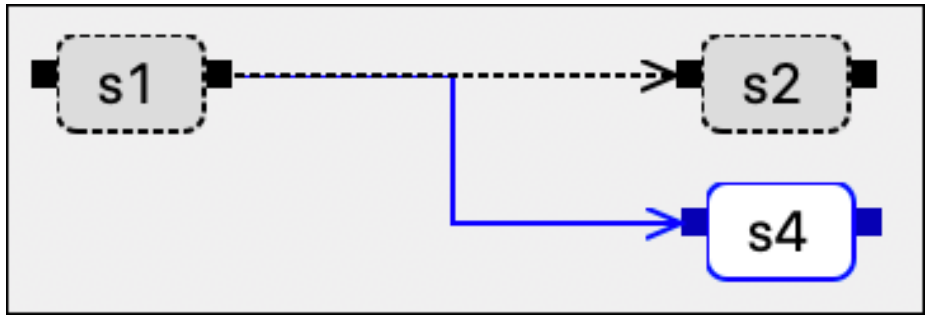


`Y1 extends X`

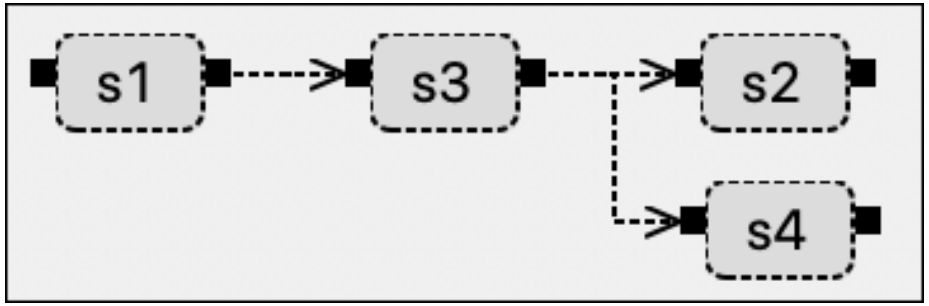


Intercepts **s1.out** (*source* port instead of *target* port)

`Y2 extends X`



`Z extends Y1, Y2`



This results in that **s4** is connected to **s3** instead of **s1**

Subtypes

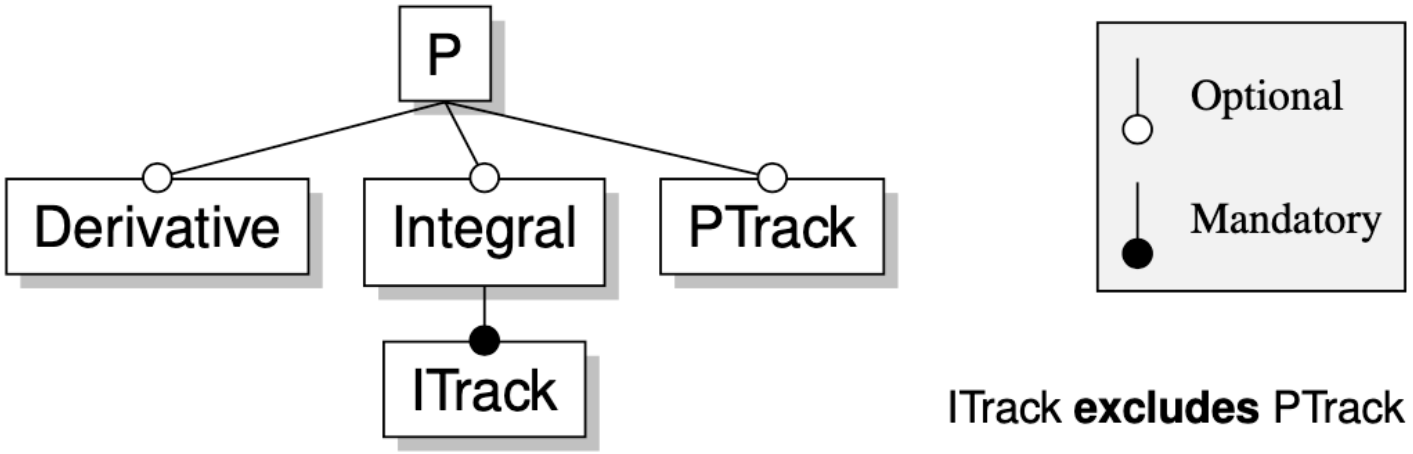
How do two subtypes **A** and **B** relate?

- Can they be combined or are they alternatives?
- If they are combined, in which order should they be applied?
- The number of combinations grows very fast
  - Only a small number of combinations can be anticipated

Feature Specifications

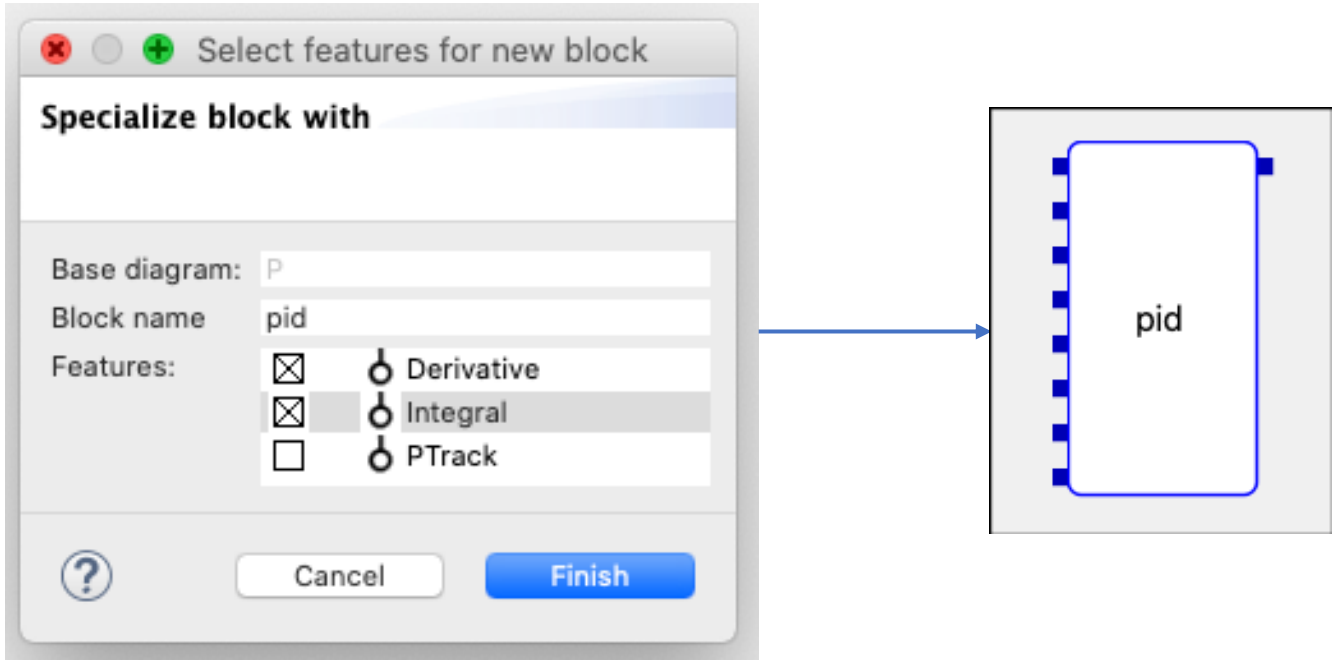
- Capturing how subtypes can be combined in **feature specifications**, where each subtype is viewed as a **feature**
- **Ordering statements** are required when two features interact
- A **variant** is a selection of features
- Generation of **feature wizards**
- *(Easier specification than previously and supports features consisting of several blocks)*

