Modular Extension of Domain-specific Languages

Thomas Degueule
Software Analysis & Transformation
CWI

https://tdegueul.github.io

CWI Scientific Meeting
24 November 2017

Foreword

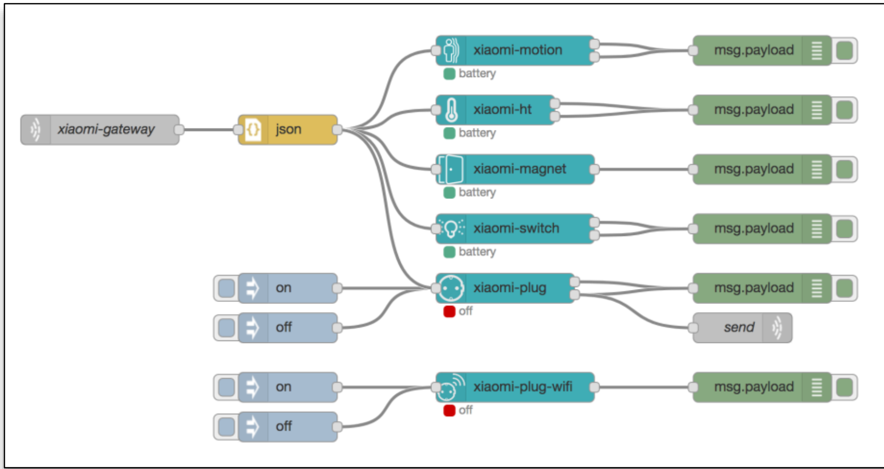
- Joint work with the DiverSE group @ Inria
- CWI—Inria associate team *Agile Language Engineering* (ALE)

Revisiting Visitors for Modular Extension of Executable DSMLs

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

MODELS 2017
austin, tx

20th ACM/IEEE International Conference on Model-driven Engineering Languages and Systems



Node-RED

Wiring the Internet of Things

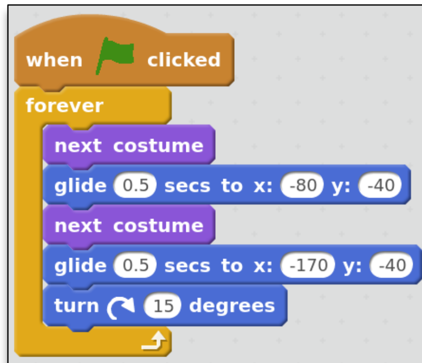
```
library IEEE;
use IEEE.std_logic_1164.all;

-- 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;
```

