Monilogging for Executable Domain-Specific Languages

Dorian Leroy, Benoît Lelandais, Marie-Pierre Oudot, Benoit Combemale

Monilogging for Executable Domain-Specific Languages

Dorian Leroy
dorian.leroy@inria.fr
Université de Rennes 1, Inria
Rennes, France

Benoît Lelandais
benoit.lelandais@cea.fr
CEA, DAM, DIF
Arpajon, France
Université Paris-Saclay, CEA, Laboratoire en Informatique
Haute Performance pour le Calcul et la Simulation
Bruyères-le-Châtel, France

Marie-Pierre Oudot
marie-pierre.oudot@cea.fr
CEA, DAM, DIF
Arpajon, France
Université Paris-Saclay, CEA, Laboratoire en Informatique
Haute Performance pour le Calcul et la Simulation
Bruyères-le-Châtel, France

Benoit Combemale
benoit.combemale@irisa.fr
Université de Rennes 1, Inria, CNRS, IRISA
Rennes, France

Abstract

Runtime monitoring and logging are fundamental techniques for analyzing and supervising the behavior of computer programs. However, supporting these techniques for a given language induces significant development costs that can hold language engineers back from providing adequate logging and monitoring tooling for new domain-specific modeling languages. Moreover, runtime monitoring and logging are generally considered as two different techniques: they are thus implemented separately which makes users prone to overlooking their potentially beneficial mutual interactions. We propose a language-agnostic, unifying framework for runtime monitoring and logging and demonstrate how it can be used to define loggers, runtime monitors and combinations of the two, *aka. moniloggers*. We provide an implementation of the framework that can be used with Java-based executable languages, and evaluate it on 2 implementations of the NabLab interpreter, leveraging in turn the instrumentation facilities offered by Truffle, and those offered by AspectJ.

CCS Concepts: • **Software and its engineering → Domain specific languages**; **Software testing and debugging**.

Keywords: executable DSLs, runtime monitoring, logging

SLE '21, October 17–18, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9111-5/21/10…$15.00
https://doi.org/10.1145/3486608.3486906

ACM Reference Format:
Dorian Leroy, Benoît Lelandais, Marie-Pierre Oudot, and Benoit Combemale. 2021. Monilogging for Executable Domain-Specific Languages. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3486608.3486906

1 Introduction

Complex and data-intensive behaviors such as those observed in scientific computing require insightful data to analyze, whether for verification, validation, or maintenance purposes. Logging and runtime monitoring are two fundamental techniques used – mostly by software developers – to gather such data about the health and behavior of a running system [9, 23, 26, 31]. However, both techniques are usually employed independently and in an *ad hoc* way, requiring the use of dedicated language constructs (*e.g.,* `printf`), or of language-specific libraries written in the same language as the program being instrumented (*e.g.,* calls to the `Logger` class in log4j) [11].

While this is acceptable for developments involving a single general purpose language (GPL), it becomes problematic when dealing with multiple languages, in particular with executable domain-specific languages (DSLs), for two main reasons. First, due to the restricted expressiveness of an executable DSL, equipping it with logging and runtime monitoring facilities requires to introduce the corresponding concerns into the language, even though such concerns might not be part of its domain. Second, the fact that logging and runtime monitoring libraries are dedicated to specific languages (*e.g.,* Apache log4x for C++, PHP, Java and .NET) means that to reuse them in an opportunistic way for an

executable DSL, logging support must be incorporated directly into the execution semantics of the DSL. This goes against the principle of separation of concerns, and induces significant language development costs. There is thus a need for language-agnostic runtime monitoring and logging to provide such capabilities for users of any modeling language and reducing development costs of individual DSLs.

Moreover, runtime monitoring and logging often depend on one another. While both techniques can be used on their own, there are numerous examples of them being used in a complementary manner. For example, runtime monitoring frameworks can operate on data collected from the system through logging, or, conversely, logging can be triggered or altered when specific behaviors are detected by runtime monitors [1]. Yet, a unified framework fitting both the runtime monitoring and the logging needs of modelers – and especially of domain experts with no technical background in computer science – still has to be proposed.

To tackle this challenge, we propose MoniLog, a reusable domain-specific language dedicated to the definition of *moniloggers*, which combine runtime monitoring and logging in a polyglot manner. MoniLog users define moniloggers in external specification files, using domain concepts to indicate how their model should be instrumented. A standard library provides basic logging functionalities to users of the language, who can then extend it with libraries tailored to their specific needs. Moniloggers can be triggered upon occurrences of their associated event. Whether a monilogger is actually triggered or not can be further conditioned by properties on the execution state of both the running model and the MoniLog framework. These properties can be expressed with the expression language embedded in MoniLog, or using languages available on the execution platform (*e.g.,* Python or JavaScript on GraalVM [30]). The actions triggered by moniloggers consist in logging messages or emitting events (both being able to carry data extracted from the execution state), evaluating expressions, and starting or stopping moniloggers.

The approach is implemented as a language with an Xtext concrete syntax, and with an operational semantics leveraging two dynamic instrumentation mechanisms: Truffle [29] and AspectJ [14]. We demonstrate the expressiveness of the approach and the interactions of logging and runtime monitoring through a demonstration case on the NabLab language [15], which is dedicated to numerical simulations. We evaluate the overhead induced by the approach over three representative monilogging scenarios for NabLab, and show that this overhead ranges from 14% on executions shorter than 30s, to 46% on executions shorter than 10s (overhead included), making it suitable for debugging activities.

The remainder of this paper is organized as follows. Section 2 presents the background and the motivation behind this work. Section 3 provides an overview of the contribution. Section 4 introduces the MoniLog language. Section 5 defines the semantics of MoniLog. Section 6 discusses the implementation details of MoniLog. Section 7 presents our evaluation of the proposed approach. Section 8 discusses related work. Finally, a conclusion and future research directions are presented in Section 9.

2 Background & Motivation

In this section, we first introduce the techniques known as runtime monitoring and logging, and precisely scope the executable DSLs we consider in the approach. We then motivate our approach using an illustrative example.

2.1 Background

Runtime Monitoring. Runtime monitoring consists in observing the internal state of a system during its execution to check whether the system satisfies or violates, on this particular execution, a correctness specification usually provided as a temporal property [13, 17]. In this paper, we focus on a refined category of online monitoring called synchronous monitoring [10]. With synchronous monitoring, the runtime monitor is tightly coupled with the system. More precisely, the execution of the system is suspended whenever the monitor checks, according to past events, whether a received event violates the property or not. Compared to asynchronous monitoring, this allows timely (as opposed to late) detection of property violation or satisfaction, at the cost of performance decrease.

Logging. Logging is a technique that consists in collecting data on the behavior of a system. It is widely used to perform activities such as failure characterization, debugging, security analysis, or performance modeling [23]. While several well-established logging frameworks such as IBM's Common Event Infrastructure or Apache log4x are dedicated to the collection of event logs, the insertion of logging points in the system is left to the developer. Thus, logging points are introduced statically in the resulting system, for example by using Aspect-Oriented Programming or by inserting direct calls to the chosen logging API(s) [11].

Executable DSLs. An executable DSL is composed of an abstract syntax defining the concepts of the considered domain and an execution semantics defining the meaning of these concepts[1]. In this paper, we focus on DSLs where (*i*) the abstract syntax is provided as a metamodel defined using a metamodeling language (*e.g.,* MOF [22] or Ecore [28]), and (*ii*) the execution semantics is provided as an operational semantics (*i.e.,* an interpreter).

