Monilogging for Executable DSLs

Dorian Leroy[1,2], Benoit Lelandais[3], Marie-Pierre Oudot[3], Benoit Combemale[1,2,4,5]

[1]Université de Rennes 1, [2]Inria, [3]CEA-DAM, [4]CNRS, [5]IRISA

Mardi 15 Juin 2021

Introduction – Logging and Runtime Monitoring

- Analyzing **complex or data-intensive behaviors** requires insightful data (*e.g.*, software intensive systems, scientific computing).
- **Logging** and **runtime monitoring** can be used to gather such data.
- Most often used in an **ad-hoc way** through language constructs or language-specific libraries.

- **Acceptable when using GPLs**, as they can address numerous concerns, less so in the context of Domain-Specific Languages (DSLs).

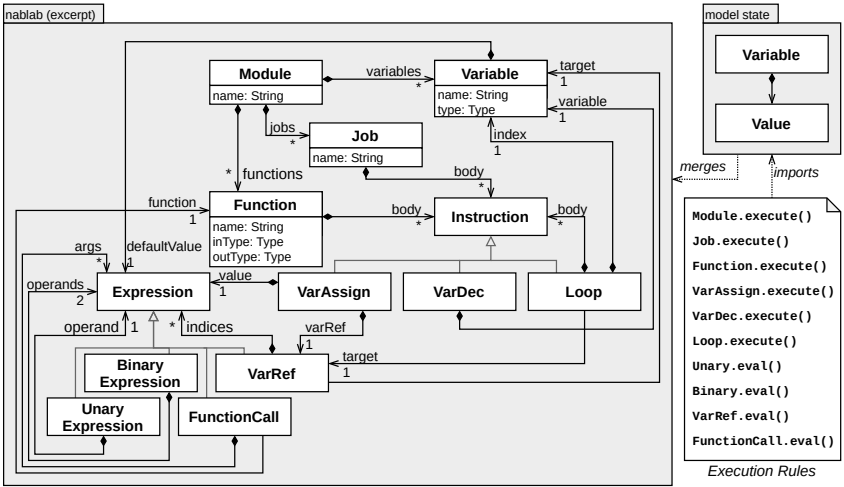
Introduction – Executable DSLs

DSLs allow to bridge the gap between domain experts and software realizing their models through:

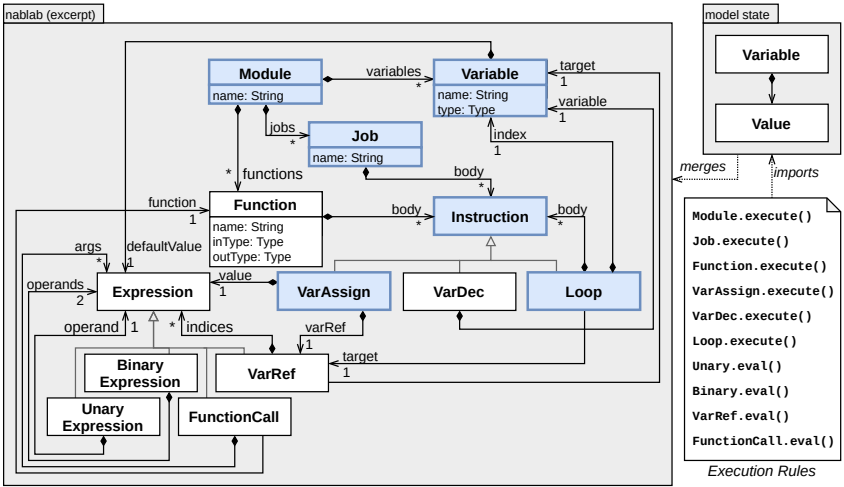
- a dedicated language reifying the **concepts of a domain**,
- an ecosystem of tools for **editing**, **manipulating** and **statically analyzing** defined models, and
- the automated **generation of artifacts** from the models defined by stakeholders.

To enable **dynamic analysis** (*e.g.*, interactive debugging) over its models, a DSL must be made executable (*i.e.*, an xDSL) by providing an execution semantics for it.

