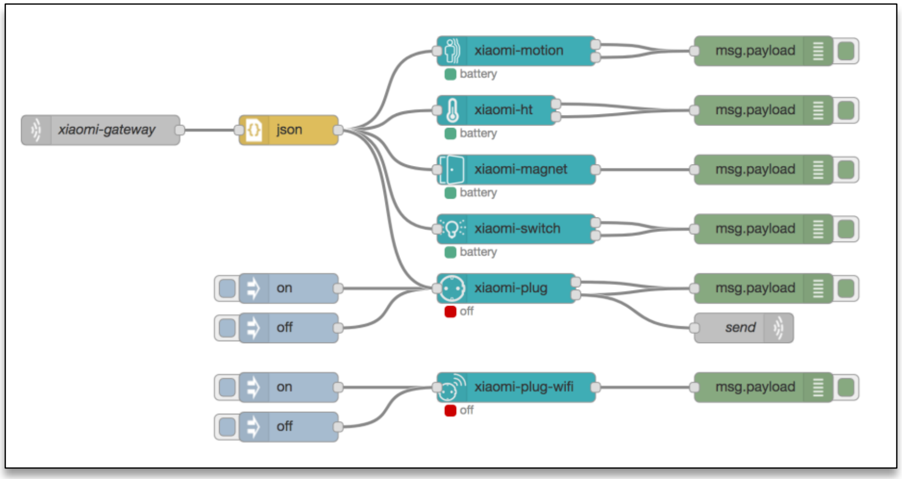
Concours N°06/02 Section 6 du CoNRS

Apprivoiser et synthétiser la diversité des langages logiciels

Thomas Degueule
Software Analysis and Transformation Group
Centrum Wiskunde & Informatica
https://tdegueul.github.io

Software languages



Node-RED

Internet of Things Language

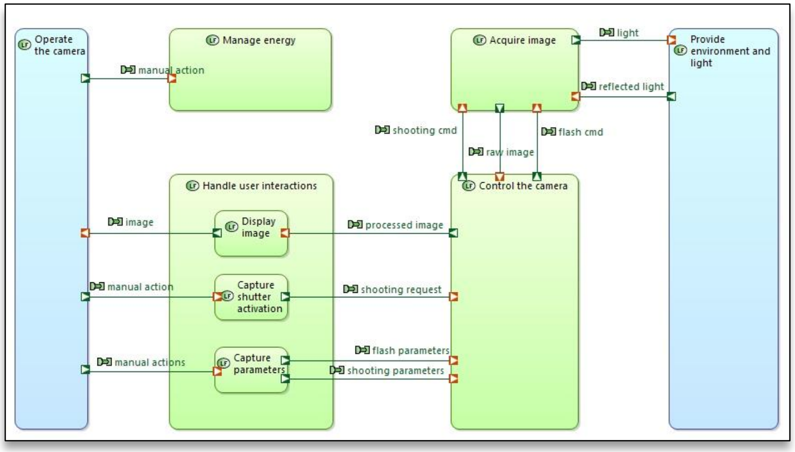
```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

-- this is the entity
entity ANDGATE is
  port (
    I1 : in  std_logic;
    I2 : in  std_logic;
    O  : out std_logic);
end entity ANDGATE;

-- this is the architecture
architecture RTL of ANDGATE is
begin
  O <= I1 and I2;
end architecture RTL;
```

