Object-Oriented Design Pattern for DSL Program Monitoring

Zoé.DREY

Ciprian.TEODOROV

@ ENSTA-bretagne.fr

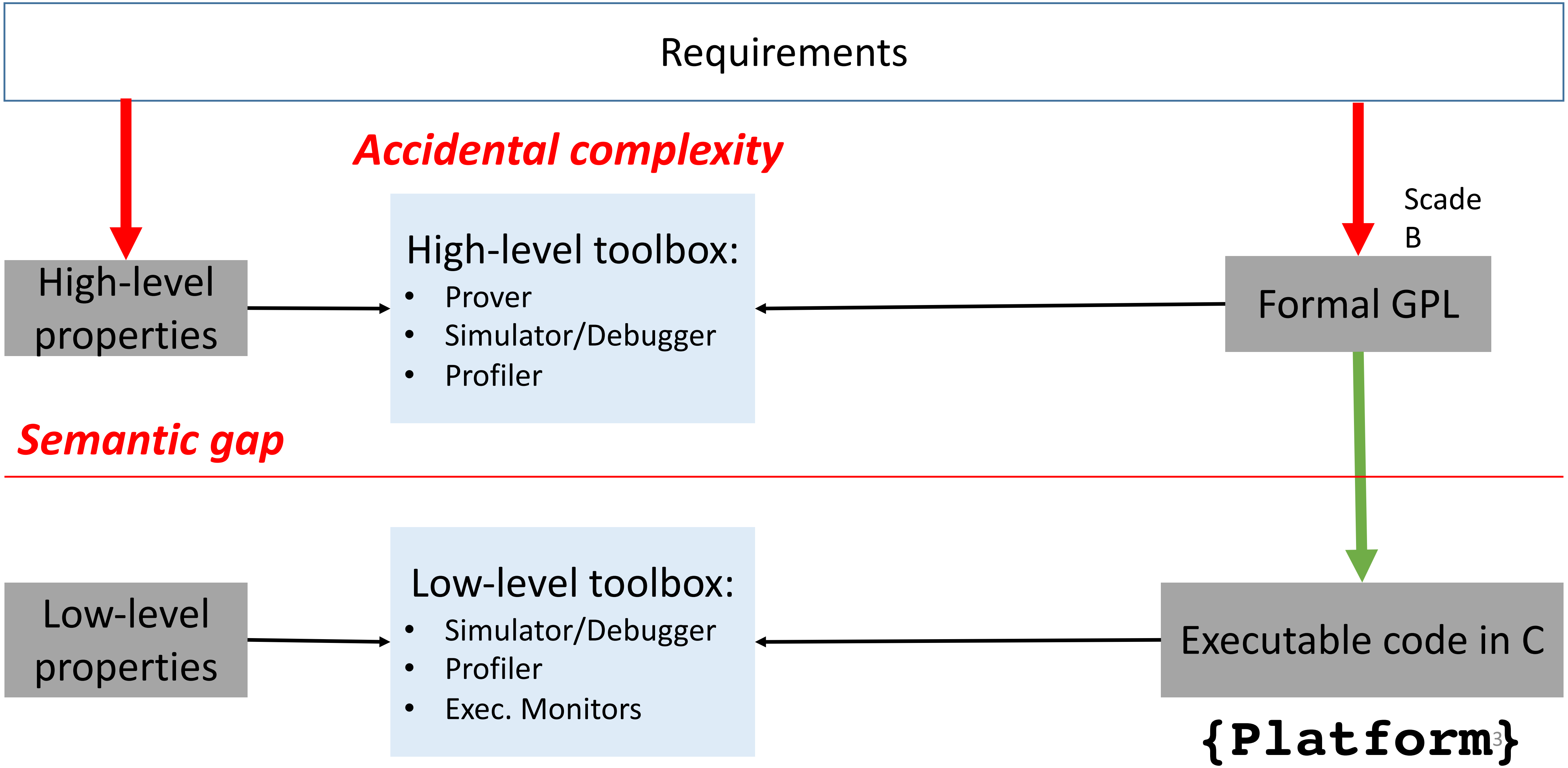
Lab-STICC, MOCS Team, Brest France

Software Language Engineering, Amsterdam, 31 october 2016

Overview

- Context: Program diagnosis 4 Critical Systems
- Problem: Gap between Language Workbenches & Diagnosis tools
- Contribution: Object-oriented DSL Monitoring Pattern
- Conclusion & Perspectives

DSL-based Diagnosis 4 Critical Systems



Requirements

Accidental complexity

High-level properties

High-level toolbox:

- Prover
- Simulator/Debugger
- Profiler

Scade
B

Formal GPL

Semantic gap

Low-level properties

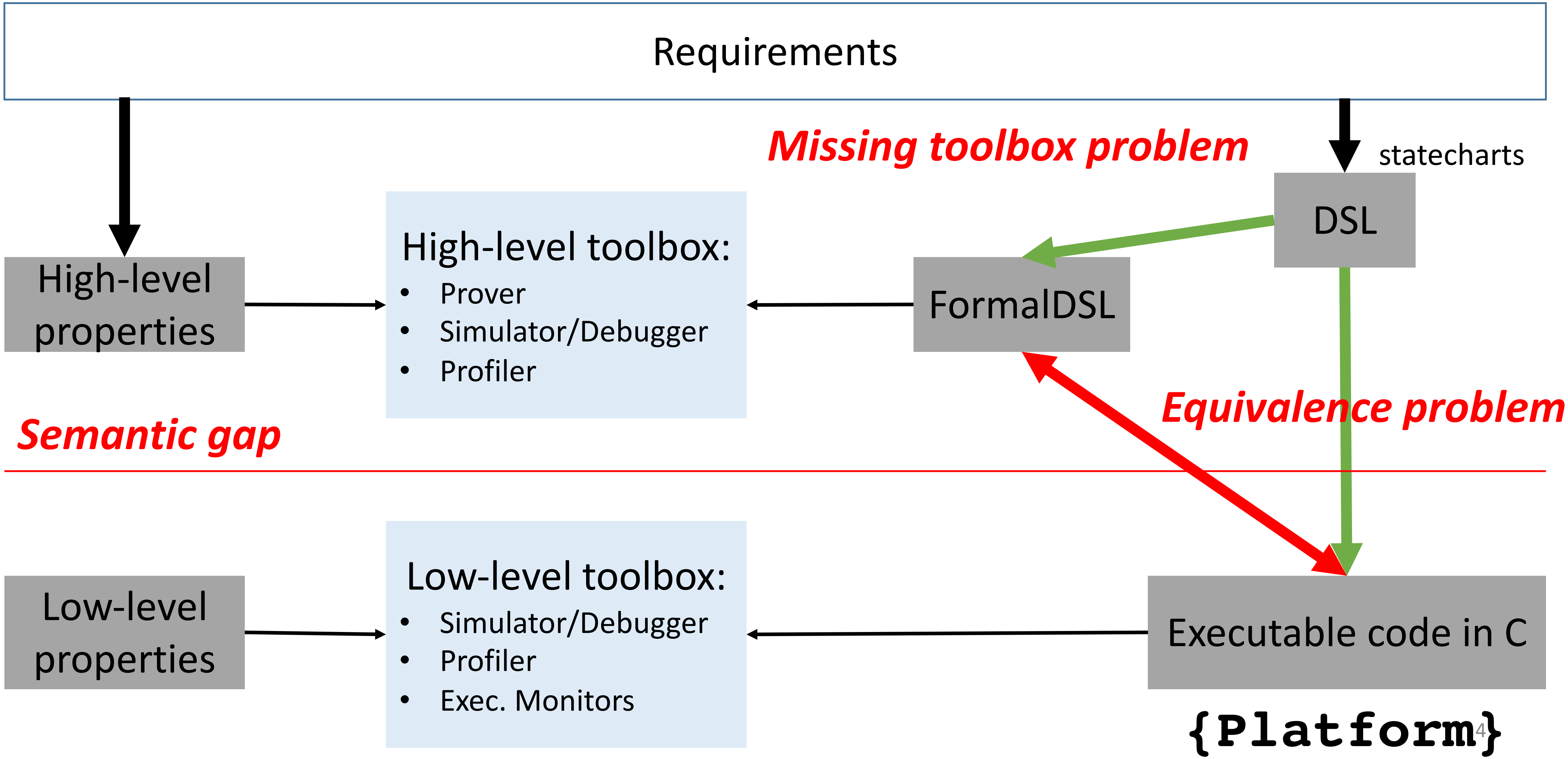
Low-level toolbox:

