

# MODEL EXECUTION

*BUILD YOUR OWN COMPILER AND VIRTUAL MACHINE*

---

MASTER 1 ICE, 2020-2021

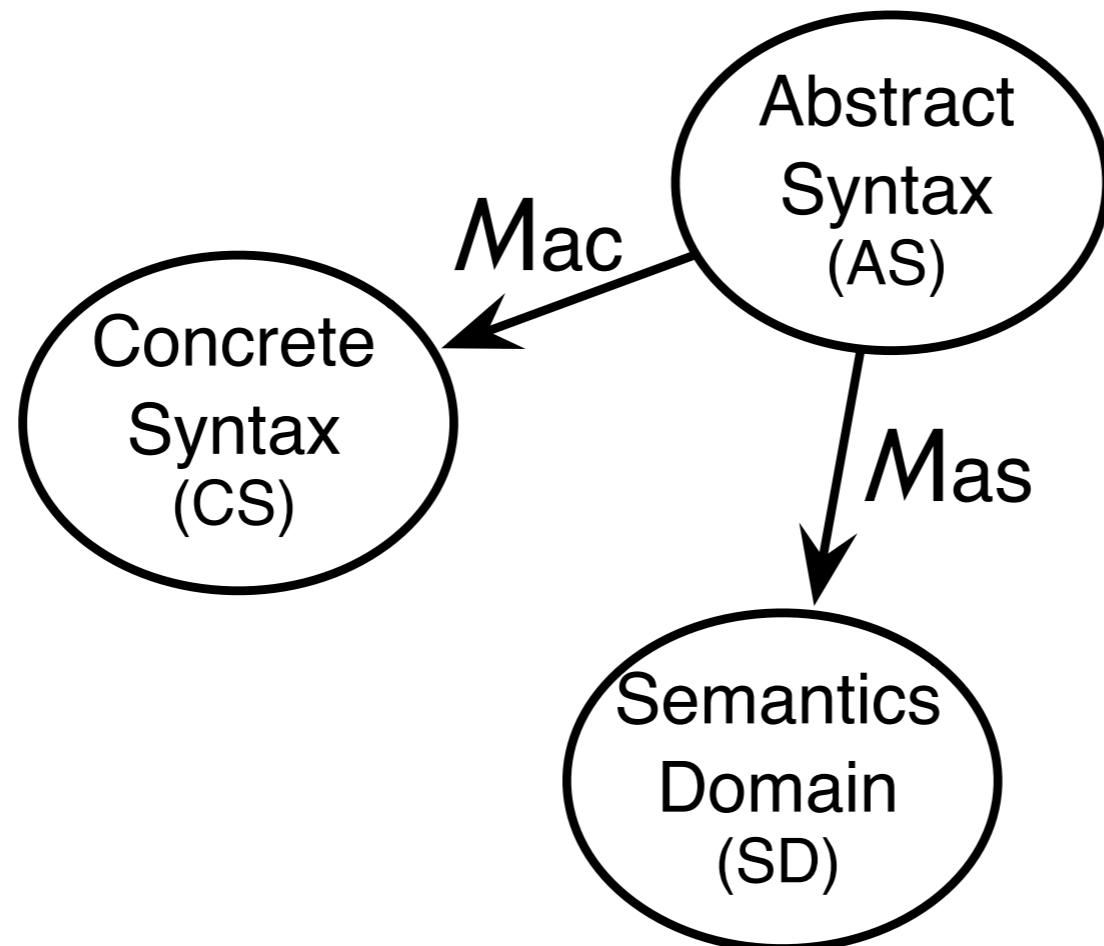
**BENOIT COMBEMALE**  
PROFESSOR, UNIV. RENNES 1 & INRIA, FRANCE

[HTTP://COMBEMALE.FR](http://COMBEMALE.FR)  
[BENOIT.COMBEMALE@IRISA.FR](mailto:BENOIT.COMBEMALE@IRISA.FR)  
[@BCOMBEMALE](https://twitter.com/bcombemale)