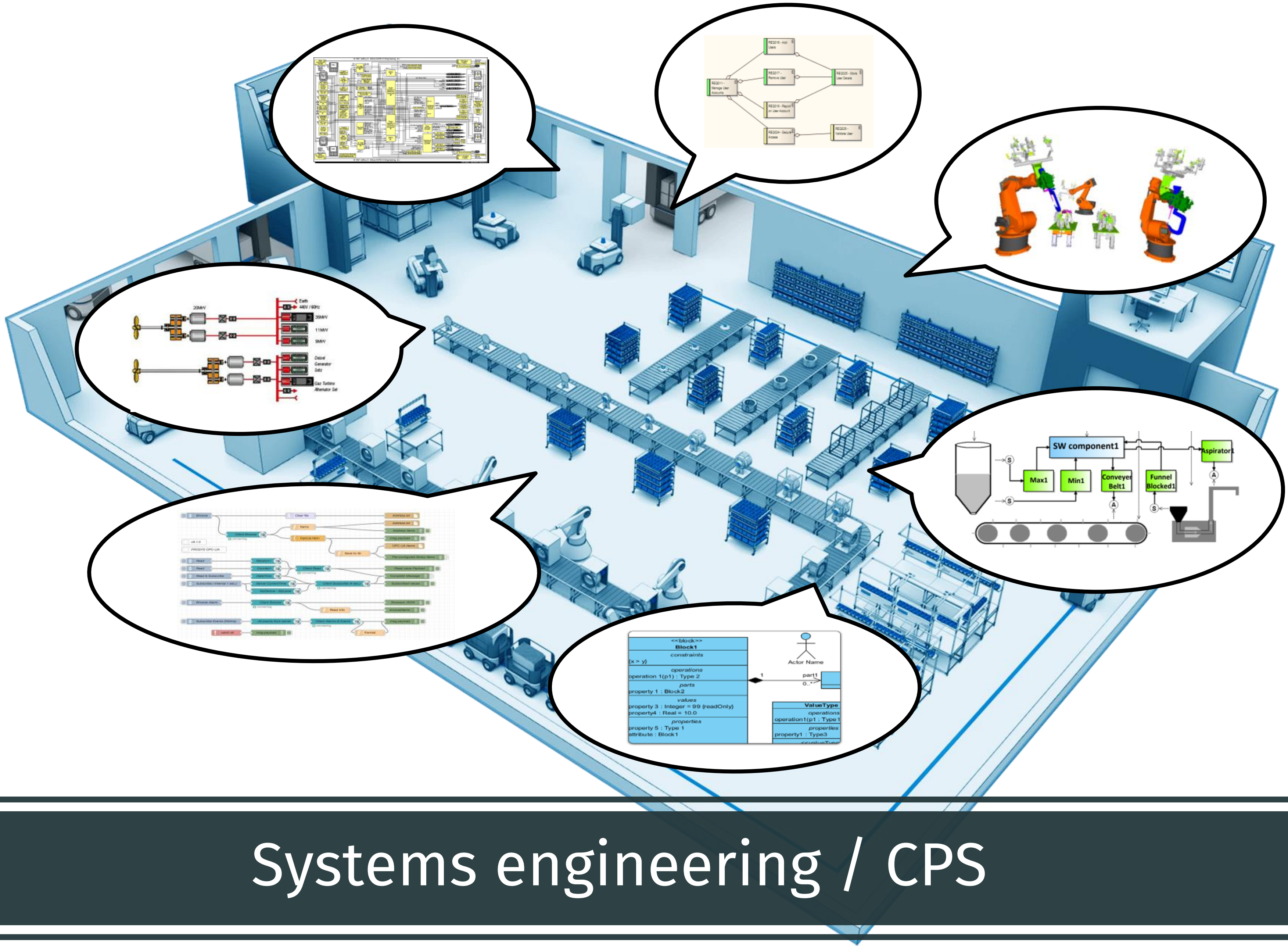
VHDL

Hardware Description Language

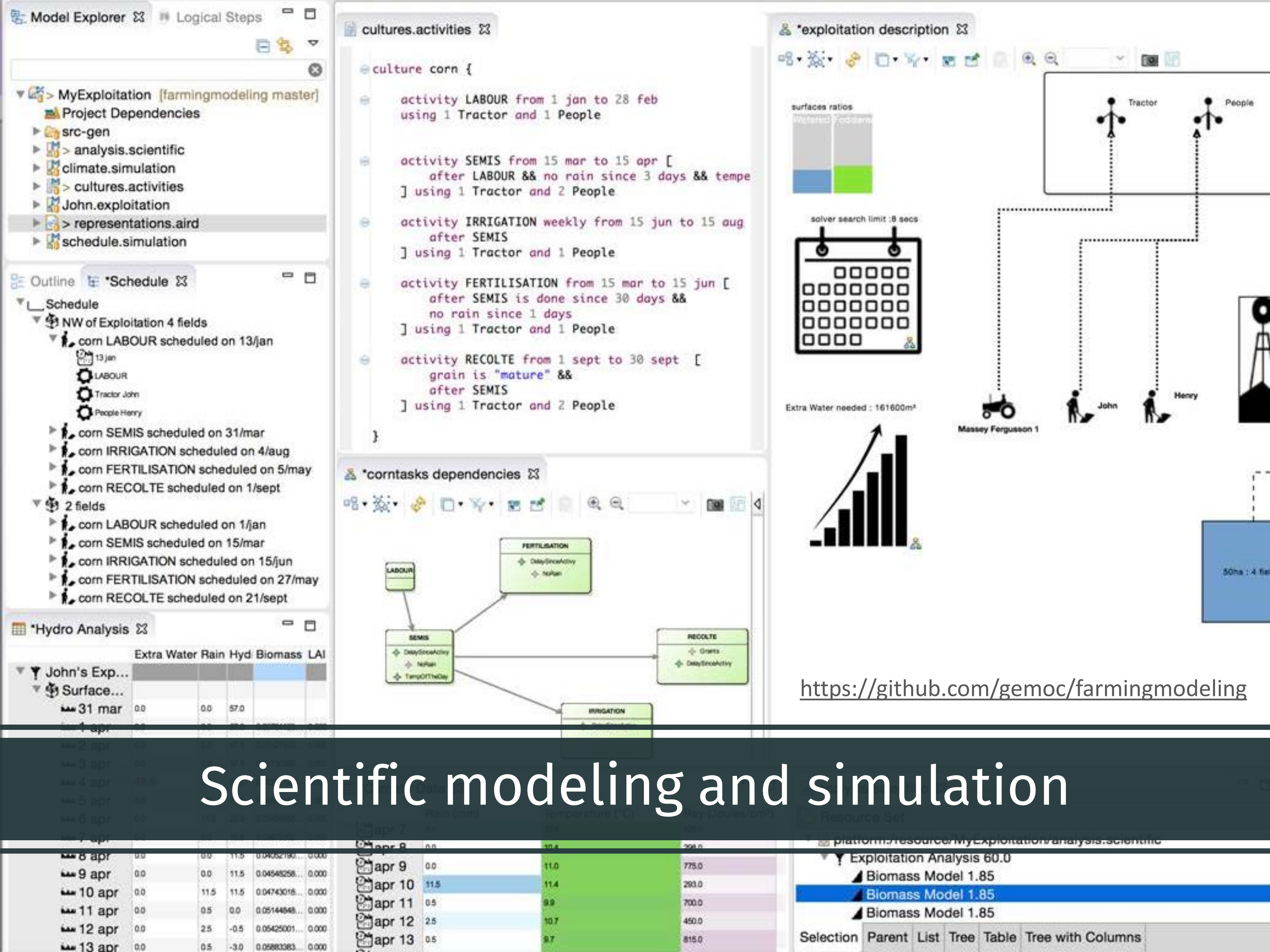


Capella

Systems Engineering Language



Systems engineering / CPS



Scientific modeling and simulation

https://github.com/gemoc/farmingmodeling

Model Explorer | Logical Steps

- MyExploitation [farmingmodeling master]
  - Project Dependencies
  - src-gen
  - > analysis.scientific
  - climate.simulation
  - > cultures.activities
  - John.exploitation
  - > representations.aird
  - schedule.simulation

cultures.activities

```
culture corn {

    activity LABOUR from 1 jan to 28 feb
    using 1 Tractor and 1 People


    activity SEMIS from 15 mar to 15 apr [
        after LABOUR && no rain since 3 days && tempe
    ] using 1 Tractor and 2 People

    activity IRRIGATION weekly from 15 jun to 15 aug
        after SEMIS
    ] using 1 Tractor and 1 People

    activity FERTILISATION from 15 mar to 15 jun [
        after SEMIS is done since 30 days &&
        no rain since 1 days
    ] using 1 Tractor and 1 People

    activity RECOLTE from 1 sept to 30 sept  [
        grain is "mature" &&
        after SEMIS
    ] using 1 Tractor and 2 People

}
```

Outline | *Schedule

- Schedule
  - NW of Exploitation 4 fields
    - corn LABOUR scheduled on 13/jan
      - 13 jan
      - LABOUR
      - Tractor John
      - People Henry
    - corn SEMIS scheduled on 31/mar
    - corn IRRIGATION scheduled on 4/aug
    - corn FERTILISATION scheduled on 5/may
    - corn RECOLTE scheduled on 1/sept
  - 2 fields
    - corn LABOUR scheduled on 1/jan
    - corn SEMIS scheduled on 15/mar
    - corn IRRIGATION scheduled on 15/jun
    - corn FERTILISATION scheduled on 27/may
    - corn RECOLTE scheduled on 21/sept

*Hydro Analysis

	Extra Water	Rain	Hyd	Biomass	LAI
John's Exp...					
Surface...					
31 mar	0.0	0.0	57.0		
8 apr	0.0	0.0	11.5	0.04052190...	0.000
9 apr	0.0	0.0	11.5	0.04548258...	0.000
10 apr	0.0	11.5	11.5	0.04743018...	0.000
11 apr	0.0	0.5	0.0	0.05144848...	0.000
12 apr	0.0	2.5	-0.5	0.05425001...	0.000
13 apr	0.0	0.5	-3.0	0.05883383...	0.000

*corntasks dependencies

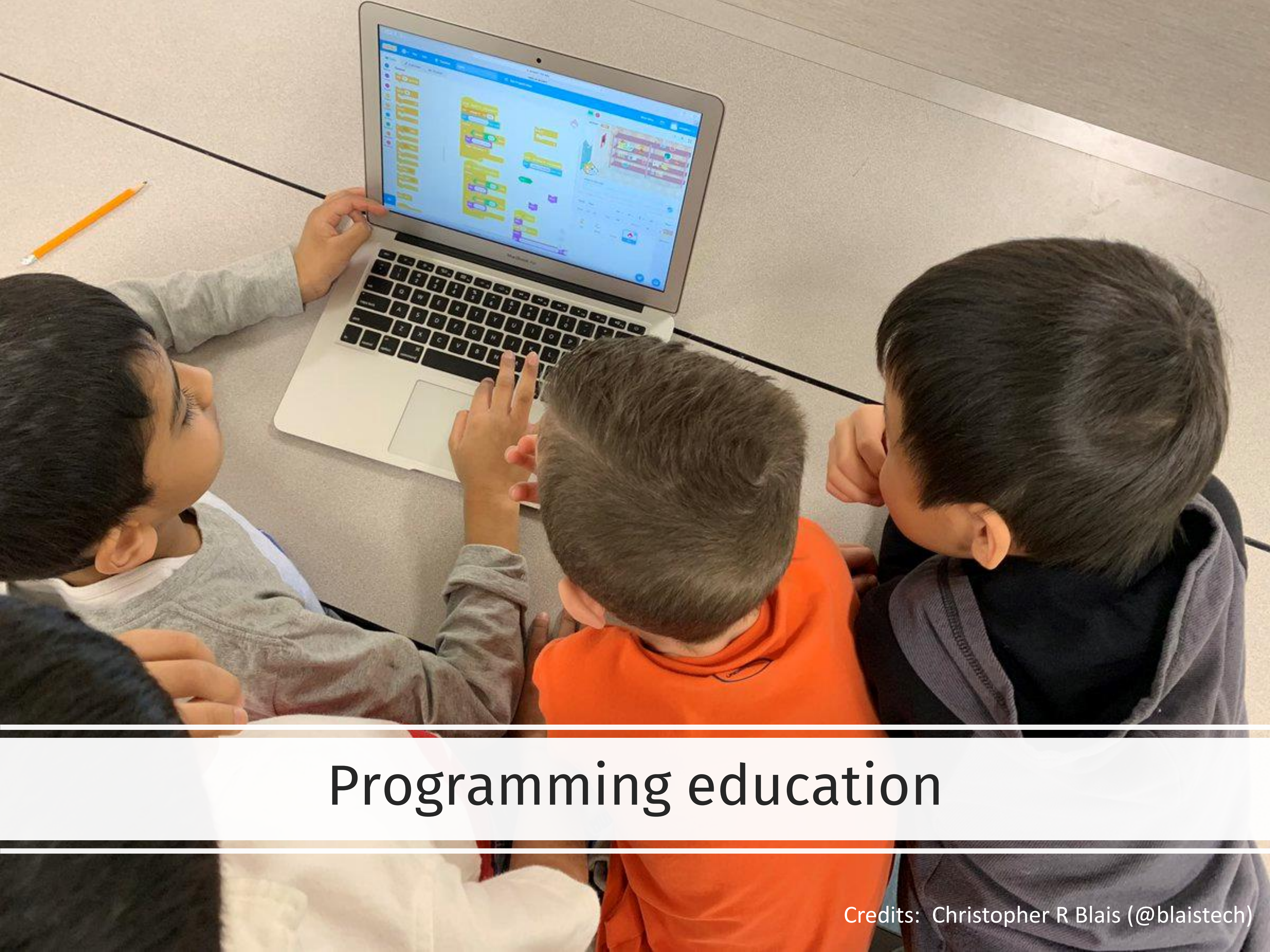
*exploitation description

	Rain (mm)	Temperature (°C)	Ray (Joules/cm²)
apr 8	0.0	10.4	298.0
apr 9	0.0	11.0	775.0
apr 10	11.5	11.4	293.0
apr 11	0.5	9.9	700.0
apr 12	2.5	10.7	450.0
apr 13	0.5	9.7	815.0

Resource Set

- platform:/resource/MyExploitation/analysis.scientific
  - Exploitation Analysis 60.0
    - Biomass Model 1.85
    - Biomass Model 1.85
    - Biomass Model 1.85

Selection | Parent | List | Tree | Table | Tree with Columns



Programming education

Credits: Christopher R Blais (@blaistech)

Software Language Engineering (SLE)