- Simulator/Debugger
- Profiler
- Exec. Monitors

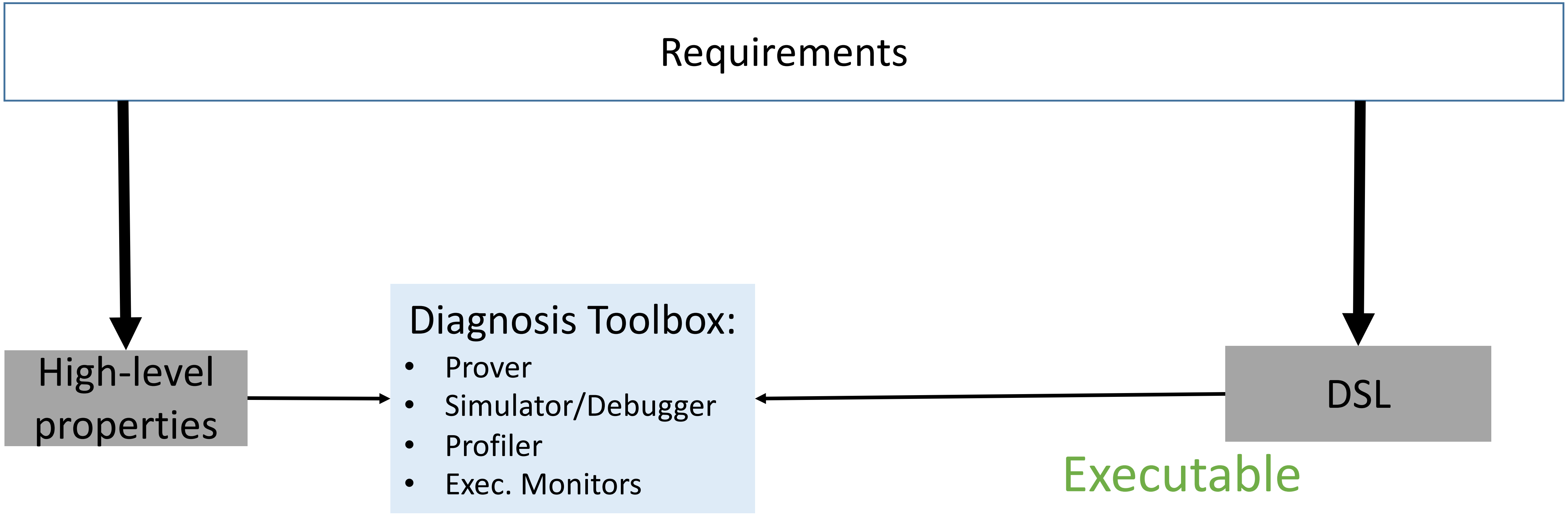
Executable code in C

`{Platform}`

DSL-based Critical System Infrastructure



DSL-based Critical System Infrastructure



Requirements

High-level properties

Diagnosis Toolbox:

- Prover
- Simulator/Debugger
- Profiler
- Exec. Monitors

DSL

Executable

Missing toolbox problem

The Problem: How to **make the *connection*** ?

Domain-specific diagnosis ⟷ ***Language workbenches***

Moldable debugger *Chis et al. CLSS'15*

DSProfile *Sloane et al. SCP'16*

MetaSpy *Ressia et al. JOT'02*

LTSMin *Kant et al. TACAS'15*

Gemoc studio *Bousse et al. SLE'15*

Spoofax *Kats et al. OOPSLA'10*

MPS *jetbrains.com/mps*

K Framework *Rosu et al. JLAP'10*

The Problem: Requirements

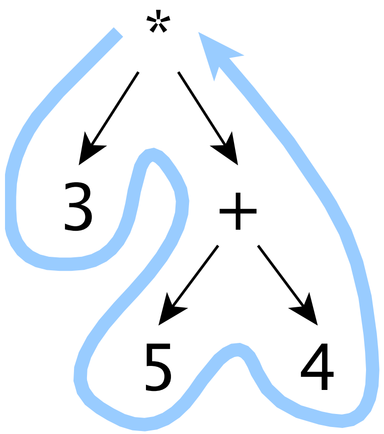
Domain-specific diagnosis ↔ ***Language workbenches***

DSL monitoring is the process of **observing the execution of a program** expressed in a DSL.