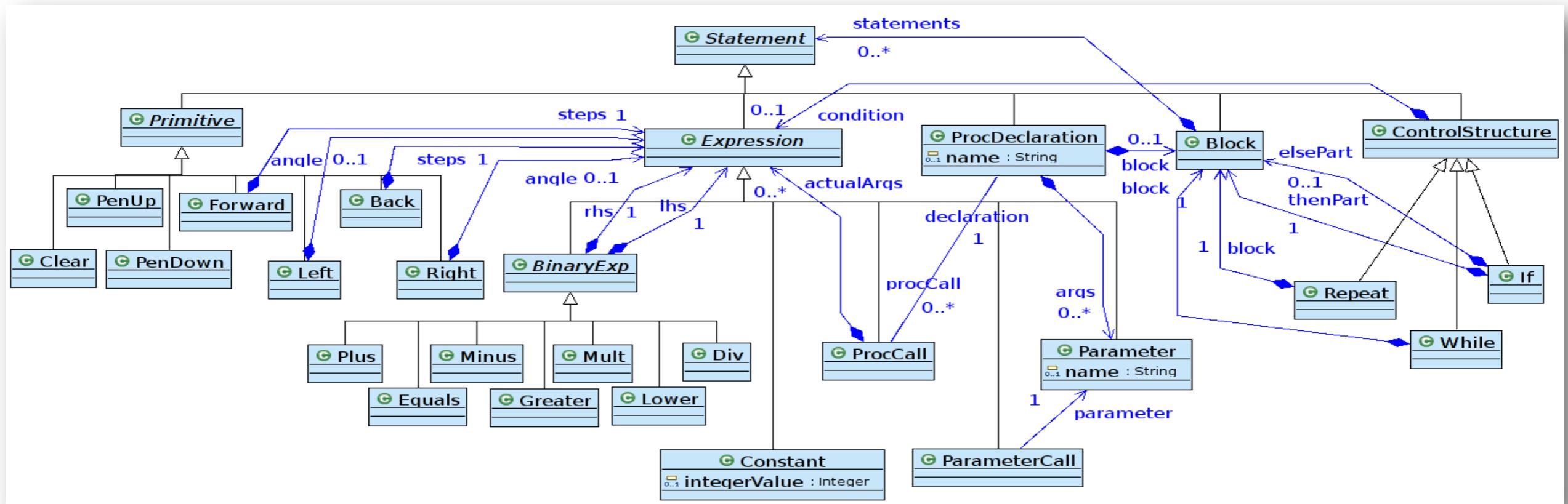


# Reminder about what is a language

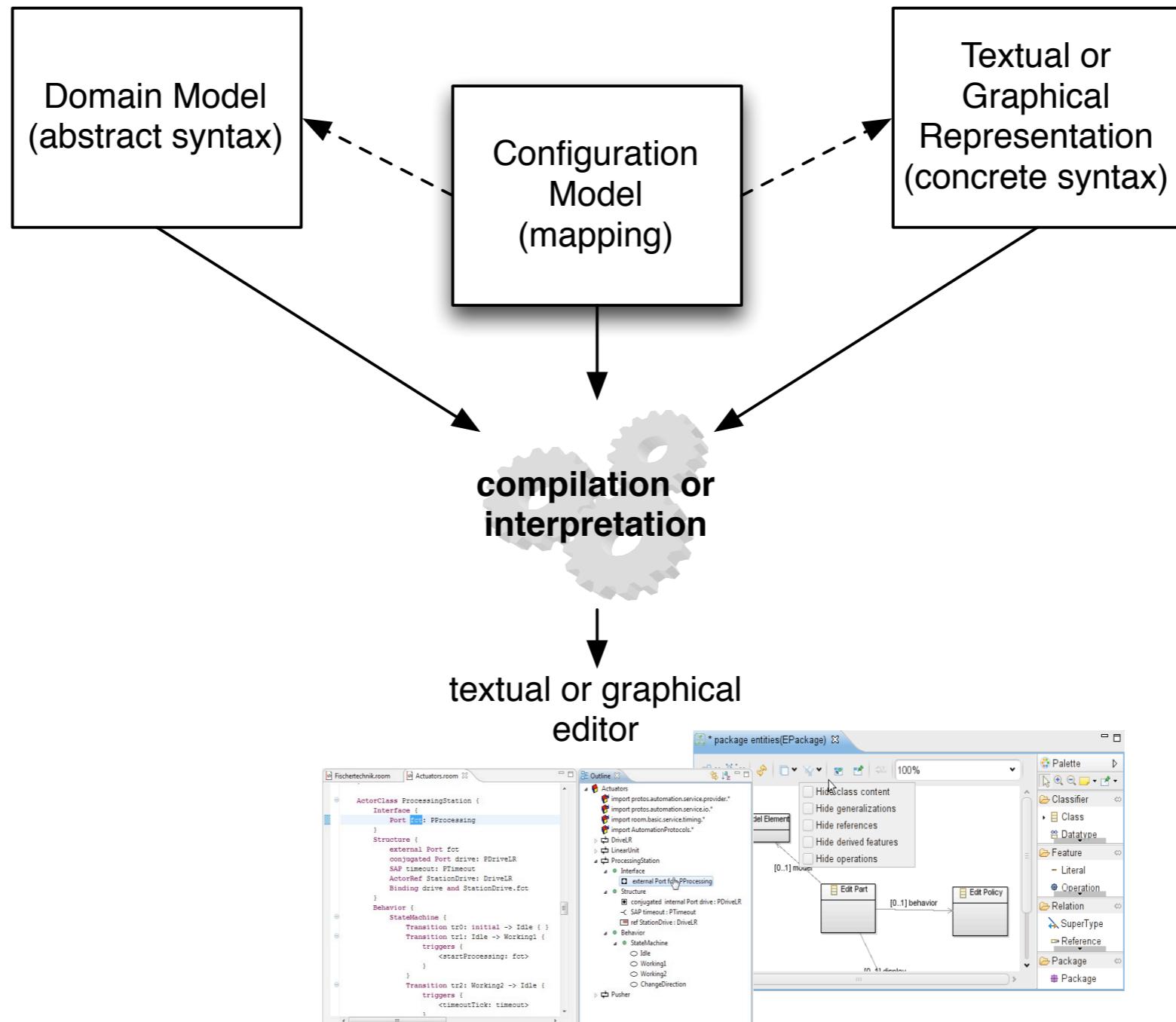
---



# Reminder about what is an abstract syntax



# Reminder about what is a concrete syntax



# Reminder about what is a semantics

---

- ▶ Any “meaning” given to the domain model
  - compiler, interpreter, analysis tool, refactoring tool, etc.
- ▶ Thanks to model transformations

program = data + algorithms ☺
- ▶ In practices?
  - It requires to “traverse” the domain model, and... do something!
  - Various languages, and underlying paradigms:
    - **Declarative** (rule-based): mostly for pattern matching (e.g., analysis, refactoring)
    - **Imperative** (visitor-based):
      - *interpreter pattern*: mostly for model interpretation (e.g., execution, simulation)
      - *template*: mostly for text generation (e.g., code/test/doc generators)

