Practical Runtime Instrumentation of Software Languages: The Case of SciHook

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

Practical Runtime Instrumentation of Software Languages: The Case of SciHook

Dorian Leroy
CEA, DAM, DIF
F-91297, Arpajon, France
Université Paris-Saclay, CEA DAM DIF, LIHPC
91297, Arpajon, France
dorian.leroy@cea.fr

Benoit Combemale
University of Rennes
Rennes, France
benoit.combemale@irisa.fr

Benoît Lelandais
CEA, DAM, DIF
F-91297, Arpajon, France
Université Paris-Saclay, CEA DAM DIF, LIHPC
91297, Arpajon, France
benoit.lelandais@cea.fr

Marie-Pierre Oudot
CEA, DAM, DIF
F-91297, Arpajon, France
Université Paris-Saclay, CEA DAM DIF, LIHPC
91297, Arpajon, France
marie-pierre.oudot@cea.fr

Abstract

Software languages have pros and cons, and are usually chosen accordingly. In this context, it is common to involve different languages in the development of complex systems, each one specifically tailored for a given concern. However, these languages create *de facto* silos, and offer little support for interoperability with other languages, be it statically or at runtime. In this paper, we report on our experiment on extracting a relevant behavioral interface from an existing language, and using it to enable interoperability at runtime. In particular, we present a systematic approach to define the behavioral interface and we discuss the expertise required to define it. We illustrate our work on the case study of SciHook, a C++ library enabling the runtime instrumentation of scientific software in Python. We present how the proposed approach, combined with SciHook, enables interoperability between Python and a domain-specific language dedicated to numerical analysis, namely NabLab, and discuss overhead at runtime.

CCS Concepts: • **Software and its engineering** → **Source code generation**; **General programming languages**; **Domain specific languages**; • **Applied computing**;

Keywords: language interoperability, instrumentation, C++, Python, domain-specific languages, scientific computing

1 Introduction

Software languages are tools providing specific abstractions to support developers in describing efficient solutions (i.e., software systems) to their problems. According to the abstractions provided, a given software language is more or less relevant for a specific concern. For instance, in the field of scientific computing, it is common to use C or C++ for implementing efficient simulation models, complemented by Python to help in data processing or debug instrumentation. This leads to polyglot development of software systems [1].

In this polyglot development context, existing approaches are either focusing on the use of specific libraries unifying on a single language runtime (*e.g.,* Truffle/GraalVM [3, 10], LLVM [5], WebAssembly [4]), or with ad-hoc bindings between different language runtimes defined at the program level (*e.g.,* CORBA [8], CCA [1], CoLoRS [13]). While the former limits to the use of a specific execution platform, the latter requires to define bindings at the program level. This either prevents the use of specific language runtimes (*e.g.,* specific C++ compilers such as GCC, or mainstream Python interpreters like CPython or Pypy), or imposes the overhead of defining *ad-hoc* bindings for each new program.

In this paper, we introduce an approach to support interoperability between different language runtimes (interpreters and compilers) through specific interfaces defined at the language level. We present our approach to define such language interfaces for NabLab[1], a Domain-Specific Language (DSL) for scientific computing, and report on our experience using it with SciHook[2], our C++ library to enable Python-based runtime instrumentation for scientific software.

Using SciHook, computational scientists can write analyses as Python scripts that will run during the simulation, also called in-situ analyses [12], with the full power of Python's libraries for scientific computing [9]. Such analyses can access the data they require in-memory, which allows to turn off expensive input/ouput operations (I/Os) and speeds up the simulation time significantly [2]. Furthermore, with write access to the execution context of the simulation, instruments can implement complex behaviors varying on a case-by-case basis such as the physical behavior of the environment of a simulation, without needing to recompile the simulator.