The considered operational semantics are those composed of a data structure representing the model state and a set of execution rules altering this model state. The model state is defined in an execution metamodel extending the abstract syntax metamodel using a non-intrusive extension mecha-

[1]We omitted concrete syntax here as it does not impact the approach

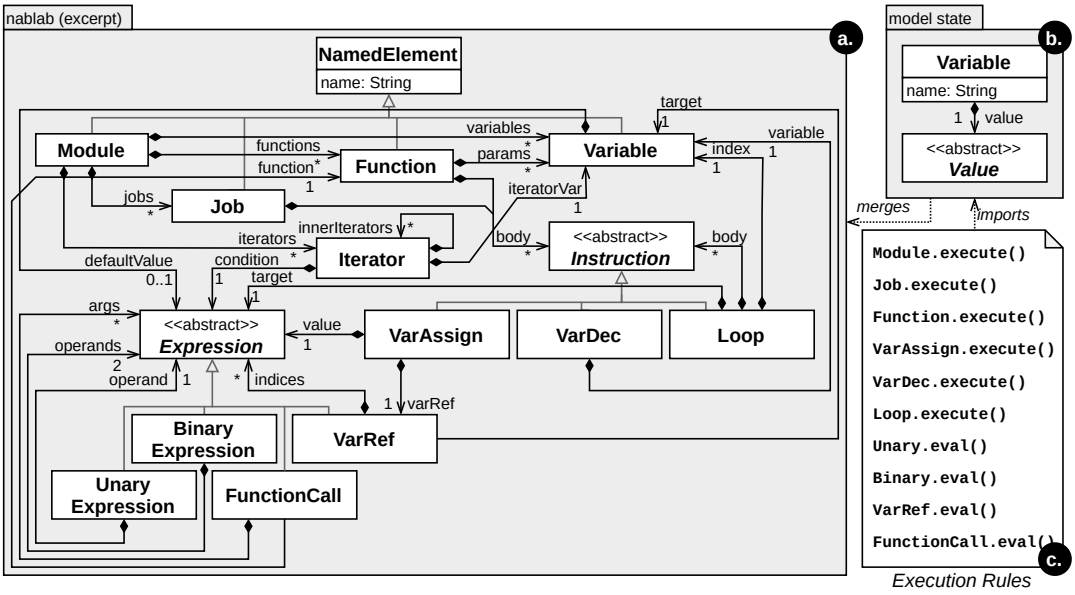


Figure 1. Excerpt of the definition of the NabLab DSL.

```
1 module HeatEquation;
2
3 ℝ t, u{cells}, f{cells}, outgoingFlux{cells};
4 let ℝ δt = 0.001;
5
6 iterate n while (t^{n+1} < stopTime && n+1 < maxIterations);
7
8 ComputeUn: ∀j∈cells(), u^{n+1}{j} = f{j} * δt + u^{n}{j} + outgoingFlux{j};
```

Figure 2. Excerpt of a NabLab Module solving the heat equation.

nism, such as package merge [21] or the open class mechanism [12]. The execution rules then perform an in-place endogenous transformation on this model state. This model transformation effectively results in the execution of the model, and can be implemented by various means (e.g., programming languages, model transformation languages).

Figure 1 shows an excerpt of the definition of NabLab, an open-source executable DSL dedicated to numerical simulation[2], which will be used as a running example throughout this paper. The abstract syntax of NabLab is defined as a metamodel (**a.** in Figure 1), and its operational semantics is composed of a metamodel representing the model state, and a set of execution rules operating on this model state (respectively **b.** and **c.** in Figure 1). A Module contains Variable, Iterator, Job, and Function elements. This is exemplified on Figure 2, which shows an excerpt of a NabLab module solving the heat equation. This module declares five variables (lines 3-4), one iterator (line 6), and a job (line 8) operating on variables defined by the module and the iterators.

Job elements constitute the top-level callable elements, and are called implicitly by their containing Module. The ordering of those calls is delegated to the execution semantics defined in the `Module.execute` rule, and relies on the data flow encompassing all jobs in the module. Job elements contain a sequence of Instruction elements to be executed when the job is called. For the sake of brevity, we only cover here the three main types of instruction of the language: (*i*) VarDec instructions declare a new Variable, (*ii*) VarAssign instructions assign the result of an Expression to a Variable, and (*iii*) Loop instructions iterate through their `index` over an array of values resulting from the evaluation of an Expression. The values manipulated by NabLab programs result from the evaluation of Expression elements, which can be UnaryExpressions (*e.g.,* negation, sign inversion), BinaryExpressions (*e.g.,* arithmetic operations over scalars, vectors and matrices), FunctionCalls with a set of `args`, or VarRefs.

In Figure 2, the `ComputeUn` job iterates over the array of all cells in the discretized spatial domain (*i.e.,* the mesh), which is obtained though a call to the `cells()` function. For each such cell `j`, the loop assigns a value to `u` for the current `n` iteration (denoted by `u^{n+1}{j}`). The assigned value depends on (*i*) the value of `f` for `j`, as denoted by `f{j}`, (*ii*) the value of `u` for `j` on the previous iteration, as denoted by `u^{n}{j}`, (*iii*) the value of `outgoingFlux` for `j`, as denoted by `outgoingFlux{j}`.

[2] https://github.com/cea-hpc/NabLab

2.2 Motivation

Runtime monitoring and logging frameworks are language-specific, being targeted at GPLs. This means that, for frameworks relying on instrumentation directed by external specifications, the developer has to identify – generally through some form of query language (*e.g.,* AspectJ pointcuts) – the locations to be instrumented to send events to runtime monitors [10] or to act as logging points [4, 27]. For frameworks that provide an API to call directly, this is more intrusive as developers have to introduce these calls in their code. In both cases, the mechanisms offered by the framework are dedicated to a specific language. This remains reasonable in the context of GPLs, as (*i*) the costs induced by developing and maintaining a framework relying on instrumentation are offset by the number of potential users, and (*ii*) GPLs offer enough expressiveness to provide a framework relying on a set of APIs to be directly called by developers. However, in the context of DSLs, several obstacles arise that hamper the provision of such frameworks.

Indeed, runtime monitoring and logging must be performed at the domain level to be usable by modelers with no technical background in computer science. For frameworks relying on instrumentation, this means adapting them to the syntactic constructs of each new DSL to be supported, which is tedious and error-prone. Frameworks relying on direct calls to their APIs cannot be used with DSLs without ignoring a few principles of MDE for two main reasons. First, enabling such external API calls may require to introduce syntactic constructs that are not part of the domain of the DSL. Second, this hampers proper separation of concerns, as runtime monitoring and logging concerns become intertwined with the domain concerns addressed by the model.

For example, in the case of NabLab, output operations are designed for production use, not debugging. Consequently, these operations allow to output the content of specific variables (*e.g.,* u in Figure 2), at a specific frequency, both specified in the configuration file of a NabLab application. Thus, to implement conditional logging of a variable, it would be possible to modify the model to compute an artificial variable and output its content under certain circumstances, but this would require to alter the original model and application configuration. Moreover, this workaround (*i*) does not allow to output intelligible logging messages that help understanding the underlying defects in the model, and (*ii*) might not be feasible altogether, if the expressivity of the language does not allow to derive the data required for debugging.

In addition, even when frameworks are available for a given GPL, runtime monitoring and logging are often used separately, each technique instrumenting a system independently. Yet, both techniques can benefit from being integrated together. For example, in NabLab, runtime monitoring can be used to monitor the trend of specific (possibly derived) variables, and produce logs displaying the values of those variables whenever they fail to follow the expected trend. However, while the techniques are mutually beneficial, existing frameworks for runtime monitoring and for logging do not interact with one another out of the box. Realizing the integration of such frameworks in the context of GPLs mostly means forgoing some of the benefits of runtime monitoring and logging frameworks. In the context of executable DSLs, it also means working at the implementation level of the DSL, when the language permits it (*e.g.,* black-box language interpreters typically prevent this).

In summary, existing frameworks for runtime monitoring or logging fulfill their purpose because they are dedicated to GPLs, which have the required expressivity to both define and use them. In the case of DSLs, where this expressivity cannot be assumed, there is a need for language-agnostic approaches enabling both logging and runtime monitoring at the domain level. Furthermore, as runtime monitoring and logging often interact with each other, such an approach needs to unify both techniques for modelers to benefit from these interactions while keeping the domain abstractions brought by DSLs. In the next section, we provide an overview of our proposed approach to tackle these challenges.

3 Overview

Figure 3 provides an overview of the proposed approach for monilogging, and supported by our MoniLog language and its associated framework. On the left part, the figure shows in grey the hypotheses of the proposed approach. We consider executable DSLs that are composed of an abstract syntax allowing to define models, and an operational semantics. More precisely, we restrict the scope of the proposed approach to languages whose operational semantics is defined according to the visitor or the interpreter design patterns. From such a definition of an executable DSL, an interpreter is generated to run on top of one or several specific execution platforms (*e.g.,* GraalVM or the HotSpot JVM), and support the execution of the conforming models. We also assume that the execution platform supports dynamic pointcut mechanisms (*e.g.,* Truffle for GraalVM, AspectJ for any JVM), which is depicted on the middle lower part of Figure 3 by the depends on relation between the dynamic pointcut implementation and the execution platform.

On the middle upper part, Figure 3 shows in orange an *instrumentation interface* defined by language engineers for a specific executable DSL. This instrumentation interface bridges the gap between the DSL and the MoniLog language, both at design time and at runtime. At design time, it allows MoniLog users to define moniloggers for models conforming to the DSL that the interface complements. At runtime, this interface is used to instrument the interpreter executing those models with the defined moniloggers, through the generation of a dynamic pointcut implementation.