# Reminder of the previous lectures / labs

---

## Build your own (Domain-Specific) Language

1. Build your abstract syntax as a domain model with Ecore (possibly additional constraints with OCL, aka. context conditions)
2. Build your concrete syntax (*textual* with Xtext, *graphical* with Sirius)
3. Build your generators
  - ▶ Documentation generator
  - ▶ Code generator (/compiler)

# Objectives of the coming lecture/labs

---

4. Build your interpreter (/ VM)

5. Build your animator

Get your own modeling workbench with  
model edition, compilation, execution,  
simulation, (graphical) animation and debugging

# Definition of the Behavioral Semantics of DSL

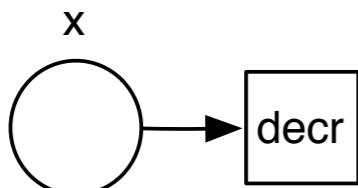
```
int x;  
void decr () {  
    if ( x>0 )  
        x = x-1;  
}
```

System
x : Int
decr()

## ► Axiomatic

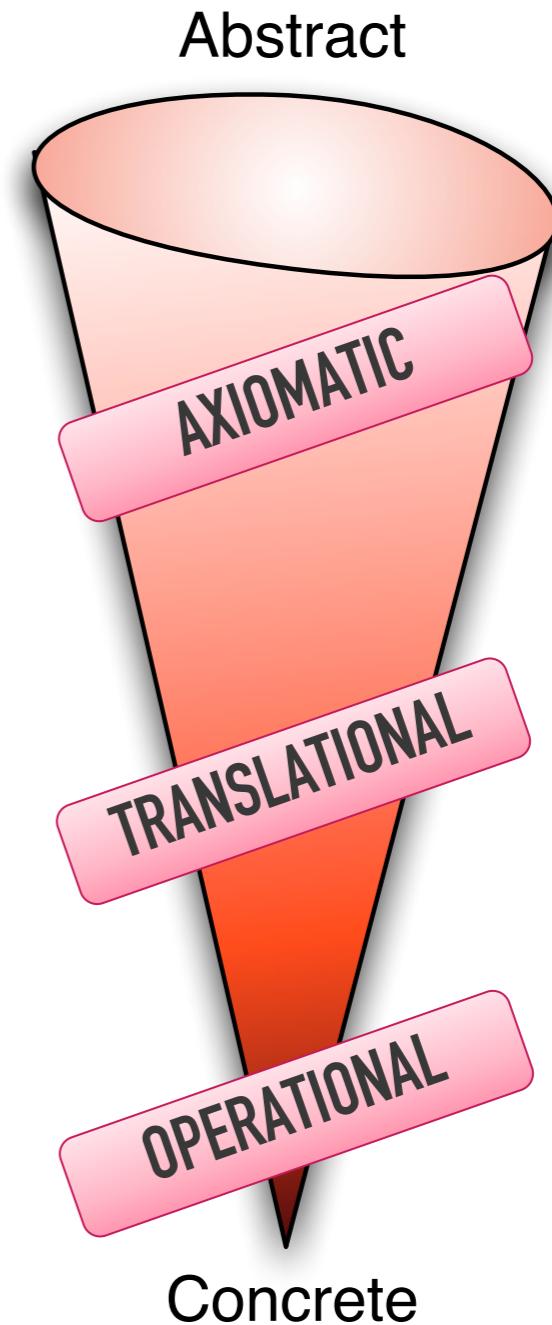
```
context System::decr() post :  
    self .x =  if ( self .x@pre>0 )  
                then self.x@pre - 1  
                else self.x@pre  
    endif
```