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

We experimented the use of SciHook with simulation models implemented in NabLab [6], a compiled DSL for numerical analysis. Based on these experiments, we discuss the required effort for the definition of the behavioral interfaces, the resulting performance at runtime, and the suitability of runtime instrumentation coupled with software language interoperability.

We demonstrate the practicality of enabling interoperability between different languages through well-defined and purposefully designed language behavioral interfaces, with a limited overhead. We also show how the combination of interoperability and runtime instrumentation capability opens up new usage scenarios for the instrumented programs.

The remainder of this paper is as follows. Section 2 presents the motivation behind this work. Section 3 details the proposed approach. Section 4 discusses how we applied the approach to NabLab, using SciHook to enable the instrumentation of NabLab programs in Python. Section 5 presents our evaluation of the overhead induced by the approach over a selection of use cases. Section 6 discusses related works, and Section 7 provides concluding remarks.

2 Motivation

In this paper, we use the field of scientific computing and the software languages used in that field as our illustrative example. The two main use cases for language interoperability in scientific computing are (*i*) C++ and Fortran, to call legacy Fortran code from C++ code [1], and (*ii*) Python and C/C++, to pilot efficient C/C++ libraries from Python-based GUIs or with libraries such as SciPy [1, 11]. In both use cases, interoperability is mainly used to provide program-level bindings for black box software components, and does not allow to interact with the internal state of such components.

Yet, specific operations might be more easily or robustly implemented in a different software language, or require scriptability to access the execution state of the component at runtime. For example, in the context of complex simulation codes, this allows to submit scripts to process data *in-situ* [2], waiving the need to persist complete data sets to disk. Another use case is to expose the execution state at a mathematical level of abstraction, in a language well-suited for mathematical operations (*e.g.,* Python with NumPy support). This in turn enables debugging and domain-specific property monitoring at an adequate level of abstraction for numerical analysts, while keeping the computation-intensive parts of the simulator efficient.

Unfortunately, support for such a gray-box usage of software language interoperability is tedious and error-prone to implement, and does not contribute directly to the business logic of the application. This can be dissuasive for practitioners of scientific computing, who are not software engineers and/or might not have the manpower to spare on these concerns. To remedy this, we present a systematic approach to define and realize a *behavioral interface* dedicated to instrumentation for existing software languages. We then illustrate the approach through a case study using SciHook, a C++ library to enable Python-based runtime instrumentation, and provide performance measurements for a range of relevant use cases for scientific computing.

3 Systematic Runtime Instrumentation of Software Languages

We define runtime instrumentation as the dynamic introduction of code at specific points in the execution of a program, and with controlled access to a subset of the execution state of the program. We detail below our proposed approach for enabling this at the language level, which relies on the systematic definition of *behavioral interfaces* for software languages, that are then realized through an *instrumentation runtime* supporting software language interoperability.

3.1 Defining the Behavioral Interface

In this paper, we adopt a definition for language behavioral interfaces similar to the definition given in [7], albeit more restricted. Indeed the behavioral interfaces defined with our approach consist of the set of language-level events that are exposed by any program written with that language, and that provide their execution context as a parameter. The process for defining such behavioral interfaces is as follows.

Identify execution events. The first step consists in identifying the set of abstract syntax tree (AST) nodes whose execution will result in the emission of an execution event, and crafting a static analysis to extract this set of AST nodes from the AST of a program. At runtime, the set of exposed execution events can then be queried by instruments, allowing them to register to and unregister from these events, and thus be triggered by their emission.

Determine execution contexts. The second step consists in providing the means for instruments to operate on the current execution state when they are triggered by the emission of an execution event. To this end, instruments must be provided an *execution context* when triggered.

However, depending on the purpose for which the behavioral interface is defined, it might not be desirable to expose the complete internal state of a program to its instruments. For example, in the context of execution events emitted on a method call, a behavioral interface designed for regular users might only expose the public fields and methods of the containing object, whereas one designed for developers or expert users might also expose private fields and methods.