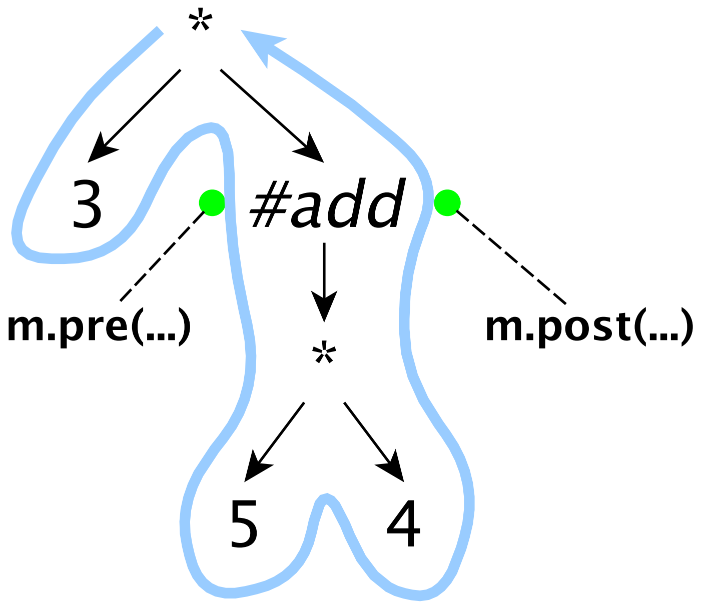
- *[R01]* Completeness
- *[R02]* Non-Interference
- *[R03]* Genericity
- *[R04]* Composability
- *[R05]* Unanticipated Monitoring
- *[R06]* Portability
- *[R07]* DSL Runtime Integration
- *[R08]* Tool Integration
- *[R09]* Minimize the Gap
- *[R10]* Break the Rules

Background: Kishon’s Monitoring Semantics

Kishon et al. PLDI’91



valuation wrapped
with pre and post



Interpreter + Monitor = Monitoring Interpreter

Continuation
Passing-style

pre: Ann → SynTerm → SemDomain → MS → MS

post: Ann → SynTerm → SemDom → IVal → MS → MS

Kishon’s Monitoring Semantics
vs Requirements

[R01] Completeness

[R02] Non-Interference

[R03] Genericity

[R04] Composability

[R05] Unanticipated Monitoring

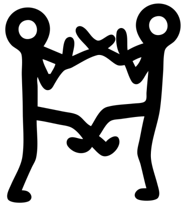
[R06] Portability

[R07] DSL Runtime Integration

[R08] Tool Integration

[R09] Minimize the Gap

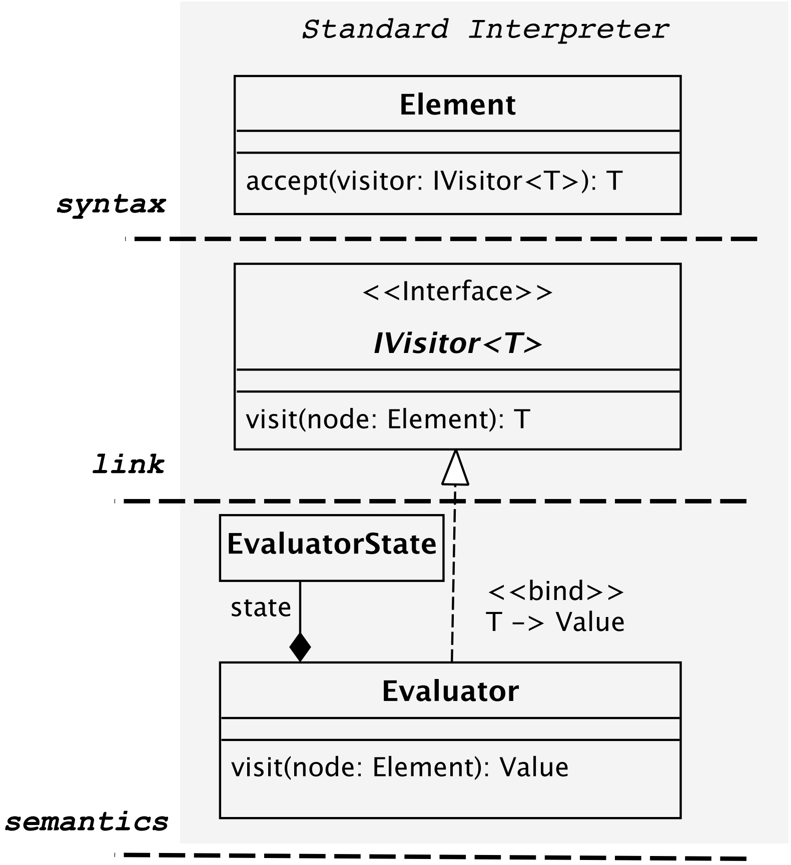
[R10] Break the Rules



Object-Oriented Design Pattern for DSL Program Monitoring

Our contribution

DSL = Syntax + Semantics



Compatibility with **Visitor** and Interpreter pattern

[R07] No need to change existing implementations.

Visitor pattern:

- Isolates the *semantics* from the *syntax*
- Prevents the mix between AST data & evaluator state

EvaluatorState factored out of the Evaluator

- Closer to the notion of semantic domains and valuation functions;
- Offers an object interface dedicated for state access & update

[R01]
[R02]
[R03]
[R04]
[R05]
[R06]
[R07]
[R08]
[R09]
[R10]

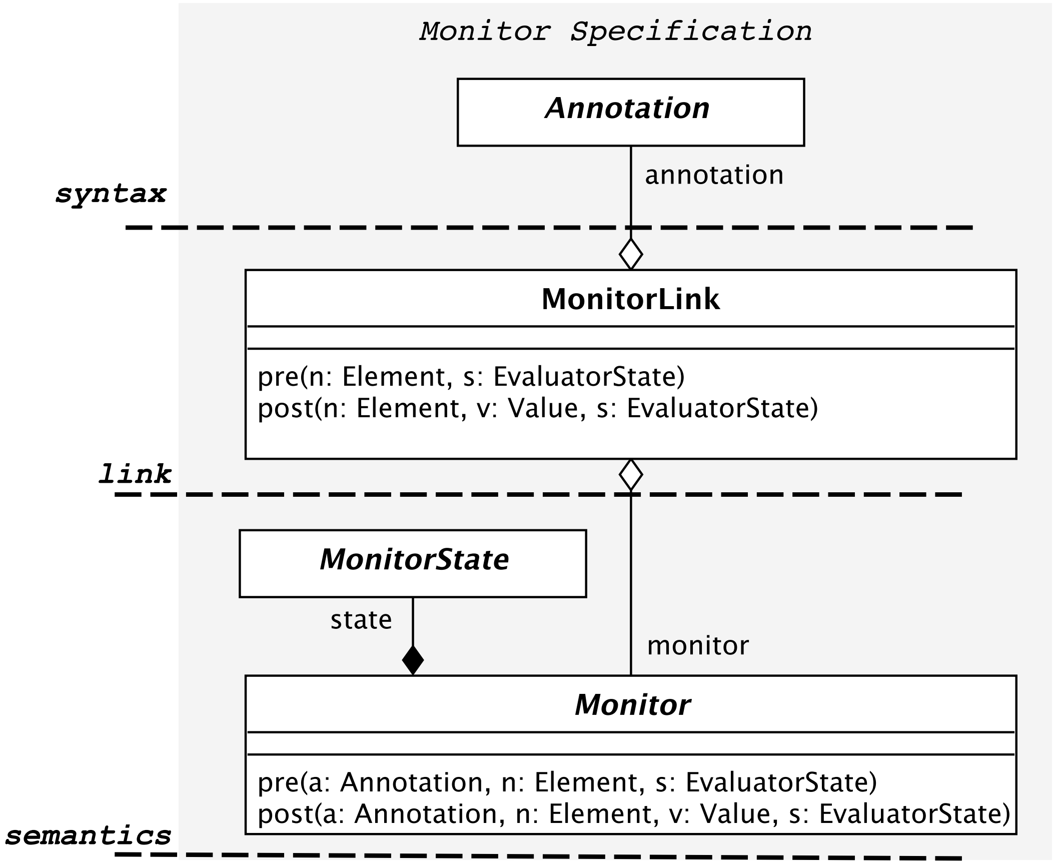
Monitor = Syntax + Semantics

The monitor as proper language construct.