PID Example (by Alfred Theorin)



Feature diagram

Feature Selection in Feature Wizard

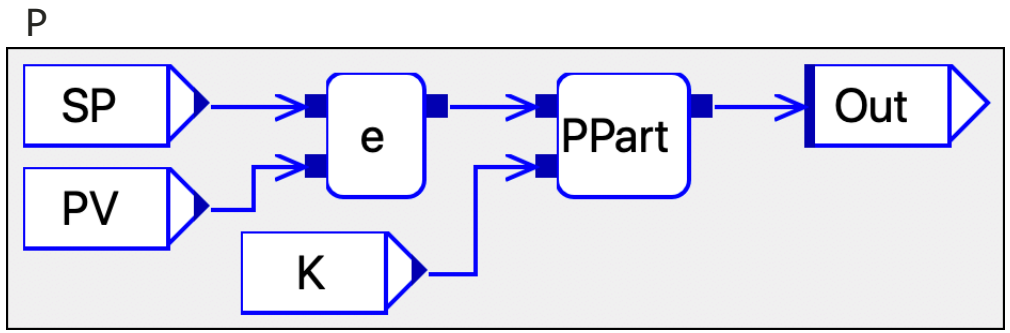


Generated Wizard

Specification Steps

1. Define a base diagram (P)
2. Define all features as separate subtypes (PD, PI, PTrack, …)
3. Specify the subtypes as features and specify the order between features if needed

P (base type)

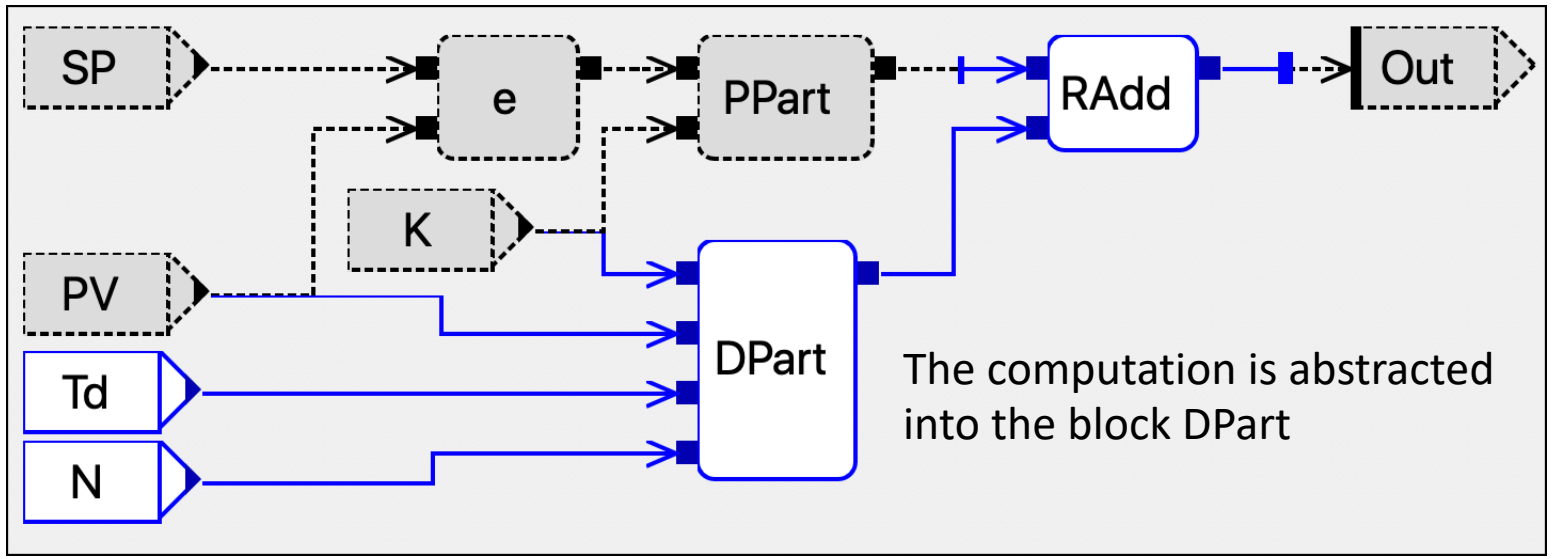



```
diagramtype P(SP: Real, PV: Real,
              K: Real => Out: Real) {
  e: RSub;
  PPart: RMul;
  connect(SP, e.in1);
  connect(PV, e.in2);
  connect(e.out, PPart.in1);
  connect(K, PPart.in2);
  connect(PPart.out, Out);
}
```

SP, PV, …, are parameters

PD

PD **extends** P

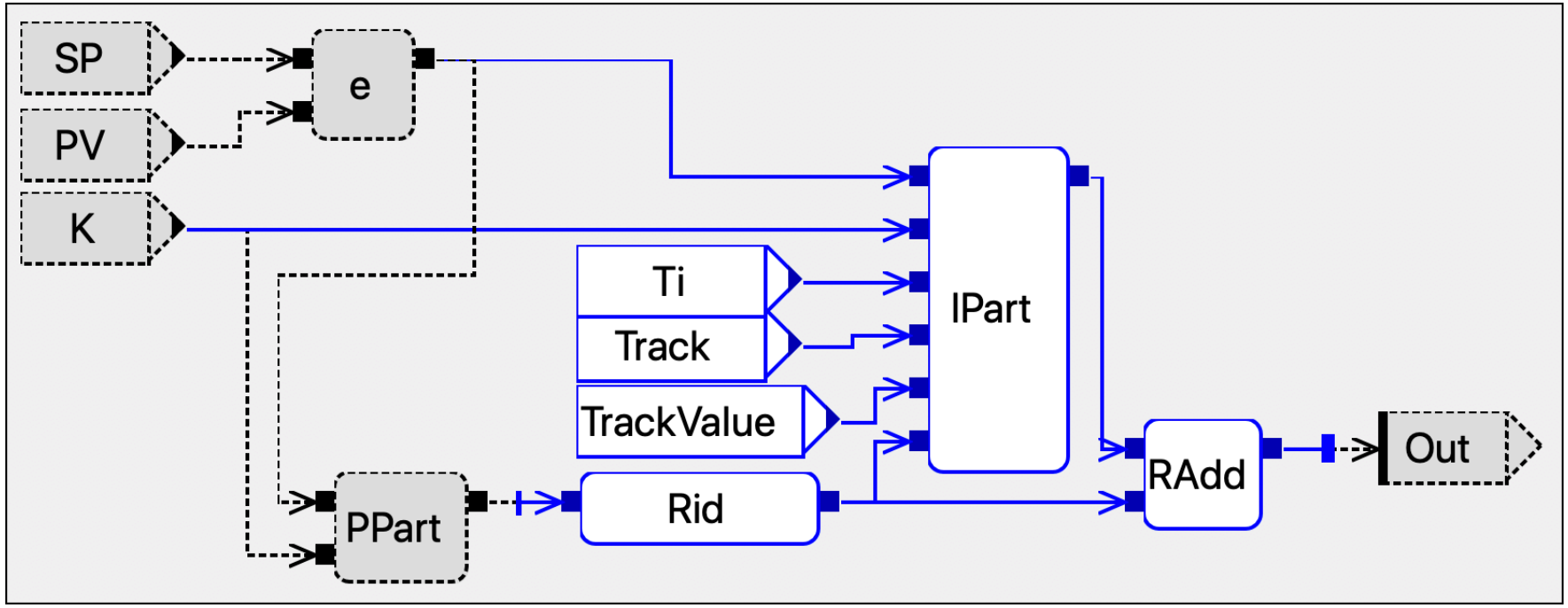


Intercepts parameter **Out**

```
diagramtype PD(Td: Real, N: Real) extends P {
  DPart: DPart;
  RAdd: RAdd;
  connect(PV, DPart.PV);
  ...
  intercept Out with RAdd.in1, RAdd.out;
}
```