## ► Denotational/translational

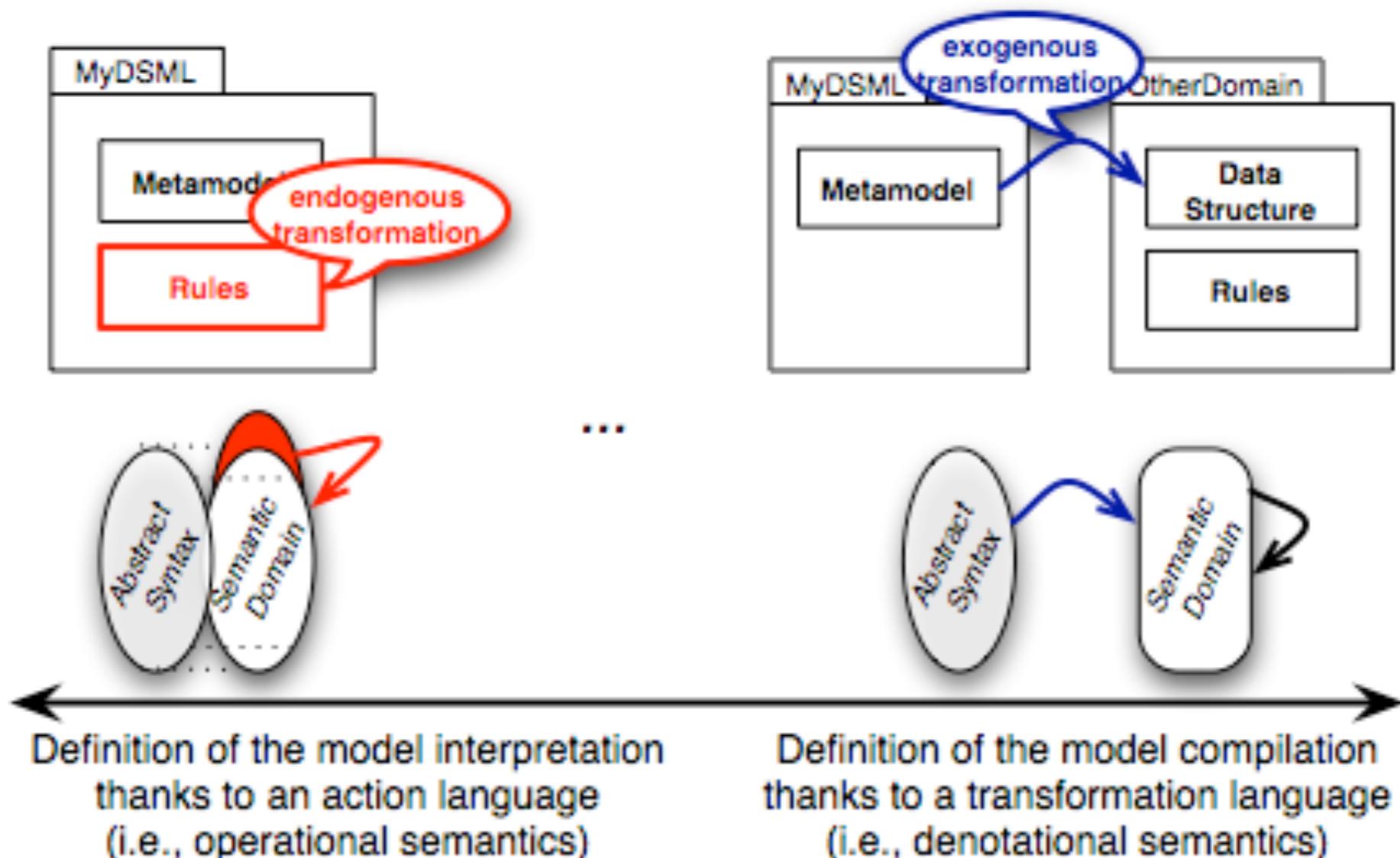


## ► Operational

```
operation decr () is do  
    if x>0 then x = x - 1 end
```

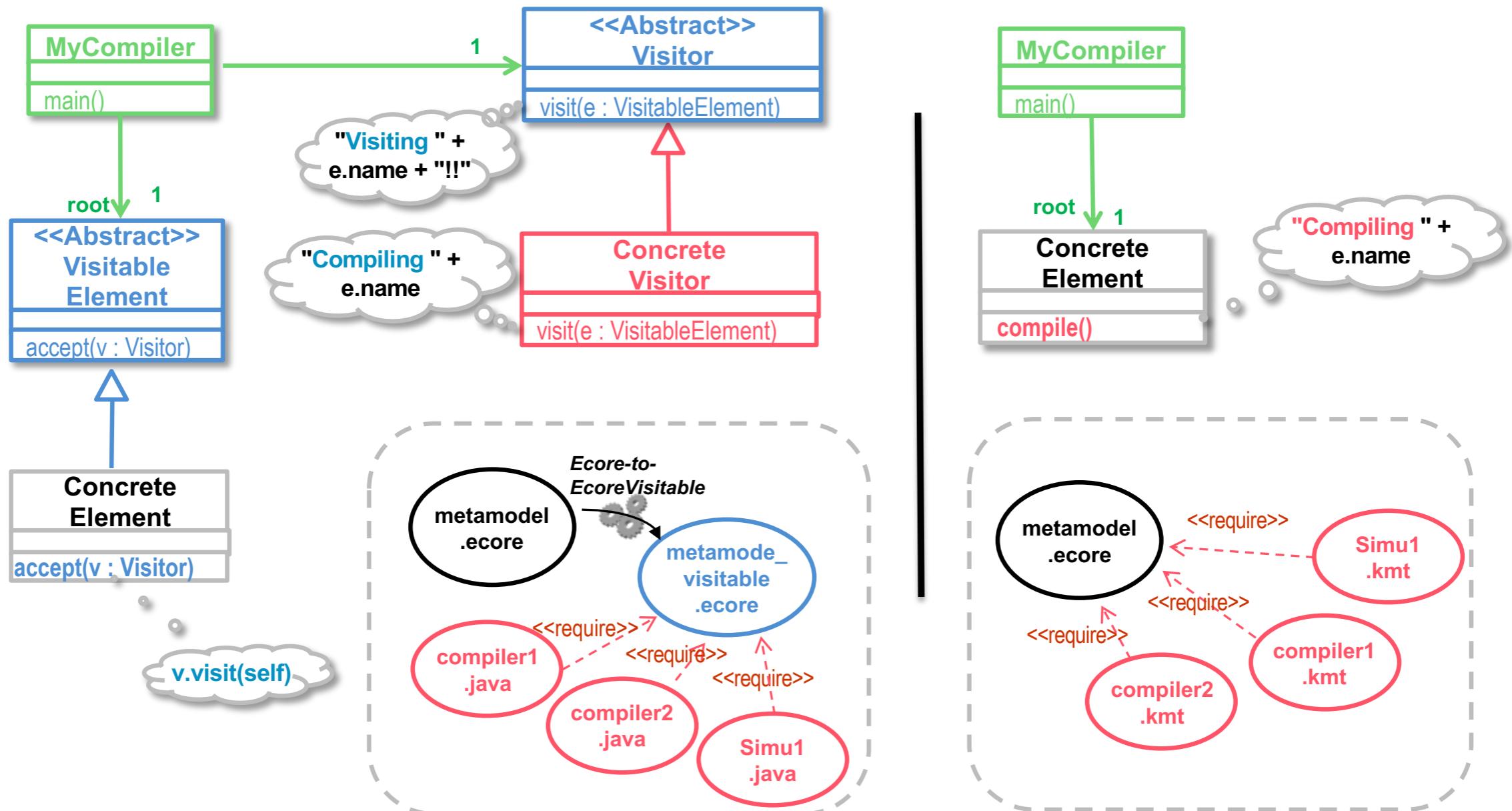


# Definition of the Behavioral Semantics of DSL

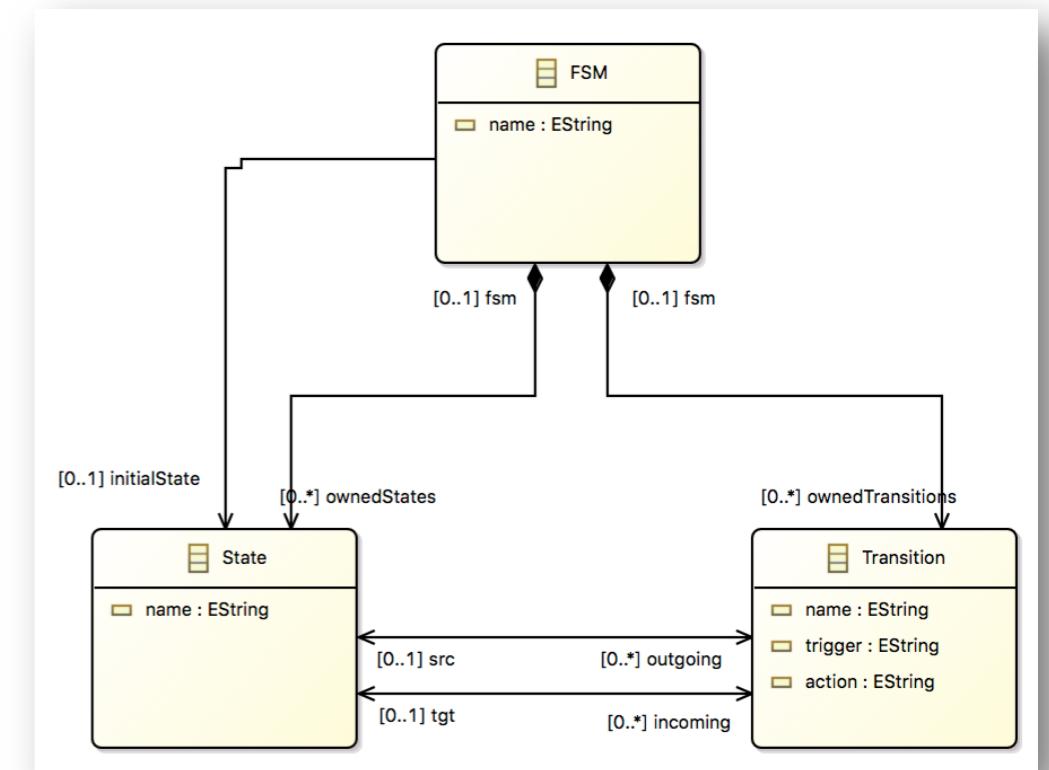


# Implement your own compiler / interpreter

- ▶ Visitor-based?
  - ▶ Interpreter/visitor patterns, static introduction (aka. open class)



# Implement your own compiler with Xtend/K3



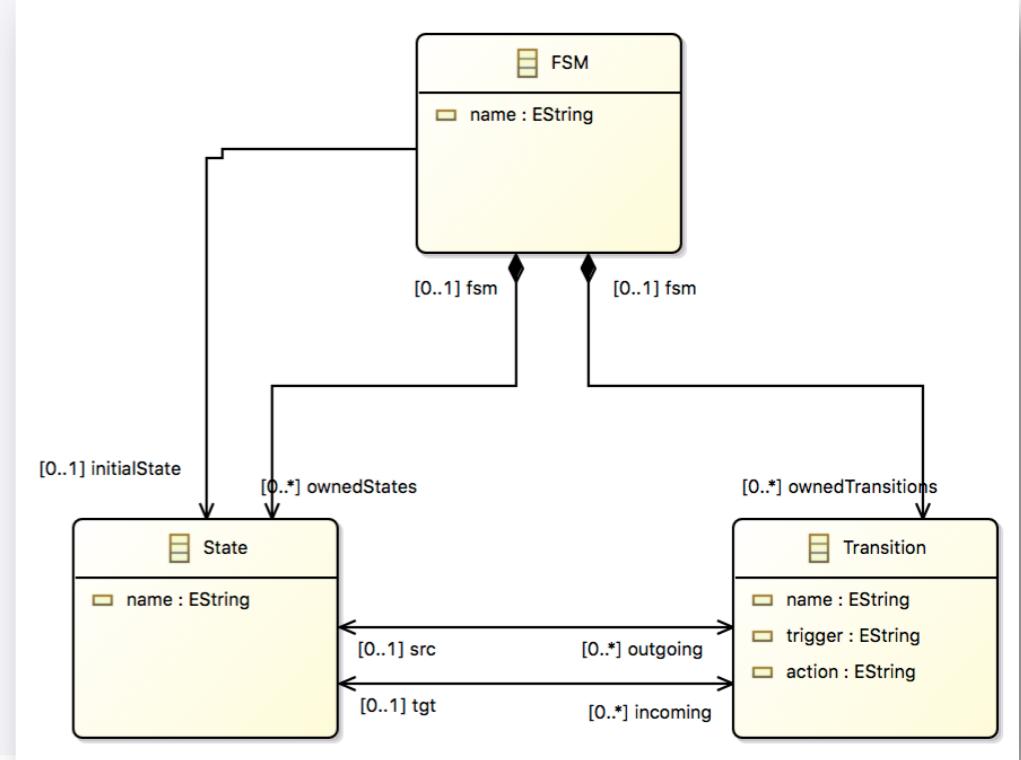
# Implement your own interpreter with Xtend/K3

```
@Aspect(className=State)
class StateAspect {
    @Step
    def public void step(String inputString) {
        // Get the valid transitions
        val validTransitions = _self.outgoing.filter[t | inputString.compareTo(t.trigger) == 0]

        if(validTransitions.empty) {
            //just copy the token to the output buffer
            _self.fsm.outputBuffer.enqueue(inputString)
        }

        if(validTransitions.size > 1) {
            throw new Exception("Non Determinism")
        }

        // Fire transition first transition (could be random%VT.size)
        if(validTransitions.size > 0){
            validTransitions.get(0).fire
            return
        }
        return
    }
}
```