Introduction – Example xDSL: NabLab



Introduction – Example xDSL: NabLab



Introduction – Debugging NabLab Models

Available facilities:

- Output capabilities ⇒ designed for **production use**, not debugging.

```
VtkOutput
{
    periodReferenceVariable = iterativeHeatEquation.n;
    outputVariables = iterativeHeatEquation.u as "Temperature";
}
```

- Interactive debugging ⇒ **impractical** for such highly iterative software.

```
UpdateU: ∀c∈cells(), u^{n+1, k+1}{c} =
           u^{n}{c} + α{c, c} * u^{n+1, k}{c} +
           ∑{d∈neighbourCells(c)} (α{c, d} * u^{n+1, k}{d});
```

Preferred approach:

- **Logging and monitoring** of domain-specific properties
  (*e.g.*, physics conservation laws, invariants on numerical scheme variables).

Introduction – Logging and Monitoring for xDSLs

General obstacles to **domain-level** logging and monitoring facilities for xDSLs:

- Restricted DSL expressivity:
  - Introducing needed language constructs **goes against SoC** (*e.g.*, `printf`, `if`)
  - **Different expressivity** than offered by the DSL might be required
- Domain-specificness:
  - Cannot reuse existing libraries with **domain concepts** (*e.g.*, Apache log4x)
  - Additional development costs **for each DSL** to support

Introduction – Logging and Monitoring Complementarities

Logging and monitoring are often **dependent on one another**:

- Monitoring can **operate on derived data** obtained through logging mechanisms
- Logging can be **triggered or altered** upon (in)validation of monitored properties

Yet, obstacles prevent **domain experts** from leveraging these complementarities:

- Requires **DSL support** for logging and monitoring frameworks
- Requires **high abstraction-level interoperability** between frameworks

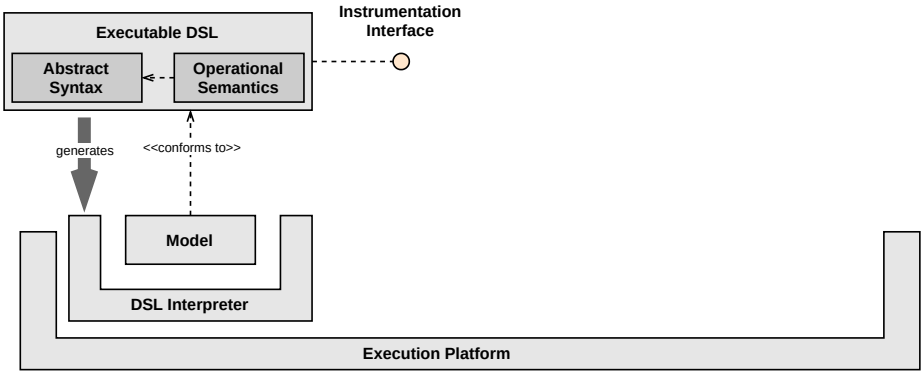
Introduction – Monilogging for xDSLs

Proposed solution:

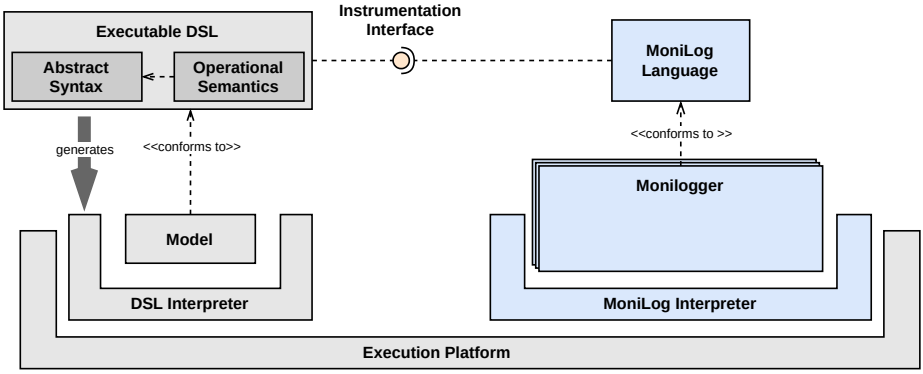
- **Language-agnostic, unifying framework** for runtime monitoring and logging allowing to define loggers, runtime monitors and combinations of the two, a.k.a. ***moniloggers***.

