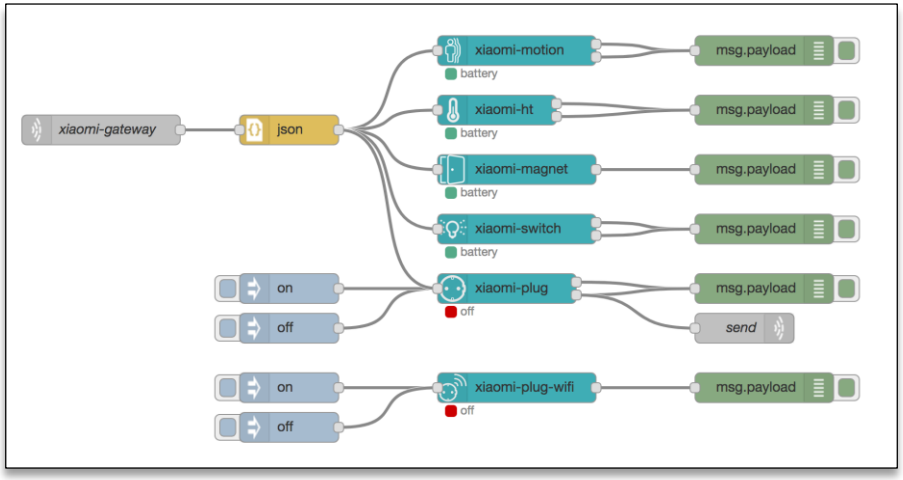
Concours N°06/02

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



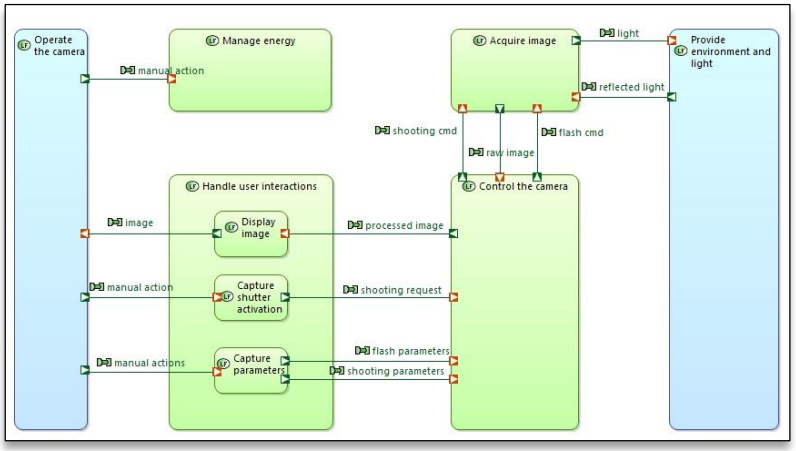
Node-RED

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;
```

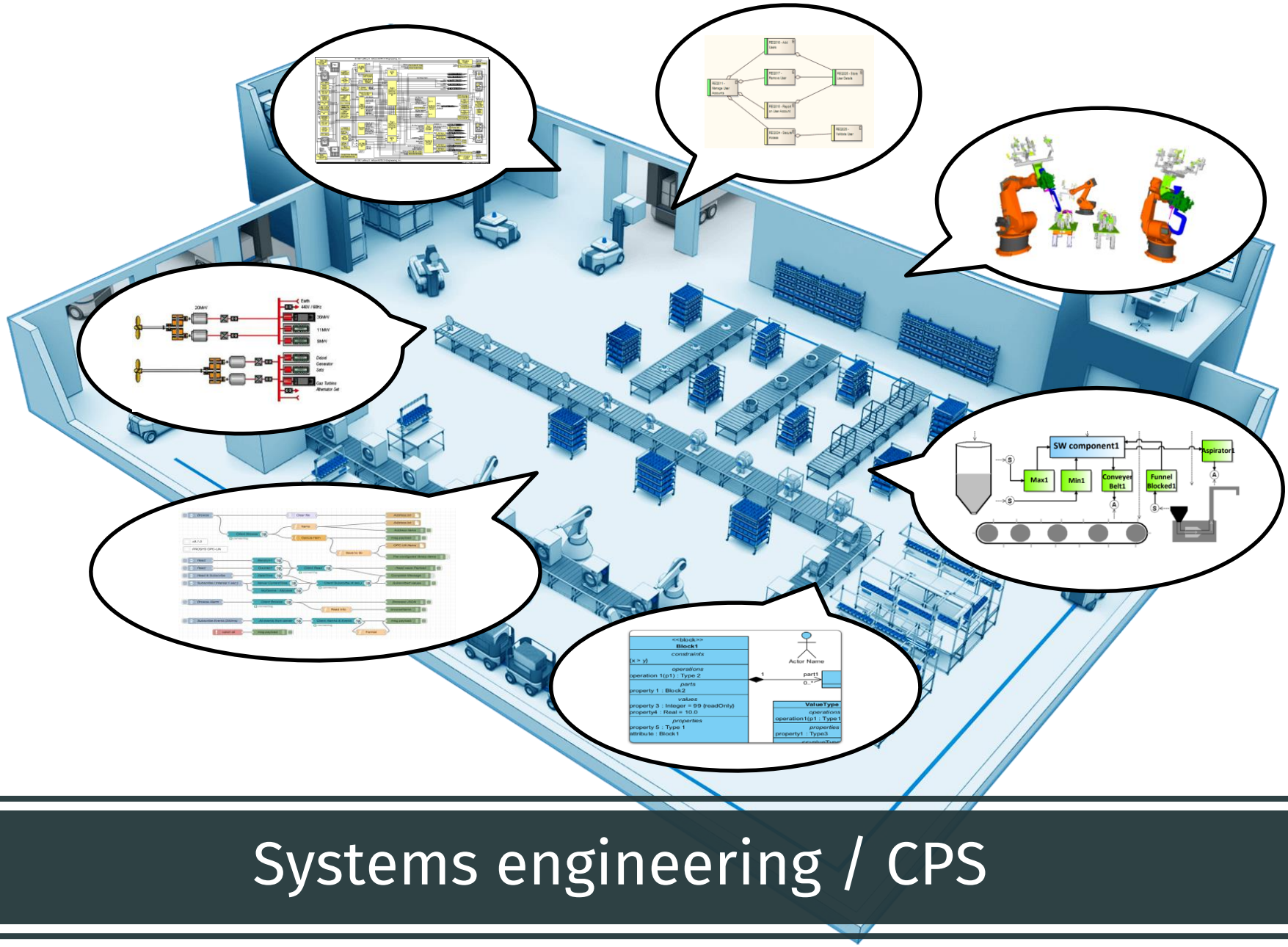
VHDL

Hardware Description Language

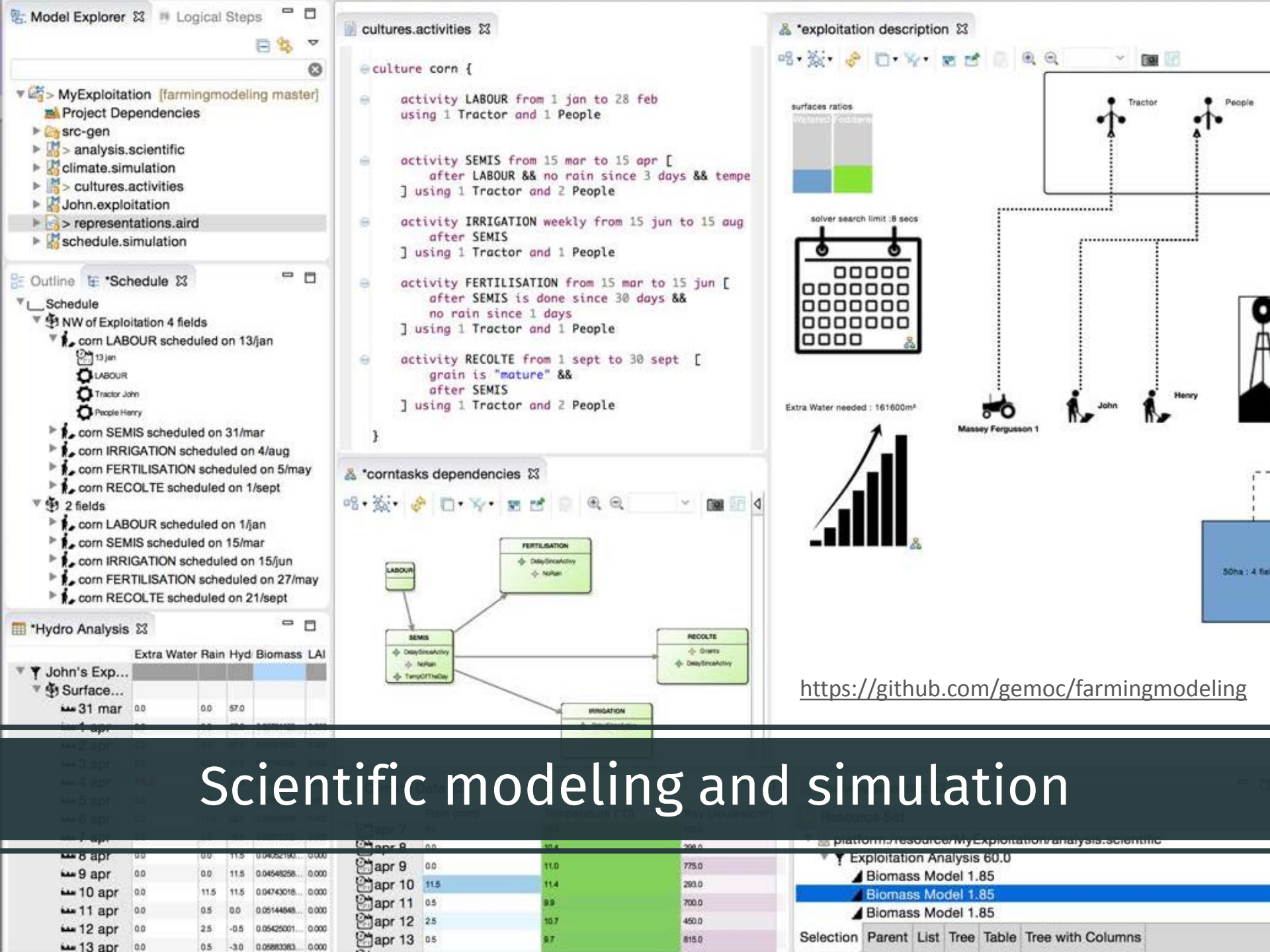


Capella

Systems Engineering Language

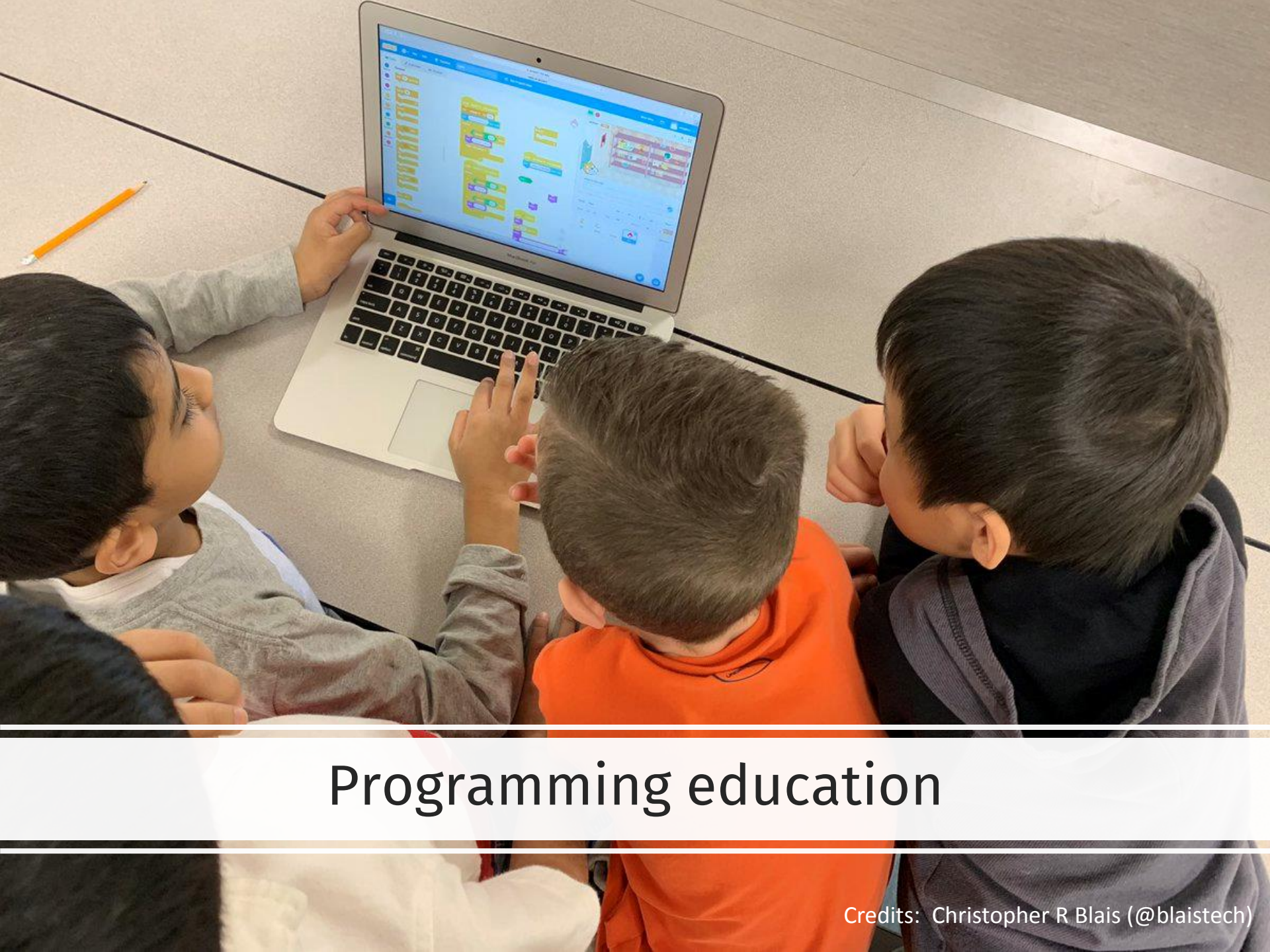


Systems engineering / CPS



https://github.com/gemoc/farmingmodeling

Scientific modeling and simulation



Programming education

Credits: Christopher R Blais (@blaistech)

Software Language Engineering (SLE)

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/

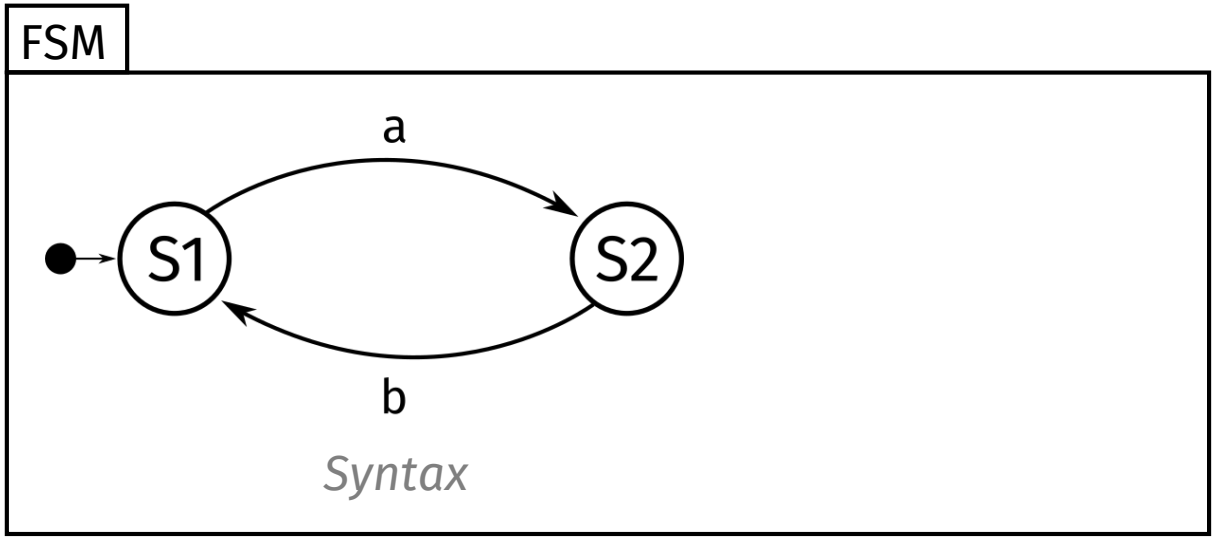
- Industrial collaborations

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

Modular Language Extension

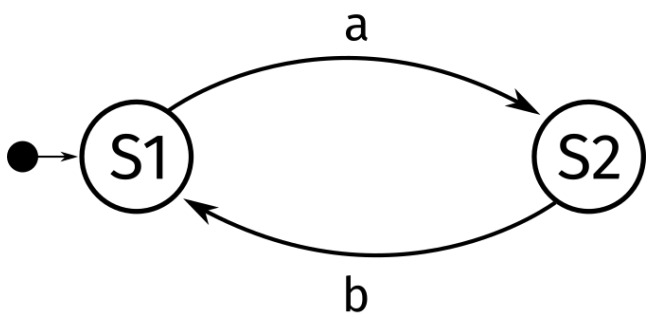
Revisiting Visitors for Modular Extension of Executable DSMLs
Manuel Leduc, Thomas Degueule, Benoit Combemale, Tijs van der Storm, Olivier Barais
20th International Conference on Model Driven Engineering Languages and Systems (MODELS'17)

Modular language extension



Modular language extension

FSM



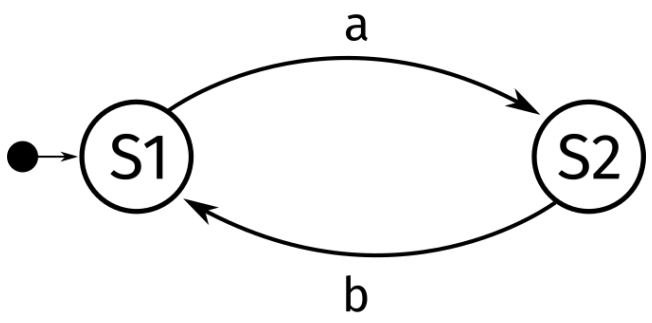
Syntax

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