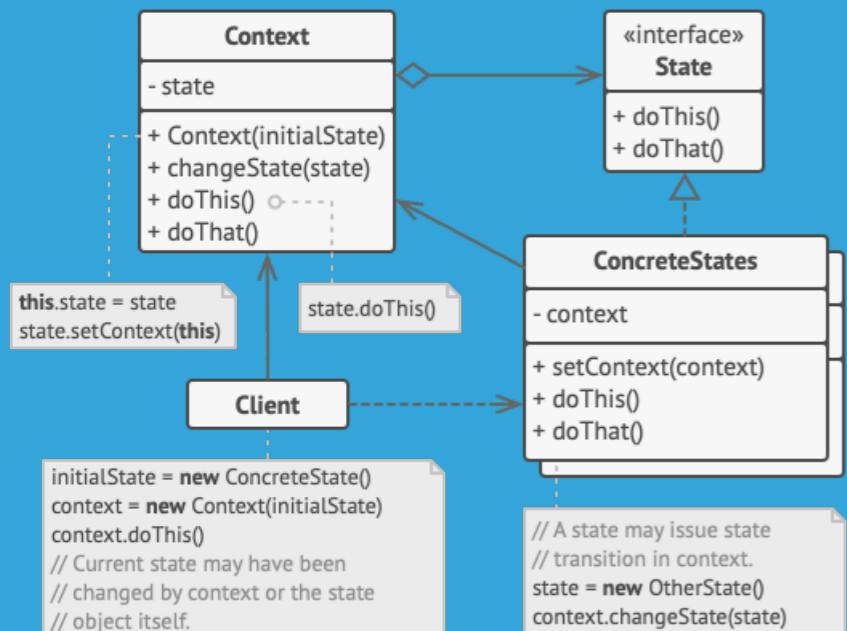
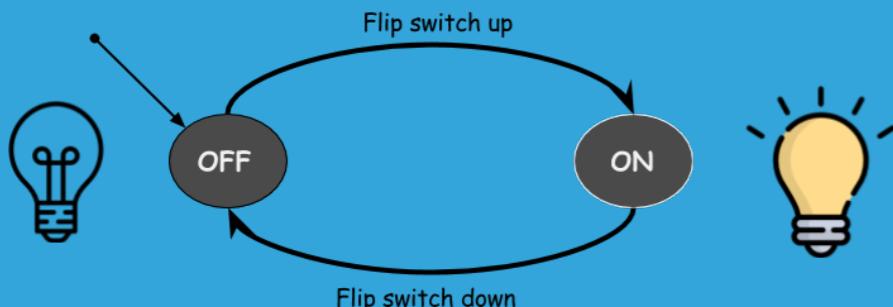


```
@Aspect(className=Transition)
class TransitionAspect {
    @Step
    def public void fire() {
        println("Firing " + _self.name + " and entering " + _self.tgt.name)
        val fsm = _self.src.fsm
        fsm.currentState = _self.tgt
        fsm.outputBuffer.enqueue(_self.action)
        fsm.consummedString = fsm.consumedString + fsm.underProcessTrigger
    }
}
```

```
@Aspect(className=FSM)
class FSMAspect {

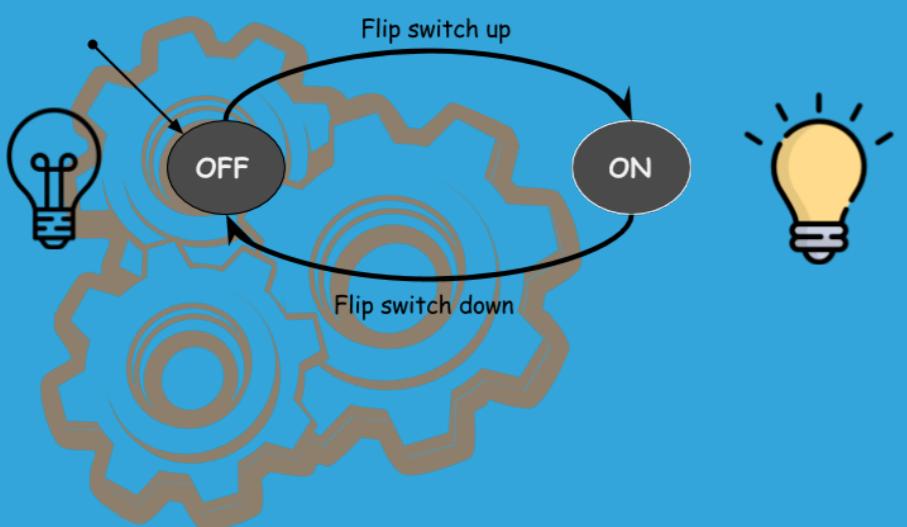
    public State currentState
}
```