Thus, extracting the proper execution context of each execution event of the behavioral interface requires a static analysis tailored to the purpose of the interface.

3.2 Specification of the Instrumentation Runtime

In the remainder of this section, we differentiate the *host language* from the *instrumentation language*. In the context

of a given program, the host language is the language used to write the program being instrumented, which we refer to as the *host program*. Instrumentation languages are the languages used to write the instrumentation code (*i.e.,* the instruments) for the host program. We describe below the API that the instrumentation runtime must provide.

Event declaration. The runtime must provide a way for the host program to declare events to which instruments can subscribe. This allows to specify which parts of the application can be instrumented. Note that events can be declared in different granularities, to open more or less parts of the application to instrumentation, similarly to logging levels.

Event subscription. Conversely, the instrumentation runtime must provide a way for the instruments to query, subscribe to, and unsubscribe from execution events. This allows to write instruments that are able to dynamically activate and deactivate themselves, and to identify specific subsets of the exposed execution events to which register.

Event emission. Finally, the runtime must offer a way for the host program to emit execution events, thereby executing the instruments registered to these events, passing along the corresponding execution context. That way, the instrumentation runtime acts as a bridge between host language runtime and instrumentation language runtime.

3.3 Realizing the Interface

Host languages must then provide facilities as part of their infrastructure to derive the instrumentation interface of any program, and realize it through the API of the instrumentation runtime, exposing (*i*) the different execution events to which instruments can register, and (*ii*) a wrapper exposing the associated execution contexts to instruments.

Depending on the host language and on the software language used to implement the instrumentation runtime, using the API of the instrumentation runtime might require the use of foreign function interface or similar technologies.

Finally, interoperability bindings must be defined over the exposed execution contexts so they can be accessed from the desired instrumentation languages. These can be defined systematically for each execution context once bindings for the basic types manipulated by the host language are defined.

4 The Case of SciHook

In this section, we first provide an overview of SciHook, our C++/Python instrumentation runtime. We then discuss the work required to apply the proposed approach to NabLab, a DSL with compiler back-ends targeting C++, using SciHook as the instrumentation runtime.

4.1 SciHook Overview

Figure 1 provides an overview of SciHook. On the left is a piece of scientific software, which can be implemented directly in C++, or generated from its specification when written in a language transpiling to C++ (such as NabLab).

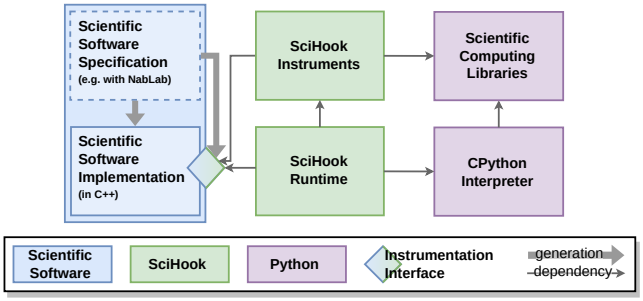

Figure 1. Overview of SciHook.

To leverage SciHook, this piece of scientific software provides an instrumentation interface, as defined in the previous Section. This instrumentation interface can be generated directly for a given program (in C++), or it can be generated from the software specification (using NabLab).

At the center of Figure 1 are the SciHook runtime and its registered instruments. Using the SciHook API, applications define their execution events as specified in their instrumentation interface, and trigger those events during the execution. Conversely, SciHook instruments use the SciHook API to register and unregister to the runtime, listing their triggering events from the instrumentation interface.

The SciHook runtime stores the registered events and instruments, and triggers instruments upon the emission of events to which they are registered. Triggered SciHook instruments are provided with the execution context of the application, on which they can perform read and possibly write operations, depending on how the context was exposed, as well as call functions exposed as part of the context.