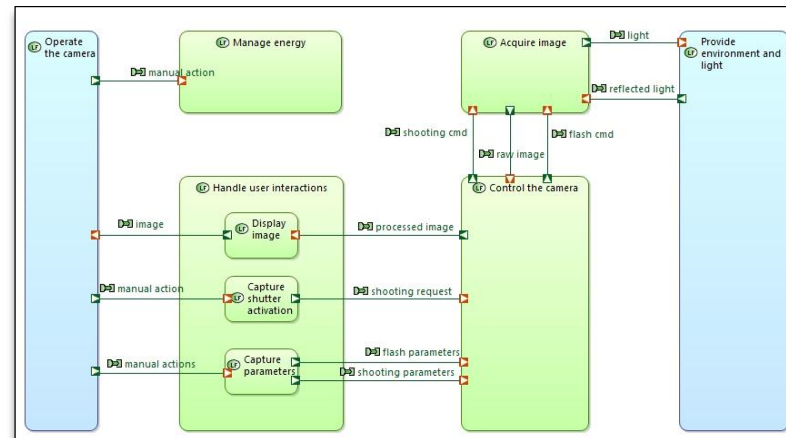
VHDL

Hardware Description Language



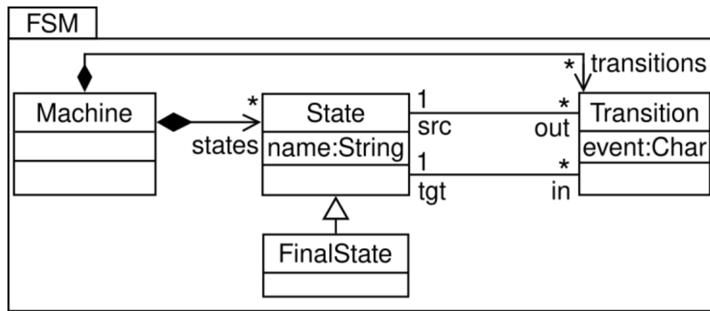
Scratch

Programming for Kids

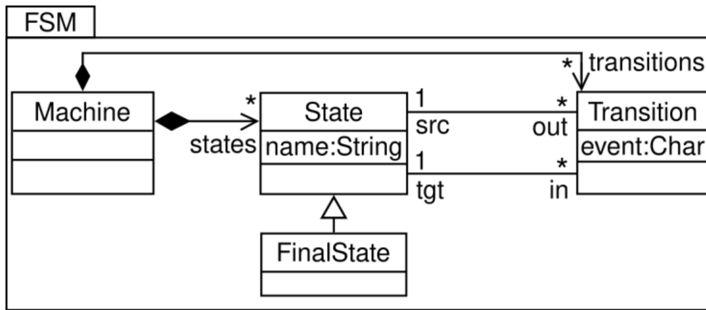


Capella

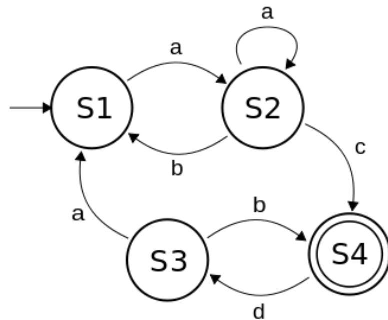
Systems Engineering Language



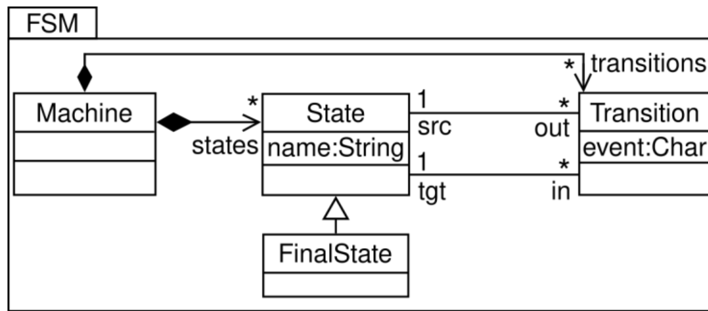
Abstract Syntax



Abstract Syntax



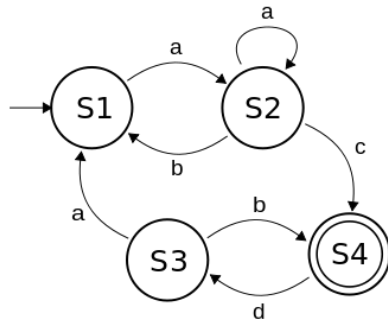
Concrete Syntax



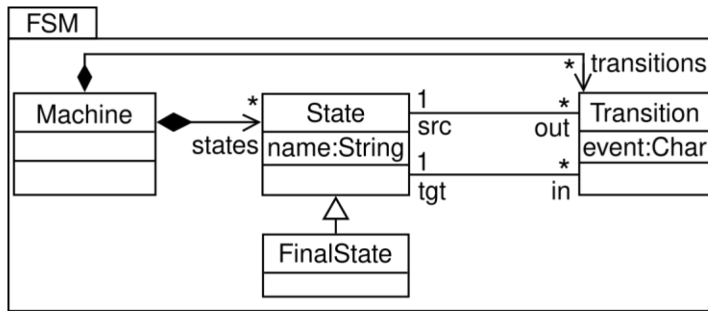
Abstract Syntax

```
step(State s, String evt) {
  val next = s.outgoing.filter[event == evt]
  if (next.size == 0)
    throw new DeadlockException
  if (next.size > 1)
    throw new IndeterminismException
  next.head.fire()
}
```