Interpreter

Modular language extension

FSM

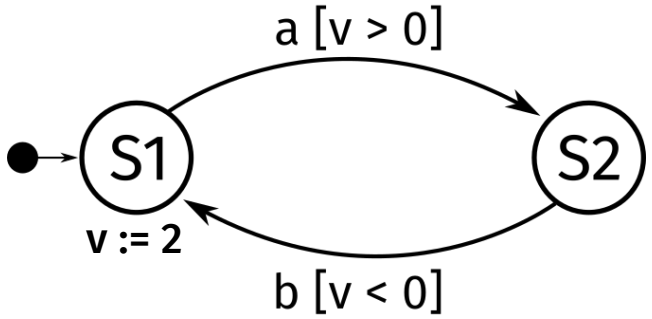


Syntax

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

Interpreter

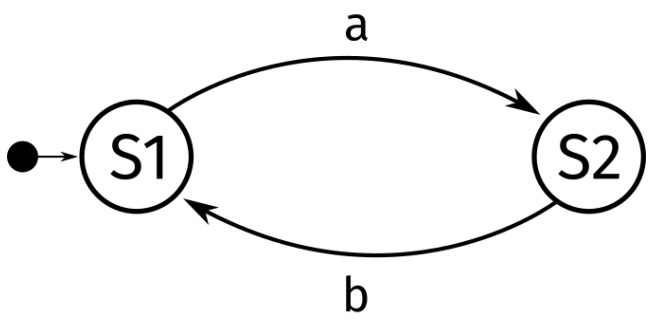
GuardedFSM



Syntax'

Modular language extension

FSM

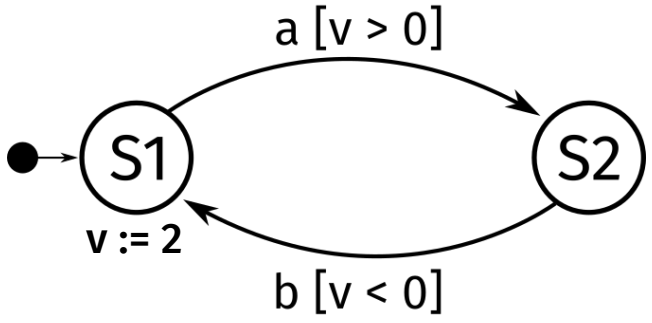


Syntax

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

Interpreter

GuardedFSM



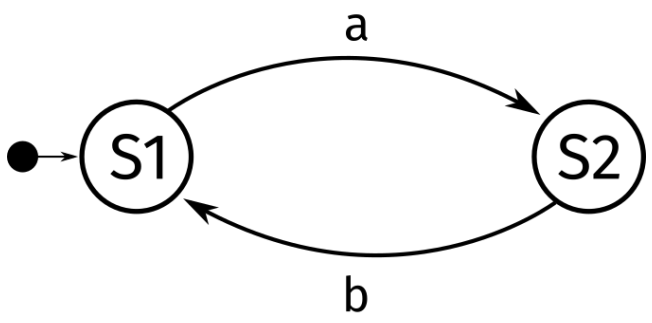
Syntax'

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

Interpreter'

Modular language extension

FSM

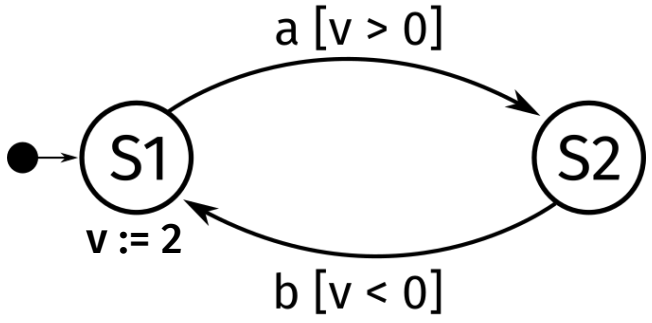


Syntax

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

Interpreter

GuardedFSM



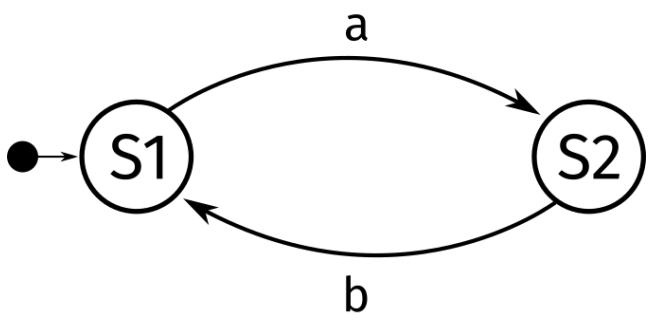
Syntax'

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

Interpreter'

Modular language extension

FSM



Syntax

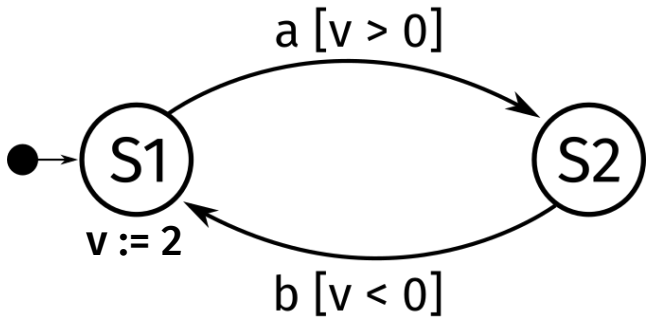
```
eval(fsm, 'ab'):
  s1 -> s2
  s2 -> s1
```

Interpreter

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

Pretty-printer

GuardedFSM



Syntax'

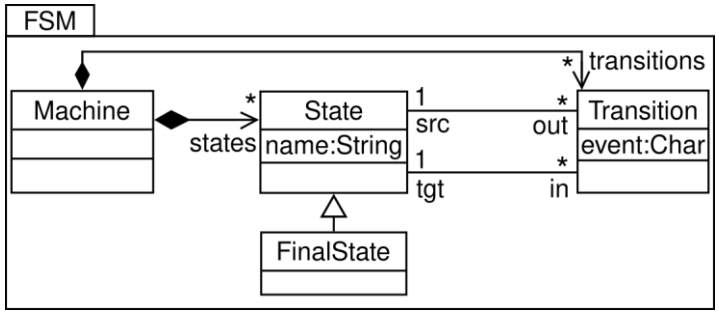
```
eval(gfsm, 'ab'):
  s1 -> s2
  <deadlock>
```

Interpreter'

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

Pretty-printer'

Modular language extension

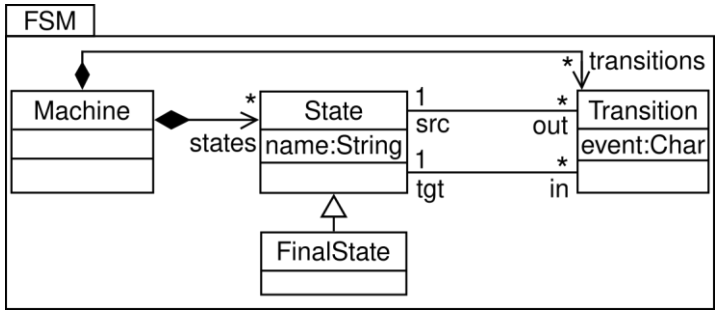


Abstract Syntax