To achieve this, the SciHook runtime depends on the CPython interpreter, as shown on the right of Figure 1, to which it delegates the execution of SciHook instruments. This means that, when writing instruments, SciHook users have access to the vast ecosystem of libraries for scientific computing such as NumPy, Matplotlib, Numba, and so on [9].

With access to scientific computing libraries and to the execution state of the application, users can craft analyses that are easily plugged into the application and that can be turned on or off at runtime. In addition, the separate "instrumentation state" (*i.e.,* the heap of the CPython interpreter) enables unanticipated, execution-wide analyses (*e.g.,* monitoring temporal properties), without relying on I/Os.

Beyond analysis and debugging, write access to the execution context also allows to configure simulation workflows with Python scripts, from input and output data processing, to simulation initialization, to system behavior specification.

4.2 Experimentation with NabLab

NabLab is a DSL for scientific computing allowing numerical analysts to define their numerical schemes at a level of abstraction close to discrete mathematics, and then generate the corresponding C++ simulator. The C++ code generation

infrastructure handles system-level concerns such as memory and programming paradigm (GPU, CPU, MPI, etc.). In this experiment, it is extended to derive the instrumentation interface from the NabLab program, and realize it.

Identifying Execution Events. The execution events exposing the instrumentation points of a NabLab program include calls to the jobs defined in a NabLab program (*i.e.,* its callable entities), as well as all variables writes. Job call events are emitted before and after the triggering calls, and write events are emitted before and after the triggering writes.

In the case of variable writes, we define two kinds of events: global writes and local writes. Global write events are emitted when writing to any global variables. Local write events are emitted when writing to a variable (global or local) in the context of a specific Job. Thus, when a global variable is written to, two write events are emitted before and after the write: a global one and a local one.

As a design decision we made when applying the approach to NabLab, when a variable is written to inside a loop but declared outside of that loop, the corresponding write events are only emitted before/after the entire execution of the loop. This allows to only be notified before and after the complete update of arrays or accumulator variables.

Exposing the Execution Context. With execution events identified, the corresponding execution contexts must be computed to be exposed as part of the instrumentation interface. In the case of job call events, only the global variables and parameters of the call are exposed in the execution context. In the case of global write events, only global variables are considered in the execution context, whereas in the case of local write events, local variables are included as well, as are the parameters provided to the encompassing job.

Realizing the Interface. We realize the instrumentation interface during the C++ code generation.

First, we insert calls to the SciHook runtime to declare the execution events exposed by the interface. Next, we generate each distinct execution context as a C++ struct holding references to the variables accessible therefrom. We then generate code instantiating these structs at each new execution context, and calls to the SciHook runtime triggering execution events with their execution context.

Finally, we generate Python bindings for these structs, exposing the variables they encapsulate to registered instruments. We also generate a Python-facing interface to expose these execution events to Python-based instruments.

Library for Code Generation. To apply the approach, we developed an Xtend library of around 900 lines of code for analyzing NabLab programs and generating C++ instrumentation code, which we added to the code generation infrastructure of NabLab, also written in Xtend. This library provides facilities for computing the set of execution events of a NabLab program, the set of corresponding execution contexts, and the set of concrete types that must be exposed to Python. To mesh well with scientific computing Python libraries, we exposed the array types of NabLab as NumPy arrays, thereby avoiding expensive copy operations.

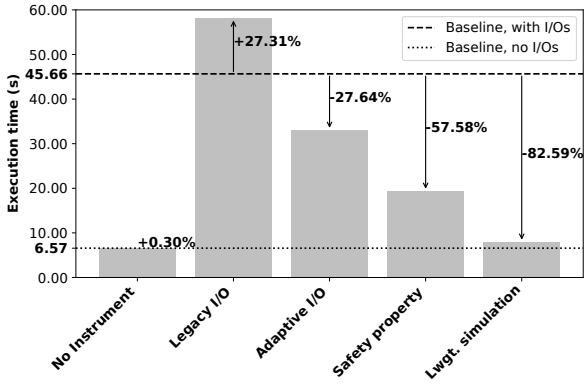

Figure 2. Performance measurements of NabLab model instrumented with SciHook in various use cases.