PI

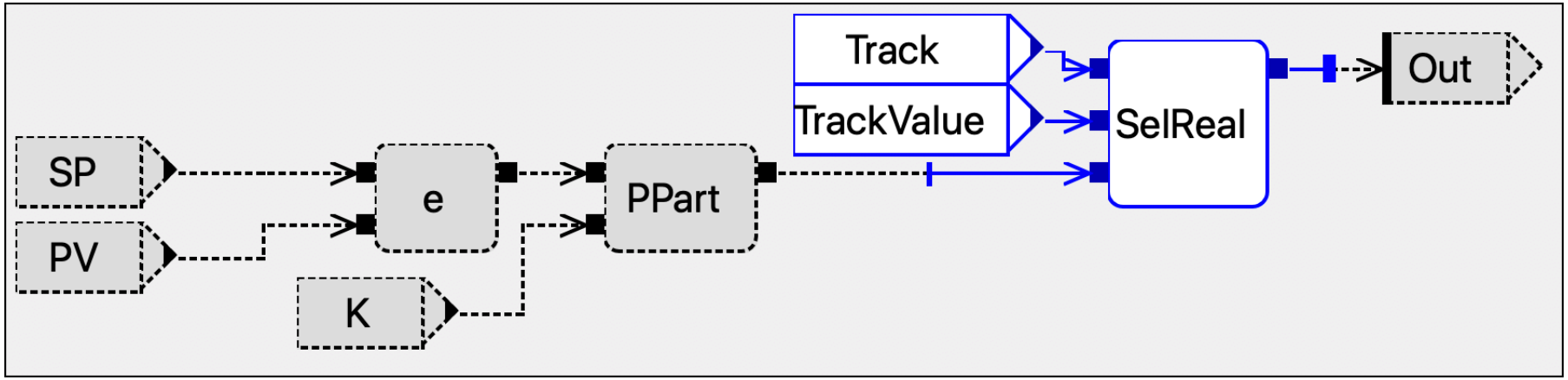
```
PI extends D
```



Intercepts parameter **Out**

PTrack

```
PTrack extends P
```

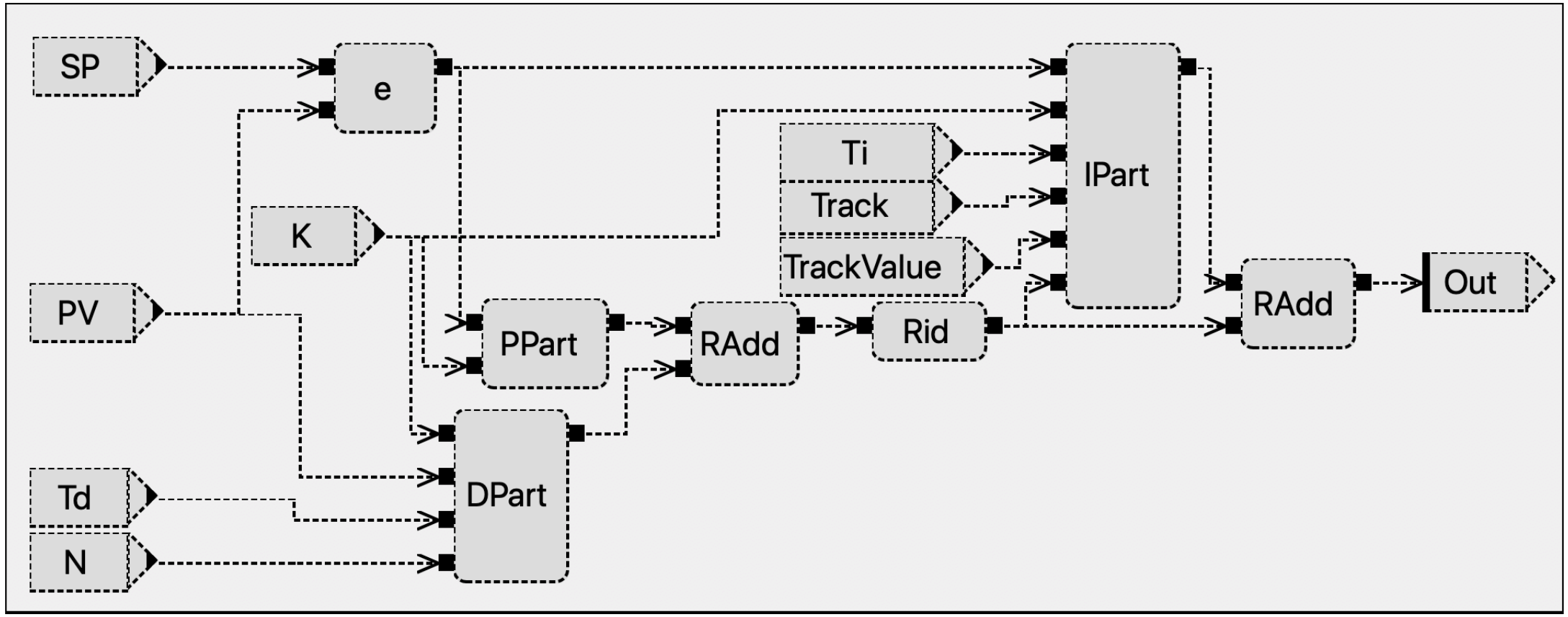


Intercepts parameter **Out**

PID

Since both **PD** and **PI** intercept **Out**,
the order in which they are inherited matters

```
PID extends PD, PI
```



PD needs to be before PI because PI has tracking

How do we know in which order supertypes should be inherited?

Feature Specifications

Optional features

```
features P {
  optional Derivative: PD;
  optional Integral: PI;
  optional PTrack: PTrack;
}
```

Optional features with feature names and subtypes where they are defined

Excluding features

```
features P {
  PTrack excludes Integral;
}
```

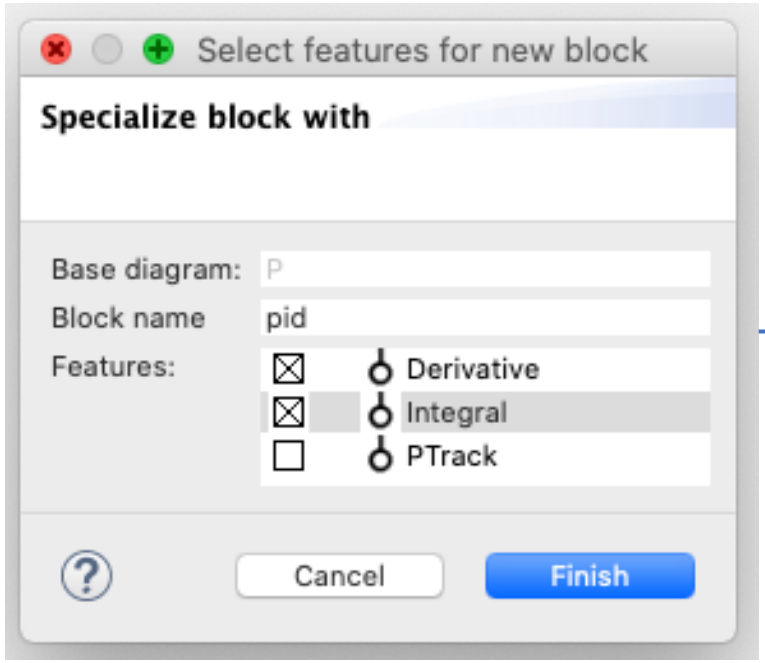
Both **PTrack** and **Integral** have tracking, so they can't be used at the same time