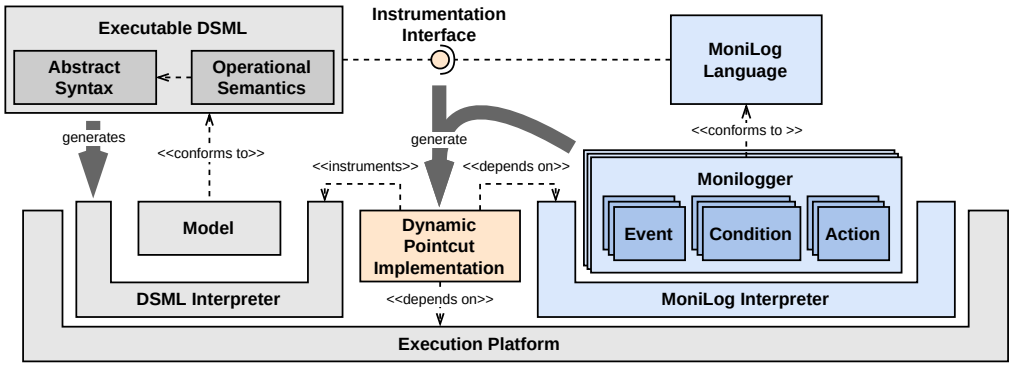


Figure 3. Overview of the MoniLog language and associated framework.

On the right part, Figure 3 shows in blue the MoniLog language and associated framework constituting the contribution of this paper. The MoniLog language allows to define moniloggers, *i.e.,* runtime monitors, loggers and combinations of both. Moniloggers are mapped to ECA (Event-Condition-Action) rules, which have been used to model reactive behavior in various domains [3, 7, 24, 32].

Intuitively, this mapping is as follows. *Events* can either be (*i*) execution events exposed by the instrumentation interface, *i.e.,* starting or returning calls to rules of the execution semantics on specific model elements, or (*ii*) user-defined events emitted by other moniloggers. *Conditions* are boolean expressions referring to the execution state of either the running model (through the instrumentation interface) or the instrumentation context, and are written in the expression language embedded with MoniLog, or in any language supported by the platform on which the model is being executed. *Actions* mostly consist of calls to standard or user-defined appenders sending messages to their configured destination according to a given layout, or emissions of user-defined events that can carry data derived from the execution state and can trigger additional moniloggers. Similarly to conditions, actions can contain expressions referring to the execution state – through the instrumentation interface – and to the instrumentation context.

At runtime, the MoniLog framework then allows to use moniloggers with models conforming to any DSL satisfying the prerequisites of the approach. Moniloggers are managed by the MoniLog interpreter, which runs on the same execution platform as the DSL interpreter. As moniloggers follow the ECA paradigm, the MoniLog interpreter is a kind of ECA engine that triggers its ECA rules (*i.e.,* its registered moniloggers) in reaction to events, and that is able to query the execution state of the running model to resolve the condition and action parts of the triggered moniloggers.

The events received by the MoniLog interpreter originate from two sources: the dynamic pointcut implementation (shown in orange on the middle lower part of Figure 3), and actions of triggered moniloggers. The dynamic pointcut implementation is generated from the instrumentation interface of the DSL and the specifications of the moniloggers to be applied to the execution, as shown on Figure 3. This allows to insert calls to the MoniLog interpreter before and after the execution by the DSL interpreter of the rules exposed as execution events by the DSL instrumentation interface. In practice, the actual instrumentation framework for which dynamic instrumentation specifications are generated depends on the execution platform. For example, in the case of GraalVM as an execution platform, the Truffle instrumentation framework can be used for any language defined using the Truffle language implementation framework, while in the case of non-Truffle-based languages (or other JVMs), Aspect-Oriented Programming (AOP) can be used to perform the instrumentation, *e.g.,* using AspectJ.

4 MoniLog Language

In this section, we present the MoniLog language in more detail. In particular, we detail the metamodel constituting the abstract syntax of MoniLog, and exemplify it on an example MoniLog specification.

Figure 4 provides this aforementioned example, while Figure 5 highlights different parts of the metamodel of the MoniLog language. Figure 5a shows an overview of the language. At the root of conforming models is a MoniLogSpecification element, which contains Monilogger, Event, Layout, and Appender elements, which can all be reused across moniloggers and MoniLog specification files. All of these are ParameterizedElements: they can declare a list of parameters for which a value must be supplied when they are called, and to which expressions may refer. A Monilogger is split in three parts: the `events`, the `conditions`, and the `actions` parts. Figure 5b, 5c, and 5d cover these parts in more details.

The event part of a Monilogger points to an Event contained in a MoniLogSpecification. Event elements can all specify a set of parameters, as denoted by their extending the ParameterizedElement class. As shown in Figure 5b, Event elements can either be ExecutionEvent or UserEvent elements. An ExecutionEvent refers to an ExposedExecutionEvent defined in the instrumentation interface of the DSL to which the

model to instrument conforms. An ExposedExecutionEvent specifies an ExecutionRule from the operational semantics of the DSL that, when called with a (possibly empty) set of specific model elements (denoted as Objects conforming to Concepts of the DSL), generates two ExecutionEvents: one before and one after the call to the execution rule. For example, the `ComputeUnReturned` event defined in Figure 4 refers to an instance of ExposedExecutionEvent named `ComputeUn`, which itself refers to the `ComputeUn` Job shown in Figure 2, and points to the `Job.execute` rule defined in the execution semantics of NabLab shown in Figure 1. Finally, an UserEvent is an event whose occurrences are directly emitted by a monilogger, as part of an EmitEvent action (see below). UserEvents are typically used when multiple ExecutionEvents can trigger a given monilogger.

The condition part of a Monilogger allows users to specify under which conditions a monilogger should perform its associated Actions when triggered by a given Event occurrence. The condition part consists in an Expression whose result is interpreted as a boolean. As shown on Figure 5c, an Expression can be a BinaryExpression, a UnaryExpression, an AtomicExpression, or a VariableAssignment. The first two allow to define arithmetic and boolean expressions primarily based on AtomicExpressions, while VariableAssigments set the `target` Variable to the result of the `value` Expression.

Aside from usual elements such as primitive values (not detailed here for the sake of brevity), AtomicExpression elements can be VariableReferences, ContextReferences, LanguageCalls, or LayoutCalls. VariableReference elements point to a declared Variable, which can be a global variable, a parameter of the current monilogger, a parameter of the triggering event of the monilogger, or one of its local variables. ContextReference elements point to ExposedObject elements provided by the instrumentation interface of the DSL. ExposedObjects in turn point to an element of the instrumented model (an Object) conforming to a Concept of the DSL). Thus, where VariableReferences allow to refer to variables specific to the state of the MoniLog interpreter, ContextReferences allow to refer to the state of the running model. In Figure 4, the condition part of the `LogTemperature` monilogger (line 17) contains two VariableReferences (`lastTimeLogged` and `outputInterval`, which are global variables declared at line 9), and a ContextReference (**`context`**`(t_n)`), which refers to the `t_n` variable (*i.e.,* the simulation time at the previous iteration over `n`) in the state of the instrumented model. This condition is used to log values at a period differing from that of the simulation.

LanguageCall elements perform a call to a `callable` element of another model from a potentially different language, with a set of VariableValue elements provided by the `args` reference to supply values for the parameters of `callable` element. In Figure 4, such a call to another language is performed on line 23. This call is performed on the `stdev` function defined in the JavaScript file imported under the `utils`