# Part 3: define a compiler from your language to Java



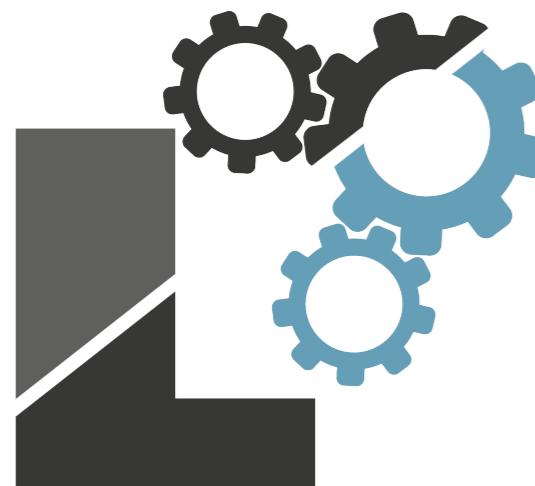
KEEP  
CALM  
AND  
DO IT  
YOURSELF

**Part 4 (optional):  
define an interpreter  
for your language**



**KEEP  
CALM  
AND  
DO IT  
YOURSELF**

# The GEMOC Studio



## *Language Workbench*

*Design and integrate your  
executable DSMLs*



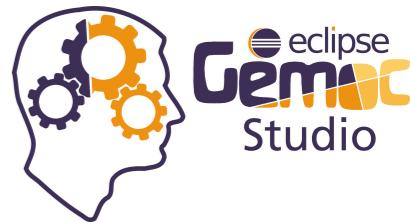
*and*  
<http://eclipse.org/gemoc>



## *Modeling Workbench*

*Edit, simulate and animate your  
heterogeneous models*

# Arduino Designer



The screenshot displays the Arduino Designer interface within the Eclipse GEMOC Studio environment. The interface is divided into several panes:

- Top Left:** A "Debug Configurations" view showing a project named "model.arduino".
- Top Center:** A "runtime-arduinoDebug - platform:/resource/org.gemoc.sample.arduino.sequential.blinker/model.aird/Sketch - Gemoc Studio" window showing the menu bar and toolbar.
- Left Column:** A "Debug" view showing a tree structure for "blinker [ALE Launcher]". The "Model debugging" node is expanded, showing a callout to "(VariableDeclaration) org.gemoc.sequential.model.arduino.impl.VariableDeclarationImpl@9b02cf4 -> execute()".
- Middle Left:** A "Hardware" view showing a schematic of an Arduino Board with three LEDs labeled "RED LED", "BLUE LED", and "WHITE LED" connected to pins 0, 1, and 2 respectively.
- Middle Right:** A "Sketch" view titled "newSketch" showing a state transition diagram. It starts with an initial state "i = 0" leading to a "Repeat 5" block. Inside the repeat loop, there are three parallel transitions: "BLUE LED : (i%2)", "RED LED : ((i/2)%2)", and "WHITE LED : ((i/4)%2)". Each transition leads to a "Set i = (i+1)" action, which then leads to the next iteration of the loop.
- Bottom Left:** A "Console" view showing the output: "Modeling workbench console", "About to initialize and run the GEMOC Execution Engine...", and "Initialization done, starting engine...".
- Bottom Right:** A "Variables" view showing a table of variables and their values:

Name	Value
i (org.gemoc.sequential.model.arduino.impl.RepeatImpl@4e523b1 (iteration: 5) :Repeat)	0
level (Arduino Board.0 :DigitalPin)	0
level (Arduino Board.1 :DigitalPin)	0
level (Arduino Board.2 :DigitalPin)	0
value (newSketch.i :IntegerVariable)	0

<https://github.com/gemoc/arduinomodeling>

Model Execution (M1ICE)  
Benoit Combemale, Feb. 2021

# UML Activity Diagram



Debug - platform:/resource/org.modelexecution.operationalsemantics.ad.samplemodels/model/test2.aird/test2 Activity Diagram - Gemoc Studio

File Edit Diagram Navigate Search Project Run Window Help

Quick Access Debug xDSML

Variables

Name	Value
heldTokens (ActivityFinalNode_finalNode2 :ActivityFinalNode)	
heldTokens (ForkNode_forkNode1 :ForkNode)	
heldTokens (InitialNode_initialNode2 :InitialNode)	[activitydiagram.impl.ControlTokenImpl]
heldTokens (JoinNode_joinNode1 :JoinNode)	

Breakpoints

Opaque Action action2

No details to display for the current selection.

Console \*test2 Activity Diagram

Properties

Opaque Action action2

Property	Value
Activity	Activity test2
Incoming	Control Flow edge4
Name	action2
Outgoing	Control Flow edge6
Running	true

Multidimensional Timeline

All execution states (11)