Ordering features

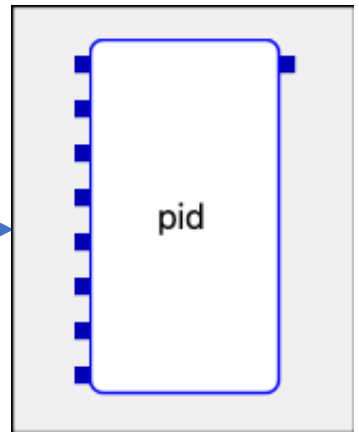
```
features P {
  Derivative before Integral;
  Derivative before PTrack;
}
```

We need to order all features that intercept the same ports

Feature Instantiation



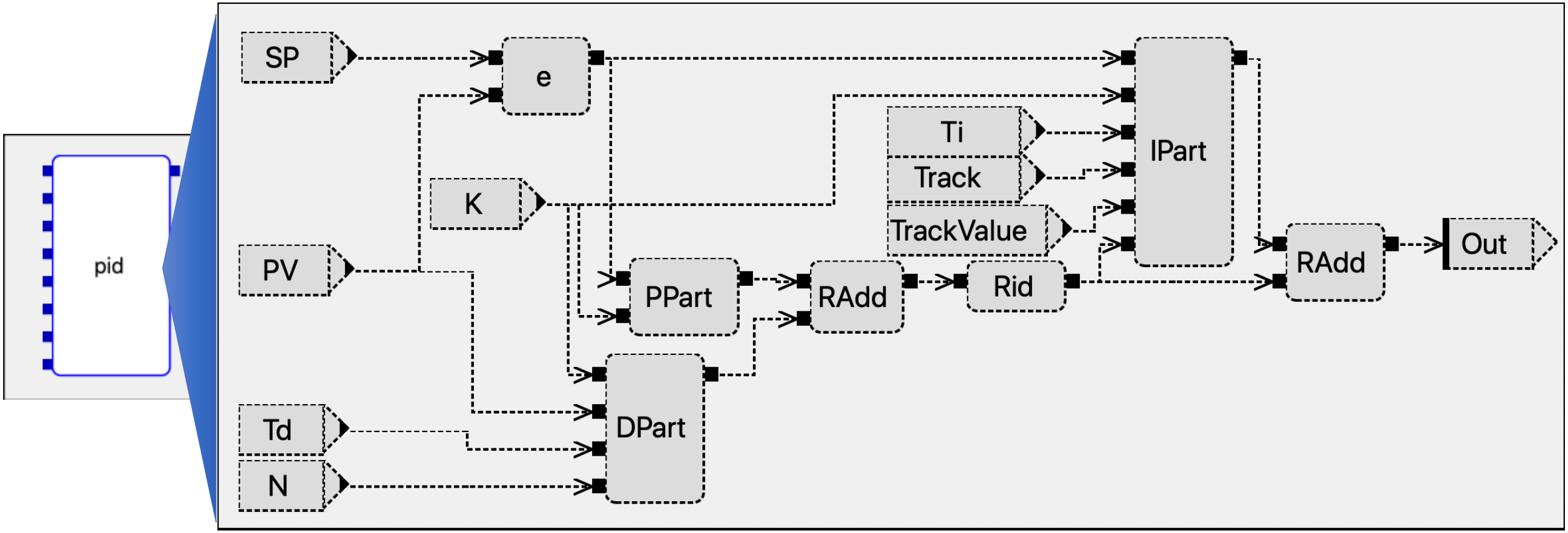
Wizard

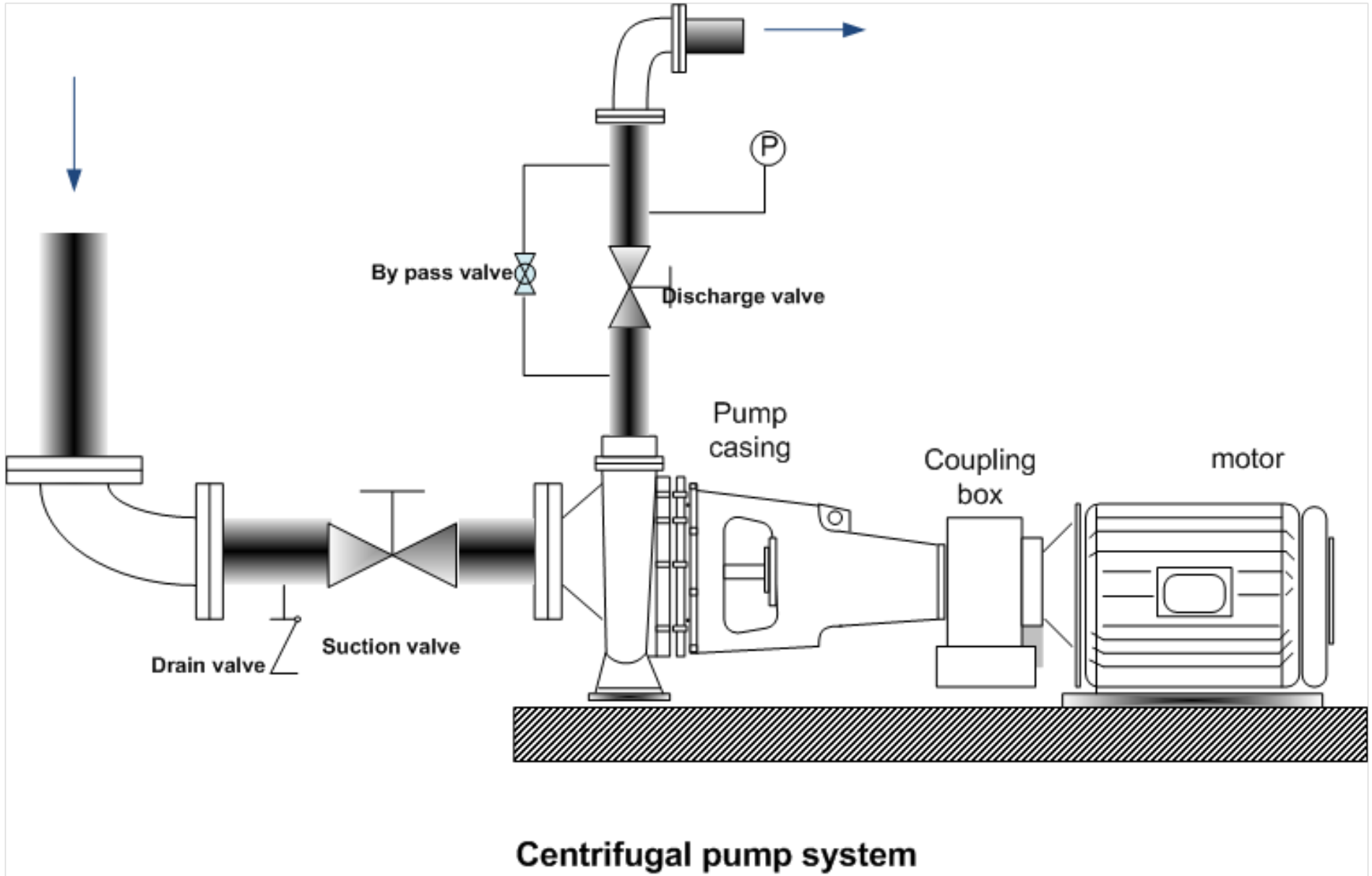



```
diagramtype Main {
  pid: P {
    feature Derivative;
    feature Integral;
  };
}
```