[R03] Genericity
[R08] Independent monitor development

The monitor syntax = the annotation
The monitor semantics = pre & post

The monitor semantics
is dependent of the monitored DSL
through the *EvaluatorState* & *Value*


Monitor Specification
Annotation
annotation
syntax
MonitorLink
pre(n: Element, s: EvaluatorState)
post(n: Element, v: Value, s: EvaluatorState)
link
MonitorState
state
monitor
Monitor
pre(a: Annotation, n: Element, s: EvaluatorState)
post(a: Annotation, n: Element, v: Value, s: EvaluatorState)
semantics

[R01]
[R02]
[R03]
[R04]
[R05]
[R06]
[R07]
[R08]
[R09]
[R10]

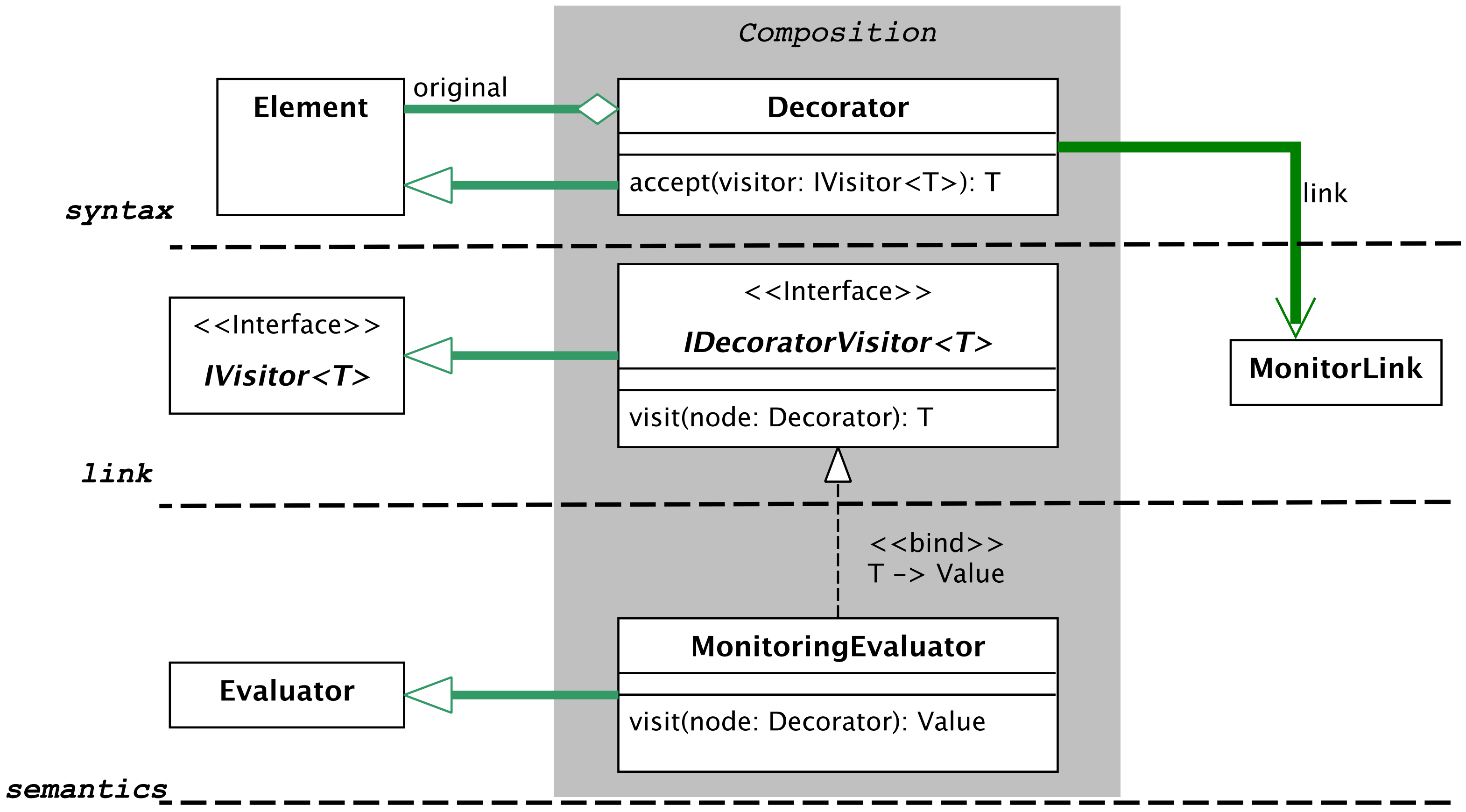
Composition Operator

[R01] decorate all terms

Only inheritance:

