Object-Oriented Design Pattern for DSL Program Monitoring

Zoé.DREY
Ciprian.TEODOROV

Lab-STICC, MOCS Team, Brest France

Software Language Engineering, Amsterdam, 31 October 2016
Overview

• **Context:** Program diagnosis for 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 toolbox:
- Prover
- Simulator/Debugger
- Profiler

Low-level toolbox:
- Simulator/Debugger
- Profiler
- Exec. Monitors

Formal GPL

**Semantic gap**

High-level properties

Low-level properties

{Platform}

Scade B

Executable code in C
DSL-based Critical System Infrastructure

Requirements

High-level properties

High-level toolbox:
- Prover
- Simulator/Debugger
- Profiler

Low-level properties

Low-level toolbox:
- Simulator/Debugger
- Profiler
- Exec. Monitors

FormalDSL

DSL

Executable code in C

(statecharts)

Missing toolbox problem

Semantic gap

Equivalence problem
DSL-based Critical System Infrastructure

Requirements

High-level properties

Diagnosis Toolbox:
- Prover
- Simulator/Debugger
- Profiler
- Exec. Monitors

DSL

Executable

Missing toolbox problem

{Platform}
The Problem: How to make the connection?

Domain-specific diagnosis → Language workbenches

Moldable debugger
- Chis et al. CLSS’15

DSPProfile
- 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. JLP’10
The Problem: Requirements

*Domain-specific diagnosis* $\leftrightarrow$ $\rightarrow$ *Language workbenches*

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

<table>
<thead>
<tr>
<th>[R01] Completeness</th>
<th>[R06] Portability</th>
</tr>
</thead>
<tbody>
<tr>
<td>[R02] Non-Interference</td>
<td>[R07] DSL Runtime Integration</td>
</tr>
<tr>
<td>[R03] Genericity</td>
<td>[R08] Tool Integration</td>
</tr>
<tr>
<td>[R04] Composability</td>
<td>[R09] Minimize the Gap</td>
</tr>
<tr>
<td>[R05] Unanticipated Monitoring</td>
<td>[R10] Break the Rules</td>
</tr>
</tbody>
</table>
Background: Kishon’s Monitoring Semantics

valuation wrapped with pre and post

Interpreter + Monitor = Monitoring Interpreter

- **Cont.**
  - **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

- 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
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`
Composition Operator

**Only inheritance:**

- **[R01]** decorate all terms
- **[R06]** no reflection needed
- **[R07]** no modifications to legacy
- **[R08]** a simple link to the monitors

```
<table>
<thead>
<tr>
<th>Composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element</td>
</tr>
<tr>
<td>Decorator</td>
</tr>
<tr>
<td>MonitorLink</td>
</tr>
<tr>
<td>Evaluator</td>
</tr>
</tbody>
</table>
```

- **syntax**
  - `<<Interface>>`
  - `IVisitor<T>`
  - `IDecoratorVisitor<T>`

- **semantics**
  - `accept(visitor: IVisitor<T>): T`
  - `visit(node: Decorator): T`
  - `visit(node: Decorator): Value`

[13]
Composable Monitors

```java
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) }
```
Unanticipated Monitoring

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

`IDecorator & MonitoringEvaluator` **out**

**Drawbacks:** code less homogeneous
Interferes with other visitors
Non-Interference vs Breaking the Rules

• **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

---

Trade-off needed!
Illustration: Lambda Calculus

Automatic generation of the Composition Layer
Monitor 1: A Simple Tracer

eval.getEnv().lookup(arg)
Monitor 1: A Simple Tracer

Usage scenario

```java
tracer = new Tracer();
link1 = new MonitorLink("mult(x y)", tracer);
link2 = new MonitorLink("fac(x)", tracer);

ast = new LambdaParser("letrec mult=x.y. [link1]exp (* x y) in
letrec fact=x. [link2]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

**DSProfile:**
- implemented in Scala,
- used as black-box

### Profiling results:

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Self</th>
<th>Desc</th>
<th>Count</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>%</td>
<td>%</td>
<td>%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>99.6</td>
<td>0.6</td>
<td>13</td>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>13</td>
<td>98.9</td>
<td>3</td>
<td>26.8</td>
<td>10</td>
<td>72.1</td>
</tr>
<tr>
<td>13</td>
<td>98.4</td>
<td>2</td>
<td>17.5</td>
<td>11</td>
<td>80.9</td>
</tr>
<tr>
<td>13</td>
<td>98.3</td>
<td>4</td>
<td>33.0</td>
<td>9</td>
<td>65.3</td>
</tr>
<tr>
<td>1</td>
<td>12.1</td>
<td>1</td>
<td>12.1</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>10.0</td>
<td>1</td>
<td>10.0</td>
<td>0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Total time: 14 ms; profiled time (95.9%)

1003 profile records

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Self</th>
<th>Desc</th>
<th>Count</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>%</td>
<td>%</td>
<td>%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>99.6</td>
<td>0.6</td>
<td>13</td>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>13</td>
<td>98.9</td>
<td>3</td>
<td>26.8</td>
<td>10</td>
<td>72.1</td>
</tr>
<tr>
<td>13</td>
<td>98.4</td>
<td>2</td>
<td>17.5</td>
<td>11</td>
<td>80.9</td>
</tr>
<tr>
<td>13</td>
<td>98.3</td>
<td>4</td>
<td>33.0</td>
<td>9</td>
<td>65.3</td>
</tr>
<tr>
<td>1</td>
<td>12.1</td>
<td>1</td>
<td>12.1</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>10.0</td>
<td>1</td>
<td>10.0</td>
<td>0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

[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

- 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

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

**Not so easy**
- Time & non-interference?
- Distributed monitoring

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

Discussion & Questions
DSL Monitoring Pattern