Execution Semantics



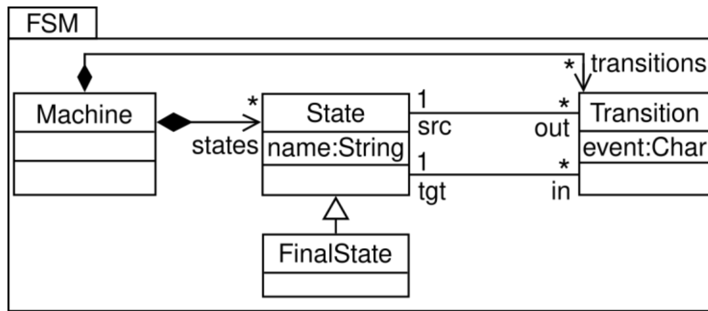
Concrete Syntax



Abstract Syntax

```
step(State s, String evt) {
  val next = s.outgoing.filter[event == evt]
  if (next.size == 0)
    throw new DeadlockException
  if (next.size > 1)
    throw new IndeterminismException
  next.head.fire()
}
```

Execution Semantics



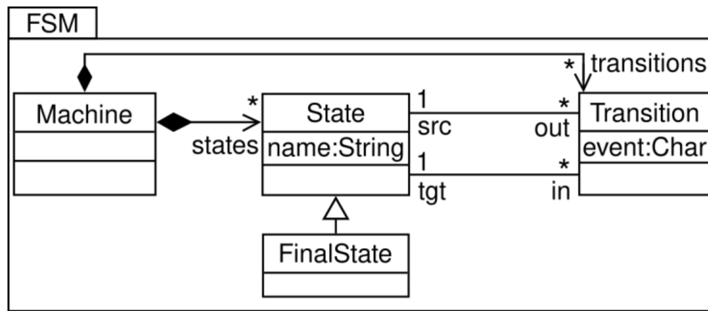
Abstract Syntax

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

AST Classes

```
step(State s, String evt) {
  val next = s.outgoing.filter[event == evt]
  if (next.size == 0)
    throw new DeadlockException
  if (next.size > 1)
    throw new IndeterminismException
  next.head.fire()
}
```