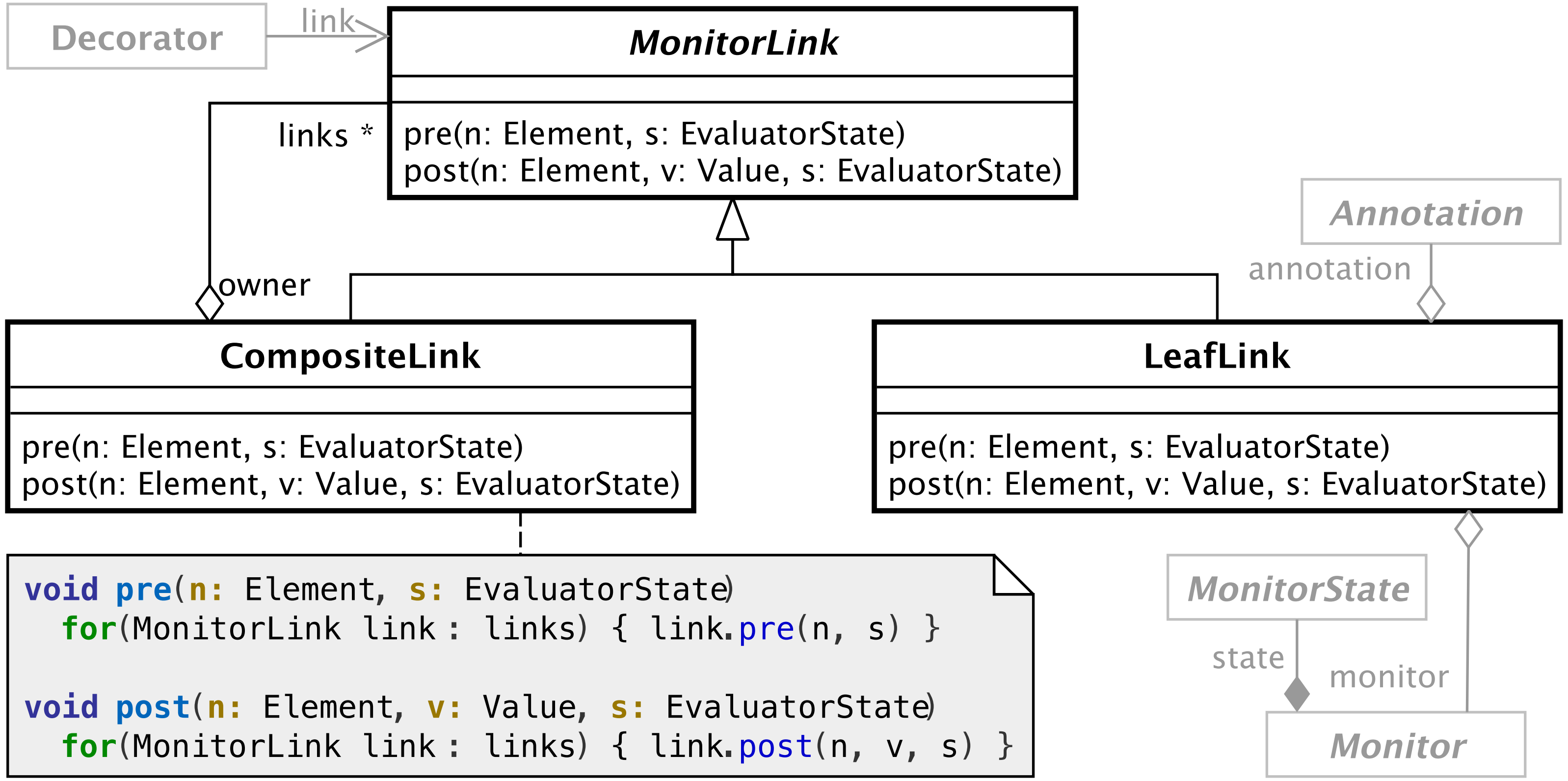
- *[R06]* no reflection needed
- *[R07]* no modifications to legacy

[R08] a simple link to the monitors



[R01]
[R02]
[R03]
[R04]
[R05]
[R06]
[R07]
[R08]
[R09]
[R10]

[R04] Composable Monitors


Decorator
link
MonitorLink
pre(n: Element, s: EvaluatorState)
post(n: Element, v: Value, s: EvaluatorState)
links *
owner
CompositeLink
pre(n: Element, s: EvaluatorState)
post(n: Element, v: Value, s: EvaluatorState)
LeafLink
pre(n: Element, s: EvaluatorState)
post(n: Element, v: Value, s: EvaluatorState)
Annotation
annotation
MonitorState
state
monitor
Monitor

```
void pre(n: Element, s: EvaluatorState)
  for(MonitorLink link : links) { link.pre(n, s) }

void post(n: Element, v: Value, s: EvaluatorState)
  for(MonitorLink link : links) { link.post(n, v, s) }
```

[R01]
[R02]
[R03]
[R04]
[R05]
[R06]
[R07]
[R08]
[R09]
[R10]

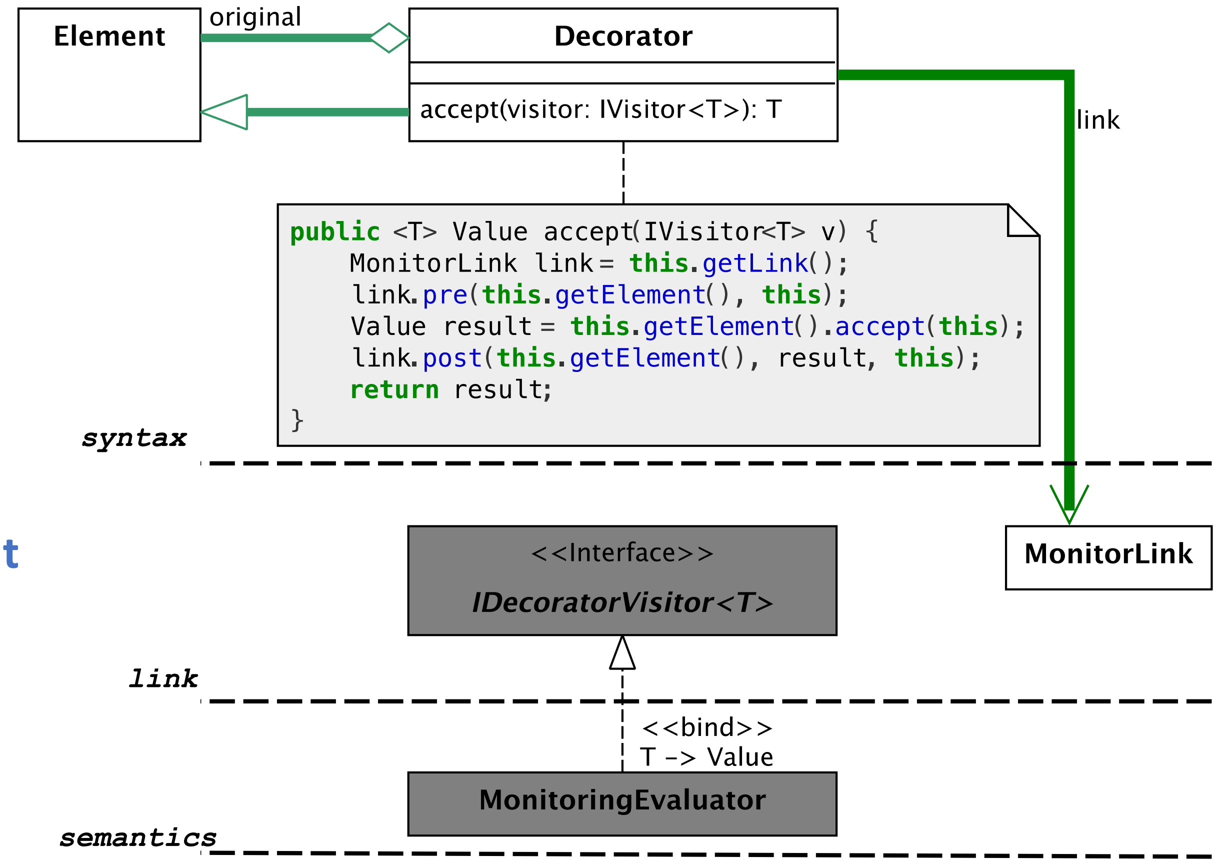
[R05] Unanticipated Monitoring

Handle the pre/post dispatch in the **accept** method

IDecorator & MonitoringEvaluator **out**

Drawbacks: code less homogeneous

Interferes with other visitors



```
public <T> Value accept(IVisitor<T> v) {
    MonitorLink link = this.getLink();
    link.pre(this.getElement(), this);
    Value result = this.getElement().accept(this);
    link.post(this.getElement(), result, this);
    return result;
}
```

[R01]
[R02]
[R03]
[R04]
[R05]
[R06]
[R07]
[R08]
[R09]
[R10]

[R02] Non-Interference
vs [R10] Breaking the Rules

Trade-off needed !