Overview of the Approach

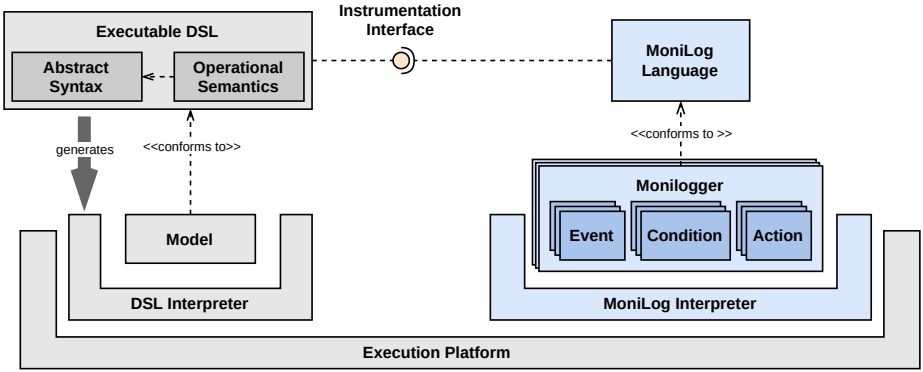
Overview of the Approach



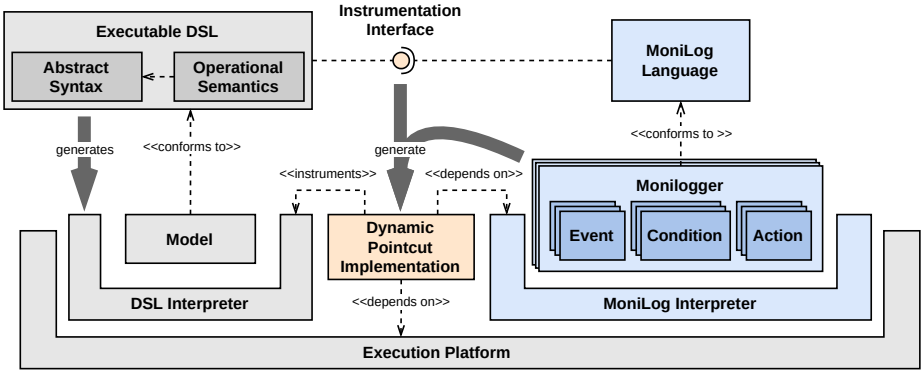
Overview of the Approach



Overview of the Approach

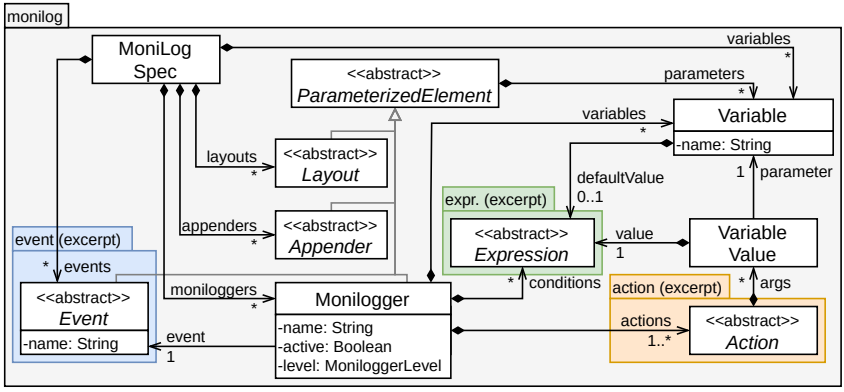


Overview of the Approach

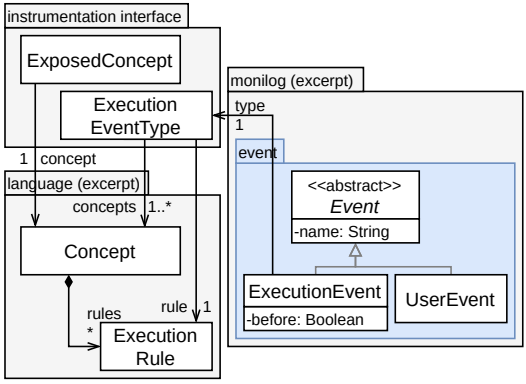


The MoniLog Language

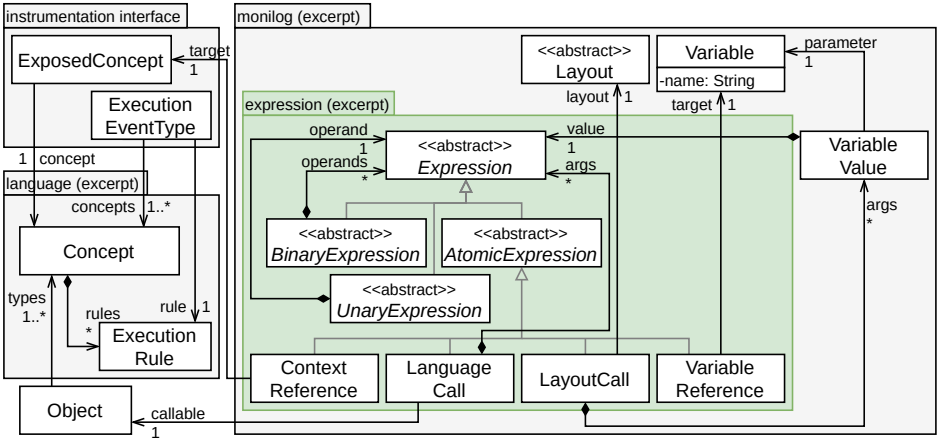
The MoniLog Language – Metamodel



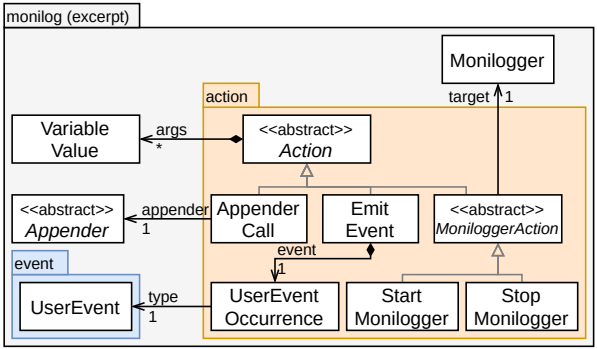