```
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

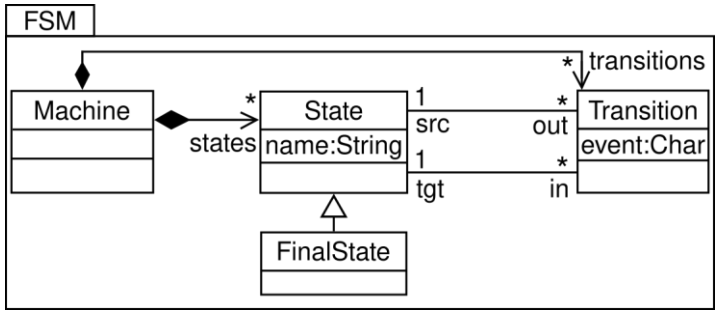
```
class Machine { List<State> states; […] }
class State   { String name; […] }
class Trans   { char event; […] }
class FS extends State { […] }
```

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



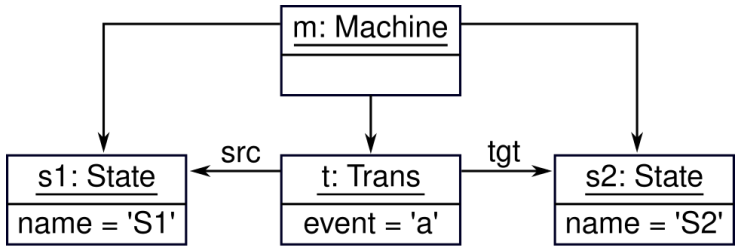
Abstract Syntax

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

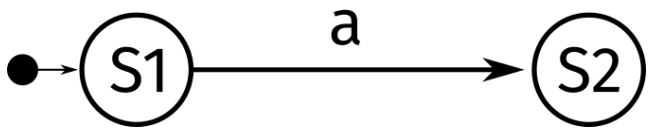
Execution Semantics

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

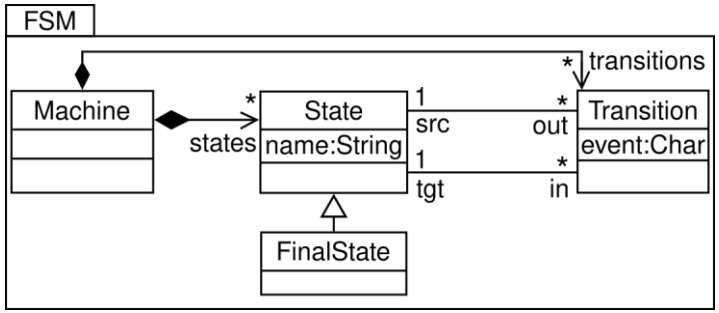
Abstract Syntax Classes



Abstract Syntax Graph



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; […] }
class FS extends State { […] }
```

Abstract Syntax Classes

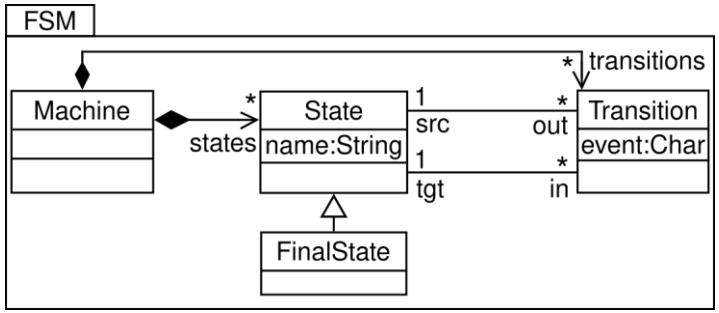
```
interface Interpret { void interpret(); }
class Machine implements Interpret { […] }
class State   implements Interpret { […] }
class Trans   implements Interpret { […] }
```

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

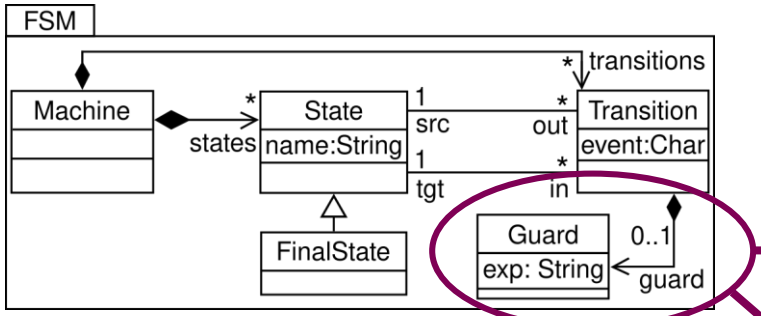
```
pretty-print(State s) {
  print("State " + s.name)
  print("Transitions:")
  for (t in s.transitions)
    pretty-print(t)
}
```

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

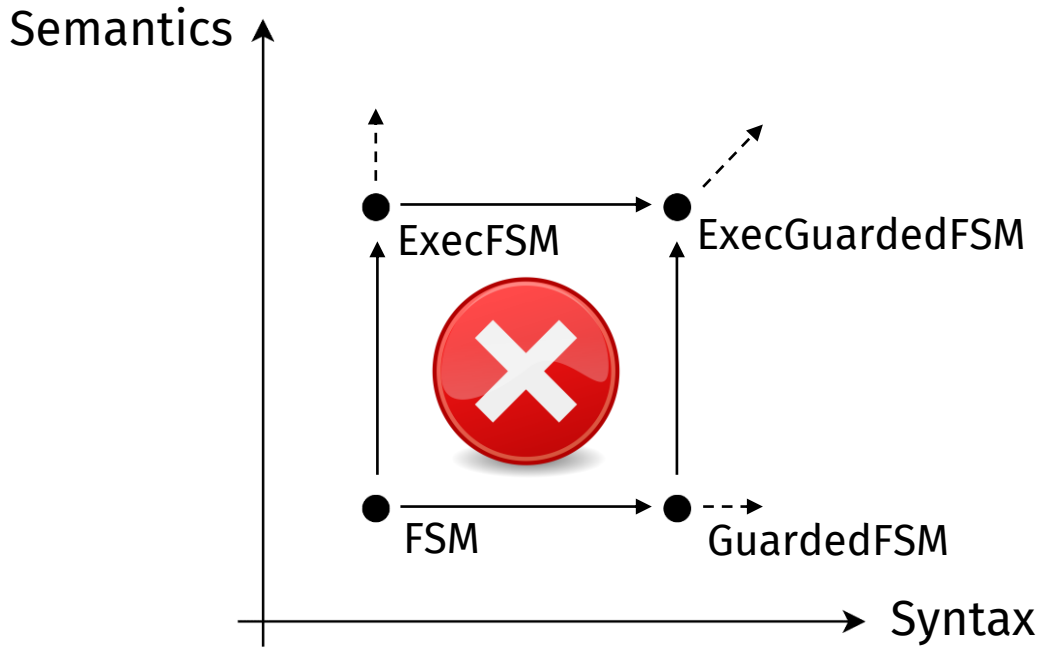
```
interface Interpret { void interpret(); }
