Debugging Scientific Software

Dorian Leroy

Inria, DiverSE

December 15, 2021
Scientific computing

Use of advanced computing capabilities to solve complex problems, aiming to predict the behavior or outcome of a system, man-made or otherwise.

Why is it important?

Aerospace engineering, mechanical engineering, material science, chemistry, medicine and many more disciplines, but also...

Basis of scientific findings shaping policy regarding wicked problems such as COVID-19 or climate change.
The engineering of scientific software
The engineering of scientific software

Stakeholders:
- Scientists
- Numerical Analysts
- Software Engineers
Scientists develop a mathematical model, i.e. a continuous numerical model (e.g. a system of integro-differential equations)
Numerical analysts discretize the mathematical model into a numerical scheme computing a numerical solution given...

- discretization of the domain (e.g. space, time),
- initial state (e.g. initial temperature), and
- inputs from environment (e.g. rainfalls).
The engineering of scientific software

Software engineering concerns tackled as part of the SE V-Model

- Stakeholders
- Requirements
- SE V-Model
- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing
- Implementation Testing
- Unit & Integration Testing
- Scientific Software
- SE Tools & Methods
- Observations
- Theories
- Discretization Method
- Mathematical Model
Aim: ensure that the numerical scheme a correct discretization of the mathematical model

Mathematical Model

Discretization Testing

Scientific Software

SE V-Model

Stakeholders

Requirements

SE Tools & Methods

Implementation Testing

System Testing

Model Testing

Acceptance Testing

Observations

Theories

Discretization Method

Numerical Scheme
The engineering of scientific software

Aim: quantify disagreement between numerical solutions and experimental data
We investigated the impact of language choice through the balance of responsibilities between language users and language designers.
Language matters! Power comes with responsibility!

We investigated the impact of language choice through the balance of responsibilities between language users and language designers.
Balance of responsibilities: languages for mathematical models

Example languages
Mathematica, MATLAB

Balance of responsibilities

Language users
- Model testing to assess mathematical model fidelity
- Discretization testing on the derived scientific software

Language designers
- Discretization testing of the provided continuous mathematical constructs
- Software engineering V&V concerns
- Providing tools for discretization testing
Balance of responsibilities: languages for scientific software

Example languages
C++, Fortran

Balance of responsibilities

Language users
- Software engineering V&V concerns
- Implementation testing
- Discretization testing
- Model testing
Balance of responsibilities: languages for numerical schemes

Example languages
NabLab, Julia, SciPy, GNU Octave

Balance of responsibilities

Language users
- Implementation testing
- Discretization testing
- Model testing

Language designers
- Software engineering V&V concerns
- Implementation testing of the provided discrete mathematical constructs
- Providing tools for implementation testing
Example language for numerical schemes: \texttt{NabLab}

\textbf{NabLab}: Executable DSL (xDSL) for \textit{numerical analysis}.

\textbf{Abstract syntax}: Metamodel reifying \textit{domain concepts}, \textit{e.g.},
- Matrices, vectors, scalars
- Algebraic expressions over those
- Mathematical functions
- Iterative control structures

\textbf{Operational semantics}:
- Metamodel defining \textit{model state} during the execution
- Set of execution rules $\Rightarrow$ \texttt{Interpreter}
Example NabLab model: Solving the heat equation with iterative numerical method.

```plaintext
iterate n while (t^{n+1} < stopTime && n+1 < maxIterations),
    k while (residual > \varepsilon && check(k+1 < maxIterationsK));

UpdateU: \forall c \in \text{cells()}, u^{n+1, k+1}{c} = u^{n+1}{c} + \alpha{c, c} * u^{n+1, k}{c} + 
    \sum_{d \in \text{neighbourCells}(c)} (\alpha{c, d} * u^{n+1, k}{d});

ComputeResidual: residual = \text{Max}_{j \in \text{cells()}}(\text{abs}(u^{n+1,k+1}{j} - u^{n+1,k}{j}));

ComputeTn: t^{n+1} = t^{n} + \Delta t;
```