The MoniLog Language – Metamodel



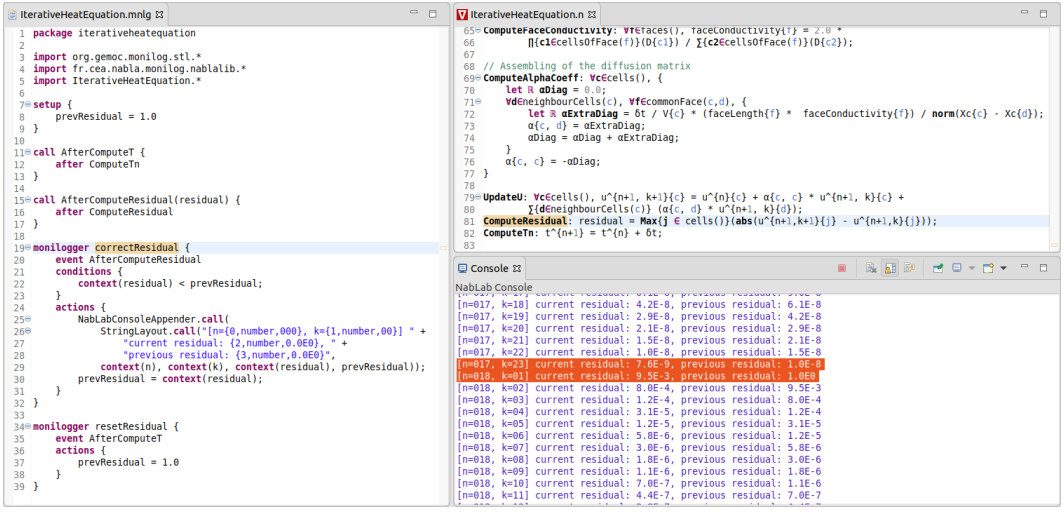
The MoniLog Language – Metamodel



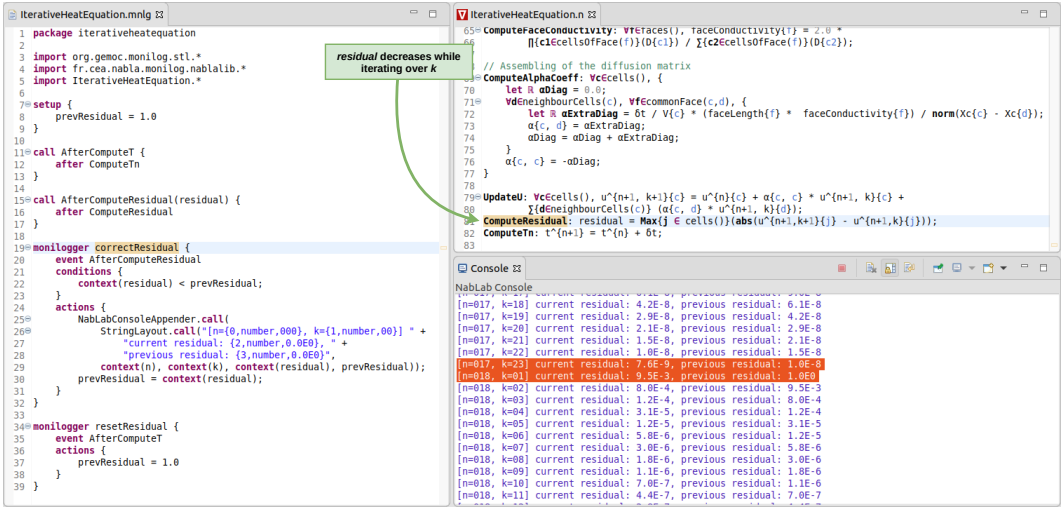
The MoniLog Language – Metamodel



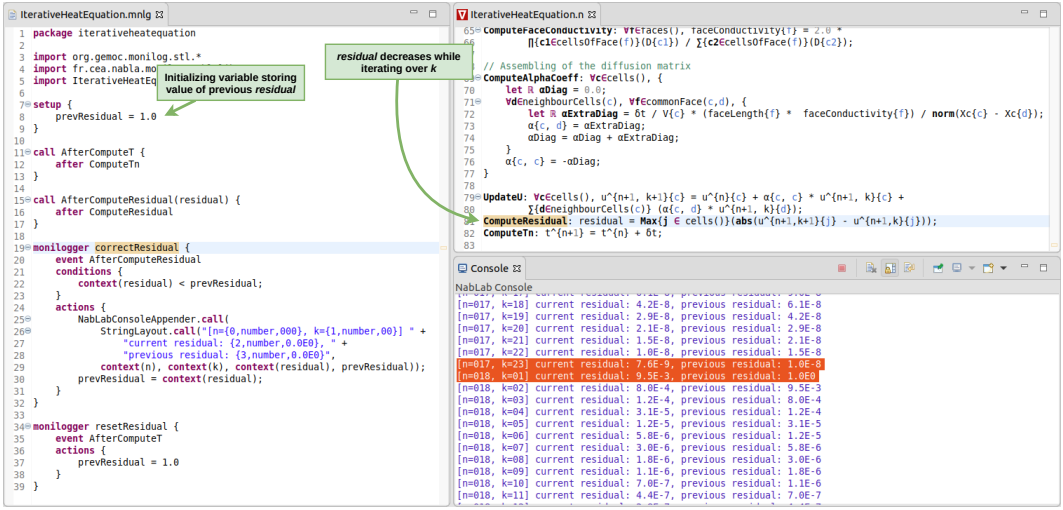
The MoniLog Language – Example



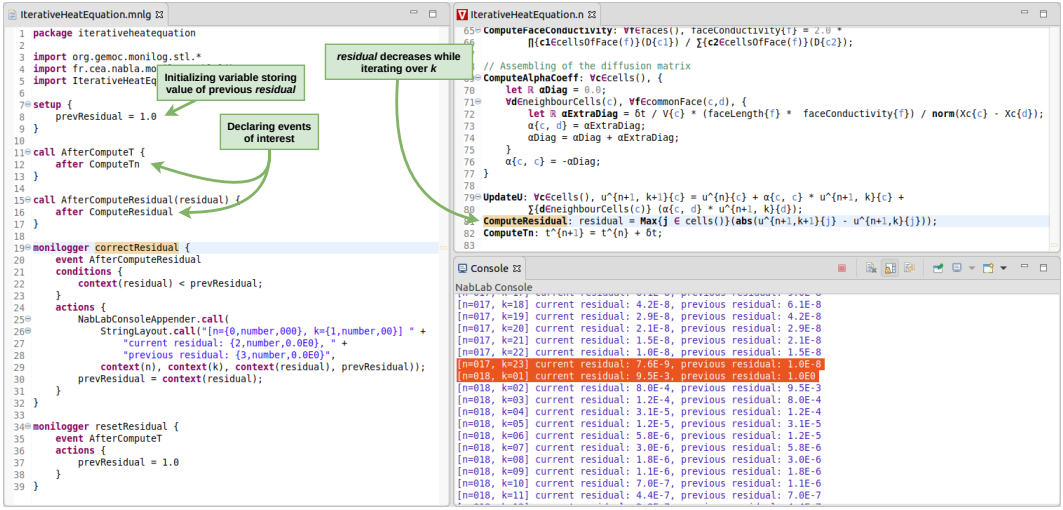
The MoniLog Language – Example



The MoniLog Language – Example



The MoniLog Language – Example