Static Analyzers

Interpreters

Checkers

Compilers

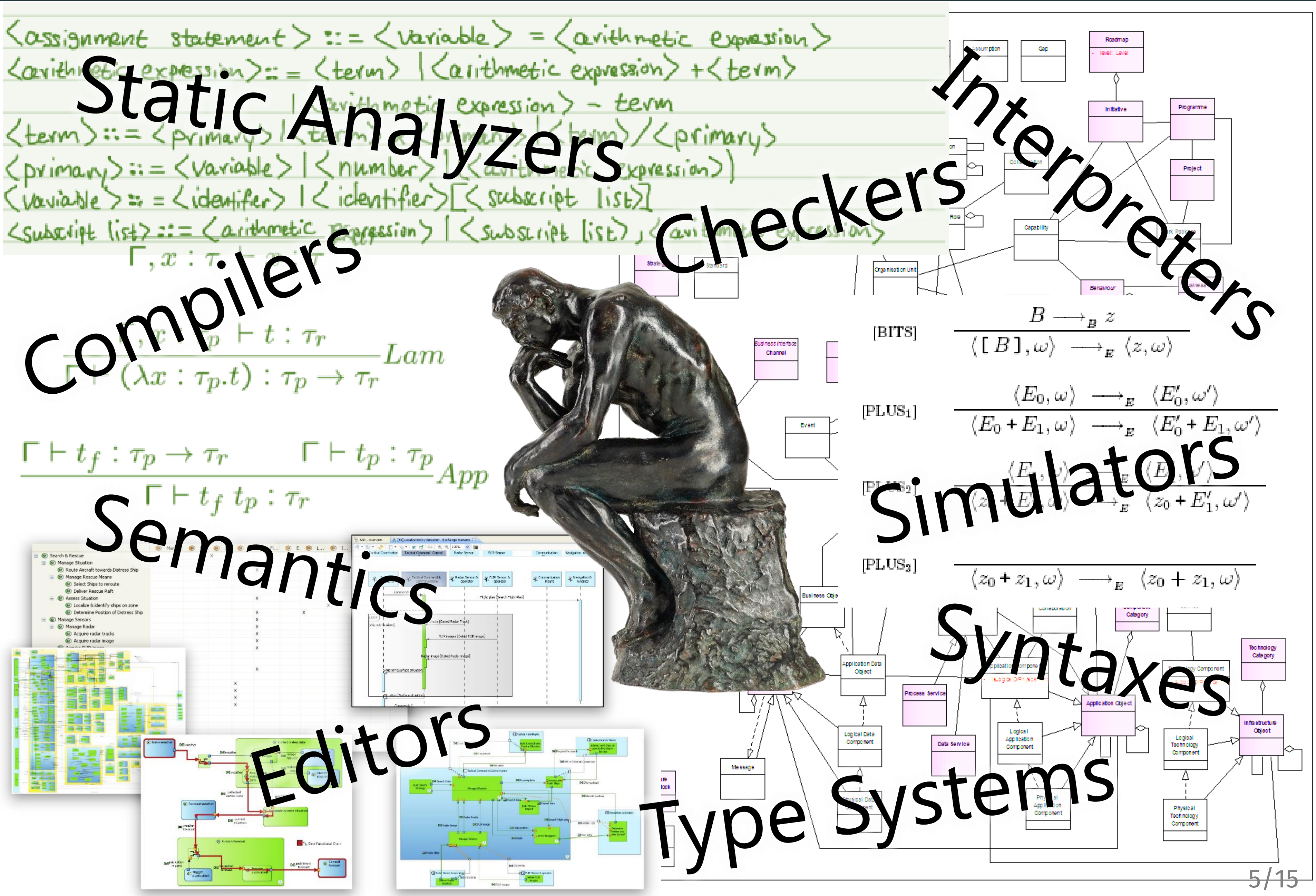
Semantics

Simulators

Syntaxes

Editors

Type Systems



Research Activity

- Scientific contributions
  - Modularity, reuse, composition (e.g. [SLE'15], [COMLAN'16], [MODELS'17])
  - Flexible modeling, live modeling (e.g. [COMLAN'17], [SLE'18$_1$], [SLE'18$_2$])
  - Empirical software engineering (e.g. [MSR'18], [ICSE'19])

- Technological contributions

http://melange.inria.fr http://www.kermeta.org https://eclipse.org/gemoc http://eclipse.org/scava/

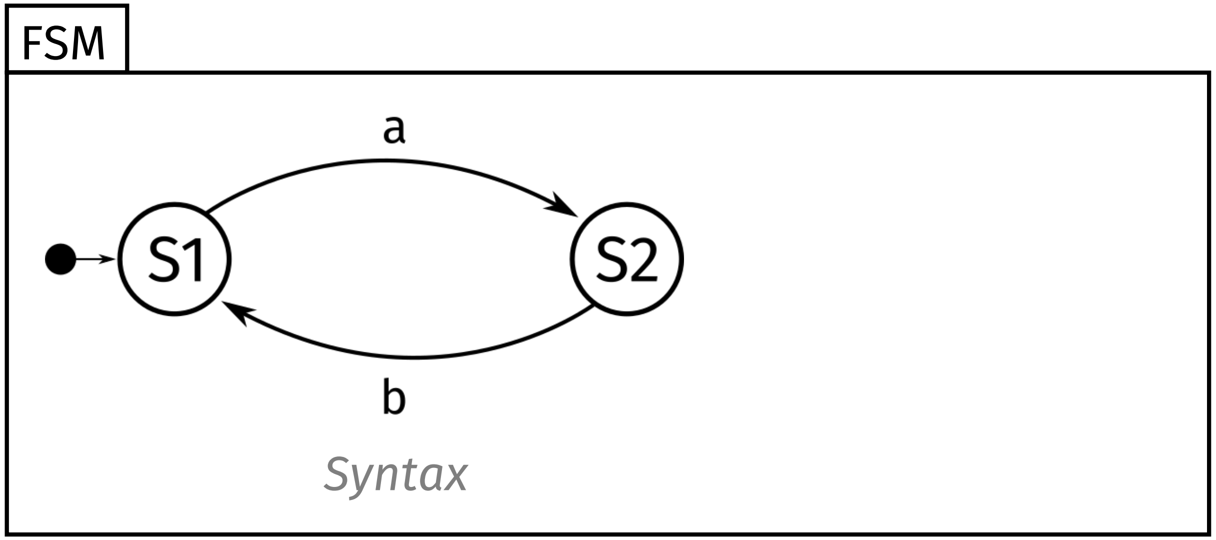
- Industrial collaborations

- Experimental validation
  - Programming languages
  - Systems engineering
  - Internet of Things
  - Open-source software

Modular Language Extension

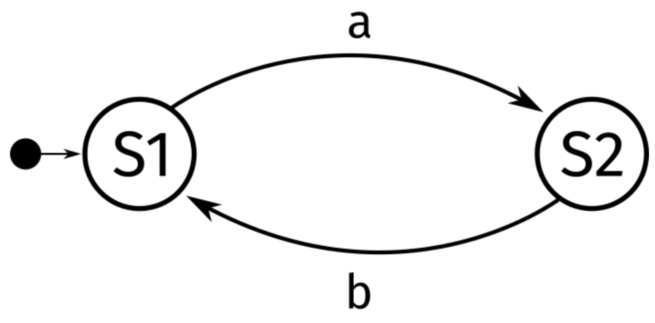
Revisiting Visitors for Modular Extension of Executable DSMLs
Manuel Leduc, Thomas Degueule, Benoit Combemale, Tijs van der Storm, Olivier Barais
20th International Conference on Model Driven Engineering Languages and Systems (MODELS'17)

Modular language extension



Modular language extension

FSM



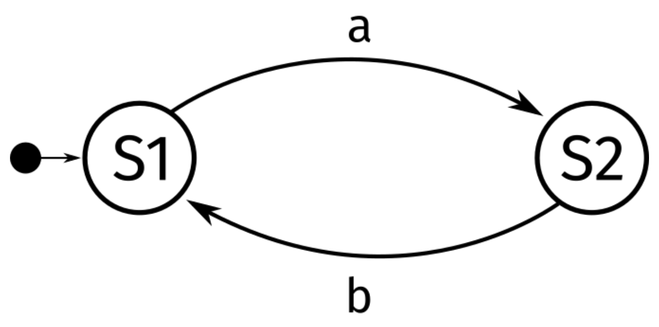
Syntax

```
eval(fsm, 'ab'):
  s1 -> s2
  s2 -> s1
```

Interpreter

Modular language extension

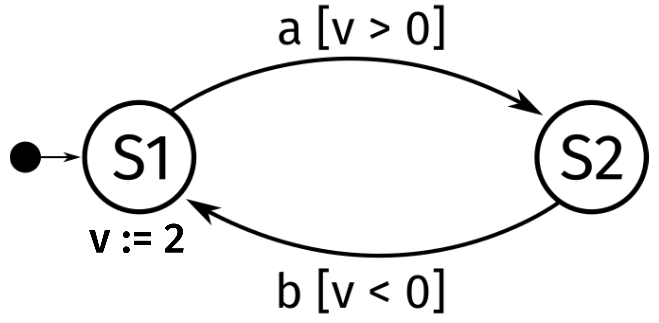
FSM



```
eval(fsm, 'ab'):
  s1 -> s2
  s2 -> s1
```

Syntax *Interpreter*

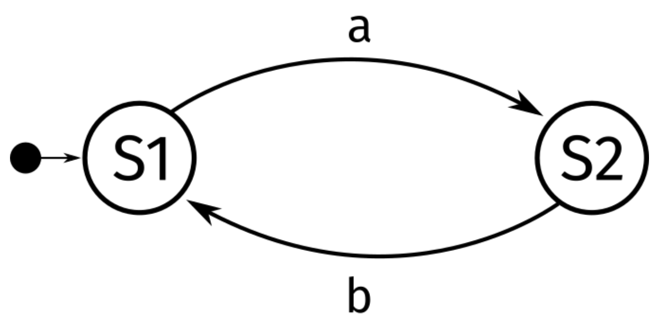
GuardedFSM



Syntax'

Modular language extension

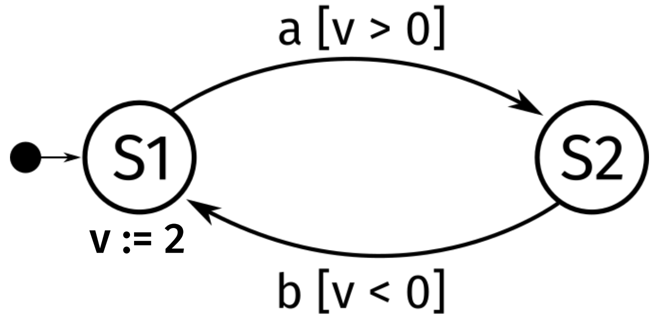
FSM



```
eval(fsm, 'ab'):
  s1 -> s2
  s2 -> s1
```

Syntax *Interpreter*

GuardedFSM

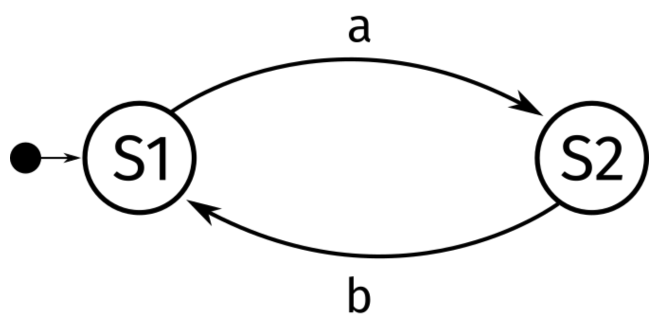


```
eval(gfsm, 'ab'):
  s1 -> s2
  s2 -> s1
```