class Machine implements Interpret { […] }
class State   implements Interpret { […] }
class Trans   implements Interpret { […] }
class Guard   implements Interpret { […] }
```

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*

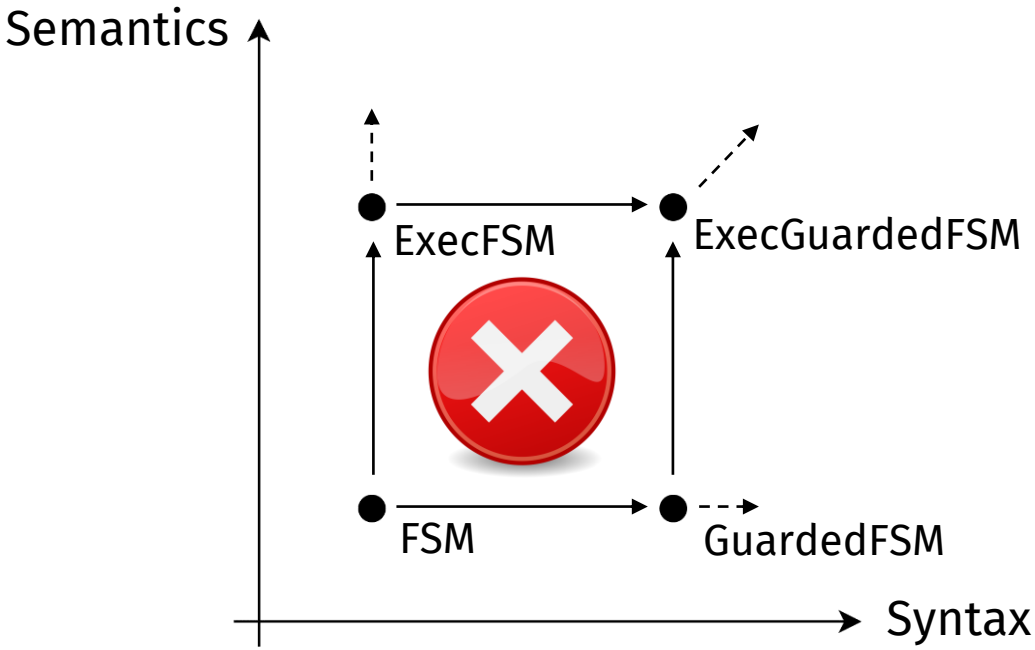


"The expression problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety."

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*

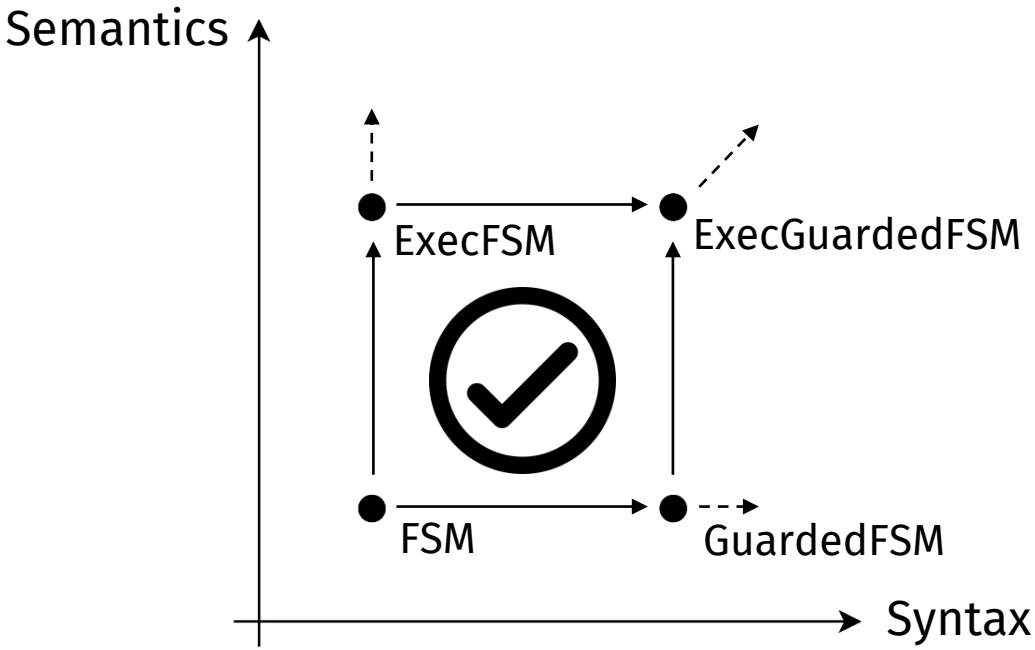


"The expression problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety."

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]

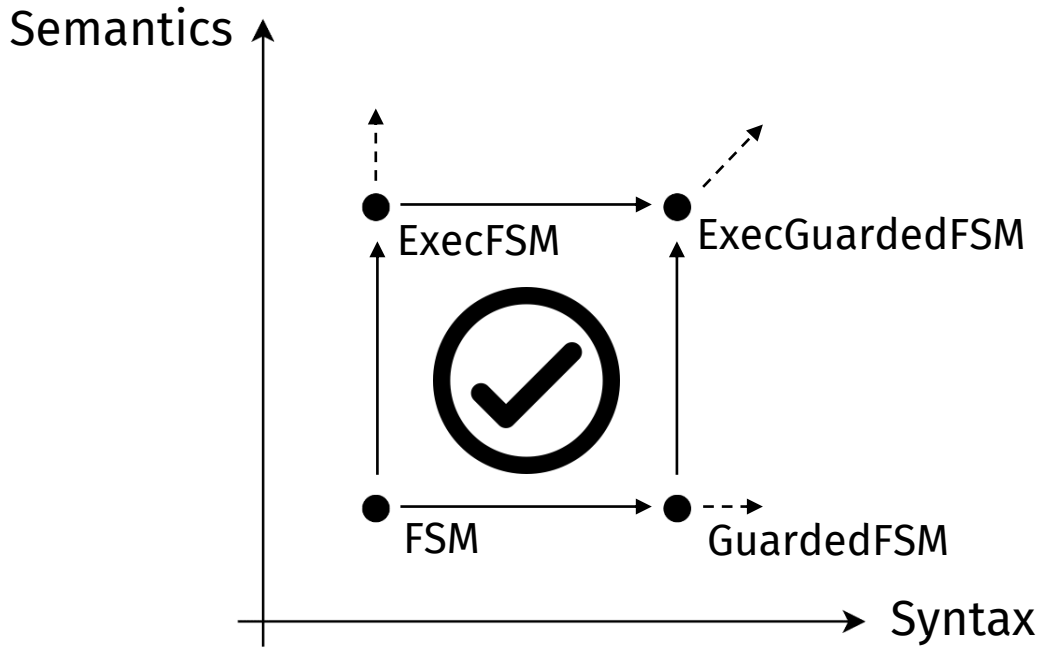
[1]National University of Singapore
bruno@ropas.snu.ac.kr
[2] University of Texas, Austin
wcook@cs.utexas.edu

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

- An encoding of algebraic signatures in the object-oriented paradigm

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

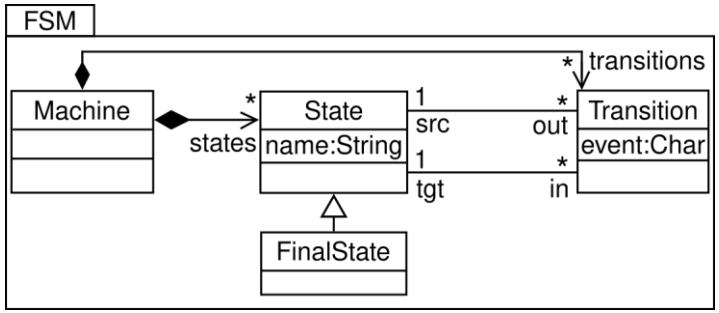
Tijs van der Storm
CWI & U of Groningen
The Netherlands
storm@cwi.nl

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

```
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

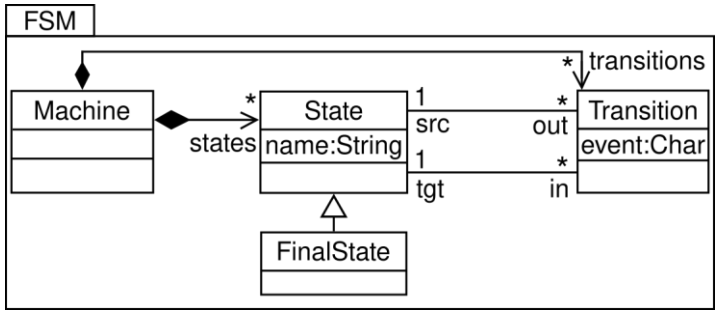
```
class Machine { List<State> states; […] }
class State   { String name; […] }
class Trans   { char event; […] }
class FS extends State { […] }
```