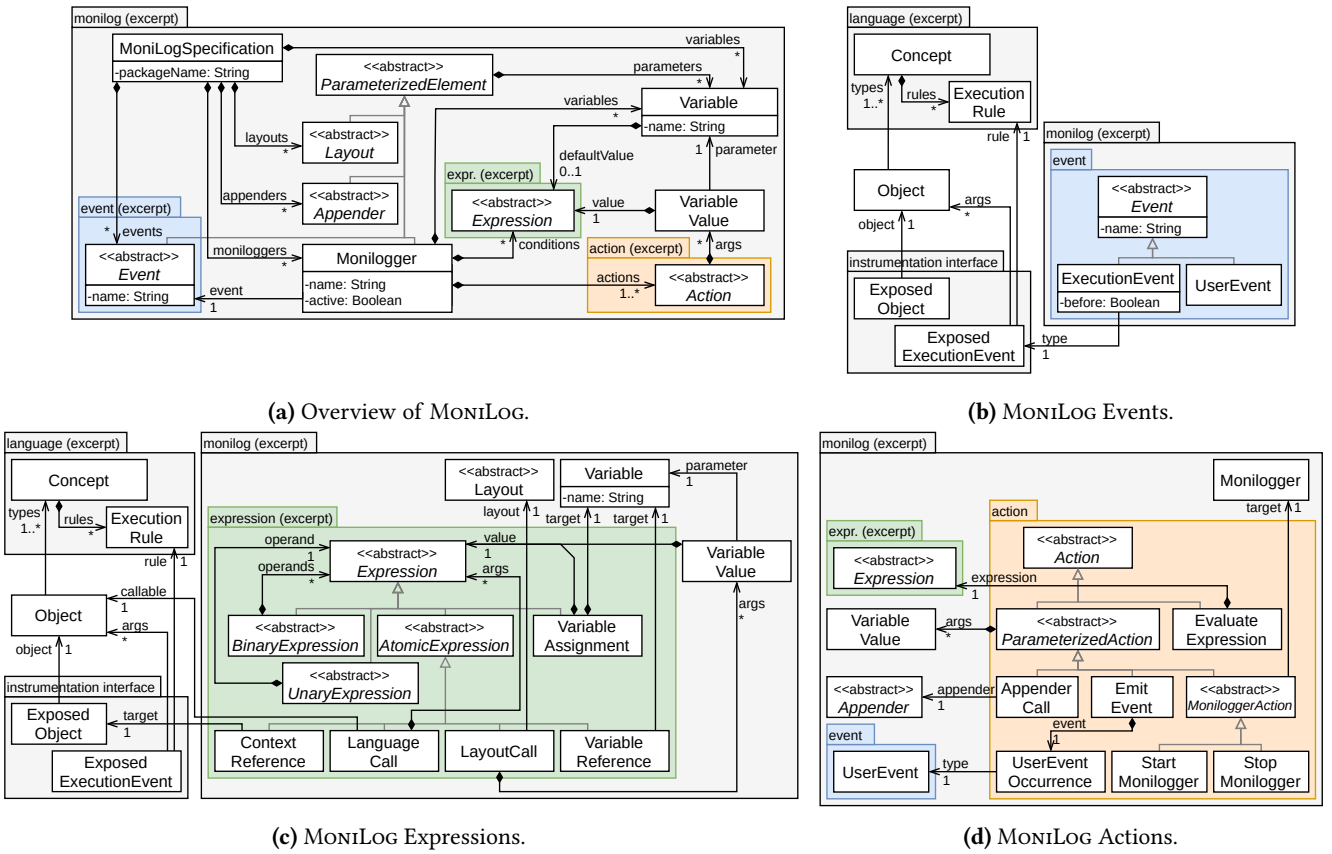
```
1  package heatequation
2
3  import org.gemoc.monilog.stl.*
4  import HeatEquation.*
5
6  import js("$MONILOG_HOME/Utils.js") as utils
7
8  setup {
9      lastTimeLogged = 0.0; outputInterval = 0.0001;
10     filePath = "u_logs.csv";
11 }
12
13 event ComputeUnReturned { after call ComputeUn }
14
15 monilogger LogTemperature {
16     when ComputeUnReturned
17     if (lastTimeLogged + outputInterval <= context(t_n))
18     then {
19         FileAppender.call(
20             StringLayout.call(
21                 "{0,number,0.0000}, {1,number,0.000E0}",
22                 context(t_n),
23                 utils.stdev(context(u_n))),
24             filePath);
25         lastTimeLogged = lastTimeLogged + outputInterval;
26     }
27 }
28
29 monilogger CoarsenInterval {
30     when ComputeUnReturned
31     if (context(t_n) >= 0.01)
32     then {
33         outputInterval = 0.01;
34         CoarsenInterval.stop;
35     }
36 }
```

Figure 4. Example MoniLog specification file.

alias in the specification, by providing the current value of the `u_n` variable in the model state (*i.e.,* the array of temperature values over the simulated domain at the current iteration of the Iterator over `n`). Finally, we describe LayoutCall elements below, as they are more relevantly used in the action part of a monilogger than in its condition part.

The action part of a Monilogger allows to specify the actions to be undertaken by the monilogger upon occurrences of its observed event, if its associated conditions is validated. As shown on Figure 5d, the MoniLog language allows to (*i*) evaluate an Expression through an EvaluateExpression element, (*ii*) deliver data at a given destination through AppenderCall elements, (*iii*) emit occurrences of a UserEvent, which may trigger other moniloggers, or (*iv*) start or stop Moniloggers. In particular, the last three may require to supply arguments under the form of VariableValues.

MoniLog comes with a standard library of Appenders (console and file appenders), but can be extended to provide additional ones suited to the needs of modelers, such as appending data to a database. AppenderCall elements expect a message to append, but can also take additional parameters such as a file path in the case of the file appender. The specific message to append is typically provided through a



(a) Overview of MoniLog. **(b)** MoniLog Events.

(c) MoniLog Expressions. **(d)** MoniLog Actions.

Figure 5. Metamodel of the MoniLog language.

LayoutCall pointing to a Layout element to obtain a message formatted in a specific way (*e.g.*, according to specific XML or JSON schemes) from a set of VariableValue elements. Similarly to Appenders, the MoniLog language comes with a standard library of Layout (the string formatting layout), but the language can be extended to provide additional Layout types, better suited to the domain of a DSL (*e.g.*, formatting data for inspection in visualization software such as ParaView [2]). In Figure 4, the `LogTemperature` monilogger uses the `FileAppender` and `StringLayout` to perform its main action. This action consists in computing the standard deviation of the `u_n` variable from the model state, and outputting a line containing the current simulation time and the computed standard deviation to a file at a path provided by the `filePath` variable. The second action performed by the monilogger is to increment the `lastTimeLogged` variable through an EvaluateExpression.

The last type of action provided by MoniLog is starting and stopping moniloggers, available as StartMonilogger and StopMonilogger elements. Both of these kinds of action point to a particular Monilogger on which to perform the action. StartMonilogger actions can also provide a set of ParameterValue elements to pass arguments to the monilogger, which will then be available for reference in the expressions it contains. In Figure 4, the `CoarsenInterval` monilogger, which is in charge of increasing the interval of simulation time between each log once the simulation time has reached `0.01`, uses the StopMonilogger action on itself to stop monitoring the `t_n` variable after it has been triggered once.

5 Formal Semantics

In this section we introduce the execution semantics for MoniLog, formalizing the possible composition of loggers and runtime monitors, as well as the coordinated execution of several moniloggers, in concert with the running model.

5.1 General Definitions

In the remainder of this section, we abstract the running model as a labeled transition system (LTS) [6]. We denote the silent event with τ, and any ExecutionEvent with λ, which we refer to as execution events in the remainder of this Section. We also denote Λ the set of all execution events. We denote with $s \xrightarrow{\lambda} s'$ the fact that there is an execution rule in the operational semantics of the DSL allowing to transition from state s to state s' by producing an execution event λ, *i.e.,* there is a transition labelled λ between s and s' in the LTS of the running model. Similarly, $s \xrightarrow{\tau} s'$ denotes a silent transition

$$\textsc{Sil}\ \frac{s \xrightarrow{\tau} s'}{\langle \mathbf{mon}, as, \sigma\rangle_m \lhd s \xrightarrow{\tau} \langle \mathbf{mon}, as, \sigma\rangle_m \lhd s'} \qquad \textsc{Mon}\ \frac{s \xrightarrow{\lambda} s' \quad \text{event}(m) \neq \lambda}{\langle \mathbf{mon}, \varepsilon, \sigma\rangle_m \lhd s \xrightarrow{\lambda} \langle \mathbf{mon}, \varepsilon, \sigma\rangle_m \lhd s'} \qquad \textsc{Trig}\ \frac{s \xrightarrow{\lambda} s' \quad \text{event}(m) = \lambda}{\langle \mathbf{mon}, \varepsilon, \sigma\rangle_m \lhd s \xrightarrow{\lambda} \langle \mathbf{trig}, \varepsilon, \sigma\rangle_m \lhd s'}$$

$$\textsc{CondF}\ \frac{\langle \text{condition}(m), \sigma, s\rangle \Downarrow \{\sigma', \mathbf{ff}\}}{\langle \mathbf{trig}, \varepsilon, \sigma\rangle_m \lhd s \xrightarrow{\tau} \langle \mathbf{mon}, \varepsilon, \sigma'\rangle_m \lhd s} \qquad \textsc{CondT}\ \frac{\langle \text{condition}(m), \sigma, s\rangle \Downarrow \{\sigma', \mathbf{tt}\} \quad \text{actions}(m) = as}{\langle \mathbf{trig}, \varepsilon, \sigma\rangle_m \lhd s \xrightarrow{\tau} \langle \mathbf{exec}, as, \sigma'\rangle_m \lhd s} \qquad \textsc{Restart}\ \frac{}{\langle \mathbf{exec}, \varepsilon, \sigma\rangle_m \lhd s \xrightarrow{\tau} \langle \mathbf{mon}, \varepsilon, \sigma\rangle_m \lhd s}$$

$$\textsc{Log}\ \frac{a \notin \{\mathbf{stop}, \mathbf{start}, \mathbf{emit}\} \quad \langle a, \sigma, s\rangle \Downarrow \{\sigma', \pi\}}{\langle \mathbf{exec}, a :: as, \sigma\rangle_m \lhd s \xrightarrow{\pi} \langle \mathbf{exec}, as, \sigma'\rangle_m \lhd s} \qquad \textsc{Stop}\ \frac{}{\langle \mathbf{exec}, \mathbf{stop}(m) :: as, \sigma\rangle_m \lhd s \xrightarrow{\tau} s}$$