A block **pid** that has the base type **P**, and with the features **Derivative** and **Integral**

Feature Instantiation





Example from Ulf Hagberg

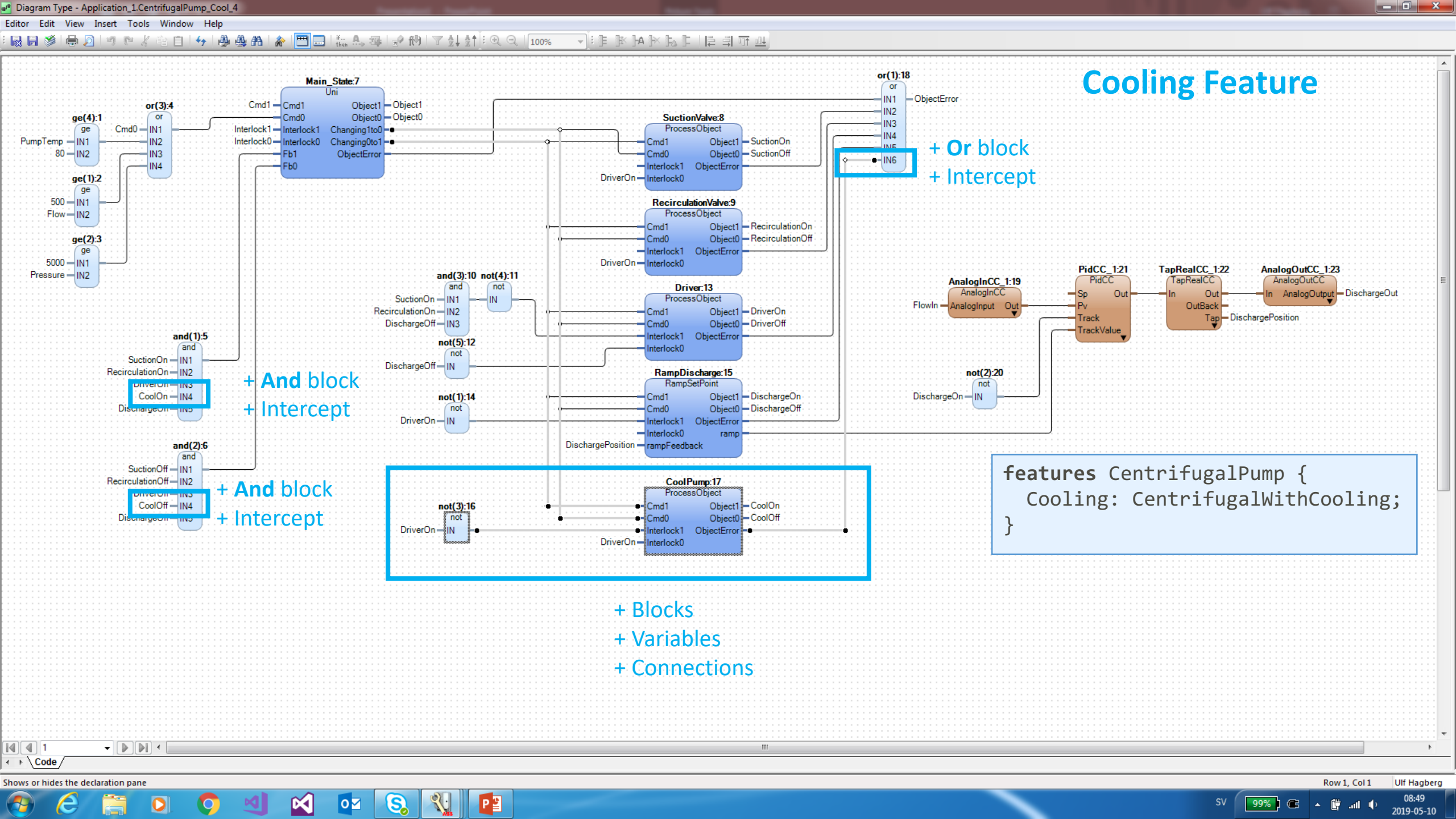


Diagram Type - Application_1.CentrifugalPump_Cool_4
Editor Edit View Insert Tools Window Help
Cooling Feature
Main_State:7
Uni
ge(4):1
ge(1):2
ge(2):3
or(3):4
and(1):5
and(2):6
SuctionValve:8
ProcessObject
RecirculationValve:9
Driver:13
and(3):10
not(4):11
not(5):12
not(1):14
RampDischarge:15
RampSetPoint
not(3):16
CoolPump:17
or(1):18
+ Or block
+ Intercept
+ And block
+ Intercept
+ And block
+ Intercept
AnalogInCC_1:19
PidCC_1:21
TapRealCC_1:22
AnalogOutCC_1:23
not(2):20
features CentrifugalPump {
Cooling: CentrifugalWithCooling;
}
+ Blocks
+ Variables
+ Connections
Code
Shows or hides the declaration pane
Row 1, Col 1
Ulf Hagberg
08:49
2019-05-10