Syntax' *Interpreter'*

Modular language extension

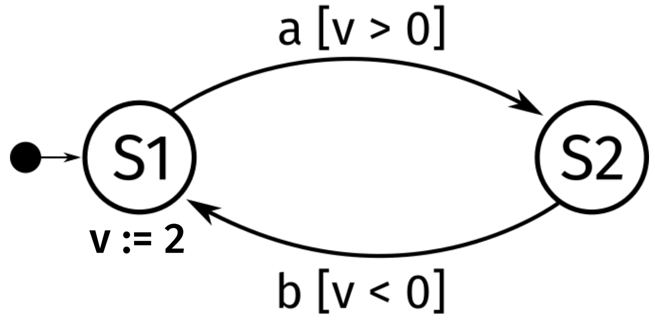
FSM



```
eval(fsm, 'ab'):
  s1 -> s2
  s2 -> s1
```

Syntax *Interpreter*

GuardedFSM

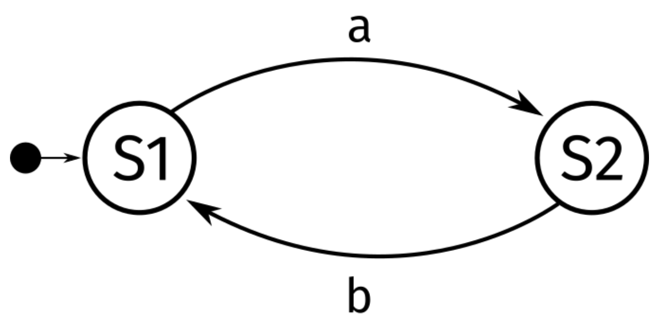


```
eval(gfsm, 'ab'):
  s1 -> s2
  <deadlock>
```

Syntax' *Interpreter'*

Modular language extension

FSM

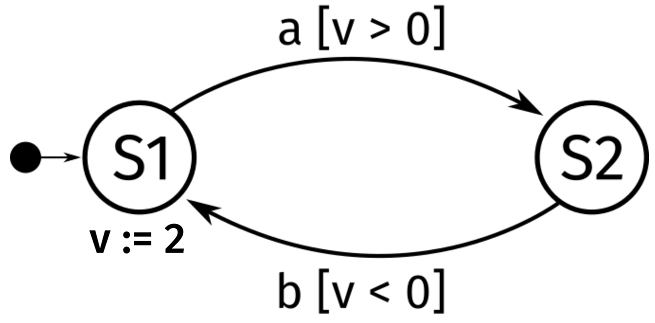


```
eval(fsm, ‘ab’):
  s1 -> s2
  s2 -> s1
```

```
pretty-print(fsm):
    machine
      initial S1
        a to S2
      state S2
        b to S1
```

Syntax *Interpreter* *Pretty-printer*

GuardedFSM

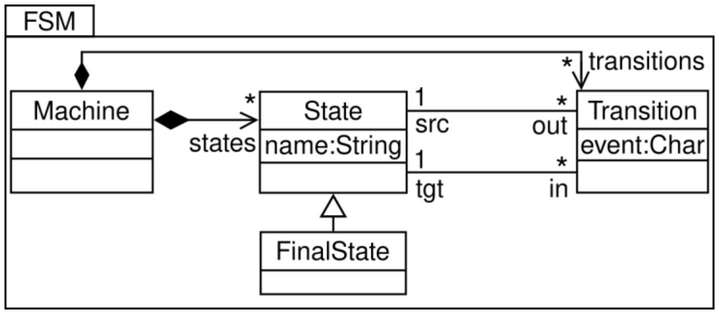


```
eval(gfsm, ‘ab’):
  s1 -> s2
  <deadlock>
```

```
pretty-print(gfsm):
machine
  initial S1
    a to S2 if v > 0
  state S2
    b to S1 if v < 0
```

Syntax’ *Interpreter’* *Pretty-printer’*

Modular language extension

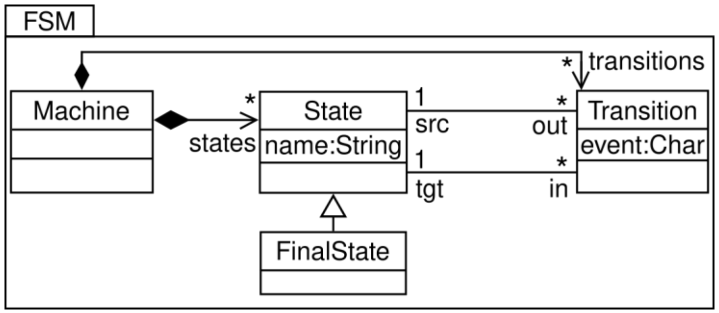


Abstract Syntax

```
interpret(State s, String evt) {
  // Find a transition for the input event
  State next = s.out.findFirst[event == evt]
  // Fire it
  next.fire()
}
```

Execution Semantics

Modular language extension



Abstract Syntax

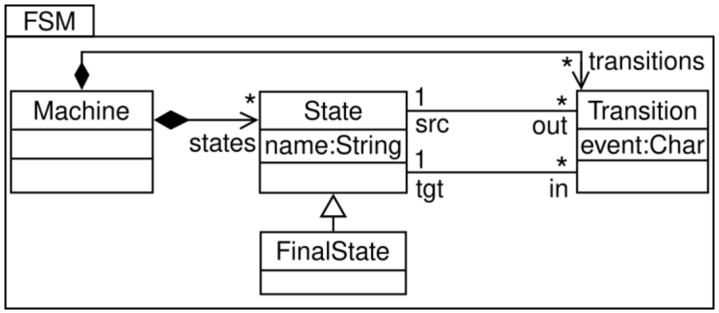
```
class Machine { List<State> states; [...] }
class State   { String name; [...] }
class Trans   { char event; [...] }
class FS extends State { [...] }
```

Abstract Syntax Classes

```
interpret(State s, String evt) {
  // Find a transition for the input event
  State next = s.out.findFirst[event == evt]
  // Fire it
  next.fire()
}
```

Execution Semantics

Modular language extension



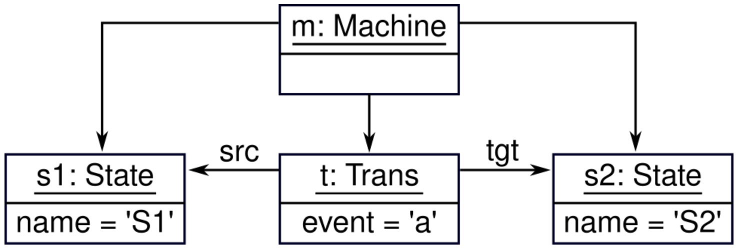
Abstract Syntax

```
class Machine { List<State> states; […] }
class State   { String name; […] }
class Trans   { char event; […] }
class FS extends State { […] }
```

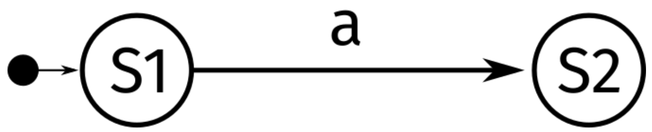
Abstract Syntax Classes

```
interpret(State s, String evt) {
  // Find a transition for the input event
  State next = s.out.findFirst[event == evt]
  // Fire it
  next.fire()
}
```

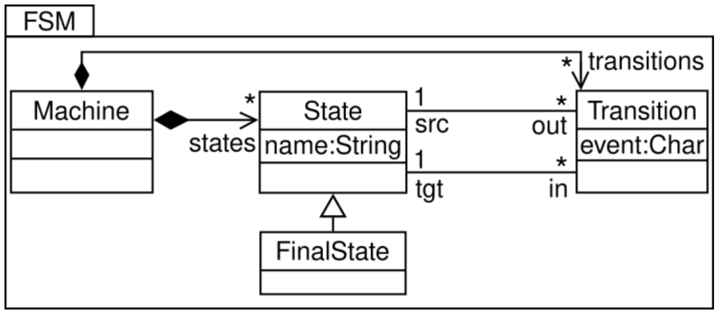
Execution Semantics



Abstract Syntax Graph



Modular language extension



Abstract Syntax

```
class Machine { List<State> states; […] }
class State   { String name; […] }
class Trans   { char event; […] }
class FS extends State { […] }
```

Abstract Syntax Classes

```
interpret(State s, String evt) {
  // Find a transition for the input event
  State next = s.out.findFirst[event == evt]
  // Fire it
  next.fire()
}
```

Execution Semantics

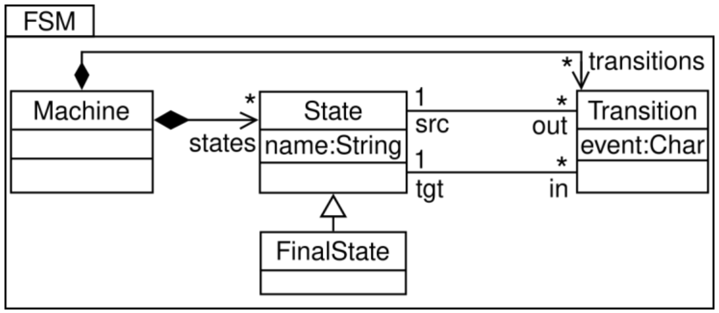
```
interface Interpret { void interpret(); }
class Machine implements Interpret { […] }
class State   implements Interpret { […] }
class Trans   implements Interpret { […] }
```