Figure 6. Inference rules of the structural operational semantics of MoniLog (single monilogger)

$$\textsc{SilSC}\ \frac{m_1 \lhd s \xrightarrow{\tau} m_1 \lhd s' \quad m_2 \lhd s \xrightarrow{\tau} m_2 \lhd s'}{m_1 \cdot m_2 \lhd s \xrightarrow{\tau} m_1 \cdot m_2 \lhd s'} \qquad \textsc{SilMC}\ \frac{m_2 = \langle e, as, \sigma\rangle \quad e \neq \mathbf{exec} \quad m_1 \lhd s \xrightarrow{\tau} m_1' \lhd s}{m_1 \cdot m_2 \lhd s \xrightarrow{\tau} m_1' \cdot m_2 \lhd s} \qquad \textsc{MonC}\ \frac{m_1 \lhd s \xrightarrow{\lambda} m_1' \lhd s' \quad m_2 \lhd s \xrightarrow{\lambda} m_2' \lhd s'}{m_1 \cdot m_2 \lhd s \xrightarrow{\lambda} m_1' \cdot m_2' \lhd s'}$$

$$\textsc{LogC}\ \frac{m_1 \lhd s \xrightarrow{\pi} m_1' \lhd s}{m_1 \cdot m_2 \lhd s \xrightarrow{\pi} m_1' \cdot m_2 \lhd s} \qquad \textsc{EmitC}\ \frac{\langle a, \sigma_1, s\rangle \Downarrow \{\sigma_1', \mu\} \quad \text{event}(m_2) = \mu}{\langle \mathbf{exec}, \mathbf{emit}(a) :: as, \sigma_1\rangle_{m_1} \cdot \langle \mathbf{mon}, \varepsilon, \sigma_2\rangle_{m_2} \lhd s \xrightarrow{\tau} \langle \mathbf{exec}, as, \sigma_1'\rangle_{m_1} \cdot \langle \mathbf{trig}, \varepsilon, \sigma_2\rangle_{m_2} \lhd s}$$

$$\textsc{StartC}\ \frac{\langle a, \sigma_1, s\rangle \Downarrow \sigma_2}{\langle \mathbf{exec}, \mathbf{start}(m_2, a) :: as, \sigma_1\rangle_{m_1} \lhd s \xrightarrow{\tau} \langle \mathbf{exec}, as, \sigma_1\rangle_{m_1} \cdot \langle \mathbf{mon}, \varepsilon, \sigma_2\rangle_{m_2} \lhd s} \qquad \textsc{StopC}\ \frac{}{\langle \mathbf{exec}, \mathbf{stop}(m_2) :: as, \sigma\rangle_{m_1} \cdot m_2 \lhd s \xrightarrow{\tau} \langle \mathbf{exec}, as, \sigma\rangle_{m_1} \lhd s}$$

Figure 7. Inference rules of the structural operational semantics of MoniLog (composed moniloggers)

from s to s', resulting from calls to execution rules that are not exposed by the instrumentation interface, and thus do not produce an executable event.

Concerning moniloggers, we denote any log event resulting from an AppenderCall with π, and any event emitted by the EmitEvent action with μ. The status of a monilogger can be **mon**, *i.e.,* the monilogger is waiting for a triggering event, **trig**, *i.e.,* the monilogger has been triggered and will evaluate its condition, or **exec**, *i.e.,* the monilogger is currently evaluating its condition or executing its actions. Given a monilogger m, event(m) returns the event monitored by m, condition(m) returns the condition of m, and actions(m) returns the ::-constructed list of actions of m. We denote the condition to be evaluated with c. We denote the action to be executed – *i.e.,* the head of the list – with a, and the actions that remain to be executed with as. ε denotes the empty list. We denote the context of a monilogger with σ. Finally, we denote the state of a monilogger as a tuple $m = \langle e, as, \sigma\rangle$ (also denoted as $\langle e, as, \sigma\rangle_m$), with $e \in \{\mathbf{mon}, \mathbf{trig}, \mathbf{exec}\}$.

The evaluation of expressions and calls to external models constituting conditions and actions of moniloggers are abstracted as follows. For a given executable model element *elt*, monilogger context σ, and execution state s, we denote $\langle elt, \sigma, s\rangle \Downarrow \{\sigma', res\}$ the fact that *elt* evaluates to *res* with resulting monilogger context σ'. In the case of conditions, $res \in \{\mathbf{tt}, \mathbf{ff}\}$, denoting a satisfied or unsatisfied condition. Concerning actions, **start**, **stop**, and **emit** denote StartMonilogger, StopMonilogger, and EmitEvent elements, while a denotes AppenderCall or EvaluateExpression elements.

Finally, we introduce the $\lhd$ instrumentation relation between a system and a monilogger. This relation is based on the instrumentation relation described by Francalanza *et al.* in [13]. We denote with $m \lhd s$ the fact that a model in current state s is instrumented with a monilogger in current state m. Figure 6 and Figure 7 present in details the structural operational semantics of the instrumentation of a running model with one (for Figure 6) or several moniloggers (for Figure 7). We present both case in the remainder of this section, starting with instrumentation with a single monilogger.

5.2 Single Monilogger Composition

According to Sil, the running model can only silently transition to a new state if the monilogger is in the **mon** state. The Mon rule states that the running model can only transition non-silently alone to a new state if the label of the transition does not match the triggering event of the monilogger. Conversely, the Trig rule states that when the label of the transition does match the triggering event of the monilogger, the running model and the monilogger transition together to a new state, where the status of the monilogger becomes **trig**. These three rules enforce the synchronous nature of the instrumentation: execution of the running model is suspended while the monilogger reacts to a triggering event.

Once in a **trig** state, a monilogger can evaluate its condition. If the condition evaluates to **ff** (CondF), the monilogger silently transitions to a new state where it resumes monitoring events. Not that any side effect resulting from the evaluation of the condition is preserved. If the condition evaluates to **tt** (CondT), the monilogger silently transitions to an **exec** state containing the list of actions to execute *as*.

Once in an **exec** state, the Log rules covers the execution of actions other than **stop**, **start**, and **emit**. This rule consists in evaluating the action, which can yield a log event π (for calls to appenders), and an updated instrumentation context

σ' (for expressions to evaluate). The RESTART rule allows a monilogger to return to a **mon** state when it has no action left to execute. The STOP rule covers the **stop** action, and simply terminates the instrumentation of the running model, letting the execution of the running model run to completion. As the **start** and **emit** actions have no use in the context of a single monilogger, they have no associated rule in Figure 6.

5.3 Multiple Monilogger Composition

For the sake of brevity, we only show the instrumentation of a system with two moniloggers, but it can be extended to 3 or more moniloggers. We hereby introduce the $\cdot$ symmetrical and associative relation between monilogger states, which is defined through the rules in Figure 7.

The first rule shown in Figure 7, SILC, states that the $m_1 \cdot m_2 \triangleleft s$ system resulting from the composition can only silently transition to a new model state if each individual monilogger-model system can also silently transition to that same new model state, *i.e.,* SIL can be individually applied to each. The second rule, MONC, states that $m_1 \cdot m_2 \triangleleft s$ can only transition to a new model state s' with observable event λ if *both* m_1 and m_2 can also transition to a new state with λ, either through the MON or through the TRIG rule.

The SILMC rule allows a monilogger to silently transition to a new state if no other monilogger is in an **exec** state. This means that once a monilogger is in an **exec** state, it is the only one on which rules can be applied. In other terms, once the condition of a monilogger has been validated, its actions must be completely executed before another potentially triggered monilogger starts executing. The running model can only resume executing once all moniloggers have either transitioned back to a **mon** state, or have been stopped.

The remaining rules handle each kind of action in the context of multiple moniloggers. The LOGC rule allows a monilogger to execute their AppenderCall or EvaluateExpression actions. The EMITC rule defines the semantics of the EmitEvent action, which emits a user-defined event μ, thereby allowing other moniloggers to transition from **mon** states to **trig** states. The STARTC rule states that the StartMonilogger action initializes a new monilogger with context σ_2 resulting from the evaluation of arguments a provided to the action. This new monilogger is composed with the currently executing one. Finally, according to the STOP rule, when a StopMonilogger action targets another monilogger, it is simply removed from system.