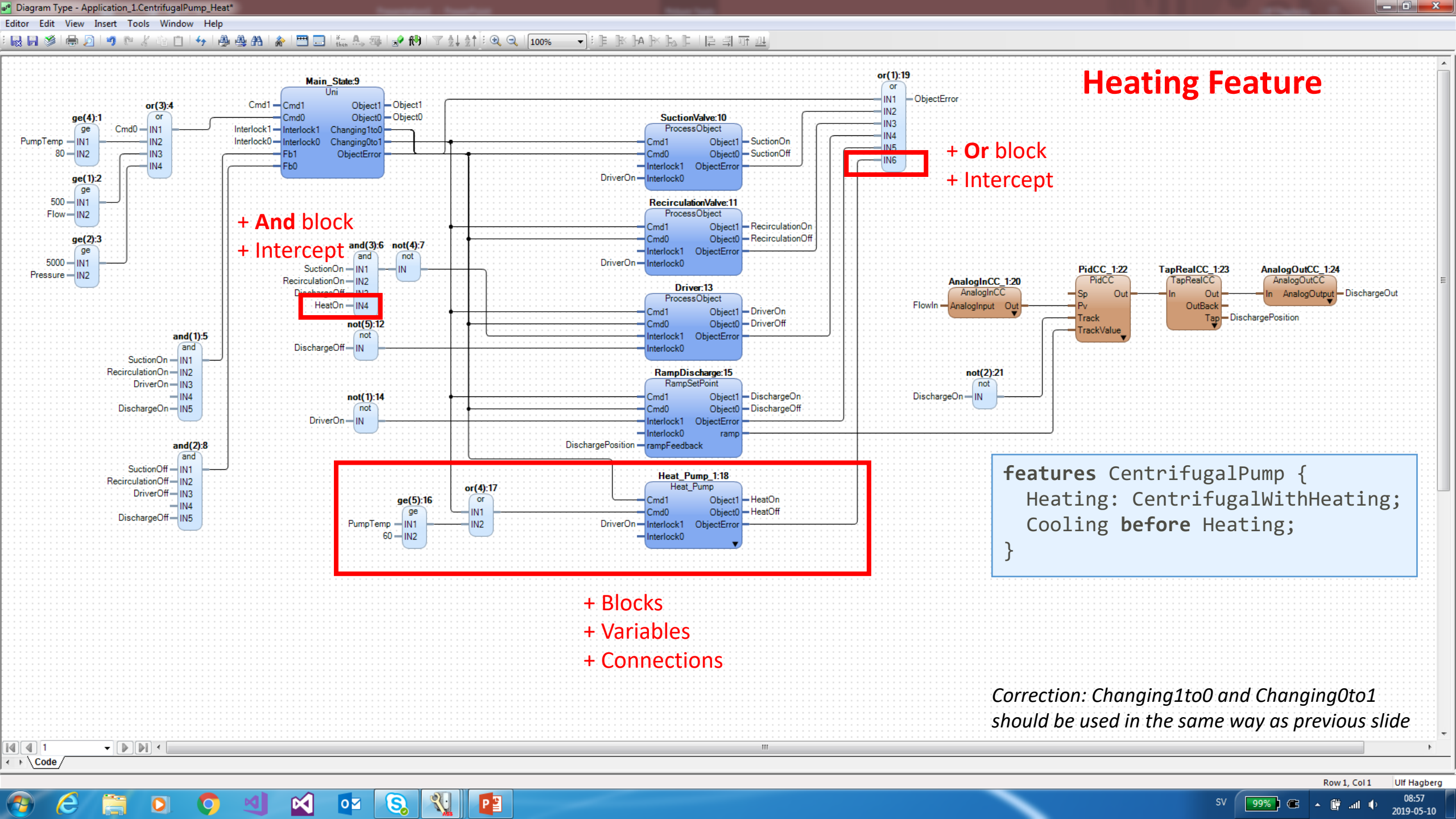


Diagram Type - Application_1.CentrifugalPump_Heat*
Editor Edit View Insert Tools Window Help
Heating Feature
+ Or block
+ Intercept
+ And block
+ Intercept
+ Blocks
+ Variables
+ Connections
features CentrifugalPump {
Heating: CentrifugalWithHeating;
Cooling before Heating;
}
Correction: Changing1to0 and Changing0to1 should be used in the same way as previous slide
Main_State:9
or(1):19
SuctionValve:10
RecirculationValve:11
Driver:13
RampDischarge:15
Heat_Pump_1:18
ge(4):1
or(3):4
ge(1):2
ge(2):3
and(3):6
not(4):7
not(5):12
and(1):5
not(1):14
and(2):8
ge(5):16
or(4):17
AnalogInCC_1:20
PidCC_1:22
TapRealCC_1:23
AnalogOutCC_1:24
not(2):21
Code

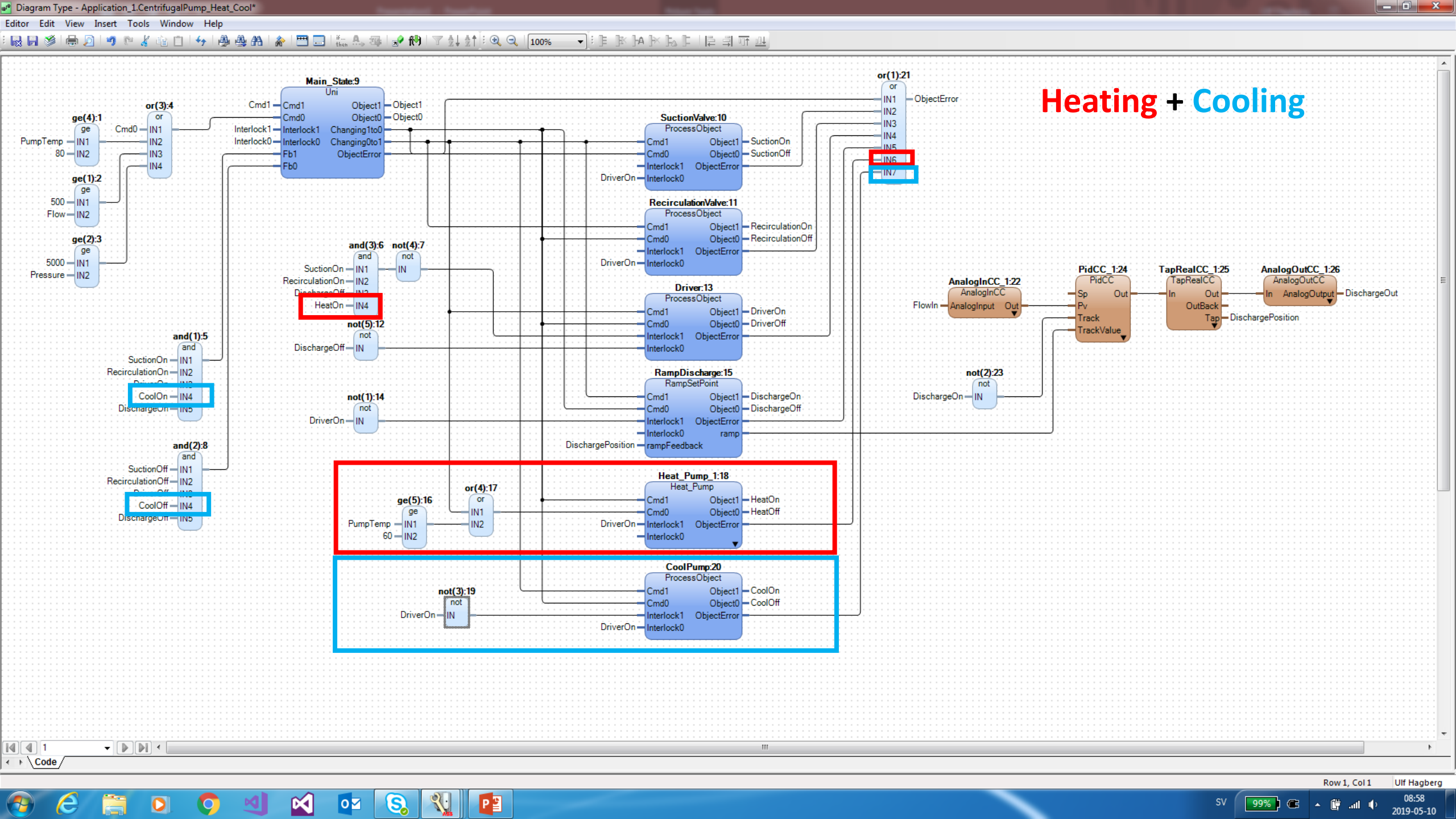
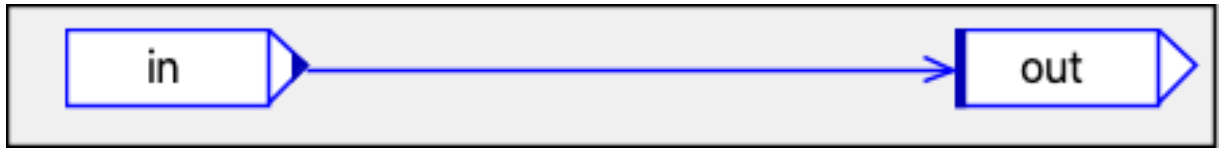