Interpreter Pattern

```
class Machine { void accept(Visitor v); }
class State   { void accept(Visitor v); }
class Trans   { void accept(Visitor v); }
interface Visitor {
  void visit(Machine m);
  void visit(State s);
  void visit(Transition t);
}
class ExecMachine implements Visitor {
  void visit(Machine m)    { […] }
  void visit(State s)      { […] }
  void visit(Transition t) { […] }
}
```

Visitor Pattern

Modular language extension



Abstract Syntax

```
class Machine { List<State> states; […] }
class State   { String name; […] }
class Trans   { char event; […] }
class FS extends State { […] }
```

Abstract Syntax Classes

```
interpret(State s, String evt) {
  // Find a transition for the input event
  State next = s.out.findFirst[event == evt]
  // Fire it
  next.fire()
}
```

Execution Semantics

```
interface Interpret { void interpret(); }
interface Print     { void print(); }
class Machine implements           { […] }
class State   implements      ✗    { […] }
class Trans   implements           { […] }
```

Interpreter Pattern

✗

✓

```
pretty-print(State s) {
  print("State " + s.name)
  print("Transitions:")
  for (t in s.transitions)
    pretty-print(t)
}
```

Printing Semantics

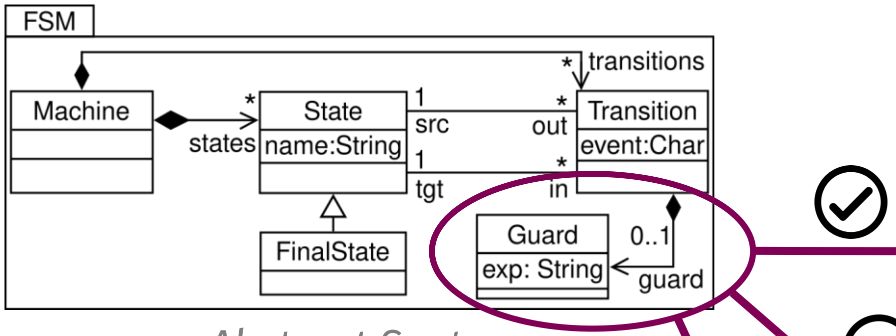
```
interface Visitor { […] }

class ExecMachine implements Visitor {
  void visit(Machine m)    { […] }
  void visit(State s)      { […] }
  void visit(Transition t) { […] }
}

class PrintMachine implements Visitor {
  void visit(Machine m)    { […] }
  void visit(State s)      { […] }   ✓
  void visit(Transition t) { […] }
}
```

Visitor Pattern

Modular language extension



Abstract Syntax

```
interpret(State s, String evt) {
  // Find a transition for the input event
  State next = s.out.findFirst[event == evt]
  // Fire it
  next.fire()
}
```

Execution Semantics

```
class Machine { List<State> states; […] }
class State   { String name; […] }
class Trans   { char event; Guard guard; }
class FS extends State { […] }
class Guard   { String exp; }
```

Abstract Syntax Classes

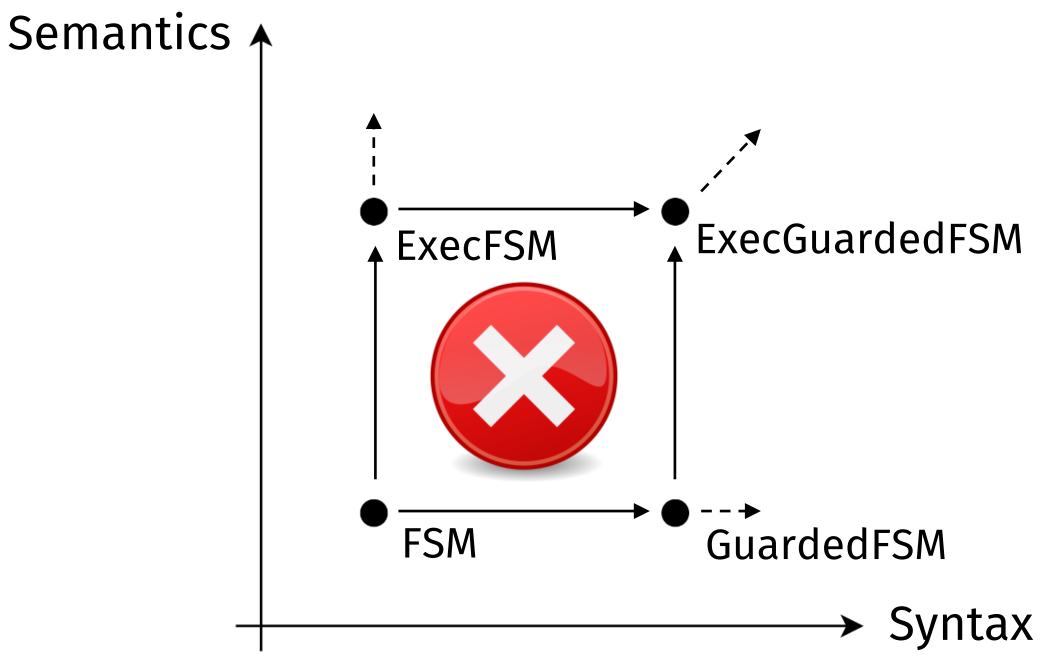
```
interface Interpret { void interpret(); } ✓
class Machine implements Interpret { […] }
class State   implements Interpret { […] }
class Trans   implements Interpret { […] }
class Guard   implements Interpret { […] } ✓
```

Interpreter Pattern

```
interface Visitor {
  void visit(Machine m);
  void visit(State s);
  void visit(Transition t);
  void visit(Guard g); ✗
}
class ExecMachine implements Visitor {
  void visit(Machine m)    { […] }
  void visit(State s)      { […] }
  void visit(Transition t) { […] }
  void visit(Guard g)      { […] } ✗
}
```

Visitor Pattern

The *Expression Problem*

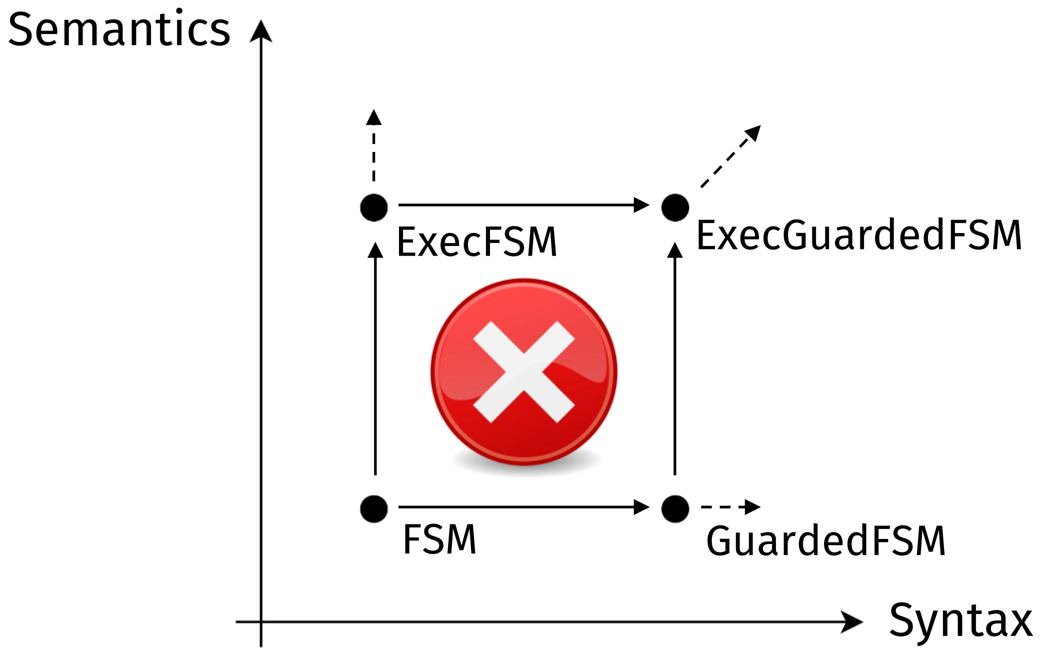


“The expression problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety.”

Philip Wadler, 1998

- Non-linear and independent extension
- Without anticipation
- Without modification or duplication
- With incremental compilation
- While ensuring static type safety