- *IDEA*: Expose a **façade** on the EvaluatorState to the monitor
- Different access policies could be enforced
  - **Non-interference**: read-only access to the EvaluatorState
  - **Breaking-the-rules**:
    - Monitor updates the EvaluatorState through its API – preserves semantics
    - Monitor accesses the Internal structure of the Evaluator – more than ES
    - Monitor changes the AST – potentially the EvaluatorState changes shape

[R01]
[R02]
[R03]
[R04]
[R05]
[R06]
[R07]
[R08]
[R09]
[R10]

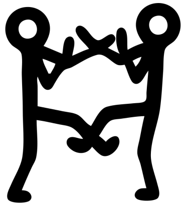
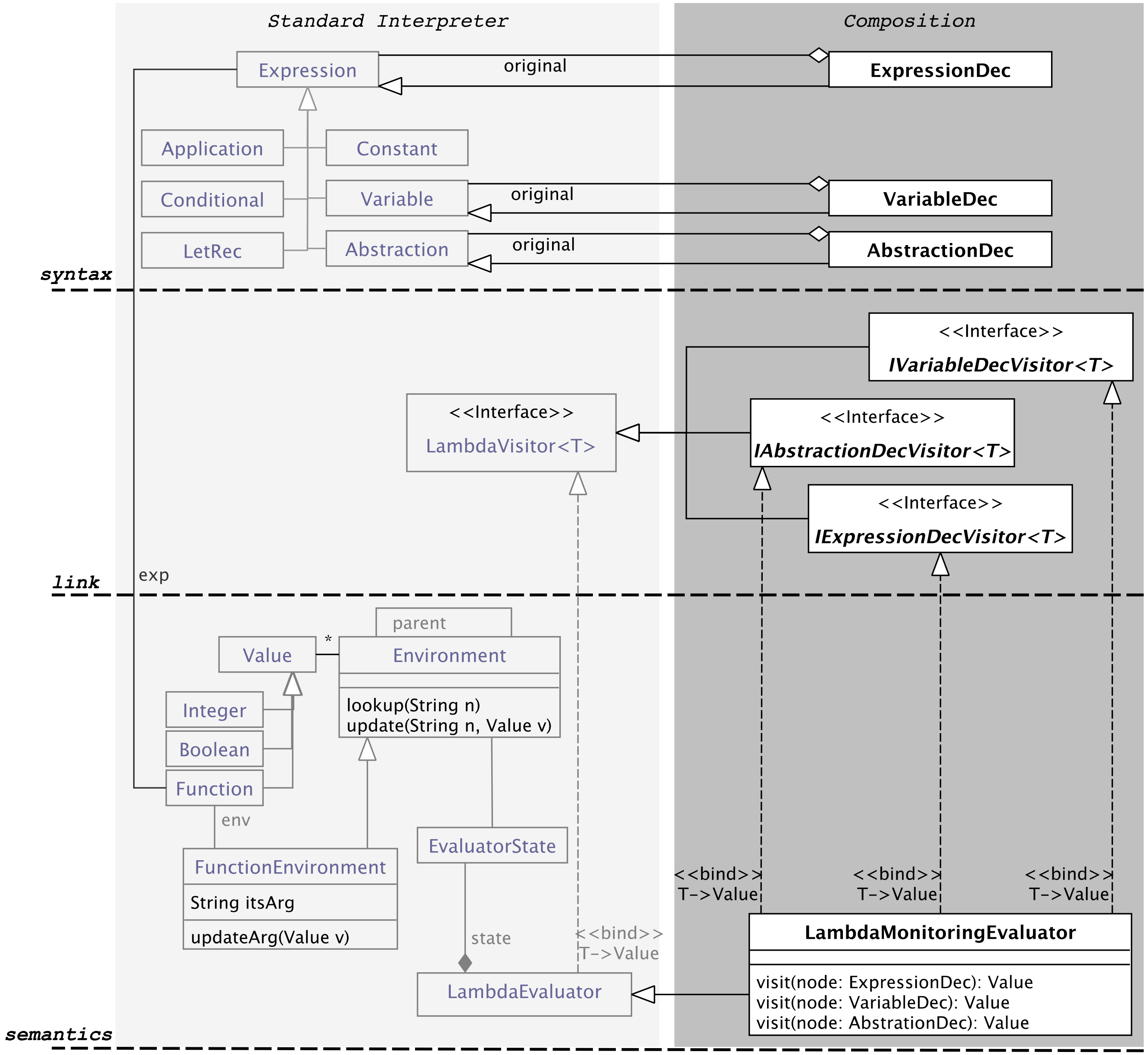
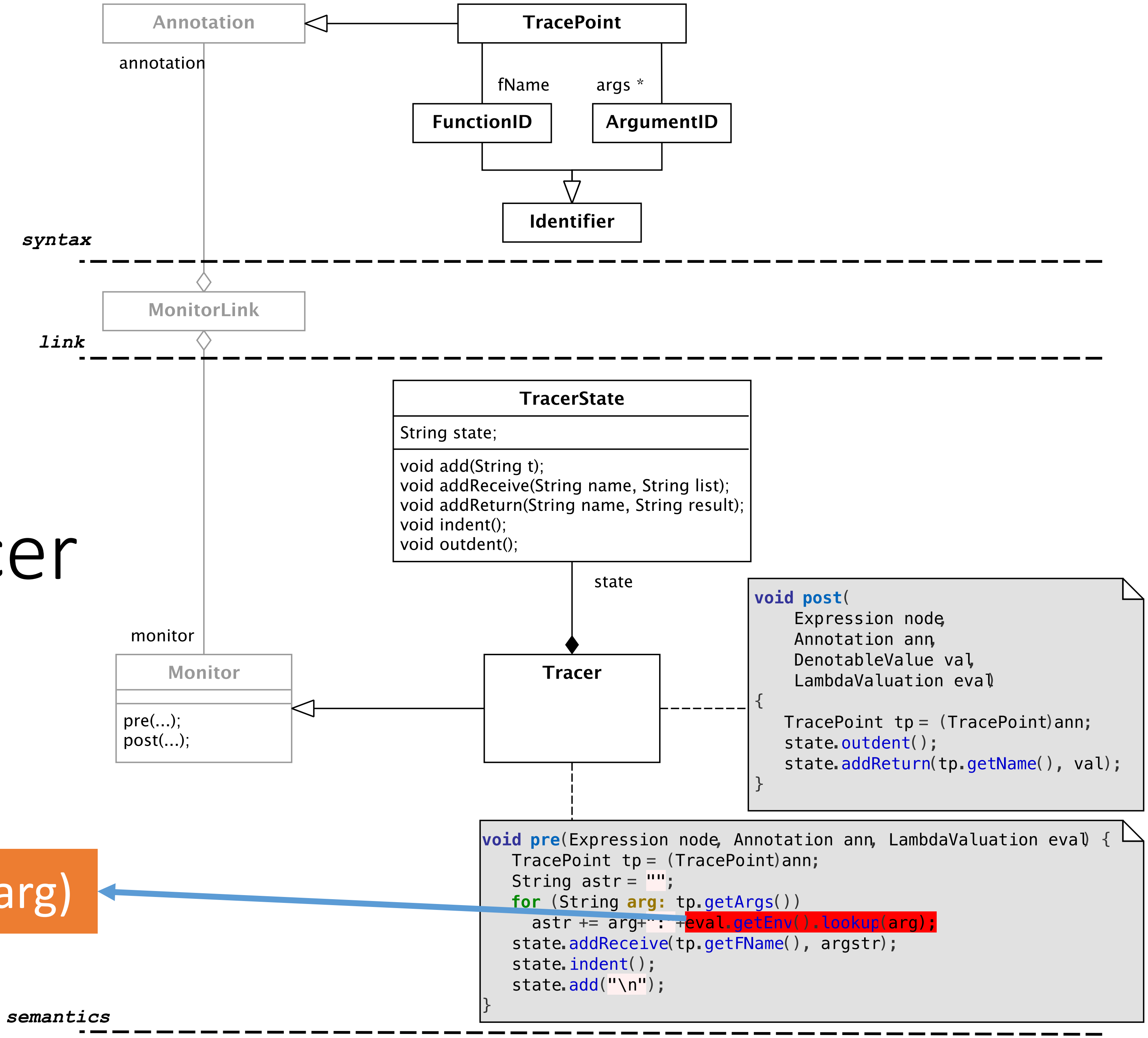