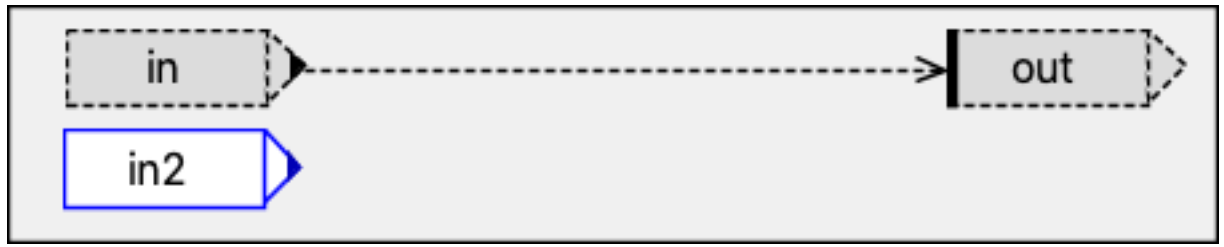
Diagram Type - Application_1.CentrifugalPump_Heat_Cool*
Editor Edit View Insert Tools Window Help
Heating + Cooling
Main_State:9
Uni
SuctionValve:10
ProcessObject
RecirculationValve:11
Driver:13
RampDischarge:15
RampSetPoint
Heat_Pump_1:18
Heat_Pump
CoolPump:20
or(1):21
or(3):4
ge(4):1
ge(1):2
ge(2):3
and(3):6
not(4):7
not(5):12
and(1):5
not(1):14
and(2):8
or(4):17
ge(5):16
not(3):19
not(2):23
AnalogInCC_1:22
AnalogInCC
PidCC_1:24
PidCC
TapRealCC_1:25
TapRealCC
AnalogOutCC_1:26
AnalogOutCC
Code
Row 1, Col 1
Ulf Hagberg
08:58
2019-05-10

Abstract Common Functionality

`Base`

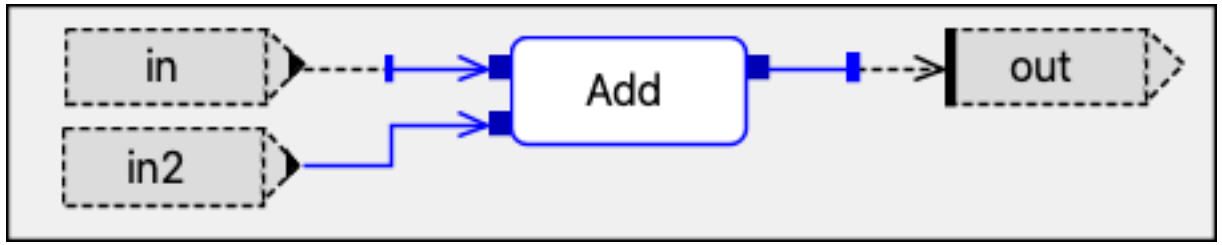


`abstract` `AbstractBase` **`extends`** `Base`

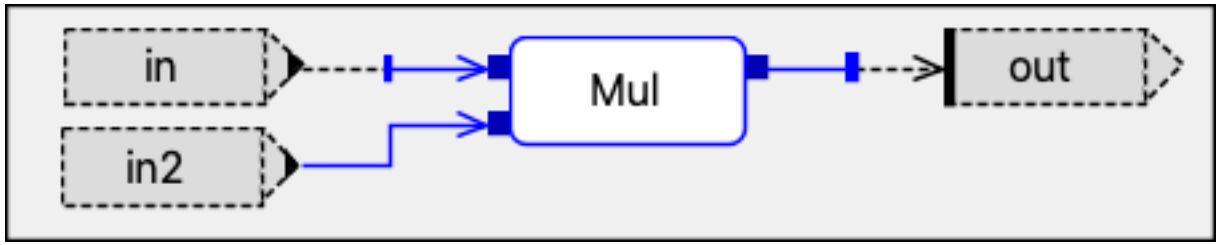


Extract functionality to common supertype.
Other subtypes might not need this functionality.

`BaseAdd` **`extends`** `AbstractBase`



`BaseMul` **`extends`** `AbstractBase`



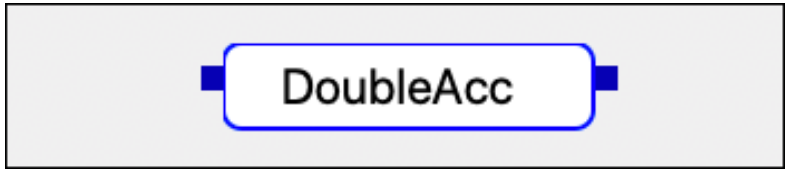
Simple State Machines

```
statemachine DoubleAcc(enable: Bool => out: Int) {
  var x: Int;
  // States
  normal {
    out = x;
    x   = x + 1;
  }
  double {
    out = x * 2;
    x   = x + 1;
  }
  // Transitions
  normal => double: enable;
  double => normal: !enable;
}
```

An accumulator state machine that returns double the value if in the double state.

A state can be prefixed with **public**, which creates an extra output parameter returning if the machine is in that state

Main




```
diagramtype Main {
  DoubleAcc: DoubleAcc;
}
```

Instantiation as a block

Execution of Bloqqi Programs

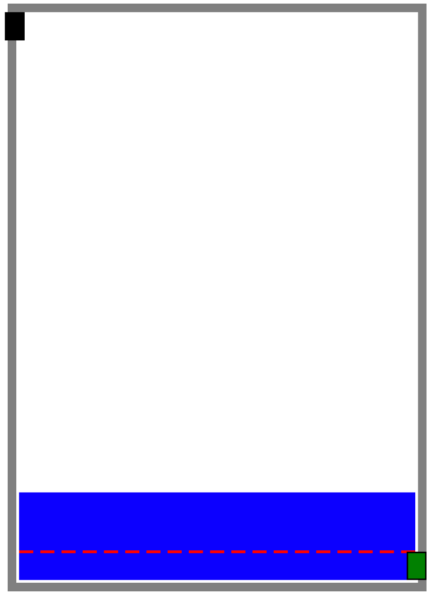
The Bloqqi compiler generates C code

- Easy to run on different platforms (e.g., embedded systems)
- Distributed execution over MQTT (publish/subscribe middleware)

The compiler also generates Functional Mock-up Units (FMUs):

- Defined in standard Functional Mock-up Interface (FMI)
- Standard for connecting simulation models defined in different tools
- Bloqqi compiler generates Co-Simulation FMUs

MQTT



Web Browser (visualization)

sub tank/upperValve
sub tank/lowerValve
sub tank/level
(over WebSocket)