The *Expression Problem*

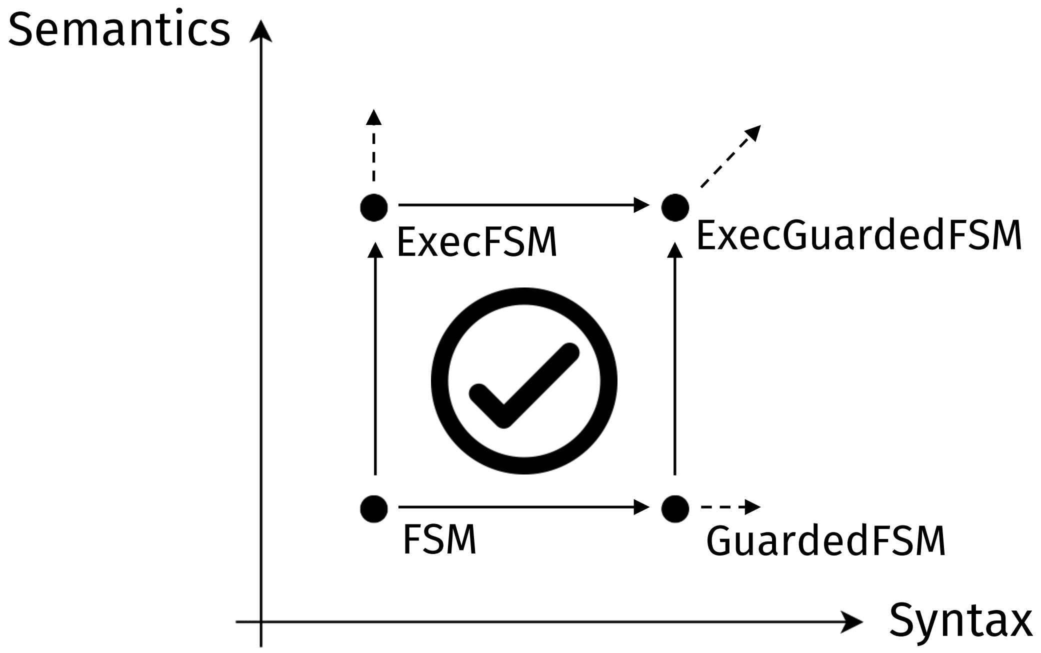


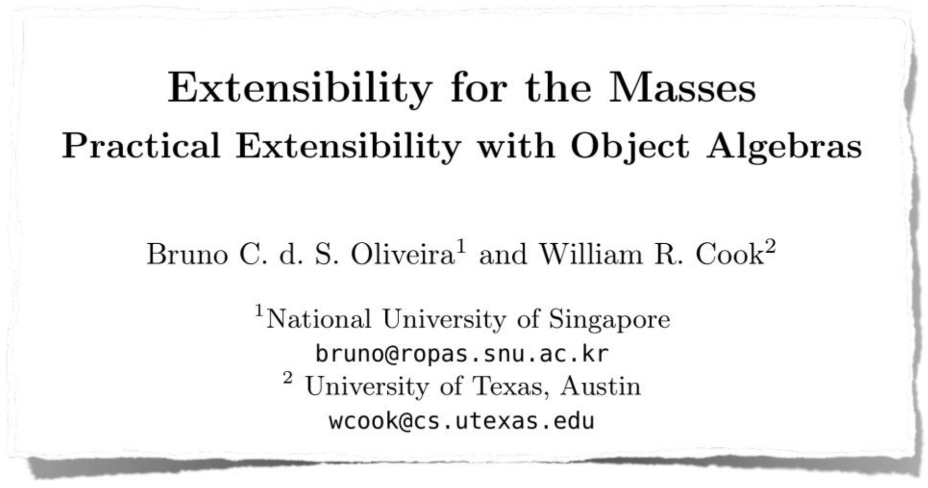
"The expression problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety."

Philip Wadler, 1998

- Multi-methods
- Open classes
- Virtual classes
- ...

The *Expression Problem*




Extensibility for the Masses

Practical Extensibility with Object Algebras

Bruno C. d. S. Oliveira[1] and William R. Cook[2]

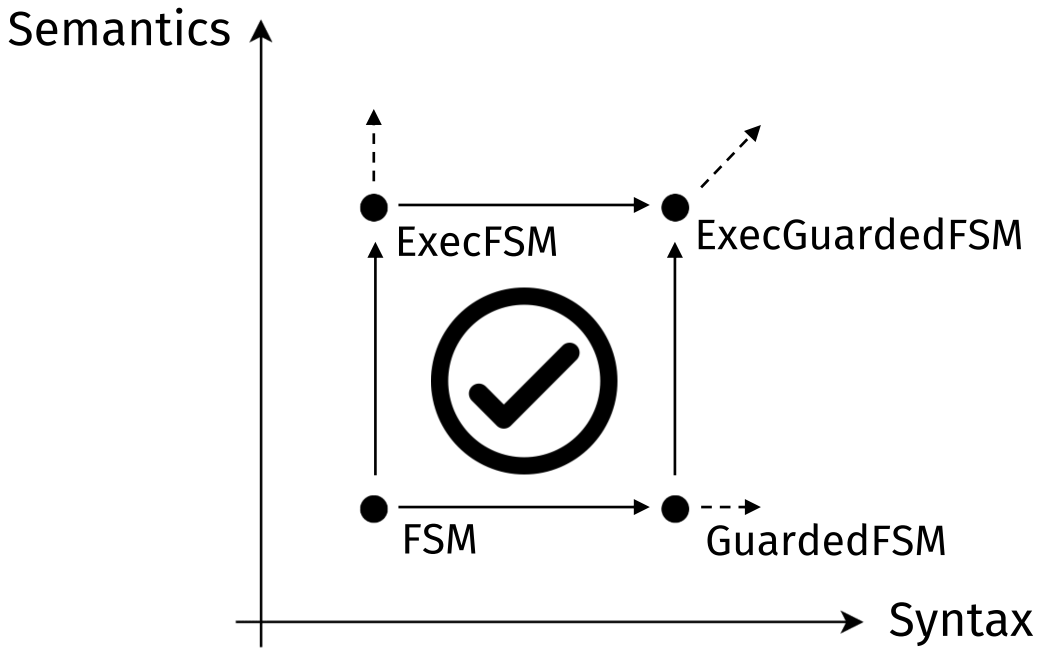
[1]National University of Singapore
bruno@ropas.snu.ac.kr
[2] University of Texas, Austin
wcook@cs.utexas.edu

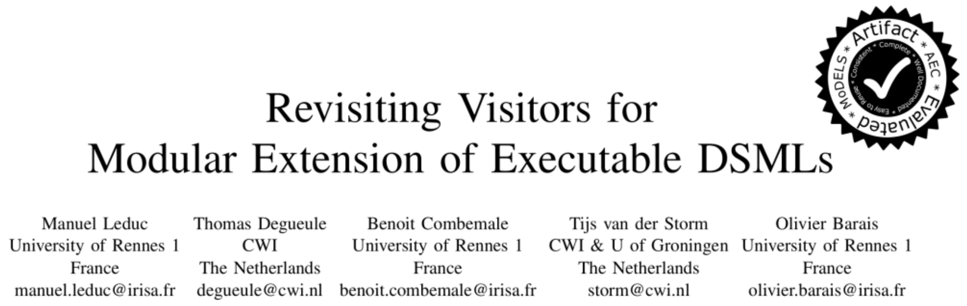
European Conference on Object-Oriented Programming (ECOOP'12)

- An Object Algebra is an object-oriented encoding of algebraic signatures

Incompatible with metamodels!

The *Expression Problem*



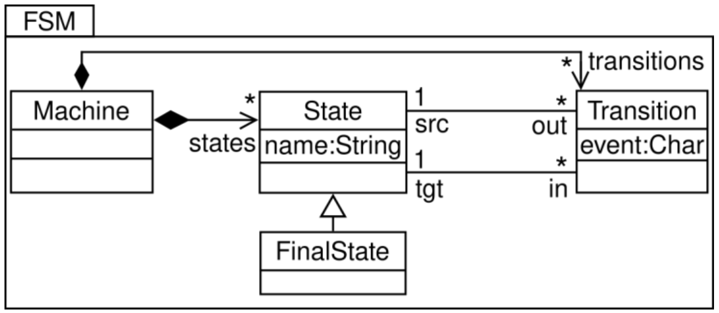

Revisiting Visitors for
Modular Extension of Executable DSMLs

Manuel Leduc, University of Rennes 1, France, manuel.leduc@irisa.fr
Thomas Degueule, CWI, The Netherlands, degueule@cwi.nl
Benoit Combemale, University of Rennes 1, France, benoit.combemale@irisa.fr
Tijs van der Storm, CWI & U of Groningen, The Netherlands, storm@cwi.nl
Olivier Barais, University of Rennes 1, France, olivier.barais@irisa.fr

- A language implementation pattern that solves the Expression Problem
- Reconcile the structural extensibility of the object-oriented style with the behavioral extensibility of the functional style

The REVISITOR *Pattern*

The Revisitor Pattern



Abstract Syntax

```
class Machine { List<State> states; […] }
class State   { String name; […] }
class Trans   { char event; […] }
class FS extends State { […] }
```

Abstract Syntax Classes

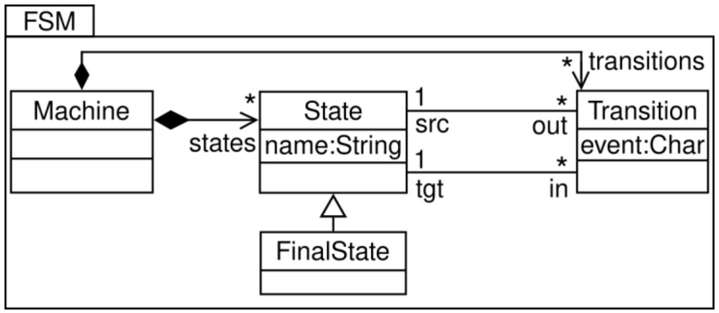
```
interface FsmRev<M, S, F extends S, T> {
  M machine(Machine it);
  T trans  (Trans it);
  default M $(Machine it) { return machine(it); }
  default T $(Trans it)   { return trans(it);   }
  […]
}
```

Revisitor *Interface*

```
interpret(State s, String evt) {
  // Find a transition for the input event
  State next = s.out.findFirst[event == evt]
  // Fire it
  next.fire()
}
```

Execution Semantics

The Revisitor Pattern



Abstract Syntax

```
class Machine { List<State> states; [...] }
class State   { String name; [...] }
class Trans   { char event; [...] }
class FS extends State { [...] }
```

Abstract Syntax Classes

```
interface FsmRev<M, S, F extends S, T> {
  M machine(Machine it);
  T trans  (Trans it);
  default M $(Machine it) { return machine(it); }
  default T $(Trans it)   { return trans(it);   }
  [...]
}
```

REVISITOR *Interface*

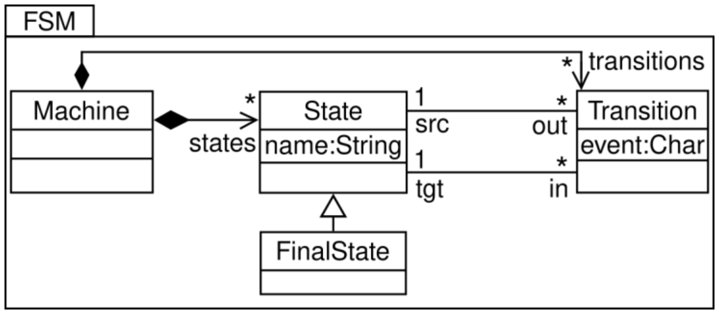
```
interpret(State s, String evt) {
  // Find a transition for the input event
  State next = s.out.findFirst[event == evt]
  // Fire it
  next.fire()
}
```