Illustration : Lambda Calculus



Automatic generation
of the **Composition Layer**

Monitor 1:
A Simple Tracer



```
void post(
     Expression node,
     Annotation ann,
     DenotableValue val,
     LambdaValuation eval)
{
   TracePoint tp = (TracePoint)ann;
   state.outdent();
   state.addReturn(tp.getName(), val);
}
```

```
void pre(Expression node, Annotation ann, LambdaValuation eval) {
   TracePoint tp = (TracePoint)ann;
   String astr = "";
   for (String arg: tp.getArgs())
     astr += arg+": "+eval.getEnv().lookup(arg);
   state.addReceive(tp.getFName(), argstr);
   state.indent();
   state.add("\n");
}
```

eval.getEnv().lookup(arg)

Monitor 1: A Simple Tracer

Usage scenario

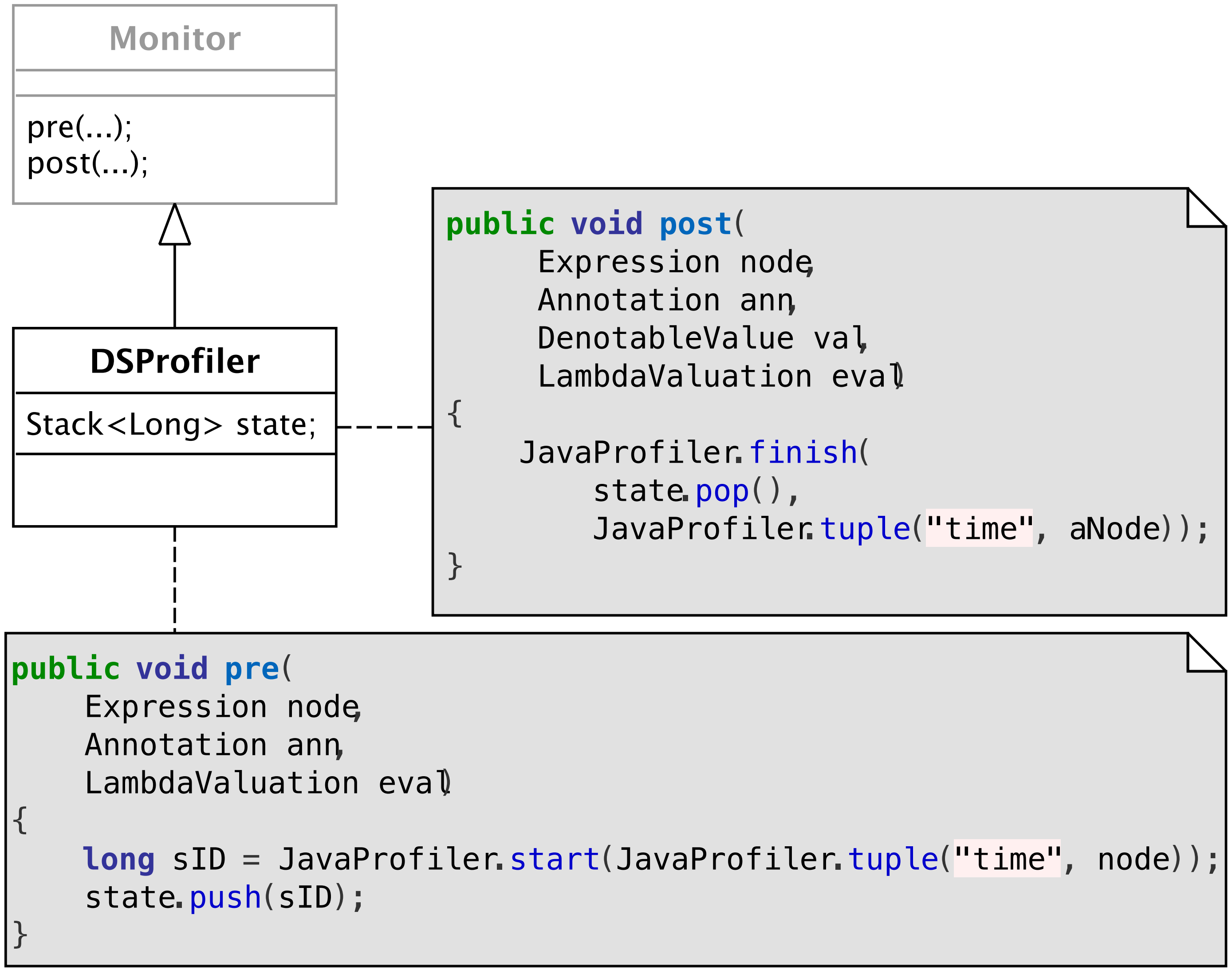
```
tracer = new Tracer();
$link_1$ = new MonitorLink ("mult(x y)", tracer);
$link_2$ = new MonitorLink( "fac(x)", tracer);

ast = new LambdaParser(
   "letrec  mult=\x.\y. $[link_1]_{exp}$ (* x y) in
    letrec fact=\x. $[link_2]_{exp}$ if (= x 1) then 1
      else (mult x) (fact (- x 1)) in fact
          4");
ast.accept(new LambdaMonitoringEvaluator());
tracer.printTrace();
```