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*

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

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

Abstract Syntax Classes

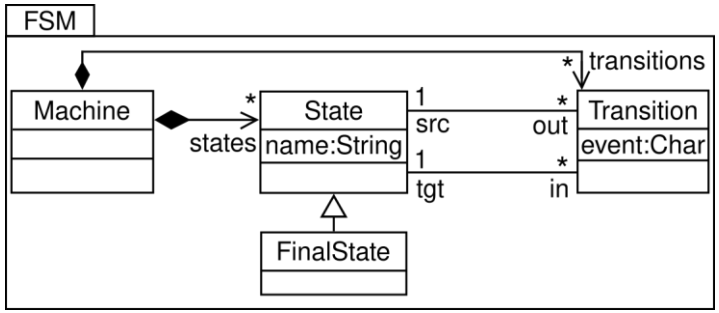
```
interface FsmRev<M, S, F extends S, T> {
  M machine(Machine it);
  T trans  (Trans it);
  default M $(Machine it) { return machine(it); }
  default T $(Trans it)   { return trans(it);   }
  […]
}
```

REVISITOR *Interface*

```
interface 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

```
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 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*

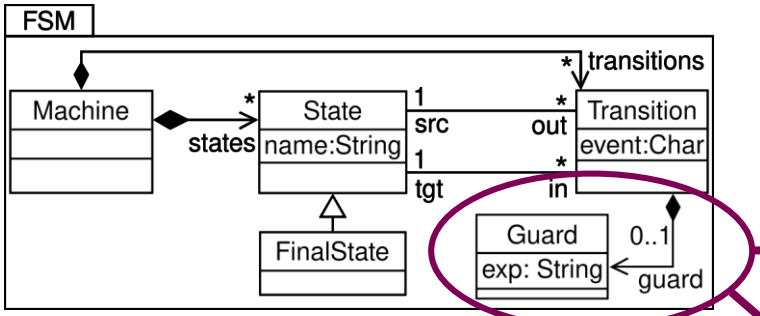
```
pretty-print(State s) {
  print("State " + s.name)
  print("Transitions:")
  for (t in s.transitions)
    pretty-print(t)
}
```

Printing Semantics

```
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

```
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; Guard guard; […] }
class FS extends State { […] }
class Guard   { String exp; }
```

Abstract Syntax Classes

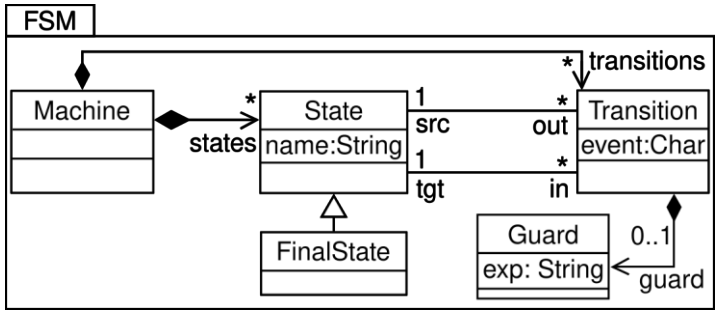
```
interface GuardFsmRev<M, S, F extends S, T, G>
  extends FsmRev<M, S, F, T> {
    G guard(Guard it);
    default G $(Guard it) { return guard(it); }
}
```

REVISITOR *Interface*

```
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*

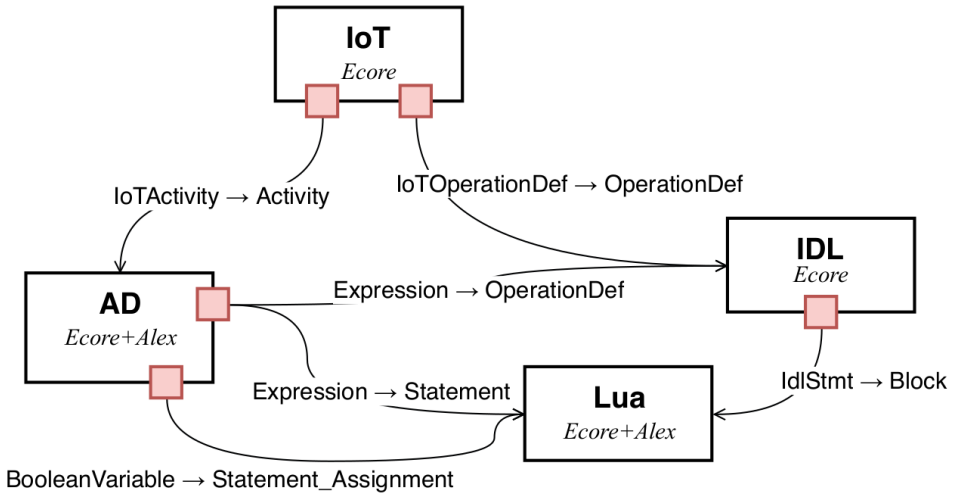
```
interface PrintGuardFsm
  extends PrintFsm,
          GuardFsmRev<Pr, Pr, Pr, Pr, Pr> {
  default Pr guard(Guard it) {
    return () -> it.exp;
  }
  @Override
  default Pr trans(Trans it) {
    return () -> super.print()+$(it.guard).print();
  }
}
```

REVISITOR *Implementation*

Summary

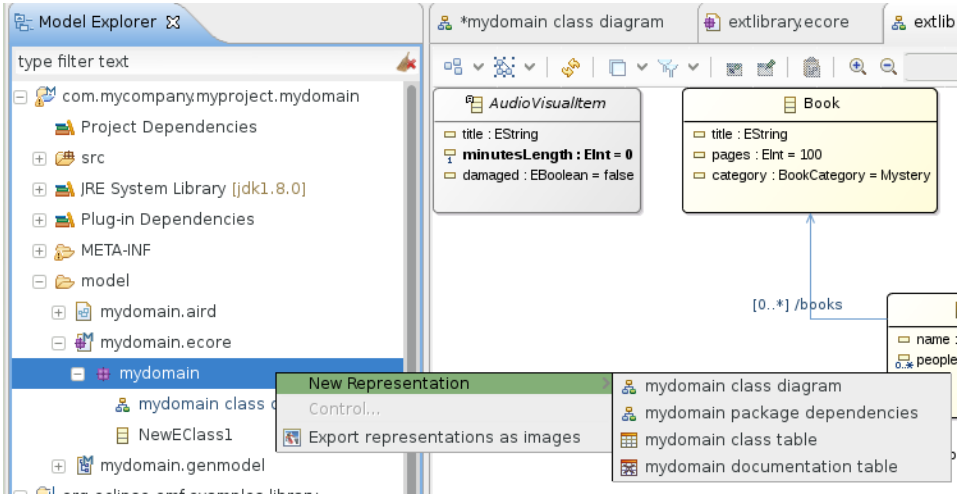
- Scientific contributions
  - First broadly-applicable solution to the *Expression Problem*
    - Applicable in C++, Java, C#, Scala, the *Eclipse Modeling Framework*, etc.
  - Strong theoretical foundations
    - Object algebras, Visitors, etc.
    - Object-oriented structural extensibility ∘ functional 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



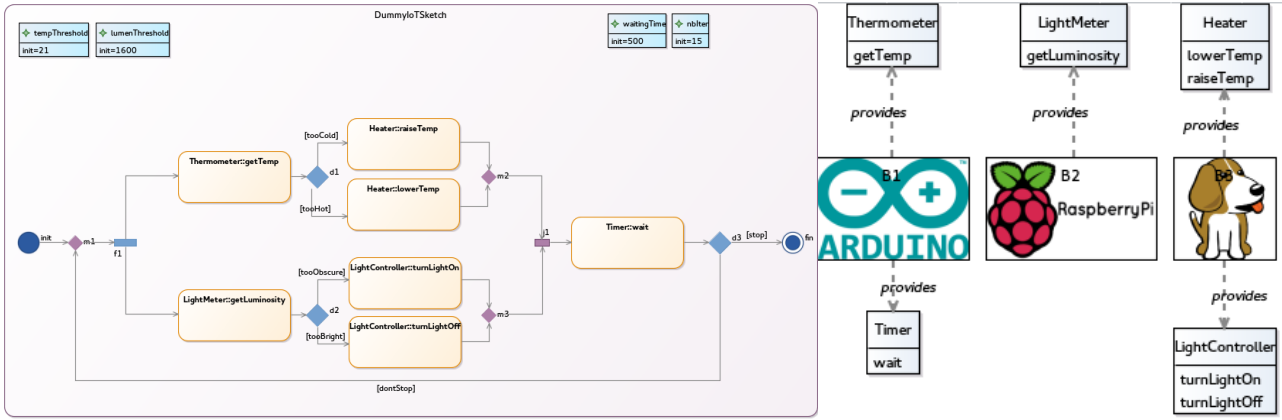
Summary

- Technological contributions



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

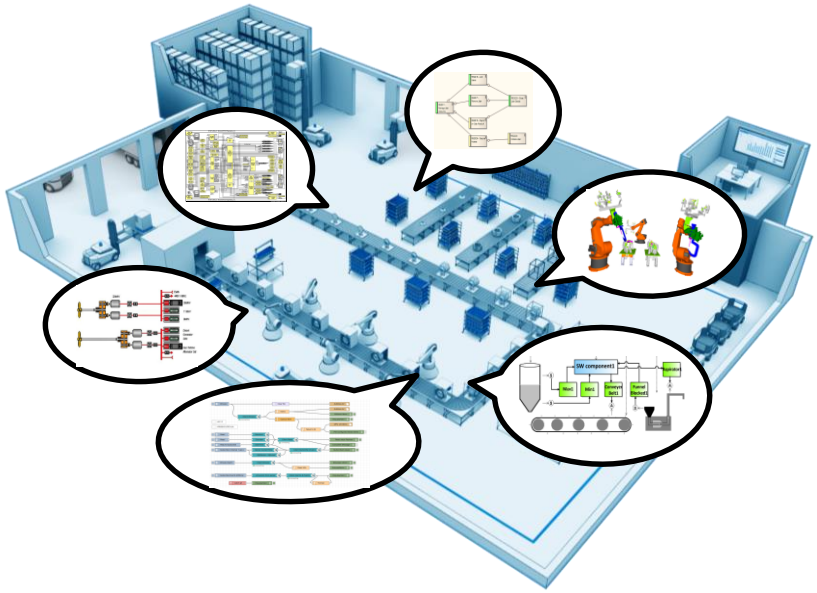
- Experimental validation
  - UML
  - State machines
  - Internet of Things
  - etc.