Execution Semantics

```
interface EvalFsm extends FsmRev<EM,ES,EFS,ET> {
  default ES state(State it) {
    return (evt) -> {
      State next = it.out.findFirst[event==evt];
      $(next).fire();
    };
  }
  [...]
}
```

REVISITOR *Implementation*

The Revisitor Pattern



Abstract Syntax

```
interpret(State s, String evt) {
  // Find a transition for the input event
  State next = s.out.findFirst[event == evt]
  // Fire it
  next.fire()
}
```

Execution Semantics

```
pretty-print(State s) {
  print("State " + s.name)
  print("Transitions:")
  for (t in s.transitions)
    pretty-print(t)
}
```

Printing Semantics

```
class Machine { List<State> states; […] }
class State   { String name; […] }
class Trans   { char event; […] }
class FS extends State { […] }
```

Abstract Syntax Classes

```
interface FsmRev<M, S, F extends S, T> {
  M machine(Machine it);
  T trans  (Trans it);
  default M $(Machine it) { return machine(it); }
  default T $(Trans it)   { return trans(it);   }
  […]
}
```

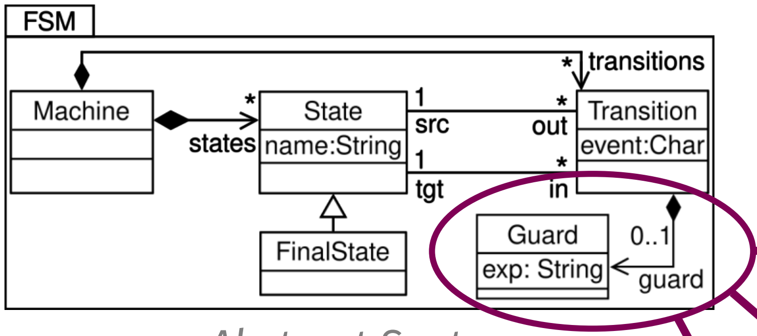
REVISITOR *Interface*

✓

```
interface PrintFsm extends FsmRev<Pr,Pr,Pr,Pr> {
  default Pr state(State it) {
    return () -> "State" + it.name + […] +
                 "Transitions:" +
                 it.trans.map[t | $(t).print()];
  }
  […]
}
```

REVISITOR *Implementation*

The Revisitor Pattern



Abstract Syntax

```
class Machine { List<State> states; […] }
class State   { String name; […] }
class Trans   { char event; Guard guard; […] }
class FS extends State { […] }
class Guard   { String exp; }
```

Abstract Syntax Classes

```
interface GuardFsmRev<M, S, F extends S, T, G>
  extends FsmRev<M, S, F, T> {
    G guard(Guard it);
    default G $(Guard it) { return guard(it); }
}
```

REVISITOR *Interface*

```
interpret(State s, String evt) {
  // Find a transition for the input event
  State next = s.out.findFirst[event == evt]
  // Fire it
  next.fire()
}
```

Execution Semantics

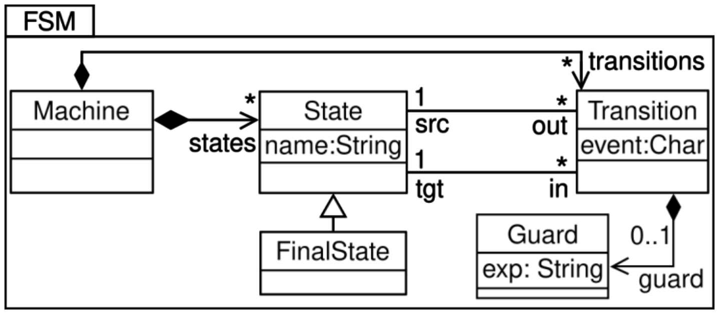
```
pretty-print(State s) {
  print("State " + s.name)
  print("Transitions:")
  for (t in s.transitions)
    pretty-print(t)
}
```

Printing Semantics

```
interface PrintGuardFsm
  extends PrintFsm,
          GuardFsmRev<Pr, Pr, Pr, Pr, Pr> {
  default Pr guard(Guard it) {
    return () -> it.exp;
  }
  @Override
  default Pr trans(Trans it) {
    return () -> super.print()+$(it.guard).print();
  }
}
```

REVISITOR *Implementation*

The ALE Language



Abstract Syntax

```
class Machine { List<State> states; […] }
class State   { String name; […] }
class Trans   { char event; Guard guard; […] }
class FS extends State { […] }
class Guard   { String exp; }
```

Abstract Syntax Classes

```
open class Machine {
  def String print() {
    return "Machine " + self.name + "\n" +
           self.states.map[s | $[s].print()];
  }
}
open class State {
  def String print() {
    return "State " + self.name + "\n" +
           self.outgoing.map[t | $[t].print()];
  }
}
open class Guard {
  def String print() {
    return self.exp;
  }
}
```

ALE: a concise and intuitive language for defining the semantics of metamodel-based languages

```
interface GuardFsmRev<M, S, F extends S, T, G>
  extends FsmRev<M, S, F, T> {
    G guard(Guard it);
    default G $(Guard it) { return guard(it); }
}
```

Revisitor *Interface*

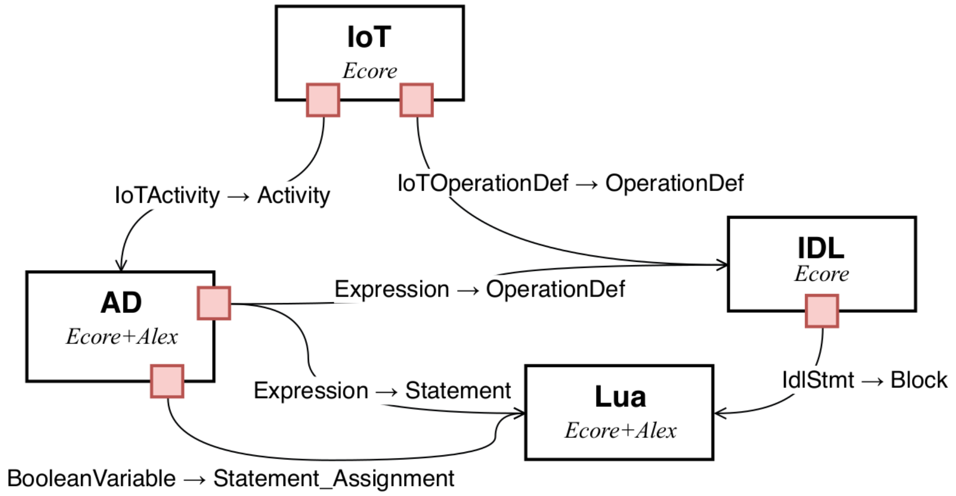
```
interface PrintGuardFsm
  extends PrintFsm,
          GuardFsmRev<Pr, Pr, Pr, Pr, Pr> {
  default Pr guard(Guard it) {
    return () -> it.exp;
  }
  @Override
  default Pr trans(Trans it) {
    return () -> super.print()+$(it.guard).print();
  }
}
```

Revisitor *Implementation*

Summary

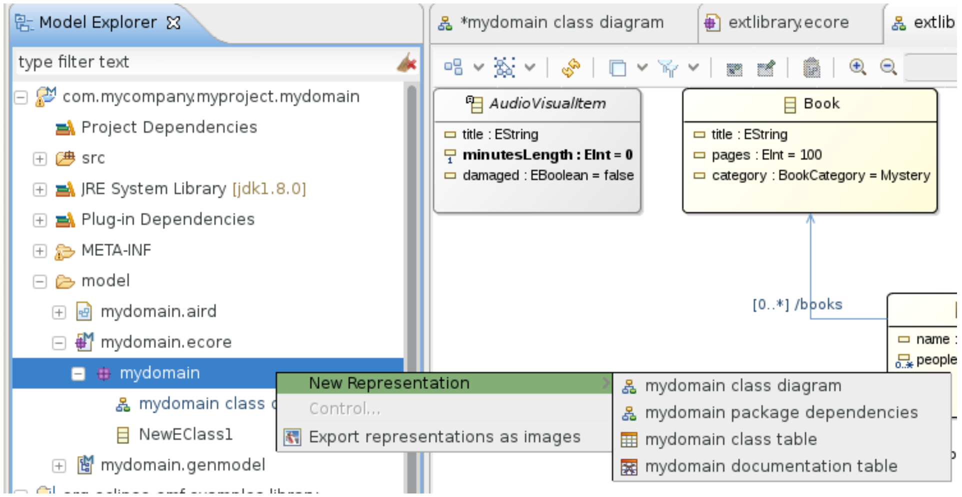
- Scientific contributions
  - First broadly-applicable solution to the *Expression Problem*
    - Applicable in C++, Java, C#, Scala, the *Eclipse Modeling Framework*, etc.
  - Strong theoretical foundations
    - Object algebras, algebraic signatures, Visitor, etc.
    - Object-oriented structural extensibility ∘ functional behavioral extensibility
  - Later extended to modular language *composition*

    Modular Language Composition for the Masses
    M. Leduc, T. Degueule, B. Combemale
    11th International Conference on Software Language Engineering (SLE'18)
    Best Artifact Award