Example invariant to check: residual_{n,k} > residual_{n,k+1}
Example NabLab model: Solving the heat equation with iterative numerical method.

```
iterate n while (t^[n+1] < stopTime && n+1 < maxIterations),
  k while (residual > ε && check(k+1 < maxIterationsK));

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;
```

Example invariant to check: residual^[n,k] > residual^[n,k+1]
Example NabLab model: Solving the heat equation with iterative numerical method.

iterate n while (t\^{n+1} < \text{stopTime} \&\& n+1 < \text{maxIterations}),
  \text{k while (residual > } \varepsilon \&\& \text{check}(k+1 < \text{maxIterationsK});

UpdateU: \forall c \in \text{cells}, u^{n+1, k+1}\{c\} = u^{n}\{c\} + \alpha\{c, c\} \times u^{n+1, k}\{c\} + \\
\sum_{d \in \text{neighbourCells}(c)} \alpha\{c, d\} \times u^{n+1, k}\{d\};

\text{ComputeResidual: residual} = \max_{j \in \text{cells}}(\abs{u^{n+1, k+1}\{j\} - u^{n+1, k}\{j\}}); \\
\text{ComputeTn: } t^{n+1} = t^{n} + \delta t;

Example invariant to check: \text{residual}_{n, k} > \text{residual}_{n, k+1}
Example NabLab model: Solving the heat equation with iterative numerical method.

iterate \( n \) while \((t^{n+1} < \text{stopTime} \&\& n+1 < \text{maxIterations})\),
   \( k \) while \((\text{residual} > \varepsilon \&\& \text{check}(k+1 < \text{maxIterationsK}))\);

UpdateU: \( \forall c \in \text{cells()}, \ u^{n+1, k+1}\{c\} = u^n\{c\} + \alpha\{c, c\} \ast u^{n+1, k}\{c\} + \sum_{d \in \text{neighbourCells}(c)} (\alpha\{c, d\} \ast u^{n+1, k}\{d\})\);

ComputeResidual: residual = Max\{j \in \text{cells()}\}(abs(u^{n+1,k+1}\{j\} - u^{n+1,k}\{j\}))

ComputeTn: \( t^{n+1} = t^n + \delta t \);

Example invariant to check: \( \text{residual}_{n,k} > \text{residual}_{n,k+1} \)
Example NabLab model: Solving the heat equation with iterative numerical method.

```plaintext
iterate n while (t{n+1} < stopTime && n+1 < maxIterations),
  k while (residual > ε && check(k+1 < maxIterationsK));

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

Example invariant to check: residualₙₖ > residualₙₖ₊₁
Example NabLab model: Solving the heat equation with iterative numerical method.

iterate n while (t^{n+1} < stopTime && n+1 < maxIterations),
    k while (residual > \varepsilon && check(k+1 < maxIterationsK));

UpdateU: \forall c \in cells(), u^{n+1, k+1} \{c\} = u^{n} \{c\} + \alpha_{c, c} * u^{n+1, k} \{c\} +
    \sum_{d \in neighbourCells(c)} (\alpha_{c, d} * u^{n+1, k} \{d\});

ComputeResidual: residual = Max\{j \in cells()\}(abs(u^{n+1, k+1} \{j\} - u^{n+1, k} \{j\}));

ComputeTn: t^{n+1} = t^{n} + \delta t;

Example invariant to check: residual_{n,k} > residual_{n,k+1}
Debugging NabLab models

Available facilities:

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

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

- Interactive debugging ⇒ impractical for such highly iterative software.