6 Implementation

The abstract syntax of MONILOG is implemented as an Ecore metamodel, and its concrete syntax is provided as an Xtext grammar. The MONILOG interpreter is provided under two implementations: a Truffle-based implementation leveraging the Truffle instrumentation framework [29], and a pure Java implementation relying on AspectJ [14]. The former can run on any JVM, while the latter runs on GraalVM. Both implementations are available on GitHub[3].

To be used with models conforming to a given DSL, both implementations rely on an instrumentation interface provided by language engineers. The instrumentation interface of an executable DSL consists of (*i*) a model transformation creating the design-time, model-specific instrumentation interface which exposes model elements that can be read or instrumented under the form of ExposedObjects, and to which users can refer when defining moniloggers, (*ii*) a context wrapper used by the MONILOG interpreter and providing the value of each model element exposed as an ExposedObject to evaluate ContextReferences, and (*iii*) a pointcut for execution rules exposed as ExposedExecutionEvents, to enable the emission of ExecutionEvents at runtime.

The model transformation providing the model-specific instrumentation interface depends on the formalism used to define the abstract syntax of the DSL. In the case of NABLAB, which uses Ecore, the model transformation was defined in Xtend, and consists in populating the model-specific instrumentation interface with ExposedObjects mapped to every Job, Function, and Variable elements found in the model.

In the case of Truffle, the context wrapper must follow the Truffle interoperability protocol, allowing any Truffle-based tool and language implementation to read from and write to the execution context of the language. This mostly means that the different types manipulated by the DSL should implement the Truffle interoperability protocol, which is also a prerequisite to benefit from other Truffle-based tooling and from interoperability between Truffle-based languages. In the case of AspectJ, the context wrapper must be able to retrieve the primitive value of an ExposedObject, if it represents such a value (*e.g.,* a real matrix in NABLAB is converted to a 2-dimensional `double` array). If a type cannot be mapped to primitive values (*i.e.,* if it is an object value), the context wrapper must instead provide a map providing the value of each property that can be read on the object value.

Concerning the execution rule pointcuts, when using Truffle, these must be provided as part of the definition of the language. This is because the Truffle Language Implementation Framework offers constructs for defining a kind of instrumentation interface addressing runtime instrumentation concerns, *i.e.,* identifying AST nodes to instrument. Thus, the AST nodes that the language engineer wishes to make instrumentable must be defined as such, which boils down to implementing a specific (Java) interface for them. While this is intrusive with regard to the operational semantics of the DSL, this allows to identify which AST nodes to instrument among nodes corresponding to the same execution rule. For example, in NABLAB, this allows to instrument only the AST nodes corresponding to a given job (*e.g.,* `ComputeUn`) among all AST nodes corresponding to the `Job.execute` rule.

[3] https://github.com/cea-hpc/Monilog4NabLab

In AspectJ, the runtime elements of the instrumentation interface can be provided *a-posteriori*, in a non-intrusive way with regard to the operational semantics, through a collection of aspects instrumenting the Java code of the language interpreter. In the case of AspectJ, pointcuts target the Java methods implementing the execution rules to instrument. As such, AspectJ pointcuts are called every time an execution rule is called, even if the model element on which it is called should not be instrumented. For example, in NabLab, the pointcut associated to assignments is called on each call to the `VarAssign.execute` rule, even if the actual Variable being assigned a value is not the target of any instrumentation.

7 Evaluation

In this section, we leverage the previously described implementations to answer the following research questions:

RQ#1 How does the proposed approach allow to combine runtime monitoring and logging to extract relevant data from running models?

RQ#2 How is the overhead induced by the approach affected by different scenarios?

7.1 Leveraging Complementarities

To evaluate whether MoniLog allows to leverage the complementarities between runtime monitoring and logging, we present a demonstration case on a NabLab model solving the heat equation. In this demonstration case, we show how to use MoniLog to *a*) compute the standard deviation of the temperature over the simulated domain, *b*) monitor that it decreases at each simulation step, and *c*) start a logger appending temperature values to a file in case the standard deviation fails to decrease on a simulation step. The combination of *a*) and *b*) highlights how logging can be leveraged by runtime monitoring, and the combination of *b*) and *c*) highlights the reverse. Figure 8 shows the corresponding MoniLog specification. We cover its key points below.

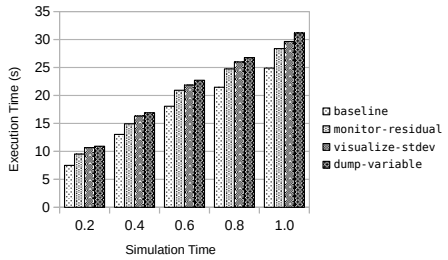
As the expressivity of MoniLog does not allow to easily compute the standard deviation of an array, the specification first imports a JavaScript file named `Utils.js` (line 6), which contains an `stdev` function, as well as a `date` function. Next, a `setup` section (lines 8-11) declares variables that can be used in the instrumentation context: `previousStdev` holds the value of the previously computed standard deviation and is initialized with an upper bound value of `1.0`, `currentStdev` holds the current standard deviation, and `filePath` is used to indicate in which file should be logged, using the aforementioned `date` function from the imported JavaScript file. Then, the `ComputeUnReturned` event is declared (line 13): it is emitted after each call to the `ComputeUn` callable element (a Job in this case). The specification then contains two monioggers: one computing and monitoring the standard deviation of the `u_n` array (*i.e.,* the temperature computed on the current iteration), and another which starts as inactive and logs the content of `u_n` to a file once it is activated.

```
1  package heatequation
2
3  import org.gemoc.monilog.stl.*
4  import HeatEquation.*
5
6  import js("$MONILOG_HOME/Utils.js") as utils
7
8  setup {
9      previousStdev = 1.0; currentStdev;
10     filePath = "u_logs" + utils.date() + ".csv";
11 }
12
13 event ComputeUnReturned { after call ComputeUn }
14
15 monilogger MonitorStdev {
16     when ComputeUnReturned
17     if ((currentStdev = utils.stdev(context(u_n))
18         > previousStdev)
19     then {
20         ConsoleAppender.call(
21             StringLayout.call("Standard deviation did not
       decrease, starting logging 'u' to {0}",
22                 filePath));
23         FileAppender.call(CSVLayout.call(context(u_n)),
24             filePath);
25         MonitorStdev.stop;
26         LogU.start;
27     } else { previousStdev = currentStdev; }
28 }
29
30 @Inactive
31 monilogger LogU {
32     when ComputeUnReturned {
33         FileAppender.call(CSVLayout.call(context(u_n)),
34             filePath);
35     }
36 }
```

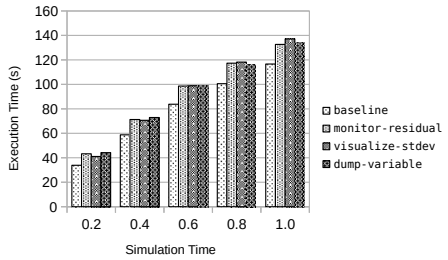
Figure 8. Monilogging temperature dispersion.

The first monilogger, `MonitorStdev`, is triggered by occurrences of the `ComputeUnReturned` event (line 15). When triggered, it computes the standard deviation of the `u_n` array using the `stdev` functions from the imported JavaScript file, assigns it to the `currentStdev` variable, and compares it to the `previousStdev` variable (lines 17-18). Then, if the comparison violates the expected behavior of the model, a message is printed to the console using calls to `StringLayout` and `ConsoleAppender` (lines 20-22), and a line is appended to the output CSV file using calls to `CSVLayout` and `FileAppender` (line 23-24). Finally, the `MonitorStdev` monilogger is stopped, and the `LogU` monilogger is activated (lines 25-26). If however the model behaves as expected, `previousStdev` is set to `currentStdev` instead (line 27). The second monilogger, `LogU`, is initially inactive, as indicated by the `@Inactive` annotation (line 30). Once started, it is triggered by occurrences of `ComputeUnReturned`, and appends the content of `u_n` to the output CSV file (lines 33-34).