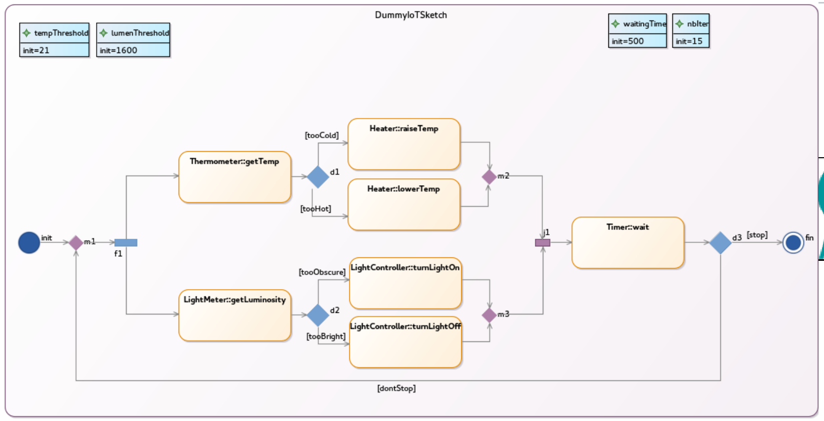
Summary

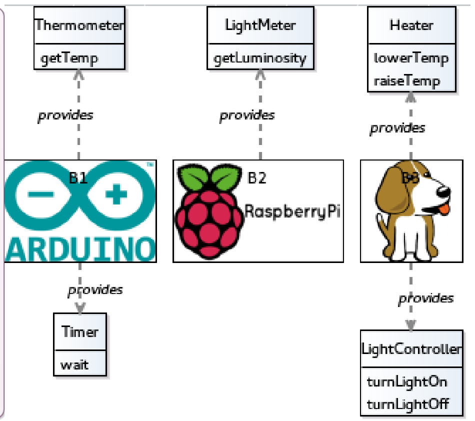
- Technological contributions



[Talk@EclipseCon'17] EcoreTools Next: Executable DSL made (more) accessible

- Experimental validation
  - UML
  - State machines
  - Internet of Things
  - etc.

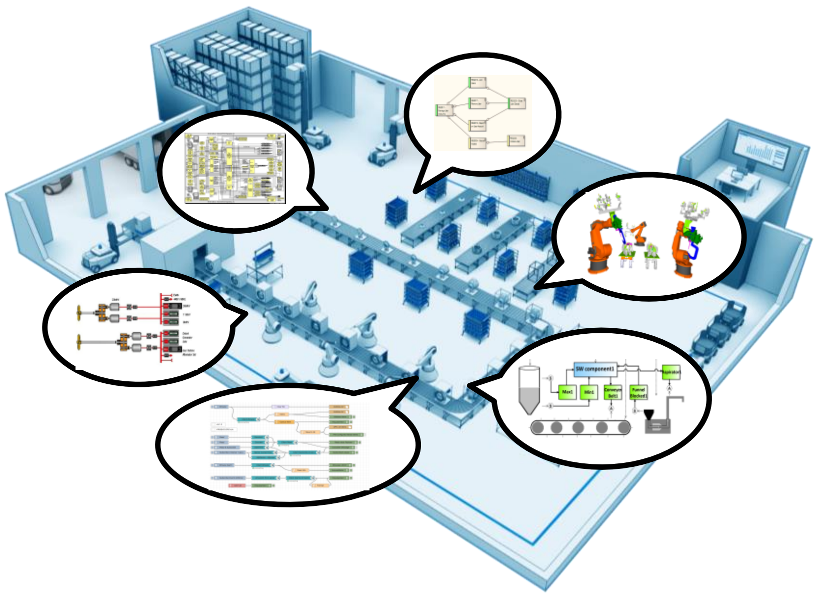




Research Project

Software language diversity
Linguistic analysis of APIs

Software language diversity

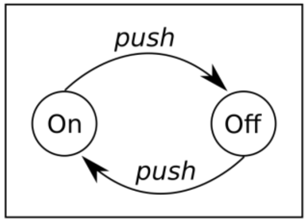


Domain diversity

API

```
FSM fsm =
  new FSMBuilder("Button")
  .init("On")
    .to("Off").on("push")
  .state("Off")
    .to("On").on("push")
  .build();
```

Graphical



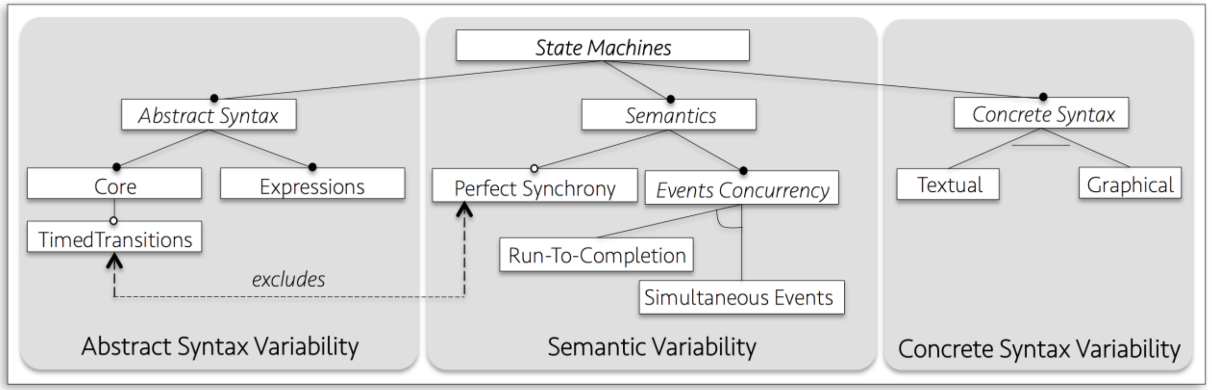
Textual

```
machine Button
  state On
    on push ⇒ Off
  end
  state Off
    on push ⇒ On
  end
end
```

Annotations

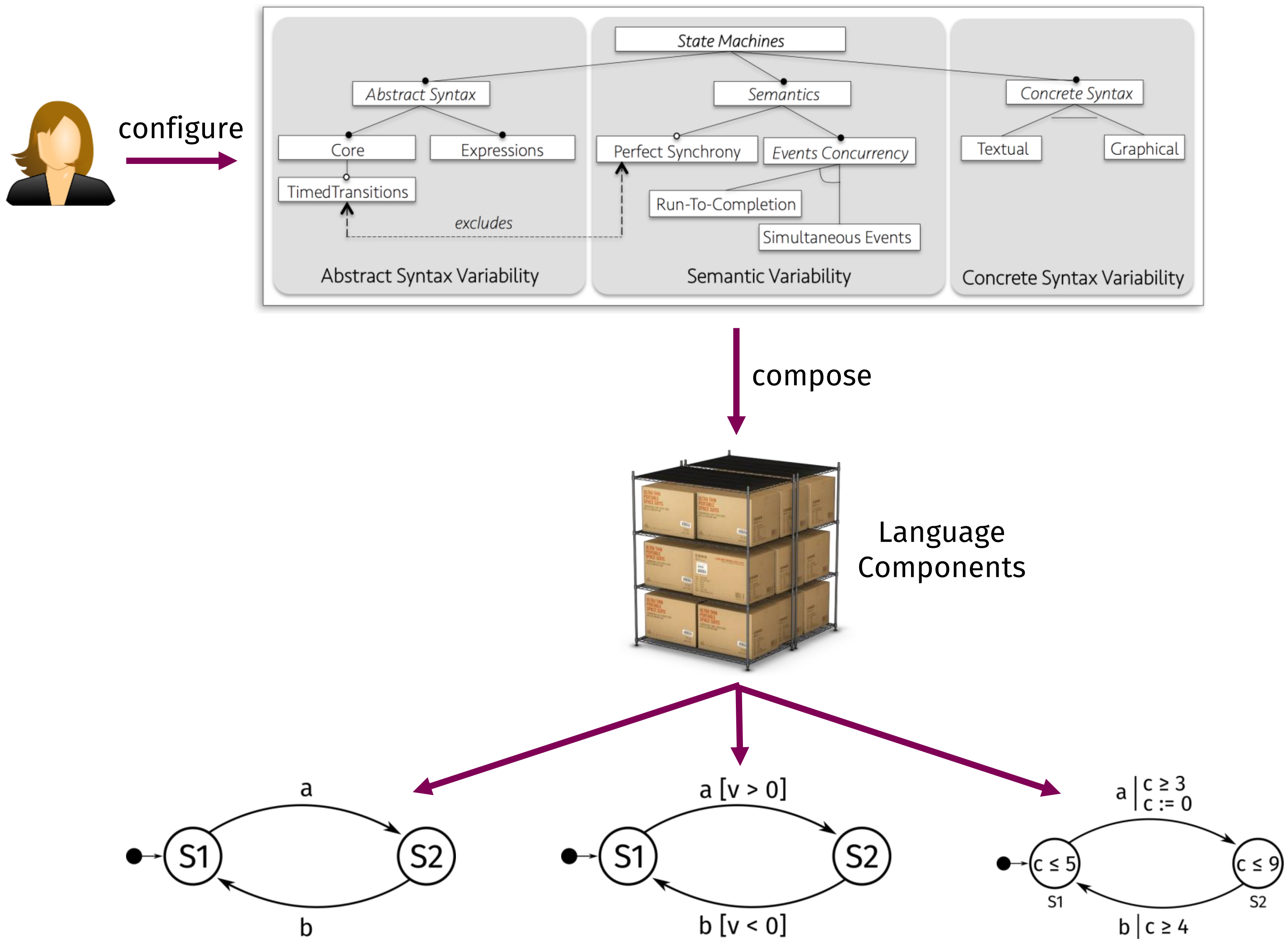
```
class Button {
  @State static String ON  = "On";
  @State static String OFF = "Off";
  @Trans(on="push", in=ON, out=OFF)
  public void doPush() {
  }
}
```

Shape diversity



Abstraction diversity

Language Families



Concours N°06/02 Section 6 du CoNRS

Apprivoiser et synthétiser la diversité des langages logiciels

Thomas Degueule
Software Analysis and Transformation Group
Centrum Wiskunde & Informatica
https://tdegueul.github.io

Intégration
UMR 9189 – CRYStAL– Lille
UMR 5505 – IRIT – Toulouse