Research Project

Software language diversity
Linguistic analysis of APIs

Software language diversity

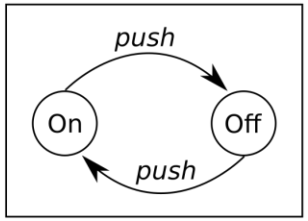


Domain diversity

API

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

Graphical



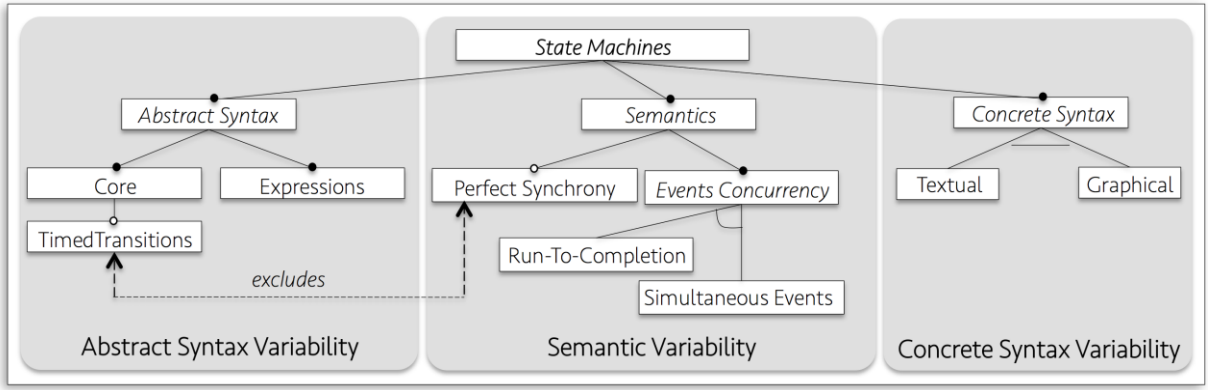
Textual

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

Annotations

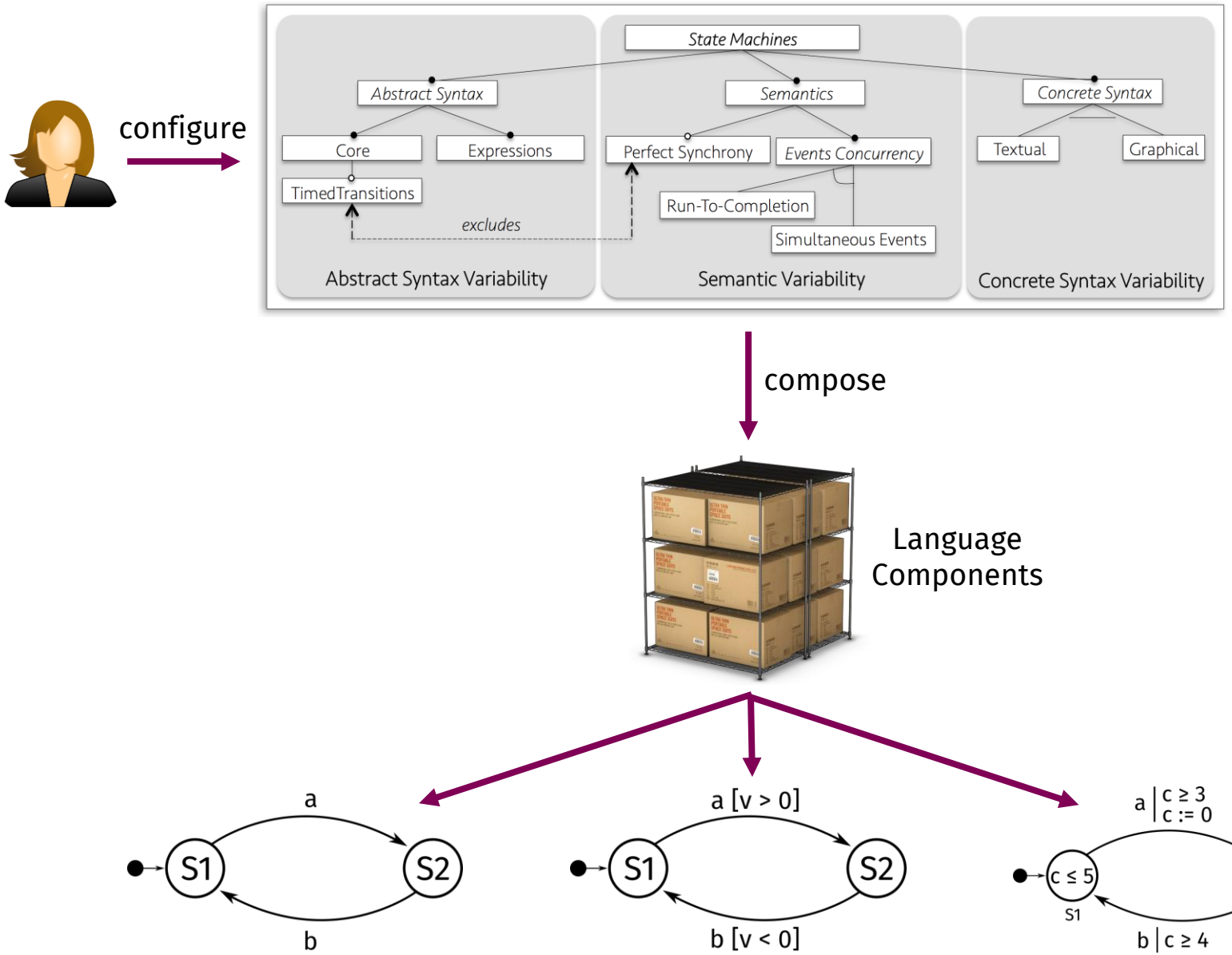
```
class Button {
  @State static String ON  = "On";
  @State static String OFF = "Off";
  @Trans(on="push", in=ON, out=OFF)
  public void doPush() {
  }
}
```

Shape diversity



Abstraction diversity

Language Families



Linguistic Analysis of APIs

- APIs are brittle, hard to learn, constantly evolving

```
MongoClient client = new MongoClient("mongodb://db.server");
DB db = client.getDB("MyDB");
DBCollection col = db.getCollection("MyCollection");
col.insert(new BasicDBObject([...]));
DBCursor cursor = col.find(new BasicDBObject([...]));
DBObject obj = cursor.one();
String name = obj.get("name");
```

- Intuition: reify APIs as languages

```
connectTo “mongodb://db.server”
  col = get MyCollection from MyDB
  insert [...] in col
  cursor = find [...] in col
  name = cursor[name]
```

- Apply linguistic methods to the analysis of APIs
  - Co-evolution of client code ↔ API ⟺ Co-evolution of model ↔ language

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

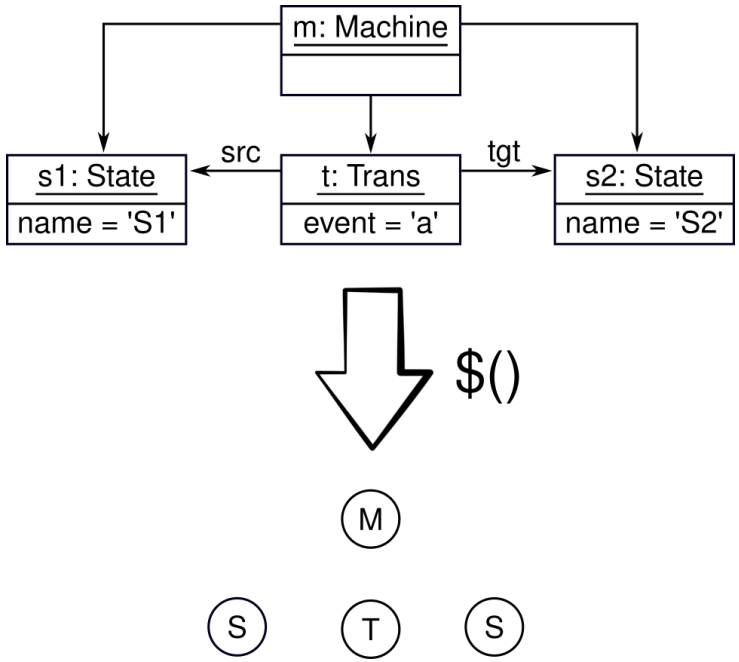
Intégration
UMR 9189 – CRYStAL– Lille
UMR 5505 – IRIT – Toulouse

<Backup>

Revisitor *Interface*

- Extensible mapping between syntactic objects and abstract semantic types

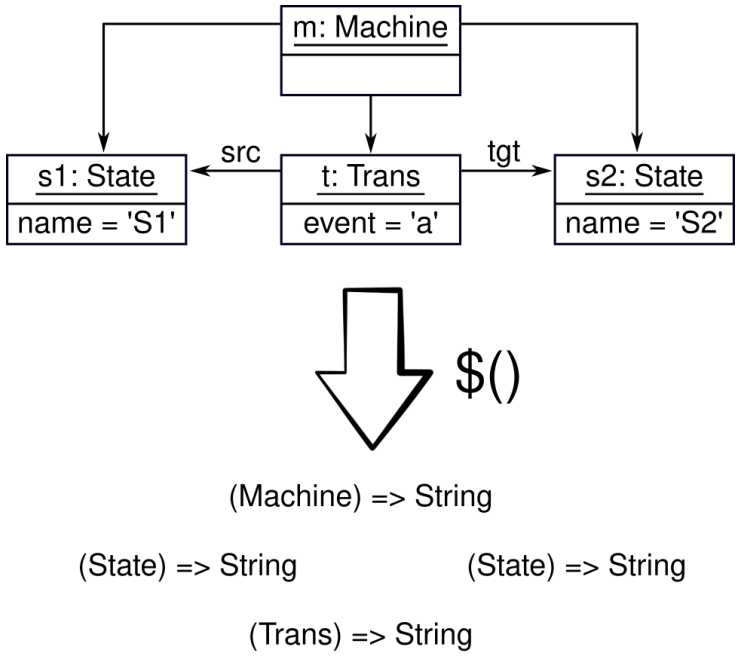
```
interface FsmRev<M, S, F extends S, T> {
  M machine(Machine it);
  S state  (State it);
  F fState (FState it);
  T trans  (Trans it);

  default M $(Machine it) { return machine(it); }
  default F $(FState it)  { return fState(it);  }
  default T $(Trans it)   { return trans(it);   }
  default S $(State it)   {
    if (it instanceof FState)
      return fState(it);
    return state(it);
  }
}
```