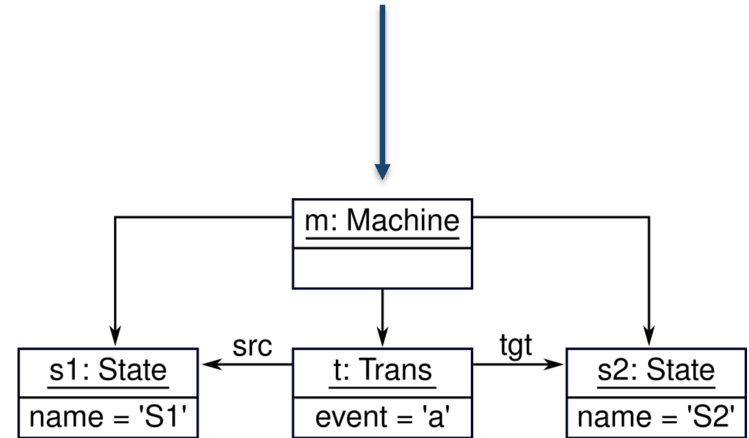
Execution Semantics



Abstract Syntax

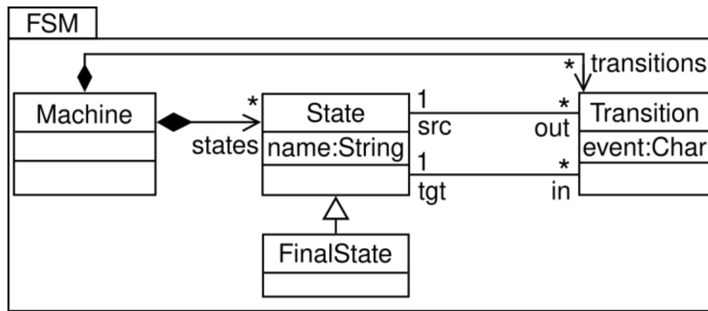
```
class Machine { List<State> states; […] }
class State   { String name; […] }
class Trans   { char event; […] }
class FS extends State { […] }
```

AST Classes




```
step(State s, String evt) {
  val next = s.outgoing.filter[event == evt]
  if (next.size == 0)
    throw new DeadlockException
  if (next.size > 1)
    throw new IndeterminismException
  next.head.fire()
}
```

Execution Semantics



Abstract Syntax

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

AST Classes

```
step(State s, String evt) {
  val next = s.outgoing.filter[event == evt]
  if (next.size == 0)
    throw new DeadlockException
  if (next.size > 1)
    throw new IndeterminismException
  next.head.fire()
}
```

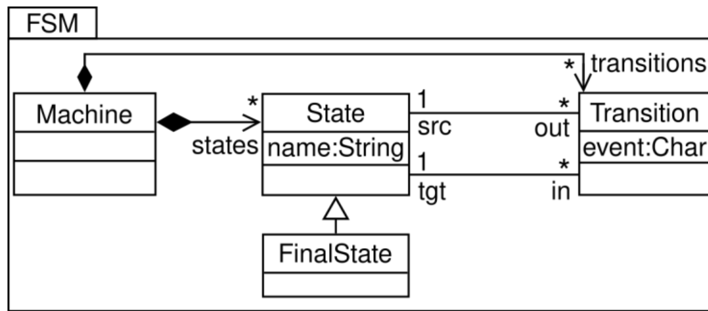
Execution Semantics

```
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



Abstract Syntax

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

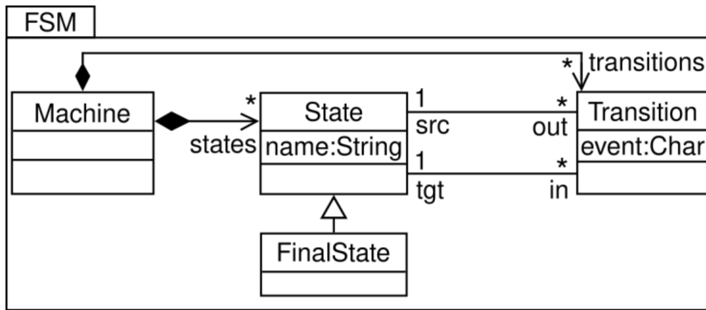
AST Classes

```
step(State s, String evt) {
  val next = s.outgoing.filter[event == evt]
  if (next.size == 0)
    throw new DeadlockException
  if (next.size > 1)
    throw new IndeterminismException
  next.head.fire()
}
```

Execution Semantics

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

Printing Semantics



Abstract Syntax

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

AST Classes

```
step(State s, String evt) {
  val next = s.outgoing.filter[event == evt]
  if (next.size == 0)
    throw new DeadlockException
  if (next.size > 1)
    throw new IndeterminismException
  next.head.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)
}
```

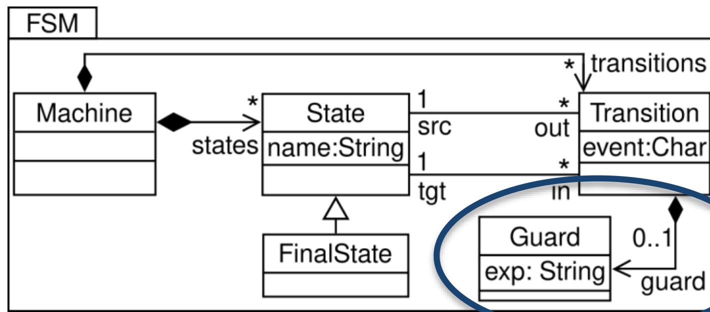
Printing Semantics