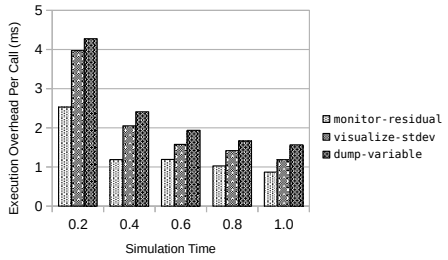
Answering RQ#1. Overall, the specification shown in Figure 8 demonstrates how the language constructs provided by



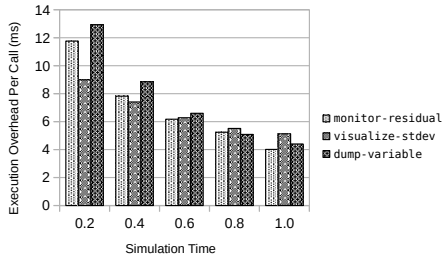
(a) Total execution time (Truffle).



(b) Total execution time (AspectJ).



(c) Execution overhead per call (Truffle).



(d) Execution overhead per call (AspectJ).

Figure 9. Quantitative evaluation results.

MoniLog allow to combine freely logging and runtime monitoring concepts, thereby allowing to leverage the complementarities between the two. This specification also shows how language calls allow to provide the required expressivity for some computations. The MoniLog specification shown in Figure 4 also leverage these complementarities, as it alters its logging behavior (*i.e.,* the frequency of log events) according to the state of a property defined over the execution state of the running model.

7.2 Induced Execution Overhead

To answer RQ#2, we consider 3 MoniLog specifications, running on both implementations (Truffle and AspectJ) and instrumenting the same NabLab model, each designed to handle a different use-case, as detailed thereafter.

- `monitor-residual`: Verifying a temporal property on a variable. This specification uses MoniLog variables to evaluate properties over several execution states.
- `visualize-stdev`: Feeding data derived from the execution state to an external visualization. This specification derives data through language calls to JavaScript, which offers the required expressivity, and sends this data to an asynchronous visualizer.
- `dump-variable`: Writing data of interest (*i.e.,* the content of the `u_n` variable) to a file for offline analysis. This specification outputs large amounts of data (around 25 MB for every 0.2 of simulation time).

Experimental Setup and Protocol. Measurements are performed on an Intel® Core™ i7-9850H CPU @ 2.60GHz × 12, running on Ubuntu 20.04.2, and using GraalVM 21.1.0 as the execution platform. The execution overhead of each specification is measured on the Truffle and AspectJ implementations as follows. First, a warm-up phase consisting of 5 executions of the instrumented model takes place. We then collect the execution time of 50 additional executions, and compute the average execution time using geometric mean. We do the same for the average execution time of the model without instrumentation, which we use as a baseline to compute the absolute and relative overhead of each specification. These measurements are done for 5 different simulation time values, ranging from 0.2 to 1.0, allowing to evaluate the induced overhead on a range of execution times. Summaries of the measurements are shown in Figure 9.

Answering RQ#2. As shown on Figure 9a, for the Truffle implementation of the NabLab interpreter, the baseline execution time of the model ranges from 7.49s to 24.90s, for simulation times ranging from 0.2 to 1.0. The execution time of the instrumented model respectively ranges from 9.52–10.91s to 28.37–31.16s, for an absolute overhead ranging from 2.02–3.42s to 3.47–6.26s, and for a relative overhead ranging from ×1.27–1.46 to ×1.14–1.25. Concerning the AspectJ implementation, Figure 9b shows that the baseline execution time ranges from 33.84s to 116.62s, while the execution time of the instrumented model ranges from 41.03–44.18s to 132.66–137.12s. The absolute overhead thus ranges from 7.19–10.34s to 16.05–20.51s, and the relative overhead ranges from ×1.21–1.31 to ×1.14–1.18.

Figures 9c and 9d show for each scenario the *overhead per call* for the Truffle implementation (resp. the AspectJ implementation), which we define as the absolute overhead divided by the number of times a monilogger is triggered. For the `dump-variable` and `visualize-stdev` scenarios, overhead per call is significantly higher than for the `monitor-residual` scenario, which stays relatively stable at less than 0.5ms (resp. around 2.5ms). This is because both scenarios

derive data from the `u_n` array, which induces significant additional overhead with regard to `monitor-residual`. The fact that `monitor-residual` is within the same order of magnitude than the other scenarios (as shown on Figure 9a) stems from the fact that moniloggers are triggered on average 3.34 times more in this scenario. In addition, a sharp decrease in overhead per call can also be observed as simulation time increases for the `dump-variable` and `visualize-stdev` scenarios, as it starts above 4ms (resp. above 12ms) and ends around 1–1.5ms (resp. above 4ms). We explain this by the fact that, while all scenarios are executed on GraalVM, those two scenarios make use of language calls to JavaScript, which are only optimized if the execution lasts long enough.

Overall, we consider the overhead induced by the approach to be suitable to debugging activities as the absolute overhead stays reasonably low on shorter execution times, and the relative overhead is reduced by around 50% as the execution time increases.

7.3 Threats to Validity

An external threat to validity is that the evaluation relies on a single DSL, NabLab. However, this DSL is (*i*) representative of the kind of languages for which interactive debugging does not work well, (*ii*) computationally intensive, and therefore a good candidate to evaluate the induced overhead of the approach, (*iii*) representative of the languages that benefit from being supplemented with MoniLog, as it does not provide the expressivity required for logging.

Another external threat to validity is that the evaluation only deals with simple temporal properties. However, while MoniLog does not directly support temporal properties requiring the use of automata (*e.g.,* complex temporal properties written in linear temporal logic), their handling can be delegated to dedicated libraries through language calls.

8 Related Work

To our knowledge, no current approach provides integrated runtime monitoring and logging facilities, even less so for executable DSLs. We thus hereby review the closest works to ours focusing on either runtime monitoring, or logging in the context of executable DSLs.

On the runtime monitoring front, a number of works provide runtime monitoring for specific DSLs or domains (*e.g.,* BPMN [8], web services [18], however we limit here our analysis to language- and domain-agnostic approaches. In our previous work [16], we provide a property language for runtime monitoring allowing domain experts to express temporal properties from domain concepts and intuitive temporal patterns. Conversely, in the present work, we aim to (*i*) complement the expressivity of existing DSLs with that of other languages in the context of runtime monitoring and logging, and (*ii*) show that logging and runtime monitoring benefit from each other on the other hand.

In [5], Ancona *et al.* detail the Runtime Monitoring Language (RML), a DSL dedicated to the definition of runtime monitors in a system- and language-agnostic way. Both RML and MoniLog rely on a kind of instrumentation interface. However, the one used by RML fully relies on the data transmitted through event occurrences. Thus, to allow users to refer to variables of the system, either the complete execution state must be included in each event occurrence, or the instrumentation of the system must be tailored to each runtime monitor, incorporating only the required variables in event occurrences. In addition, RML relies on its own expressivity to manipulate data received from events, preventing data processing with dedicated languages or libraries.

In [19], Mertz *et al.* introduce TigrisDSL, a language for specifying relevance criteria for execution events to be monitored. TigrisDSL is mostly complementary with MoniLog, as it provides another way to selected the subset of monitored events. In [25], Rabiser *et al.* present their development process of a DSL for the runtime monitoring of systems of systems. Similarly to RML, this DSL relies on its own expressivity for constraint evaluation, whereas MoniLog can delegate this to languages that are well-suited to the task and with which the user is proficient.

On the logging front, Morin and Ferry propose in [20] a platform-independent logging framework for ThingML, a DSL for modeling the behavior of distributed systems allowing to generate code for multiple languages such as Java, JavaScript, and Go. This framework provides a kind of instrumentation interface for ThingML, exposing ThingML functions, properties, and events. Compared to the logging aspect of our proposed approach, this framework is designed for systematic logging instrumentation, and thus does not enable conditional logging, nor does it allow users to tailor logging messages, or to log data derived from the execution state (*e.g.,* average, standard deviation).

9 Conclusion

We presented MoniLog, a new language to define runtime monitors, loggers, and combinations of the two to instrument models comforming to executable DSLs with an operational semantics. The language provides monitoring and logging capabilities to executable DSLs lacking them due to their restricted expressivity. In addition, as it allows to freely combine logging and monitoring constructs, MoniLog allows domain experts to leverage their complementarities.

Concerning future work, we plan to explore the use of MoniLog in the context of polyglot applications, to enable the definition of moniloggers across models conforming to different languages. Adapting MoniLog to executable DSLs with a translational semantics is also part of our lines for future work, in particular in the context of the C++ code generated from NabLab models.

Quantitative Evaluation Results

Table 1. Execution overhead induced by the approach measured over three scenarios.