Revisitor *Implementation*

- Assign concrete types to the semantic objects
- Retroactive implementation of the semantics

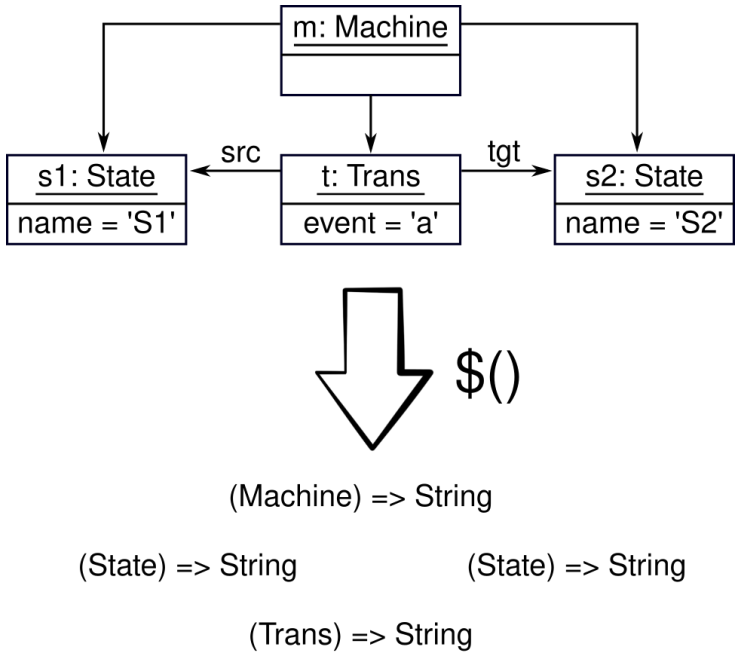
```
interface Pr { String print(); }

interface PrintFsm extends FsmRev<Pr, Pr, Pr, Pr> {









}
```



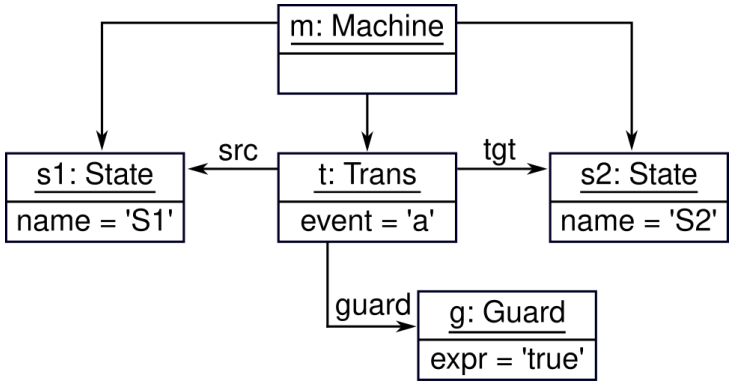
Revisitor *Implementation*

- Assign concrete types to the semantic objects
- Retroactive implementation of the semantics

```
interface Pr { String print(); }

interface PrintFsm extends FsmRev<Pr, Pr, Pr, Pr> {
  default Pr machine(Machine it) {
    return () -> it.name + ":\n" +
                 it.states.map(s -> $(s).print());
  }
  default Pr state(State it) {
    return () -> /*    print a state    */;
  }
  default Pr fState(FState it) {
    return () -> /* print a final state */;
  }
  default Pr trans(Trans it) {
    return () -> /* print a transition  */;
  }
}
```



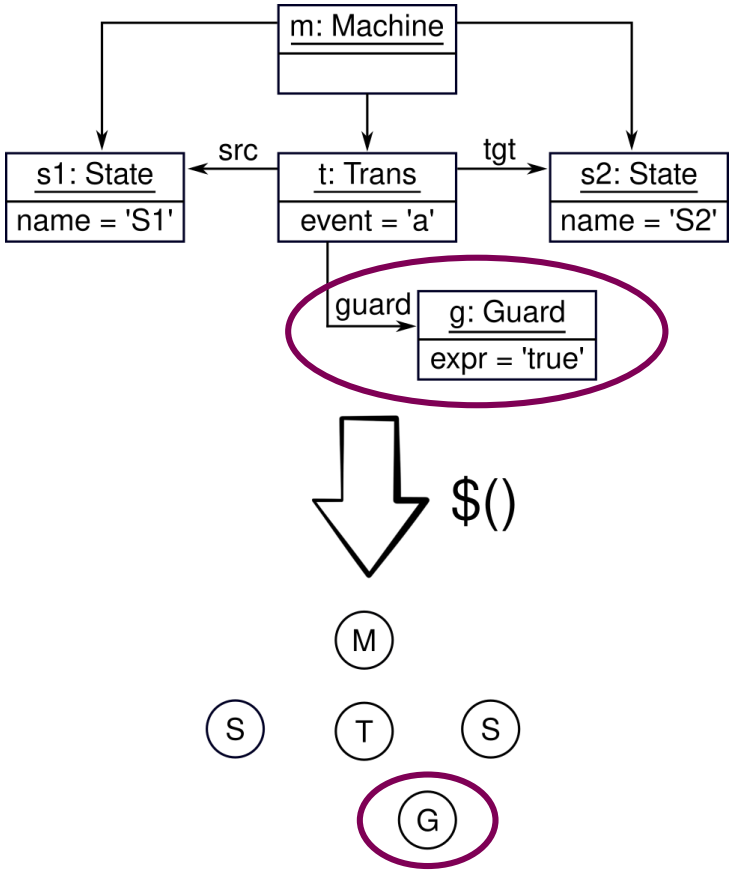
Modular Extension with Revisitors



Modular Extension with Revisitors

```
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); }
}
```

Modularly extend the mapping



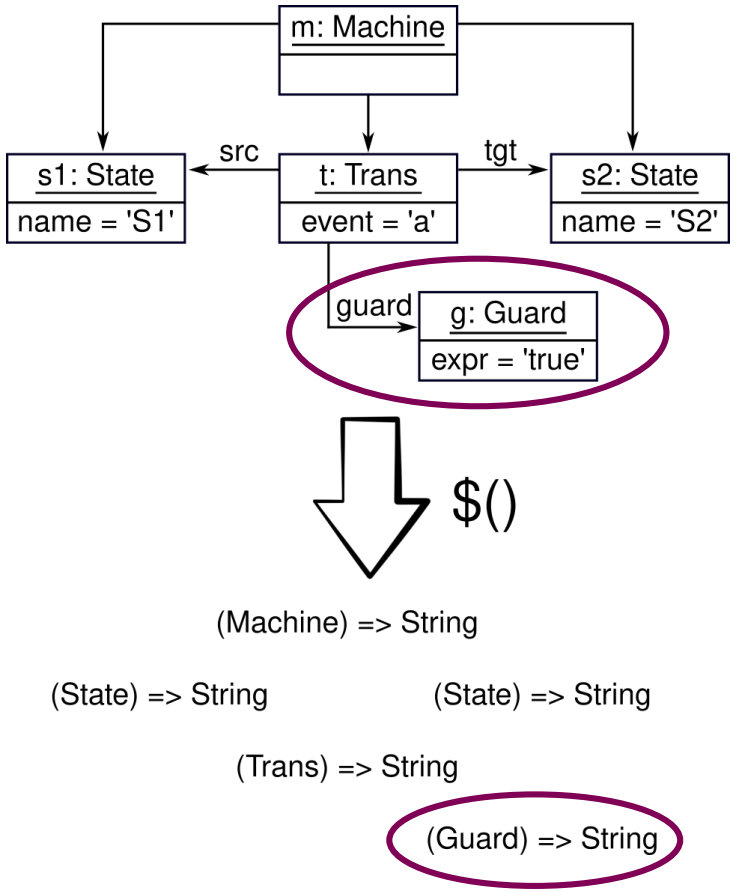
Modular Extension with Revisitors

```
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); }
}
```