```csharp
UpdateU: Vc Cells(), u^{n+1, k+1}[c] = 
u^{n}[c] + \alpha[c, c] \ast u^{n+1, k}[c] + \sum_{d \in neighbourCells(c)} (\alpha[c, d] \ast u^{n+1, k}[d]);
```

Preferred approach:

- Logging and monitoring of domain-specific properties (e.g., physics conservation laws, numerical invariants).
Logging and monitoring for xDSLs

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

- Restricted DSL expressivity:
  - Introducing language constructs goes against SoC (e.g., printf, if)
  - Different expressivity than offered by the DSL might be required

- Domain-specificness:
  - Cannot reuse libraries through domain concepts (e.g., Apache log4x)
  - Additional development costs for each DSL to support
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 **domain-level interoperability** between frameworks
Proposed solution: **MoniLog**

- **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

Executable DSL

Abstract Syntax
Operational Semantics

Model
DSL Interpreter

Instrumentation Interface

Execution Platform

generates

<<conforms to>>
Overview of the approach

Executable DSL

- Abstract Syntax
- Operational Semantics

<<conforms to>>

Model

<<conforms to>>

MoniLog Language

Instrumentation Interface

Monilogger

DSL Interpreter

MoniLog Interpreter

Execution Platform
Overview of the approach

Executable DSL
- Abstract Syntax
- Operational Semantics

MoniLog Language
- Event
- Condition
- Action

Monilogger

Model

DSL Interpreter

MoniLog Interpreter

Execution Platform
Overview of the approach

Executable DSL

- Abstract Syntax
- Operational Semantics

<<conforms to>>

Model

MoniLog

Language

<<conforms to>>

Monilogger

Event
Condition
Action

<<depends on>>

Dynamic
Pointcut
Implementation

<<depends on>>

Instrumentation
Interface

<<depends on>>

Instrumentation
Interface

Execution Platform

DSL Interpreter

MoniLog Interpreter
Overview of MoniLog

A MoniLog specification allows to

- define instrumentation-specific variables,
- declare the execution events of interest,
- register moniloggers to these events, which can
  - update instrumentation variables,
  - access the execution state of the running model,
  - evaluate expressions with languages available on the execution platform,
  - call various appenders (e.g., file, message queue, console),
  - start/stop moniloggers
Example use on NabLab

```java
import org.gencore.manilog.*;
import fr.sceph.nlabia.manilog.nablablib.*;
import IterativeHeatEquation.*;

setup {
    prevResidual = 1.0;
}

computeTnReturned {
    after call ComputeTn
}

residualUpdated {
    after call ComputeResidual
}

monologger correctResidual {
    when residualUpdated
    if (context(residual) > prevResidual)
        then {
            NabLabConsoleAppender.call{
                StringLayout.call("\n=[n=0, number=090], k=[1, number=060] \n* Incorrect residual | previous residual = 2.0 (number=0.000) | \n* previous residual = 3.0 (number=0.000) | \n* context\(\text{residual}\), context\(\text{context(residual)}\), prevResidual\);\n                correctResidual.stop();
            }
        }
        else {
            NabLabConsoleAppender.call{
                StringLayout.call("\n=[n=0, number=090], k=[1, number=060] \n* Incorrect residual | previous residual = 3.0 (number=0.000) | \n* previous residual = 2.0 (number=0.000) | \n* context\(\text{context(residual)}\), prevResidual\);\n                prevResidual = context\(\text{residual}\);
            }
        }
    }

monologger resetResidual {
    when ComputeTnReturned
    prevResidual = 1.0;
}
```
Example use on **NabLab**

**IterativeHeatEquation-MonitorResidual.mxml**

```java
package iterativeheatequation

import org.gnome.monolog.*;
import fr.cea.nablab.monolog.nablalib.*;
import IterativeHeatEquation.*;

setup {
    prevResidual = 1.0;
}

event ComputeTnReturned {
    after call ComputeTn
}

event ResidualUpdated {
    after call ComputeResidual
}

monilogModule correctResidual {
    when ResidualUpdated {
        if (context(residual) > prevResidual) {
            NabLabConsoleAppender.call("[n=\text{0}, number, 000], k=\text{1}, number, 000] +
                \text{incorrect residual} (2, number, 0.000) * +
                \text{previous residual} (3, number, 0.000). context(n), context(residual), prevResidual);\n            correctResidual() +
        } else {
            NabLabConsoleAppender.call("[n=\text{0}, number, 000], k=\text{1}, number, 000] +
                \text{incorrect residual} (2, number, 0.000) * +
                \text{previous residual} (3, number, 0.000). context(n), context(residual), prevResidual);\n            prevResidual = context(residual);
        }
    }
}

monilogModule resetResidual {
    when ComputeTnReturned {
        prevResidual = 1.0;
    }

```
Example use on NabLab