Simulation time	0.2	0.4	0.6	0.8	1.0
Baseline (Truffle-Based / Pure Java)					
Execution time (s)	7.49 / 33.84	13.04 / 58.77	18.07 / 83.79	21.46 / 100.55	24.90 / 116.62
Dump Variables (Truffle / AspectJ)					
Execution time (s)	10.91 / 44.18	16.89 / 72.94	22.72 / 99.62	26.77 / 116.83	31.16 / 134.23
Absolute overhead (s)	3.42 / 10.34	3.86 / 14.16	4.65 / 15.83	5.31 / 16.28	6.26 / 17.60
Relative overhead	×1.46 / ×1.31	×1.30 / ×1.24	×1.26 / ×1.19	×1.25 / ×1.16	×1.25 / ×1.15
Number of calls	800	1600	2400	3200	4000
Monitor Residual (Truffle / AspectJ)					
Execution time (s)	9.52 / 43.24	14.94 / 71.28	20.93 / 98.60	24.75 / 117.32	28.37 / 132.66
Absolute overhead (s)	2.02 / 9.41	1.90 / 12.52	2.86 / 14.82	3.29 / 16.77	3.47 / 16.05
Relative overhead	×1.27 / ×1.28	×1.15 / ×1.21	×1.16 / ×1.18	×1.15 / ×1.17	×1.14 / ×1.14
Number of calls	3718	6118	8496	10096	11696
Visualize Standard Deviation (Truffle / AspectJ)					
Execution time (s)	10.67 / 41.03	16.32 / 70.62	21.85 / 98.86	25.99 / 118.20	29.64 / 137.13
Absolute overhead (s)	3.18 / 7.19	3.28 / 11.85	3.78 / 15.07	4.53 / 17.65	4.74 / 20.51
Relative overhead	×1.42 / ×1.21	×1.25 / ×1.20	×1.21 / ×1.18	×1.21 / ×1.18	×1.19 / ×1.18
Number of calls	800	1600	2400	3200	4000
Summary (Truffle / AspectJ)					
Absolute overhead (s)	2.80 / 8.88	2.89 / 12.81	3.69 / 15.24	4.29 / 16.89	4.69 / 17.96
Relative overhead	×1.37 / ×1.26	×1.22 / ×1.22	×1.20 / ×1.18	×1.20 / ×1.17	×1.19 / ×1.15

References

[1] Florian-Michael Adolf, Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Christoph Torens. 2017. Stream runtime monitoring on UAS. In *International Conference on Runtime Verification*. Springer, 33–49. https://doi.org/10.1007/978-3-319-67531-2_3

[2] James Ahrens, Berk Geveci, and Charles Law. 2005. Paraview: An end-user tool for large data visualization. *The visualization handbook* 717, 8 (2005).

[3] E Emanuel Almeida, Jonathan E Luntz, and Dawn M Tilbury. 2007. Event-condition-action systems for reconfigurable logic control. *IEEE Transactions on Automation Science and Engineering* 4, 2 (2007), 167–181. https://doi.org/10.1109/TASE.2006.880857

[4] Sepehr Amir-Mohammadian, Stephen Chong, and Christian Skalka. 2016. Correct audit logging: Theory and practice. In *International Conference on Principles of Security and Trust*. Springer, 139–162. https://doi.org/10.1007/978-3-662-49635-0_8

[5] Davide Ancona, Luca Franceschini, Angelo Ferrando, and Viviana Mascardi. 2021. RML: Theory and practice of a domain specific language for runtime verification. *Science of Computer Programming* 205 (2021), 102610. https://doi.org/10.1016/j.scico.2021.102610

[6] André Arnold. 1994. *Finite transition systems: semantics of communicating systems*. Prentice Hall International (UK) Ltd.

[7] James Bailey, George Papamarkos, Alexandra Poulovassilis, and Peter T Wood. 2004. An event-condition-action language for XML. In *Web Dynamics*. Springer, 223–248. https://doi.org/10.1007/978-3-662-10874-1_10

[8] Ahmed Barnawi, Ahmed Awad, Amal Elgammal, Radwa El Shawi, Abduallah Almalaise, and Sherif Sakr. 2015. BP-MaaS: A Runtime Compliance-Monitoring System for Business Processes. In *BPM (Demos)*. 25–29.

[9] Guillaume Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Arnaud Venet, Willem Visser, and Rich Washington. 2004. Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods in System Design* 25, 2 (2004), 167–198. https://doi.org/10.1023/B:FORM.0000040027.28662.a4

[10] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. 2017. A survey of runtime monitoring instrumentation techniques. *arXiv preprint arXiv:1708.07229* (2017). https://doi.org/10.4204/EPTCS.254.2

[11] Boyuan Chen and Zhen Ming Jiang. 2021. A Survey of Software Log Instrumentation. *ACM Computing Surveys (CSUR)* 54, 4 (2021), 1–34. https://doi.org/10.1145/3448976

[12] Curtis Clifton, Gary T Leavens, Craig Chambers, and Todd Millstein. 2000. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *ACM Sigplan Notices*, Vol. 35. ACM, 130–145. https://doi.org/10.1145/353171.353181

[13] Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. 2017. A foundation for runtime monitoring. In *International Conference on Runtime Verification*. Springer, 8–29.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. 2001. An overview of AspectJ. In *European Conference on Object-Oriented Programming*. Springer, 327–354. https://doi.org/10.1007/3-540-45337-7_18

[15] Benoît Lelandais, Marie-Pierre Oudot, and Benoit Combemale. 2018. Fostering metamodels and grammars within a dedicated environment for HPC: the NabLab environment (tool demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*. 200–204. https://doi.org/10.1145/3276604.3276620

[16] Dorian Leroy, Pierre Jeanjean, Erwan Bousse, Manuel Wimmer, and Benoit Combemale. 2020. Runtime Monitoring for Executable DSLs. *The Journal of Object Technology* 19, 2 (2020), 1–23. https://doi.org/10.5381/jot.2020.19.2.a6

[17] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming* 78, 5 (2009), 293–303. https://doi.org/10.1016/j.jlap.2008.08.004

[18] Zheng Li, Yan Jin, and Jun Han. 2006. A runtime monitoring and validation framework for web service interactions. In *Australian Software Engineering Conference (ASWEC'06)*. IEEE, 10–pp. https://doi.org/10.1109/ASWEC.2006.6

[19] Jhonny Mertz and Ingrid Nunes. 2021. Tigris: A DSL and framework for monitoring software systems at runtime. *Journal of Systems and Software* 177 (2021), 110963. https://doi.org/10.1016/j.jss.2021.110963

[20] Brice Morin and Nicolas Ferry. 2019. Model-based, platform-independent logging for heterogeneous targets. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 172–182. https://doi.org/10.1109/MODELS.2019.000-4

[21] Object Management Group. 2013. OMG Unified Modeling Language (OMG UML), V 2.5. http://www.omg.org/spec/UML/2.5.

[22] Object Management Group. 2016. Meta Object Facility (MOF) Core Specification, V 2.5. http://www.omg.org/spec/MOF/2.5.

[23] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. 2015. Industry practices and event logging: Assessment of a critical software development process. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 169–178. https://doi.org/10.1109/ICSE.2015.145

[24] Thinagaran Perumal, Md Nasir Sulaiman, and Chui Yew Leong. 2013. ECA-based interoperability framework for intelligent building. *Automation in Construction* 31 (2013), 274–280. https://doi.org/10.1016/j.autcon.2012.12.009

[25] Rick Rabiser, Jürgen Thanhofer-Pilisch, Michael Vierhauser, Paul Grünbacher, and Alexander Egyed. 2018. Developing and evolving a DSL-based approach for runtime monitoring of systems of systems. *Automated Software Engineering* 25, 4 (2018), 875–915. https://doi.org/10.1007/11580850

[26] Filippo Schiavio, Haiyang Sun, Daniele Bonetta, Andrea Rosà, and Walter Binder. 2019. Nodemop: Runtime verification for node. js applications. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 1794–1801. https://doi.org/10.1145/3297280.3297456

[27] Roy Shea, Y Cho, and M Srivastava. 2009. Lis is more: Improved diagnostic logging in sensor networks with log instrumentation specifications. *Univ. California, Los Angeles, CA, USA, Tech. Rep. TR-UCLA-NESL-200906-01* (2009).

[28] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional.

[29] Michael Van De Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. 2018. Fast, flexible, polyglot instrumentation support for debuggers and other tools. *arXiv preprint arXiv:1803.10201* (2018). https://doi.org/10.22152/programming-journal.org/2018/2/14

[30] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 187–204. https://doi.org/10.1145/2509578.2509581

[31] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 102–112. https://doi.org/10.1109/ICSE.2012.6227202

[32] Michael Zoumboulakis, George Roussos, and Alexandra Poulovassilis. 2004. Active rules for sensor databases. In *Proceeedings of the 1st international workshop on Data management for sensor networks: in conjunction with VLDB 2004*. 98–103. https://doi.org/10.1145/1052199.1052215