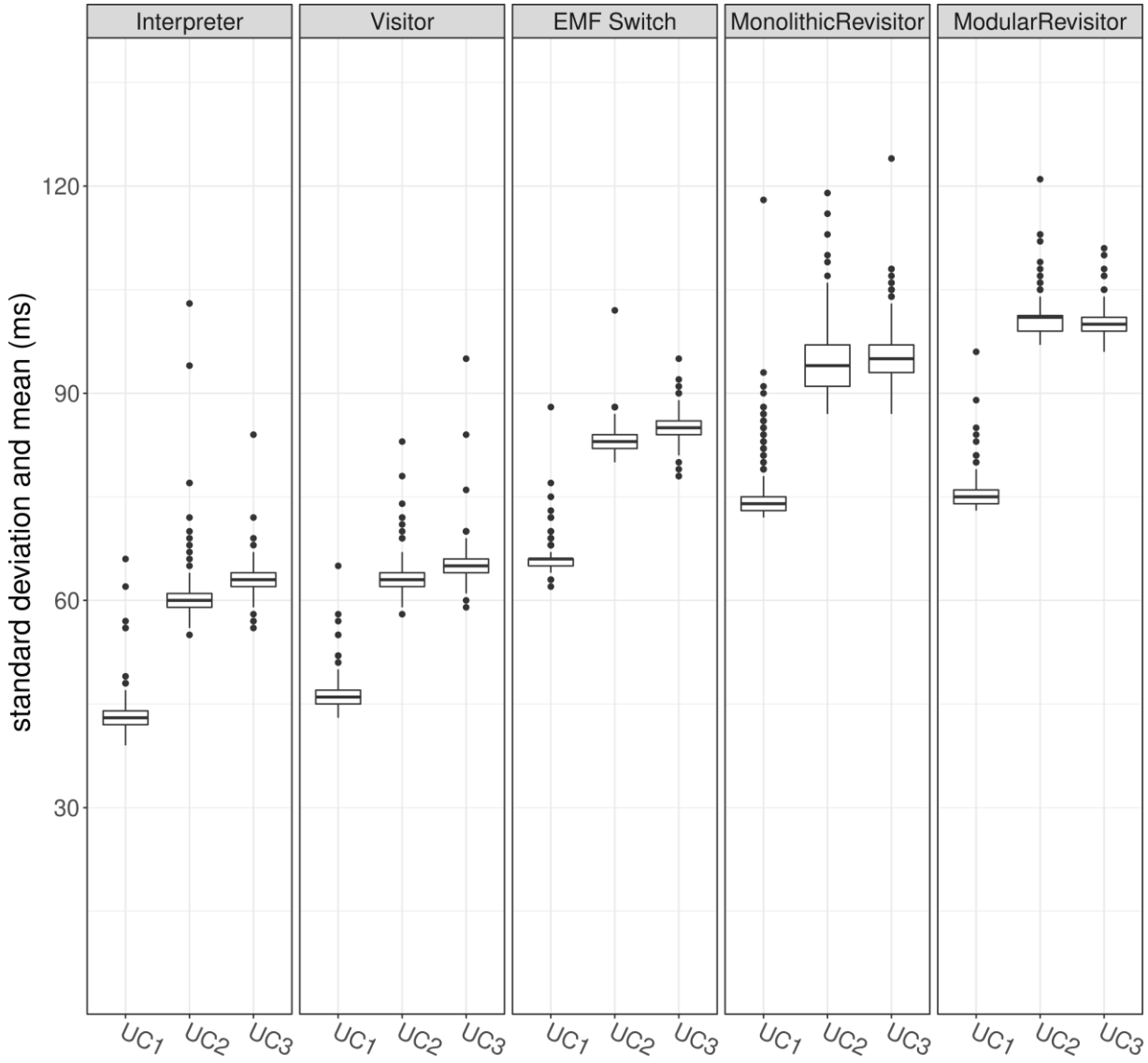
Modularly extend the mapping

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

Modularly extend the semantics



Revisitors Benchmark



Specify Equality Condition

To specify equality conditions, use the `com.mongodb.client.model.Filters.eq_` method to create the query filter document:

```
and(eq( <field1>, <value1>), eq( <field2>, <value2>) ...)
```

The following example selects from the `inventory` collection all documents where the `status` equals `"D"`:

```
findIterable = collection.find(eq("status", "D"));
```

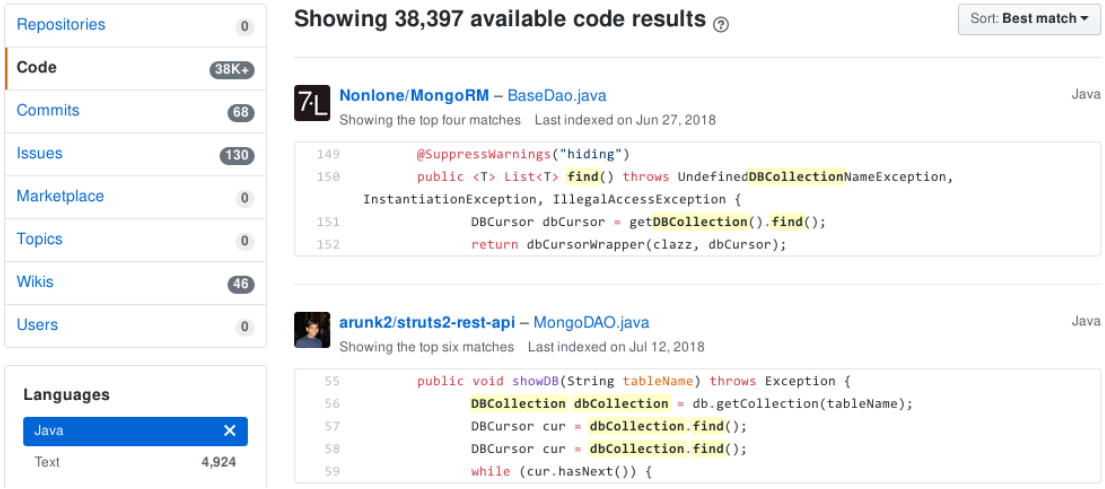
This operation corresponds to the following SQL statement:

```
SELECT * FROM inventory WHERE status = "D"
```

Official Documentation

DBCursor	find(DBObject query) Select documents in collection and get a cursor to the selected documents.
DBCursor	find(DBObject query, DBCollectionFindOptions options) Select documents in collection and get a cursor to the selected documents.
DBCursor	find(DBObject query, DBObject projection) Select documents in collection and get a cursor to the selected documents.
DBCursor	find(DBObject query, DBObject projection, int numToSkip, int batchSize) **Deprecated.** use DBCursor.skip(int) and DBCursor.batchSize(int) on the DBCursor returned from find(DBObject, DBObject)
DBCursor	find(DBObject query, DBObject projection, int numToSkip, int batchSize, int options) **Deprecated.** use DBCursor.skip(int), DBCursor.batchSize(int) and DBCursor.setOptions(int) on the DBCursor returned from find(DBObject, DBObject)

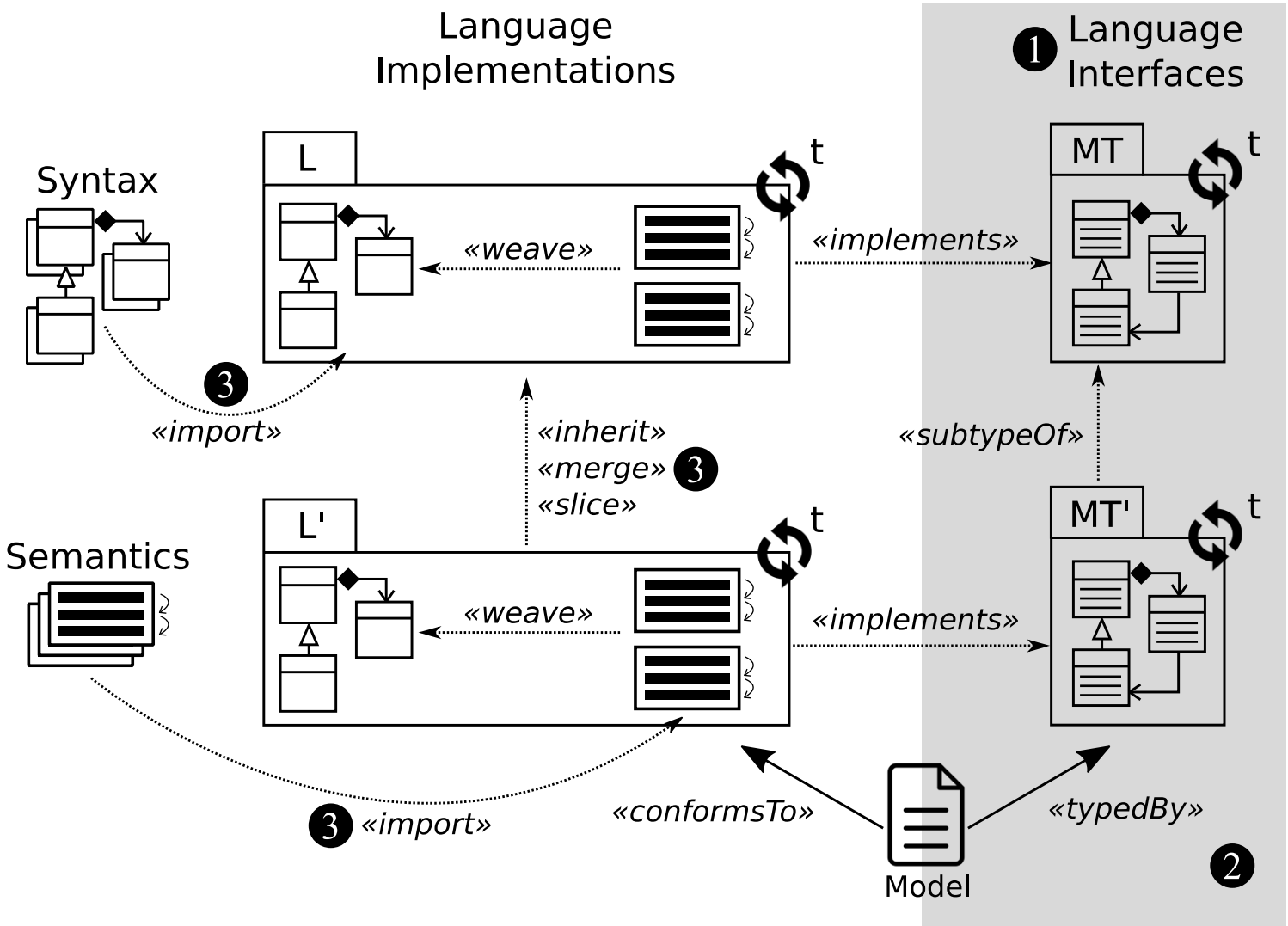
API Documentation



Similar Projects

Q&A Websites

Thèse (2013 – 2016)



Post-doctorat (2017 – 2020)

- Groupe Software Analysis and Transformation au CWI, Amsterdam

- CWI—Inria ALE (*Agile Language Engineering*)
  - Implémentation modulaire de langages logiciels
  - Modélisation flexible

SCAVA http://eclipse.org/scava/

[MODELS'17][SLE'18$_1$][COMLAN'18]
[SLE'18$_2$][SLE'18$_3$]

- H2020 CROSSMINER (*Developer-centric Knowledge Mining from OSS Repositories*)
  - Génie logiciel empirique, rétro-ingénierie
  - Analyse d'APIs, systèmes de recommandations

[MSR'18]
[ICSE'19][ICSME'19 – en rédaction]

CWI

Lina Ochoa
ICSE'19 (accepté)
MSR'18 (publié)

Inria INVENTEURS DU MONDE NUMÉRIQUE

Manuel Leduc
MODELS'17 (publié)
COMLAN'18 (publié)
SLE'18$_1$ (publié)

UNIVERSITÉ TOULOUSE Jean Jaurès

Fabien Coulon
SLE'18$_3$ (publié)