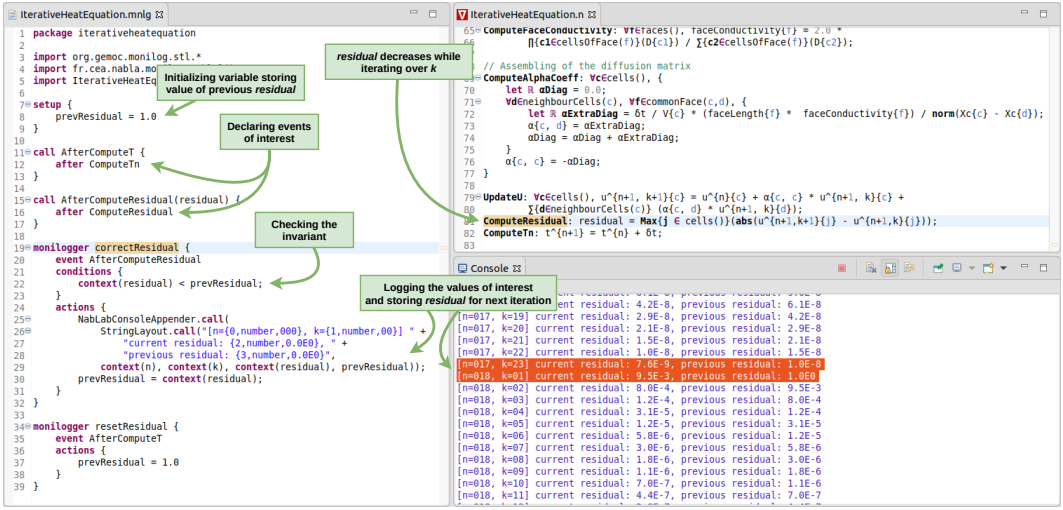



```
package iterativeheatequation

import org.gemoc.monilog.stl.*
import fr.cea.nabla.mo
import IterativeHeatEq

setup {
    prevResidual = 1.0
}

call AfterComputeT {
    after ComputeTn
}

call AfterComputeResidual(residual) {
    after ComputeResidual
}

monilogger correctResidual {
    event AfterComputeResidual
    conditions {
        context(residual) < prevResidual;
    }
    actions {
        NabLabConsoleAppender.call(
            StringLayout.call("[n={0,number,000}, k={1,number,00}] " +
                "current residual: {2,number,0.0E0}, " +
                "previous residual: {3,number,0.0E0}",
            context(n), context(k), context(residual), prevResidual));
        prevResidual = context(residual);
    }
}

monilogger resetResidual {
    event AfterComputeT
    actions {
        prevResidual = 1.0
    }
}
```