```
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



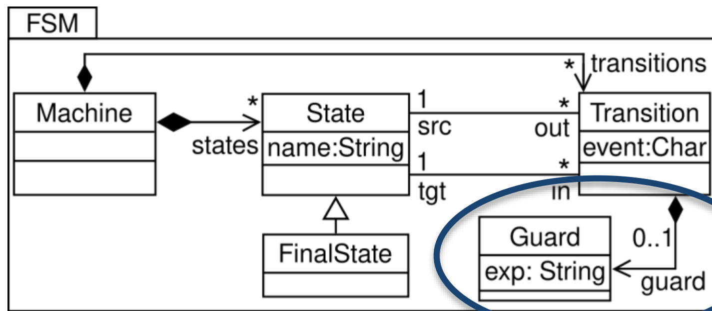
Abstract Syntax

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

AST Classes

```
step(State s, String evt) {
  val next = s.outgoing.filter[event == evt]
  if (next.size == 0)
    throw new DeadlockException
  if (next.size > 1)
    throw new IndeterminismException
  next.head.fire()
}
```

Execution Semantics



Abstract Syntax

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

AST Classes

```
step(State s, String evt) {
  val next = s.outgoing.filter[event == evt]
  if (next.size == 0)
    throw new DeadlockException
  if (next.size > 1)
    throw new IndeterminismException
  next.head.fire()
}
```

Execution Semantics

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

Interpreter Pattern

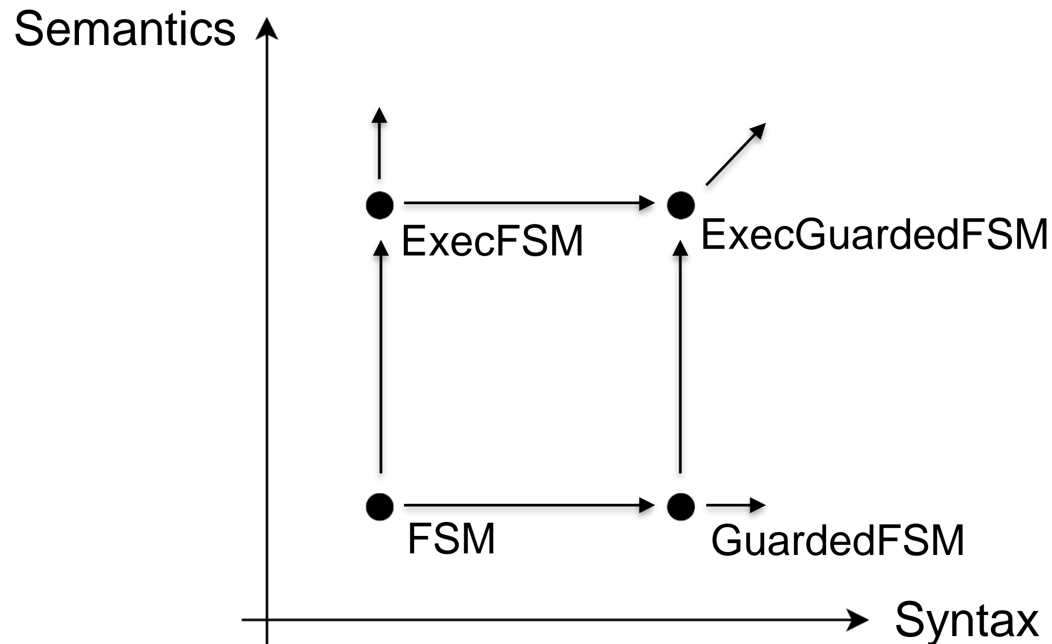
```
interface Visitor {
  void visit(Machine m);
  void visit(State s);
  void visit(Transition t);
  void visit(Guard g);
}
class ExecMachine implements Visitor {
  void visit(Machine m)    { […] }
  void visit(State s)      { […] }
  void visit(Transition t) { […] }
  void visit(Guard g)      { […]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

Visitor Pattern

cf. **The Expression Problem**
Philip Wadler
1998

Challenges of *Modular* Extension

- How to extend (syntax and semantics of) DSLs
  1. Without anticipating the extension
  2. Without modifying or duplicating existing code
  3. While ensuring type safety

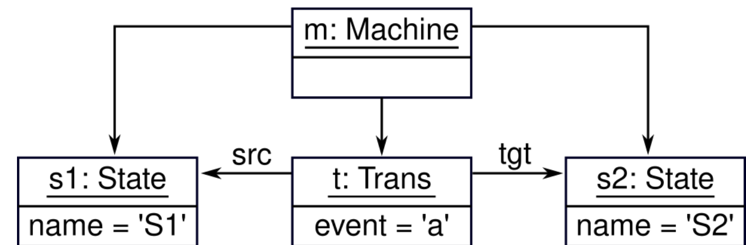


The REVISITOR Pattern

Revisitor *Interface*

- Maps syntactic objects to 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);
}
```