```java
import org.gnomelab.nabl;
import fr.cnrs.nabl;
import IterativeNab;

setup {
    prevResidual = 1.0;
}

event ComputeTnReturned {
    after call ComputeTn
}

event ResidualUpdated {
    after call ComputeResidual
}

nonlogger correctResidual {
    when ResidualUpdated
    if (context(residual) > prevResidual)
    then {
        NabLabConsoleAppender.call("
            "prevResidual = context(residual);
        }"
    } else {
        NabLabConsoleAppender.call("
            "prevResidual = context(residual);
        }
}

nonlogger resetResidual {
    when ComputeTnReturned
    prevResidual = 1.0;
}
```

Residual decreases while iterating over \( k \)

Initializing variable storing value of previous residual
Example use on **NabLab**

```java
import org.gemoc.modelingcore.constraints.*;
import fr.cnrs.nabl.
import IterativeMesh;

setup {
    prevResidual = 1.0;
}

event ComputeTnReturned {
    after call ComputeTn
}

event ResidualUpdated {
    after call ComputeResidual
}

nonlogger correctResidual {
    when ResidualUpdated
        if (context(residual) > prevResidual)
            then {
                NabLabConsole.printer.call("r", [n, number, 000], [k, 1, number, 000], [prevResidual], [context(residual)], [context(residual)], [prevResidual]);
                resetResidual.stop();
            } else {
                NabLabConsole.printer.call("r", [n, number, 000], [k, 1, number, 000], [prevResidual], [context(residual)], [context(residual)], [prevResidual]);
                prevResidual = context(residual);
            }
}

nonlogger resetResidual {
    when ComputeTnReturned
        prevResidual = 1.0;
}

// Initializing variable storing value of previous residual
residual decreases while iterating over k
Declaring events of interest
```
Initialize variable storing value of previous residual

Declaring events of interest

Checking the invariant

residual decreases while iterating over $k$

Initializing variable storing value of previous residual

Declaring events of interest

Checking the invariant

Example use on NabLab
Example use on NabLab

```java
import org.geocentral.nablab.
import fr.cea.nablab.
import IterativeM

package iterativeheat.equation

setup { prevResidual = 1.0; }

event ComputeTnReturned { after call ComputeTn }

event ResidualUpdated { after call ComputeResidual }

monilogger correctResidual {
  when ResidualUpdated
  if (context(residual) > prevResidual)
    then { NabLabConsoleAppender.call( StringLayout.call("\n\nIncorrect residual" + "current residual: \(2, number, 0.000\) + \n\nprevious residual: \(3, number, 0.000\)\n\ncontext(n), context(residual), prevResidual); correctResidual.stop(); resetResidual.stop(); } else { NabLabConsoleAppender.call( StringLayout.call("\n\nIncorrect residual" + "current residual: \(2, number, 0.000\) + \n\nprevious residual: \(3, number, 0.000\)\n\ncontext(n), context(residual), prevResidual); prevResidual = context(residual); }

monilogger resetResidual {
  when ComputeTnReturned
  then { prevResidual = 1.0; }
}
```
Example use on **NabLab**

- Initializing variable storing value of previous residual
- Declaring events of interest
- Checking the invariant
- Resetting stored residual to 1.0 after each iteration over \( n \)
- Residual decreases while iterating over \( k \)
- Initializing variable storing value of previous residual
- Declaring events of interest
- Checking the invariant
- Logging the values of interest and storing residual for next iteration
- Resetting stored residual to 1.0 after each iteration over \( n \)
Implementation – AspectJ

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