Timeline for dynamic information

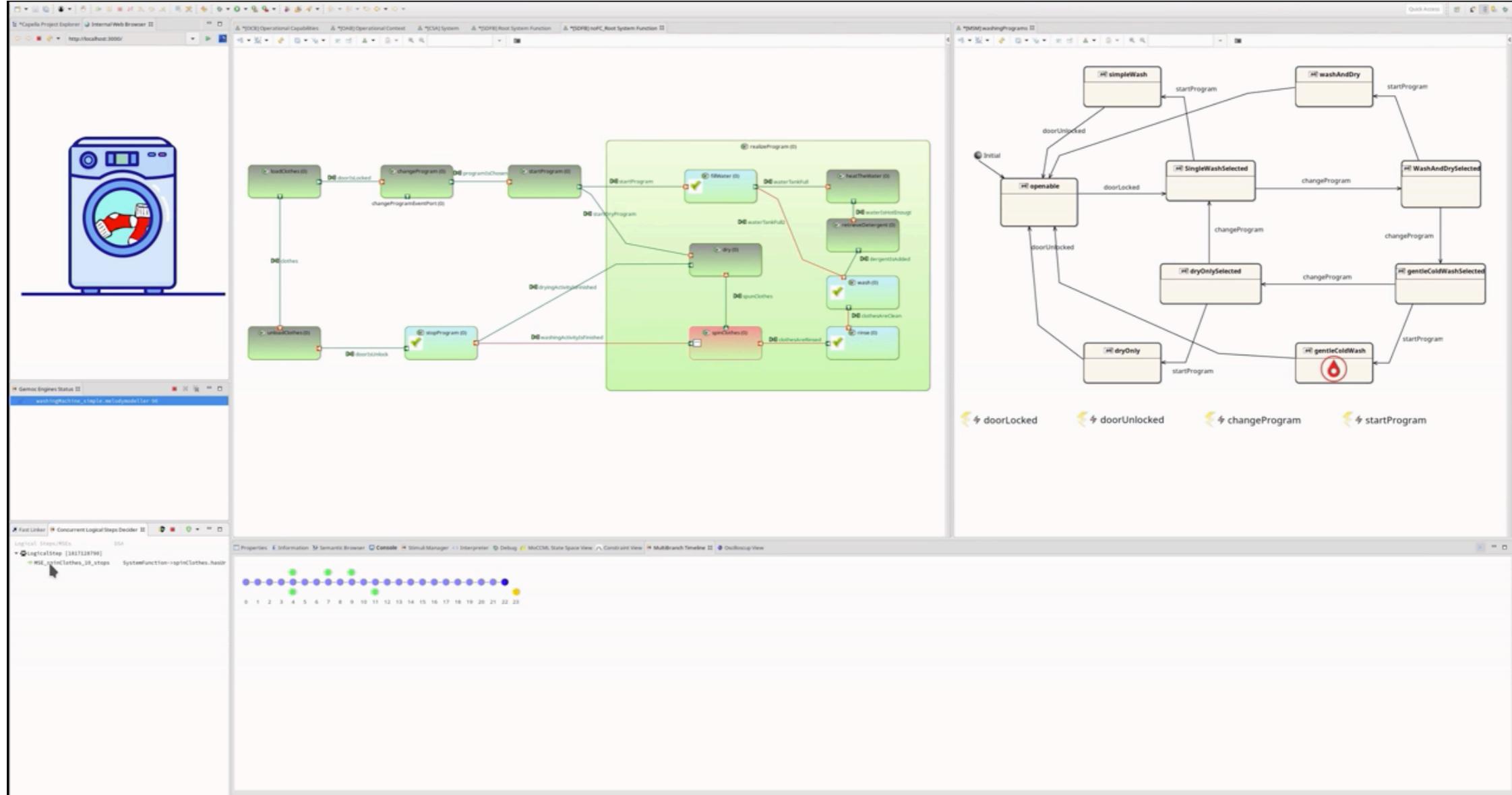
- trace (test2 :Activity)
- heldTokens (test2.initialNode2 :InitialNode)
- heldTokens (test2.forkNode1 :ForkNode)
- heldTokens (test2.action2 :OpaqueAction)
- heldTokens (test2.action3 :OpaqueAction)
- heldTokens (test2.joinNode1 :JoinNode)
- heldTokens (test2.finalNode2 :ActivityFinalNode)

Gemoc Engines Status

test2.ac 8

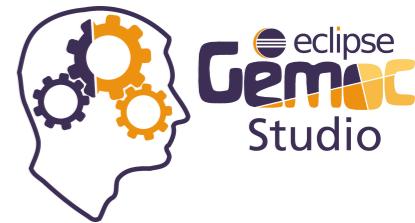
<https://github.com/gemoc/activitydiagram>

# xCapella



Model Execution (M1ICE)  
Benoit Combemale, Feb. 2021

# FCL

A screenshot of the Eclipse GEMOC Studio interface. The interface is divided into several panes:

- Project Explorer:** Shows the project structure, including files like "ExternalFiles", "miniquadcopter.fcl", and "MinQuadCopter.efd".
- Model Explorer:** Shows the selected model "QuadCtrlrNoFMU Function Diagram".
- QuadCtrlrNoFMU Function Diagram:** A functional block diagram showing various functional blocks connected by arrows.
- MinQuadCopter.efd:** An EFD (Executable Functional Diagram) showing a state transition graph with states like "RCCommandReceived", "PowerButtonPressed", and "MasterCommandReceived".
- QuadCtrlrNoFMU Mode Diagram:** A mode diagram showing different flight modes and their transitions.
- FCL Input View:** A table showing FCL input values for various ports.
- Code Editor:** Displays FCL code for the "lightControllerModel". The code defines automata for "ManualStabilizedFlight" and "AutonomousFlight", and handles functions for flight control.
- Outline:** Shows a tree view of the model elements.
- Status Bar:** Shows memory usage ("72.7M of 2890M") and other system information.

A context menu is open over a node in the mode diagram, with options like "Edit", "Show/hide", "Layout", "Reset Origin", "Format", "Show EClass information", "Show References", "OCL", and "Open in textual editor".