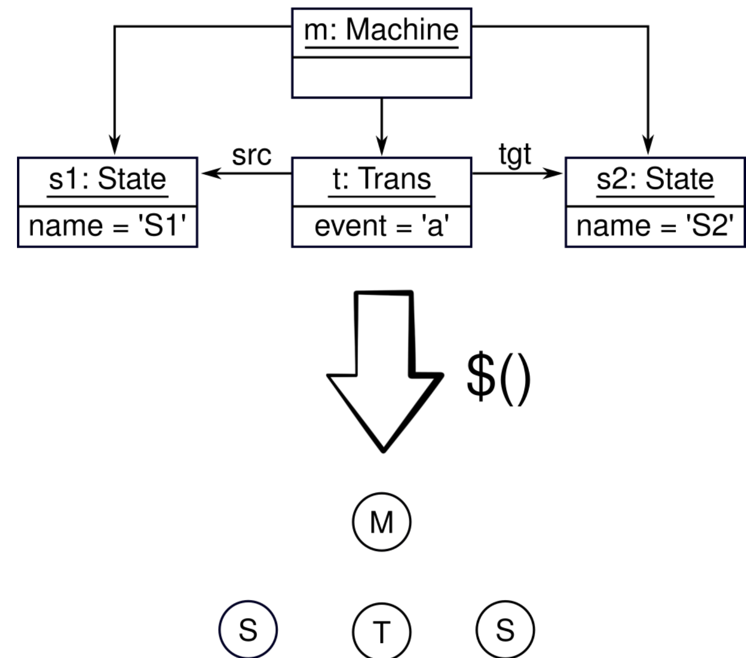


Revisitor *Interface*

- Maps syntactic objects to 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);
  }
}
```

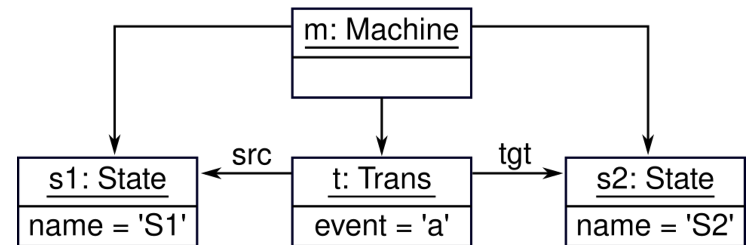


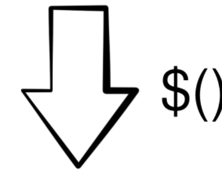
Revisitor *Implementation*

- Implements the semantic objects