```java
pointcut interpreteJob(Job job, Context context) :
call(public static void fr.cea.nabla.ir.interpreter.JobInterpreter.interpette(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);
}
```
Applicable to languages with a **Truffle-based** interpreter (e.g., Python, R)
- Can evaluate expressions in **any language** installed on the GraalVM
- Instrumentation interface **part of language definition**:

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

Research questions:

RQ#1 How does the proposed approach allow to combine runtime monitoring and logging to extract relevant data from running models?
  ► Answered through demonstration cases similar to the provided example.

RQ#2 How is the overhead induced by the approach affected by different scenarios?
  ► Answered through quantitative evaluation
RQ#1: Demonstration case (Coarsen Interval)

- Log standard deviation of $u_n$ to file at interval of 0.0001
- When standard deviation less than 0.2, increase interval to 0.01
  - Derived data leveraged by monitor
  - Monitor affects logger behavior
RQ#2: Quantitative evaluation

Setup:
CPU: Intel® Core™ i7-9850H CPU @ 2.60GHz × 12
OS: Ubuntu 20.04.2, VM: GraalVM 21.1.0

Overhead induced by 3 MoniLog specifications over simulation times from 0.2 to 1.0

AspectJ:
baseline: from ≈42.6s to ≈134.89s
absolute: from ≈8.75s to ≈18.27s
relative: from ≈26% to ≈16%

Truffle:
baseline: from ≈10.21s to ≈29.76s
absolute: from ≈2.75s to ≈4.85s
relative: from ≈36.5% to ≈19.5%

Suitable for debugging as absolute overhead reasonably low on shorter execution times, and relative overhead decreases by 40 to 50% on longer execution times.
Prerequisites:

- \texttt{MoniLog} host language interpreter embeddable in target language
- Extend code generators to generate model-specific instrumentation interface

The generated model-specific instrumentation interface is split between:

- the target language of the DSL’s code generator, to expose runtime data and events of the model, and
- the host language of \texttt{MoniLog}, to execution events of the model from moniloggers.
MoniLog for compiled DSLs (ongoing work)

NabLab Model

Model-Specific Instrumentation Interface (Python side)
- Exposes call events (jobs and functions)
- Exposes write events (local and global variables)

Moniloggers (Python)

MoniLog (Python)

MoniLog (C++)

Generated C++ Code

Model-Specific Instrumentation Interface (C++)
- Exposes scoped runtime data in Python module
- Wraps execution events with calls to MoniLog

Embedded CPython Interpreter

User-defined

Generated

Generic

User-defined → Generated

dependency

generation

Generic user-defined user-defined

User-defined → Generated

dependency

generation

Generated → Generic

dependency

generation

User-defined

Generated

Generic
Example use of Python-based **MONILOG**

```python
src > iterativeheatequation > logStddev.py > ...
1  import iterativeheatequation as ihe
2  from monilog import *
3  import statistics
4
5  currentTime = 0
6  outputInterval = 0.0001
7  stddev = 1.0
8
9  @after(ihe.ComputeTn)
10  def logTemperature(context):
11      global currentTime
12      global outputInterval
13      global stddev
14      currentTime += outputInterval
15      if (currentTime <= context.t_n):
16          stddev = statistics.stdev(context.u_n)
17          print("[", + str(context.t_n) + "] stddev=" + str(stddev))
18
19  @after(ihe.ComputeTn)
20  def coarsenInterval(context):
21      global stddev
22      global outputInterval
23      if (stddev <= 0.20):
24          outputInterval = 0.01
25          coarsenInterval.stop()
```
High-level languages allow scientists and numerical analysts to focus on their area of expertise and associated V&V concerns.

Designers of high-level languages must guarantee correctness and performance of derived scientific software.

Designers must furthermore give tools to address the V&V concerns corresponding to the level of abstraction of the language.

In the context of languages for numerical schemes such as NabLab, MoniLog is particularly suited to this thanks to:
- its combination of logging and monitoring,
- its ability to use the best suited languages for the task (e.g., Python for data analysis)
Thank you for your attention!

When Scientific Software Meets Software Engineering
Leroy, Dorian, Sallou, June, Bourcier, Johan, Combemale, Benoit Computer 2021

Monilogging for executable domain-specific languages
Leroy, Dorian, Lelandais, Benoît, Oudot, Marie-Pierre, Combemale, Benoit In Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering 2021