Resulting Trace

```
[#fac receives (x:4 )]
|   [#fac receives (x:3 )]
|   |   [#fac receives (x:2 )]
|   |   |   [#fac receives (x:1 )]
|   |   |   [#fac returns 1]
|   |   |   [#mult receives (x:2 y:1 )]
|   |   |   [#mult returns 2]
|   |   [#fac returns 2]
|   |   [#mult receives (x:3 y:2 )]
|   |   [#mult returns 6]
|   [#fac returns 6]
|   [#mult receives (x:4 y:6 )]
|   [#mult returns 24]
[#fac returns 24]
```

Monitor 3 : An external DSL Profiler



```
public void post(
        Expression node,
        Annotation ann,
        DenotableValue val,
        LambdaValuation eval)
{
      JavaProfiler.finish(
            state.pop(),
            JavaProfiler.tuple("time", aNode));
}
```

```
public void pre(
      Expression node,
      Annotation ann,
      LambdaValuation eval)
{
      long sID = JavaProfiler.start(JavaProfiler.tuple("time", node));
      state.push(sID);
}
```

Sloane et al. SCP'16

DSProfile:

- implemented in Scala,
- used as black-box

Profiling results:

```
    14 ms total time; 14 ms profiled time (95.9%)
    1003 profile records
Total Total  Self  Self  Desc  Desc Count Count
   ms     %    ms     %    ms     %           %
   14  99.6     0   0.6    13  98.9     1   0.1  [1]
   13  98.9     3  26.8    10  72.1   201  20.0  [2]
   13  98.4     2  17.5    11  80.9   200  19.9  [3]
   13  98.3     4  33.0     9  65.3   200  19.9  [4]
    1  12.1     1  12.1     0   0.0   200  19.9  (- x 1)
    1  10.0     1  10.0     0   0.0   201  20.0  (= x 0)

[1] letrec fact=\x.if (= x 0) then 1
                   else (* x (fact (+ x -1))) in (fact 200)
[2] if (= x 0) then 1 else (* x (fact (+ x -1)))
[3] (* x (fact (+ x -1)))
[4] (fact (+ x -1))
```

Object-Oriented Monitoring Pattern

[R01] Completeness

[R02] Non-Interference

[R03] Genericity

[R04] Composability

[R05] Unanticipated Monitoring

[R06] Portability

[R07] DSL Runtime Integration

[R08] Tool Integration

[R09] Minimize the Gap

[R10] Break the Rules

Conclusion & Perspectives

Today

- The DSL Monitoring Pattern*: an object-oriented solution
- Improves over Kishon’s monitoring semantics
- Illustration through:
  - Simple lambda calculus
  - Creating a tracer from scratch
  - Integration of a COTS tool

*Pattern = Knowledge transfer
Implementations available today: Java & Smalltalk

Tomorrow

Easy:

- From **pattern** to **framework.**
- Tool support for AST decoration: MPS?

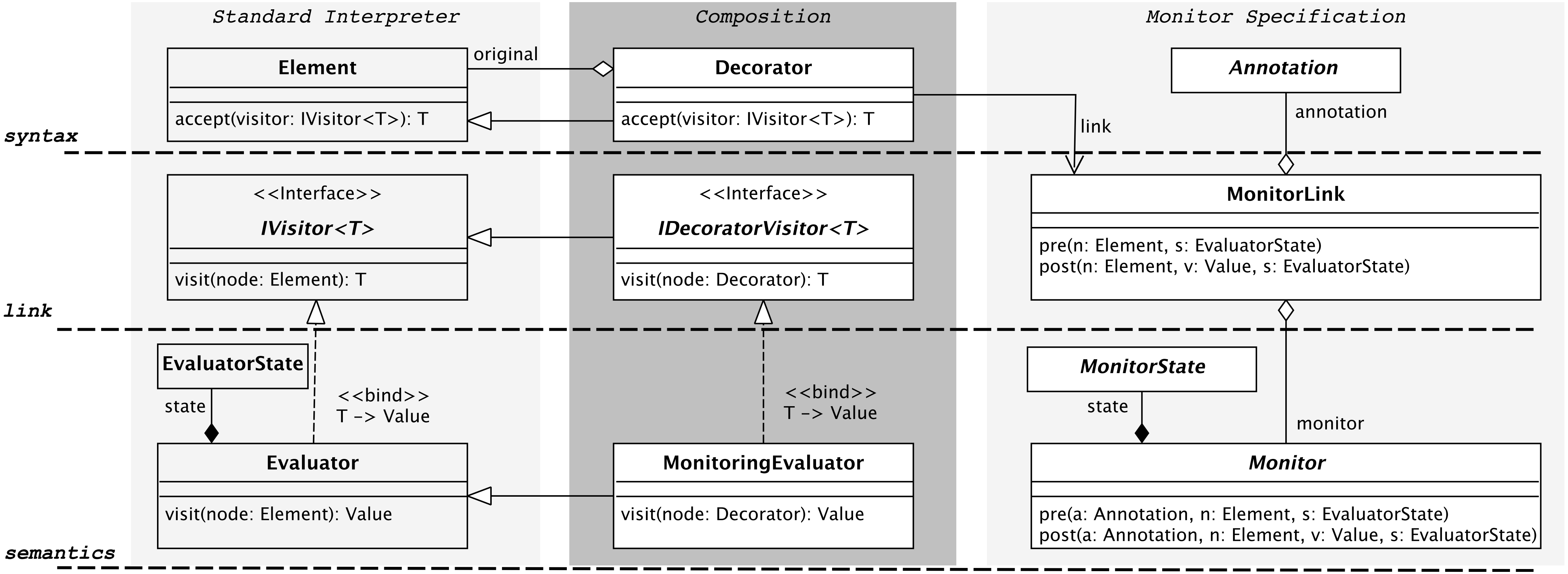
Not so easy

- Time & non-interference?
- Distributed monitoring

The End

Discussion & Questions

DSL Monitoring Pattern



Standard Interpreter
Composition
Monitor Specification
Element
accept(visitor: IVisitor<T>): T
original
Decorator
accept(visitor: IVisitor<T>): T
Annotation
annotation
link
syntax
<<Interface>>
IVisitor<T>
visit(node: Element): T
<<Interface>>
IDecoratorVisitor<T>
visit(node: Decorator): T
MonitorLink
pre(n: Element, s: EvaluatorState)
post(n: Element, v: Value, s: EvaluatorState)
link
EvaluatorState
state
<<bind>>
T -> Value
<<bind>>
T -> Value
MonitorState
state
monitor
Evaluator
visit(node: Element): Value
MonitoringEvaluator
visit(node: Decorator): Value
Monitor
pre(a: Annotation, n: Element, s: EvaluatorState)
post(a: Annotation, n: Element, v: Value, s: EvaluatorState)
semantics

[R01]
[R02]
[R03]
[R04]
[R05]
[R06]
[R07]
[R08]
[R09]
[R10]