```
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  */;
  }
}
```





$()

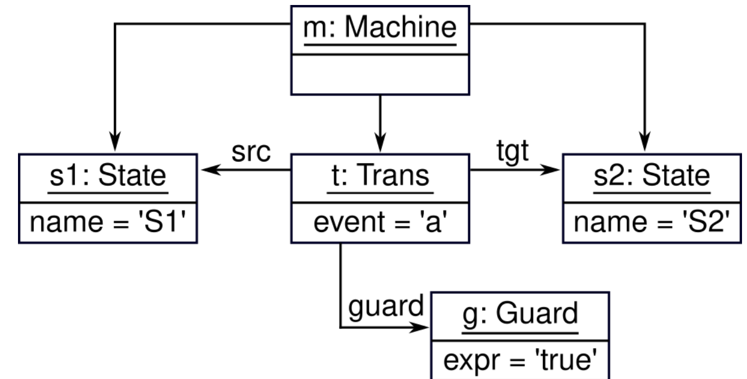
(Machine) => String

(State) => String

(State) => String

(Trans) => String

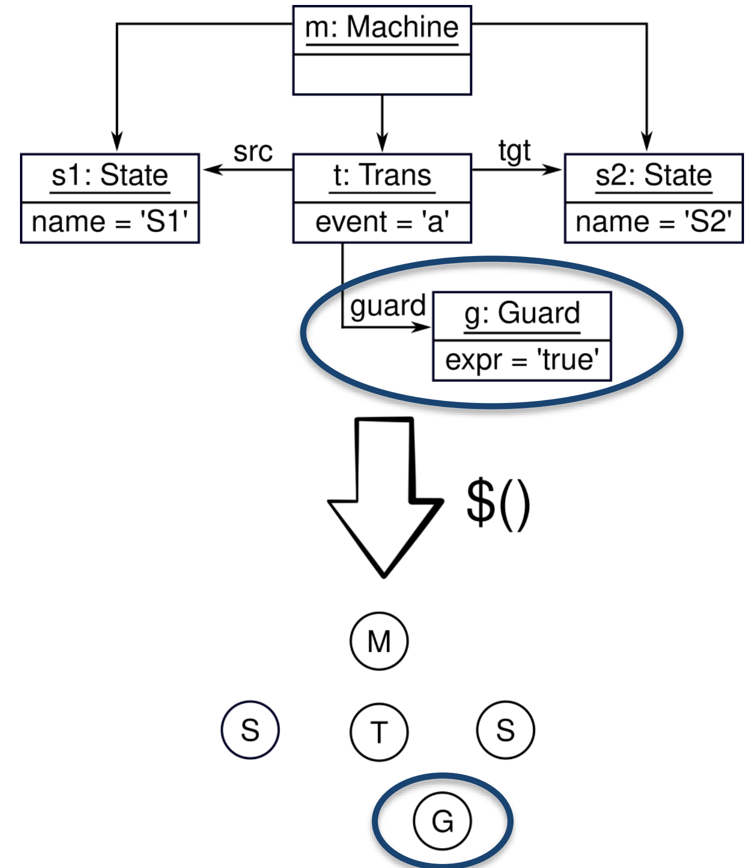
Modular Extension with Revisitors



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); }
}
```