The code generation library also provides facilities to generate the Python bindings for the execution contexts, and the CMake build files integrating SciHook into the application. This code generation library is disabled by default and, when enabled, places all instrumentation code between `#ifdef`/`#ifndef` directives. As a result, C++ code is generated without instrumentation by default, and when generated, the instrumentation must be turned on at compile-time, allowing the instrumented code to be used in production.

5 Performance Evaluation

In this Section we evaluate and discuss the overhead induced by SciHook-based instrumentation of NabLab in various use cases. For each use case, we measured the execution time of 30 runs of the same simulation, and provide the average execution time in Figure 2, as well as its relative overhead with regard to the baseline execution time. We performed the measurements on a 11th Gen Intel® Core™ i5-1145G7 @ 2.60GHz × 8, on Ubuntu 20.04.4.

Baseline. For this evaluation, we consider two baselines (dashed lines in Figure 2): the average execution time of the non-instrumented code with, and without I/Os. As we use SciHook instruments instead of post-processing to address the use cases, we disable the built-in I/Os. However, we still compare execution times against the "with I/Os" baseline, as they are mandatory for addressing the use cases with non-instrumented code, through post-processing. Note that we do not consider the overhead of this post-processing in our comparison.

No instrument. This use case shows the overhead induced by the instrumentation alone, without any instrument registered. For the evaluated model, the overhead with regard to the "no I/O" baseline is minimal as it stands at 0.30%.

Legacy I/O. This use case reproduces the original I/Os by calling the original C++ code responsible for I/Os, but through Python bindings, via a SciHook instrument. We

estimate that the 27.31% induced overhead is due to the back-and-forth between the Python interpreter and the simulator, and to the absence of link-time optimization. This shows that SciHook is not best used to naively reproduce core functionalities of a C++ simulator such as I/Os. However, the dynamic nature of SciHook allows developers to write adaptive instruments, as discussed next.

Adaptive I/O. This use case leverages the separate Python interpreter state to adapt the frequency of I/Os at runtime, dividing their frequency by 10 once the maximum temperature over the simulation domain goes below 75% percent of its starting value. In our case, this happens after 733 iterations out of 1628, with 895 iterations remaining, yielding 27.64% shorter execution times. However, this requires to be able to determine the "points of interest" of a simulation.

Safety property. In this use case, we monitor a safety property ensuring that the difference between a computed quantity of interest (temperature in this case) and its reference value never exceeds a given threshold across the simulated domain. The 57.58% shorter execution time corresponds to runs where the property is never violated, and is thus monitored during the entire execution.

Lightweight simulation. This use case exemplifies the use of scientific computing for exploratory purposes, where simulations are run in a fast and iterative process. All input data are provided through SciHook instruments, and a plot of the final state of the simulation is the only produced output, meaning no time is spent on I/Os. The 82.59% shorter execution time is an important speed-up compared to non-instrumented code, which allows to quickly obtain insights on a simulated physics problem.

These results demonstrate the practicality of enabling runtime instrumentation at the language level, and instrumenting scientific software to perform analyses during the execution, reducing the need for highly sequential workflows relying on writing data to disk and reading it back in another tool. In particular, the use case of lightweight simulation greatly benefits from this, as the approach allows to prototype simulations quickly. In the case of simulations where the output is kept as a reference and analyzed multiple times by a variety of tools, the I/O-intensive approach remains best, as long as the computing infrastructure is able to handle the amount of data produced by the simulation.

6 Related Work

We identified two categories of related works in the context of enabling software language interoperability.