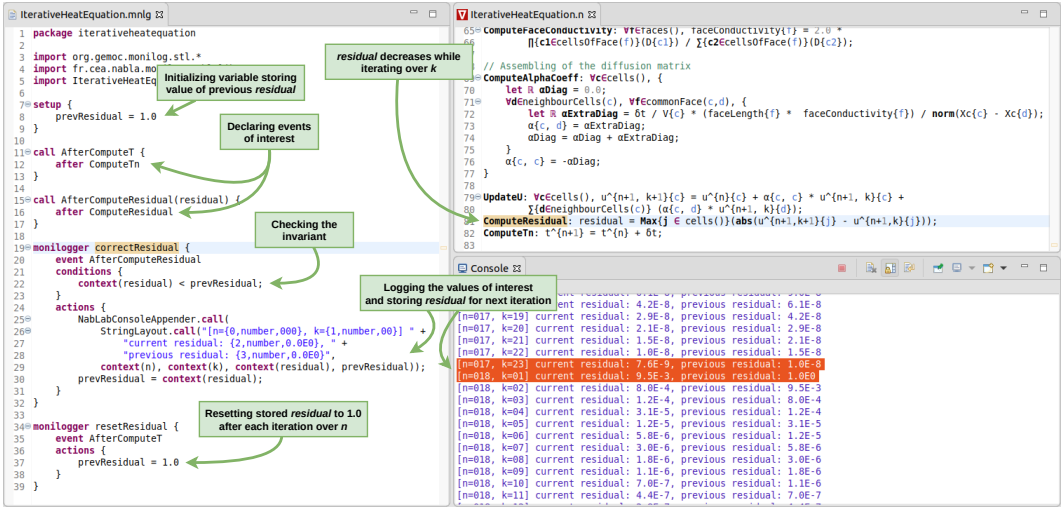
```
ComputeFaceConductivity: ∀f∈Faces(), faceConductivity{f} = 2.0 *
        ∏{c1∈cellsOfFace(f)}(D{c1}) / ∑{c2∈cellsOfFace(f)}(D{c2});

// Assembling of the diffusion matrix
ComputeAlphaCoeff: ∀c∈cells(), {
    let ℝ αDiag = 0.0;
    ∀d∈neighbourCells(c), ∀f∈commonFace(c,d), {
        let ℝ αExtraDiag = δt / V{c} * (faceLength{f} *  faceConductivity{f}) / norm(Xc{c} - Xc{d});
        α{c, d} = αExtraDiag;
        αDiag = αDiag + αExtraDiag;
    }
    α{c, c} = -αDiag;
}

UpdateU: ∀c∈cells(), u^{n+1, k+1}{c} = u^{n}{c} + α{c, c} * u^{n+1, k}{c} +
        ∑{d∈neighbourCells(c)} (α{c, d} * u^{n+1, k}{d});
ComputeResidual: residual = Max{j ∈ cells()}(abs(u^{n+1,k+1}{j} - u^{n+1,k}{j}));
ComputeTn: t^{n+1} = t^{n} + δt;
```

```
NabLab Console
[n=017, k=18] current residual: 4.2E-8, previous residual: 6.1E-8
[n=017, k=19] current residual: 2.9E-8, previous residual: 4.2E-8
[n=017, k=20] current residual: 2.1E-8, previous residual: 2.9E-8
[n=017, k=21] current residual: 1.5E-8, previous residual: 2.1E-8
[n=017, k=22] current residual: 1.0E-8, previous residual: 1.5E-8
[n=017, k=23] current residual: 7.6E-9, previous residual: 1.0E-8
[n=018, k=01] current residual: 9.5E-3, previous residual: 1.0E0
[n=018, k=02] current residual: 8.0E-4, previous residual: 9.5E-3
[n=018, k=03] current residual: 1.2E-4, previous residual: 8.0E-4
[n=018, k=04] current residual: 3.1E-5, previous residual: 1.2E-4
[n=018, k=05] current residual: 1.2E-5, previous residual: 3.1E-5
[n=018, k=06] current residual: 5.8E-6, previous residual: 1.2E-5
[n=018, k=07] current residual: 3.0E-6, previous residual: 5.8E-6
[n=018, k=08] current residual: 1.8E-6, previous residual: 3.0E-6
[n=018, k=09] current residual: 1.1E-6, previous residual: 1.8E-6
[n=018, k=10] current residual: 7.0E-7, previous residual: 1.1E-6
[n=018, k=11] current residual: 4.4E-7, previous residual: 7.0E-7
```