Extend the (existing) mapping



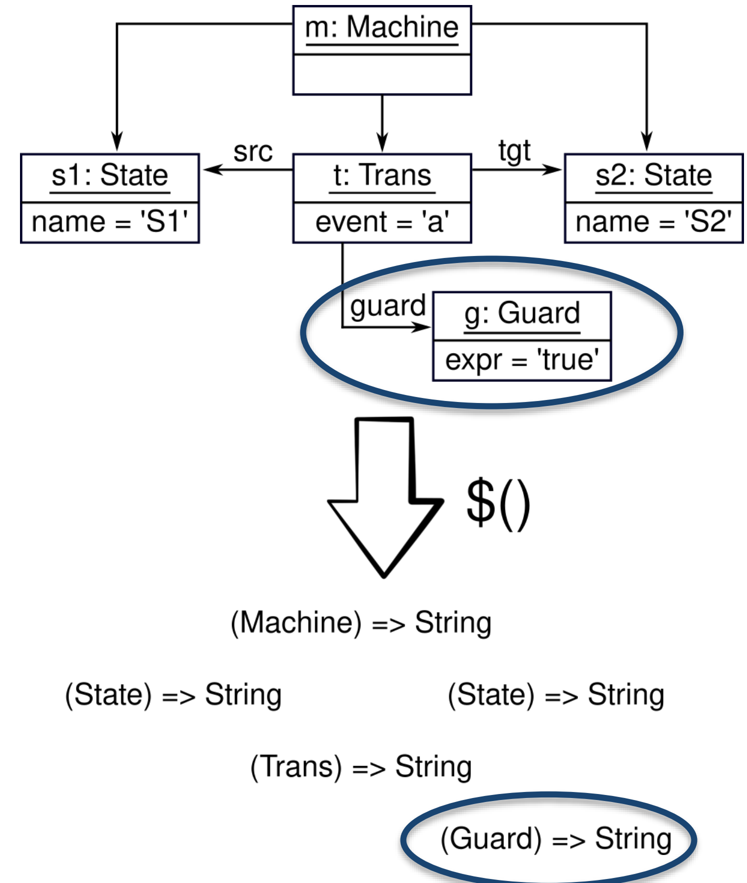
Modular Extension with Revisitors

```
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); }
}
```

Extend the (existing) mapping

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

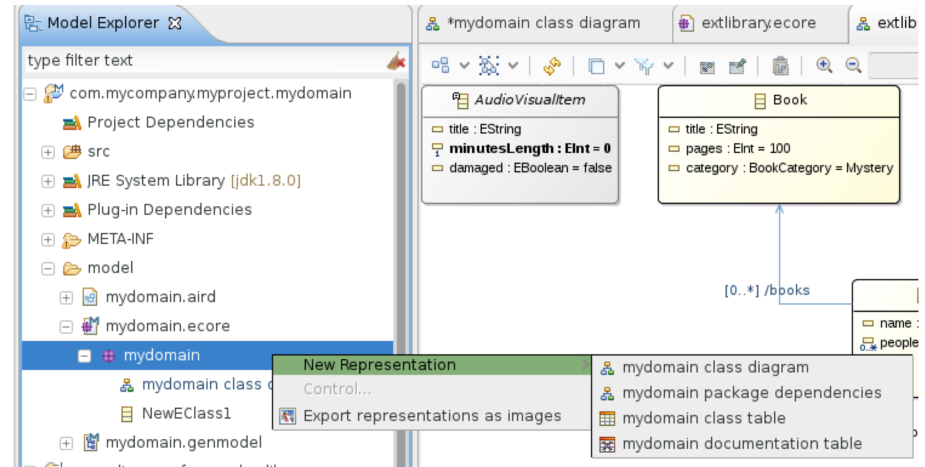
Extend the (existing) semantics



ALE: the *Action Language for EMF*

```
open class Machine {
  def String print() {
    String ret = "";
    for (State s in self.states)
      ret = ret + $[s].print();
    return ret;
  }
}
open class State {
  def String print() { /* ... */ }
}
open class FinalState {
  def String print() {
    return "*" + $[super].print();
  }
}
open class Transition {
  def String print() {
    return self.event + "=>" + self.tgt.name;
  }
}
```

Printing FSMs in ALE



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

Wrap-up

- The story so far
  - Independent and modular extensibility of syntax & semantics
  - With incremental compilation, without anticipation
  - Applicable in any "mainstream" OO language
  - The ALE language, soon™ in Eclipse, makes it user-friendly

- What's next?
  - Separate compilation is the first step towards language components
  - Off-the-shelf language components have explicit provided & required interfaces
  - Pick, assemble, and customize language components to create new DSLs

EOF