Regulator

pub tank/upperValve
pub tank/lowerValve

sub tank/level

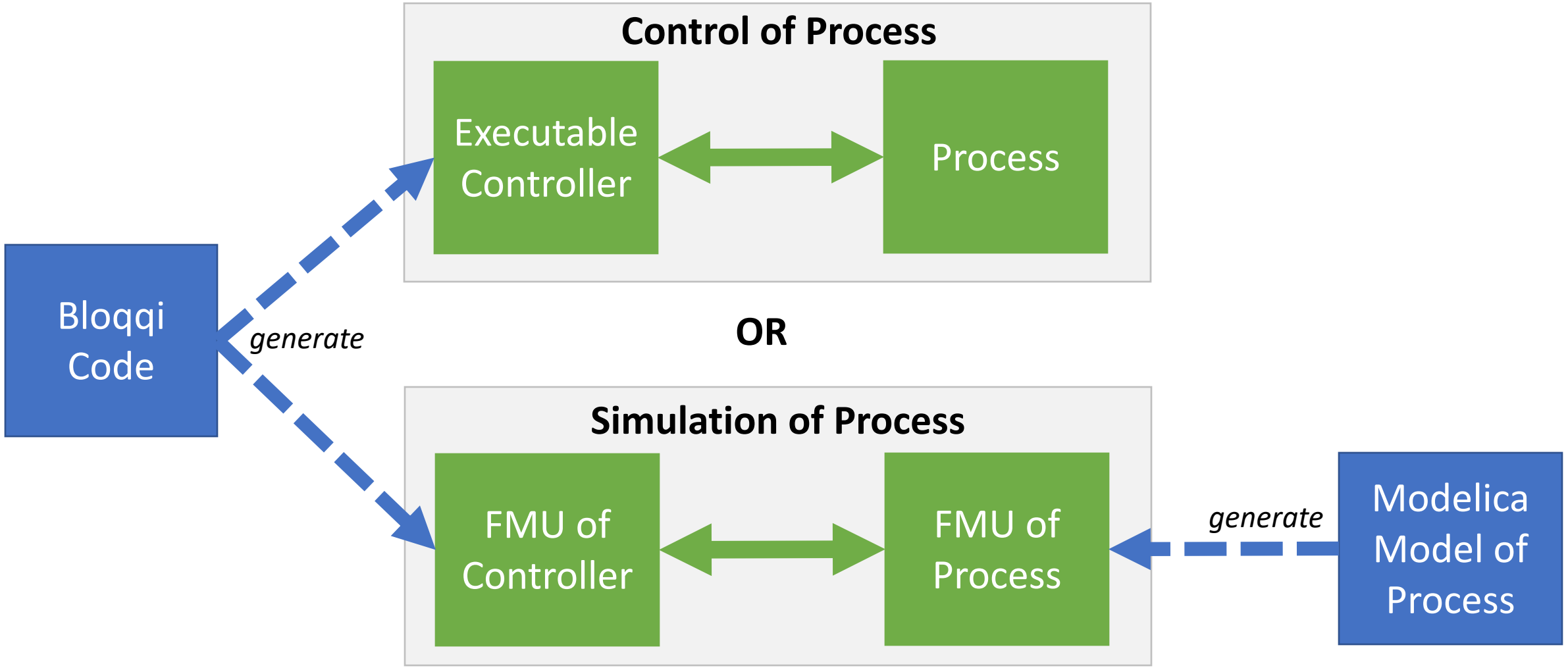
Broker

pub tank/level

sub tank/upperValve
sub tank/lowerValve

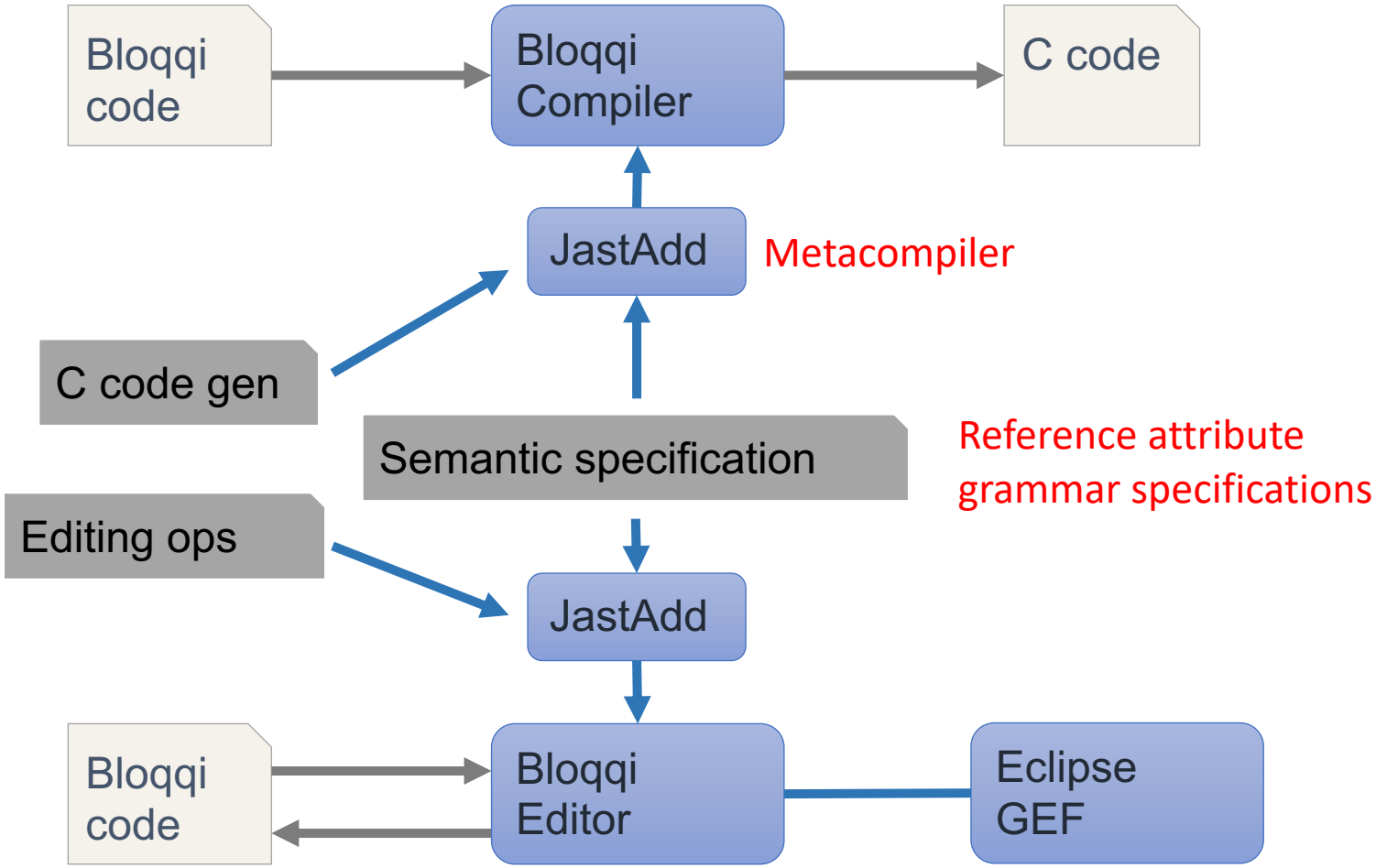
Tank (simulation)

Test Control Program by Simulating Process



FMUs can be connected using SSP, for example, using OMSimulator or FMI Composer

Modular Tool Implementation



Conclusions

- Bloqqi
  - Prototype language for exploring code reuse constructs
  - Inheritance, connection interception, block redeclare
  - Features and Feature wizards
  - Support for state machines
  - FMI integration for simulating process
- Current/Future work
  - Add support of nested features
  - More examples
  - Hopefully use FMI 3.0 to communicate events to master algorithm