The first category regroups approaches providing interoperability inside a single language runtime, through a unified intermediate representation used by all supported languages. This is the case of Truffle/GraalVM [14], LLVM [5], or WebAssembly [4]. This means that interoperability with specific language runtimes such as Pypy, CPython, GCC, or the Intel compiler must be implemented at the program level. In comparison, our proposed approach aims to support interoperability between language runtimes.

The second category regroups approaches like CORBA [8] and CCA [1], which relies on interfaces defined for each component to allow them to communicate, whether they are defined in the same language or not. However, as interfaces are defined at the component level, each new component necessitates the definition of its interface, and its implementation by the component. In our proposed approach, the interface is instead defined at the language level, and realized at compile-time, and can thus be reused for each new component defined with that language.

7 Concluding Remarks and Perspectives

In this paper, we presented an approach to support interoperability between different software languages, relying on the definition of language behavioral interfaces, and on the use of an instrumentation runtime to realize those interfaces. We demonstrated the approach on the NabLab DSL, using SciHook as our instrumentation runtime to realize the interface and provide interoperability between C++ (the target language of NabLab), and Python. The ability to instrument scientific software in Python allowed for greater agility when prototyping and debugging, illustrating the benefits of opening language-induced silos to other languages.

From here, we envision several threads of future work. A first perspective is to explore the interplay between the instrumentation interface of a language and testing frameworks, with the goal to provide testing support out-of-the-box to languages exposing an instrumentation interface. Another perspective is to explore solutions based on just-in-time compilation to reduce the time spent in the Python interpreter, to circumvent the pitfalls of Python such as its global interpreter lock, minimizing the crossing of language boundaries, and optimizing Python code implementing complex behaviors and/or acting as glue between native libraries.

References

[1] David E Bernholdt, Benjamin A Allan, Robert Armstrong, Felipe Bertrand, Kenneth Chiu, Tamara L Dahlgren, Kostadin Damevski, Wael R Elwasif, Thomas GW Epperly, Madhusudhan Govindaraju, et al. 2006. A component architecture for high-performance scientific computing. *The International Journal of High Performance Computing Applications* 20, 2 (2006), 163–202.

[2] Hank Childs, Janine Bennett, Christoph Garth, and Bernd Hentschel. 2019. In Situ Visualization for Computational Science. *IEEE Computer Graphics and Applications* 39, 6 (2019), 76–85.

[3] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, Mikel Luján, and Hanspeter Mössenböck. 2018. Cross-Language Interoperability in a Multi-Language Runtime. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 40, 2 (2018), 1–43.

[4] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of*

the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. 185–200.

[5] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[6] Benoit 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.

[7] Dorian Leroy, Erwan Bousse, Manuel Wimmer, Tanja Mayerhofer, Benoit Combemale, and Wieland Schwinger. 2020. Behavioral Interfaces for Executable DSLs. *Software and Systems Modeling* 19 (2020), 1015–1043.

[8] Object Management Group. [n. d.]. Common Object Request Broker Architecture. https://www.omg.org/spec/CORBA.

[9] Travis E Oliphant. 2007. Python for Scientific Computing. *Computing in science & engineering* 9, 3 (2007), 10–20.

[10] Michael Van De Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. 2018. Fast, Flexible, Polyglot Instrumentation Support for Debuggers and Other Tools. *The Art, Science, and Engineering of Programming* 2, 3 (2018), 14–1.

[11] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature methods* 17, 3 (2020), 261–272.

[12] Venkatram Vishwanath, Mark Hereld, and Michael E Papka. 2011. Toward Simulation-Time Data Analysis and I/O Acceleration on Leadership-Class Systems. In *2011 IEEE Symposium on Large Data Analysis and Visualization*. IEEE, 9–14.

[13] Michal Wegiel and Chandra Krintz. 2010. Cross-Language, Type-Safe, and Transparent Object Sharing for Co-Located Managed Runtimes. *ACM Sigplan Notices* 45, 10 (2010), 223–240.

[14] 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.

Received 2023-07-07; accepted 2023-09-01