The MoniLog Language – Example



The MoniLog Language – Example



The MoniLog Language – Semantics

Semantics of MoniLog formalized as a structural operational semantics, and implemented in both AspectJ and Truffle.

At a glance:

- A single event can trigger several moniloggers
- A triggered monilogger executes fully before the execution resumes
- When several moniloggers are triggered, they execute in arbitrary order

Implementation

Implementation – AspectJ

- Applicable to **Java-based** interpreters
- **Non-intrusive** w.r.t. language definition
- Instrumentation interface = aspects **weaved into the interpreter**:

```
pointcut interpreteJob(Job job, Context context) :
call(public static void fr.cea.nabla.ir.interpreter.JobInterpreter.interprete(Job, Context)) &&
        args(job, context);

after(Job job, Context context) : interpreteJob(job, context) {
    notifyAfter(job.getName(), null, context);
}

before(Job job, Context context) : interpreteJob(job, context) {
    notifyBefore(job.getName(), context);
}
```

Implementation – Truffle Language Implementation Framework

- Applicable to **Truffle-based** interpreters (*e.g.*, JavaScript, Python, R)
- Can evaluate expressions in **any language** installed on the GraalVM
- Instrumentation interface **part of language definition**:

```
public abstract class NablaWriteVariableNode
        extends NablaInstructionNode
        implements InstrumentableNode, TruffleObject {
    @Override
    public boolean hasTag(Class<? extends Tag> tag) {
        return tag.equals(StandardTags.WriteVariableTag.class) || super.hasTag(tag);
    }
```

Evaluation

Evaluation

Setup:

CPU: Intel® Core™ i7-9850H CPU @ 2.60GHz × 12

OS: Ubuntu 20.04.2

VM: GraalVM 21.1.0

Preliminary results:

Interpreter	Baseline Execution Time	Overhead (Console)	Overhead (File)
Java-based	≈ 21s	≈ 3s / 12.5%	≈ 1s / 4.7%
Truffle-based	≈ 8s	≈ 3s / 37.5%	≈ 1s / 12.5%

Overhead Source	Overhead Contribution
Handling *prevResidual*	≈ 0.5s
Logging to console	≈ 2.5s
Logging to file	≈ 0.5s

Conclusion

Conclusion

Conclusion:

- We propose MoniLog, a new language to define *moniloggers*, *i.e.*, a combination of loggers and runtime monitors
- Moniloggers are defined in a language-agnostic way, relying on an **instrumentation interface** provided by DSLs
- Running models are instrumented with moniloggers, allowing domain experts to **leverage the complementarities** between logging and monitoring

Future work:

- Monilogging for compiled DSLs
- Polyglot editing support for language expressions